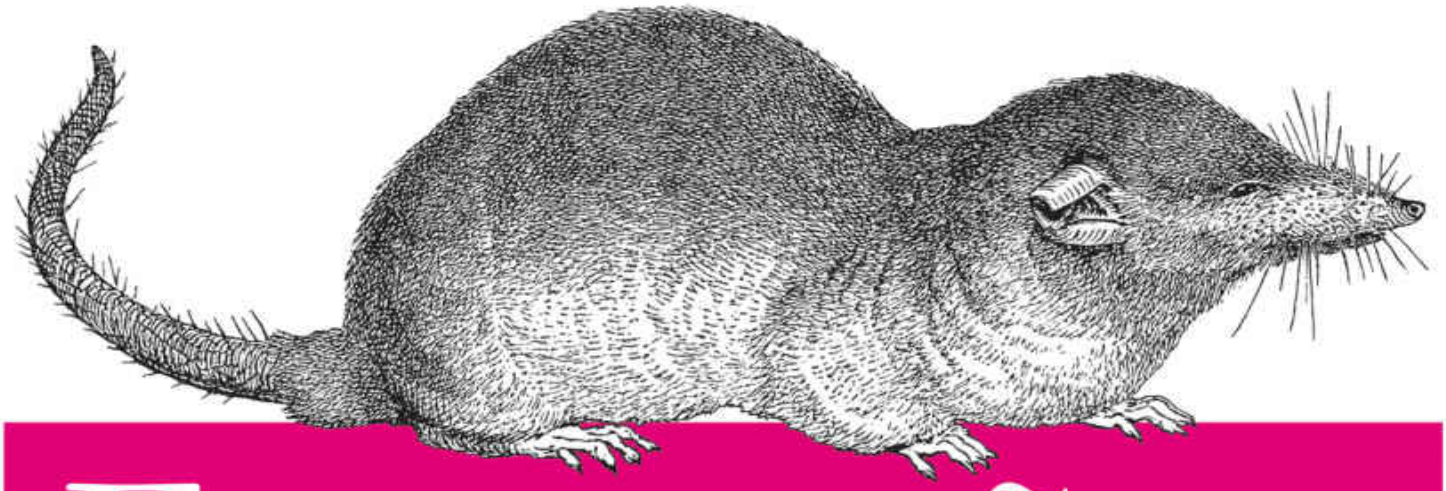
Soluções detalhadas em oito linguagens de programação

Inclui um tutorial sobre expressões regulares



Expressões Regulares Cookbook

O'REILLY®
novatec

Jan Goyvaerts
& Steven Levithan

Expressões Regulares Cookbook

Jan Goyvaerts e Steven Levithan

O'REILLY
Novatec
São Paulo | 2019

Tradução em português autorizada da edição em inglês da obra Regular Expressions Cookbook ISBN 9780596520687 © 2009, Jan Goyvaerts e Steve Levithan. Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., detentora de todos os direitos para publicação e venda desta obra.

Editor: Rubens Prates
Tradução: Rafael Contatori e Edgard Damiani
Revisão gramatical: Jeferson Ferreira
Editoração eletrônica: Camila Kuwabata e Carolina Kuwabata

ISBN: 978-85-7522-795-4

Histórico de edições impressas:

Abril/2011 Primeira edição

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
E-mail: novatec@novatec.com.br
Site: www.novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec

Sumário

Prefácio

Na última década, as expressões regulares experimentaram um incrível aumento de popularidade. Hoje em dia, todas as linguagens de programação populares incluem uma poderosa biblioteca de expressões regulares, ou mesmo um suporte a expressões regulares embutido na própria linguagem. Muitos desenvolvedores se aproveitaram das características das expressões regulares para fornecer a seus usuários a capacidade de pesquisar ou filtrar seus dados utilizando uma expressão regular. Expressões regulares estão em toda a parte.

Muitos livros foram publicados com o intuito de surfar na onda da adoção da expressão regular. A maioria faz um bom trabalho na explicação da sintaxe, juntamente com alguns exemplos e referências. Mas não existem muitos livros que apresentem soluções para uma ampla gama de problemas práticos do mundo real, ao lidar com textos em um computador e em uma variedade de aplicações da Internet baseadas em expressões regulares. Nós, Steve e Jan, decidimos preencher essa necessidade com este livro.

Queríamos, particularmente, mostrar como você pode utilizar expressões regulares em situações nas quais pessoas com experiência limitada diriam que algo não pode ser feito, ou em que os puristas da programação diriam que uma expressão regular não é a ferramenta certa para o trabalho. Como as expressões regulares estão em toda parte hoje em dia, elas se configuram, muitas vezes, como ferramentas facilmente disponíveis que podem ser utilizadas por usuários finais sem a necessidade de envolver uma equipe de programadores. Mesmo os programadores podem economizar tempo utilizando algumas expressões regulares para a recuperação de informações e alterações de tarefas que levariam horas, ou dias,

para o processamento, ou que de outra maneira iriam requerer uma biblioteca de terceiros, que necessitaria de uma análise prévia e aprovação da gerência.

Pego no emaranhado das diferentes versões

Como ocorre com qualquer coisa que se torna popular na indústria de TI, as expressões regulares vêm em muitas implementações diferentes, com diferentes graus de compatibilidade. Isso resultou em muitos *sabores* diferentes de expressões regulares, que nem sempre agem ou trabalham da mesma maneira em uma expressão regular específica.

Muitos livros mencionam que existem diferentes sabores e apontam algumas das diferenças. Mas, muitas vezes, eles deixam de fora determinados sabores, aqui e ali; particularmente quando um dado sabor não tem certas características, ao invés de oferecer soluções alternativas ou provisórias. É frustrante quando você tem que trabalhar com expressões regulares de diferentes sabores, em diferentes aplicações ou linguagens de programação.

Declarações casuais na literatura, como “todo mundo utiliza expressões regulares no estilo Perl, agora”, infelizmente banalizam um vasto leque de incompatibilidades. Até mesmo os pacotes “Estilo-Perl” têm diferenças importantes e, entretanto, o Perl continua a evoluir. Impressões simplificadas em demasia podem levar os programadores a desperdiçar seu tempo inutilmente executando o depurador, ao invés de verificar os detalhes de sua implementação de expressões regulares. Mesmo quando descobrem que alguma característica da qual dependiam não está presente, nem sempre sabem como contornar o problema.

Este livro é o primeiro no mercado que discute os sabores mais populares das expressões regulares, além de suas características, lado a lado; e o faz de forma consistente ao longo do livro.

Público-alvo

Se você trabalha regularmente com texto em um computador, deve ler esse livro, seja na pesquisa de uma pilha de documentos, na manipulação de texto em um editor de texto, ou no desenvolvimento de programas que necessitem pesquisar ou manipular textos. As Expressões regulares são uma excelente ferramenta para esse tipo de trabalho. O livro *Expressões Regulares Cookbook* ensina tudo que você precisa saber sobre expressões regulares. Você não precisa de nenhuma experiência prévia, porque nós explicamos até os aspectos mais básicos das expressões regulares.

Se você já tem experiência com expressões regulares, encontrará uma grande riqueza de detalhes que outros livros e artigos on-line muitas vezes encobrem. Se você já foi confundido por uma expressão regular (regex) que funciona em um aplicativo, mas não em outro, aqui você encontrará uma cobertura detalhada de sete dos mais populares e valiosos sabores do mundo das expressões regulares. Nós organizamos esse livro como um livro de receitas, para que você possa pular diretamente para os tópicos que desejar ler. Se você ler o livro de capa a capa, vai se tornar um “chef” das expressões regulares.

Este livro ensina tudo o que você precisa saber sobre expressões regulares, independentemente de ser, ou não, um programador. Se quiser usar expressões regulares com um editor de texto, com uma ferramenta de busca ou com qualquer aplicação que tenha uma caixa de entrada com o rótulo “expressões regulares”, pode ler este livro sem nenhuma experiência prévia em programação. A maioria das receitas neste livro tem soluções puramente baseadas em uma ou mais expressões regulares.

Se você é programador, o capítulo 3 fornece todas as informações de que precisa para implementar as expressões regulares em seu código-fonte. Este capítulo assume que você está familiarizado com os recursos básicos da linguagem de programação de sua escolha, mas não assume que você já tenha utilizado uma expressão regular

em seu código-fonte.

Tecnologia abrangida

.NET, Java, JavaScript, PCRE, Perl, Python, e Ruby não são apenas termos de contracapa. Estes são os sete sabores de expressões regulares cobertos por este livro. Cobriremos todos os sete sabores igualmente. Tomamos a precaução de apontar todas as inconsistências que pudemos encontrar entre esses sabores de expressões regulares.

O capítulo de programação (Capítulo 3) tem listas de códigos em C#, Java, JavaScript, PHP, Perl, Python, Ruby e VB.NET. Novamente, cada receita tem soluções e explicações para todas as oito linguagens. Enquanto isso torna o capítulo um pouco repetitivo, você pode facilmente ignorar as discussões sobre as linguagens nas quais não está interessado, sem perder nada que deva saber sobre a linguagem de sua escolha.

Organização deste livro

Os três primeiros capítulos deste livro abrangem ferramentas úteis e informações básicas que lhe darão base para o uso de expressões regulares; cada um dos capítulos subsequentes apresenta uma variedade de expressões regulares ao investigar uma área de processamento de texto em profundidade.

Capítulo 1, *Introdução às Expressões Regulares*, explica o papel das expressões regulares e introduz uma série de ferramentas que facilitarão seu aprendizado, criação e depuração.

Capítulo 2, *Habilidades básicas de Expressão Regular*, cobre cada elemento e característica das expressões regulares, junto com importantes diretrizes para seu uso efetivo.

Capítulo 3, *Programando com Expressões Regulares*, especifica técnicas de codificação e inclui listas de código para o uso de expressões regulares em cada linguagem de programação coberta

por este livro.

Capítulo 4, *Validação e formatação*, contém receitas para manipular entradas típicas do usuário, tais como datas, números de telefone e códigos postais de vários países.

Capítulo 5, *Palavras, linhas e caracteres especiais*, explora tarefas comuns de processamento de texto, como verificação de linhas que contenham, ou não, certas palavras.

Capítulo 6, *Números*, mostra como detectar inteiros, números de ponto-flutuante e muitos outros formatos para esse tipo de entrada.

Capítulo 7, *URLs, caminhos e endereços de Internet*, mostra como desmontar e manipular as strings comumente usadas na Internet e nos sistemas Windows, para que possamos encontrar coisas.

Capítulo 8, *Marcação e intercâmbio de dados*, abrange a manipulação de HTML, XML, valores separados por vírgulas (CSV), e arquivos de configuração de estilo INI.

Convenções usadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico
: Indica novos termos, URLs, endereços de e-mail, nomes de arquivos e extensões de arquivo.

Largura constante
: Usado para listagens de programas, elementos do programa, tais como variáveis ou nomes de funções, valores retornados como resultado de uma substituição de expressão regular e para um assunto ou texto de entrada que seja aplicado a uma expressão regular. Pode ser o conteúdo de uma caixa de texto em um aplicativo, um arquivo no disco rígido ou o conteúdo de uma variável de string.

Largura constante com itálico
: Mostra o texto que deve ser substituído com valores fornecidos

pelo usuário ou por valores determinados pelo contexto.

‹Expressão regular›

Representa uma expressão regular, isolada, ou como você a digitaria na caixa de busca de uma aplicação. Espaços em expressões regulares são indicados com círculos cinzas, exceto quando usados no modo de espaçamento livre.

‹‹Texto substituto››

Representa o texto que substituirá as correspondências da expressão regular em uma operação de busca-e-substituição. Espaços no texto substituto são indicados por círculos cinzas.

Texto correspondente

Representa a parte do texto que corresponde a uma expressão regular.

…

A elipse cinza em uma expressão regular indica que você tem que “preencher o espaço em branco”, antes de usar a expressão regular. O texto de acompanhamento explica com o que você pode preenchê-lo.

CR, LF, e CRLF

CR, LF, e CRLF, dentro de caixas, representam caracteres reais de quebra de linha nas strings, ao invés de caracteres de escape, como \r, \n, e \r\n. Tais strings podem ser criadas pressionando Enter em um controle de edição multilinha de um aplicativo, ou utilizando strings constantes multilinhas no código-fonte, como as strings literais em C# ou as strings envolvidas em aspas triplas no Python.

↵

A seta de retorno, como você pode ver na tecla Enter ou Return em seu teclado, indica que tivemos de quebrar uma linha para ajustá-la à largura da página impressa. Ao digitar o texto em seu

código fonte, você não deve pressionar Enter, mas digitar tudo em uma única linha.

Este ícone significa uma dica, sugestão ou uma nota geral.

Este ícone indica uma advertência ou precaução.

Uso de códigos-exemplos, de acordo com a política da O'Reilly

Este livro está aqui para ajudá-lo. Geralmente, você pode utilizar o código deste livro em seus programas e documentação. Não há necessidade de nos contatar para permissões, a menos que você reproduza uma parte significativa do código. Por exemplo, escrever um programa que utilize vários pedaços de código deste livro não requer autorização. A venda ou distribuição de um CD-ROM com exemplos dos livros O'Reilly requer permissão. Responder uma pergunta citando este livro e dando o exemplo do código não precisa de permissão. Incorporar uma quantidade significativa de exemplos de código deste livro na documentação do seu produto requer permissão.

Agradecemos, mas não exigimos atribuição. Uma atribuição, geralmente, inclui o título, o autor, o editor e o ISBN (International Standard Book Number). Por exemplo: Livro de *receitas de Expressões Regulares*, por Jan Goyvaerts e Steven Levithan. "Direitos Autorais 2009 Jan Goyvaerts e Steven Levithan, 978-0-596-2068-7".

Se você sente que a utilização dos exemplos de código está fora dos padrões normais aqui citados, não hesite em contatar-nos em: *permissions@oreilly.com,* ou na Novatec (*novatec@novatec.com.br*).

Como entrar em contato conosco

Envie comentários e dúvidas sobre este livro para:

novatec@novatec.com.br

Temos uma página da web para este livro, onde incluímos lista de erratas, exemplos e qualquer outra informação adicional.

- Página da edição em português:

  http://www.novatec.com.br/livros/ercookbook

- Página da edição original, em inglês:

  http://oreilly.com/catalog/9780596520687

Este livro também tem outro website (em inglês):

http://www.regexcookbook.com

Para obter mais informações sobre livros da Novatec, acesse nosso site em:

http://www.novatec.com.br

Agradecimentos

Agradecemos Andy Oram, nosso editor na O'Reilly Media, Inc., por nos ajudar a enxergar como esse projeto seria, do início ao fim. Agradecemos também Jeffrey Friedl, Zak Greant, Nikolaj Lindberg, e Ian Morse, pela cuidadosa revisão técnica, o que torna este livro mais abrangente e preciso.

CAPÍTULO 1

Introdução às expressões regulares

Ao abrir este livro de receitas, você provavelmente ficará ansioso para inserir diretamente em seu código as deselegantes strings de parênteses e pontos de interrogação que encontrará nos próximos capítulos. Se estiver pronto para começar, fique à vontade: as expressões regulares práticas estão listadas e descritas nos capítulos 4 a 8.

Porém, os capítulos iniciais deste livro irão economizar muito tempo a longo prazo. Por exemplo, este capítulo introduz uma série de utilitários – alguns deles criados por um dos autores, Jan – que permitem testar e depurar uma expressão regular antes de embuti-la no código, onde os erros ficam mais difíceis de serem achados. Estes capítulos iniciais também mostram como utilizar as várias características e opções das expressões regulares para facilitar sua vida, ajudam a entender as expressões regulares, a fim de melhorar sua performance, e ensinam as diferenças sutis de tratamento das expressões regulares pelas diferentes linguagens de programação – e até mesmo pelas diferentes versões de sua linguagem favorita.

Dedicamos muitos esforços para estas questões de bastidores, confiantes de que você as lerá antes de começar a trabalhar, ou quando se sentir frustrado com a utilização de expressões regulares e, por isso, desejar reforçar seu nível de compreensão.

Definição de expressões regulares

No contexto deste livro, uma *expressão regular* é um tipo específico

de texto-padrão que você pode utilizar em muitos aplicativos modernos e em linguagens de programação. Você pode usá-las para verificar se a entrada de dados encaixa-se no padrão de texto, para encontrar um texto que corresponda a um padrão dentro de um conjunto maior de textos, para substituir o texto padrão por outro ou reorganizar bits de texto correspondentes, para dividir um bloco de texto em uma lista de subtextos e para dar um tiro no pé. Este livro o ajudará a entender exatamente o que você está fazendo, evitando, assim, desastres.

História do termo 'Expressão Regular'

O termo *expressão regular* vem da matemática e da teoria da ciência da computação. Ele reflete uma peculiaridade das expressões matemáticas, chamada *regularidade*. Essa expressão pode ser implementada em programas usando um autômato finito determinístico (DFA). Um DFA é uma máquina de estados finitos que não usa retrocesso (backtracking).

Os padrões de texto utilizados pelas primeiras ferramentas *grep* (Global Regular Expression and Print) eram expressões regulares no sentido matemático. Embora o nome tenha pegado, as expressões regulares modernas, no estilo Perl, não são expressões regulares no sentido matemático. Elas são implementadas com um autômato finito não determinístico (NFA). Você aprenderá tudo sobre retrocesso em breve. Tudo de que um programador prático precisa se lembrar é que alguns cientistas elitistas irritam-se com sua bem definida terminologia sendo sobrecarregada com usos e noções inerentes à tecnologia, que é muito mais útil no mundo real.

Se você utilizar expressões regulares com habilidade, elas podem simplificar muitas tarefas de programação e processamento de texto, além de permitir outras que não seriam possíveis sem elas. Você precisaria de dezenas, se não centenas de linhas de código procedural para extrair todos os endereços de e-mail de um documento – código tedioso de escrever e difícil de manter. Mas, com a expressão regular correta, como mostrado na receita 4.1, são necessárias apenas algumas linhas de código, ou talvez até uma única linha.

Porém, se você tentar fazer muita coisa com apenas uma expressão regular, ou utilizar expressões regulares onde não for realmente necessário, irá descobrir o sentido da seguinte citação[1]:

> Algumas pessoas, quando confrontadas com um problema, pensam: "Já sei, vou utilizar expressões regulares". Agora, elas têm dois problemas.

O segundo problema destas pessoas é o fato de não terem lido o manual do proprietário; este mesmo que você está segurando. Leia-o agora. A expressão regular é uma ferramenta poderosa. Se seu trabalho envolve manipulação ou extração de texto em um computador, o domínio das expressões regulares irá lhe poupar muitas horas de esforço.

Muitos sabores de expressões regulares

Tudo bem, o título da seção anterior era uma mentira. Nós não definimos o que são expressões regulares. Não podemos. Não existe nenhuma norma oficial que defina exatamente quais padrões de texto são expressões regulares e quais não são. Como você pode imaginar, cada programador de linguagens e cada desenvolvedor de aplicações de processamento de texto tem uma ideia diferente do que, exatamente, uma expressão regular deva ser. Por isso, agora estamos amarrados a um leque enorme de *sabores* de expressões regulares.

Felizmente, a maioria dos designers e programadores é preguiçosa. Por que criar algo totalmente novo, quando você pode copiar o que já foi feito? Como resultado, todos os sabores modernos de expressões regulares, incluindo os discutidos neste livro, podem traçar sua história a partir da linguagem de programação Perl. Chamamos esses sabores de *expressões regulares no estilo Perl*. A sintaxe desses sabores de expressões regulares é muito semelhante, e geralmente compatível, mas não totalmente.

Os escritores são preguiçosos, também. Geralmente digitamos *regex*, ou *regexp*, para designar uma única expressão regular, e *regexes* para indicar o plural.

Os sabores regex não correspondem, um-para-um, às linguagens de programação. Linguagens de script tendem a ter seus próprios sabores de expressão regular embutidos. Outras linguagens de programação contam com bibliotecas para suporte a expressão regular. Algumas bibliotecas estão disponíveis para várias

linguagens, enquanto algumas linguagens podem recorrer a diferentes bibliotecas.

Este capítulo introdutório trata apenas dos sabores de expressões regulares, e ignora completamente quaisquer considerações a respeito de programação. O capítulo 3 inicia as listagens de código, então você pode dar uma espiada no tópico “Linguagens de programação e sabores da expressão regular” daquele capítulo, para descobrir com quais sabores irá trabalhar. Por enquanto, ignore todas as coisas relacionadas à programação. As ferramentas listadas na próxima seção facilitam a exploração da sintaxe da expressão regular por meio do “aprenda fazendo”.

Sabores regex abordados neste livro

Para este livro, foram selecionados os sabores regex mais populares em uso, atualmente. Todos são sabores regex no *estilo Perl*. Alguns sabores têm mais recursos do que outros. Mas, se dois sabores têm uma mesma característica, eles tendem a utilizar a mesma sintaxe. Vamos apontar as poucas incoerências irritantes, conforme formos nos deparando com elas.

Todos estes sabores regex fazem parte de linguagens de programação e bibliotecas que estão em desenvolvimento ativo. A lista de sabores lhe diz quais versões este livro cobre. Mais adiante no livro, mencionamos o sabor sem quaisquer versões, caso a expressão regular atual funcione da mesma forma com todos os sabores. Quase sempre esse será o caso. Salvo as correções de bugs que afetam casos isolados, sabores regex não tendem a mudar, exceto pela adição de recursos que dão um novo significado à sintaxe, tratada anteriormente como um erro:

Perl

O suporte embutido do Perl para expressões regulares é a principal razão pela qual as regexes são populares hoje. Este livro cobre Perl 5.6, 5.8 e 5.10.

Muitos aplicativos e bibliotecas regex que dizem utilizar o Perl, ou

expressões regulares compatíveis com o Perl, na realidade, meramente utilizam expressões regulares no estilo Perl. Elas utilizam uma sintaxe de expressão regular similar à do Perl, mas não suportam o mesmo conjunto de características. Muito provavelmente, elas usam um dos sabores regex logo adiante. Esses sabores são todos no estilo Perl.

PCRE

PCRE é a biblioteca C das “Expressões regulares compatíveis com Perl”, desenvolvida por Philip Hazel. Você pode baixar esta biblioteca de fonte aberta em *http://www.pcre.org*. Este livro cobre as versões 4 a 7 da PCRE.

Embora PCRE afirme ser compatível com Perl, e provavelmente o é mais do que qualquer outro sabor neste livro, trata-se, realmente, de apenas um estilo Perl. Algumas funções, tais como o suporte a Unicode, são ligeiramente diferentes, e você não pode misturar código Perl em sua expressão regular, como o próprio Perl permite.

Devido a sua licença de código aberto e programação sólida, o PCRE encontrou seu caminho em muitas linguagens de programação e aplicações. Ele está embutido no PHP e empacotado em numerosos componentes Delphi. Se a aplicação diz suportar expressões regulares “compatíveis com Perl”, sem listar especificamente o sabor regex usado, ela provavelmente está usando o PCRE.

. NET

O Framework Microsoft .NET fornece um conjunto completo de sabores de expressões regulares no estilo Perl por meio do pacote System.Text.RegularExpressions. Este livro cobre do .NET 1.0 até o 3.5. Estritamente falando, só existem duas versões do System.Text.RegularExpressions: 1.0 e 2.0. Nenhuma alteração foi feita para as classes Regex nas versões de .NET 1.1, 3.0 e 3.5.

Qualquer linguagem de programação .NET, incluindo o C#, VB.NET, Delphi para .NET e até mesmo COBOL.NET, tem

acesso completo aos sabores das expressões regulares .NET. Se um aplicativo desenvolvido com .NET lhe oferecer suporte às expressões regulares, pode estar certo de que ele usa o sabor .NET, mesmo que diga utilizar “expressões regulares de Perl”. A única exceção é o Visual Studio (VS) em si. O ambiente de desenvolvimento integrado do VS (IDE) ainda usa o velho sabor de expressão regular utilizado desde o início, que não é o estilo Perl.

Java

Java 4 é a primeira versão Java a fornecer suporte interno a expressões regulares por meio do pacote java.util.regex. Ele ofuscou rapidamente as várias bibliotecas Java de expressões regulares de terceiros. Além de ser padrão e integrada, oferece um sabor de expressões regulares no estilo Perl, completo e com excelente desempenho, mesmo quando comparado com aplicações escritas em C. Este livro aborda o pacote java.util.regex em Java 4, 5 e 6.

Se você vem utilizando programas desenvolvidos em Java ao longo dos últimos anos, qualquer suporte a expressão regular oferecido por eles provavelmente utiliza o sabor Java.

JavaScript

Neste livro, usamos o termo *JavaScript* para indicar o sabor da expressão regular definida na versão 3 do padrão ECMA-262. Esta norma define a linguagem de programação ECMAScript, mais conhecida por suas implementações de JavaScript e JScript em diferentes navegadores. O Internet Explorer, do 5.5 até o 8.0, o Firefox, o Opera e o Safari, todos implementam a 3 ª edição da ECMA-262. No entanto, todos os navegadores apresentam vários bugs, levando-os a desviarem-se da norma. Apontamos essas questões em situações em que elas importam.

Se um site permite que você busque ou filtre usando uma expressão regular sem esperar por uma resposta do servidor web, ele usa o sabor regex JavaScript, o único sabor de

expressão regular que trabalha com todos os navegadores no lado do cliente. Mesmo o Microsoft VBScript e o Adobe ActionScript 3 utilizam-no.

Python

Python suporta expressões regulares por meio do seu módulo `re`. Este livro cobre o Python 2.4 e 2.5. O suporte às expressões regulares do Python mantém-se inalterado há muitos anos.

Ruby

O suporte às expressões regulares do Ruby é parte da própria linguagem Ruby, similar ao Perl. Este livro cobre o Ruby 1.8 e 1.9. Uma compilação padrão do Ruby 1.8 utiliza o sabor de expressões regulares fornecido diretamente pelo código-fonte Ruby. Uma compilação padrão do Ruby 1.9 utiliza a biblioteca Oniguruma de expressões regulares. O Ruby 1.8 pode ser compilado para utilizar a Oniguruma, e o Ruby 1.9 pode ser compilado para utilizar expressões regulares Ruby mais antigas. Neste livro, identificamos o sabor Ruby nativo como sendo o Ruby 1.8, e o sabor Oniguruma como o Ruby 1.9.

Para testar qual tipo de expressão regular Ruby seu site utiliza, tente utilizar a expressão regular `‹a++›`. O Ruby 1.8 dirá que a expressão regular é inválida, porque não suporta quantificadores possessivos, enquanto que no Ruby 1.9 ela corresponderá a uma sequência de um ou mais caracteres.

A biblioteca Oniguruma foi projetada para ser compatível com o Ruby 1.8, adicionando novas funcionalidades que não irão conflitar com as expressões regulares existentes. Os implementadores até deixaram funcionalidades que provavelmente deveriam ter sido alteradas, como o uso do `(?m)` para significar “o ponto corresponde a quebras de linha”, onde outros sabores de expressões regulares utilizam `(?s)`.

Pesquisa e substituição com

expressões regulares

Pesquisa-e-substituição é uma tarefa comum para as expressões regulares. Uma função de pesquisa-e-substituição envolve, como entrada, uma string de assunto, uma expressão regular e uma string de substituição. A saída é a string de assunto com todas as correspondências da expressão regular trocadas pelo texto de substituição.

Embora o texto de substituição não seja uma expressão regular, você pode utilizar certas sintaxes especiais para construir textos de substituição dinâmicos. Todos os sabores permitem que você reinsira o texto correspondido pela expressão regular ou por um grupo de captura dentro do texto de substituição. As receitas 2.20 e 2.21 explicam isso. Alguns sabores também suportam a inserção do texto correspondido no texto de substituição, como mostra a receita 2.22. No capítulo 3, a receita 3.16 ensina como gerar um texto de substituição diferente para cada correspondência no código.

Muitos sabores de textos de substituição

Ideias diferentes, de diferentes desenvolvedores de programas de expressões regulares, levaram a uma ampla gama de sabores, cada um com uma sintaxe e conjuntos de recursos diferentes. A história, no caso dos textos de substituição, não é diferente. Na verdade, há mais sabores de textos de substituição do que sabores de expressões regulares. Construir um mecanismo de expressão regular não é fácil. A maioria dos programadores prefere reutilizar um já existente, e a aplicação de uma função de pesquisa-e-substituição em um mecanismo de expressão regular existente é bem fácil. O resultado é que existem muitos sabores de textos de substituição, no caso das bibliotecas de expressões regulares, que não têm recursos nativos de pesquisa-e-substituição.

Felizmente, todos os sabores de expressões regulares neste livro têm sabores de textos de substituição correspondentes, exceto o

PCRE. Esta lacuna no PCRE complica a vida dos programadores que utilizam sabores nele baseados. A biblioteca de código aberto do PCRE não inclui as funções necessárias para fazer substituições. Assim, todos os aplicativos e linguagens de programação que se baseiam no PCRE precisam providenciar sua própria função de pesquisa-e-substituição. A maioria dos programadores tenta copiar sintaxes existentes, mas nunca as fazem exatamente da mesma maneira.

Este livro cobre os seguintes sabores de textos de substituição. Consulte “Muitos sabores de expressões regulares”, para mais detalhes sobre os sabores de expressões regulares que correspondem aos sabores de textos de substituição:

Perl

Perl tem o suporte embutido para substituição de expressões regulares por meio do operador s/regex/replace/. O sabor de texto de substituição do Perl corresponde ao sabor de expressão regular do Perl. Este livro cobre do Perl 5.6 ao Pearl 5.10. Esta última versão adiciona suporte a retrorreferências (backreferences) no texto de substituição, ao acrescentar capturas nomeadas à sintaxe da expressão regular.

PHP

Neste livro, o sabor PHP de texto de substituição refere-se à função preg_replace. Essa função utiliza o sabor das expressões regulares PCRE e o sabor de texto de substituição do PHP.

Outras linguagens de programação que utilizam o PCRE não utilizam o mesmo sabor de texto de substituição que o PHP. Dependendo de onde o designer de sua linguagem de programação obteve inspiração, a sintaxe de texto de substituição pode ser semelhante à do PHP, ou a qualquer outro sabor de textos de substituição deste livro.

O PHP também possui uma função ereg_replace. Essa função utiliza um sabor diferente de expressão regular (POSIX ERE), além de usar um sabor diferente de texto de substituição. As

funções ereg do PHP não são discutidas neste livro.

.NET

O pacote System.Text.RegularExpressions fornece várias funções de pesquisa-e-substituição. O sabor de texto de substituição do .NET corresponde ao sabor de expressão regular .NET. Todas as versões .NET utilizam o mesmo sabor de texto de substituição. Os novos recursos de expressão regular no .NET 2.0 não afetam a sintaxe de texto de substituição.

Java

O pacote java.util.regex tem funções internas de pesquisa-e-substituição. Este livro aborda o Java 4, 5 e 6. Todos utilizam a mesma sintaxe de texto de substituição.

JavaScript

Neste livro, utilizamos o termo *JavaScript* para indicar tanto o sabor de texto de substituição, quanto o sabor de expressão regular, definidos na 3ª edição do padrão ECMA-262.

Python

O módulo re do Python oferece a função sub para pesquisa-e-substituição. O sabor de texto de substituição do Python corresponde ao sabor de expressão regular Python. Este livro cobre o Python 2.4 e 2.5. O suporte às expressões regulares do Python tem se mantido estável por muitos anos.

Ruby

O suporte às expressões regulares do Ruby é parte constituinte da linguagem em si, incluindo a função de pesquisa-e-substituição. Este livro aborda o Ruby 1.8 e 1.9. Uma compilação padrão do Ruby 1.8 utiliza expressões regulares fornecidas diretamente pelo código-fonte, enquanto uma compilação padrão do Ruby 1.9 utiliza a biblioteca Oniguruma de expressões regulares. O Ruby 1.8 pode ser compilado para utilizar a biblioteca Oniguruma, e o Ruby 1.9 pode ser compilado para utilizar uma expressão regular Ruby mais antiga. Neste livro,

identificamos o sabor nativo Ruby como sendo o Ruby 1.8 e o sabor Oniguruma como sendo o Ruby 1.9.

A sintaxe de texto de substituição das versões 1.8 e 1.9 do Ruby é a mesma, exceto pelo fato de que o Ruby 1.9 adiciona retrorreferências ao texto de substituição. Captura nomeada é um recurso novo nas expressões regulares do Ruby 1.9.

Ferramentas para se trabalhar com expressões regulares

A menos que você já venha programando com expressões regulares por algum tempo, recomendamos, em primeiro lugar, a manipulação das expressões regulares em uma ferramenta, e não diretamente no código-fonte. O exemplo das expressões regulares, neste capítulo e no capítulo 2, são expressões regulares simples que não contêm o escape extra que uma linguagem de programação (mesmo em um Unix shell) exige. Você pode digitar estas expressões regulares diretamente na caixa de busca de um aplicativo.

O capítulo 3 explica como corresponder expressões regulares ao seu código-fonte. Citar uma expressão regular literal como uma string torna-a ainda mais difícil de ler, porque as regras de escape da string se misturam com as regras de escape das expressões regulares. Evitaremos isso até a receita 3.1. Depois de entender os conceitos básicos das expressões regulares, você será capaz de enxergar em meio à floresta de barras invertidas.

As ferramentas descritas nesta seção também fornecem depuração, verificação de sintaxe e outros feedbacks, elementos que você não teria na maioria dos ambientes de programação. Portanto, ao desenvolver expressões regulares para seus aplicativos, você pode considerar útil construir expressões regulares complicadas em uma dessas ferramentas, antes de inseri-las em seu programa.

RegexBuddy

No momento em que escrevemos isso, RegexBuddy (Figura 1.1) é a ferramenta mais completa disponível para criação, teste e implementação de expressões regulares. Ela tem a habilidade única de emular todos os sabores de expressões regulares discutidos neste livro, e até mesmo de fazer conversões entre os diferentes sabores.

RegexBuddy foi projetado e desenvolvido por Jan Goyvaerts, um dos autores deste livro. Projetar e desenvolver RegexBuddy fez de Jan um especialista em expressões regulares, e a utilização do RegexBuddy ajudou a tornar o coautor Steven em um viciado em expressões regulares, a ponto de produzir este livro para a O'Reilly.

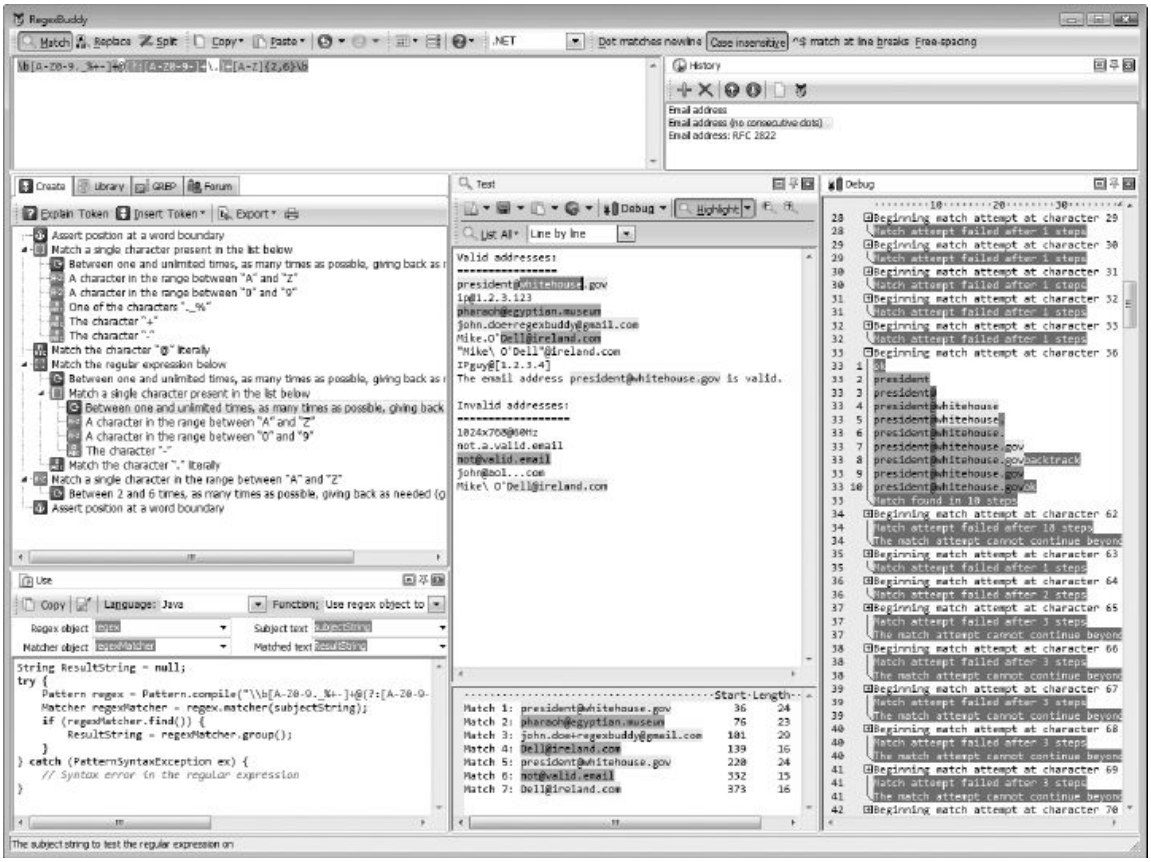


Figura 1.1 – RegexBuddy.

Se a captura de tela (Figura 1.1) parece um pouco carregada é porque quisemos dispor a maioria dos painéis lado a lado, para mostrar a extensa funcionalidade do RegexBuddy. A exibição

padrão acopla todos os painéis perfeitamente em uma linha de guias. Você também pode arrastar os painéis para um monitor secundário.

Para experimentar uma das expressões regulares mostradas neste livro, simplesmente digite-a na caixa de edição, no topo da janela do RegexBuddy. O RegexBuddy aplica automaticamente a sintaxe, destacando a sua expressão regular e fazendo com que os erros e parênteses incompatíveis fiquem óbvios.

O painel Create cria automaticamente uma análise detalhada, em inglês, enquanto digitamos a regex. Dê um duplo clique em qualquer descrição, na árvore da expressão regular, para editar essa parte de sua expressão regular. Você pode inserir novas partes manualmente em sua expressão regular, ou clicar no botão Insert Token e selecionar o que quiser em um menu. Por exemplo, se você não se lembra da complexa sintaxe para um procedimento de antecipação positiva (positive lookahead)), pode solicitar ao RegexBuddy que insira os caracteres apropriados.

Digite ou cole algum texto de exemplo no painel Test. Quando o botão Highlight estiver ativo, o RegexBuddy automaticamente sublinha o texto correspondido pela regex.

Alguns dos botões mais utilizados são:

List All

Exibe uma lista de todas as correspondências.

Replace

O botão Replace, na parte superior, exibe uma nova janela que permite a digitação do texto de substituição. O botão Replace, na caixa Test, permite que você visualize o texto de assunto após as substituições feitas.

Split (O botão no painel Test, e não o do topo)

Trata a expressão regular como um separador e divide o assunto em tokens, de acordo com os locais em que as correspondências foram encontradas em seu texto de assunto, usando a sua

expressão regular.

Clique em qualquer um desses botões e selecione Update Automatically para fazer o RegexBuddy manter os resultados em sincronia, dinamicamente, ao editar sua regex ou texto de assunto.

Para ver exatamente como funcionará sua regex (ou não), clique em uma correspondência destacada, ou no ponto onde a regex falha no painel Test, e clique no botão Debug. O RegexBuddy mudará para o painel Debug, mostrando passo a passo todos os processos de correspondências. Clique em qualquer lugar no texto de saída do depurador para ver qual símbolo regex corresponde ao texto em que você clicou. Clique em sua expressão regular para destacar esta parte da regex no depurador.

No painel Use, escolha sua linguagem de programação favorita. Em seguida, selecione uma função para gerar instantaneamente o código-fonte que implementa sua regex. Os modelos de código-fonte do RegexBuddy são totalmente editáveis por meio de seu editor interno. Você pode adicionar novas funções e até mesmo novas linguagens, ou alterar as fornecidas.

Para testar a sua regex em um conjunto maior de dados, alterne para o painel GREP, para pesquisar (e substituir) em qualquer número de arquivos e pastas.

Quando você encontrar uma regex em seu código-fonte, copie-a para a área de transferência, incluindo as aspas delimitadoras ou barras. No RegexBuddy, clique no botão Paste, na parte superior, e selecione o estilo de string de sua linguagem de programação. Sua regex aparecerá, então, no RegexBuddy como uma regex pura, sem as aspas extras e escapes necessários para literais strings. Utilize o botão Copy, na parte superior, para criar uma string na sintaxe desejada, e cole-a de volta em seu código-fonte.

Conforme sua experiência aumentar, é possível construir uma biblioteca de expressões regulares no painel Library. Certifique-se de adicionar uma descrição detalhada e um assunto de teste ao armazenar uma regex. As expressões regulares podem ser

enigmáticas, mesmo para especialistas.

Se você realmente não consegue entender uma regex, clique no painel Forum e, em seguida, no botão Login. Se você comprou o RegexBuddy, a tela de login aparecerá. Clique em OK, e você estará imediatamente conectado ao Forum do Usuário RegexBuddy. Steven e Jan às vezes estão por lá.

RegexBuddy roda em Windows 98, ME, 2000, XP e Vista. Para os fãs do Linux e da Apple, o RegexBuddy também vai bem no VMware, Parallels, CrossOver Office e, com algumas restrições, no WINE. Você pode baixar uma cópia de avaliação gratuita do RegexBuddy em *http://www.regexbuddy.com/RegexBuddyCookbook.exe*. Exceto pelo fórum de usuários, a cópia é totalmente funcional para sete dias de uso.

RegexPal

RegexPal (Figura 1.2) é um testador on-line de expressões regulares criado por Steven Levithan, um dos autores deste livro. Tudo de que você precisa para usá-lo é um navegador moderno. RegexPal é inteiramente escrito em JavaScript, portanto ele suporta apenas o sabor regex JavaScript implementado no navegador que estiver usando para acessá-lo.

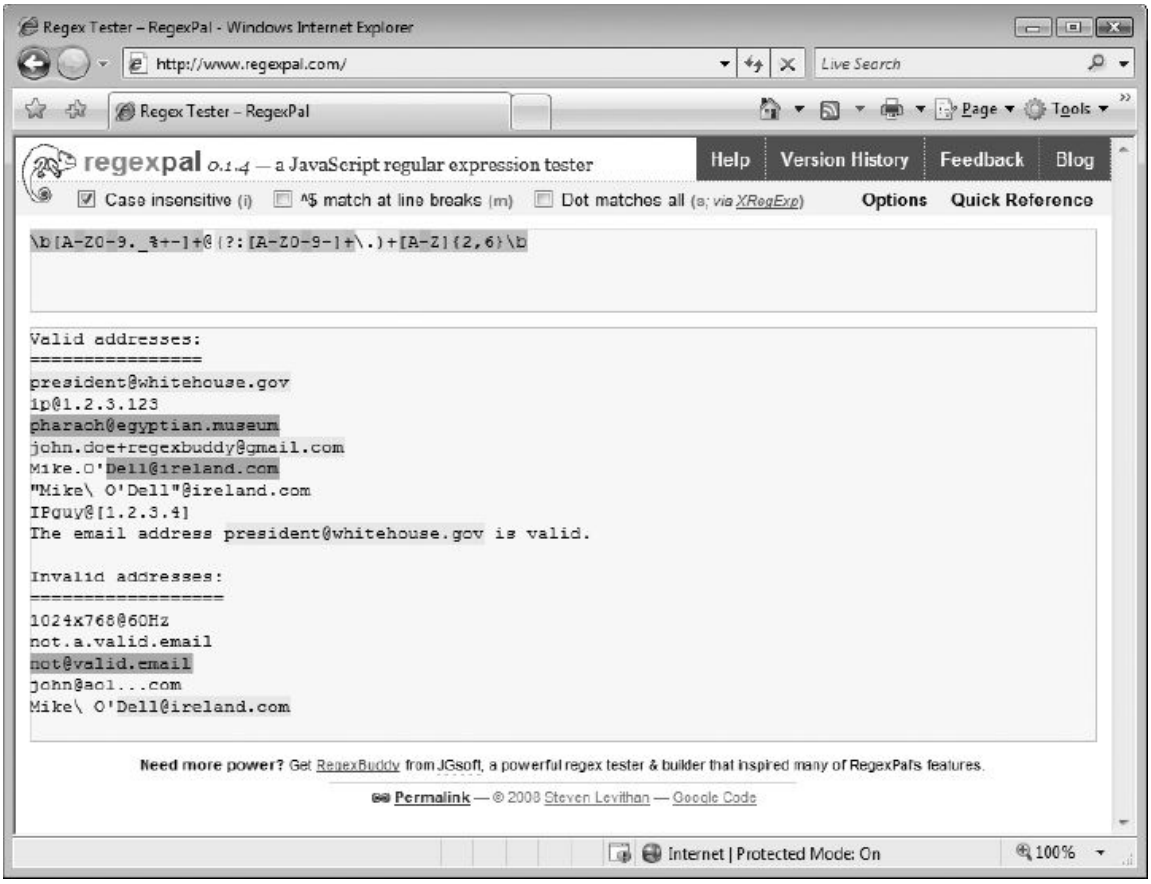


Figura 1.2 – RegexPal.

Para experimentar uma das expressões regulares mostradas neste livro, vá a *http://www.regexpal.com*. Digite a regex dentro da caixa que diz: “Enter regex here”. O RegexPal aplica automaticamente o destacamento de sintaxe na sua expressão regular, revelando imediatamente qualquer erro de sintaxe. O RegexPal tem consciência das diferenças entre navegadores, e dos problemas, oriundos destas diferenças, que podem arruinar seu trabalho ao lidar com expressões regulares em JavaScript. Se uma determinada sintaxe não funcionar corretamente em alguns navegadores, o RegexPal vai destacá-la como um erro.

Agora, digite ou cole algum exemplo de texto na caixa que diz: “Enter test data here.” O RegexPal destaca automaticamente o texto correspondido por sua regex.

Não há botões nos quais clicar, fazendo do RegexPal um dos mais convenientes testadores on-line para expressões regulares.

Mais testadores regex online

Criar um simples testador de expressão regular on-line é fácil. Se você tem algumas habilidades básicas de desenvolvimento web, a informação no capítulo 3 é tudo de que precisa para criar seu próprio testador. Centenas de pessoas já o fizeram, mas algumas delas acrescentaram recursos extras, tornando-os dignos de menção.

regex.larsolavtorvik.com

Lars Olav Torvik disponibilizou um excelente testador de expressão regular on-line em *http://regex.larsolavtorvik.com* (ver figura 1.3).

Figura 1.3 – regex.larsolavtorvik.com.

Para começar, selecione o sabor da expressão regular com a qual esteja trabalhando, clicando sobre o nome do sabor no topo da página. Lars oferece PHP PCRE, PHP POSIX e JavaScript. PHP PCRE, sabor regex PCRE discutido neste livro, é usado para funções preg PHP. O POSIX é um sabor regex antigo e limitado, usado pelas funções ereg do PHP, não discutidas neste livro. Se você selecionar o JavaScript, estará trabalhando com a implementação JavaScript do seu navegador.

Digite sua expressão regular no campo Pattern (padrão) e seu texto de assunto no campo Subject (assunto). Em seguida, o campo Matches (correspondências) exibirá o texto de assunto com as correspondências da regex em destaque. O campo Code exibe uma única linha de código-fonte, que aplica sua regex a seu texto de assunto. Copiar e colar este código em seu editor de código lhe poupará o trabalho enfadonho de converter manualmente sua regex em uma string literal. Qualquer string ou array retornado pelo código é exibido no campo Result. Como o Lars utilizou a tecnologia Ajax para construir seu site, os resultados são atualizados em apenas alguns momentos, para todos os sabores. Para utilizar a ferramenta você tem que estar on-line, pois o PHP é processado no servidor, e não no navegador.

A segunda coluna exibe uma lista de comandos e opções regex. Estas dependem do sabor regex. Os comandos regex, tipicamente, incluem operações de correspondência, substituição e divisão. As opções da regex são comuns, tais como não-diferenciação entre letras maiúsculas e minúsculas (case insensitivity), bem como opções específicas a cada implementação. Estes comandos e opções são descritos no capítulo 3.

Nregex

http://www.nregex.com (Figura 1.4) é um testador simples, construído com a tecnologia .NET por David Seruyange. Embora o site não diga qual sabor ele implementa, o .NET 1.x era usado no momento da redação deste texto.

O layout da página é um pouco confuso. Digite sua expressão regular no campo, sob o rótulo Regular Expression, e defina as opções da regex utilizando as caixas de seleção abaixo do campo. Digite o texto de assunto, na caixa grande que aparece na parte inferior, substituindo o texto padrão If I just had $5,00 then "she" wouldn't be so @#$! mad.. Se o assunto for uma página da Web, digite a URL no campo “Load Target From URL” e clique no botão Load, abaixo do campo de entrada. Se o seu assunto for um arquivo em seu disco rígido, clique no botão Browse, localize o arquivo desejado e, em seguida, clique no botão Load, abaixo do campo de entrada.

Seu texto de assunto aparecerá duplicado no campo “Matches & Replacements” (Correspondências & Substituições) no centro da página web, com as correspondências da regex em destaque. Se você digitar algo no campo Replacement String (String de Substituição), o resultado da pesquisa-e-substituição será apresentado. Se sua expressão regular for inválida, aparecerá ....

A correspondência da regex é feita em um código .NET rodando no servidor, então você precisa estar on-line para funcionar. Se as atualizações automáticas estiverem lentas, talvez seja porque o texto de amostra seja muito longo, então assinale a opção “Manually Evaluate Regex” (Avaliar a Expressão Regular Manualmente), na caixa de seleção acima do campo da sua expressão regular, para mostrar o botão Evaluate. Clique nesse botão para atualizar a visualização em “Matches & Replacements”.

Rubular

Michael Lovitt disponibilizou um testador de regex minimalista on-line em *http://www.rubular.com* (Figura 1.5), usando o sabor regex Ruby 1.8.

Figura 1.4 – Nregex.

Digite a sua expressão regular na caixa entre as duas barras, abaixo de “Your Regular Expression”. Você pode ativar a não-diferenciação entre maiúsculas e minúsculas, digitando um i na pequena caixa posicionada após a segunda barra. Da mesma forma, se você preferir, ative a opção “a dot matches a line break” (“um ponto corresponde a uma quebra de linha”), digitando um m nessa mesma caixa. im ativa as duas opções. Embora estas convenções possam parecer confusas se você for iniciante no Ruby, elas satisfazem a

sintaxe /regex/im usada para especificar uma regex no código-fonte do Ruby.

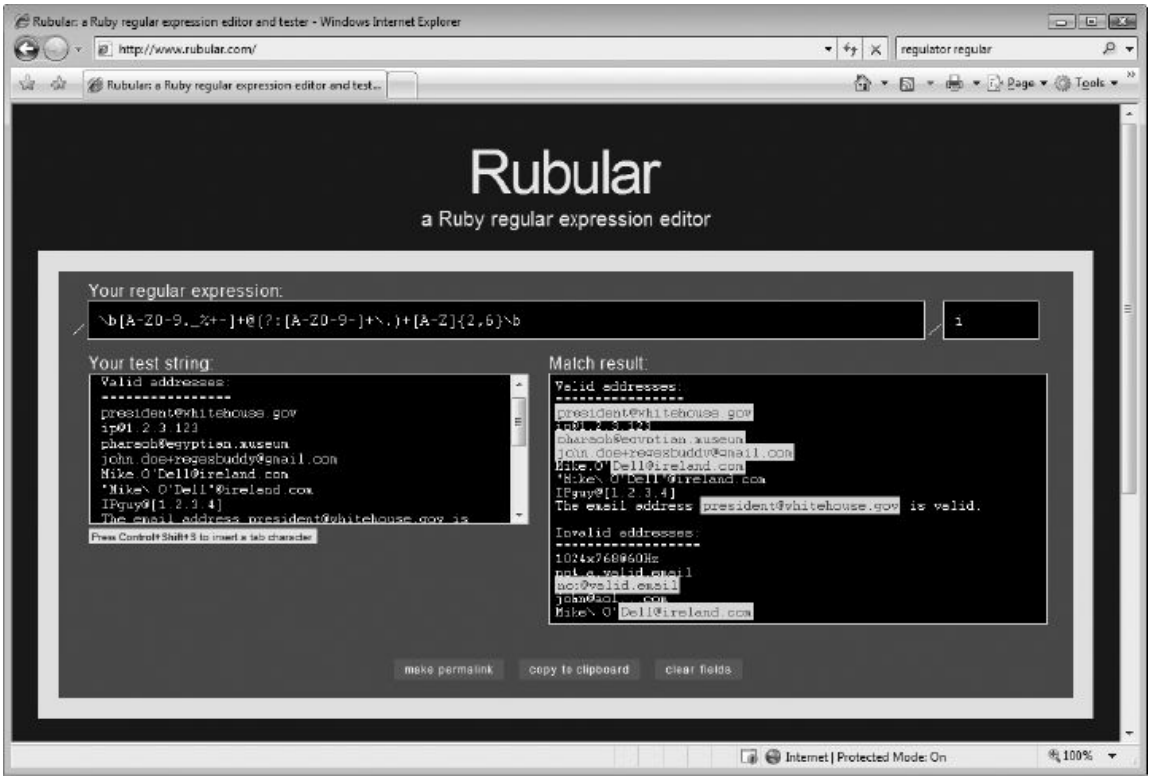


Figura 1.5 – Rubular.

Digite ou cole seu texto de assunto na caixa “Your test string” e aguarde um instante. Uma nova caixa “Match result” aparecerá à direita, mostrando seu texto de assunto com todas as correspondências da expressão regular em destaque.

myregexp.com

Sergey Evdokimov criou vários testadores de expressão regular para desenvolvedores Java. A página em *http://www.myregexp.com* (Figura 1.6) oferece um testador regex on-line. É um applet Java que roda no seu navegador. O Java 4 (ou posterior) precisa estar instalado em seu computador. O applet usa o pacote java.util.regex para avaliar suas expressões regulares, uma funcionalidade nova do Java 4. Neste livro, o sabor regex “Java” refere-se a este pacote.

Digite sua expressão regular na caixa Regular Expression. Use o menu Flags para definir as opções da regex que você desejar. Três

dessas opções também possuem caixas de seleção diretas.

Se você quiser testar uma regex já existente como uma string de código Java, copie toda a sequência para a área de transferência. No testador myregexp.com, clique no menu Edit e, em seguida, “Paste Regex from Java String” (“Cole a expressão regular a partir de uma String Java”). No mesmo menu, escolha “Copy Regex for Java Source” (“Copie a expressão regular para um fonte Java”) quando terminar de editar a expressão regular. O menu Edit também possui comandos semelhantes para JavaScript e XML.

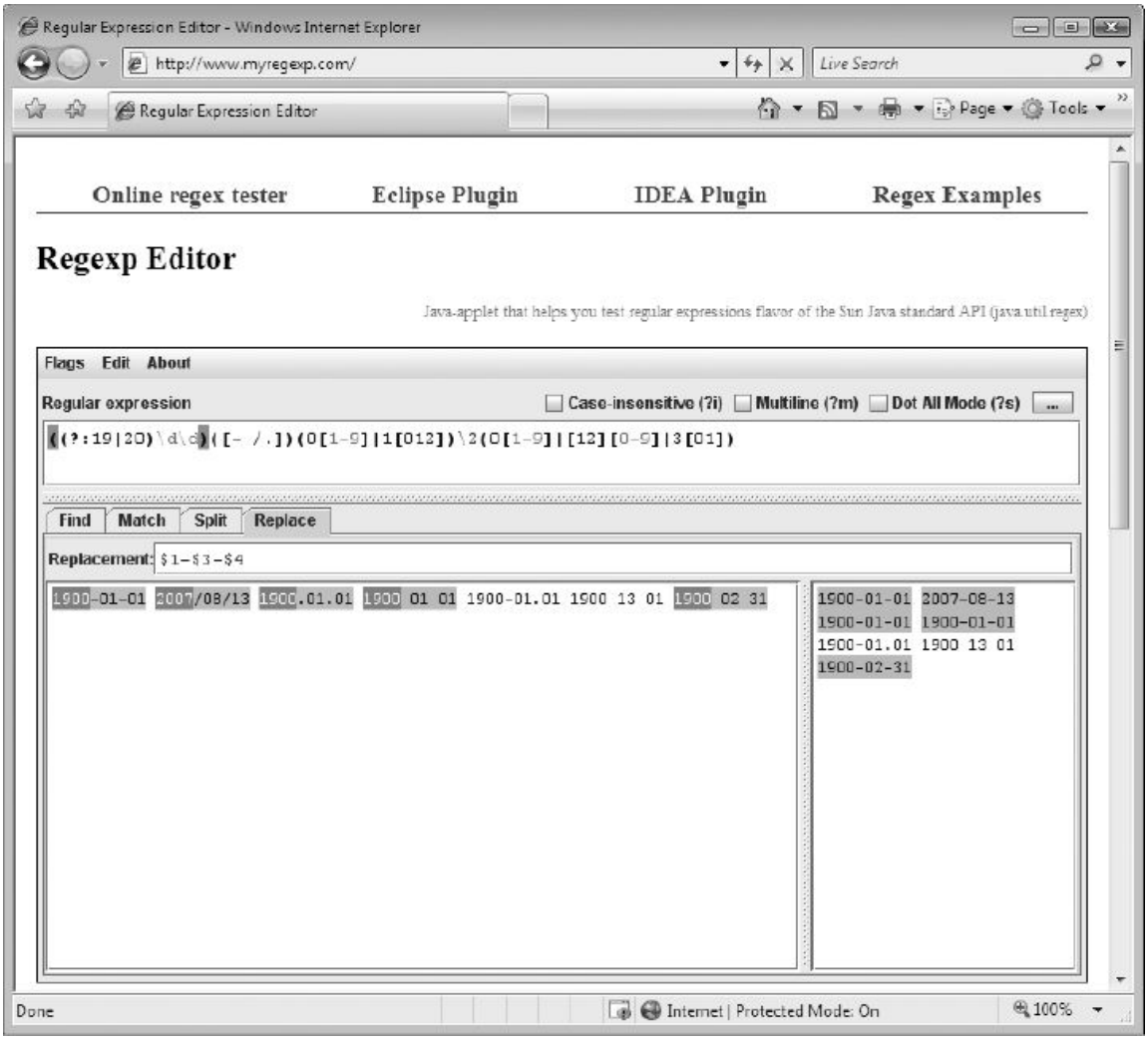


Figura 1.6 – myregexp.com

Abaixo da expressão regular, existem quatro guias que rodam quatro testes diferentes:

Find

Destaca todas as correspondências da expressão regular no texto de assunto. Estas são as correspondências encontradas pelo método Matcher.find() do Java.

Match

Testa se a expressão regular corresponde inteiramente ao texto de amostra. Caso isso aconteça, todo o texto é realçado. Isto é o que fazem os métodos String.matches() e Matcher.matches().

Split

A segunda caixa à direita mostra o array de strings retornado por String.split() ou Pattern.split(), quando usados com sua expressão regular e o texto de assunto.

Replace

Digite o texto de substituição e a caixa à direita exibirá o texto retornado por String.replaceAll() ou Matcher.replaceAll().

Você pode encontrar outros testadores regex de Sergey por meio dos links no topo da página em *http://www.myregexp.com*. Um deles é um plug-in para o Eclipse, e o outro, um plug-in para o IntelliJ IDEA.

reAnimator

O reAnimator, de Oliver Steele, encontrado em *http://osteele.com/tools/reanimator* (Figura 1.7), não vai trazer uma regex morta de volta à vida. Pelo contrário, ele é uma pequena e divertida ferramenta que mostra uma representação gráfica das máquinas de estados finitos que um mecanismo de expressão regular utiliza para realizar uma pesquisa de expressão regular.

A sintaxe de expressão regular do reAnimator é muito limitada. É compatível com todos os sabores discutidos neste livro. Qualquer regex que você animar com o reAnimator trabalhará com qualquer um dos sabores deste livro, mas o inverso, definitivamente, não é verdade. Isso acontece porque as expressões regulares do

reAnimator são regulares no sentido matemático. A barra lateral “História do termo Expressão Regular”, explica isso brevemente.

Comece indo até a caixa Pattern, na parte superior da página, e pressione o botão Edit.Digite sua expressão regular no campo Pattern e clique em Set. Digite lentamente o texto de assunto no campo Input.

Ao digitar cada caractere, bolas coloridas irão mover-se através da máquina de estados para indicar o ponto final obtido, até o momento, na máquina. Bolas azuis indicam que a máquina de estados aceita a entrada, mas necessita de mais entradas para uma correspondência completa. Bolas verdes indicam que a entrada corresponde ao padrão. Nenhuma bola significa que a máquina de estados não corresponde à entrada.

O reAnimator mostrará uma correspondência apenas se a expressão regular corresponder à string de entrada completa, como se você a tivesse colocado entre âncoras ‹^› e ‹$›. Esta é outra propriedade das expressões que são regulares no sentido matemático.

Mais testadores de expressões regulares para desktop

Expresso

Expresso (não confundir com o café expresso) é um aplicativo .NET para criar e testar expressões regulares. Você pode baixá-lo em *http://www.ultrapico.com/Expresso.htm*. O Framework .NET 2.0, ou posterior, deve estar instalado em seu computador.

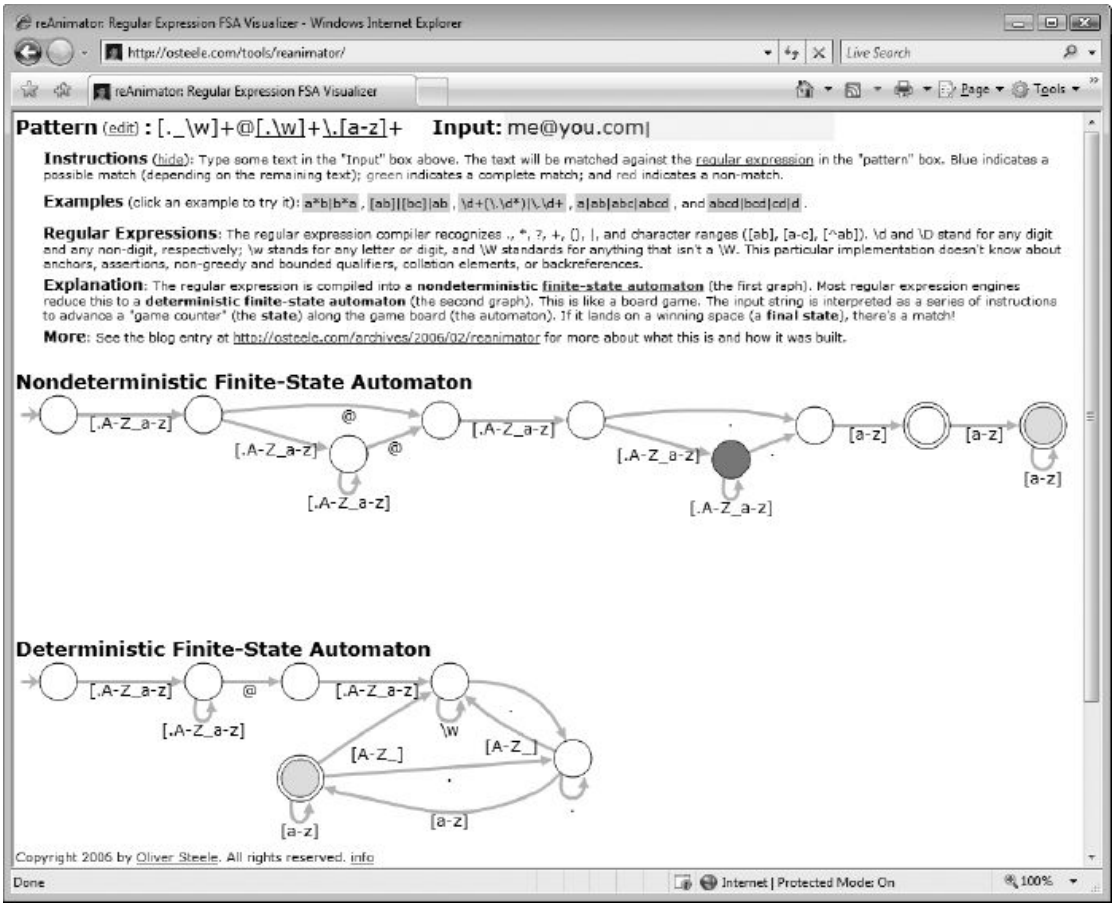


Figura 1.7 – reAnimator.

O programa é gratuito para testes por 60 dias. Após o período de experimentação, é necessário registrá-lo, ou ele irá parar de funcionar. A inscrição é gratuita, mas requer seu endereço de e-mail para a Ultrapico. A chave de registro é enviada por e-mail.

O Expresso exibe uma tela, como a mostrada na figura 1.8. A caixa Regular Expression, onde você digita sua expressão regular, fica sempre visível. O realce de sintaxe não está disponível. A caixa Regex Analyzer cria automaticamente uma breve análise, em inglês, da sua expressão regular. Ela também fica sempre visível.

No modo Design, você pode definir as opções de correspondências, como “Ignore Case”, na parte inferior da tela. A maior parte do espaço da tela é ocupado por uma fileira de guias, nas quais você pode selecionar o token de expressão regular que deseja inserir. Se tiver dois monitores, ou um monitor grande, clique no botão Undock,

para que a fileira de guias flutue. Então, você também poderá construir sua expressão regular no outro modo (Test Mode, ou Modo Teste).

Em modo de teste, digite,ou cole o texto de assunto no canto inferior esquerdo. Em seguida, clique no botão Run Match (Executar Correspondência) para obter uma lista de todas as correspondências na caixa Search Results (Resultados da Pesquisa). Não é aplicado nenhum realce ao texto de assunto. Clique em uma correspondência, nos resultados, para selecionar a correspondência no texto de assunto.

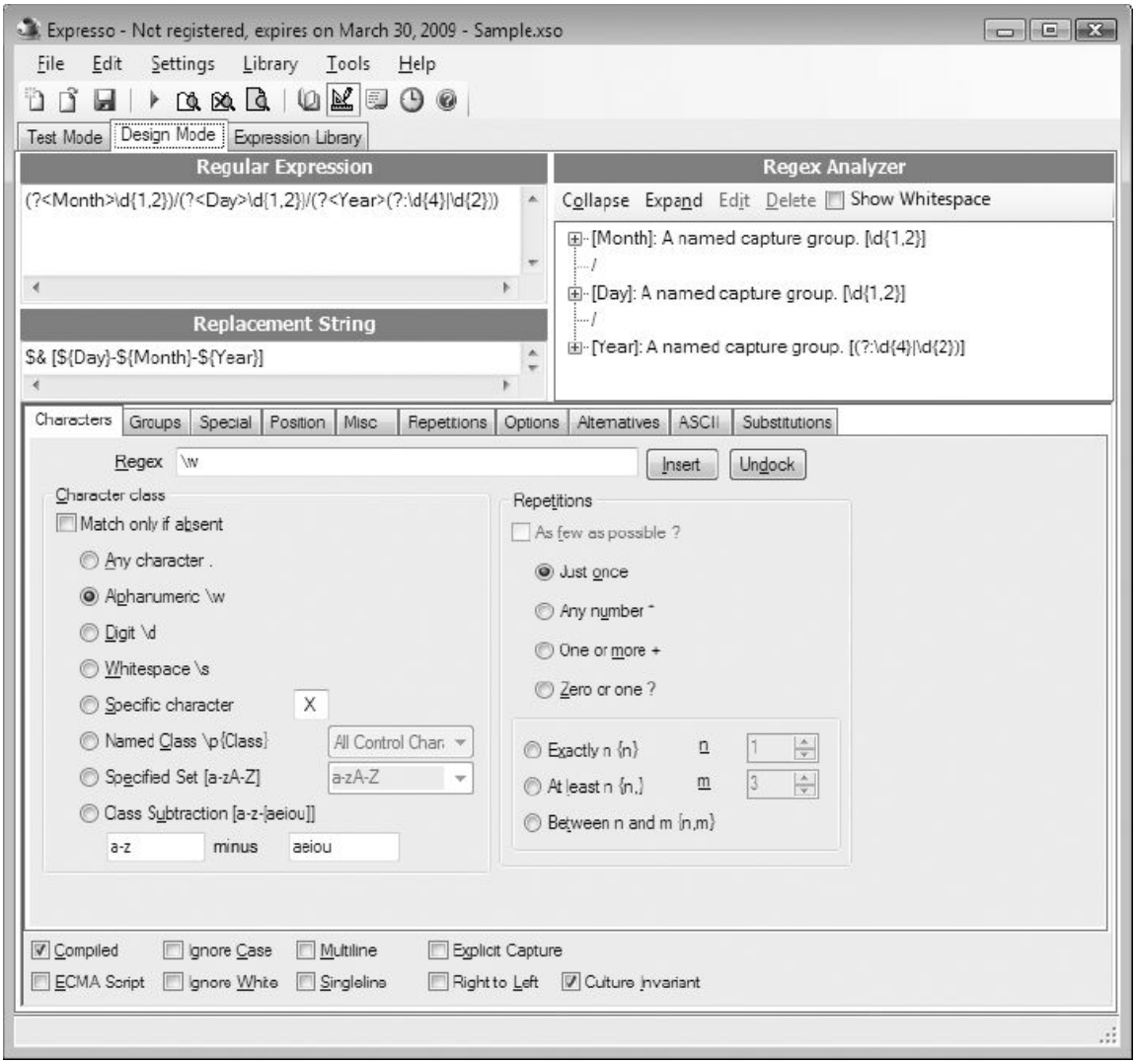


Figura 1.8 – Expresso.

A guia Expression Library (Biblioteca de Expressões) mostra uma

lista de assuntos de expressões regulares e uma lista de expressões regulares recentes. Sua regex é adicionada à lista cada vez que você pressiona Run Match.

Você pode editar a biblioteca por meio do menu Library, na barra de menu principal.

The Regulator

The Regulator, que você pode baixar em *http://sourceforge.net/projects/regulator*, não é seguro para uso em mergulhos submarinos ou em botijões de gás de cozinha[2]; trata-se de outra aplicação .NET para criação e teste de expressões regulares. A versão mais recente requer .NET 2.0 ou posterior. As versões mais antigas para .NET 1.x ainda podem ser baixadas. The Regulator possui código aberto, e nenhum pagamento ou registro é necessário.

The Regulator faz tudo em uma tela (Figura 1.9). Na guia New Document, você digita sua expressão regular. Realce de sintaxe é aplicado automaticamente, mas os erros de sintaxe em sua regex não ficam óbvios. Clique com o botão direito do mouse para selecionar o token de regex que desejar inserir a partir de um menu. Você pode definir as opções da expressão regular por meio dos botões na barra de ferramentas principal. Os ícones são um pouco enigmáticos. Espere a dica para ver qual opção você está definindo em cada botão.

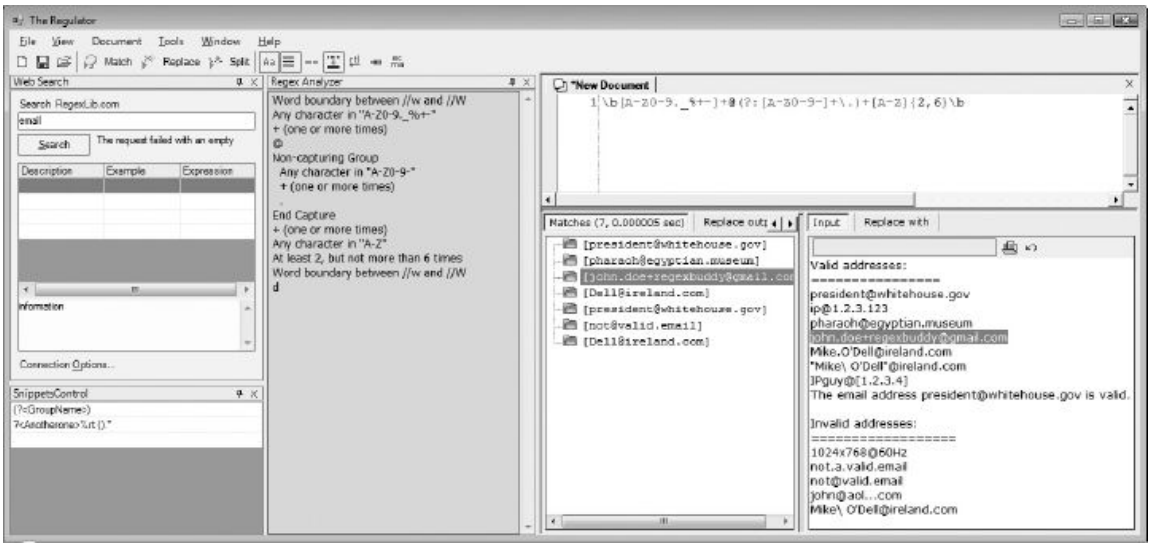


Figura 1.9 – The Regulator.

Abaixo da área para sua regex, à direita, clique no botão Input, para exibir o espaço para colar seu texto de assunto. Clique no botão Replace with para digitar o texto de substituição, caso queira fazer uma pesquisa-e-substituição. Abaixo da regex, à esquerda, você verá os resultados da sua operação regex. Os resultados não são atualizados automaticamente: você deve clicar no botão Match, Replace ou Split, na barra de ferramentas, para atualizar os resultados. Nenhum realce é aplicado à entrada. Clique em uma correspondência nos resultados para selecioná-la no texto de assunto.

O painel Regex Analyzer mostra, em inglês, uma análise simples de sua expressão regular, mas ele não é automático, ou interativo. Para atualizar a análise, selecione Regex Analyzer no menu View, mesmo que já esteja visível. Clicar sobre a análise somente move o cursor de texto.

grep

O nome *grep* é derivado do comando g/re/p, que realizava uma pesquisa de expressão regular no editor de texto *ed* do Unix, uma das primeiras aplicações a suportar expressões regulares. Este comando era tão popular que, atualmente, todos os sistemas Unix

possuem um utilitário grep dedicado à pesquisa em arquivos utilizando uma expressão regular. Se você estiver usando Unix, Linux ou Mac OS X, digite man grep em uma janela do terminal para aprender tudo sobre ele.

As três ferramentas seguintes são aplicativos do Windows que fazem o mesmo que o grep, e muito mais.

PowerGREP

PowerGREP, desenvolvido por Jan Goyvaerts, um dos autores deste livro, é provavelmente a ferramenta *grep* mais completa disponível para a plataforma Microsoft Windows (Figura 1.10). PowerGREP usa um sabor regex personalizado, que combina o melhor dos sabores discutidos neste livro. Este sabor é identificado como “JGsoft” dentro do RegexBuddy.

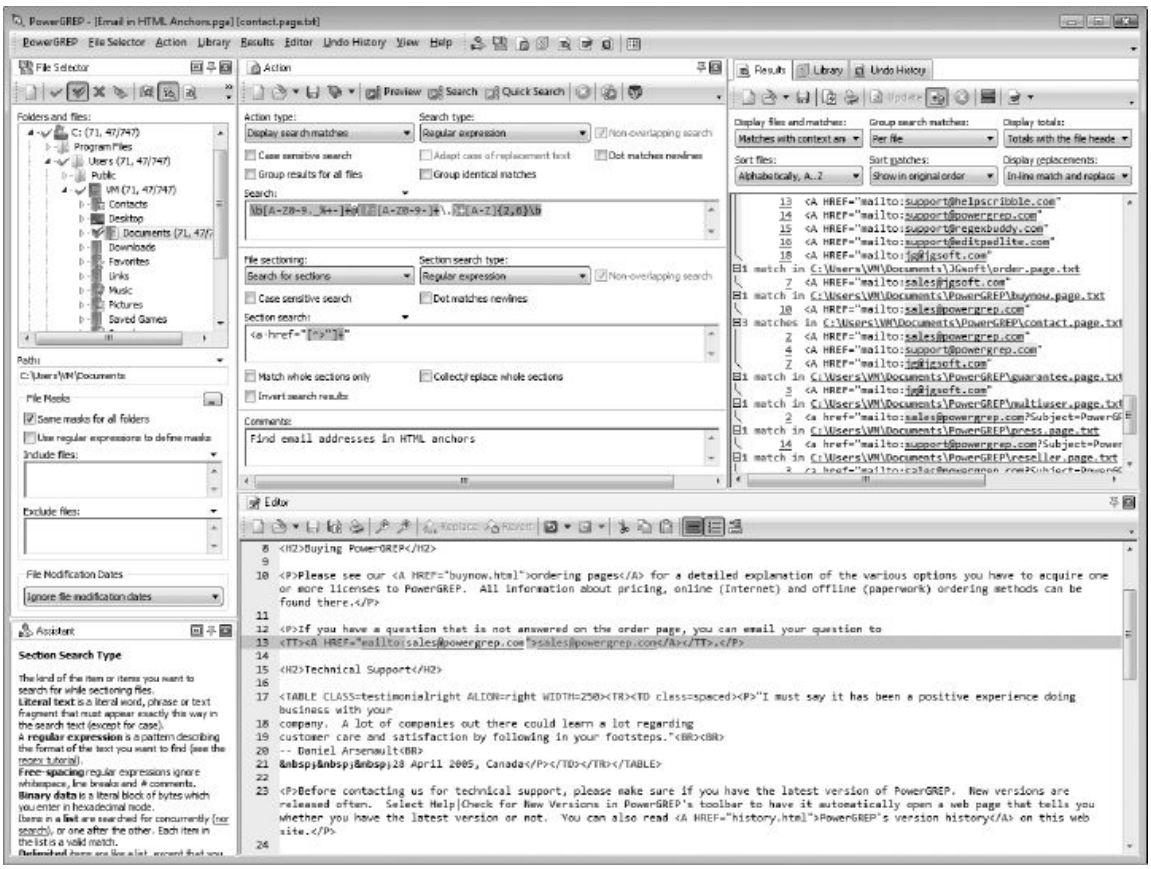


Figura 1.10 – PowerGREP.

Para executar uma busca rápida de expressão regular, basta

selecionar Clear (Limpar), no menu Action (Ação), e digitar sua expressão regular na caixa de busca no painel Action. Clique em uma pasta no painel File Selector (Seletor de Arquivos), e selecione Include File or Folder (Incluir Arquivo ou Pasta) ou Include Folder and Subfolders (Incluir Pastas e Subpastas) no menu File Selector. Em seguida, selecione Execute (Executar), no menu Action, para executar sua pesquisa.

Para executar uma pesquisa-e-substituição, selecione search-and-replace (pesquisar e substituir) na lista suspensa action type (tipo de ação), no canto superior esquerdo do painel Action, depois de limpar a ação. A caixa Replace (Substituir) aparecerá abaixo da caixa Search (Pesquisa). Digite seu texto de substituição. As demais etapas são semelhantes à pesquisa.

O PowerGREP tem a capacidade única de utilizar até três listas de expressões regulares ao mesmo tempo, com qualquer número de expressões regulares em cada lista. Enquanto os dois últimos parágrafos fornecem tudo de que você precisa para executar pesquisas simples, como em qualquer ferramenta grep, liberar todo o potencial do PowerGREP pode exigir a leitura da documentação detalhada da ferramenta.

PowerGREP roda em Windows 98, ME, 2000, XP e Vista. Você pode baixar uma cópia de avaliação gratuita em *http://www.powergrep.com/PowerGREPCookbook.exe*. Com exceção da gravação de resultados e bibliotecas, a cópia é totalmente funcional durante 15 dias de uso efetivo. Embora a cópia não salve os resultados mostrados no painel Results, ela modificará todos os seus arquivos durante as ações de pesquisa-e-substituição, tal como ocorre na versão completa.

Windows Grep

Windows Grep (*http://www.wingrep.com*) é uma das ferramentas grep mais antigas para Windows. Sua idade aparece um pouco na interface de usuário (Figura 1.11), mas ele faz bem o que se propõe

a fazer. Ele suporta um sabor de expressão regular limitado, chamado POSIX ERE. No caso das funcionalidades suportadas, ele utiliza a mesma sintaxe que os sabores deste livro. Windows Grep é shareware, o que significa que você pode baixá-lo gratuitamente, mas o pagamento é esperado, caso queira mantê-lo em seu computador.

Para preparar uma pesquisa, selecione Search no menu Search. A tela que aparece pode ser diferente, dependendo do que você selecionou; Beginner Mode (Modo Iniciante) ou Expert Mode (Modo Especialista), no menu Options. Os iniciantes obtêm um passo-a-passo do assistente, enquanto os especialistas obtêm um diálogo com guias.

Ao configurar uma pesquisa, o Windows Grep imediatamente a executa, presenteando-lhe com uma lista de arquivos em que foram encontradas correspondências. Clique uma vez em um arquivo para ver suas correspondências no painel inferior, e dê um duplo clique para abrir o arquivo. Selecione “All Matches” (Todas as Correspondências) no menu View para mostrar tudo.

Para executar uma pesquisa-e-substituição, selecione Replace no menu Search.

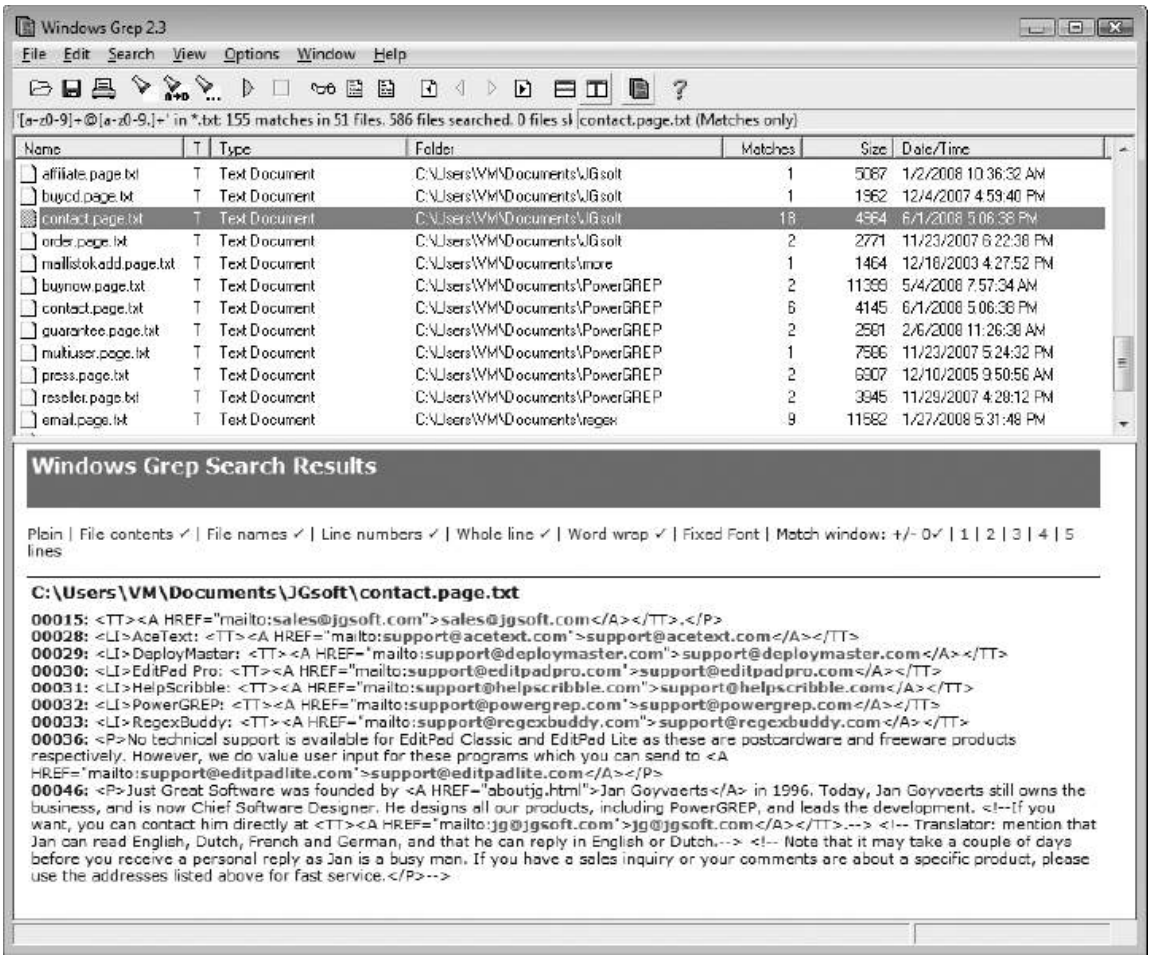


Figura 1.11 – Windows Grep.

RegexRenamer

RegexRenamer (Figura 1.12) não é realmente uma ferramenta grep. Ao invés de pesquisar o conteúdo de arquivos, ele pesquisa e substitui nomes de arquivos. Você pode baixá-lo em *http://regexrenamer.sourceforge.net*. RegexRenamer requer a versão 2.0 ou posterior do Microsoft .NET Framework.

Digite sua expressão regular na caixa Match, e o texto de substituição na caixa Replace. Clique em /i para ativar a não-diferenciação de maiúsculas e minúsculas, e em /g para substituir todas as correspondências encontradas em cada arquivo, em vez de substituir apenas a primeira. /x ativa a sintaxe de espaçamento livre, que não é muito útil, pois você tem apenas uma linha para digitar sua expressão regular.

Use a árvore do lado esquerdo para selecionar a pasta que contém os arquivos que deseja renomear. Você pode definir uma máscara de arquivo ou um filtro de expressão regular no canto superior direito. Isso restringirá a lista de arquivos nos quais sua expressão regular de procura-e-substituição será aplicada. Utilizar uma expressão regular para filtrar e outra para substituir é um procedimento muito mais prático do que tentar executar as duas funções com apenas uma regex.

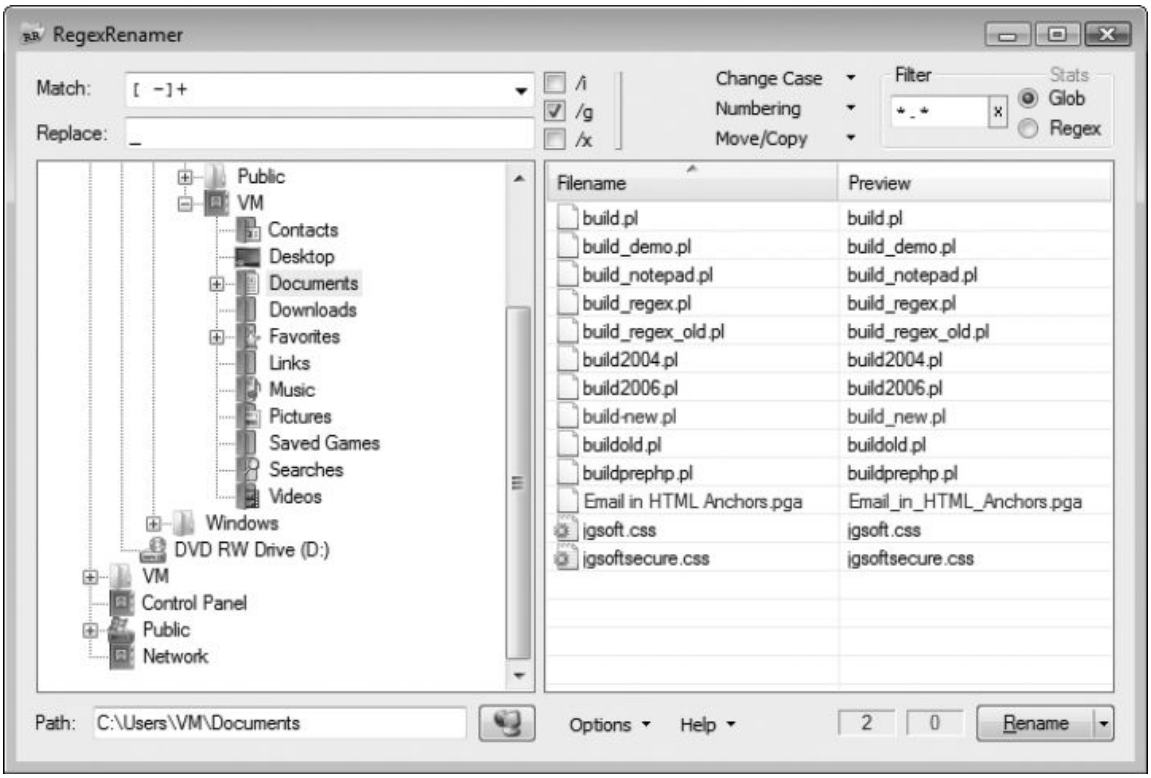


Figura 1.12 – RegexRenamer.

Editores de texto famosos

A maioria dos editores de texto modernos tem pelo menos um suporte básico a expressões regulares. No painel search (pesquisa) ou search-and-replace (pesquisar e substituir), normalmente você encontra uma caixa de seleção para ativar o modo de expressão regular. Alguns editores, como o EditPad Pro, também utilizam expressões regulares em várias funcionalidades de processamento de texto, tais como o realce de sintaxe ou de classes, e as listas de

função. A documentação de cada editor explica todas essas funcionalidades. Editores de texto conhecidos, com suporte a expressões regulares, incluem:

- Boxer Text Editor (PCRE)
- Dreamweaver (JavaScript)
- EditPad Pro (sabor personalizado que combina o melhor dos sabores discutidos neste livro, chamado “JGsoft” em RegexBuddy)
- Multi-Edit (PCRE, se você selecionar a opção “Perl”)
- NoteTab (PCRE)
- UltraEdit (PCRE)
- TextMate (Ruby 1.9 [Oniguruma])

---

1 Jeffrey Friedl narra a história desta citação em seu blog em *http://regex.info/blog/2006-09-15/247.*

2 N.T.: Trocadilho com a palavra regulator, que pode ser entendida como uma válvula de regulagem.

CAPÍTULO 2

Habilidades básicas de expressões regulares

Os problemas apresentados neste capítulo não são os do tipo que seu chefe ou seus clientes lhe pedem para resolver. São problemas técnicos que você vai encontrar durante a criação e edição de expressões regulares para a solução de problemas reais. A primeira receita, por exemplo, explica como corresponder um texto literal com uma expressão regular. Não se trata de um objetivo em si, porque você não vai precisar de uma regex, quando tudo que deseja fazer é uma busca por um texto literal. Porém, ao criar uma expressão regular, é bem provável que você precise dela para corresponder a um determinado texto, literalmente, e você precisa saber quais caracteres devem ser escapados. A receita 2.1 diz como fazer isso.

As receitas começam com várias técnicas básicas de expressões regulares. Se você já utilizou expressões regulares antes, provavelmente poderá passar rapidamente por elas, ou mesmo ignorá-las. As receitas, mais adiante neste capítulo, certamente lhe ensinarão algo de novo, a menos que você já tenha lido *Mastering Regular Expressions*, escrito por Jeffrey EF Friedl (O'Reilly), de capa a capa.

Nós planejamos as receitas neste capítulo de tal forma que cada uma delas explica um aspecto da sintaxe da expressão regular. Juntas, elas formam um tutorial abrangente para as expressões regulares. Leia do começo ao fim, para obter um bom domínio do conceito das expressões regulares. Ou mergulhe diretamente no mundo real das expressões regulares, nos capítulos 4 a 8, e volte

para as referências deste capítulo, sempre que aquelas receitas usarem alguma sintaxe com a qual não esteja familiarizado.

Este tutorial trata de expressões regulares apenas, e ignora completamente quaisquer considerações de programação. O próximo capítulo mostrará todas as listagens de código. Você pode conferir mais à frente, no tópico “Linguagens de programação e sabores Regex” do capítulo 3, e descobrir qual tipo de expressão regular sua linguagem de programação utiliza. Os sabores em si, que são o objetivo deste capítulo, foram introduzidos em “Sabores regex abordados neste livro”.

2.1 Corresponder a um texto literal

Problema

Criar uma expressão regular para coincidir exatamente com esta frase artificial:

> The punctuation characters in the ASCII table are: !"#$%&'()*+,-./:;<=>?
> @[\]^_`{|}~

Solução

> The␣punctuation␣characters␣in␣the␣ASCII␣table␣are:␣↲
> !"#\$%&'\(\)*\+,-\./:;<=>\?@\[\\]\^_`\{\|}~
>
> **Opções Regex**: Nenhuma
> **Sabores Regex**:. NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Qualquer expressão regular que não inclua nenhum dos 12 caracteres `$()*+.?[\^{|` simplesmente corresponde a si mesma. Para descobrir se “Mary had a little lamb” aparece no texto que você está editando, basta pesquisar por ‹Mary␣had␣a␣little␣lamb›. Não importa se a caixa de seleção de “expressão regular” está ativada em seu editor de texto.

Os 12 caracteres de pontuação, que fazem a magia das expressões regulares, são chamados de *metacaracteres*. Se você quiser que

sua regex corresponda a eles, literalmente, precisará aplicar um *escape*, colocando uma barra invertida na frente deles. Assim, a regex:

```
\$\(\)\*\+\.\?\[\\\^\{\|
```

corresponde ao texto:

```
$()*+.?[\^{|
```

Notadamente ausentes da lista estão o colchete de fechamento], o hífen -, e a chave de fechamento }. Os dois primeiros só se tornam metacaracteres após um colchete de abertura [não-escapado, e a chave de fechamento } somente após a ocorrência de uma chave de abertura { não-escapada. Não há necessidade de sempre escapar o caractere }. Regras de metacaracteres para os blocos que aparecem entre [e] são explicados na receita 2.3.

Inserir qualquer outro caractere não-alfanumérico não irá alterar o funcionamento da sua expressão regular pelo menos não enquanto se trabalhar com qualquer um dos sabores discutidos neste livro. Escapar um caractere alfanumérico dá a ele um significado especial, ou gera um erro de sintaxe.

Os iniciantes em expressões regulares, normalmente, escapam todos os caracteres de pontuação que surgem. Não deixe que ninguém saiba que você é um novato. Insira caracteres escapados criteriosamente. A selva de barras invertidas desnecessárias torna as expressões regulares difíceis de serem lidas, especialmente quando todas as barras invertidas precisam ser duplicadas para citar a regex como uma string literal dentro do código-fonte.

Variações

Escape em bloco

```
The punctuation characters in the ASCII table are: ↲
\Q!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~\E
```

Opções Regex: Nenhuma
Sabores Regex: Java 6, PCRE, Perl

Perl, PCRE e Java suportam os símbolos regex ‹\Q› e ‹\E›. ‹\Q›

suprime o significado de todos os metacaracteres, incluindo a barra invertida, até o ‹\E›. Se você omitir o ‹\E›, todos os caracteres após o ‹\Q› até o final da regex serão tratados como literais.

A única vantagem do ‹\Q...\E› é o fato de ser mais fácil de ler do que ‹\.\.\.›.

Embora Java 4 e 5 suportem esta função, você não deve usá-la. Bugs na implementação fazem com que expressões regulares com ‹\Q...\E› correspondam a coisas diferentes, em relação ao que você esperaria, e ao que o PCRE, Perl, ou Java 6 corresponderiam. Estes bugs foram corrigidos no Java 6, fazendo com que se comportem como no PCRE e Perl.

Correspondência sem diferenciação entre maiúsculas e minúsculas

ascii

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

(?i)ascii

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Por padrão, as expressões regulares diferenciam maiúsculas de minúsculas. ‹regex› corresponde a regex, mas não a Regex, REGEX ou ReGeX. Para fazer ‹regex› corresponder a todos eles, você precisa ativar a não-diferenciação de maiúsculas e minúsculas.

Na maioria das aplicações, é uma simples questão de marcar ou não uma caixa de seleção. Todas as linguagens de programação discutidas no próximo capítulo possuem uma sinalização ou uma propriedade que você pode definir para tornar sua regex insensível aos caracteres maiúsculos e minúsculos. A receita 3.4, no próximo capítulo, explica como aplicar em seu código-fonte as opções de regex listadas em cada solução de expressão regular deste livro.

Se você não puder ativar a insensibilidade fora da regex, você poderá fazê-lo internamente, utilizando o modificador de modo ‹(?i)›, como em ‹(?i)regex›. Isso funciona com os sabores .NET, Java, PCRE, Perl, Python e Ruby.

.NET, Java, PCRE, Perl e Ruby suportam modificadores de modo

locais, que afetam apenas uma parte da expressão regular. ‹sensitive(?i)caseless(?-i)sensitive› corresponde a sensitiveCASELESSsensitive, mas não a SENSITIVEcaselessSENSITIVE. ‹(?i)› ativa a insensibilidade para o resto da expressão, e ‹(?-i)› a desativa para o resto da expressão. Eles atuam como interruptores.

A receita 2.10 mostra como usar modificadores de modo local com grupos, em vez de interruptores.

Veja também:

Receitas 2.3 e 5.14.

2.2 Corresponder a caracteres não-imprimíveis

Problema

Corresponder a uma string com os seguintes caracteres ASCII de controle: sino, escape, form feed, line feed, carriage return, tabulação horizontal, tabulação vertical. Estes caracteres possuem os códigos ASCII hexadecimais 07, 1B, 0C, 0A, 0D, 09, 0B.

Solução

\a\e\f\n\r\t\v

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, PCRE, Python, Ruby

\x07\x1B\f\n\r\t\v

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, JavaScript, PCRE, Python, Ruby

\a\e\f\n\r\t\0x0B

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Discussão

Sete dos caracteres ASCII de controle mais comumente usados

possuem sequências de escape dedicadas. Todas consistem em uma barra invertida seguida por uma letra. Esta é a mesma sintaxe usada para literais de strings, em muitas linguagens de programação. A tabela 2.1 mostra os caracteres não-imprimíveis mais comuns, e como eles estão representados.

Tabela 2.1 – Caracteres não-imprimíveis

Representação	Significado	Representação hexadecimal
‹\a›	*sino*	*0x07*
‹\e›	*escape*	*0x1B*
‹\f›	*alimentação de página (form feed)*	*0x0C*
‹\n›	*alimentação de linha (line feed ou newline)*	*0x0A*
‹\r›	*retorno de carro (carriage return)*	*0x0D*
‹\t›	*tabulação horizontal*	*0x09*
‹\v›	*tabulação vertical*	*0x0B*

Perl não suporta \v, então temos de usar uma sintaxe diferente para a tabulação vertical.

O padrão ECMA-262 não suporta ‹\a› e ‹\e›. Portanto, usaremos, neste livro, uma sintaxe diferente para os exemplos de JavaScript, apesar de que muitos navegadores não suportam ‹\a› e ‹\e›.

Estes caracteres de controle, bem como sua sintaxe alternativa indicada na seção seguinte, podem ser usados igualmente dentro e fora de classes de caracteres em sua expressão regular.

Variações das representações de caracteres não-imprimíveis

Os 26 caracteres de controle

\cG\x1B\cL\cJ\cM\cI\cK

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Ruby 1.9

Utilizando de ‹\cA› a ‹\cZ›, você pode corresponder a um dos 26 caracteres de controle que ocupam as posições de 1 a 26 na tabela

ASCII. O c deve estar em letra minúscula. A letra que se segue ao c não diferencia maiúsculo de minúsculo, na maioria dos sabores. Recomendamos que você use sempre uma letra maiúscula. O Java exige isso.

Esta sintaxe pode ser útil se você estiver acostumado a digitar caracteres de controle em sistemas, pressionando a tecla Ctrl junto com uma letra. Em um terminal, Ctrl-H envia um backspace. Em uma regex, ‹\cH› corresponde ao backspace.

Python e o mecanismo clássico do Ruby, no Ruby 1.8, não suportam esta sintaxe. O mecanismo Oniguruma, no Ruby 1.9, suporta essa sintaxe.

O caractere de controle de escape, na posição 27 da tabela ASCII, está além do alcance do alfabeto em inglês; assim, vamos deixá-lo como ‹\x1B› em nossa expressão regular.

O conjunto de caracteres de 7 bits

\x07\x1B\x0C\x0A\x0D\x09\x0B

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Um \x em letra minúscula, seguido por dois dígitos hexadecimais em letras maiúsculas correspondem a um caractere específico no conjunto ASCII. A figura 2.1 mostra quais combinações hexadecimais, de ‹\x00› a ‹\x7F›, correspondem a qual caractere em todo o conjunto ASCII. A tabela é organizada com o primeiro dígito hexadecimal crescendo de baixo para cima no lado esquerdo e o segundo dígito crescendo da esquerda para a direita correndo ao longo do topo da tabela.

	0	1	2	3	4	5	6	7	8	9	BA	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Figura 2.1 – Tabela ASCII.

A quais caracteres os escapes de ‹\x80› a ‹\xFF› correspondem, depende de como seu mecanismo de expressão regular os interpreta, e em qual página do código (code page) seu texto de amostra está codificado. Recomendamos que você não utilize de ‹\x80› a ‹\xFF›. Em vez disso, utilize o token de ponto de código Unicode, descrito na receita 2.7.

Se você estiver utilizando o Ruby 1.8, ou se compilou o PCRE sem suporte a UTF-8, não poderá utilizar pontos de código Unicode. Ruby 1.8 e PCRE sem UTF-8 são mecanismos regex de 8 bits. Eles são totalmente ignorantes a respeito de codificações de textos e caracteres multibyte. ‹\xAA›, nestes mecanismos, simplesmente corresponde ao byte 0xAA, independentemente de qual caractere 0xAA represente, ou se 0xAA faz parte de um caractere multibyte.

Veja também:

Receita 2.7.

2.3 Corresponder a um dentre vários caracteres

Problema

Crie uma expressão regular que corresponda aos erros ortográficos mais comuns da palavra calendar, para que você possa encontrar

essa palavra em um documento sem ter de confiar na habilidade de ortografia do autor. Permita que um a ou um e seja utilizado em cada uma das posições das vogais. Crie outra expressão regular para corresponder a um único caractere hexadecimal. Crie uma terceira regex para corresponder a um caractere simples que não seja um caractere hexadecimal.

Solução

Palavra calendar com erros ortográficos

```
c[ae]l[ae]nd[ae]r
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Caractere hexadecimal

```
[a-fA-F0-9]
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Caractere não-hexadecimal

```
[^a-fA-F0-9]
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

A notação com colchetes é chamada de *classe de caracteres*. Uma classe de caracteres corresponde a um único caractere, dentre uma lista de caracteres possíveis. As três classes na primeira expressão regular correspondem tanto a um a, quanto a um e. Elas fazem isso de forma independente. Quando você testa a palavra calendar contra essa expressão regular, a primeira classe de caracteres corresponde a um a, a segunda a um e, e a terceira a um a.

Fora das classes de caracteres, doze dos caracteres de pontuação são metacaracteres. Dentro de uma classe de caracteres, somente

quatro possuem função especial: \, ^, -, e]. Se você estiver utilizando Java, ou .NET, o colchete de abertura [também é um metacaractere, dentro das classes de caracteres. Todos os outros são caracteres literais, e são simplesmente incluídos à classe de caracteres. A expressão regular ‹[$()*+.?{|]› corresponde a qualquer um dos nove caracteres entre os colchetes.

A barra invertida sempre escapa o caractere que o segue, assim como ela faz fora das classes de caracteres. O caractere escapado pode ser um caractere único, o início ou o final de um intervalo. Os outros quatro metacaracteres adquirem seus significados especiais apenas quando colocados em determinadas posições. É possível incluí-los como caracteres literais em uma classe de caracteres sem escapá-los, posicionando-os de modo que não adquiram seus significados especiais. ‹[][^-]› realiza esse truque, pelo menos se você não estiver utilizando uma implementação JavaScript que siga estritamente o padrão. Mas recomendamos que você sempre escape estes metacaracteres; assim, a regex anterior deveria ser ‹[\]\[\^\-]›. Escapar os metacaracteres faz com que sua expressão regular fique mais fácil de ser entendida.

Caracteres alfanuméricos não podem ser escapados com uma barra invertida. Ao fazer isso, ele gera um erro ou cria um token de expressão regular (alguma coisa com um significado especial na expressão regular). Em nossas discussões sobre certos tokens de expressão regular, como na receita 2.2, mencionamos que eles podem ser usados dentro das classes de caracteres. Todos esses tokens consistem em uma barra invertida e uma letra, algumas vezes seguidos por alguns outros caracteres. Assim, ‹[\r\n]› corresponde a um retorno de carro (\r), ou uma quebra de linha (\n).

O *circunflexo* (^), se colocado imediatamente após o colchete de abertura, nega a classe de caracteres. Ele faz a classe de caracteres corresponder a qualquer caractere que *não* esteja na lista. Uma classe de caracteres negada corresponde a caracteres de quebra de linha, a menos que você os adicione à classe de

caracteres de negação.

O *hífen* (-) cria um *intervalo*, quando colocado entre dois caracteres. O intervalo consiste em uma classe de caracteres que inclui o caractere antes do hífen, o caractere após o hífen e todos os caracteres que ficam entre eles, em ordem numérica. Para saber quais são estes caracteres, você precisa ver a tabela de caracteres ASCII ou Unicode. ‹[A-z]› inclui todos os caracteres na tabela ASCII, entre o A maiúsculo e o z minúsculo. O intervalo inclui algumas pontuações. Então ‹[A-Z\[\\\]\^_`a-z]› corresponde mais explicitamente aos mesmos caracteres. Recomendamos que você crie intervalos somente entre dois dígitos, ou entre duas letras que sejam, ambas, maiúsculas ou minúsculas.

Intervalos invertidos, como ‹[z-a]›, não são permitidos.

Variações

Abreviações

[a-fA-F\d]

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Seis tokens regex que consistem em uma barra invertida e uma letra formam classes de caracteres *abreviadas*. Você pode utilizá-los dentro e fora das classes de caracteres. ‹\d› e ‹[\d]› correspondem a um único dígito. Cada caractere abreviado minúsculo tem um caractere abreviado maiúsculo associado, com o sentido oposto. Assim, ‹\D› corresponde a qualquer caractere que *não* seja um dígito, o que é equivalente a ‹[^\d]›.

‹\w› corresponde a um único *caractere de palavra*. Um caractere de palavra é um caractere que pode ocorrer como parte de uma palavra. Isso inclui letras, números e o caractere _ (underscore). A escolha específica dos caracteres, aqui, pode parecer meio estranha, mas foi feita porque estes são os caracteres que normalmente são permitidos em identificadores nas linguagens de

programação. ‹\W› corresponde a qualquer caractere que não faça parte de uma palavra muito técnica.

Em Java, JavaScript, PCRE, e Ruby, ‹\w› é sempre idêntico a ‹[a-zA-Z0-9_]›. Em .NET e Perl, ele inclui letras e números de todos os outros alfabetos (Cirílico, Tailandês etc.). Em Python, os outros alfabetos são incluídos somente se você usar a flag UNICODE ou U ao criar a regex. ‹\d› segue a mesma regra em todos estes sabores. Em .NET e Perl, dígitos de outros alfabetos são sempre incluídos, enquanto o Python os inclui se você passar a flag UNICODE ou U.

‹\s› corresponde a qualquer *caractere de espaço em branco*. Isso inclui espaços, tabulações e quebras de linhas. No .NET, Perl e JavaScript, ‹\s› também corresponde a qualquer caractere definido como espaço em branco pelo padrão Unicode. Note que o JavaScript utiliza Unicode no caso de ‹\s›, mas ASCII para ‹\d› e ‹\w›. ‹\S› corresponde a qualquer caractere que não corresponda a ‹\s›.

Mais inconsistências surgem quando acrescentamos ‹\b› à mistura. ‹\b› não é uma classe de caracteres abreviados, mas um limite de palavra. Embora você possa esperar que ‹\b› suporte Unicode, quando ‹\w› o suporta, e ser somente ASCII quando ‹\w› for somente ASCII, este nem sempre é o caso. A subseção “Caracteres de palavras”, na receita 2.6, mostra os detalhes.

Não-diferenciação entre maiúsculas e minúsculas

(?i)[A-F0-9]

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

(?i)[^A-F0-9]

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

A não-diferenciação entre maiúsculas e minúsculas, seja ela definida por uma flag externa (receita 3.4) ou por um modificador de modo dentro da regex (veja a receita 2.1), também afeta as classes

de caracteres. As duas regexes mostradas aqui são equivalentes às da solução original.

O JavaScript segue a mesma regra, mas não suporta ‹(?i)›. Para fazer uma expressão regular não diferenciar caracteres maiúsculos e minúsculos, defina a flag /i ao criá-la.

Funcionalidades específicas de cada sabor

Subtração de classe de caracteres no .NET

```
[a-zA-Z0-9-[g-zG-Z]]
```

Esta expressão regular corresponde a um único caractere hexadecimal, mas de forma indireta. A classe-base de caracteres corresponde a qualquer caractere alfanumérico; então, uma classe aninhada subtrai as letras de g a z. Esta classe aninhada deve aparecer no final da classe-base, precedida por um hífen: ‹[class-[subtract]]›.

A *subtração* de classe de caracteres é particularmente útil ao trabalharmos com propriedades Unicode, blocos e scripts. Como exemplo, ‹\p{IsThai}› corresponde a qualquer caractere no bloco Tailandês. ‹\P{N}› corresponde a qualquer caractere que não tenha a propriedade Number. Combiná-los com a subtração ‹[\p{IsThai}-[\P{N}]]› os fará corresponder a qualquer um dos 10 dígitos em Tailandês.

União, subtração e intersecção de classes de caracteres no Java

```
[a-f[A-F][0-9]]
[a-f[A-F[0-9]]]
```

O Java permite que uma classe de caracteres seja aninhada a outra. Se a classe aninhada for incluída diretamente, a classe resultante será a *união* das duas. Você pode aninhar quantas classes quiser. Ambas as regexes mostradas têm o mesmo efeito da expressão regular original, sem os colchetes extras.

```
[\w&&[a-fA-F0-9\s]]
```

Esta expressão poderia ganhar um prêmio em um concurso de expressões regulares obscuras. A classe-base de caracteres corresponde a qualquer caractere de palavra. A classe aninhada corresponde a qualquer dígito hexadecimal e a qualquer espaço em branco. A classe resultante é a *intersecção* das duas, correspondendo a dígitos hexadecimais e nada mais. Como a classe-base não corresponde a espaços em branco e a classe aninhada não corresponde a ‹[g-zG-Z_]›, tais caracteres são descartados da classe de caracteres final, deixando apenas os dígitos hexadecimais:

```
[a-zA-Z0-9&&[^g-zG-Z]]
```

Esta expressão regular corresponde a um único caractere hexadecimal, também de forma indireta. A classe-base de caracteres corresponde a qualquer caractere alfanumérico e, então, uma classe aninhada subtrai as letras de g a z. Essa classe aninhada deve ser uma classe de negação de caracteres, precedida por dois caracteres &: ‹[class&&[^subtract]]›.

A intersecção e a subtração de classes de caracteres são particularmente úteis quando trabalhamos com propriedades Unicode, blocos e scripts. Assim, ‹\p{InThai}› corresponde a qualquer caractere no bloco Tailandês, considerando que ‹\p{N}› corresponde a qualquer caractere que contenha a propriedade Number. Em consequência, ‹[\p{InThai}&&[\p{N}]]› corresponde a qualquer um dos 10 dígitos tailandeses.

Se você estiver pensando a respeito das diferenças sutis nos tokens ‹\p› da regex, encontrará uma explicação completa na receita 2.7.

Veja também:

Receitas 2.1, 2.2 e 2.7.

2.4 Corresponder a qualquer

caractere

Problema

Corresponda a um caractere entre aspas. Forneça uma solução que permita qualquer caractere único, exceto uma quebra de linha, entre as aspas. Forneça outra solução que realmente permita qualquer caractere, incluindo quebras de linha.

Solução

Qualquer caractere, exceto quebras de linhas

'.'

Opções Regex: Nenhuma (a opção “ponto corresponde a quebras de linha” não deve estar ativada).

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Qualquer caractere, incluindo quebras de linhas

'.'

Opções Regex: Ponto corresponde a quebras de linhas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

'[\s\S]'

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Discussão

Qualquer caractere, exceto quebras de linhas

O ponto é uma das funcionalidades mais simples e antigas das expressões regulares. Seu significado sempre foi o de corresponder a qualquer caractere único.

Há, no entanto, alguma confusão a respeito do que *qualquer caractere* verdadeiramente significa. As ferramentas mais antigas, para trabalhar com arquivos de expressões regulares, processavam linha por linha, de modo que nunca havia oportunidade para que o

texto de assunto incluísse uma quebra de linha. As linguagens de programação discutidas neste livro processam o texto de assunto como um todo, não importa quantas quebras de linha você tenha inserido. Se quiser um verdadeiro processamento linha por linha, terá de escrever um pouco de código que divida o assunto em um array de linhas, e aplicar a regex em cada linha do array. A receita 3.21, no próximo capítulo, mostra como fazer isso.

Larry Wall, o criador do Perl, queria que a linguagem mantivesse o comportamento tradicional das ferramentas baseadas em linhas, na qual o ponto nunca correspondia a uma quebra de linha (\n). Todos os outros sabores discutidos neste livro seguiram seu exemplo. Assim, ‹.› corresponde a qualquer caractere único, *exceto* o caractere de nova linha (newline).

Qualquer caractere, incluindo quebras de linhas

Se você realmente deseja permitir que sua expressão regular se estenda a múltiplas linhas, ative a opção “ponto corresponde a quebras de linhas” (dot matches line breaks). Esta opção esconde-se por trás de diferentes nomes. Perl e muitos outros a chamam, erroneamente, de “linha simples” (single line), enquanto o Java a chama de “modo ponto tudo” (dot all). A receita 3.4, no próximo capítulo, tem todos os detalhes. Seja qual for o nome desta opção, em sua linguagem de programação favorita, pense nela como o modo “ponto corresponde a quebras de linha“. É tudo o que esta opção faz.

Uma solução alternativa torna-se necessária para o JavaScript, que não tem uma opção “ponto corresponde a quebras de linha”. Como explica a receita 2.3, ‹\s› corresponde a qualquer caractere de espaço em branco, enquanto ‹\S› corresponde a qualquer caractere que não tenha sido correspondido com ‹\s›. A combinação ‹[\s\S]› resultará em uma classe de caracteres que inclui todos os caracteres, até as quebras de linha. ‹[\d\D]› e ‹[\w\W]› têm o mesmo efeito.

Abuso no uso de pontos

O ponto é a funcionalidade mais abusada das expressões regulares. ‹\d\d.\d\d.\d\d› não é uma boa maneira de se corresponder a uma data. Ela corresponde muito bem a 05/16/08, mas também a 99/99/99. Pior, ela corresponde a 12345678.

Uma regex adequada, que corresponda apenas a datas válidas, é um assunto para um capítulo posterior. Mas, substituir o ponto por uma classe de caracteres mais adequada é muito fácil. ‹\d\d[/.\-]\d\d[/.\-]\d\d› permite que uma barra, um ponto ou um hífen seja usado como separador de data. Essa regex ainda corresponde a 99/99/99, mas não a 12345678.

É apenas uma coincidência que o exemplo anterior inclua um ponto dentro das classes de caracteres. Dentro de uma classe de caracteres, o ponto é apenas um caractere literal. Vale a pena incluí-lo nesta expressão regular em particular, porque em alguns países, como a Alemanha, ele é usado como separador de datas.

Use o ponto somente quando você realmente quiser permitir qualquer caractere. Use uma classe de caracteres ou uma classe de caracteres negada em qualquer outra situação.

Variações

(?s)'.'

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Python

(?m)'.'

Opções Regex: Nenhuma

Sabor Regex: Ruby

Se não puder ativar o modo “ponto corresponde a quebras de linha” fora da expressão regular, você pode inserir um modificador de modo em seu início. Explicamos o conceito dos modificadores de modo, e da falta de suporte a eles no JavaScript, na subseção “Correspondência sem diferenciação entre maiúsculas e minúsculas”, na receita 2.1.

‹(?s)› é o modificador para o modo “ponto corresponde a quebras de linha” no .NET, Java, PCRE, Perl e Python. O s deriva do termo

“single line” (linha única), o nome confuso do Perl para “ponto corresponde a quebras de linha”.

A terminologia é tão confusa que o desenvolvedor do mecanismo de expressões regulares do Ruby a copiou erroneamente. O Ruby utiliza ‹(?m)› para ativar o modo “ponto corresponde a quebras de linha”. Apesar da letra ser diferente, a funcionalidade é exatamente a mesma. O novo mecanismo do Ruby 1.9 continua a utilizar ‹(?m)› para “ponto corresponde a quebras de linha”. O significado de ‹(?m)› no Perl é explicado na receita 2.5.

Veja também:

Receitas 2.3, 3.4 e 3.21.

2.5 Corresponder a alguma coisa no começo e/ou final de uma linha

Problema

Crie quatro expressões regulares. Corresponda à palavra alpha, mas somente se ela aparecer no início do texto de assunto. Corresponda à palavra omega, mas somente se ela aparecer no final do texto de assunto. Corresponda à palavra begin, mas somente se aparecer no início de uma linha. Corresponda à palavra end, mas somente se aparecer no final de uma linha.

Solução

Início do assunto

^alpha

Opções Regex: Nenhuma (“^ e $ correspondem em quebras de linha” não deve estar ativado)

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

\Aalpha

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Final do assunto

omega$

Opções Regex: Nenhuma (“^ e $ correspondem em quebras de linha” não deve estar ativado)

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

omega\Z

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Início de uma linha

^begin

Opções Regex: ^ e $ correspondem em quebras de linhas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Fim de uma linha

end$

Opções Regex: ^ e $ correspondem em quebras de linhas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Âncoras e linhas

Os tokens de expressão regular ‹^›, ‹$›, ‹\A›, ‹\Z›, e ‹\z› são chamados de *âncoras*. Eles não correspondem a nenhum caractere. Em vez disso, correspondem a determinadas posições, ancorando efetivamente a correspondência da expressão regular nessas posições.

Uma *linha* é a parte do texto de assunto que se situa entre o início do assunto e uma quebra de linha, entre duas quebras de linha, ou entre uma quebra de linha e o final do assunto. Se não existirem quebras de linha no assunto, então todo o assunto será considerado como sendo uma linha. Assim, o texto a seguir consiste em quatro

linhas: o texto one, o texto two, uma string vazia, e o texto four:

```
um
dois

quatro
```

O texto poderia ser representado em um programa como one **LF** two **LF LF** four.

Início do assunto

A âncora ‹\A› sempre corresponde ao início do texto de assunto, antes do primeiro caractere. Este é o único local onde ela corresponde. Insira ‹\A› no início de sua expressão regular para testar se o texto de assunto inicia com o texto que você deseja corresponder. O “A” deve ser maiúsculo.

JavaScript não suporta ‹\A›.

A âncora ‹^› é equivalente a ‹\A›, contanto que você não ative a opção “^ e $ correspondem em quebras de linha”. Esta opção está desativada por padrão em todos os sabores de expressões regulares, exceto no Ruby. O Ruby não oferece uma maneira de desativar esta opção.

A menos que você esteja utilizando JavaScript, recomendamos que utilize sempre ‹\A›, ao invés de ‹^›. O significado de ‹\A› nunca muda, evitando qualquer confusão ou erro nas definições das opções da regex.

Final do assunto

As âncoras ‹\Z› e ‹\z› sempre correspondem ao final do texto de assunto, após o último caractere. Insira ‹\Z› ou ‹\z› no final de sua expressão regular para testar se o texto de assunto termina com o texto com ao qual você deseja corresponder.

.NET, Java, PCRE e Ruby suportam tanto ‹\Z›, quanto ‹\z›. Python suporta somente ‹\Z›. JavaScript não suporta nem ‹\Z›, nem ‹\z›.

A diferença entre ‹\Z› e ‹\z› entra em cena quando o último caractere

em seu texto de assunto for uma quebra de linha. Nesse caso, ‹\Z› pode corresponder ao final do texto de assunto, após a quebra de linha final, bem como imediatamente antes desta quebra de linha. A vantagem disso é que você pode pesquisar por ‹omega\Z›, sem ter que se preocupar em retirar a quebra de linha no final do texto de assunto. Ao ler um arquivo linha por linha, algumas ferramentas incluem a quebra de linha ao final, enquanto outras não; ‹\Z› mascara essa diferença. ‹\z› corresponderá somente ao final do texto de assunto, para que não haja outras correspondências no caso de ocorrer uma quebra de linha no final.

A âncora ‹$› é equivalente a ‹\Z›, desde que você não ative a opção "^ e $ correspondem em quebras de linha". Esta opção está desativada, por padrão, em todos os sabores de expressões regulares, exceto o Ruby. O Ruby não oferece uma maneira de desativar esta opção. Assim como ‹\Z›, ‹$› corresponde ao final do texto de assunto, bem como antes da quebra de linha final, se houver.

Para ajudar a esclarecer esta situação sutil e um tanto confusa, vamos ver um exemplo em Perl. Supondo que $/ (separador de registro atual) esteja definido com seu valor padrão \n, a declaração Perl a seguir lê uma única linha a partir do terminal (entrada padrão):

```
$line = <>;
```

Perl coloca a nova linha no conteúdo da variável $line. Portanto, uma expressão como ‹end of input.\z› não corresponderá à variável. Mas, tanto ‹end of input.\Z› quanto ‹end of input.$› corresponderão, pois elas ignoram o caractere de nova linha no final.

Para tornar o processamento mais fácil, muitas vezes os programadores Perl retiram os caracteres de nova linha com:

```
chomp $line;
```

Após efetuar esta operação, todas as três âncoras corresponderão. Tecnicamente, chomp remove o separador de registro atual da string.

A menos que você esteja utilizando JavaScript, recomendamos que

sempre utilize ‹\Z›, ao invés de ‹$›. O significado de ‹\Z› nunca muda, evitando qualquer confusão ou erro nas definições das opções da regex.

Início de uma linha

Por padrão, ‹^›‹^› corresponde somente ao início do texto de assunto, assim como ‹\A›. Somente no Ruby ‹^› sempre corresponde ao início de uma linha. Todos os outros sabores exigem que você ative a opção para fazer com que o acento circunflexo (^) e o cifrão ($) correspondam a quebras de linha. Esta opção é normalmente referida como modo “linhas múltiplas” (multiline).

Não confunda esse modo com o modo de “linha simples” (single line), mais conhecido como “ponto corresponde a quebras de linha”. O modo “linhas múltiplas” afeta somente o acento circunflexo (^) e o cifrão ($); o modo “linha simples” afeta somente o ponto, como explica a receita 2.4. É perfeitamente possível ativar os modos “linha simples” e “linhas múltiplas” ao mesmo tempo. Por padrão, ambas as opções estão desativadas.

Com a opção correta ajustada, ‹^› irá corresponder ao início de cada linha no texto de assunto. A rigor, ele corresponde antes do primeiro caractere do texto, como sempre faz, e também após cada caractere de quebra de linha no texto de assunto. O acento circunflexo em ‹\n^› é redundante, porque ‹^› sempre corresponderá após ‹\n›.

Fim de uma linha

Por padrão, ‹$› corresponderá somente ao final do texto de assunto, ou, tal como ‹\Z›, antes da quebra de linha final. Somente no Ruby, ‹$› sempre corresponde ao final de cada linha. Todos os outros sabores exigem que você ative a opção “linhas múltiplas” para fazer com que o acento circunflexo (^) e o cifrão ($) combinem em quebras de linha.

Com a opção correta ajustada, ‹$› corresponderá ao final de cada

linha no texto de assunto. Claro, também corresponderá após o último caractere no texto, pois se trata, também, do final de uma linha. O cifrão em ‹$\n› é redundante, porque ‹$› sempre corresponderá antes de ‹\n›.

Correspondências de comprimento zero

É perfeitamente válido que uma expressão regular não tenha nada além de uma ou mais âncoras. Essa expressão regular encontrará uma correspondência de comprimento zero em cada posição em que a âncora puder corresponder. Se você colocar várias âncoras juntas, todas precisarão corresponder à mesma posição, para que a regex gere uma correspondência.

Você poderia utilizar uma expressão regular desse tipo em uma pesquisa-e-substituição. Substitua ‹\A›, ou ‹\Z›, para preceder ou acrescentar alguma coisa ao assunto como um todo. Substitua ‹^› ou ‹$›, no modo "^ e $ correspondem em quebras de linha", para preceder ou acrescentar alguma coisa em cada linha do texto de assunto.

Combine duas âncoras para testar linhas em branco, ou entradas de texto faltantes. ‹\A\Z› corresponde à string vazia, assim como à string que consiste em um único caractere de nova linha. ‹\A\z› corresponde somente à string vazia. ‹^$› no modo "^ e $ correspondem em quebras de linha" corresponde a cada linha em branco no texto de assunto.

Variações

(?m)^begin

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Python

(?m)end$

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Python

Se não for possível ativar o modo "^ e $ correspondem em quebras

de linha” fora da expressão regular, você poderá inserir um modificador de modo em seu início. Explicamos o conceito dos modificadores de modo, e da falta de suporte para eles no JavaScript, na subseção “Correspondência sem diferenciação entre maiúsculas e minúsculas”, receita 2.1.

‹(?m)› é o modificador para o modo “ponto corresponde a quebras de linha” em .NET, Java, PCRE, Perl e Python. O m deriva de “linhas múltiplas” (multiple lines), nome confuso do Perl para “^ e $ correspondem em quebras de linha”.

Como explicado anteriormente, a terminologia é tão confusa que o desenvolvedor do mecanismo de expressão regular do Ruby a copiou erroneamente. O Ruby utiliza ‹(?m)› para ativar o modo “ponto corresponde a quebras de linha”. O ‹(?m)› do Ruby não tem nada a ver com as âncoras de acento circunflexo (^) e cifrão ($). No Ruby, ‹^› e ‹$› sempre correspondem ao início e ao final de cada linha.

Exceto pela infeliz mistura de letras, a escolha do Ruby por usar ‹^› e ‹$› exclusivamente para linhas é boa. A menos que esteja usando JavaScript, recomendamos que adote esta escolha em suas próprias expressões regulares.

Jan seguiu a mesma ideia em seus projetos EditPad Pro e PowerGREP. Você não vai encontrar uma caixa de seleção chamada “^ and $ match at line breaks” (^ e $ correspondem em quebras de linha), embora haja uma chamada “dot matches newlines” (ponto corresponde a caracteres de nova linha). A menos que você inicie sua expressão regular com ‹(?-m)›, terá de usar ‹\A› e ‹\Z› para ancorar sua regex, no início ou no final de seu arquivo.

Veja também:

Receitas 3.4 e 3.21.

2.6 Corresponder a palavras inteiras

Problema

Crie uma regex que corresponda a cat em My cat is brown, mas não em category ou bobcat.

Crie outra regex que corresponda a cat em staccato, mas não em qualquer uma das três strings anteriores.

Solução

Nas extremidades da palavra

\bcat\b

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Dentro da palavra

\Bcat\B

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Nas extremidades da palavra

O token ‹\b› da expressão regular é chamado de *extremidade de palavra*. Ele corresponde no início ou no final de uma palavra. Porém, sozinho, resulta em uma correspondência nula. ‹\b› é uma *âncora*, assim como os tokens introduzidos na seção anterior.

A rigor, ‹\b› corresponde nessas três posições:

- Antes do primeiro caractere no assunto, se o primeiro caractere for um caractere de palavra.
- Após o último caractere no assunto, se o último caractere for um caractere de palavra.
- Entre dois caracteres no assunto, quando um deles for um caractere de palavra, e o outro não.

Nenhum dos sabores discutidos neste livro tem tokens separados

para corresponder somente antes, ou somente depois de uma palavra. A menos que você queira criar uma regex que consista em nada além de uma extremidade de palavra, isso não é necessário. Os tokens, antes ou depois de ‹\b›, em sua expressão regular, determinarão onde ‹\b› poderá corresponder. O ‹\b› em ‹\bx›, e em ‹!\b›, poderia corresponder somente no início de uma palavra. O ‹\b›, em ‹x\b› e ‹\b!›, poderia corresponder somente no final de uma palavra. ‹x\bx› e ‹!\b!› nunca poderão corresponder, seja onde for.

Para realizar uma pesquisa do tipo “apenas palavras inteiras” utilizando uma expressão regular, simplesmente coloque a palavra entre dois tokens de extremidade de palavra, como fizemos em ‹\bcat\b›. O primeiro ‹\b› exige que ‹c› ocorra no início da string, ou após um pseudocaractere. O segundo ‹\b› exige que ‹t› ocorra no final da string, ou antes de um pseudocaractere.

Caracteres de quebras de linha são pseudocaracteres. ‹\b› corresponderá após uma quebra de linha se ela for imediatamente seguida por um caractere de palavra. Ele também corresponderá antes de uma quebra de linha imediatamente precedida por um caractere de palavra. Então, uma palavra que ocupe uma linha inteira será encontrada pela pesquisa “apenas palavras inteiras”. ‹\b› não é afetado pelo modo “linhas múltiplas”, ou ‹(?m)›, uma das razões deste livro referir-se ao modo “linhas múltiplas” como “^ e $ correspondem em quebras de linha”.

Dentro da palavra

‹\B› corresponde, em todas as posições, no texto de assunto em que ‹\b› não corresponde. ‹\B› corresponde em qualquer posição que não esteja no início ou no final de uma palavra.

A rigor, ‹\B› corresponde nessas cinco posições:

- Antes do primeiro caractere no assunto, se o primeiro caractere não for de palavra.
- Após o último caractere no assunto, se o último caractere não for de palavra.

- Entre dois caracteres de palavras.
- Entre dois pseudocaracteres.
- A string vazia.

‹\Bcat\B› corresponde a cat em staccato, mas não em My cat is brown, ou bobcat.

Para fazer o oposto de uma pesquisa do tipo “apenas palavras inteiras” (isto é, excluindo My cat is brown e incluindo staccato, category e bobcat), você precisará usar a alternância para combinar ‹\Bcat› e ‹cat\B› em ‹\Bcat|cat\B›. ‹\Bcat› corresponde a cat em staccato e bobcat. ‹cat\B› corresponde a cat em category (e staccato, se ‹\Bcat› já não cuidou disso). A receita 2.8 explica a alternância.

Caracteres de palavras

Falamos muito a respeito de extremidades de palavras, mas não falamos o que é um *caractere de palavra*. Um caractere de palavra pode ocorrer como parte de uma palavra. A subseção “Abreviações”,, na receita 2.3, discute quais caracteres estão incluídos em ‹\w›, que corresponde a um único caractere de palavra. Infelizmente, a história não é a mesma para ‹\b›.

Apesar de todos os sabores neste livro suportarem ‹\b› e ‹\B›, eles diferem quando o assunto é quais caracteres são considerados de palavras.

.NET, JavaScript, PCRE, Perl, Python e Ruby fazem ‹\b› corresponder entre dois caracteres, nos casos em que um caractere é correspondido por ‹\w›, e o outro por ‹\W›. ‹\B› sempre corresponde entre dois caracteres, nos casos em que ambos são correspondidos por ‹\w› ou ‹\W›.

JavaScript, PCRE, e Ruby enxergam somente caracteres ASCII como de palavras. ‹\w› é idêntico a ‹[a-zA-Z0-9_]›. Com estes sabores, você pode fazer uma pesquisa do tipo “apenas palavras inteiras” em palavras de idiomas que usem apenas as letras de A a Z sem sinais diacríticos, como o inglês. Mas estes sabores não podem fazer

pesquisas do tipo “apenas palavras inteiras” em idiomas como o espanhol ou russo.

.NET e Perl tratam as letras e dígitos de todos os alfabetos como caracteres de palavras. Com esses sabores, você pode fazer uma pesquisa do tipo “apenas palavras inteiras” em palavras de qualquer língua, incluindo as que não utilizam o alfabeto latino.

O Python lhe dá uma opção. Caracteres não-ASCII são incluídos somente se você utilizar a sinalização UNICODE ou U ao criar a regex. Essa sinalização afeta ‹\b› e ‹\w› de maneira igual.

O Java se comporta de maneira inconsistente. ‹\w› corresponde somente a caracteres ASCII. Mas ‹\b› está habilitado para Unicode, suportando qualquer alfabeto. Em Java, ‹\b\w\b› corresponde a uma única letra em inglês; um dígito ou um underscore que não faça parte de uma palavra em qualquer idioma. ‹\bкошка\b› corresponderá corretamente à palavra em russo para cat (gato), pois ‹\b› suporta Unicode. Mas ‹\w+› não corresponderá a qualquer palavra em russo, porque ‹\w› suporta apenas caracteres ASCII.

Veja também:

Receita 2.3.

2.7 Pontos de código, propriedades, blocos e alfabetos Unicode

Problema

Use uma expressão regular para localizar o sinal de marca registrada (™), especificando seu ponto de código Unicode, em vez de copiar e colar o caractere em si. Se você gosta de copiar e colar, o sinal de marca registrada é apenas outro caractere literal, embora você não possa digitá-lo diretamente em seu teclado. Caracteres literais são discutidos na receita 2.1.

Crie uma expressão regular que corresponda a qualquer caractere

com a propriedade Unicode “Símbolo de moeda” (Currency Symbol). Propriedades Unicode também são chamadas de categorias Unicode.

Crie uma expressão regular que corresponda a qualquer caractere no bloco Unicode “Grego estendido”.

Crie uma expressão regular que corresponda a qualquer caractere que, de acordo com o padrão Unicode, faça parte do alfabeto Grego.

Crie uma expressão regular que corresponda a um grafema, ou ao que geralmente é considerado um caractere: um caractere-base com todas as suas marcas de combinação.

Solução

Ponto de código Unicode

```
\u2122
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, Python

Esta regex funciona em Python somente quando está entre aspas, na forma de uma string Unicode: `u"\u2122"`.

```
\x{2122}
```

Opções Regex: Nenhuma

Sabores Regex: PCRE, Perl, Ruby 1.9

O PCRE deve ser compilado com suporte a UTF-8; no PHP, ative o suporte UTF-8 com o modificador de padrão `/u`. O Ruby 1.8 não suporta expressões regulares em Unicode.

Propriedade ou categoria Unicode

```
\p{Sc}
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Ruby 1.9

O PCRE deve ser compilado com suporte a UTF-8; no PHP, ative o suporte UTF-8 com o modificador de padrão `/u`. JavaScript e Python

não suportam propriedades Unicode. O Ruby 1.8 não suporta expressões regulares em Unicode.

Bloco Unicode

\p{IsGreekExtended}

Opções Regex: Nenhuma

Sabores Regex: .NET, Perl

\p{InGreekExtended}

Opções Regex: Nenhuma

Sabores Regex: Java, Perl

JavaScript, PCRE, Python e Ruby não suportam blocos Unicode.

Alfabeto Unicode

\p{Greek}

Opções Regex: Nenhuma

Sabores Regex: PCRE, Perl, Ruby 1.9

O suporte a alfabetos Unicode exige o PCRE 6.5 ou posterior, e o PCRE deve ser compilado com suporte a UTF-8. No PHP, ative o suporte UTF-8 com o modificador de padrão /u. .NET, JavaScript e Python não suportam propriedades Unicode. O Ruby 1.8 não suporta expressões regulares em Unicode.

Grafema Unicode

\X

Opções Regex: Nenhuma

Sabores Regex: PCRE, Perl

PCRE e Perl possuem um símbolo dedicado para corresponder a grafemas, mas também suportam a sintaxe alternativa utilizando propriedades Unicode.

\P{M}\p{M}*

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Ruby 1.9

O PCRE deve ser compilado com suporte a UTF-8; no PHP, ative o

suporte UTF-8 com o modificador de padrão /u. JavaScript e Python não suportam propriedades Unicode. O Ruby 1.8 não suporta expressões regulares em Unicode.

Discussão

Ponto de código Unicode

Um *ponto de código* é uma entrada no banco de dados de caracteres Unicode. Um ponto de código não é o mesmo que um *caractere*, dependendo do sentido que você der ao termo "caractere". O que aparece como um caractere na tela é chamado de *grafema* no Unicode.

O ponto de código Unicode U+2122 representa o caractere "marca registrada". Você pode corresponder a isso usando ‹\u2122› ou ‹\x{2122}›, dependendo do sabor regex em que estiver trabalhando.

A sintaxe ‹\u› requer exatamente quatro dígitos hexadecimais. Isso significa que você poderá usá-la apenas para os pontos de código Unicode, que vão de U+0000 a U+FFFF. A sintaxe ‹\x› permite qualquer quantidade de dígitos hexadecimais, suportando todos os pontos de código, de U+000000 a U+10FFFF. Você pode corresponder a U+00E0 usando ‹\x{E0}› ou ‹\x{00E0}›. Os pontos de código U+100000 são usados muito raramente, e são pouco suportados por fontes e sistemas operacionais.

Pontos de código podem ser utilizados dentro e fora das classes de caracteres.

Propriedade ou categoria Unicode

Cada ponto de código Unicode possui exatamente uma propriedade Unicode, ou se encaixa em uma única categoria Unicode. Existem 30 categorias Unicode agrupadas em sete super categorias:

‹\p{L}›

Qualquer tipo de letra em qualquer idioma.

‹\p{Ll}›

Uma letra minúscula que tenha uma variante maiúscula.

‹\p{Lu}›

Uma letra maiúscula que tenha uma variante minúscula.

‹\p{Lt}›

Uma letra que apareça no início de uma palavra, quando somente sua primeira letra for maiúscula.

‹\p{Lm}›

Um caractere especial que é usado como uma letra.

‹\p{Lo}›

Uma letra ou ideograma que não tenha variantes maiúsculas e minúsculas.

‹\p{M}›

Um caractere destinado a ser combinado com outro caractere (acentos, tremas, caixas de cerco etc.).

‹\p{Mn}›

Um caractere destinado a ser combinado com outro caractere e que não ocupe espaço extra (por exemplo, acentos, tremas etc.).

‹\p{Mc}›

Um caractere destinado a ser combinado com outro caractere, e que ocupe espaço extra (por exemplo, sinais de vogais em várias escritas orientais).

‹\p{Me}›

Um caractere que englobe outro caractere (círculo, quadrado, keycap[1] etc.).

‹\p{Z}›

Qualquer tipo de espaço em branco ou separador invisível.

‹\p{Zs}›

Um caractere de espaço em branco invisível, mas que ocupe espaço.

‹\p{Zl}›

O caractere separador de linha U+2028.

‹\p{Zp}›

O caractere separador de parágrafo U+2029.

‹\p{S}›

Símbolos matemáticos, sinais de moeda, dingbats, caracteres de desenho de caixa etc.

‹\p{Sm}›

Qualquer símbolo matemático.

‹\p{Sc}›

Qualquer sinal de moeda.

‹\p{Sk}›

Um caractere de combinação (marca), como um caractere completo em si.

‹\p{So}›

Vários símbolos que não sejam símbolos matemáticos, sinais de moeda, ou caracteres de combinação.

‹\p{N}›

Qualquer tipo de caractere numérico em qualquer alfabeto.

‹\p{Nd}›

Um dígito de 0 a 9 em qualquer alfabeto, exceto alfabetos ideográficos.

‹\p{Nl}›

Um número que se pareça com uma letra, como um numeral Romano.

‹\p{No}›

Um dígito subscrito ou sobrescrito, ou um número que não seja um dígito de 0 a 9 (excluindo os números dos alfabetos ideográficos).

‹\p{P}›

Qualquer tipo de caractere de pontuação.

‹\p{Pd}›

Qualquer tipo de hífen ou travessão.

‹\p{Ps}›

Qualquer tipo de delimitador de abertura.

‹\p{Pe}›

Qualquer tipo de delimitador de fechamento.

‹\p{Pi}›

Qualquer tipo de aspa de abertura.

‹\p{Pf}›

Qualquer tipo de aspa de fechamento.

‹\p{Pc}›

Um caractere de conexão, como um underscore que conecte palavras.

‹\p{Po}›

Qualquer tipo de caractere de pontuação que não seja um travessão, delimitador, aspa ou conector.

‹\p{C}›

Caracteres de controle invisíveis e pontos de código não atribuídos.

‹\p{Cc}›

Um caractere de controle ASCII, de 0x00 a 0x1F, ou Latin-1, de 0x80 a 0x9F.

‹\p{Cf}›

Um indicador de formatação invisível.

‹\p{Co}›

Qualquer ponto de código reservado para uso privado.

‹\p{Cs}›

Metade de um par substituto (surrogate pair) na codificação UTF-16.

‹\p{Cn}›

Qualquer ponto de código para o qual nenhum caractere tenha sido atribuído.

‹\p{Ll}› corresponde a um ponto de código único que contenha Ll, ou a propriedade “letra minúscula” (lowercase letter). ‹\p{L}› é uma maneira rápida de escrever ‹[\p{Ll}\p{Lu}\p{Lt}\p{Lm}\p{Lo}]›, que

corresponde a um único ponto de código em qualquer uma das categorias de “letra”.

‹\P› é a versão negada de ‹\p›. ‹\P{Ll}› corresponde a um ponto de código único que não tenha a propriedade Ll. ‹\P{L}› corresponde a um ponto de código único que não tenha a propriedade “letra”. Não é o mesmo que ‹[\P{Ll}\P{Lu}\P{Lt}\P{Lm}\P{Lo}]›, que corresponde a todos os pontos de código. ‹\P{Ll}› corresponde aos pontos de código com a propriedade Lu (e todas as outras propriedades, exceto Ll), enquanto o valor ‹\P{Lu}› inclui os pontos de código Ll. Combinar apenas estes dois em uma classe de ponto de código já corresponderá a todos os pontos de código possíveis.

Bloco Unicode

O banco de dados de caracteres Unicode divide todos os pontos de código em blocos. Cada bloco é composto por um intervalo único de pontos de código. Os pontos de código de U+0000 a U+FFFF estão divididos em 105 blocos:

U+0000…U+007F ‹\p{InBasic_Latin}›
U+0080…U+00FF ‹\p{InLatin-1_Supplement}›
U+0100…U+017F ‹\p{InLatin_Extended-A}›
U+0180…U+024F ‹\p{InLatin_Extended-B}›
U+0250…U+02AF ‹\p{InIPA_Extensions}›
U+02B0…U+02FF ‹\p{InSpacing_Modifier_Letters}›
U+0300…U+036F ‹\p{InCombining_Diacritical_Marks}›
U+0370…U+03FF ‹\p{InGreek_and_Coptic}›
U+0400…U+04FF ‹\p{InCyrillic}›
U+0500…U+052F ‹\p{InCyrillic_Supplementary}›
U+0530…U+058F ‹\p{InArmenian}›
U+0590…U+05FF ‹\p{InHebrew}›
U+0600…U+06FF ‹\p{InArabic}›
U+0700…U+074F ‹\p{InSyriac}›
U+0780…U+07BF ‹\p{InThaana}›
U+0900…U+097F ‹\p{InDevanagari}›
U+0980…U+09FF ‹\p{InBengali}›
U+0A00…U+0A7F ‹\p{InGurmukhi}›
U+0A80…U+0AFF ‹\p{InGujarati}›
U+0B00…U+0B7F ‹\p{InOriya}›
U+0B80…U+0BFF ‹\p{InTamil}›

U+0C00…U+0C7F ‹\p{InTelugu}›
U+0C80…U+0CFF ‹\p{InKannada}›
U+0D00…U+0D7F ‹\p{InMalayalam}›
U+0D80…U+0DFF ‹\p{InSinhala}›
U+0E00…U+0E7F ‹\p{InThai}›
U+0E80…U+0EFF ‹\p{InLao}›
U+0F00…U+0FFF ‹\p{InTibetan}›
U+1000…U+109F ‹\p{InMyanmar}›
U+10A0…U+10FF ‹\p{InGeorgian}›
U+1100…U+11FF ‹\p{InHangul_Jamo}›
U+1200…U+137F ‹\p{InEthiopic}›
U+13A0…U+13FF ‹\p{InCherokee}›
U+1400…U+167F ‹\p{InUnified_Canadian_Aboriginal_Syllabics}›
U+1680…U+169F ‹\p{InOgham}›
U+16A0…U+16FF ‹\p{InRunic}›
U+1700…U+171F ‹\p{InTagalog}›
U+1720…U+173F ‹\p{InHanunoo}›
U+1740…U+175F ‹\p{InBuhid}›
U+1760…U+177F ‹\p{InTagbanwa}›
U+1780…U+17FF ‹\p{InKhmer}›
U+1800…U+18AF ‹\p{InMongolian}›
U+1900…U+194F ‹\p{InLimbu}›
U+1950…U+197F ‹\p{InTai_Le}›
U+19E0…U+19FF ‹\p{InKhmer_Symbols}›
U+1D00…U+1D7F ‹\p{InPhonetic_Extensions}›
U+1E00…U+1EFF ‹\p{InLatin_Extended_Additional}›
U+1F00…U+1FFF ‹\p{InGreek_Extended}›
U+2000…U+206F ‹\p{InGeneral_Punctuation}›
U+2070…U+209F ‹\p{InSuperscripts_and_Subscripts}›
U+20A0…U+20CF ‹\p{InCurrency_Symbols}›
U+20D0…U+20FF ‹\p{InCombining_Diacritical_Marks_for_Symbols}›
U+2100…U+214F ‹\p{InLetterlike_Symbols}›
U+2150…U+218F ‹\p{InNumber_Forms}›
U+2190…U+21FF ‹\p{InArrows}›
U+2200…U+22FF ‹\p{InMathematical_Operators}›
U+2300…U+23FF ‹\p{InMiscellaneous_Technical}›
U+2400…U+243F ‹\p{InControl_Pictures}›
U+2440…U+245F ‹\p{InOptical_Character_Recognition}›
U+2460…U+24FF ‹\p{InEnclosed_Alphanumerics}›
U+2500…U+257F ‹\p{InBox_Drawing}›
U+2580…U+259F ‹\p{InBlock_Elements}›
U+25A0…U+25FF ‹\p{InGeometric_Shapes}›
U+2600…U+26FF ‹\p{InMiscellaneous_Symbols}›

U+2700…U+27BF ‹\p{InDingbats}›
U+27C0…U+27EF ‹\p{InMiscellaneous_Mathematical_Symbols-A}›
U+27F0…U+27FF ‹\p{InSupplemental_Arrows-A}›
U+2800…U+28FF ‹\p{InBraille_Patterns}›
U+2900…U+297F ‹\p{InSupplemental_Arrows-B}›
U+2980…U+29FF ‹\p{InMiscellaneous_Mathematical_Symbols-B}›
U+2A00…U+2AFF ‹\p{InSupplemental_Mathematical_Operators}›
U+2B00…U+2BFF ‹\p{InMiscellaneous_Symbols_and_Arrows}›
U+2E80…U+2EFF ‹\p{InCJK_Radicals_Supplement}›
U+2F00…U+2FDF ‹\p{InKangxi_Radicals}›
U+2FF0…U+2FFF ‹\p{InIdeographic_Description_Characters}›
U+3000…U+303F ‹\p{InCJK_Symbols_and_Punctuation}›
U+3040…U+309F ‹\p{InHiragana}›
U+30A0…U+30FF ‹\p{InKatakana}›
U+3100…U+312F ‹\p{InBopomofo}›
U+3130…U+318F ‹\p{InHangul_Compatibility_Jamo}›
U+3190…U+319F ‹\p{InKanbun}›
U+31A0…U+31BF ‹\p{InBopomofo_Extended}›
U+31F0…U+31FF ‹\p{InKatakana_Phonetic_Extensions}›
U+3200…U+32FF ‹\p{InEnclosed_CJK_Letters_and_Months}›
U+3300…U+33FF ‹\p{InCJK_Compatibility}›
U+3400…U+4DBF ‹\p{InCJK_Unified_Ideographs_Extension_A}›
U+4DC0…U+4DFF ‹\p{InYijing_Hexagram_Symbols}›
U+4E00…U+9FFF ‹\p{InCJK_Unified_Ideographs}›
U+A000…U+A48F ‹\p{InYi_Syllables}›
U+A490…U+A4CF ‹\p{InYi_Radicals}›
U+AC00…U+D7AF ‹\p{InHangul_Syllables}›
U+D800…U+DB7F ‹\p{InHigh_Surrogates}›
U+DB80…U+DBFF ‹\p{InHigh_Private_Use_Surrogates}›
U+DC00…U+DFFF ‹\p{InLow_Surrogates}›
U+E000…U+F8FF ‹\p{InPrivate_Use_Area}›
U+F900…U+FAFF ‹\p{InCJK_Compatibility_Ideographs}›
U+FB00…U+FB4F ‹\p{InAlphabetic_Presentation_Forms}›
U+FB50…U+FDFF ‹\p{InArabic_Presentation_Forms-A}›
U+FE00…U+FE0F ‹\p{InVariation_Selectors}›
U+FE20…U+FE2F ‹\p{InCombining_Half_Marks}›
U+FE30…U+FE4F ‹\p{InCJK_Compatibility_Forms}›
U+FE50…U+FE6F ‹\p{InSmall_Form_Variants}›
U+FE70…U+FEFF ‹\p{InArabic_Presentation_Forms-B}›
U+FF00…U+FFEF ‹\p{InHalfwidth_and_Fullwidth_Forms}›
U+FFF0…U+FFFF ‹\p{InSpecials}›

Um bloco Unicode é um intervalo único e contíguo de pontos de

código. Embora muitos blocos tenham os nomes de alfabetos e categorias Unicode, eles não correspondem 100% a eles. O nome de um bloco indica seu uso principal.

O bloco de moedas (Currency) não inclui os símbolos do dólar e iene. Estes são encontrados nos blocos Basic_Latin e Latin-1_Supplement por razões históricas. Ambos possuem a propriedade de símbolo de moeda (currency symbol). Para corresponder a qualquer símbolo de moeda, use \p{Sc} em vez de \p{InCurrency}.

A maioria dos blocos inclui pontos de código não atribuídos, cobertos pela propriedade ‹\p{Cn}›. Nenhuma outra propriedade Unicode, e nenhum dos alfabetos Unicode, inclui pontos de código não atribuídos.

A sintaxe ‹\p{InBlockName}› funciona com o .NET e Perl. O Java utiliza a sintaxe ‹\p{IsBlockName}›.

Perl também suporta a variante Is, mas recomendamos que você fique com a sintaxe In, para evitar confusão com alfabetos Unicode. No caso de scripts, Perl suporta ‹\p{Script}› e ‹\p{IsScript}›, mas não ‹\p{InScript}›.

Alfabeto Unicode

Cada ponto de código Unicode, exceto os não atribuídos, faz parte de um alfabeto Unicode. Pontos de código não atribuídos não fazem parte de nenhum alfabeto. Os pontos de código atribuídos até U+FFFF fazem parte destes alfabetos:

‹\p{Bengali}› ‹\p{Latin}›
‹\p{Bopomofo}› ‹\p{Limbu}›
‹\p{Braille}› ‹\p{Malayalam}›
‹\p{Buhid}› ‹\p{Mongolian}›
‹\p{CanadianAboriginal}› ‹\p{Myanmar}›
‹\p{Cherokee}› ‹\p{Ogham}›
‹\p{Cyrillic}› ‹\p{Oriya}›
‹\p{Devanagari}› ‹\p{Runic}›
‹\p{Ethiopic}› ‹\p{Sinhala}›
‹\p{Georgian}› ‹\p{Syriac}›
‹\p{Greek}› ‹\p{Tagalog}›

‹\p{Gujarati}› ‹\p{Tagbanwa}›
‹\p{Gurmukhi}› ‹\p{TaiLe}›
‹\p{Han}› ‹\p{Tamil}›
‹\p{Hangul}› ‹\p{Telugu}›
‹\p{Hanunoo}› ‹\p{Thaana}›
‹\p{Hebrew}› ‹\p{Thai}›
‹\p{Hiragana}› ‹\p{Tibetan}›
‹\p{Inherited}› ‹\p{Yi}›
‹\p{Kannada}›

Um alfabeto (script) é um grupo de pontos de código utilizados por um sistema particular de escrita humana. Alguns scripts, como o Thai (tailandês), dizem respeito a um único idioma humano. Outros alfabetos, como o Latin (latino), compreendem vários idiomas. Alguns idiomas são compostos de múltiplos alfabetos. Por exemplo, não há nenhum alfabeto Unicode japonês; em vez disso, o Unicode oferece os alfabetos Hiragana, Katakana, Han, e Latin, que geralmente compõem os documentos japoneses.

Nós listamos o alfabeto Common (comum) em primeiro lugar, fora da ordem alfabética. Este alfabeto contém todos os tipos de caracteres comuns a uma grande variedade de scripts, tais como pontuação, espaços em branco e símbolos diversos.

Grafema Unicode

A diferença entre os pontos de código e os caracteres vem à tona quando existem *marcas de combinação*. O ponto de código Unicode U+0061 é a “letra minúscula Latina ‘a’”, enquanto U+00E0 é a “letra minúscula Latina ‘a’, com acento grave”. Ambos representam o que a maioria das pessoas descreve como um caractere.

U+0300 é a marca de combinação “acento grave combinado”. Ele só pode ser usado de maneira razoável após uma letra. Uma string de caracteres composta pelos pontos de código Unicode U+0061 e U+0300 será exibida como à, assim como U+00E0. A marca de combinação U+0300 é exibida no topo do caractere U+0061.

A razão da existência de duas formas diferentes de apresentar uma letra acentuada é que muitos conjuntos de caracteres históricos

codificam o “‘a’ com acento grave” como um caractere único. Designers do Unicode acharam que seria útil ter um mapeamento um-por-um dos conjuntos de caracteres legados populares, assim como da forma do Unicode de separar as marcas e as letras-base, o que torna possível combinações arbitrárias não suportadas por conjuntos de caracteres legados.

O que importa para você, como usuário regex, é que todos os sabores regex discutidos neste livro operam com pontos de código, ao invés de caracteres gráficos. Quando dizemos que a expressão regular ‹.› corresponde a um único caractere, ela realmente corresponde a apenas um único ponto de código. Se seu texto de assunto consiste em dois pontos de código, U+0061 e U+0300, que podem ser representados como a string literal "\u0061\u0300" em uma linguagem de programação como o Java, ‹.› corresponderá apenas ao ponto de código U+0061, ou a, sem o acento U+0300. A regex ‹..› corresponderá a ambos os códigos.

Perl e PCRE oferecem um símbolo regex especial ‹\X›, que corresponde a qualquer grafema Unicode. Essencialmente, é a versão Unicode do ponto venerável. Ele corresponde a qualquer ponto de código Unicode que não seja uma marca de combinação, junto com todas as marcas de combinação que o seguem, caso haja alguma. ‹\P{M}\p{M}*› faz a mesma coisa utilizando a sintaxe de propriedade Unicode. ‹\X› encontrará duas correspondências no texto àà, independentemente da forma em que estiver codificado. Se estiver codificado como "\u00E0\u0061\u0300", a primeira correspondência será "\u00E0", e a segunda, "\u0061\u0300".

Variações

Variante negada

A maiúscula ‹\P› é a variante negada da minúscula ‹\p›. Por exemplo, ‹\P{Sc}› corresponde a qualquer caractere que não contenha a propriedade Unicode “Símbolo de moeda”. ‹\P› é suportada por todos os sabores que suportem ‹\p›, e por todas as

propriedades, blocos, e alfabetos que eles suportem.

Classes de caracteres

Todos os sabores permitem que todos os tokens ‹\u›, ‹\x›, ‹\p›, e ‹\P›, suportados por eles, sejam utilizados dentro de classes de caracteres. O caractere representado pelo ponto de código, ou os caracteres da categoria, bloco, ou alfabeto são, então, adicionados à classe de caracteres. Por exemplo, você poderia corresponder a um caractere que seja uma aspa de abertura (propriedade de pontuação inicial), uma aspa de fechamento (propriedade de pontuação final) ou o símbolo de marca registrada (U +2122) com:

```
[\p{Pi}\p{Pf}\x{2122}]
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Ruby 1.9

Listando todos os caracteres

Se o seu sabor de expressão regular não suporta categorias, blocos ou alfabetos Unicode, você pode listar os caracteres que fazem parte da categoria, bloco ou alfabeto dentro de uma classe de caracteres.

No caso dos blocos, isso é muito fácil: cada bloco é apenas um intervalo entre dois pontos de código. O bloco Grego Estendido compreende os caracteres de U+1F00 a U+1FFF:

```
[\u1F00-\u1FFF]
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, Python

```
[\x{1F00}-\x{1FFF}]
```

Opções Regex: Nenhuma

Sabores Regex: PCRE, Perl, Ruby 1.9

No caso da maioria das categorias e de vários alfabetos, a classe de caracteres equivalente é uma longa lista de pontos de código individuais, de intervalos curtos. Os caracteres que compõem cada categoria, e muitos dos alfabetos, estão espalhados por toda a

tabela Unicode. Este é o alfabeto Grego:

```
[\u0370-\u0373\u0375\u0376-\u0377\u037A\u037B-\u037D\u0384\u0386↲
\u0388-\u038A\u038C\u038E-\u03A1\u03A3-\u03E1\u03F0-\u03F5\u03F6↲
\u03F7-\u03FF\u1D26-\u1D2A\u1D5D-\u1D61\u1D66-\u1D6A\u1DBF\u1F00-
\u1F15↲
\u1F18-\u1F1D\u1F20-\u1F45\u1F48-\u1F4D\u1F50-
\u1F57\u1F59\u1F5B\u1F5D↲
\u1F5F-\u1F7D\u1F80-\u1FB4\u1FB6-\u1FBC\u1FBD\u1FBE\u1FBF-\u1FC1↲
\u1FC2-\u1FC4\u1FC6-\u1FCC\u1FCD-\u1FCF\u1FD0-\u1FD3\u1FD6-\u1FDB↲
\u1FDD-\u1FDF\u1FE0-\u1FEC\u1FED-\u1FEF\u1FF2-\u1FF4\u1FF6-\u1FFC↲
\u1FFD-\u1FFE\u2126]
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, Python

Nós construímos essa expressão regular copiando a listagem do alfabeto grego de *http://www.unicode.org/Public/UNIDATA/Scripts.txt*, e depois pesquisando e substituindo por meio de três expressões regulares:

1. Pesquisar pela expressão regular ‹;.*›, e substituir suas correspondências com nada, termina por apagar os comentários. Se ela apagar tudo, desfaça a ação e desative “ponto corresponde a quebras de linha”.
2. Pesquisar por ‹^› com “^ e $ correspondem em quebras de linha” ativado, e substituir por «\u», prefixando os pontos de código com \u. Substituir ‹\.\.› por «-\u» corrige os intervalos.
3. Por fim, substituir ‹\s+› por nada remove as quebras de linha. Adicionar os colchetes em torno da classe de caracteres finaliza a regex. Talvez você tenha de adicionar \u no início da classe de caracteres ou removê-lo do final, caso você tenha incluído alguma linha em branco, inicial ou final, ao copiar a lista de Scripts.txt.

Pode parecer que isso dá muito trabalho, mas levou menos de um minuto. Escrever a descrição demorou muito mais. Fazer isso para a sintaxe \x{} também é muito fácil:

1. Pesquisar pela expressão regular ‹;.*›, e substituir suas correspondências por nada, termina por apagar os comentários. Se ela apagar tudo, desfaça a ação e desative “ponto corresponde a quebras de linha”.
2. Pesquisar por ‹^› com “^ e $ correspondem em quebras de linha” ativado, e substituir por «\x{», prefixando os pontos de código com \x{. Substituir ‹\.\.› por «}-\x{» corrige os intervalos.
3. Finalmente, substituir ‹\s+› por «}» adiciona a chave de fechamento e remove as quebras de linhas. Adicionar os colchetes na classe de caracteres finaliza a regex. Talvez você precise adicionar \x{ no início da classe de caracteres ou removê-lo do final, caso você tenha incluído alguma linha em branco, inicial ou final, ao copiar a lista de Scripts.txt.

O resultado fica assim:

```
[\x{0370}-\x{0373}\x{0375}\x{0376}-\x{0377}\x{037A}\x{037B}-\x{037D}↲
\x{0384}\x{0386}\x{0388}-\x{038A}\x{038C}\x{038E}-\x{03A1}↲
\x{03A3}-\x{03E1}\x{03F0}-\x{03F5}\x{03F6}\x{03F7}-\x{03FF}↲
\x{1D26}-\x{1D2A}\x{1D5D}-\x{1D61}\x{1D66}-\x{1D6A}\x{1DBF}↲
\x{1F00}-\x{1F15}\x{1F18}-\x{1F1D}\x{1F20}-\x{1F45}\x{1F48}-\x{1F4D}↲
\x{1F50}-\x{1F57}\x{1F59}\x{1F5B}\x{1F5D}\x{1F5F}-\x{1F7D}↲
\x{1F80}-\x{1FB4}\x{1FB6}-\x{1FBC}\x{1FBD}\x{1FBE}\x{1FBF}-\x{1FC1}↲
\x{1FC2}-\x{1FC4}\x{1FC6}-\x{1FCC}\x{1FCD}-\x{1FCF}\x{1FD0}-\x{1FD3}↲
\x{1FD6}-\x{1FDB}\x{1FDD}-\x{1FDF}\x{1FE0}-\x{1FEC}\x{1FED}-\x{1FEF}↲
\x{1FF2}-\x{1FF4}\x{1FF6}-\x{1FFC}\x{1FFD}-\x{1FFE}\x{2126}↲
\x{10140}-\x{10174}\x{10175}-\x{10178}\x{10179}-\x{10189}↲
\x{1018A}\x{1D200}-\x{1D241}\x{1D242}-\x{1D244}\x{1D245}]
```

Opções Regex: Nenhuma

Sabores Regex: PCRE, Perl, Ruby 1.9

Veja também:

http://www.unicode.org é o site oficial do Consórcio Unicode, no qual você poderá baixar todos os documentos Unicode oficiais, tabelas de caracteres etc.

Unicode é um tema muito vasto, tema este que motivou a publicação de vários livros. Um dos livros é '*Unicode Explained*' de Jukka K. Korpela (O'Reilly).

Não podemos explicar tudo o que você precisa saber sobre pontos de código, propriedades, blocos e alfabetos Unicode em apenas uma seção. Nós nem sequer tentamos explicar porque você deve se importar com o Unicode – mas você deve. A simplicidade confortável da tabela estendida ASCII tornou-se um lugar solitário no atual mundo globalizado.

2.8 Corresponder a uma dentre várias alternativas

Problema

Crie uma expressão regular que, quando aplicada repetidamente ao texto Mary, Jane, and Sue went to Mary's house, corresponderá a Mary, Jane, Sue, e, em seguida, a Mary novamente. Outras tentativas de correspondências devem falhar.

Solução

```
Mary|Jane|Sue
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

A *barra vertical,* ou *símbolo pipe*, divide a expressão regular em múltiplas *alternativas*. ‹Mary|Jane|Sue› corresponde a Mary, Jane ou Sue, a cada tentativa de correspondência. Somente um nome por vez será correspondido, mas, a cada vez, poderá corresponder a um nome diferente.

Todos os sabores de expressões regulares discutidos nesse livro utilizam um mecanismo orientado a expressões. Esse mecanismo é, simplesmente, o programa que faz a expressão regular funcionar.

Orientado a expressões regulares[2] significa que todas as permutações possíveis da expressão regular são testadas em cada posição de caractere no texto de assunto, antes da regex ser testada na posição de caractere seguinte.

Quando você aplica ‹Mary|Jane|Sue› a Mary, Jane, and Sue went to Mary's house, a correspondência Mary é encontrada imediatamente no início da string.

Quando você aplica a mesma regex para o restante da string – por exemplo, clicando em “Localizar próxima” em seu editor de texto, o mecanismo de expressão regular tenta corresponder ‹Mary› à primeira vírgula da string. E ocorre uma falha. Em seguida, ele tenta corresponder ‹Jane› à mesma posição, e falha de novo. A tentativa de corresponder ‹Sue› à vírgula também falha. Só então é que o mecanismo de expressão regular avança para o próximo caractere na string. Iniciando no primeiro espaço, todas as três alternativas falharão do mesmo modo.

Iniciando no J, a primeira alternativa, ‹Mary›, falha na correspondência. Então, a segunda alternativa, ‹Jane›, é, então, testada, começando no J. Ela corresponde a Jane. O mecanismo de expressão regular declara vitória.

Observe que Jane foi encontrado, embora haja outra ocorrência de Mary no texto de assunto, e que ‹Mary› aparece antes de ‹Jane› na regex. Pelo menos nesse caso, a ordem das alternativas na expressão regular não importa. A expressão regular encontra a correspondência *mais à esquerda*. Ela examina o texto da esquerda para a direita e tenta todas as alternativas na expressão regular a cada etapa, parando na primeira posição do texto em que qualquer uma das alternativas produza uma correspondência válida.

Se fizermos outra pesquisa com o restante da string, Sue será encontrada. A quarta pesquisa encontrará Mary mais uma vez. Se você pedir para que o mecanismo faça uma quinta pesquisa, ele irá falhar, pois nenhuma das três alternativas corresponde ao restante da string ('s house).

A ordem das alternativas na expressão regular importa apenas quando duas delas podem corresponder à mesma posição na string. A regex ‹Jane|Janet› tem duas alternativas que correspondem à mesma posição no texto Her name is Janet. Não há extremidades de palavras nessa expressão regular. O fato de que ‹Jane› corresponde apenas parcialmente à palavra Janet, em Her name is Janet, não importa.

‹Jane|Janet› corresponde a Jane em Her name is Janet, pois o mecanismo orientado a expressões regulares é *ansioso*. Além de verificar o texto de assunto da esquerda para a direita, encontrando a correspondência mais à esquerda, ele também verifica as alternativas, na expressão regular, da esquerda para a direita. O mecanismo para assim que encontra uma alternativa que corresponda.

Quando ‹Jane|Janet› atinge o J em Her name is Janet, a primeira alternativa, ‹Jane›, corresponde. A segunda alternativa não é tentada. Se dissermos para o mecanismo testar uma segunda correspondência, o t é tudo o que resta no texto de assunto. Nenhuma alternativa corresponderá nessa posição.

Há duas maneiras de fazer Jane parar de roubar a cena de Janet. Uma forma é colocar a alternativa mais longa em primeiro lugar: ‹Janet|Jane›. Uma solução mais sólida seria explicitar o que estamos tentando fazer: estamos à procura de nomes, e nomes são palavras completas. As expressões regulares não lidam com palavras, mas podem lidar com extremidades de palavras.

Assim, tanto ‹\bJane\b|\bJanet\b› quanto ‹\bJanet\b|\bJane\b› corresponderão a Janet em Her name is Janet. Devido às extremidades de palavra, só uma alternativa poderá corresponder. A ordem das alternativas, novamente, é irrelevante.

A receita 2.12 explica a melhor solução: ‹\bJanet?\b›.

Veja também:

Receita 2.9.

2.9 Agrupar e capturar partes da correspondência

Problema

Melhore a expressão regular para corresponder a Mary, Jane ou Sue, forçando a correspondência a ser uma palavra inteira. Use agrupamento para obter tal resultado com apenas um par de extremidades de palavra, para toda a expressão regular, ao invés de um par para cada alternativa.

Crie uma expressão regular que corresponda a qualquer data no formato aaaa-mm-dd, e capture separadamente ano, mês e dia. O objetivo é facilitar o trabalho com esses valores, em separado, no código que processa a correspondência. Você pode assumir que todas as datas no texto de assunto são válidas. A expressão regular não precisa excluir coisas como 9999-99-99, pois isso não ocorrerá no texto de assunto, de forma alguma.

Solução

```
\b(Mary|Jane|Sue)\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
\b(\d\d\d\d)-(\d\d)-(\d\d)\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

O operador de alternância, explicado na seção anterior, tem a prioridade mais baixa entre todos os operadores regex. Se você tentar ‹\bMary|Jane|Sue\b› as três alternativas serão ‹\bMary›, ‹Jane› e ‹Sue\b›. Esta regex corresponde a Jane em Her name is Janet.

Se quiser que alguma coisa seja excluída da alternância em sua regex, você terá de *agrupar* as alternativas. O agrupamento é feito

com parênteses. Eles têm a prioridade mais alta entre todos os operadores, assim como na maioria das linguagens de programação. ‹\b(Mary|Jane|Sue)\b› possui três alternativas – ‹Mary›, ‹Jane›, e ‹Sue› – entre duas extremidades de palavra. Esta regex não corresponde a nada em Her name is Janet.

Quando o mecanismo de expressão regular atinge o J em Janet, no texto de assunto, a primeira extremidade de palavra é correspondida. Em seguida, o mecanismo entra no grupo. A primeira alternativa do grupo, ‹Mary›, falha. A segunda alternativa, ‹Jane›, é bem-sucedida. O mecanismo sai do grupo. Tudo o que resta é ‹\b›. A extremidade de palavra falha ao tentar corresponder entre o e e o t no final do assunto. A tentativa de correspondência global, iniciada a partir de J, falha.

Um par de parênteses não é apenas um grupo, é um *grupo de captura*. No caso da regex Mary-Jane-Sue, a captura não é muito útil, pois ela simplesmente representa a correspondência global da regex. As capturas tornam-se úteis quando cobrem apenas uma parte da expressão regular, como em ‹\b(\d\d\d\d)-(\d\d)-(\d\d)\b›.

Essa expressão regular corresponde a uma data no formato aaaa-mm-dd. A regex ‹\b\d\d\d\d-\d\d-\d\d\b› faz exatamente o mesmo. Pelo fato desta expressão regular não usar qualquer alternância ou repetição, a função de agrupamento dos parênteses não é necessária. Porém, a função de captura é muito útil.

A regex ‹\b(\d\d\d\d)-(\d\d)-(\d\d)\b› possui três grupos de captura. Os grupos são numerados pela contagem dos parênteses de abertura, da esquerda para a direita. ‹(\d\d\d\d)› é o grupo número 1. ‹(\d\d)› é o número 2. O segundo ‹(\d\d)› forma o grupo número 3.

Durante o processo de correspondência, quando o mecanismo de expressão regular sai do grupo, após atingir o parêntese de fechamento, ele armazena a parte do texto correspondido pelo grupo de captura. Quando nossa regex corresponde a 2008-05-24, 2008 é armazenado na primeira captura, 05 na segunda e 24, na terceira.

Existem três maneiras de usar o texto capturado. A receita 2.10, neste capítulo, explica como corresponder novamente ao texto capturado dentro da mesma correspondência regex. A receita 2.21 mostra como inserir o texto capturado no texto de substituição, ao fazer uma pesquisa-e-substituição. A receita 3.9, no próximo capítulo, descreve como seu aplicativo pode usar as partes da correspondência da regex.

Variações

Grupos de não-captura

Na regex ‹\b(Mary|Jane|Sue)\b›, precisamos dos parênteses somente para agrupamento.Ao invés de usar um grupo de captura, poderíamos usar um grupo de não-captura:

```
\b(?:Mary|Jane|Sue)\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Os três caracteres ‹(?:› abrem o grupo de não-captura. O parêntese ‹)› o fecha. O grupo de não-captura fornece a mesma funcionalidade do grupo de captura, mas não captura nada.

Ao contar os parênteses de abertura dos grupos de captura para determinar seus números, não conte o parêntese do grupo de não-captura. Este é o principal benefício dos grupos de não-captura: você pode adicioná-los a uma regex existente, sem prejudicar as referências a grupos de captura numerados.

Outra vantagem dos grupos de não-captura é o desempenho. Se você não for utilizar uma retrorreferência em um determinado grupo (Receita 2.10) recoloque-a no texto de substituição (Receita 2.21), ou recupere sua correspondência no código-fonte (Receita 3.9); um grupo de captura adiciona uma sobrecarga desnecessária, que pode ser eliminada por meio de um grupo de não-captura. Na prática, dificilmente você notará alguma diferença no desempenho, a menos que você use a regex dentro de um loop curto ou em uma grande

massa de dados.

Grupo com modificadores de modo

Na variação “Correspondência sem diferenciação entre maiúsculas e minúsculas”, da receita 2.1, explicamos que o .NET, Java, PCRE, Perl e Ruby suportam modificadores de modo local, utilizando o modo de chaveamento: ‹sensitive(?i)caseless(?-i)sensitive›. Embora esta sintaxe também envolva os parênteses, um chaveamento como ‹(?i)› não envolve nenhum tipo de agrupamento.

No lugar de usar chaveamentos, você pode especificar modificadores de modo em um grupo de não-captura:

```
\b(?i:Mary|Jane|Sue)\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Ruby

```
sensível(?i:insensível)sensível
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Ruby

Adicionar modificadores de modo em um grupo de não-captura define aquele modo para a parte da expressão regular que está dentro do grupo. As configurações anteriores serão restauradas no parêntese de fechamento. Como a sensibilidade entre maiúsculas e minúsculas é o padrão, somente a parte de dentro da regex:

```
(?i:...)
```

não diferencia maiúsculas e minúsculas.

Você pode combinar vários modificadores ‹(?ism:group)›. Use um hífen para desativar os modificadores: ‹(?-ism:group)› desativa as três opções. ‹(?i-sm)› ativa a insensibilidade (i), e desativa “ponto corresponde a quebras de linha” (s) e “^ e $ correspondem em quebras de linha” (m). Estas opções são explicadas nas receitas 2.4 e 2.5.

Veja também:

Receitas 2.10, 2.11, 2.21, e 3.9.

2.10 Corresponder novamente a textos previamente correspondidos

Problema

Crie uma expressão regular que corresponda a datas “mágicas”, no formato aaaa-mm-dd. A data é mágica se a dezena do ano, o mês e o dia possuírem os mesmos números. Por exemplo, 2008-08-08 é uma data mágica. Você pode assumir que todas as datas no texto de assunto são válidas. A expressão regular não precisa excluir coisas como 9999-99-99, pois elas não ocorrem no texto de assunto. Você só precisa encontrar as datas mágicas.

Solução

```
\b\d\d(\d\d)-\1-\1\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Para corresponder a um texto previamente correspondido dentro de uma regex, em primeiro lugar, temos que capturar o texto anterior. Fazemos isso com um grupo de captura, como mostrado na receita 2.9. Depois, podemos corresponder ao mesmo texto em qualquer lugar da regex usando uma *retrorreferência* (backreference). Você pode fazer referência aos primeiros nove grupos de captura com uma barra invertida seguida de um único dígito, de um a nove. Para grupos de 10 a 99, use de ‹\10› a ‹\99›.

Não utilize ‹\01›. Este procedimento gerará um escape octal ou um erro. Nós não utilizamos escape octal neste livro, porque os escapes hexadecimais ‹\xFF› são muito mais fáceis de entender.

Quando a expressão regular ‹\b\d\d(\d\d)-\1-\1\b› encontra 2008-08-08, o primeiro ‹\d\d› corresponde a 20. Em seguida, o mecanismo entra no grupo de captura, observando a posição atual no texto de assunto.

O ‹\d\d› dentro do grupo de captura corresponde a 08, e o

mecanismo atinge o parêntese de encerramento do grupo. Neste ponto, a correspondência parcial 08 é armazenada no grupo de captura 1.

O próximo símbolo é o hífen, que corresponde literalmente. Depois vem a retrorreferência. O mecanismo verifica o conteúdo do primeiro grupo de captura, 08, e tenta corresponder a este texto literalmente. Se a expressão regular não diferenciar maiúsculas e minúsculas, o texto capturado é correspondido dessa maneira. Aqui a retrorreferência obtém êxito. O próximo hífen e a próxima retrorreferência também alcançam êxito. Finalmente, a extremidade de palavra corresponde ao final do texto de assunto e uma correspondência global é encontrada: 2008-08-08. O grupo de captura ainda retém 08.

Se um grupo de captura for repetido, seja por um quantificador (Receita 2.12), ou por retrocesso (Receita 2.13), a correspondência armazenada será sobrescrita toda vez que o grupo de captura corresponder a algo. Uma retrorreferência ao grupo corresponderá apenas ao texto capturado por último.

Se a mesma regex encontra 2008-05-24 2007-07-07, a primeira vez em que o grupo capturará algo ocorrerá quando ‹\b\d\d(\d\d)› corresponder a 2008, armazenando 08 no primeiro (e único) grupo de captura. Em seguida, o hífen corresponderá a si mesmo. A retrorreferência, que tenta corresponder a ‹08›, falhará contra 05.

Como não existem outras alternativas na expressão regular, o mecanismo desiste da tentativa de correspondência. Isso envolve limpar todos os grupos de captura. Quando o mecanismo tentar novamente, começando no primeiro 0 do assunto, ‹\1› não terá nenhum texto.

Ainda processando 2008-05-24 2007-07-07, a próxima vez em que o grupo capturará algo ocorrerá quando ‹\b\d\d(\d\d)› corresponder a 2007, armazenando 07. Em seguida, o hífen corresponderá a si mesmo. Agora, a retrorreferência tenta corresponder a ‹07›. Ela obtém êxito, assim como o próximo hífen, a próxima retrorreferência

e a extremidade de palavra. 2007/07/07 foi encontrado.

Como o mecanismo progride do início ao fim, você deve colocar os parênteses de captura antes da retrorreferência. As expressões regulares ‹\b\d\d\1-(\d\d)-\1› e ‹\b\d\d\1-\1-(\d\d)\b› nunca corresponderão. Como a retrorreferência é encontrada antes do grupo de captura, ele ainda não capturou nada. A menos que você esteja usando o JavaScript, uma retrorreferência sempre falhará, caso aponte para um grupo que ainda não tenha participado de uma tentativa de correspondência.

Um grupo que não tenha participado não é o mesmo que um grupo que tenha capturado uma correspondência de comprimento zero. Uma retrorreferência a um grupo com uma captura de comprimento zero será sempre bem-sucedida. Quando ‹(^)\1› corresponde ao início da string, o primeiro grupo captura a correspondência de comprimento zero do acento circunflexo, fazendo com que ‹\1› obtenha sucesso. Na prática, isso pode acontecer quando o conteúdo do grupo de captura for completamente opcional.

JavaScript é o único sabor que conhecemos que contraria décadas de tradição em retrorreferências de expressões regulares. Em JavaScript, ou pelo menos em implementações que seguem o padrão do JavaScript, uma retrorreferência a um grupo que ainda não tenha participado sempre obtém êxito, semelhante a uma retrorreferência a um grupo que tenha capturado uma correspondência de comprimento zero. Assim, em JavaScript, ‹\b\d\d\1-\1-(\d\d)\b› pode corresponder a 12--34.

Veja também:

Receitas 2.9, 2.11, 2.21, e 3.9.

2.11 Capturar e nomear partes da correspondência

Problema

Crie uma expressão regular que corresponda a qualquer data no formato aaaa-mm-dd e que capture separadamente o ano, o mês e o dia. O objetivo é facilitar o trabalho com esses valores, em

separado, no código que processa a correspondência. Contribua para este objetivo atribuindo os nomes descritivos “year”, “month” e “day” ao texto capturado.

Crie outra expressão regular que corresponda a datas “mágicas”, no formato aaaa-mm-dd. Uma data é mágica se a dezena do ano, o mês e o dia do mês possuírem os mesmos números. Por exemplo, 2008-08-08 é uma data mágica. Capture o número mágico (08 no exemplo), e rotule-o como “magic”.

Você pode assumir que todas as datas no texto do assunto são válidas. As expressões regulares não precisam excluir coisas como 9999-99-99, pois elas não ocorrerão no texto de assunto.

Solução

Captura nomeada

```
\b(?<year>\d\d\d\d)-(?<month>\d\d)-(?<day>\d\d)\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\b(?'year'\d\d\d\d)-(?'month'\d\d)-(?'day'\d\d)\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\b(?P<year>\d\d\d\d)-(?P<month>\d\d)-(?P<day>\d\d)\b
```

Opções Regex: Nenhuma

Sabores Regex: PCRE 4 e superior, Perl 5.10, Python

Retrorreferências nomeadas

```
\b\d\d(?<magic>\d\d)-\k<magic>-\k<magic>\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\b\d\d(?'magic'\d\d)-\k'magic'-\k'magic'\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\b\d\d(?P<magic>\d\d)-(?P=magic)-(?P=magic)\b
```

Opções Regex: Nenhuma
Sabores Regex: PCRE 4 e superior, Perl 5.10, Python

Discussão

Captura nomeada

As receitas 2.9 e 2.10 ilustram *grupos de captura* e *retrorreferências*. Para ser mais preciso: essas receitas utilizam grupos de captura *numerados* e retrorreferências numeradas. Cada grupo recebe automaticamente um número, que você utiliza nas retrorreferências.

Sabores regex modernos suportam grupos de captura *nomeados*, além de grupos numerados. A única diferença entre os grupos nomeados e os numerados é a capacidade, do primeiro, de atribuir nome descritivo, em vez de se prender a números automáticos. Grupos nomeados tornam sua expressão regular mais legível e mais fácil de manter. Inserir um grupo de captura em uma regex existente pode alterar os números atribuídos a todos os grupos de captura. Nomes que você atribui permanecem os mesmos.

O Python foi o primeiro sabor de expressão regular a suportar a captura nomeada. Ele utiliza a sintaxe ‹(?P<name>regex)›. O nome deve ser composto por caracteres de palavra que possam ser correspondidos por ‹\w›. ‹(?P<name>› é o delimitador de abertura do grupo, e ‹)› é o delimitador de fechamento.

Os designers da classe Regex do .NET vieram com uma sintaxe própria para a captura nomeada, utilizando duas variantes intercambiáveis. ‹(?<name>regex)› imita a sintaxe do Python, menos o P. O nome deve ser composto por caracteres de palavra que possam ser correspondidos por ‹\w›. ‹(?<name>› é o delimitador de abertura do grupo, e ‹)› é o delimitador de fechamento.

Os colchetes angulares (<>), na sintaxe de captura nomeada, são irritantes quando você está programando em XML, ou escrevendo este livro em DocBook XML. Essa é a razão para a sintaxe alternativa de captura nomeada do .NET: ‹(?'name'regex)›. Os

colchetes angulares são substituídos por aspas simples. Escolha a sintaxe que for mais fácil para você. A funcionalidade delas é idêntica.

Talvez devido à popularidade do .NET em relação ao Python, a sintaxe .NET parece ser a preferida, ao ser copiada por outros desenvolvedores de bibliotecas. Perl 5.10 a possui, e o mecanismo Oniguruma, no Ruby 1.9, também.

O PCRE copiou a sintaxe do Python há muito tempo, numa época em que o Perl não suportava capturas nomeadas. PCRE 7, versão que acrescenta as novas funcionalidades do Perl 5.10, suporta tanto a sintaxe .NET quanto a sintaxe Python. Talvez, como uma prova do sucesso do PCRE, em um movimento de compatibilidade reversa, o Perl 5.10 também suporta a sintaxe do Python. No PCRE e no Perl 5.10, a funcionalidade das sintaxes .NET e Python, para captura nomeada, é idêntica.

Escolha a sintaxe que for mais útil para você. Se estiver programando em PHP, e quiser que seu código trabalhe com versões mais antigas do PHP, que, por sua vez, incorporam versões mais antigas do PCRE, use a sintaxe do Python. Se você não precisa de compatibilidade com versões mais antigas, e também trabalha com .NET ou Ruby, a sintaxe .NET facilita o processo de copiar-e-colar entre todas essas linguagens. Se estiver inseguro, utilize a sintaxe do Python no PHP/PCRE. Pessoas que recompilarem seu código com uma versão mais antiga do PCRE ficarão chateadas, caso as expressões regulares, em seu código, de repente parem de funcionar. Ao copiar uma regex para .NET ou Ruby, apagar alguns Ps é uma tarefa fácil.

A documentação para o PCRE 7 e o Perl 5.10 praticamente não mencionam a sintaxe do Python, mas ela não está, de forma alguma, obsoleta. Na verdade, nós a recomendamos para o PCRE e o PHP.

Retrorreferências nomeadas

Com a captura nomeada, temos as retrorreferências nomeadas. Assim como os grupos de captura nomeados são funcionalmente idênticos aos grupos numerados, as retrorreferências nomeadas são funcionalmente idênticas às retrorreferências numeradas. Elas são apenas mais fáceis de ler e manter.

Python usa a sintaxe ‹(?P=name)› para criar uma retrorreferência ao grupo name. Embora esta sintaxe utilize parênteses, a retrorreferência não é um grupo. Você não pode colocar nada entre o nome e o parêntese de fechamento. Uma retrorreferência ‹(?P=name)› é um token de expressão regular único, como ‹\1›.

.NET utiliza a sintaxe ‹\k<name>› e ‹\k'name'›. As duas variantes são idênticas, em termos de funcionalidade, e você pode misturá-las livremente. Um grupo nomeado criado com a sintaxe dos parênteses pode ser referenciado se usarmos a sintaxe das aspas, e vice-versa.

Recomendamos firmemente que você não misture grupos nomeados e numerados na mesma regex. Sabores diferentes seguem regras diferentes para a numeração de grupos não-nomeados que apareçam entre grupos nomeados. Perl 5.10 e Ruby 1.9 copiaram a sintaxe do .NET, mas não seguem a forma de numeração do .NET para grupos de captura ou para mistura de grupos numerados com grupos nomeados. Ao invés de tentar explicar as diferenças, simplesmente recomendamos não misturar grupos nomeados e numerados. Evite a confusão e dê um nome a todos os grupos não identificados ou transforme-os em grupos de não-captura.

Veja também:

Receitas 2.9, 2.10, 2.21, e 3.9.

2.12 Repetir parte da Regex um certo número de vezes

Problema

Crie expressões regulares que correspondam aos seguintes tipos de números:

- Um googol (um número decimal com 100 dígitos).
- Um número hexadecimal de 32 bits.
- Um número hexadecimal de 32 bits com um sufixo opcional `h`.
- Um número de ponto flutuante com uma parte inteira opcional, uma parte fracionária obrigatória e um expoente opcional. Cada parte permite qualquer número de dígitos.

Solução

Googol

```
\b\d{100}\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Número hexadecimal

```
\b[a-z0-9]{1,8}\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Número hexadecimal

```
\b[a-z0-9]{1,8}h?\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Número de ponto flutuante

```
\d*\.\d+(e\d+)?
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Repetição fixa

O *quantificador* ‹{*n*}›, em que *n* é um número positivo, repete o token regex anterior *n* vezes. O ‹\d{100}›, em ‹\b\d{100}\b›, corresponde a uma sequência de 100 dígitos. Você pode conseguir o mesmo resultado digitando ‹\d› 100 vezes.

‹{1}› repete o símbolo anterior uma vez, tal como seria sem qualquer quantificador. ‹ab{1}c› é a mesma regex que ‹abc›.

‹(0)› repete o símbolo anterior zero vezes, essencialmente excluindo-o da expressão regular. ‹ab{0}c› é a mesma regex que ‹ac›.

Repetição variável

No caso da *repetição variável*, usamos o quantificador ‹{*n*,*m*}›, em que *n* é um número positivo, e *m* é maior que *n*. ‹\b[a-f0-9]{1,8}\b› corresponde a um número hexadecimal de um a oito dígitos. Com a repetição variável, a ordem em que as alternativas são tentadas entra em cena. A receita 2.13 explica isso em detalhe.

Se *n* e *m* forem iguais, temos uma repetição fixa. ‹\b\d{100,100}\b› é a mesma regex que ‹\b\d{100}\b›.

Repetição Infinita

O quantificador ‹{*n*,}›, em que *n* é um número positivo, permite a *repetição infinita*. Essencialmente, a repetição infinita é uma repetição variável sem um limite superior.

‹\d{1,}› corresponde a um ou mais dígitos, e ‹\d+› faz o mesmo. Um sinal de soma depois de um token de expressão regular que não seja um quantificador significa “um ou mais”. A receita 2.13 mostra o significado de um sinal de soma após um quantificador.

‹\d{0,}› corresponde a zero ou mais dígitos, e ‹\d*› faz o mesmo. O asterisco significa sempre “zero ou mais”. Além de permitir a repetição infinita, ‹{0,}› e o asterisco tornam o token anterior opcional.

Tornando algo opcional

Se utilizarmos a repetição variável com *n* ajustado para zero, estamos efetivamente tornando opcional o token que precede o quantificador. ‹h{0,1}› corresponde a ‹h› uma vez, ou não corresponde a ele. Se não houver um h, ‹h{0,1}› resultará em uma correspondência de comprimento zero. Se você utilizar ‹h{0,1}› como uma expressão regular por si só, ela encontrará uma correspondência de comprimento zero antes de cada caractere, no texto de assunto, que não seja um h. Cada h resultará em uma correspondência de um caractere (o h).

‹h?› faz o mesmo que ‹h{0,1}›. Um ponto de interrogação após um token de expressão regular válido e completo, que não seja um quantificador, significa “zero ou uma vez”. A receita seguinte mostra o significado de um ponto de interrogação depois de um quantificador.

Um ponto de interrogação, ou qualquer outro quantificador após um parêntese de abertura, constitui um erro de sintaxe. Perl e os sabores que o copiam usam este procedimento para adicionar “extensões Perl” à sintaxe da expressão regular. Receitas anteriores mostram grupos de não-captura e grupos de captura nomeados que usam um ponto de interrogação após um parêntese de abertura como parte de sua sintaxe. Essas interrogações não são quantificadores; elas apenas fazem parte da sintaxe para grupos de não-captura e grupos de captura nomeados. Receitas posteriores mostrarão mais estilos de grupos que usam a sintaxe ‹(?›.

Grupos de repetição

Se você colocar um quantificador após o parêntese de fechamento de um grupo, o grupo inteiro será repetido. ‹(?:abc){3}› é o mesmo que ‹abcabcabc›.

Quantificadores podem ser aninhados. ‹(e\d+)?› corresponde a um e, seguido por um ou mais dígitos, ou gera uma correspondência de comprimento zero. Em nossa expressão regular de ponto flutuante, isso forma o expoente opcional.

Grupos de captura podem ser repetidos. Como explicado na receita 2.9, a correspondência do grupo é capturada sempre que o mecanismo sai do grupo, substituindo qualquer texto previamente correspondido pelo grupo. ‹(\d\d){1,3}› corresponde a uma sequência de dois, quatro ou seis dígitos. O mecanismo sai do grupo três

vezes. Quando essa regex corresponde a 123456, o grupo de captura armazena 56, porque 56 foi armazenado pela última iteração do grupo. As outras duas correspondências do grupo, 12 e 34, não poderão ser recuperadas.

‹(\d\d){3}› captura o mesmo texto que ‹\d\d\d\d(\d\d)›. Se quiser que o grupo capture dois, quatro ou seis dígitos, em vez de apenas os dois últimos, você terá de colocar o grupo de captura em torno do quantificador, ao invés de repetir o grupo de captura: ‹((?:\d\d){1,3})›. Aqui, usamos um grupo de não-captura para assumir a função de agrupamento do grupo de captura. Também poderíamos ter usado dois grupos de captura: ‹((\d\d){1,3})›. Quando esta última regex corresponde a 123456 ‹\1› armazena 123456, e ‹\2› armazena 56.

O mecanismo de expressão regular do .NET é o único que permite recuperar todas as iterações de um grupo de captura repetido. Se você consultar diretamente a propriedade Value do grupo, que retorna uma string, você vai obter 56, como em qualquer outro mecanismo de expressão regular. Retrorreferências na expressão regular e no texto de substituição também substituem 56, mas se você usar a propriedade CaptureCollection do grupo, obterá uma pilha contendo 56, 34 e 12.

Veja também:

Receitas 2.9, 2.13, 2.14.

2.13 Escolher entre repetição mínima ou máxima

Problema

Corresponder a um par de tags XHTML ‹p› e ‹/p›, e o texto entre elas. O texto entre as tags pode incluir outras tags XHTML.

Solução

```
<p>.*?</p>
```

Opções Regex: O ponto corresponde a quebras de linha

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Todos os quantificadores discutidos na receita 2.12 são *mesquinhos*, ou seja, eles tentam repetir tantas vezes quanto possível, retornando o controle somente quando obrigados, para permitir que o restante da expressão regular corresponda.

Isso pode dificultar a paridade das tags em XHTML (uma versão do XML e, portanto, exige que cada tag de abertura seja acompanhada por uma tag de fechamento). Considere o seguinte trecho simples em XHTML:

```
<p>
The very <em>first</em> task is to find the beginning of a paragraph. </p>
<p>
Then you have to find the end of the paragraph
</p>
```

Existem duas tags de abertura e duas tags de fechamento no trecho. Você deseja corresponder o primeiro ao primeiro , porque eles marcam um parágrafo único. Note que este parágrafo contém uma tag aninhada , por isso a expressão regular não pode, simplesmente, parar quando encontrar um caractere <.

Dê uma olhada em uma solução incorreta para o problema nesta receita:

```
<p>.*</p>
```

Opções Regex: O ponto corresponde a quebras de linha

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

A única diferença é que nesta solução incorreta falta o ponto de interrogação extra após o asterisco. A solução incorreta utiliza o mesmo asterisco mesquinho explicado na receita 2.12.

Após corresponder à primeira tag no assunto, o mecanismo chega em ‹.*›. O ponto corresponde a qualquer caractere, incluindo quebras de linha. O asterisco repete-o zero vezes ou mais. O

asterisco é mesquinho e, assim, ‹.*› corresponde a tudo, até o final do texto de assunto. Deixe-me dizer isso novamente: ‹.*› come o seu arquivo XHTML inteiro, começando com o primeiro parágrafo.

Quando o ‹.*› fica com a barriga cheia, o mecanismo tenta corresponder ao ‹<› no final do texto de assunto. E ele falha. Mas esse não é o fim da história: o mecanismo de expressão regular *retrocede (backtracks)*.

O asterisco prefere pegar tanto texto quanto possível, mas também fica perfeitamente satisfeito quando corresponde a nada (zero repetição). Após cada repetição de um quantificador além de seu mínimo, a expressão regular armazena uma posição de retrocesso (backtracking). Estas são as posições para as quais o mecanismo pode voltar, no caso de parte da regex após o quantificador falhar.

Ao ‹<› falhar, o mecanismo retrocede, fazendo o ‹.*› desistir de um caractere de sua correspondência. Então, ‹<› é tentado novamente no último caractere, no arquivo. Se ele falhar novamente, o mecanismo retorna, mais uma vez tentando ‹<› no penúltimo caractere do arquivo. Esse processo continua até que ‹<› obtenha êxito. Se ‹<› não obtiver êxito, o ‹.*› eventualmente ficará sem posições de retrocesso, e a tentativa global falhará.

Se ‹<› corresponder em algum ponto durante o retrocesso, ‹/› é tentado. Se ‹/› falhar, o mecanismo retrocede novamente. Isso se repete até que ‹ › possa ser correspondido totalmente.

Então, qual é o problema? Como o asterisco é mesquinho, a expressão regular incorreta corresponde a tudo, desde o primeiro do arquivo XHTML até o último . No entanto, para corresponder corretamente a um parágrafo XHTML, precisamos corresponder o primeiro ao primeiro que o segue.

É onde os quantificadores *preguiçosos* (lazy) entram em cena. Você pode tornar qualquer quantificador preguiçoso inserindo um ponto de interrogação depois dele: ‹*?›, ‹+?›, ‹??›, e ‹{7,42}?› são todos quantificadores preguiçosos.

Quantificadores preguiçosos também retrocedem, mas no sentido oposto. Um quantificador preguiçoso repete o mínimo de vezes necessário, armazena uma posição de retrocesso e permite que a regex continue. Se o restante da expressão falhar, e o mecanismo retroceder, o quantificador preguiçoso repete mais uma vez. Se a regex continuar retrocedendo, o quantificador expandirá até seu número máximo de repetições, ou até que o token de expressão regular repetido por ele falhe na correspondência.

‹ .*? › usa um quantificador preguiçoso para corresponder corretamente a um parágrafo XHTML. Quando ‹ › corresponde, ‹.*?›, preguiçoso como é, inicialmente não faz nada, a não ser procrastinar. Se ‹ › ocorrer imediatamente após , um parágrafo vazio será correspondido. Caso contrário, o mecanismo retrocede para ‹.*?›, que corresponde a um caractere. Se após isso ‹ › falhar, ‹.*?› corresponderá ao caractere seguinte. O procedimento continua até que ‹ › obtenha êxito, ou que ‹.*?› falhe ao expandir. Como o ponto corresponde a tudo, a falha não ocorrerá até que ‹.*?› tenha correspondido a tudo, até o final do arquivo XHTML.

Os quantificadores ‹*› e ‹*?› permitem as mesmas correspondências de expressões regulares. A única diferença é a ordem na qual as correspondências possíveis são tentadas. O quantificador guloso encontrará a mais longa correspondência possível. O quantificador preguiçoso vai encontrar a correspondência mais curta possível.

Se puder, a melhor solução é certificar-se de que exista apenas uma correspondência possível. As expressões regulares de correspondência de números, na receita 2.12, ainda corresponderão aos mesmos números, caso você torne todos os seus quantificadores preguiçosos. O motivo é que as partes dessas expressões regulares que possuem quantificadores e as partes que as seguem são mutuamente exclusivas. ‹\d› corresponde a um dígito, e ‹\b› corresponde após ‹\d› somente se o próximo caractere não for um dígito (ou letra).

Para ajudar na compreensão do funcionamento das repetições gulosa e preguiçosa, compare-as a partir da forma como ‹\d+\b› e ‹\d+?\b› agem em alguns textos de assunto diferentes. As versões gulosa e preguiçosa produzem os mesmos resultados, mas testam o texto de assunto em uma ordem diferente.

Se usarmos ‹\d+\b› em 1234, ‹\d+› corresponderá a todos os dígitos. Em seguida, ‹\b› corresponde, e uma correspondência global é encontrada. Se usarmos ‹\d+?\b›, ‹\d+?› primeiramente corresponde apenas a 1. ‹\b› falha entre 1 e 2. ‹\d+?› expande até 12, e ‹\b› falha novamente. Isso continua até ‹\d+?› corresponder a 1234, e ‹\b› obter sucesso.

Se nosso texto de assunto for 1234X, a primeira regex, ‹\d+\b›, ainda tem ‹\d+›, que corresponde a 1234. Porém, então, ‹\b› falha. ‹\d+› retrocede para 123. ‹\b› ainda falha. O erro continua até que ‹\d+› retroceda para seu mínimo, o caractere 1; ‹\b› ainda vai falhar. Toda tentativa de correspondência global falha.

Se usarmos ‹\d+?\b› em 1234X, ‹\d+?› primeiramente corresponde apenas a 1. ‹\b› falha entre 1 e 2. ‹\d+?› expande até 12. ‹\b› ainda falha. Isso continua até que ‹\d+?› corresponda a 1234, e ‹\b›, ainda assim, irá falhar. O mecanismo de expressão regular tenta expandir ‹\d+?› mais uma vez, mas ‹\d› não corresponde a X. A tentativa de correspondência global falha.

Se colocarmos ‹\d+› entre extremidades de palavra, ele deverá corresponder a todos os dígitos no texto de assunto, ou falhará. Tornar o quantificador preguiçoso não afeta a correspondência final da expressão regular, ou sua eventual falha. Na verdade, seria melhor que ‹\b\d+\b› não realizasse qualquer tipo de retrocesso. A próxima receita explica como você pode usar um quantificador possessivo ‹\b\d++\b› para conseguir isso, pelo menos em alguns sabores.

Veja também:

Receitas 2.8, 2.9, 2.12, 2.14, e 2.15.

2.14 Eliminar os retrocessos desnecessários

Problema

A receita anterior explica a diferença entre os quantificadores mesquinhos e os preguiçosos, e como eles retrocedem. Em algumas situações, este retrocesso é desnecessário.

‹\b\d+\b› utiliza um quantificador mesquinho, e ‹\b\d+?\b› utiliza um quantificador preguiçoso. Ambos encontraram a mesma coisa: um número inteiro. Dado o mesmo texto de assunto, ambos encontrarão as mesmas correspondências. Qualquer retrocesso realizado é desnecessário. Reescreva essa expressão regular, a fim de eliminar explicitamente todos os retrocessos, tornando a expressão regular mais eficiente.

Solução

```
\b\d++\b
```

Opções Regex: Nenhuma

Sabores Regex: Java, PCRE, Perl 5.10, Ruby 1.9

A solução mais fácil é utilizar um quantificador possessivo. Porém, ele é suportado apenas em alguns sabores regex recentes.

```
\b(?>\d+)\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Ruby

Um grupo atômico fornece exatamente a mesma funcionalidade, utilizando uma sintaxe um pouco menos legível. O suporte a agrupamentos atômicos é mais difundido do que o suporte a quantificadores possessivos.

JavaScript e Python não suportam quantificadores possessivos, nem agrupamentos atômicos. Não há nenhuma maneira de eliminar o retrocesso desnecessário nesses dois sabores regex.

Discussão

Um *quantificador possessivo* é semelhante a um quantificador guloso: ele tenta repetir tantas vezes quanto possível. A diferença é que um quantificador possessivo nunca retrocede, nem mesmo quando retroceder é a única maneira do restante da expressão regular corresponder. Quantificadores possessivos não guardam posições de retrocesso.

Você pode tornar qualquer quantificador possessivo colocando um sinal de adição logo após ele. Por exemplo, ‹*+›, ‹++›, ‹?+›, e ‹{7,42}+› são todos quantificadores possessivos.

Quantificadores possessivos são suportados pelo Java 4 e versões posteriores, ou seja, desde a primeira versão Java a incluir o pacote java.util.regex. Todas as versões do PCRE discutidas neste livro (versões de 4 a 7) suportam quantificadores possessivos. Perl os suporta a partir do Perl 5.10. As expressões regulares clássicas do Ruby não suportam quantificadores possessivos, mas o mecanismo Oniguruma, o padrão no Ruby 1.9, lhes dá suporte.

Empacotar um quantificador mesquinho dentro de um *grupo atômico* tem o mesmo efeito de usar um quantificador possessivo. Quando o mecanismo de expressão regular sai do grupo atômico, todas as posições de retrocesso lembradas pelos quantificadores, e as alternâncias dentro do grupo, são jogadas fora. A sintaxe é ‹(?>*regex*)›, em que ‹*regex*› é qualquer expressão regular. Um grupo atômico é essencialmente um grupo de não-captura, com a função extra de se recusar a retroceder. O ponto de interrogação não é um quantificador; o delimitador de abertura consiste simplesmente em três caracteres ‹(?>›.

Quando você aplica a regex ‹\b\d++\b› (possessiva) em 123abc 456, ‹\b› corresponderá no início do assunto, e ‹\d++› corresponderá a 123. Até agora, isso não é diferente do que a regex ‹\b\d+\b› (mesquinha) faria. Mas, então, o segundo ‹\b› falha ao corresponder entre 3 e a.

O quantificador possessivo não armazenou nenhuma posição de

retrocesso. Dado que não existem outros quantificadores, nem alternâncias nesta expressão regular, não há outras opções a tentar, quando a segunda extremidade de palavra falha. O mecanismo de expressão regular imediatamente declara fracasso na tentativa de corresponder a partir de 1.

O mecanismo de expressão regular tenta a regex, iniciando nas próximas posições de caracteres na string; a utilização de um quantificador possessivo não vai mudar esse fato. Se a regex precisa corresponder a todo o assunto, utilize âncoras, como discutido na receita 2.5. Eventualmente, o mecanismo de expressão regular tentará a regex iniciando no 4, e encontrará a correspondência em 456.

A diferença para o quantificador mesquinho é que, quando o segundo ‹\b› falhar durante a primeira tentativa de correspondência, ele vai retroceder. Então, o mecanismo de expressão regular (desnecessariamente) testa ‹\b› entre 2 e 3 e entre 1 e 2.

O processo de correspondência que utiliza o agrupamento atômico é essencialmente o mesmo. Quando você aplicar a regex ‹\b(?>\d+)\b› (possessiva) em 123abc 456, a extremidade de palavra corresponde ao início do assunto. O mecanismo de expressão regular entra no grupo atômico, e ‹\d+› corresponde a 123.

Agora, o mecanismo sai do grupo atômico. Neste ponto, as posições de retrocesso lembradas por ‹\d+› são jogadas para fora. Quando o segundo ‹\b› falhar, o mecanismo de expressão regular fica sem outras opções, fazendo com que a tentativa de correspondência falhe imediatamente. Tal como ocorre com o quantificador possessivo, eventualmente 456 será encontrado.

Nós descrevemos o quantificador possessivo como o que não se lembra das posições de retrocesso, e o grupo atômico como o que as joga para fora. Isso facilita a compreensão do processo de correspondência, mas não fique muito preocupado com essa diferença, porque ela pode nem mesmo existir no sabor regex com o qual você estiver trabalhando. Em diversos sabores, ‹x++› é

meramente açúcar sintático para ‹(?>x+)›, e ambos são implementados exatamente da mesma maneira. Se o mecanismo não se lembra das posições de retrocesso, ou as joga fora mais tarde, é irrelevante para o resultado final.

O ponto em que quantificadores possessivos e agrupamentos atômicos diferem é que um quantificador possessivo só se aplica a um único token de expressão regular, enquanto um grupo atômico pode envolver a expressão regular como um todo.

‹\w++\d++› e ‹(?>\w+\d+)› definitivamente não são iguais. ‹\w++\d++›, o mesmo que ‹(?>\w+)(?>\d+)›, não corresponderá a abc123. ‹\w++› corresponde a abc123 integralmente. Em seguida, o mecanismo de expressão regular tenta ‹\d++› no final do texto de assunto. Dado que não existem outros caracteres que possam ser correspondidos, ‹\d++› falha. Sem qualquer posição de retrocesso memorizada, a tentativa de correspondência falha.

‹(?>\w+\d+)› tem dois quantificadores mesquinhos dentro do mesmo grupo atômico. Dentro do grupo atômico, o retrocesso ocorre normalmente. Posições de retrocesso são jogadas fora apenas quando o mecanismo sai de todo o grupo. Quando o assunto é abc123, ‹\w+› corresponde a abc123. O quantificador mesquinho lembra-se das posições de retrocesso. Quando ‹\d+› não corresponde, ‹\w+› abandona um caractere. ‹\d+›, então, corresponde a 3. Agora, o mecanismo sai do grupo atômico, jogando para fora todas as posições de retrocesso lembradas por ‹\w+› e ‹\d+›. Como o final da regex foi atingido, isso realmente não faz muita diferença. Uma correspondência global é encontrada.

Se o final não foi alcançado, como em ‹(?>\w+\d+)\d+›, estaríamos na mesma situação de ‹\w++\d++›. O segundo ‹\d+› não tem nada para corresponder no final do assunto. Como as posições de retrocesso foram jogadas fora, o mecanismo de expressão regular só pode declarar uma falha.

Quantificadores possessivos e agrupamentos atômicos não apenas otimizam as expressões regulares. Eles podem alterar os resultados

encontrados por uma expressão regular, eliminando aqueles que seriam alcançados por meio de retrocessos.

Esta receita mostrou como utilizar os quantificadores possessivos e os agrupamentos atômicos para fazer pequenas otimizações que não conseguem sequer mostrar alguma diferença em benchmarks. A próxima receita mostrará como o agrupamento atômico pode fazer uma diferença dramática.

Veja também:

Receitas 2.12 e 2.15.

2.15 Prevenir repetições descontroladas

Problema

Utilize uma única expressão regular para corresponder a um arquivo HTML completo, verificando se as tags html, head, title e body estão corretamente aninhadas. A expressão regular deve falhar em arquivos HTML que não possuam as tags apropriadas.

Solução

```
<html>(?>.*?<head>)(?>.*?<title>)(?>.*?</title>)↵
(?>.*?</head>)(?>.*?<body[^>]*>)(?>.*?</body>).*?</html>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, ponto corresponde a quebras de linha

Sabores Regex: .NET, Java, PCRE, Perl, Ruby

JavaScript e Python não suportam agrupamento atômico. Não há nenhuma maneira de eliminar o retrocesso desnecessário nestes dois sabores regex. Ao programar em JavaScript ou Python, você poderá resolver esse problema fazendo uma pesquisa de texto literal para cada uma das tags, uma a uma, procurando a próxima tag no restante do texto de assunto, após a última encontrada.

Discussão

A solução adequada para esse problema poderá ser mais facilmente compreendida se partirmos desta solução ingênua:

```
<html>.*?<head>.*?<title>.*?</title>↲
.*?</head>.*?<body[^>]*>.*?</body>.*?</html>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, ponto corresponde a quebras de linhas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Quando você testar esta regex em um arquivo HTML apropriado, ela funcionará perfeitamente bem. ‹.*?› pula qualquer coisa, porque ativamos a opção “ponto corresponde a quebras de linha”. O asterisco preguiçoso garante que a regex siga em frente apenas um caractere por vez, verificando se a próxima tag pode ser correspondida. As receitas 2.4 e 2.13 explicam tudo isso.

Mas esta regex coloca-o em apuros quando precisa lidar com um texto de assunto que não possui todas as tags HTML. O pior caso ocorre quando </html> está faltando.

Imagine que o mecanismo de expressão regular tenha correspondido a todas as tags anteriores, e agora está ocupado expandindo o último ‹.*?›. Como ‹</html>› nunca será correspondido, ‹.*?› expande até o final do arquivo. Quando já não puder expandir, falhará.

Porém, esse não é o fim da história. Os outros seis ‹.*?› memorizaram uma posição de retrocesso que os permite expandir ainda mais. Quando o último ‹.*?› falha, o anterior expande, correspondendo gradualmente a </body>. Esse mesmo texto foi previamente correspondido pela literal ‹</body>› na regex. Este ‹.*?› também expandirá até o final do arquivo, assim como todos os pontos preguiçosos anteriores. Quando o primeiro chegar ao final do arquivo, o mecanismo de expressão regular irá declarar a falha.

Na pior das hipóteses, essa expressão regular possui uma complexidade de $O(n^7)$, o comprimento do texto de assunto à sétima potência. Existem sete pontos preguiçosos que podem,

potencialmente, expandir até o final do arquivo. Se o arquivo tiver o dobro do tamanho, a regex poderá precisar de até 128 vezes mais passos, para descobrir que não há correspondência.

Chamamos isso de *retrocesso catastrófico*. Ocorre tanto retrocesso que a regex leva uma eternidade ou trava a sua aplicação. Algumas implementações regex são inteligentes, e abortarão as tentativas de correspondências descontroladas logo no início, mas, mesmo assim, a regex ainda vai prejudicar o desempenho do aplicativo.

Retrocesso catastrófico é um exemplo de fenômeno conhecido como *explosões combinatórias*, em que várias condições ortogonais entrecruzam-se e todas as combinações têm de ser tentadas. Pode-se dizer, também, que a regex é um *produto Cartesiano* dos vários operadores de repetição.

A solução é utilizar o agrupamento atômico para evitar o retrocesso desnecessário. Não há nenhuma necessidade para o sexto ‹.*?› expandir depois que ‹</body>› correspondeu. Se ‹</html>› falhar, expandir o sexto ponto preguiçoso não irá, magicamente, produzir uma tag html de fechamento.

Para fazer um token de expressão regular quantificado parar, quando o delimitador seguinte corresponder, insira a parte quantificada da regex e o delimitador juntos em um grupo atômico: ‹(?>.*?</body>)›. Agora, o mecanismo de expressão regular jogará fora todas as posições correspondidas por ‹.*?</body>› quando ‹</body>› for encontrado. Se ‹</html>› falhar mais tarde, o mecanismo de expressão regular esquecerá de ‹.*?</body>›, e nenhuma nova expansão ocorrerá.

Se fizermos o mesmo para todos os outros ‹.*?› na regex, nenhum deles expandirá ainda mais. Embora ainda existam sete pontos preguiçosos na regex, eles nunca irão se sobrepor. Isso reduz a complexidade da expressão regular para O(*n*), linear com respeito à extensão do texto de assunto. Uma expressão regular nunca pode ser mais eficiente do que isso.

Variações

Se você realmente quiser ver o retrocesso catastrófico em ação,

tente ‹(x+x+)+y› em xxxxxxxxxx. Se ela falhar rapidamente, adicione um x no assunto. Repita este procedimento até a regex começar a levar muito tempo para corresponder ou até sua aplicação falhar. Não precisará de muitos caracteres x, a menos que você esteja usando Perl.

Dos sabores regex discutidos neste livro, apenas o Perl é capaz de detectar que a expressão regular é muito complexa para, em seguida, abortar a tentativa de correspondência sem travar.

A complexidade desta regex é $O(2^n)$. Quando ‹y› não corresponde, o mecanismo de expressão regular vai tentar todas as permutações possíveis de repetição de cada ‹x+› e do grupo que os contém. Por exemplo, uma permutação, lá no meio da tentativa de correspondência, seria o primeiro ‹x+› correspondendo a xxx, o segundo ‹x+› correspondendo a x e, em seguida, o grupo sendo repetido mais três vezes, com cada ‹x+› correspondendo a x. Com 10 caracteres x, temos 1.024 permutações. Se aumentarmos o número de caracteres x para 32, ficaremos com mais de 4 bilhões de permutações, o que certamente causará falta de memória em qualquer mecanismo de expressão regular, a menos que ele tenha um interruptor de segurança que o permita desistir, indicando que a expressão regular é muito complicada.

Neste caso, essa expressão regular absurda é facilmente reescrita como ‹xx+y›, que encontra exatamente as mesmas correspondências em tempo linear. Na prática, a solução pode não ser tão óbvia, no caso de regexes mais complicadas.

Basicamente, você precisa tomar cuidado quando duas ou mais partes da expressão regular puderem corresponder ao mesmo texto. Nestes casos, você pode precisar de um agrupamento atômico, para se certificar de que o mecanismo de expressão regular não tente, de todas as formas possíveis, dividir o texto de assunto entre duas partes da expressão regular. Sempre teste a sua regex (longa) em assuntos-testes contendo texto que possa ser parcialmente, mas não totalmente, correspondido pela regex.

Veja também:

Receitas 2.13 e 2.14.

2.16 Testar uma correspondência sem acrescentá-la à correspondência global

Problema

Encontre qualquer palavra que aconteça entre um par de tags HTML (negrito), sem incluir as tags na correspondência da regex. Por exemplo, se o assunto for My cat is furry, a única correspondência válida deverá ser cat.

Solução

```
(?<=<b>)\w+(?=</b>)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby 1.9

JavaScript e Ruby 1.8 suportam o lookahead ‹(?=)›, mas não o lookbehind ‹(?<=)›.

Discussão

Lookaround

Os quatro tipos de grupos *lookaround* (“olhar em volta”) suportados por sabores regex modernos possuem a propriedade especial de abandonar o texto correspondido à parte da regex interna ao lookaround. Essencialmente, o lookaround verifica se determinado texto pode ser correspondido, sem, na realidade, combiná-lo.

O lookaround que olha para trás é chamado *lookbehind*. Esta é a única construção de expressão regular que percorre o texto da direita para a esquerda, e não da esquerda para a direita. A sintaxe para o *lookbehind positivo* é ‹(?<=*text*)›. Os quatro caracteres ‹(?<=›

formam o delimitador de abertura. O que você pode colocar dentro do lookbehind, aqui representado por ‹*text*›, varia entre os sabores de expressões regulares. Mas um texto literal simples, como ‹(?<=)›, sempre funciona.

O lookbehind verifica se o texto dentro dele ocorre imediatamente à esquerda da posição que o mecanismo de expressão regular alcançou. Se você corresponder ‹(?<=)› em My cat is furry, o lookbehind falhará na correspondência, até que a expressão regular tente corresponder à letra "c" no assunto. O mecanismo de expressão regular, então, entra no grupo lookbehind, dizendo-lhe para olhar para a esquerda. ‹› corresponde à esquerda de c. O mecanismo sai do lookbehind, neste momento, e descarta qualquer texto correspondido pelo lookbehind em relação à tentativa global de correspondência. Em outras palavras, a correspondência em andamento volta ao ponto em que estava, antes do mecanismo ter entrado no lookbehind. Neste caso, a correspondência em andamento é a correspondência de comprimento zero, antes do c na string de assunto. O lookbehind apenas testa, ou afirma que ‹› pode ser correspondido; ele não corresponde de verdade. Construções lookaround são, portanto, chamadas de *afirmações de comprimento zero*.

Após o lookbehind ter correspondido, a classe de caracteres abreviada ‹\w+› tenta corresponder a um ou mais caracteres de palavra. Ela corresponde a cat. O ‹\w+› não está dentro de qualquer tipo de lookaround ou grupo, e assim ele corresponde ao texto cat, normalmente. Dizemos que ‹\w+› corresponde a, e *consome*, cat enquanto o lookaround pode corresponder a alguma coisa, mas nunca poderá consumir nada.

O lookaround que olha para frente, no mesmo sentido em que a expressão regular normalmente percorre o texto, é chamado de *lookahead*. O lookahead é igualmente suportado por todos os sabores regex deste livro. A sintaxe para o *lookahead positivo* é ‹(?=*regex*)›. Os três caracteres ‹(?=› formam o delimitador de abertura do

grupo. Tudo o que você pode usar em uma expressão regular poderá ser usado dentro do lookahead, aqui representado por ‹*regex*›.

Quando o ‹\w+› em ‹(?<=)\w+(?=)› corresponde a cat em My cat is furry, o mecanismo de expressão regular entra no lookahead. O único comportamento especial para o lookahead, neste momento, é que o mecanismo de expressão regular lembra qual parte do texto ele correspondeu até então, associando-o com o lookahead. Então, ‹› é correspondido normalmente. Agora, o mecanismo de expressão regular sai do lookahead. A regex dentro do lookahead corresponde, então o próprio lookahead corresponde. O mecanismo de expressão regular descarta o texto correspondido pelo lookahead, restaurando a correspondência em andamento, que ele memorizou ao entrar no lookahead. Nossa correspondência global em andamento retorna a cat. Uma vez que este é também o fim da nossa expressão regular, cat torna-se o resultado final da correspondência.

Lookaround negativo

‹(?!*regex*)›, com um ponto de exclamação, em vez de um sinal de igual, é um *lookahead negativo*. O lookahead negativo funciona como o positivo, exceto pelo fato de que, enquanto o lookahead positivo corresponde quando a regex dentro do lookahead também corresponde, o lookahead negativo corresponde quando a regex dentro do lookahead não corresponde.

O processo de correspondência é exatamente o mesmo. O mecanismo salva a correspondência em andamento ao entrar no lookahead negativo e tenta corresponder normalmente à regex dentro do lookahead. Se a sub-regex corresponder, o lookahead falha, e o mecanismo de expressão regular retrocede. Se a sub-regex não corresponder, o mecanismo restaura a correspondência em andamento e prossegue com o restante da regex.

Da mesma forma, (?<!*text*) é um *lookbehind negativo*. O lookbehind

negativo corresponde quando nenhuma das alternativas dentro do lookbehind puder ser encontrada olhando para trás, a partir da posição que a regex alcançou no texto de assunto.

Diferentes níveis de lookbehind

Lookahead é fácil. Todos os sabores regex discutidos neste livro permitem que você coloque uma expressão regular completa dentro do lookahead. Tudo que você pode usar em uma expressão regular pode ser usado dentro do lookahead. Você pode até mesmo aninhar grupos de lookahead e lookbehind dentro de um lookahead. Seu cérebro pode começar a pirar, mas o mecanismo de expressão regular vai gerenciar tudo direitinho.

Com o lookbehind a história é diferente. Softwares de expressão regular sempre foram projetados para pesquisar o texto somente da esquerda para a direita. Fazer a busca ao contrário requer um pouco de astúcia: o mecanismo de expressão regular determina a quantidade de caracteres colocados dentro do lookbehind, recua aquele tanto de caracteres e, em seguida, compara o texto no lookbehind com o texto no assunto, da esquerda para a direita.

Por esse motivo, as implementações mais antigas permitiam somente textos literais de comprimento fixo dentro do lookbehind. Perl, Python, e Ruby 1.9 levaram isso um passo adiante, permitindo que você utilize alternância e classes de caracteres, a fim de colocar várias strings literais de comprimento fixo dentro do lookbehind. Algo como ‹(?<=one|two|three|forty-two|gr[ae]y)› é tudo que eles podem tratar.

Internamente, Perl, Python e Ruby 1.9 expandem isso em seis testes de lookbehind. Primeiro, eles recuam três caracteres para testar ‹one|two›, então quatro caracteres para testar ‹gray|grey›, em seguida cinco para testar ‹three› e, finalmente, nove para testar ‹forty-two›.

PCRE e Java levam o lookbehind um passo adiante. Eles permitem qualquer expressão regular de comprimento finito dentro do lookbehind. Isso significa que você pode usar qualquer coisa, exceto

os quantificadores infinitos ‹*›, ‹+›, e ‹{42,}› dentro do lookbehind. Internamente, PCRE e Java calculam o comprimento mínimo e máximo do texto que poderia ser possivelmente correspondido pela parte da regex no lookbehind. Então, eles recuam o número mínimo de caracteres e aplicam a regex no lookbehind, da esquerda para a direita. Se falhar, recuam mais um caractere, e tentam novamente até que o lookbehind corresponda ou o número máximo de caracteres tenha sido tentado.

Tudo isso soa bastante ineficiente, e realmente o é. O lookbehind é muito conveniente, mas não vai quebrar recordes de velocidade. Mais tarde, apresentaremos uma solução para o Javascript e o Ruby 1.8, que não suportam lookbehind. Essa solução é, de fato, muito mais eficiente do que usar lookbehind.

O mecanismo de expressão regular do .NET é o único no mundo[3] que pode realmente aplicar uma expressão regular completa, da direita para a esquerda. O .NET permite que você use qualquer coisa dentro do lookbehind, e ele vai realmente aplicar a expressão regular da direita para a esquerda. Tanto a expressão regular, dentro do lookbehind, quanto o texto de assunto são varridos da direita para a esquerda.

Correspondendo ao mesmo texto duas vezes

Se você usar um lookbehind no início da regex, ou um lookahead no final dela, você está exigindo o aparecimento de algo antes ou depois da correspondência da regex, sem incluir esse algo na correspondência. Se você usar um lookaround no meio da expressão regular, poderá aplicar vários testes ao mesmo texto.

Em “Funcionalidades específicas de cada sabor” (subseção da receita 2.3), mostramos como usar a subtração de classe de caracteres para corresponder a um dígito tailandês. Só o .NET e o Java suportam a subtração de classe de caracteres.

Um caractere é um dígito tailandês se for, ao mesmo tempo, um caractere Tailandês (qualquer tipo) e um dígito (qualquer alfabeto).

Com o lookahead, você pode testar ambos os requisitos no mesmo caractere:

```
(?=\p{Thai})\p{N}
```

Opções Regex: Nenhuma

Sabores Regex: PCRE, Perl, Ruby 1.9

Esta regex funciona apenas com os três sabores que suportam alfabetos Unicode, como explicamos na receita 2.7. Porém, o princípio de usar um lookahead para corresponder ao mesmo caractere mais de uma vez funciona com todos os sabores discutidos neste livro.

Quando o mecanismo de expressões regulares busca por ‹(?=\p{Thai})\p{N}›, ele começa entrando no lookahead em cada posição da string, onde inicia uma tentativa de correspondência. Se o caractere naquela posição não faz parte do alfabeto Tailandês (ou seja, ‹\p{Thai}› falha na tentativa de correspondência), o lookahead falha. Isso faz com que toda a tentativa de correspondência falhe, forçando o mecanismo de expressão regular a começar tudo de novo no próximo caractere.

Quando a regex atinge um caractere tailandês, ‹\p{Thai}› corresponde. Assim, o lookaround ‹(?=\p{Thai})› também corresponde. Quando o mecanismo sai do lookaround, ele restaura a correspondência em andamento. Neste caso, seria a correspondência de comprimento zero que existia antes do caractere tailandês recém-encontrado. Em seguida, vem o ‹\p{N}›. Como o lookahead descartou sua correspondência, ‹\p{N}› é comparado com o mesmo caractere que ‹\p{Thai}› já correspondeu. Se esse caractere possui a propriedade Unicode Number, ‹\p{N}› gera uma correspondência. Desde que ‹\p{N}› não esteja dentro de um lookaround, ele consome o caractere, e descobrimos o nosso dígito tailandês.

Lookaround é atômico

Quando o mecanismo de expressões regulares sai de um grupo

lookaround, ele descarta o texto correspondido pelo lookaround. Como o texto é descartado, qualquer posição de retrocesso lembrada por alternância ou por quantificadores no interior do lookaround também é descartada. Isso, efetivamente, tornou o lookahead e o lookbehind atômicos. A receita 2.15 explica grupos atômicos em detalhes.

Na maioria das situações, a natureza atômica do lookaround é irrelevante. O lookaround é apenas uma afirmativa, para verificar se a regex dentro do lookaround corresponde ou não. De quantas maneiras diferentes ele pode corresponder é irrelevante, uma vez que não consome qualquer parte do texto de assunto.

A natureza atômica entra em cena somente quando você usa grupos de captura dentro do lookahead (e lookbehind, se seu sabor regex permitir). Apesar do lookahead não consumir qualquer texto, o mecanismo de expressão regular vai lembrar quais partes do texto foram correspondidas por um grupo de captura dentro do lookahead. Se o lookahead estiver no final da regex, de fato você vai acabar com grupos de captura correspondendo a texto não correspondido pela expressão regular em si. Se o lookahead estiver no meio da regex, você pode acabar com grupos de captura correspondendo a partes sobrepostas do texto do assunto.

A única situação em que a natureza atômica do lookaround poderá alterar a correspondência global da expressão regular é quando você usa uma retrorreferência fora do lookaround para um grupo de captura criado dentro do lookaround. Considere a seguinte expressão regular:

```
(?=(\d+))\w+\1
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

À primeira vista, você pode pensar que esta regex corresponderia a 123x12. ‹\d+› capturaria 12 no primeiro grupo de captura; em seguida, ‹\w+› corresponderia a 3x e, finalmente, ‹\1› corresponderia a 12 novamente.

Mas isso nunca acontece. A expressão regular entra no lookaround e no grupo de captura. O mesquinho ‹\d+› corresponde a 123. Esta correspondência é armazenada no primeiro grupo de captura. O mecanismo, em seguida, sai do lookahead, redefinindo a correspondência em andamento para o início da string e descartando as posições de retrocesso lembradas pelo sinal de adição mesquinho, mas mantendo 123 armazenado no primeiro grupo de captura.

Agora, o mesquinho ‹\w+› é tentado no início da string. Ele vai comendo até 123x12. ‹\1›, que faz referência a 123, irá falhar no final da string. ‹\w+› retrocede um caractere. ‹\1› falha novamente. ‹\w+› mantém o retrocesso até que ele tenha desistido de tudo, exceto do primeiro 1 no assunto. ‹\1› também falha, após o primeiro 1.

O 12 final corresponderia a ‹\1›, se o mecanismo de expressão regular pudesse retornar ao lookahead e abandonar 123 a favor de 12, mas o dito mecanismo não procede dessa forma.

O mecanismo de expressão regular não tem mais posições de retrocesso para seguir. ‹\w+› retrocedeu totalmente, e o lookaround forçou ‹\d+› a desistir de suas posições de retrocesso. A tentativa de correspondência falha.

Solução sem o lookbehind

Todo esse esoterismo não tem utilidade se estiver usando Python ou JavaScript, porque neles você não pode usar o lookbehind de maneira alguma. Não há nenhuma maneira de resolver o problema com estes sabores regex, mas você pode contornar a necessidade de utilização do lookbehind usando grupos de captura. Esta solução também funciona com todos os outros sabores regex:

```
(<b>)(\w+)(?=</b>)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex : .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Em vez de usar o lookbehind, utilizamos um grupo de captura para a tag de abertura ‹<b>›. Também colocamos a parte da

correspondência que nos interessa, o ‹\w+›, em um grupo de captura.

Quando você aplica essa expressão regular em My cat is furry, a correspondência regex global será ‹b>cat. O primeiro grupo de captura armazena ‹b>, e o segundo, cat.

Se a exigência é corresponder apenas a cat (a palavra entre as tags), porque você quer extrair apenas isso do texto, você pode alcançar esse objetivo simplesmente armazenando o texto correspondido pelo segundo grupo de captura, em vez da regex global.

Se a exigência é que você deseja fazer uma pesquisa-e-substituição, substituindo apenas a palavra entre as tags, basta utilizar uma retrorreferência ao primeiro grupo de captura, para reintroduzir a tag de abertura no texto de substituição. Neste caso, você não precisa do grupo de captura, pois a tag de abertura é sempre a mesma. Mas, quando for variável, o grupo de captura introduz, de novo, exatamente o que foi correspondido. A receita 2.21 explica isso em detalhes.

Finalmente, se você realmente deseja simular um lookbehind, poderá fazê-lo com duas expressões regulares. Primeiro, pesquise por sua regex sem o lookbehind. Quando ela corresponder, copie a parte do texto de assunto, antes da correspondência, em uma nova variável de string. Faça o teste que você fez dentro do lookbehind com uma segunda regex, anexando uma âncora de final-de-string (‹\z› ou ‹$›). A âncora certifica que a correspondência da segunda regex terminará ao final da string. Uma vez que você corta a string no ponto em que a primeira regex correspondeu, tal atitude, efetivamente, coloca a segunda correspondência imediatamente à esquerda da primeira.

Em JavaScript, você poderia codificar isso assim:

```
var mainregexp = /\w+(?=<\/b>)/;
var lookbehind = /<b>$/;
if (match = mainregexp.exec("My <b>cat</b> is furry")) {
```

```
    // Encontrou uma palavra antes de uma tag de fechamento </b>
    var potentialmatch = match[0];
    var leftContext = match.input.substring(0, match.index);
    if (lookbehind.exec(leftContext)) {
      // Lookbehind correspondeu: potentialmatch ocorre entre um par de tags <b>
    } else {
      // Lookbehind falhou: potentialmatch não é útil
    }
  } else {
    // Não foi possível encontrar uma palavra antes da tag de fechamento </b>
  }
```

Veja também:

Receitas 5.5 e 5.6.

2.17 Corresponder a uma de duas alternativas com base em uma condição

Problema

Crie uma expressão regular que corresponda a uma lista, separada por vírgulas, das palavras one, two e three. Cada palavra pode aparecer várias vezes, mas cada palavra deve aparecer pelo menos uma vez.

Solução

```
\b(?:(?:(one)|(two)|(three))(?:,|\b)){3,}(?(1)|(?!))(?(2)|(?!))(?(3)|(?!))
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE, Perl, Python

Java, JavaScript e Ruby não suportam condicionais. Ao programar em Java, JavaScript ou Ruby (ou qualquer outra linguagem), você pode usar a expressão regular sem condicionais, e escrever código extra, para verificar se cada um dos três grupos de captura correspondeu a alguma coisa.

```
\b(?:(?:(one)|(two)|(three))(?:,|\b)){3,}
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

.NET, PCRE, Perl e Python suportam *condicionais* utilizando grupos de captura numerados. ‹(?(1)*then*|*else*)› é uma condicional que verifica se o primeiro grupo de captura já correspondeu a alguma coisa. Se tiver correspondido, o mecanismo de expressão regular tenta corresponder a ‹*then*›. Se o grupo de captura não participou da tentativa de correspondência até agora, a parte ‹*else*› é tentada.

Os parênteses, o ponto de interrogação e a barra vertical são partes da sintaxe para a condicional. Eles não possuem seus significados usuais. Você pode usar qualquer tipo de expressão regular para as partes ‹*then*› e ‹*else*›. A única restrição é que, caso você queira usar alternância em uma das partes, terá de usar um grupo para manter as alternativas juntas. Apenas uma barra vertical é permitida diretamente na condicional.

Se quiser, poderá omitir a parte ‹*then*›, ou a parte ‹*else*›. A regex vazia sempre encontra uma correspondência de comprimento zero. A solução para esta receita usa três condicionais que possuem uma parte ‹então› vazia. Se o grupo de captura participou, a condicional simplesmente corresponde.

Um lookahead vazio negativo, ‹(?!)›, preenche a parte ‹*else*›. Como a regex vazia sempre corresponde, um lookahead negativo contendo uma regex vazia sempre falha. Assim, a condicional ‹(?(1)|(?!))› sempre falha quando o primeiro grupo de captura não corresponder a nada.

Ao colocar cada uma das três alternativas necessárias em seu próprio grupo de captura, podemos utilizar três condicionais no final da expressão regular para testar se todos os grupos de captura capturaram algo.

O .NET também suporta condicionais nomeadas. ‹(?(*name*)*then*|*else*)› verifica se o grupo de captura nomeado participou da tentativa de

correspondência até o momento.

Para entender melhor como trabalham as condicionais, vamos examinar a expressão regular ‹(a)?b(?(1)c|d)›. Essencialmente, esta é uma maneira complicada de escrever ‹abc|bd›.

Se o texto de assunto começa com um a, ele é capturado no primeiro grupo de captura. Caso contrário, o primeiro grupo de captura não participa da tentativa de correspondência. É importante que o ponto de interrogação esteja fora do grupo de captura, pois isso torna todo o grupo opcional. Se não houver um a, o grupo não é repetido, e nunca terá a chance de capturar coisa alguma. Ele não pode capturar uma string de comprimento zero.

Se você usar ‹(a?)›, o grupo sempre participa da tentativa de correspondência. Não há nenhum quantificador após o grupo, por isso ele é repetido exatamente uma vez. O grupo captura a, caso contrário, não capturará nada.

Independentemente de ‹a› ser correspondido, o próximo token é ‹b›. A condicional vem em seguida. Se o grupo de captura participou da tentativa de correspondência, mesmo que ele tenha capturado a string de comprimento zero (o que não é possível aqui), ‹c› será tentado. Caso contrário, será a vez de ‹d›.

Em inglês: ‹(a)?b(?(1)c|d)› corresponde a ab, seguido de um c, ou corresponde a b, seguido por um d.

No .NET, PCRE, e Perl, mas não no Python, condicionais também podem usar lookaround. ‹(?(?=*if*)*then*|*else*)› primeiro testa ‹(?=*if*)› como um lookahead normal. A receita 2.16 explica como isso funciona. Se o lookaround tiver êxito, a parte ‹*then*› é tentada. Se não, a parte ‹*else*› terá vez. Como o lookaround tem comprimento zero, as expressões regulares ‹*then*› e ‹*else*› são tentadas na mesma posição no texto de assunto em que ‹*if*› correspondeu ou falhou.

Você pode usar um lookbehind, em vez de um lookahead, na condicional. Você também pode usar um lookaround negativo, embora não recomendemos, pois só confunde as coisas, invertendo

o sentido de “então” e “senão.”

A condicional usando um lookaround pode ser escrita sem a condicional, no formato ‹(?=*if*)*then*|(?!*if*)*else*›. Se o lookahead positivo tiver êxito, a parte ‹*then*› é tentada. Se o lookahead positivo falhar, a alternância entra em cena. O lookahead negativo faz o mesmo teste. O lookahead negativo obtém êxito quando ‹*if*› falha, o que já é garantido, pois ‹(?=if)› falhou. Assim, ‹*else*› tem vez. Colocar o lookahead em uma condicional economiza tempo, pois a condicional tenta o ‹*if*› apenas uma vez.

Veja também:

Receitas 2.9 e 2.16.

2.18 Adicionar comentários à expressão regular

Problema

‹\d{4}-\d{2}-\d{2}› corresponde a uma data no formato aaaa-mm-dd, sem fazer qualquer validação dos números. Uma expressão regular tão simples como essa é apropriada quando você sabe que seus dados não contêm datas inválidas. Adicione comentários a esta expressão regular, para indicar o que cada parte da expressão faz.

Solução

```
\d{4} # Ano
- # Separador
\d{2} # Mês
- # Separador
\d{2} # Dia
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Discussão

Modo de espaçamento livre

As expressões regulares podem, rapidamente, tornar-se complicadas e de difícil compreensão. Assim como você deve comentar o código-fonte, deve também comentar todas as

expressões regulares, exceto as mais triviais. Todos os sabores de expressões regulares deste livro, exceto o JavaScript, oferecem uma sintaxe de expressão regular alternativa, que facilita muito o comentário.

Você pode habilitar essa sintaxe com a opção *espaçamento livre*. Ela possui diferentes nomes, nas várias linguagens de programação.

No .NET, defina a opção RegexOptions.IgnorePatternWhitespace. No Java, passe a flag Pattern.COMMENTS. O Python espera o re.VERBOSE. PHP, Perl e Ruby usam a flag /x.

Ativar o modo de espaçamento livre tem dois efeitos. Fora de classes de caracteres, ele transforma o símbolo hash (#) em um metacaractere. O hash inicia um comentário, que vai até o final da linha ou até o final da regex (o que ocorrer primeiro). O hash, e tudo depois dele, é simplesmente ignorado pelo mecanismo de expressão regular. Para corresponder a um sinal hash literal, coloque-o dentro de uma classe de caracteres ‹[#]› ou aplique o escape ‹\#›.

O outro efeito é que espaços em branco, que incluem espaços, tabulações e quebras de linha, também são ignorados fora das classes de caracteres. Para corresponder a um espaço literal, coloque-o dentro de uma classe de caracteres ‹[ ]›, ou aplique o escape ‹\ ›. Se você estiver preocupado com a legibilidade, pode usar o escape hexadecimal ‹\x20› ou, em vez disso, o escape Unicode ‹\u0020› ou ‹\x{0020}›. Para corresponder a uma tabulação, use ‹\t›. Para as quebras de linha, use ‹\r\n› (Windows) ou ‹\n› (Unix/Linux/OS X).

O modo de espaçamento livre em nada muda o interior das classes de caracteres. Uma classe de caracteres é um token único. Quaisquer caracteres em branco ou hashes dentro de classes de caracteres, são caracteres literais adicionados à classe de caracteres. Você não pode dividir classes de caracteres para comentar suas partes.

Java possui classes de caracteres com espaçamento livre

As expressões regulares não fariam jus a sua reputação se pelo menos um sabor não fosse compatível com os demais. Neste caso, Java é o estranho no ninho.

Em Java, classes de caracteres não são analisadas como tokens individuais. Se você ativar o modo de espaçamento livre, o Java irá ignorar espaços em branco dentro das classes de caracteres, e hashes dentro das classes de caracteres iniciam os comentários. Isto significa que você não poderá usar ‹[ ]› e ‹[#]› para corresponder a esses caracteres literalmente. Em vez disso, use ‹\u0020› e ‹\#›.

Variações

```
(?#Year)\d{4}(?#Separator)-(?#Month)\d{2}-(?#Day)\d{2}
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE, Perl, Python, Ruby

Se, por algum motivo, você não puder (ou não quiser) usar a sintaxe de espaçamento livre, ainda poderá adicionar comentários no formato ‹(?#comment)›. Todos os caracteres entre ‹(?#› e ‹)› são ignorados.

Infelizmente, JavaScript, o único sabor neste livro que não suporta espaçamento livre, também não suporta essa sintaxe de comentário. O Java também não a suporta.

```
(?x)\d{4} # Ano
- # Separador
\d{2} # Mês
- # Separador
\d{2} # Dia
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Se você não puder ativar o modo de espaçamento livre fora da expressão regular, poderá colocar o modificador de modo ‹(?x)› logo

em seu início. Certifique-se de que não tenha espaços em branco antes de ‹(?x)›. O modo de espaçamento livre começa apenas a partir do modificador de modo; qualquer espaço em branco prévio é significativo.

Modificadores de modo são explicados em detalhes em “Correspondência sem diferenciação entre maiúsculas e minúsculas”, subseção da receita 2.1.

2.19 Inserir texto literal no texto de substituição

Problema

Pesquisar uma expressão regular e substituir qualquer correspondência, literalmente, pelos oito caracteres $%*$1\1.

Solução

```
$%\*$$1\1
```

Sabores de texto de substituição: .NET, JavaScript

```
\$%\\*\$1\\1
```

Sabor de texto de substituição: Java

```
$%\*\$1\\1
```

Sabor de texto de substituição: PHP

```
\$%\*\$1\\1
```

Sabor de texto de substituição: Perl

```
$%\*$1\\1
```

Sabores de texto de substituição: Python, Ruby

Discussão

Quando e como escapar caracteres em textos de substituição

Esta receita mostra as diferentes regras de escape usadas pelos vários sabores de texto de substituição. Talvez, os dois únicos

caracteres que precisemos escapar em textos de substituição são o cifrão e a barra invertida. Os caracteres de escape também são o cifrão e a barra invertida.

O sinal de porcentagem e o asterisco, neste exemplo, são sempre caracteres literais, apesar de uma barra invertida precedente poder ser tratada como um escape, em vez de uma barra invertida literal. «$1» ou «\1» são retrorreferências a um grupo de captura. A receita 2.21 diz quais sabores utilizam quais sintaxes de retrorreferências.

O fato deste problema ter cinco soluções para sete sabores de textos de substituição demonstra que, realmente, não existe nenhum padrão para a sintaxe de textos de substituição.

.NET e JavaScript

.NET e JavaScript sempre tratam uma barra invertida como um caractere literal. Não a escape com outra barra invertida, ou você acabará com duas barras invertidas na substituição.

Um cifrão solitário é um caractere literal. Cifrões precisam ser escapados apenas quando seguidos por um dígito, um caractere de conjunção (&), crase, aspas, underscore (_), sinal de adição ou por outro cifrão. Para escapar um cifrão, preceda-o com outro cifrão.

Você pode duplicar todos os cifrões, se sentir que isso torna seu texto de substituição mais legível. Esta solução é igualmente válida:

```
$$%\*$$1\1
```

Sabores de texto de substituição: .NET, JavaScript

O .NET também exige que um cifrão seguido por uma chave de abertura seja escapado. «${group}» é uma retrorreferência nomeada no .NET. O JavaScript não suporta retrorreferências nomeadas.

Java

No Java, a barra invertida é usada para escapar barras invertidas e cifrões, no texto de substituição. Todas as barras invertidas literais e todos os cifrões literais devem ser escapados. Se você não escapá-los, o Java levantará uma exceção.

PHP

O PHP exige que barras invertidas ou cifrões seguidas por um dígito, ou seguidos por uma chave de abertura, sejam escapados por uma barra invertida.

A barra invertida também escapa outra barra invertida. Assim, você precisa escrever «\\\\» para usar duas barras invertidas literais na substituição. Todas as outras barras invertidas são tratadas como barras invertidas literais.

Perl

Perl é um pouco diferente dos outros sabores de texto de substituição: ele não tem, realmente, um sabor de texto de substituição. Enquanto as outras linguagens de programação possuem uma lógica especial em suas rotinas de busca-e-substituição para substituir coisas como «$1», no Perl isso é apenas uma interpolação de variável comum. No texto de substituição, você precisa escapar todos os cifrões literais com uma barra invertida, tal como faria em qualquer string entre aspas duplas.

Uma exceção é que o Perl suporta a sintaxe «\1» para retrorreferências. Assim, você precisa escapar uma barra invertida, seguida por um dígito, se quiser que a barra invertida seja literal. Uma barra invertida seguida por um cifrão também precisa ser escapada, para evitar que a barra invertida escape o cifrão.

Uma barra invertida também escapa outra barra invertida. Assim, você precisa escrever «\\\\» para usar duas barras invertidas literais na substituição. Todas as outras barras invertidas são tratadas como barras invertidas literais.

Python e Ruby

O cifrão não tem um significado especial no texto de substituição do Python e do Ruby. Barras invertidas precisam ser escapadas com outra barra invertida, quando seguidas por um caractere que dê significado especial à barra invertida.

No caso do Python, de «\1» a «\9» e «\g<» criam retrorreferências. Essas barras invertidas precisam ser escapadas.

No caso do Ruby, você precisa escapar uma barra invertida seguida por um dígito, um caractere de conjunção (&), crase, aspas ou sinal de adição.

Em ambas as linguagens, uma barra invertida escapa outra barra invertida. Assim, você precisa escrever «\\\\» para usar duas barras invertidas literais na substituição. Todas as outras barras invertidas são tratadas como literais.

Mais regras de escape para strings literais

Lembre-se de que, neste capítulo, lidamos apenas com as expressões regulares e com os textos de substituição em si. O próximo capítulo aborda linguagens de programação e strings literais.

Os textos de substituição mostrados anteriormente funcionarão quando a variável de string, que você estiver passando para a função replace(), armazenar tais textos. Em outras palavras, se seu aplicativo fornecer uma caixa de texto para que o usuário digite o texto de substituição, estas soluções mostram o que o usuário terá que digitar para que a pesquisa-e-substituição funcione como desejado. Se você testar seus comandos de pesquisa-e-substituição com o RegexBuddy, ou com outro testador de expressões regulares, os textos de substituição incluídos nesta receita mostrarão os resultados esperados.

Porém, estes mesmos textos de substituição não vão funcionar se você colá-los diretamente em seu código-fonte e colocá-los entre aspas. Strings literais, em linguagens de programação, têm suas próprias regras de escape; você precisa adicionar tais regras por sobre as regras de escape do texto de substituição. De fato, você pode acabar em uma confusão de barras invertidas.

Veja também:

Receita 3.14.

2.20 Inserir a correspondência da expressão regular no texto de substituição

Problema

Realize uma pesquisa-e-substituição que converta URLs em links HTML que apontem para o URL, usando o URL como texto do link. Para este exercício, defina um URL como "http:" e todos os caracteres seguintes que não sejam espaços em branco. Por exemplo, Please, visit http://www.regexcookbook.com torna-se Please, visit http://www.regexcookbook.com.

Solução

Expressão regular

```
http:\S+
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Substituição

```
<a href="$&">$&</a>
```

Sabores de texto de substituição: .NET, JavaScript, Perl

```
<a href="$0">$0</a>
```

Sabores de texto de substituição: .NET, Java, PHP

```
<a href="\0">\0</a>
```

Sabores de texto de substituição: PHP, Ruby

```
<a href="\&">\&</a>
```

Sabor de texto de substituição: Ruby

```
<a href="\g<0>">\g<0></a>
```

Sabor de texto de substituição: Python

Discussão

Reinserir toda a correspondência regex no texto de substituição é uma maneira fácil de inserir um novo texto antes, depois, em torno do texto correspondido ou até mesmo no meio de várias cópias do texto correspondido. A menos que esteja usando o Python, você não precisa adicionar nenhum grupo de captura em sua expressão regular para conseguir reutilizar a correspondência total.

No Perl, «`$&`» é, na verdade, uma variável. O Perl armazena a correspondência regex total nessa variável após cada correspondência regex realizada com êxito.

O .NET e o JavaScript adotaram a sintaxe «`$&`» para inserir a correspondência regex no texto de substituição. O Ruby utiliza barras invertidas, em vez de cifrões como tokens de texto de substituição; então, use «`\&`» para a correspondência global.

Java, PHP e Python não possuem um token especial para reinserir a correspondência regex global, mas eles permitem texto correspondido ao capturar os grupos a serem inseridos no texto de substituição, como a próxima seção explica. A correspondência total é um grupo de captura implícito de número 0. No caso do Python, precisamos usar a sintaxe de captura nomeada para referenciar o grupo zero. O Python não suporta «`\0`».

.NET e Ruby também suportam a sintaxe do grupo de captura zero, mas não importa qual sintaxe você usa. O resultado será o mesmo.

Veja também:

“Pesquisa e substituição com expressões regulares” no capítulo 1, e receita 3.15.

2.21 Inserir parte da correspondência da expressão regular no texto de substituição

Problema

Corresponda a qualquer sequência contígua de 10 dígitos, como 1234567890. Converta a sequência em um número de telefone formatado corretamente, por exemplo, (123) 456-7890.

Solução

Expressão regular

```
\b(\d{3})(\d{3})(\d{4})\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Substituição

```
($1) $2-$3
```

Sabores de texto de substituição: .NET, Java, JavaScript, PHP, Perl

```
(${1}) ${2}-${3}
```

Sabores de texto de substituição: .NET, PHP, Perl

```
(\1) \2-\3
```

Sabores de texto de substituição: PHP, Python, Ruby

Discussão

Substituições utilizando grupos de captura

A receita 2.10 explica como você pode usar grupos de captura, em sua expressão regular, para corresponder ao mesmo texto mais de uma vez. O texto correspondido por cada grupo de captura em sua regex também fica disponível após cada correspondência bem-sucedida. Você pode inserir o texto de alguns ou de todos os grupos de captura – em qualquer ordem, ou mesmo mais de uma vez – no texto de substituição.

Alguns sabores, como Python e Ruby, utilizam a mesma sintaxe «\1» para retrorreferências na expressão regular e no texto de substituição. Outros sabores utilizam a sintaxe Perl «$1», usando um

cifrão em vez de uma barra invertida. O PHP suporta ambos.

No Perl, «$1» e valores posteriores são, na verdade, variáveis definidas após cada correspondência regex bem-sucedida. Você pode usá-las em qualquer lugar, em seu código, até a execução da próxima correspondência regex. .NET, Java, JavaScript e PHP suportam «$1» apenas na sintaxe de substituição. Tais linguagens de programação oferecem outras formas de acesso a grupos de captura, dentro do código. O capítulo 3 explica isso em detalhes.

$10 e posteriores

Todos os sabores regex, neste livro, suportam até 99 grupos de captura em uma expressão regular. No texto de substituição, a ambiguidade pode ocorrer com «$10» ou «\10» e com valores posteriores. Eles podem ser interpretados como o décimo grupo de captura, ou como o primeiro grupo de captura seguido por um zero literal.

.NET, PHP, e Perl permitem que você coloque chaves em torno do número para tornar clara sua intenção. «${10}» é sempre o décimo grupo de captura, e «${1}0» é sempre o primeiro, seguido por um zero literal.

O Java e o JavaScript tentam ser inteligentes no caso do «$10». Se um grupo de captura, com o número especificado de dois dígitos, existir em sua expressão regular, os dois dígitos serão utilizados para este grupo. Se existirem menos grupos de captura, apenas o primeiro dígito será usado para fazer referência ao grupo, deixando o segundo como um literal. Assim, «$23» é o grupo de captura 23, caso ele exista. Se não existir, será o segundo grupo de captura, seguido de um «3» literal.

.NET, PHP, Perl, Python e Ruby sempre tratam «$10» e «\10» como o décimo grupo de captura, independentemente de ele existir. Caso ele não exista, o comportamento para grupos inexistentes entra em cena.

Referências a grupos não existentes

A expressão regular na solução desta receita possui três grupos de captura. Se você digitar «$4» ou «\4», no texto de substituição, estará adicionando uma referência a um grupo de captura que não existe. Isso dispara um de três comportamentos diferentes.

O Java e o Python vão esbravejar, levantando uma exceção, ou retornando uma mensagem de erro. Não use retrorreferências inválidas nestes sabores. Na verdade, você não deve usar retrorreferências inválidas em nenhum sabor. Se deseja inserir «$4» ou «\4», literalmente, escape o cifrão ou a barra invertida. A receita 2.19 explica isso em detalhes.

PHP, Perl e Ruby substituem todas as retrorreferências contidas no texto de substituição, incluindo as que apontam para grupos que não existem. Grupos inexistentes não capturam qualquer texto e, portanto, as referências a esses grupos são simplesmente substituídas por nada.

Finalmente, o .NET e o JavaScript tratam retrorreferências a grupos inexistentes como textos literais na substituição.

Todos os sabores substituem grupos que existam de fato na expressão regular, mas que não capturaram nada. Estes serão substituídos por nada.

Solução usando captura nomeada

Expressão regular

```
\b(?<area>\d{3})(?<exchange>\d{3})(?<number>\d{4})\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\b(?'area'\d{3})(?'exchange'\d{3})(?'number'\d{4})\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\b(?P<area>\d{3})(?P<exchange>\d{3})(?P<number>\d{4})\b
```

Opções Regex: Nenhuma

Sabores Regex: PCRE 4 e superior, Perl 5.10, Python

Substituição

(${area}) ${exchange}-${number}

Sabor de texto de substituição: .NET

(\g<area>) \g<exchange>-\g<number>

Sabor de texto de substituição: Python

(\k<area>) \k<exchange>-\k<number>

Sabor de texto de substituição: Ruby 1.9

(\k'area') \k'exchange'-\k'number'

Sabor de texto de substituição: Ruby 1.9

($1) $2-$3

Sabores de texto de substituição: .NET, PHP, Perl 5.10

(${1}) ${2}-${3}

Sabores de texto de substituição: .NET, PHP, Perl 5.10

(\1) \2-\3

Sabores de texto de substituição: PHP, Python, Ruby 1.9

Sabores que suportam captura nomeada

.NET, Python e Ruby 1.9 permitem que você utilize retrorreferências nomeadas no texto de substituição, caso tenha utilizado grupos de captura nomeados em sua expressão regular.

No caso do .NET e do Python, a sintaxe para retrorreferências nomeadas funciona, igualmente, com grupos de captura nomeados e numerados. Basta especificar o nome ou o número do grupo entre chaves ou colchetes angulares.

O Ruby utiliza a mesma sintaxe para retrorreferências no texto de substituição e na expressão regular. No caso de grupos de captura nomeados no Ruby 1.9, tal sintaxe seria «\k<group>» ou «\k'group'». A escolha entre colchetes angulares e aspas simples é mera conveniência notacional.

Perl 5.10 e PHP (utilizando PCRE) suportam grupos de captura nomeados em expressões regulares, mas não no texto de substituição. Você pode usar retrorreferências numeradas, no texto de substituição, que remetam a grupos de captura nomeados na expressão regular. O Perl 5.10 e o PCRE atribuem números, tanto para os grupos nomeados quanto para os grupos não-nomeados, da esquerda para a direita.

.NET, Python e Ruby 1.9 também permitem referências numeradas a grupos nomeados. No entanto, o .NET utiliza um sistema de numeração diferente para os grupos nomeados, como a receita 2.11 explica. Misturar nomes e números no .NET, Python ou Ruby não é recomendado. Ou você nomeia todos os seus grupos, ou não nomeia nenhum grupo. Sempre use retrorreferências nomeadas a grupos nomeados.

Veja também:

“Pesquisa e substituição com expressões regulares”, no capítulo 1, e receitas 2.9, 2.10, 2.11, e 3.15.

2.22 Inserir o contexto de correspondência no texto de substituição

Problema

Crie um texto que substitua a correspondência regex, antes dessa correspondência regex, seguido por todo o texto de assunto e pelo texto após a correspondência regex. Por exemplo, se Match for encontrado em BeforeMatchAfter, substitua a correspondência por BeforeBeforeMatchAfterAfter, resultando no novo texto BeforeBeforeBeforeMatchAfterAfterAfter.

Solução

```
$`$_$'
```
Sabores de texto de substituição: .NET, Perl
```
\`\`\&\'\'
```
Sabor de texto de substituição: Ruby
```
$`$`$&$'$'
```
Sabor de texto de substituição: JavaScript

Discussão

O termo *contexto* refere-se ao texto de assunto sobre o qual a expressão regular foi aplicada. Existem três partes de contexto: o texto de assunto antes da correspondência regex, o texto de assunto após a correspondência regex, e todo o texto assunto. O texto antes da correspondência é chamado, às vezes, de *contexto esquerdo*, e o texto após a correspondência é, de maneira similar, o *contexto direito*. O texto de assunto total seria o contexto esquerdo, a correspondência e o contexto direito.

.NET e Perl suportam «$`», «$'» e «$_» para inserir as três formas de contexto no texto de substituição. Na verdade, no Perl elas representam variáveis definidas após uma correspondência regex bem-sucedida, e estão disponíveis para qualquer código até a próxima tentativa de correspondência. Cifrão-crase é o contexto à esquerda. Você pode digitar o sinal de crase em um teclado padrão americano, pressionando a tecla à esquerda da tecla 1, no canto superior esquerdo do teclado. Cifrão-aspa reta forma o contexto à direita. A aspa reta é a aspa simples, usual. Em um teclado padrão americano, ela fica entre o ponto-e-vírgula e a tecla Enter. Cifrão-underscore corresponde ao texto de assunto total. Tal como o .NET e o Perl, o JavaScript utiliza «$`» e «$'» para os contextos esquerdo e direito. No entanto, o JavaScript não possui um token para inserir todo o texto de assunto. Você pode recompor o texto de assunto inserindo a correspondência total da regex com «$&» entre os contextos esquerdo e direito.

O Ruby suporta os contextos esquerdo e direito por meio de «\`» e «\'», e utiliza «\&» para inserir a correspondência total da regex. Tal

como no JavaScript, não há um token para o texto de assunto total.

Veja também:

“Pesquisa e substituição com expressões regulares” no capítulo 1, e receita 3.15.

---

1 Um molde de tecla (keycap) é uma pequena peça de plástico colocada sobre um interruptor de tecla (keyswitch), em um teclado de computador. O molde de tecla recebe ilustrações para indicar a função da tecla ou o caractere alfanumérico ao qual corresponde. (Fonte: Wikipédia).

2 Outro tipo de mecanismo é o orientado a texto. A diferença fundamental é que um mecanismo orientado a texto visita cada caractere no texto de assunto apenas uma vez, enquanto um mecanismo orientado a expressões regulares pode visitar cada caractere várias vezes. Mecanismos orientados a texto são muito mais rápidos, mas suportam expressões regulares apenas no sentido matemático, descrito no início do capítulo 1. O estilo pomposo das expressões regulares do Perl, que torna este livro tão interessante, pode ser implementado somente com um mecanismo orientado a expressões regulares.

3 O mecanismo de expressão regular RegexBuddy também permite uma regex completa dentro do lookbehind, mas (ainda) não possui uma funcionalidade semelhante ao RegexOptions.RightToLeft do .NET para reverter toda a expressão regular.

CAPÍTULO 3

Programando com expressões regulares

Linguagens de programação e sabores Regex

Este capítulo explica como implementar as expressões regulares na linguagem de programação de sua escolha. As receitas deste capítulo pressupõem que você já tenha uma expressão regular funcional à disposição; os capítulos anteriores podem ajudar nesse sentido. Agora, você vai encarar o trabalho de colocar uma expressão regular em seu código-fonte e fazê-la realizar algo.

Nós demos nosso melhor, neste capítulo, para explicar exatamente como, e por qual motivo, cada parte do código funciona. Devido ao nível de detalhamento, lê-lo do começo ao fim pode ser um pouco entediante. Se você estiver lendo o *Expressões Regulares Cookbook* pela primeira vez, recomendamos que passe rapidamente por este capítulo, para ter uma ideia do que pode ou deve ser feito. Mais tarde, quando quiser implementar uma das expressões regulares dos capítulos seguintes, volte aqui, para saber exatamente como integrar as expressões regulares com sua linguagem de programação escolhida.

Os capítulos de 4 a 8 utilizam expressões regulares para resolver problemas reais. Esses capítulos focam nas expressões regulares em si, e muitas receitas neles apresentadas não mostram qualquer código-fonte. Para fazer as expressões regulares que você encontra naqueles capítulos funcionarem, basta conectá-las a um dos trechos de código deste capítulo.

Devido ao foco dos outros capítulos nas expressões regulares, elas apresentam soluções para determinados sabores de expressões regulares, ao invés de linguagens de programação específicas. Os sabores de expressões regulares não correspondem, um-a-um, com as linguagens de programação. Linguagens de script tendem a ter seus próprios sabores de expressão regular, e outras linguagens de programação contam com bibliotecas para obter suporte a expressão regular. Algumas bibliotecas estão disponíveis para várias linguagens, enquanto algumas linguagens têm várias bibliotecas disponíveis.

"Muitos sabores de expressões regulares", é um tópico que descreve todos os sabores de expressões regulares abordados neste livro. "Muitos sabores de texto de substituição", lista todos os sabores de texto de substituição usados para pesquisa e substituição com uma expressão regular.

Todas as linguagens de programação abordadas neste capítulo utilizam um destes sabores.

Linguagens abrangidas neste capítulo

Este capítulo cobre oito linguagens de programação. Cada receita tem soluções distintas para todas as oito linguagens, e muitas receitas também têm discussões separadas para todas as oito linguagens. Se uma técnica se aplicar a mais de uma linguagem, nós a repetimos na discussão para cada uma dessas linguagens. Fizemos isso para que você possa pular, com segurança, as discussões sobre linguagens de programação nas quais não estiver interessado:

C#

O C# utiliza o Framework Microsoft .NET. As classes `System.Text.RegularExpressions` utilizam os sabores de expressão regular e de texto de substituição ".NET". Este livro aborda do C# 1.0 ao 3.5, ou do Visual Studio 2002 ao 2008.

VB.NET

Este livro utiliza VB.NET e Visual Basic.NET para se referir ao Visual Basic 2002 e posteriores, com o objetivo de distinguir essas versões do Visual Basic 6 e anteriores. O Visual Basic, agora, utiliza o Framework Microsoft .NET. As classes System.Text.RegularExpressions utilizam os sabores de expressão regular e de texto de substituição ".NET". Este livro aborda do Visual Basic 2002 ao 2008.

Java

O Java 4 foi a primeira versão Java a fornecer suporte interno a expressões regulares por meio do pacote java.util.regex. O pacote java.util.regex utiliza os sabores de expressão regular e de texto de substituição "Java". Este livro aborda as versões 4, 5 e 6 do Java.

JavaScript

Este é o sabor de expressão regular utilizado na linguagem de programação conhecida como JavaScript. Todos os navegadores modernos a implementaram: Internet Explorer (a partir da versão 5.5), Firefox, Opera, Safari e Chrome. Muitas outras aplicações também utilizam o JavaScript como linguagem de script.

Propriamente dito, neste livro utilizamos o termo *JavaScript* para indicar a linguagem de programação definida na versão 3 do padrão ECMA-262. Esta norma define a linguagem de programação ECMAScript, mais conhecida por suas implementações de JavaScript e JScript em diferentes navegadores.

O ECMA-262v3 também define os sabores de expressão regular e de texto de substituição utilizados pelo JavaScript. Esses sabores são rotulados como "JavaScript" neste livro.

PHP

O PHP tem três conjuntos de funções de expressões regulares. Recomendamos utilizar as funções preg. Portanto, este livro

abrange apenas as funções preg, incluídas no PHP a partir da versão 4.2.0. Este livro aborda as versões 4 e 5 do PHP. As funções preg são encapsulamentos PHP em torno da biblioteca PCRE. O sabor de expressão regular PCRE é indicado como “PCRE” neste livro. Como o PCRE não inclui funcionalidades de pesquisa, os desenvolvedores PHP planejaram sua própria sintaxe de texto de substituição para preg_replace. Este sabor de texto de substituição é rotulado como “PHP”, neste livro.

As funções mb_ereg fazem parte das funções ”multibyte” do PHP, projetadas para trabalhar bem com idiomas tradicionalmente codificados com conjuntos de caracteres multibyte, como o japonês e o chinês. No PHP 5, as funções mb_ereg utilizam a biblioteca de expressão regular Oniguruma, originalmente desenvolvida para o Ruby. O sabor de expressão regular Oniguruma é indicado como “Ruby 1.9”, neste livro. O uso das funções mb_ereg é recomendado apenas se você tiver uma exigência específica para lidar com páginas de código multibyte, e se você já estiver familiarizado com as funções mb_ em PHP.

O grupo de funções ereg é o conjunto mais antigo de funções de expressões regulares do PHP, e está oficialmente obsoleto no PHP 5.3.0. Ele não depende de bibliotecas externas e implementa o sabor POSIX ERE. Este sabor oferece apenas um conjunto limitado de funcionalidades, e não será discutido neste livro. O POSIX ERE é um subconjunto restrito dos sabores Ruby 1.9 e PCRE. Você pode pegar a expressão regular de qualquer chamada de função ereg e usá-la com mb_ereg ou preg. No caso do preg, você terá que adicionar delimitadores no estilo Perl (Receita 3.1).

Perl

O suporte nativo do Perl a expressões regulares é a principal razão pela qual as expressões regulares são populares hoje. Os sabores de expressão regular e de texto de substituição, usados pelos operadores Perl m// e s///, são rotulados como “Perl” neste

livro. Este livro aborda as versões 5.6, 5.8 e 5.10 do Perl.

Python

Python suporta expressões regulares por meio de seu módulo `re`. Os sabores de expressão regular e de texto de substituição usados por este módulo são rotulados como “Python”, neste livro. Este livro cobre o Python 2.4 e 2.5.

Ruby

O Ruby tem suporte nativo a expressões regulares. Este livro aborda o Ruby 1.8 e o Ruby 1.9. Estas duas versões do Ruby têm diferentes mecanismos-padrão de expressões regulares. O Ruby 1.9 utiliza o mecanismo Oniguruma, que tem mais recursos do que o mecanismo de expressão regular clássico do Ruby 1.8. “Sabores regex abordados neste livro”, traz mais detalhes sobre este assunto.

Neste capítulo, nós não falamos muito das diferenças entre o Ruby 1.8 e 1.9. As expressões regulares deste capítulo são muito básicas, e não utilizam os novos recursos do Ruby 1.9. Como o suporte a expressões regulares é compilado dentro da linguagem Ruby em si, o código Ruby que você utiliza para implementar suas expressões regulares é o mesmo, independentemente de ter compilado o Ruby utilizando o mecanismo de expressão regular classic, ou o mecanismo Oniguruma. Você poderia recompilar o Ruby 1.8 para utilizar o mecanismo Oniguruma, caso requeira suas funcionalidades.

Mais linguagens de programação

As linguagens de programação, na lista a seguir, não são abordadas neste livro, mas fazem uso de algum dos sabores de expressões regulares aqui apresentados. Se você utilizar uma dessas linguagens, pule este capítulo; ainda assim, todos os outros capítulos serão úteis:

ActionScript

ActionScript é a implementação Adobe do padrão ECMA-262. A partir da versão 3.0, o ActionScript possui suporte completo às expressões regulares do ECMA-262v3. Este sabor de expressão regular é por nós rotulado “JavaScript”. A linguagem ActionScript é, também, muito próxima à linguagem JavaScript. Você deve ser capaz de adaptar os exemplos em JavaScript deste capítulo para o ActionScript.

C

A linguagem C pode utilizar uma grande variedade de bibliotecas de expressão regular. A biblioteca de código aberto PCRE é, provavelmente, a melhor escolha entre os sabores abrangidos por este livro. Você pode baixar o código-fonte completo em C no endereço *http://www.pcre.org*. O código é escrito para compilar em uma vasta gama de compiladores e em uma ampla gama de plataformas.

C + +

O C++ pode utilizar uma grande variedade de bibliotecas de expressão regular. A biblioteca de código aberto PCRE é, provavelmente, a melhor escolha entre os sabores abrangidos por este livro. Você pode usar a API em C diretamente, ou usar os encapsuladores de classe C++ incluídos no próprio download do PCRE (veja *http://www.pcre.org*).

No Windows, você poderia importar o objeto COM RegExp do VBScript 5.5, como explicado mais adiante, para o Visual Basic 6. Isso poderia ser útil para conseguir consistência entre as expressões regulares de um backend em C ++, e um frontend em JavaScript.

Delphi para Win32

No momento em que escrevíamos este livro, a versão Win32 do Delphi não possuía suporte nativo a expressões regulares. Há muitos componentes VCL disponíveis que fornecem suporte a expressões regulares. Recomendo que você escolha um componente baseado no PCRE. O Delphi tem a capacidade de

vincular arquivos de objeto em C nas suas aplicações, e muitos encapsuladores VCL para PCRE usam tais arquivos de objeto. Tal procedimento permite manter o aplicativo como um único arquivo *.exe*.

Você pode baixar meu próprio componente TPerlRegEx em *http://www.regexp.info/delphi.html*. É um componente VCL que se instala na paleta de componentes, assim você pode facilmente inseri-lo em um formulário. Outro encapsulador PCRE popular do Delphi é a classe TJclRegEx da biblioteca JCL, disponível em *http://www.delphi-jedi.org*. TJclRegEx descende de TObject, então você não poderá inseri-lo em um formulário.

Ambas as bibliotecas possuem código aberto sob a Licença Pública do Mozilla.

Delphi Prism

No Delphi Prism, você pode usar o suporte a expressão regular oferecido pelo framework .NET. Basta adicionar System.Text.RegularExpressions à cláusula uses de qualquer unidade Delphi Prism, na qual você queira usar as expressões regulares.

Uma vez feito isso, você pode utilizar as mesmas técnicas mostradas nos trechos de código C# e VB.NET deste capítulo.

Groovy

Você pode utilizar expressões regulares no Groovy com o pacote java.util.regex, tal como faria no Java. Na verdade, todas as soluções em Java deste capítulo também deveriam funcionar com o Groovy. A sintaxe própria do Groovy para expressões regulares apenas fornece atalhos de notação. Uma expressão regular literal, delimitada por barras, é uma instância de java.lang.String, e o operador =~ instancia java.util.regex.Matcher. Você pode misturar livremente a sintaxe Groovy com a sintaxe padrão Java – as classes e objetos são todos iguais.

PowerShell

PowerShell é a linguagem de script em shell da Microsoft,

baseada no framework .NET. Os operadores -match e -replace, nativos do Powershell, utilizam os sabores de expressão regular e de texto de substituição .NET, como descrito neste livro.

R

O projeto R suporta expressões regulares por meio das funções grep, sub e regexpr do pacote base. Todas estas funções possuem um argumento rotulado perl, definido como FALSE, caso você o omita. Defina-o como TRUE para utilizar o sabor regex PCRE, tal como descrito neste livro. As expressões regulares mostradas para o PCRE 7 trabalham com o R 2.5.0 e versões posteriores. Para versões anteriores do R, utilize as expressões regulares marcadas, neste livro, como “PCRE 4 e posterior”. Os sabores “básico” e “estendido”, suportados pelo R, são sabores velhos e limitados, e não são discutidos.

REALbasic

O REALbasic possui uma classe RegEx nativa. Internamente, essa classe utiliza a versão UTF-8 da biblioteca do PCRE. Isso significa que você pode utilizar o suporte a Unicode do PCRE, mas terá que utilizar a classe TextConverter do REALbasic para converter texto não-ASCII em UTF-8, antes de passá-lo para a classe RegEx.

Todas as expressões regulares mostradas neste livro para o PCRE 6 funcionarão no REALbasic. Uma ressalva é que, no REALbasic, as opções “não diferenciar maiúsculas de minúsculas“ e “circunflexo e cifrão correspondem em quebras de linha” (“linhas múltiplas”) estão ativadas por padrão. Se você quiser utilizar uma expressão regular deste livro que não lhe peça para ativar esses modos de correspondência, terá de desativá-los explicitamente no REALbasic.

Scala

O Scala fornece suporte nativo a expressões regulares por meio do pacote scala.util.matching. Tal suporte é construído sobre o mecanismo de expressões regulares do pacote Java java.util.regex.

Os sabores de expressão regular e de texto de substituição usados pelo Java e pelo Scala são rotulados como “Java”, neste livro.

Visual Basic 6

O Visual Basic 6 é a última versão do Visual Basic que não exige o framework .NET. Isso significa, também, que o Visual Basic 6 não pode utilizar o excelente suporte a expressões regulares do framework .NET. Os exemplos de código VB.NET, neste capítulo, não funcionarão com o VB 6.

O Visual Basic 6 facilita bastante o uso das funcionalidades fornecidas pelas bibliotecas ActiveX e COM. Uma delas é a biblioteca de script VBScript, da Microsoft, que possui funcionalidades decentes de expressões regulares a partir da versão 5.5. A biblioteca de script implementa o mesmo sabor de expressão regular utilizado no JavaScript, tal como padronizado no ECMA-262v3. Ela faz parte do Internet Explorer 5.5 e posterior, e está disponível em todos os computadores rodando o Windows XP ou Vista, e em versões anteriores do Windows, caso o usuário tenha atualizado para o IE 5.5 ou posterior. Isso inclui quase todos os PCs Windows usados para se conectar à Internet.

Para usar esta biblioteca em seu aplicativo Visual Basic, selecione Project | References no menu da IDE do VB. Percorra a lista para localizar o item “Microsoft VBScript Regular Expressions 5.5”, imediatamente abaixo de “Microsoft VBScript Regular Expressions 1.0”. Certifique-se de assinalar a versão 5.5, e não a versão 1.0. A versão 1.0 é fornecida apenas para compatibilidade com versões anteriores; suas funcionalidades não são satisfatórias.

Depois de adicionar a referência, você poderá ver quais classes e membros de classe que a biblioteca oferece. Selecione View | Object Browser, no menu. No Object Browser, selecione a biblioteca “VBScript_RegExp_55”, na lista de seleção no canto

superior esquerdo.

3.1 Expressões regulares literais no código-fonte

Problema

A expressão regular ‹[$"'\n\d/\\]› foi dada como a solução de um problema. Esta expressão regular consiste em uma única classe de caracteres que corresponde a um cifrão, uma aspa dupla, uma aspa simples, um caractere de alimentação de linha (line feed), um dígito entre 0 e 9, e uma barra normal ou invertida. Você vai codificar esta expressão regular em seu código-fonte como uma constante em string ou um operador de expressão regular.

Solução

C#

Como uma string normal:

```
"[$\"'\n\\d/\\\\]"
```

Como uma string verbatim:

```
@"[$""'\n\d/\\]"
```

VB.NET

```
"[$""'\n\d/\\]"
```

Java

```
"[$\"'\n\\d/\\\\]"
```

JavaScript

```
/[$"'\n\d\/\\]/
```

PHP

```
'%[$"\'\n\d/\\\\]%'
```

Perl

Operador de correspondência a padrões:

```
/[\$"'\n\d\/\\]/
m![\$"'\n\d/\\]!
```

Operador de substituição:

```
s![\$"'\n\d/\\]!!
```

Python

String bruta (raw), triplamente aspada:

```
r"""[$"'\n\d/\\]"""
```

String normal:

```
"[$\"'\n\\d/\\\\]"
```

Ruby

Expressão regular literal delimitada por barras:

```
/[$"'\n\d\/\\]/
```

Expressão regular literal delimitada por pontuação de sua escolha:

```
%r![$"'\n\d/\\]!
```

Discussão

Quando este livro mostra uma expressão regular por si mesma (e não como parte de um trecho maior de código-fonte), ele sempre apresenta expressões regulares sem adornos. Esta receita é a única exceção. Se você estiver usando um testador de expressões regulares, como o RegexBuddy ou RegexPal, você deve digitar a expressão regular sem adornos. Se seu aplicativo aceitar uma expressão regular como entrada do usuário, o usuário deve digitá-la sem adornos.

Porém, se você quiser codificar a expressão regular em seu código-fonte, terá um trabalho extra. Copiar e colar as expressões regulares de um testador de expressão regular em seu código-fonte, ou vice-versa, sem o devido cuidado, muitas vezes o fará coçar a cabeça, tentando entender por que a expressão regular funciona em sua

ferramenta, mas não em seu código fonte, ou por que o testador falha em uma expressão regular que você copiou a partir do código de outra pessoa. Todas as linguagens de programação discutidas neste livro requerem que as expressões regulares literais sejam delimitadas de uma certa forma, com algumas linguagens exigindo o uso de strings e outras exigindo uma constante especial de expressão regular. Se sua expressão regular inclui delimitadores da linguagem ou outros caracteres com significados especiais, você terá de escapá-los.

A barra invertida é o caractere de escape mais utilizado. É por isso que a maioria das soluções deste problema possui mais barras invertidas, além das quatro barras invertidas da expressão regular original.

C#

Em C#, você pode passar expressões regulares literais para o construtor `Regex()` e para várias funções-membro da classe `Regex`. O parâmetro que leva a expressão regular é sempre declarado como uma string.

O C# suporta dois tipos de literais strings. O tipo mais comum é a string entre aspas duplas, bastante conhecida em linguagens como C++ e Java. Dentro de strings entre aspas duplas, as aspas duplas e as barras invertidas devem ser escapadas com uma barra invertida. Escapes para caracteres não imprimíveis, como `‹\n›`, também são suportados em strings. Existe uma diferença entre `"\n"` e `"\\n"` quando utilizamos `RegexOptions.IgnorePatternWhitespace` (veja a receita 3.4) para ativar o modo de espaçamento livre, como explicado na receita 2.18. `"\n"` é uma string com uma quebra de linha literal, interpretada e ignorada como um espaço em branco. `"\\n"` é uma string contendo o token de expressão regular `‹\n›`, que corresponde a uma nova linha.

Strings verbatim começam com um sinal de arroba e aspa dupla, e terminam com outra aspa dupla. Para incluir aspas duplas em uma

string verbatim, duplique-a. Barras invertidas não precisam ser escapadas, resultando em uma expressão regular muito mais legível. @"\n" será sempre o token de expressão regular ‹\n›, que corresponde a uma nova linha, inclusive no modo de espaçamento livre. Strings verbatim não suportam ‹\n› no nível da string, mas, ao invés disso, podem abranger várias linhas. Isso torna as strings verbatim ideais para uso em expressões regulares de espaçamento livre.

A escolha é clara: use strings verbatim para colocar expressões regulares em seu código-fonte C#.

VB.NET

No VB.NET, você pode passar expressões regulares literais ao construtor Regex(), e a várias funções-membro da classe Regex. O parâmetro que leva a expressão regular é sempre declarado como string.

O Visual Basic utiliza strings de aspas duplas. Aspas duplas dentro da string devem ser duplicadas. Nenhum outro caractere precisa ser escapado.

Java

Em Java, você pode passar expressões regulares literais para a fábrica de classes Pattern.compile() e a várias funções da classe String. O parâmetro que leva a expressão regular é sempre declarado como uma string.

O Java utiliza strings de aspas duplas. Dentro de strings em aspas duplas, as aspas duplas e as barras invertidas devem ser escapadas com uma barra invertida. O escape de caracteres não imprimíveis, como ‹\n›, e escapes Unicode, como ‹\uFFFF›, também são suportados nas strings.

Existe uma diferença entre "\n" e "\\n" quando utilizamos Pattern.COMMENTS (veja receita 3.4) para ativar o modo de espaçamento livre, como explicado na receita 2.18. "\n" é uma string

com uma quebra de linha literal, interpretada e ignorada como um espaço em branco. "\\n" é uma string contendo o token de expressão regular ‹\n›, e corresponde a uma nova linha.

JavaScript

Em Javascript, expressões regulares são mais bem criadas se usarmos a sintaxe especial de declaração de expressões regulares literais. Basta colocar sua expressão regular entre duas barras. Se houver alguma dentro da própria expressão regular, escape-a com uma barra invertida.

Embora seja possível criar um objeto RegExp a partir de uma string, não faz sentido usar, em seu código, a notação de string literal para expressões regulares. Você teria de escapar aspas e barras invertidas, o que geralmente leva a uma floresta de barras invertidas.

PHP

Expressões regulares literais para uso com funções preg do PHP são uma engenhoca curiosa. Ao contrário do JavaScript ou Perl, o PHP não possui um tipo de expressão regular nativo. As expressões regulares devem ser sempre tratadas como strings entre aspas. Isso também é verdadeiro para as funções ereg e mb_ereg. Mas, em sua busca para imitar o Perl, os desenvolvedores das funções de encapsulamento do PHP para o PCRE acrescentaram um requisito adicional.

Dentro da string, a expressão regular deve ser delimitada como uma expressão regular literal, no estilo Perl. Isso significa que, onde você escreveria /regex/ em Perl, a string para as funções preg do PHP torna-se '/regex/'. Tal como em Perl, você pode usar qualquer par de caracteres de pontuação como delimitadores. Se o delimitador de expressão regular estiver dentro da expressão regular, ele deverá ser escapado com uma barra invertida. Para evitar isso, escolha um delimitador que não esteja na expressão regular. Nesta receita, eu usei o sinal de porcentagem porque a barra está na expressão

regular, mas o sinal de porcentagem, não. Se a barra não estiver na expressão regular, utilize-a, pois se trata do delimitador mais utilizado em Perl, e do delimitador exigido em JavaScript e Ruby.

O PHP suporta strings com aspas simples e duplas. Ambas exigem que as ocorrências de aspas (simples, ou duplas) e barras invertidas, dentro de uma expressão regular, sejam escapadas por uma barra invertida. Em strings com aspas duplas, o sinal de cifrão também precisa ser escapado. No caso das expressões regulares, você deve usar strings com aspas simples, a menos que você realmente queira interpolar variáveis em sua expressão regular.

Perl

As expressões regulares literais em Perl são usadas com o operador de correspondência a padrões e com o operador de substituição. O operador de correspondência a padrões consiste em duas barras, com a expressão regular entre elas. Barras dentro da expressão regular devem ser escapadas por uma barra invertida. Não há necessidade de escapar qualquer outro caractere, exceto, talvez, `$` e `@`, como explicado no final desta subseção.

Uma notação alternativa para o operador de correspondência a padrões coloca a expressão regular entre qualquer par de caracteres de pontuação, precedido da letra `m`. Se você usar qualquer tipo de pontuação de abertura e fechamento (parênteses, chaves ou colchetes) como delimitador, eles precisam combinar: por exemplo, `m{`*`regex`*`}`. Se você usar outra pontuação, basta usar o mesmo caractere duas vezes. A solução para esta receita usa o ponto de exclamação. Isso nos poupa de precisar escapar a barra literal na expressão regular. Se os delimitadores de abertura e de fechamento forem diferentes, apenas o delimitador de fechamento precisará ser escapado com uma barra invertida, caso ele ocorra como um caractere literal dentro da expressão regular.

O operador de substituição é similar ao operador de correspondência a padrões. Ele começa com `s`, em vez de `m`, e se

acopla ao texto de substituição. Ao usar parênteses ou pontuação semelhante como delimitadores, você precisará de dois pares: s[*regex*][*replace*]. No caso de todos os outros sinais de pontuação, use-os três vezes: s/*regex*/*replace*/.

O Perl analisa os operadores de correspondência a padrões e de substituição como strings entre aspas duplas. Se você escrever m/I am $name/, e se $name armazenar a string "Jan", você terminará com a expressão regular ‹I□am□Jan›. $", também uma variável em Perl; por isso, nesta receita, tivemos de escapar o sinal de cifrão literal na classe de caracteres da nossa expressão regular.

Nunca escape um sinal de cifrão que você queira usar como âncora (veja receita 2.5). Um sinal de cifrão escapado será sempre um literal. O Perl é inteligente o suficiente para diferenciar os cifrões usados como âncoras e os cifrões usados como interpolação de variáveis, já que as âncoras podem ser usadas, de forma sensata, apenas no final de um grupo, no final da expressão regular ou antes de um caractere de nova linha. Você não deve escapar o cifrão em ‹m/^*regex*$/›, caso queira verificar se “regex“ corresponde completamente à string de assunto.

O sinal de arroba não tem um significado especial nas expressões regulares, mas é usado para a interpolação de variáveis no Perl. Você precisa escapá-la em expressões regulares literais no código Perl, tal como faz com strings entre aspas duplas.

Python

As funções no módulo re do Python esperam que as expressões regulares literais sejam passadas como strings. Você pode usar qualquer uma das várias formas que o Python oferece para colocar as strings entre aspas. Dependendo dos caracteres em sua expressão regular, formas diferentes de colocá-los entre aspas podem reduzir o número de caracteres que você precisa escapar com barras invertidas.

Geralmente, as strings brutas (raw) são a melhor opção. As strings

brutas do Python não requerem o escape de caracteres. Se você usar uma string bruta, não precisará duplicar as barras invertidas em sua expressão regular. r"\d+" é mais legível do que "\\d+", particularmente quando sua expressão regular vai ficando extensa.

A única situação em que as strings brutas não são ideais é quando a expressão regular inclui tanto aspas simples quanto aspas duplas. Assim, você não poderá usar uma string bruta delimitada por um par de aspas simples ou duplas, porque não há uma maneira de escapar as aspas dentro da expressão regular. Nesse caso, você pode triplicar as aspas da string bruta, como fizemos na solução Python para esta receita. A string normal é mostrada a título de comparação.

Se você quiser utilizar os recursos Unicode em sua expressão regular, explicados na receita 2.7, será necessário utilizar strings Unicode. Você pode transformar uma string em uma string Unicode precedendo-a com um u.

Strings brutas não suportam caracteres de escape não imprimíveis, como \n. As strings brutas tratam sequências de escape como texto literal. Tal ocorrência não é um problema para o módulo re. Ele suporta esses escapes como parte da sintaxe da expressão regular, e como parte da sintaxe do texto de substituição. Um \n literal, em uma string bruta, continuará a ser interpretado como uma nova linha em suas expressões regulares e textos de substituição.

Há uma diferença entre a string "\n", por um lado, e a string bruta r"\n", por outro, quando utilizamos re.VERBOSE (veja receita 3.4) para ativar o modo de espaçamento livre, como explicado na receita 2.18. "\n" é uma string com uma quebra de linha literal, interpretada e ignorada como espaço em branco. "\\n" e r"\n" são ambas strings contendo o token de expressão regular ‹\n›, que corresponde a um caractere de nova linha.

As strings brutas entre aspas triplas, como r"""\n""", são a melhor solução quando usamos o modo de espaçamento livre, pois podem ocupar várias linhas. Além disso, ‹\n› não é interpretado no nível da

string; então, ele pode ser interpretado no nível da expressão regular para corresponder a uma quebra de linha.

Ruby

Expressões regulares, em Ruby, são mais bem criadas se utilizarmos a sintaxe especial de declaração de expressões regulares literais. Basta colocar sua expressão regular entre duas barras. Se existir alguma barra dentro da própria expressão regular, escape-a com uma barra invertida.

Se você não quiser escapar as barras em sua expressão regular, poderá prefixá-la com `%r` e, em seguida, usar qualquer caractere de pontuação de sua escolha como delimitador.

Embora seja possível criar um objeto `Regexp` a partir de uma string, não faz sentido usar, em seu código, a notação de string para expressões regulares literais. Dessa forma, você teria de escapar aspas e barras invertidas, o que geralmente leva a uma floresta de barras invertidas.

O Ruby é, neste aspecto, muito similar ao JavaScript, exceto pelo fato de que o nome da classe é `Regexp`, tal como uma palavra em Ruby, enquanto em JavaScript a classe chama-se `RegExp`, usando a notação-camelo.

Veja também:

A receita 2.3 explica como as classes de caracteres funcionam, e por que são necessárias duas barras invertidas na expressão regular para incluir apenas uma na classe de caracteres.

A receita 3.4 explica como definir as opções da expressão regular, o que é realizado como parte das expressões regulares literais em algumas linguagens de programação.

3.2 Importar a biblioteca de expressões regulares

Problema

Para ser capaz de usar expressões regulares em sua aplicação, você deseja importar a biblioteca de expressões regulares ou o espaço de nomes (namespace) em seu código-fonte.

O restante dos trechos de código-fonte, neste livro, assume que você já tenha feito isso, se necessário.

Solução

C#

```
using System.Text.RegularExpressions;
```

VB.NET

```
Imports System.Text.RegularExpressions
```

Java

```
import java.util.regex.*;
```

Python

```
import re
```

Discussão

Algumas linguagens de programação possuem expressões regulares nativas. No caso dessas linguagens, você não precisa fazer nada para ativar o suporte a expressões regulares. Outras linguagens fornecem funcionalidades de expressão regular por meio de uma biblioteca, que precisa ser importada com uma declaração de importação em seu código-fonte. Algumas linguagens não têm suporte a expressões regulares. Nesse caso, por conta própria, você terá de compilar e vincular o suporte a expressões regulares.

C#

Se você colocar a declaração `using` no início do seu arquivo-fonte em C#, poderá fazer referência direta às classes que oferecem funcionalidades de expressão regular, sem ter de qualificá-las. Por exemplo, você poderá escrever `Regex()`, ao invés de

System.Text.RegularExpressions.Regex().

VB.NET

Se colocar a declaração Imports na parte superior do seu arquivo-fonte em VB.NET, você poderá fazer referência direta às classes que oferecem funcionalidades de expressão regular, sem a necessidade de qualificá-las. Por exemplo, você poderá escrever Regex(), ao invés de System.Text.RegularExpressions.Regex().

Java

Você precisa importar o pacote java.util.regex em sua aplicação para poder utilizar a biblioteca nativa de expressões regulares do Java.

JavaScript

O suporte a expressões regulares do JavaScript é nativo, e está sempre disponível.

PHP

As funções preg são nativas e estão sempre disponíveis no PHP 4.2.0 e posterior.

Perl

O suporte a expressões regulares do Perl é nativo e está sempre disponível.

Python

Você precisa importar o módulo re em seu script para poder utilizar as funções de expressão regular do Python.

Ruby

O suporte a expressões regulares do Ruby é nativo e está sempre disponível.

3.3 Criar objetos de expressão regular

Problema

Você quer instanciar um objeto de expressão regular ou, então, compilar uma expressão regular para que possa utilizá-la de forma eficiente ao longo de sua aplicação.

Solução

C#

Se você sabe que a expressão regular está correta:

```
Regex regexObj = new Regex("regex pattern");
```

Se a expressão regular é fornecida pelo usuário final (UserInput sendo uma variável de string):

```
try {
  Regex regexObj = new Regex(UserInput);
} catch (ArgumentException ex) {
  // Erro de sintaxe na expressão regular
}
```

VB.NET

Se você sabe que a expressão regular está correta:

```
Dim RegexObj As New Regex("regex pattern")
```

Se a expressão regular é fornecida pelo usuário final (UserInput sendo uma variável de string):

```
Try
  Dim RegexObj As New Regex(UserInput)
Catch ex As ArgumentException
  'Erro de sintaxe na expressão regular
End Try
```

Java

Se você sabe que a expressão regular está correta:

```
Pattern regex = Pattern.compile("regex pattern");
```

Se a expressão regular é fornecida pelo usuário final (userInput sendo uma variável de string):

```
try {
```

```
    Pattern regex = Pattern.compile(userInput);
  } catch (PatternSyntaxException ex) {
    // Erro de sintaxe na expressão regular
  }
```

Para poder usar a expressão regular em uma string, crie um Matcher:

```
  Matcher regexMatcher = regex.matcher(subjectString);
```

Para usar a expressão regular em outra string, você pode criar um novo Matcher, como no exemplo anterior, ou reutilizar um existente:

```
  regexMatcher.reset(anotherSubjectString);
```

JavaScript

Expressão regular literal em seu código:

```
  var myregexp = /regex pattern/;
```

Expressão regular recuperada a partir da entrada do usuário, como uma string armazenada na variável userinput:

```
  userinput:
  var myregexp = new RegExp(userinput);
```

Perl

```
  $myregex = qr/regex pattern/
```

Expressão regular recuperada a partir da entrada do usuário, como uma string armazenada na variável $userinput:

```
  $myregex = qr/$userinput/
```

Python

```
  reobj = re.compile("regex pattern")
```

Expressão regular recuperada a partir da entrada do usuário, como uma string armazenada na variável userinput:

```
  reobj = re.compile(userinput)
```

Ruby

Expressão regular literal em seu código:

```
  myregexp = /regex pattern/;
```

Expressão regular recuperada a partir da entrada do usuário, como

uma string armazenada na variável userinput:

```
myregexp = Regexp.new(userinput);
```

Discussão

Antes do mecanismo de expressão regular corresponder uma expressão regular a uma string, ela precisa ser compilada. A compilação acontece enquanto o aplicativo está sendo executado. O construtor de expressões regulares, ou a função de compilação, analisa a string que contém sua expressão regular, e a converte em uma estrutura de árvore, ou em uma máquina de estados. A função que executa, de fato, a correspondência do padrão percorrerá esta árvore, ou máquina de estados, conforme varre a string. As linguagens de programação que suportam expressões regulares literais fazem a compilação quando a execução chega no operador de expressão regular.

.NET

Em C# e VB.NET, a classe .NET System.Text.RegularExpressions.Regex armazena uma expressão regular compilada. O construtor mais simples recebe apenas um parâmetro: uma string que armazena sua expressão regular.

Se há um erro de sintaxe na expressão regular, o construtor Regex() levantará uma ArgumentException. A mensagem de exceção indica exatamente qual erro foi encontrado. É importante tratar essa exceção, caso a expressão regular seja fornecida pelo usuário de sua aplicação. Mostre a mensagem de exceção, e peça para o usuário corrigir a expressão regular. Se sua expressão regular for uma string literal inserida diretamente no código, você poderá omitir a captura da exceção, caso utilize uma ferramenta de cobertura de código para se certificar de que a linha é executada sem lançar uma exceção. Não há mudanças possíveis de estado ou modo que possam fazer com que a mesma expressão regular literal compile em uma situação, mas não compile em outra. Observe que, se houver um erro de sintaxe em sua expressão regular literal, a

exceção ocorrerá quando o aplicativo for executado, e não quando seu aplicativo for compilado.

Você deverá construir um objeto Regex se estiver usando a expressão regular dentro de um loop, ou repetidamente durante sua aplicação. Construir o objeto de expressão regular não envolve nenhum trabalho extra. Os membros estáticos da classe Regex, que recebem a expressão regular como um parâmetro de string, de qualquer forma constroem um objeto Regex internamente; então, você poderia muito bem fazê-lo em seu próprio código, mantendo uma referência ao objeto.

Se planeja usar a expressão regular apenas algumas vezes, você pode, então, empregar os membros estáticos da classe Regex para salvar uma linha de código. Os membros estáticos de Regex não jogam fora, imediatamente, o objeto de expressão regular construído internamente; ao invés disso, eles mantêm um cache das 15 expressões regulares usadas mais recentemente. Você pode alterar o tamanho do cache, ajustando a propriedade Regex.CacheSize. A pesquisa de cache é feita procurando por sua string de expressão regular no cache. Mas não exagere no uso do cache. Se precisar com frequência de muitos objetos de expressão regular, mantenha um cache próprio em que possa procurar de forma mais eficiente.

Java

No Java, a classe Pattern armazena uma expressão regular compilada. Você pode criar objetos desta classe com a fábrica de classes Pattern.compile(), que requer apenas um parâmetro: uma string com a sua expressão regular.

Se houver um erro de sintaxe na expressão regular, a fábrica Pattern.compile() levantará uma PatternSyntaxException. A mensagem de exceção indica, exatamente, qual erro foi encontrado. É importante tratar essa exceção, caso a expressão regular seja fornecida pelo usuário de seu aplicativo. Mostre a mensagem de exceção, e peça

para que o usuário corrija a expressão regular. Se sua expressão regular for uma string literal inserida diretamente no código, você poderá omitir a captura da exceção, caso utilize uma ferramenta de cobertura de código, para se certificar de que a linha é executada sem lançar uma exceção. Não há mudanças possíveis de estado ou de modo que possam fazer com que a mesma expressão regular literal compile em uma situação, mas não compile em outra. Observe que, se houver um erro de sintaxe na sua expressão regular literal, a exceção ocorrerá quando o aplicativo for executado, e não quando for compilado.

A menos que pretenda usar uma expressão regular apenas uma vez, você deve criar um objeto Pattern, em vez de usar os membros estáticos da classe String. Embora precise de algumas linhas extras, o código será executado de forma mais eficiente. As chamadas estáticas recompilam sempre a expressão regular. Na verdade, o Java oferece chamadas estáticas apenas para algumas tarefas muito básicas de expressão regular.

Um objeto Pattern apenas armazena uma expressão regular compilada. Ele não faz nenhum trabalho real. A correspondência da expressão regular é feita pela classe Matcher. Para criar um Matcher, chame o método matcher() em sua expressão regular compilada. Passe a string de assunto como o único argumento de matcher().

Você pode chamar matcher() quantas vezes quiser, para usar a mesma expressão regular em múltiplas strings. Você pode trabalhar, ao mesmo tempo, com várias classes Matcher que utilizem a mesma expressão regular, desde que você as mantenha em uma mesma thread. As classes Pattern e Matcher não são seguras para serem usadas em threads múltiplas. Se você quiser usar a mesma expressão regular em várias threads, chame Pattern.compile() em cada thread.

Se você terminou de aplicar uma expressão regular a uma string, e deseja aplicar essa mesma expressão regular em outra string, poderá reutilizar o objeto Matcher chamando reset(). Passe a próxima

string de assunto como o único argumento do método. Isso é mais eficiente do que criar um novo objeto Matcher. O reset() retorna o mesmo Matcher que você chamou, permitindo facilmente que você redefina, e use, um Matcher em uma linha de código, como, por exemplo regexMatcher.reset(nextString).find().

JavaScript

A notação para expressões regulares literais, apresentada na receita 3.2, cria um novo objeto de expressão regular. Para usar o mesmo objeto repetidamente, simplesmente o atribua a uma variável.

Se você possui uma expressão regular armazenada em uma variável de string (por exemplo, se você pediu ao usuário que digitasse uma expressão regular), use o construtor RegExp() para compilar a expressão regular. Observe que a expressão regular dentro da string não é delimitada por barras. Essas barras são parte da notação JavaScript para objetos RegExp literais, em vez de fazerem parte da expressão regular em si.

Já que é trivial atribuir uma expressão regular literal a uma variável, a maioria das soluções JavaScript, neste capítulo, omitem esta linha de código e utilizam diretamente a expressão regular literal. Ao utilizar a mesma expressão regular mais de uma vez em seu próprio código, você deve atribuí-la a uma variável e utilizar a variável, ao invés de colar a mesma expressão regular literal várias vezes em seu código. Isso aumenta o desempenho e torna seu código mais fácil de manter.

PHP

O PHP não fornece uma maneira de armazenar uma expressão regular compilada em uma variável. Sempre que você quiser fazer algo com uma expressão regular, terá de passá-la como uma string para uma das funções preg.

As funções preg mantêm um cache de até 4.096 expressões regulares compiladas. Embora a pesquisa de cache baseada em tabela hash não seja tão rápida quanto referenciar uma variável, o impacto no desempenho não chega a ser tão dramático quanto ter de recompilar várias vezes a mesma expressão regular.

Quando o cache estiver cheio, a expressão regular compilada há mais tempo será removida.

Perl

Você pode utilizar o operador “quote regex” (“regex entre aspas”) para compilar uma expressão regular e atribuí-la a uma variável. Ele usa a mesma sintaxe do operador de correspondência descrito na receita 3.1, exceto que ele começa com as letras qr, em vez da letra m.

O Perl, geralmente, é bem eficiente no reuso de expressões regulares previamente compiladas. Portanto, não utilizaremos qr// nos exemplos de código deste capítulo. Apenas a receita 3.5 demonstra seu uso.

qr// é útil quando você estiver interpolando variáveis na expressão regular, ou quando tiver recuperado toda a expressão regular como uma string (por exemplo, a partir da entrada de usuário). Com qr/$regexstring/, você pode controlar quando a expressão regular será recompilada, para refletir os novos conteúdos de $regexstring. m/$regexstring/ recompilaria a expressão regular sempre, enquanto m/$regexstring/o nunca a recompila. A receita 3.4 explica /o.

Python

A função compile() do módulo re do Python pega uma string contendo sua expressão regular e retorna um objeto com sua expressão regular compilada.

Você deve chamar compile() explicitamente, se pretende utilizar a mesma expressão regular de maneira repetida. Todas as funções no módulo re, primeiramente, chamam compile() e, em seguida, chamam a função desejada no objeto de expressão regular compilado.

A função compile() mantém uma referência para as últimas 100 expressões regulares que ela compilou. Isso reduz a recompilação de qualquer uma das últimas 100 expressões regulares utilizadas a um dicionário de pesquisas. Quando o cache encher, ele será

esvaziado completamente.

Se o desempenho não for problema, o cache funciona bem o suficiente para que você possa usar diretamente as funções do módulo re. Porém, quando o desempenho importa, seria uma boa ideia chamar compile().

Ruby

A notação de expressões regulares literais, apresentada na receita 3.2, já cria um novo objeto de expressão regular. Para usar o mesmo objeto repetidamente, simplesmente atribua-o a uma variável.

Se você tem uma expressão regular armazenada em uma variável de string (suponha, por exemplo, que você pediu para que o usuário digitasse uma expressão regular), use a fábrica Regexp.new() ou seu sinônimo Regexp.compile() para compilar a expressão regular. Observe que a expressão regular dentro da string não é delimitada por barras. Essas barras fazem parte da notação do Ruby para objetos Regexp literais, e não da expressão regular em si.

Já que é trivial atribuir uma expressão regular literal a uma variável, a maioria das soluções Ruby, neste capítulo, omite esta linha de código, usando diretamente a expressão regular literal. Ao utilizar a mesma expressão regular mais de uma vez, você deve atribuí-la a uma variável e utilizá-la várias vezes, em vez de colar a mesma expressão regular literal em seu código. Isso aumenta o desempenho e torna seu código mais fácil de manter.

Compilando uma Expressão Regular ao nível da CIL (Common Intermediate Language)

C#

```
Regex regexObj = new Regex("regex pattern", RegexOptions.Compiled);
```

VB.NET

```
Dim RegexObj As New Regex("regex pattern", RegexOptions.Compiled)
```

Discussão

Quando você constrói um objeto Regex em .NET sem passar

quaisquer opções, a expressão regular é compilada como descrita em “Discussão”. Se você passar RegexOptions.Compiled como um segundo parâmetro, para o construtor Regex(), a classe Regex faz algo bem diferente: ela compila sua expressão regular ao nível da CIL, também conhecida como MSIL. CIL significa Common Intermediate Language (Linguagem Intermediária Comum), linguagem de programação de baixo nível que se aproxima mais do Assembly do que do C# ou do Visual Basic. Todos os compiladores .NET produzem CIL. Na primeira vez em que o aplicativo for executado, o framework .NET compila a CIL para um código de máquina adequado ao computador do usuário.

A vantagem de compilar uma expressão regular com RegexOptions.Compiled é que ela pode ser até 10 vezes mais rápida do que uma expressão regular compilada sem esta opção.

A desvantagem é que essa compilação pode ser até duas ordens de magnitude mais lenta do que seria, simplesmente, analisar e dividir a string da expressão regular em uma árvore. O código CIL também se torna parte permanente de sua aplicação até que seja finalizado. O código CIL não entra na coleta de lixo.

Utilize RegexOptions.Compiled somente se uma expressão regular for muito complexa, ou se precisar processar tanto texto que o usuário acabe enfrentando uma espera perceptível durante as operações com a expressão regular. A compilação e a sobrecarga não valem a pena, em relação às expressões regulares que fazem seus trabalhos em uma fração de segundo.

Veja também:

Receitas 3.1, 3.2, e 3.4.

3.4 Definir opções das expressões regulares

Problema

Você deseja compilar uma expressão regular com todos os modos de correspondência disponíveis: espaçamento livre, não diferenciar maiúsculas e minúsculas, ponto corresponde a quebras de linha, e circunflexo e cifrão correspondem em quebras de linhas.

Solução

C#

```
Regex regexObj = new Regex("padrão regex",
  RegexOptions.IgnorePatternWhitespace | RegexOptions.IgnoreCase |
  RegexOptions.Singleline | RegexOptions.Multiline);
```

VB.NET

```
Dim RegexObj As New Regex("padrão regex",
  RegexOptions.IgnorePatternWhitespace Or RegexOptions.IgnoreCase Or
  RegexOptions.Singleline Or RegexOptions.Multiline)
```

Java

```
Pattern regex = Pattern.compile("padrão regex",
  Pattern.COMMENTS | Pattern.CASE_INSENSITIVE |
Pattern.UNICODE_CASE |
  Pattern.DOTALL | Pattern.MULTILINE);
```

JavaScript

Expressão regular literal em seu código:

```
var myregexp = /padrão regex/im;
```

Expressão regular obtida a partir da entrada de usuário, como uma string:

```
var myregexp = new RegExp(userinput, "im");
```

PHP

```
regexstring = '/padrão regex/simx';
```

Perl

```
m/padrão regex/simx;
```

Python

```
reobj = re.compile("padrão regex",
  re.VERBOSE | re.IGNORECASE |
  re.DOTALL | re.MULTILINE)
```

Ruby

Expressão regular literal em seu código:

```
myregexp = /padrão regex/mix;
```

Expressão regular obtida a partir da entrada de usuário, como uma string:

```
myregexp = Regexp.new(userinput,
  Regexp::EXTENDED or Regexp::IGNORECASE or
  Regexp::MULTILINE);
```

Discussão

Muitas das expressões regulares apresentadas neste livro, e aquelas que você vai encontrar em outros lugares, são escritas para serem utilizadas com certos modos de correspondência. Existem quatro modos básicos, que quase todos os sabores de expressões regulares modernos suportam. Infelizmente, alguns sabores utilizam nomes inconsistentes e confusos para as opções que implementam os modos. Utilizar os modos errados geralmente quebra a expressão regular.

Todas as soluções desta receita utilizam sinalizações ou as opções, fornecidas pela linguagem de programação ou classe de expressões regulares para definir os modos. Outra maneira de definir os modos é utilizar modificadores de modo dentro da expressão regular. Este procedimento substitui as opções ou sinalizações definidas fora da expressão regular.

.NET

O construtor `Regex()` aceita um segundo parâmetro opcional com opções de expressões regulares. Você pode encontrar as opções disponíveis na enumeração `RegexOptions`.

Espaçamento livre: `RegexOptions.IgnorePatternWhitespace`

Não diferenciar maiúsculas de minúsculas: RegexOptions.IgnoreCase

Ponto corresponde a quebras de linha: RegexOptions.Singleline

Circunflexo e cifrão correspondem em quebras de linha: RegexOptions.Multiline

Java

A fábrica de classes Pattern.compile() aceita um segundo parâmetro opcional com opções de expressões regulares. A classe Pattern define várias constantes que configuram várias opções. Você pode definir várias opções, combinando-as com o operador bitwise | (ou inclusivo).

Espaçamento livre: Pattern.COMMENTS

Não diferenciar maiúsculas de minúsculas: Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE

Ponto corresponde a quebras de linha: Pattern.DOTALL

Circunflexo e cifrão correspondem em quebras de linha: Pattern.MULTILINE

Há, na verdade, duas opções para não-diferenciação de maiúsculas e minúsculas, e você deve definir ambas para uma não-diferenciação completa. Se definir somente Pattern.CASE_INSENSITIVE, apenas as letras de A a Z serão correspondidas usando a não-diferenciação entre maiúsculas e minúsculas. Se você definir ambas as opções, todos os caracteres de todos os alfabetos serão correspondidos usando a não-diferenciação entre maiúsculas e minúsculas. A única razão para não usar Pattern.UNICODE_CASE é seu desempenho, no caso de se saber com antecedência que estará lidando somente com texto ASCII. Ao utilizar modificadores de modo dentro da expressão regular, utilize ‹(?i)›, para não-diferenciação ASCII, e ‹(?iu)›, para uma não-diferenciação completa.

JavaScript

Você pode especificar opções em JavaScript, anexando uma ou

mais letras de sinalização à literal RegExp após a barra que encerra a expressão regular. Ao se falar sobre estes marcadores na documentação, eles geralmente são escritos como /i e /m, mesmo que a sinalização, em si, seja formada somente pela letra. Nenhuma barra adicional deve ser acrescentada para especificar as sinalizações de modo da expressão regular.

Ao utilizar o construtor RegExp() para compilar uma string em uma expressão regular, você pode passar-lhe um segundo parâmetro opcional com sinalizações. O segundo parâmetro deve ser uma string com as letras das opções que você deseja definir. Não coloque barras na string.

Espaçamento livre: Não suportado pelo JavaScript.

Não diferenciar maiúsculas de minúsculas: /i

Ponto corresponde a quebras de linha: Não suportado pelo JavaScript.

Circunflexo e cifrão correspondem em quebras de linha: /m

PHP

A receita 3.1 explica que as funções preg do PHP demandam que expressões regulares literais sejam delimitadas por dois caracteres de pontuação, geralmente barras, e que o conjunto deve ser formatado como uma string literal. Você pode especificar opções de expressão regular anexando uma ou mais letras modificadoras ao final da string. Ou seja, as letras modificadoras vêm depois do delimitador de fechamento da expressão regular, mas ainda dentro das aspas simples ou duplas da string. Ao falar sobre esses modificadores na documentação, eles geralmente são escritos como /x, mesmo que a sinalização em si seja apenas a letra, e mesmo que o delimitador entre a expressão regular e os modificadores não seja necessariamente uma barra.

Espaçamento livre: /x

Não diferenciar maiúsculas de minúsculas: /i

Ponto corresponde a quebras de linha: /s

Circunflexo e cifrão correspondem em quebras de linha: /m

Perl

Você pode especificar opções de expressão regular, anexando uma ou mais letras modificadoras no final do operador de correspondência de padrão ou do operador de substituição. Ao comentar a respeito desses modificadores na documentação, eles geralmente são escritos como /x, mesmo que a sinalização em si seja formada apenas pela letra, e mesmo que o delimitador entre a expressão regular e os modificadores não seja necessariamente uma barra.

Espaçamento livre: /x

Não diferenciar maiúsculas de minúsculas: /i

Ponto corresponde a quebras de linha: /s

Circunflexo e cifrão correspondem em quebras de linha: /m

Python

A função compile() (explicada na receita anterior) aceita um segundo parâmetro opcional com opções de expressão regular. Você pode construir esse parâmetro utilizando o operador | para combinar as constantes definidas no módulo re. Muitas das outras funções do módulo re que recebem uma expressão regular literal como parâmetro também aceitam opções de expressão regular como um parâmetro final e opcional.

As constantes para as opções de expressão regular vêm em pares. Cada opção pode ser representada como uma constante com um nome completo, ou apenas como uma letra simples. Sua funcionalidade é equivalente. A única diferença é que o nome completo torna seu código mais fácil de ler por desenvolvedores que não estejam familiarizados com a sopa de letrinhas das opções de expressão regular. As opções básicas listadas nesta seção são as mesmas do Perl.

Espaçamento livre: re.VERBOSE ou re.X

Não diferenciar maiúsculas de minúsculas: re.IGNORECASE ou re.I

Ponto corresponde a quebras de linha: re.DOTALL ou re.S

Circunflexo e cifrão correspondem em quebras de linha: re.MULTILINE ou re.M

Ruby

No Ruby, você pode especificar opções anexando uma ou mais sinalizações de letras simples à literal Regexp, após a barra que encerra a expressão regular. Ao falar sobre essas sinalizações na documentação, elas geralmente são escritas como /i e /m, mesmo que o marcador em si seja formado apenas pela letra. Nenhuma barra adicional deve ser acrescentada para especificar sinalizações de modo de expressão regular.

Ao utilizar a fábrica Regexp.new() para compilar uma string em uma expressão regular, você pode passar um segundo parâmetro opcional com sinalizações para o construtor. O segundo parâmetro deve ser nil, para desativar todas as opções, ou uma combinação das constantes da classe Regexp, combinadas com o operador or.

Espaçamento livre: /r ou Regexp::EXTENDED

Não diferenciar maiúsculas de minúsculas: /i ou Regexp::IGNORECASE

Ponto corresponde a quebras de linha: /m ou Regexp::MULTILINE. O Ruby usa “m” e “linhas múltiplas” aqui, enquanto todos os outros sabores utilizam “s” ou “linha simples” para “ponto corresponde a quebras de linha”.

Circunflexo e cifrão correspondem em quebras de linha: No Ruby, o circunflexo e o cifrão sempre correspondem nas quebras de linha. Você não pode desativar esta opção. Use ‹\A› e ‹\Z› para corresponder no início ou no final da string de assunto.

Opções adicionais específicas para cada linguagem

.NET

RegexOptions.ExplicitCapture transforma todos os grupos, exceto os

grupos nomeados, em grupos de não-captura. Com esta opção, ‹(grupo)› é o mesmo que ‹(?:grupo)›. Se você sempre nomeia seus grupos de captura, ative esta opção para deixar sua expressão regular mais eficiente, sem a necessidade de usar a sintaxe ‹(?:grupo)›. Ao invés de usar RegexOptions.ExplicitCapture, você pode ativar esta opção colocando ‹(?n)› no início de sua expressão regular. Veja a receita 2.9 para estudar agrupamentos. A receita 2.11 explica os grupos nomeados.

Especifique RegexOptions.ECMAScript, se você estiver usando a mesma expressão regular em seu código .NET e no código JavaScript, e deseja certificar-se de que ele vai se comportar da mesma maneira. Este procedimento é bem útil quando você está desenvolvendo uma aplicação web em JavaScript no lado do cliente, e em ASP.NET no lado do servidor. O efeito mais importante é que, com esta opção, \w e \d ficam restritos aos caracteres ASCII, tal como no JavaScript.

Java

Uma opção exclusiva do Java é Pattern.CANON_EQ, que habilita a “equivalência canônica”. Como explicado na discussão sobre “Grafemas Unicode”, o Unicode fornece diferentes formas de representar caracteres com sinais diacríticos. Ao ativar esta opção, a expressão regular corresponderá a um caractere, mesmo que ele esteja codificado de forma diferente na string de assunto. Por exemplo, a expressão regular ‹\u00E0› corresponderá tanto a "\u00E0" quanto a "\u0061\u0300", porque são canonicamente equivalentes. Ambos aparecem como “à” quando exibidos na tela, indistinguíveis para o usuário final. Sem a equivalência canônica, a expressão regular ‹\u00E0› não corresponderá à string "\u0061\u0300". É assim que todos os outros sabores de expressão regular discutidos neste livro se comportam.

Por fim, Pattern.UNIX_LINES diz ao Java para tratar apenas ‹\n› como um caractere de quebra de linha quando se utiliza o ponto, o acento circunflexo e o cifrão. Por padrão, todas as quebras de linha

Unicode são tratadas como caracteres de quebra de linha.

JavaScript

Se você deseja aplicar uma expressão regular repetidamente na mesma string – por exemplo, para iterar todas as correspondências, ou para procurar e substituir todas as correspondências, em vez de apenas a primeira – especifique a sinalização /g (“global”).

PHP

/u diz ao PCRE para interpretar tanto a expressão regular quanto a string de assunto como strings UTF-8. Este modificador também permite tokens de expressão regular Unicode, como ‹\p{FFFF}› e ‹\p{L}›. Estes são explicados na receita 2.7. Sem este modificador, o PCRE trata cada byte como um caractere separado, e tokens de expressão regular Unicode causam erro.

/U inverte os comportamentos “mesquinho” e “preguiçoso”, no que diz respeito a adicionar um ponto de interrogação extra a um quantificador. Normalmente, ‹.*› é mesquinho, e ‹.*?› é preguiçoso. Com /U, ‹.*› é preguiçoso e ‹.*?› é mesquinho. Eu recomendo firmemente que você nunca utilize esse marcador, uma vez que irá confundir os programadores que lerão seu código mais tarde e que deixarão passar o modificador /U extra, exclusivo do PHP. Além disso, não confunda /U com /u se encontrá-lo no código de outra pessoa. Modificadores de expressão regular diferenciam maiúsculas e minúsculas.

Perl

Se você deseja aplicar uma expressão regular várias vezes na mesma string (por exemplo, para iterar todas as correspondências ou para pesquisar e substituir todas as correspondências, ao invés de apenas a primeira), especifique a sinalização /g (“global”).

Se você interpolar uma variável em uma expressão regular – por exemplo, m/I am $name/ – o Perl irá recompilar a expressão regular sempre que precisar ser usada, pois o conteúdo de $name pode ter

mudado. Você pode evitar isso com a sinalização /o. m/I am $name/o é compilado na primeira vez em que o Perl precisar usá-lo e, então, é reutilizado do jeito em que estiver. Se o conteúdo de $name mudar, a expressão regular não refletirá a mudança. Veja a receita 3.3, caso queira controlar quando a expressão regular for recompilada.

Python

O Python tem duas opções extras que mudam o sentido das extremidades de palavra (veja receita 2.6) e das classes de carateres abreviados ‹\w›, ‹\d›, e ‹\s›, bem como das suas contrapartes negativas (veja receita 2.3). Por padrão, esses tokens lidam apenas com letras ASCII, dígitos e espaços em branco.

A opção re.LOCALE, ou re.L, torna esses tokens dependentes da localidade atual. A localidade, em seguida, determina quais caracteres serão tratados como letras, dígitos e espaços em branco por esses tokens de expressão regular. Você deve especificar essa opção quando a string de assunto não for uma string Unicode, e quiser que certos caracteres, como letras com sinais diacríticos, sejam tratados como tal.

A opção re.UNICODE, ou re.U, torna esses tokens dependentes do padrão Unicode. Todos os caracteres que o Unicode define como letras, números e espaços em branco serão, então, tratados como tal, por estes tokens de expressão regular. Você deve especificar essa opção quando a string de assunto na qual estiver aplicando a expressão regular for uma string Unicode.

Ruby

A fábrica Regexp.new() aceita um terceiro parâmetro opcional, para selecionar a codificação de string que sua expressão regular suporta. Se você não especificar uma codificação para sua expressão regular, ela usará a mesma codificação dos arquivos-fontes. Na maioria das vezes, utilizar a codificação do arquivo-fonte é a coisa certa a fazer.

Para selecionar uma codificação explicitamente, passe um único caractere para este parâmetro. O parâmetro não faz diferença entre maiúsculas e minúsculas. Os valores possíveis são:

n
: Isso significa “nenhuma” (none). Cada byte na sua string é tratado como um caractere. Use isso para textos ASCII.

e
: Ativa a codificação “EUC” para as línguas do Extremo Oriente.

s
: Ativa a codificação japonesa “Shift-JIS”.

u
: Ativa o UTF-8, que utiliza de um a quatro bytes por caractere e suporta todos os idiomas do padrão Unicode (que inclui todas as línguas vivas, com qualquer grau de importância).

Ao usar uma expressão regular literal, você pode definir a codificação com os modificadores /n, /e, /s, e /u. Apenas um desses modificadores pode ser usado em uma única expressão regular. Eles podem ser usados em combinação com qualquer um dos modificadores /x, /i, e /m, ou com todos eles.

Não confunda /s no Ruby, com o /s, do Perl, Java ou .NET. No Ruby, /s força a codificação Shift-JIS. No Perl, e na maioria dos outros sabores de expressão regular, ele ativa o modo “ponto corresponde a quebras de linha”. No Ruby, você pode fazer isso com /m.

Veja também:

Os efeitos dos modos de correspondência são explicados, em detalhes, no capítulo 2. Aquelas seções também explicam o uso dos modificadores de modo dentro da expressão regular.

- **Espaçamento livre**: receita 2.18
- **Não diferenciar maiúsculas de minúsculas**: “Correspondência sem diferenciação entre maiúsculas e minúsculas”, receita 2.1
- **Ponto corresponde a quebras de linha**: receita 2.4
- **Circunflexo e cifrão correspondem em quebras de linha**: receita 2.5

As receitas 3.1 e 3.3 explicam como usar expressões regulares literais em seu código-fonte e como criar objetos de expressão regular. Você define as opções da expressão regular ao criá-la.

3.5 Testar se uma correspondência pode ser encontrada dentro de uma string de assunto

Problema

Você quer verificar se uma correspondência pode ser encontrada por uma expressão regular em uma string em particular. Uma correspondência parcial já é suficiente. Por exemplo, a expressão regular ‹regex□pattern› corresponde parcialmente a The regex pattern can be found. Você não se importa com os detalhes da correspondência, pois só quer saber se a expressão regular corresponde à string.

Solução

C#

No caso de testes rápidos, você pode usar a chamada estática:

```
bool foundMatch = Regex.IsMatch(subjectString, "regex pattern");
```

Se a expressão regular for fornecida pelo usuário final, você deve usar a chamada estática com um tratamento de exceção completo:

```
bool foundMatch = false;
try {
  foundMatch = Regex.IsMatch(subjectString, UserInput);
} catch (ArgumentNullException ex) {
  // Não pode passar null como expressão regular ou string de assunto
} catch (ArgumentException ex) {
  // Erro de sintaxe na expressão regular
}
```

Para usar a mesma expressão regular repetidamente, construa um objeto Regex:

```
Regex regexObj = new Regex("regex pattern");
bool foundMatch = regexObj.IsMatch(subjectString);
```

Se a expressão regular for fornecida pelo usuário final, você deve usar o objeto Regex com um tratamento de exceção completo:

```
bool foundMatch = false;
try {
  Regex regexObj = new Regex(UserInput);
  try {
    foundMatch = regexObj.IsMatch(subjectString);
  } catch (ArgumentNullException ex) {
    // Não pode passar null como expressão regular ou string de assunto
  }
} catch (ArgumentException ex) {
  // Erro de sintaxe na expressão regular
}
```

VB.NET

No caso de testes rápidos, você pode usar a chamada estática:

```
Dim FoundMatch = Regex.IsMatch(SubjectString, "regex pattern")
```

Se a expressão regular for fornecida pelo usuário final, você deve usar a chamada estática com um tratamento de exceção completo:

```
Dim FoundMatch As Boolean
Try
  FoundMatch = Regex.IsMatch(SubjectString, UserInput)
Catch ex As ArgumentNullException
  'Não pode passar Nothing como expressão regular ou string de assunto
Catch ex As ArgumentException
  'Erro de sintaxe na expressão regular
End Try
```

Para utilizar a mesma expressão regular repetidamente, construa um objeto Regex:

```
Dim RegexObj As New Regex("regex pattern")
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

A chamada a IsMatch() deve ter a SubjectString (ou seja, a string de assunto) como parâmetro único, e a chamada deve ser feita na instância RegexObj, em vez de ser feita na classe Regex:

```
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

Se a expressão regular for fornecida pelo usuário final, você deve usar o objeto Regex com um tratamento de exceção completo:

```
Dim FoundMatch As Boolean
Try
  Dim RegexObj As New Regex(UserInput)
  Try
    FoundMatch = Regex.IsMatch(SubjectString)
  Catch ex As ArgumentNullException
    'Não pode passar Nothing como expressão regular ou string de assunto
  End Try
Catch ex As ArgumentException
  'Erro de sintaxe na expressão regular
End Try
```

Java

A única forma de testar uma correspondência parcial é criando um Matcher:

```
Pattern regex = Pattern.compile("regex pattern");
Matcher regexMatcher = regex.matcher(subjectString);
boolean foundMatch = regexMatcher.find();
```

Se a expressão regular for fornecida pelo usuário final, você deve utilizar o tratamento de exceção:

```
boolean foundMatch = false;
try {
  Pattern regex = Pattern.compile(UserInput);
  Matcher regexMatcher = regex.matcher(subjectString);
  foundMatch = regexMatcher.find();
} catch (PatternSyntaxException ex) {
  // Erro de sintaxe na expressão regular
}
```

JavaScript

```
if (/regex pattern/.test(subject)) {
  // Correspondência bem sucedida
} else {
  // Tentativa de correspondência falhou
}
```

PHP

```
if (preg_match('/regex pattern/', $subject)) {
  # Correspondência bem sucedida
} else {
  # Tentativa de correspondência falhou
}
```

Perl

Com a string de assunto armazenada na variável especial $_:

```
if (m/regex pattern/) {
  # Correspondência bem sucedida
} else {
  # Tentativa de correspondência falhou
}
```

Com a string de assunto armazenada na variável $subject:

```
if ($subject =~ m/regex pattern/) {
  # Correspondência bem sucedida
} else {
  # Tentativa de correspondência falhou
}
```

Utilizando uma expressão regular pré-compilada:

```
$regex = qr/regex pattern/;
if ($subject =~ $regex) {
  # Correspondência bem sucedida
} else {
  # Tentativa de correspondência falhou
}
```

Python

No caso de testes rápidos, você pode usar a função global:

```
if re.search("regex pattern", subject):
  # Correspondência bem sucedida
else:
  # Tentativa de correspondência falhou
```

Para utilizar a mesma expressão regular repetidamente, utilize um objeto compilado:

```
reobj = re.compile("regex pattern")
if reobj.search(subject):
  # Correspondência bem sucedida
```

```
else:
  # Tentativa de correspondência falhou
```

Ruby

```
if subject =~ /regex pattern/
  # Correspondência bem sucedida
else
  # Tentativa de correspondência falhou
end
```

Este código faz exatamente a mesma coisa:

```
if /regex pattern/ =~ subject
  # Correspondência bem sucedida
else
  # Tentativa de correspondência falhou
end
```

Discussão

A tarefa mais básica de uma expressão regular é verificar se a string corresponde à expressão regular. Na maioria das linguagens de programação, uma correspondência parcial é o suficiente para a função de correspondência retornar um valor verdadeiro. A função de correspondência fará a varredura por toda a string de assunto, para ver se a expressão regular corresponde a qualquer parte dela. A função retorna verdadeiro tão logo uma correspondência for encontrada. Ela retorna falso somente quando alcança o final da string sem encontrar nenhuma correspondência.

Os exemplos de código nesta receita são úteis para verificar se uma string contém certos dados. Se você quiser verificar se uma string se encaixa completamente em um determinado padrão (por exemplo, para a validação de entrada), utilize a próxima receita.

C# e VB.NET

A classe `Regex` oferece quatro versões diferentes do método `IsMatch()`, duas das quais são estáticas. Isso torna possível chamar `IsMatch()` com parâmetros diferentes. A string de assunto será sempre o primeiro parâmetro. Esta é a string em que a expressão

regular tentará encontrar uma correspondência. O primeiro parâmetro não deve ser null. Se isso acontecer, IsMatch() lançará uma ArgumentNullException.

Você pode realizar o teste em uma única linha de código chamando Regex.IsMatch(), sem construir um objeto Regex. Basta passar a expressão regular como o segundo parâmetro, além de passar as opções da expressão regular como um terceiro parâmetro opcional. Se a sua expressão regular tiver um erro de sintaxe, uma ArgumentException será lançada pela IsMatch(). Se sua expressão regular for válida, a chamada retornará true se uma correspondência parcial for encontrada, ou false se nenhuma correspondência puder ser encontrada.

Se quiser usar a mesma expressão regular em várias strings, você pode tornar seu código mais eficiente, ao construir primeiro um objeto Regex e, em seguida, chamando IsMatch() neste objeto. O primeiro parâmetro, que armazena a string de assunto, é o único parâmetro obrigatório. Você pode especificar um segundo parâmetro opcional para indicar o índice do caractere em que a expressão regular deve iniciar a verificação. Essencialmente, o número que você passar como segundo parâmetro será o número de caracteres no início da string de assunto que a expressão regular deverá ignorar. Isso pode ser útil quando você já processou a string até certo ponto, mas quer verificar se o restante precisa ser processado. Se você especificar um número, ele deve ser maior ou igual a zero, e menor ou igual ao comprimento da string de assunto. Caso contrário, IsMatch() lança uma ArgumentOutOfRangeException.

As sobrecargas estáticas não permitem o uso do parâmetro que especifica onde a tentativa da expressão regular deve começar na string. Não há sobrecarga que permita dizer a IsMatch() que ela deve parar antes do final da string. Se quiser fazer isso, poderia chamar Regex.Match("subject", start, stop) e verificar a propriedade Success do objeto Match devolvido. Veja a receita 3.8 para mais detalhes.

Java

Para testar se uma expressão regular corresponde a uma string de maneira parcial ou total, instancie um objeto Matcher, tal como explicado na receita 3.3. Em seguida, chame o método find() em seu Matcher recém-criado ou redefinido.

Não chame String.matches(), Pattern.matches(), ou Matcher.matches(). Todos eles exigem que a expressão regular corresponda à string toda.

JavaScript

Para testar se uma expressão regular pode corresponder a uma dada parte de uma string, chame o método test() em sua expressão regular. Passe a string de assunto como o único parâmetro.

regexp.test() retorna true se a expressão regular corresponder parcial ou totalmente à string de assunto, e false em caso contrário.

PHP

A função preg_match() pode ser usada para várias finalidades. A maneira mais básica de chamá-la é com apenas os dois parâmetros obrigatórios: a string com a sua expressão regular e a string com o texto de assunto, na qual você deseja que a expressão regular faça uma pesquisa. preg_match() retorna 1 se uma correspondência puder ser encontrada, e 0 quando a expressão regular não puder corresponder ao assunto.

Receitas posteriores, neste capítulo, explicam os parâmetros opcionais, que você pode passar para preg_match().

Perl

No Perl, m// é, na verdade, um operador de expressão regular, e não um simples recipiente de expressão regular. Se você usar m// por si só, ele usará a variável $_ como string de assunto.

Se quiser usar o operador de correspondência no conteúdo de outra variável, use o operador de ligação =~ para associar o operador de expressão regular a sua variável. Vincular a expressão regular a

uma string a executa imediatamente. O operador de correspondência a padrões retorna verdadeiro se a expressão regular corresponder a parte da string de assunto, e falso se não corresponder.

Caso queira verificar se uma expressão regular não corresponde a uma string, você pode usar !~, a versão negada de =~.

Python

A função search() do módulo re pesquisa ao longo de uma string para descobrir se a expressão regular corresponde a parte dela. Passe a sua expressão regular como o primeiro parâmetro e a string de assunto como o segundo parâmetro. Você pode passar as opções de expressão regular no terceiro parâmetro opcional.

A função re.search() chama re.compile() e, em seguida, chama o método search() no objeto de expressão regular compilado. Este método aceita apenas um parâmetro: a string de assunto.

Se a expressão regular encontrar uma correspondência, search() retornará uma instância de MatchObject. Se a expressão regular não corresponder, search() retornará None. Ao avaliar o valor retornado em uma declaração if, MatchObject será avaliado como True, enquanto None será avaliado como False. Receitas posteriores, neste capítulo, mostram como usar as informações armazenadas em MatchObject.

Não confunda search() com match(). Você não pode usar match() para encontrar uma correspondência no meio de uma string. A próxima receita usa match().

Ruby

O operador =~ é o operador de correspondência a padrões. Coloque-o entre uma expressão regular e uma string, para encontrar a primeira correspondência da expressão regular. O operador retorna um valor inteiro, com a posição na qual a correspondência da expressão regular começa na string. Ele retorna nil, se nenhuma correspondência puder ser encontrada.

Esse operador é implementado em ambas as classes Regexp e String.

No Ruby 1.8, não importa qual classe você coloca à esquerda e qual você coloca à direita do operador. No Ruby 1.9, isso pode criar um efeito colateral, envolvendo grupos de captura nomeados. A receita 3.9 explica isso.

Em todos os outros trechos de código Ruby deste livro, colocamos a string de assunto à esquerda do operador =~, e a expressão regular à direita. Isso mantém a coerência com o Perl, do qual o Ruby emprestou a sintaxe =~, e evita a magia do Ruby 1.9 com grupos de captura nomeados, o que, normalmente, as pessoas não esperam.

Veja também:

Receitas 3.6 e 3.7.

3.6 Testar se uma regex corresponde totalmente à string de assunto

Problema

Você deseja verificar se uma string encaixa-se totalmente em um determinado padrão. Ou seja, você deseja verificar se a expressão regular com o padrão pode corresponder à string do começo ao fim. Por exemplo, se sua expressão regular for ‹regex□pattern›, ela corresponderá ao texto de entrada consistindo de regex pattern, mas não à string mais longa The regex pattern can be found.

Solução

C#

No caso de testes rápidos, você pode usar a chamada estática:

```
bool foundMatch = Regex.IsMatch(subjectString, @"\Aregex pattern\Z");
```

Para usar a mesma expressão regular repetidamente, construa um objeto Regex:

```
Regex regexObj = new Regex(@"\Aregex pattern\Z");
bool foundMatch = regexObj.IsMatch(subjectString);
```

VB.NET

No caso de testes rápidos, você pode usar a chamada estática:

```
Dim FoundMatch = Regex.IsMatch(SubjectString, "\Aregex pattern\Z")
```

Para usar a mesma expressão regular repetidamente, construa um objeto Regex:

```
Dim RegexObj As New Regex("\Aregex pattern\Z")
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

A chamada a IsMatch()deve ter SubjectString (ou seja, a string de assunto) como único parâmetro, e a chamada deve ser feita na instância RegexObj, e não na classe Regex:

```
Dim FoundMatch = RegexObj.IsMatch(SubjectString)
```

Java

Se quiser testar apenas uma string, você pode usar a chamada estática:

```
boolean foundMatch = subjectString.matches("regex pattern");
```

Se quiser usar a mesma expressão regular em múltiplas strings, compile sua expressão regular e crie um Matcher:

```
Pattern regex = Pattern.compile("regex pattern");
Matcher regexMatcher = regex.matcher(subjectString);
boolean foundMatch = regexMatcher.matches(subjectString);
```

JavaScript

```
if (/^regex pattern$/.test(subject)) {
  // Correspondência bem sucedida
} else {
  // Tentativa de correspondência falhou
}
```

PHP

```
if (preg_match('/\Aregex pattern\Z/', $subject)) {
  # Correspondência bem sucedida
} else {
  # Tentativa de correspondência falhou
}
```

Perl

```
if ($subject =~ m/\Aregex pattern\Z/) {
  # Correspondência bem sucedida
} else {
  # Tentativa de correspondência falhou
}
```

Python

No caso de testes rápidos, você pode usar a função global:

```
if re.match(r"regex pattern\Z", subject):
  # Correspondência bem sucedida
else:
  # Tentativa de correspondência falhou
```

Para usar a mesma expressão regular repetidamente, use um objeto compilado:

```
reobj = re.compile(r"regex pattern\Z")
if reobj.match(subject):
  # Correspondência bem sucedida
else:
  # Tentativa de correspondência falhou
```

Ruby

```
if subject =~ /\Aregex pattern\Z/
  # Correspondência bem sucedida
else
  # Tentativa de correspondência falhou
end
```

Discussão

Normalmente, uma correspondência de expressão regular bem sucedida diz que o padrão que você deseja está em *algum lugar* dentro do texto de assunto. Em muitas situações, você também quer se certificar de que a correspondência seja *completa*, com nada mais no texto de assunto. Provavelmente, a situação mais comum para uma correspondência completa seria validar a entrada de dados. Se um usuário digita um número de telefone ou endereço IP, mas inclui caracteres estranhos, você vai querer rejeitar a entrada.

As soluções que utilizam as âncoras ‹$› e ‹\Z› também funcionam

quando você está processando um arquivo linha por linha (receita 3.21); o mecanismo que estiver usando para recuperar as linhas deixa as quebras de linha no final das linhas. Como a receita 2.5 explica, essas âncoras também correspondem antes de uma quebra de linha final, essencialmente permitindo que ela seja ignorada.

Nas subseções a seguir, vamos explicar em detalhes as soluções adotadas por cada linguagem.

C# e VB.NET

A classe Regex do framework .NET não possui uma função para testar se uma expressão regular corresponde totalmente a uma string. A solução é adicionar a âncora de início de string, ‹\A›, no início da sua expressão regular, e a âncora de final de string, ‹\Z›, no final de sua expressão regular. Dessa forma, a expressão regular poderá corresponder apenas à string completa, não correspondendo em qualquer outro caso. Se sua expressão regular usa alternância, como em ‹one|two|three›, certifique-se de agrupar a alternância antes de adicionar as âncoras: ‹\A(?:one|two|three)\Z›.

Com sua expressão regular alterada para corresponder a strings inteiras, você pode usar o mesmo método IsMatch(), tal como descrito na receita anterior.

Java

O Java possui três métodos chamados matches(). Todos verificam se uma expressão regular pode corresponder totalmente à string. Esses métodos são uma maneira rápida de fazer a validação da entrada de dados, sem ter de colocar sua expressão regular entre âncoras de início e de final de string.

A classe String possui um método matches(), que aceita uma expressão regular como único parâmetro. Ela retorna true ou false, para indicar se a expressão regular pode corresponder a toda a string. A classe Pattern possui um método estático matches(), que aceita duas strings: a primeira é a expressão regular, e a segunda, a

string de assunto. Na verdade, você pode passar qualquer CharSequence como string de assunto para Pattern.matches(). Essa é a única razão para usar Pattern.matches(), em vez de String.matches().

Tanto String.matches() quanto Pattern.matches() recompilam sempre a expressão regular, chamando Pattern.compile("regex").matcher(subjectString).matches(). Como a expressão regular é recompilada toda vez, você deveria usar essas chamadas somente quando desejar usar a expressão regular uma única vez (por exemplo, para validar um campo em um formulário de entrada) ou quando a eficiência não for problema. Esses métodos não fornecem uma maneira de especificar opções de correspondência fora da expressão regular. Uma PatternSyntaxException será lançada se sua expressão regular tiver um erro de sintaxe.

Caso queira utilizar a mesma expressão regular para testar eficientemente várias strings, você deve compilar sua expressão regular, além de criar e reutilizar um Matcher, tal como explicado na receita 3.3. Em seguida, chame matches() em sua instância de Matcher. Essa função não recebe nenhum parâmetro, pois você já especificou a string de assunto ao criar ou redefinir o Matcher.

JavaScript

O JavaScript não possui uma função para testar se uma expressão regular corresponde totalmente a uma string. A solução é acrescentar ‹^› no início de sua expressão regular, e ‹$› no final. Certifique-se de não definir a sinalização /m em sua expressão regular. Quando a sinalização /m não estiver definida, o circunflexo e o cifrão corresponderão apenas no início e no final da string de assunto. Quando você define a sinalização /m, eles também corresponderão em quebras de linha no meio da string.

Com as âncoras adicionadas em sua expressão regular, você poderá utilizar o mesmo método regexp.test(), descrito na receita anterior.

PHP

O PHP não possui função para testar se uma expressão regular corresponde totalmente a uma string. A solução é adicionar a âncora de início de string, ‹\A›, no início de sua expressão regular, e a âncora de final de string, ‹\Z›, no final. Dessa forma, a expressão regular poderá corresponder apenas à string completa, não correspondendo em qualquer outro caso. Se a sua expressão regular utiliza alternância, como em ‹one|two|three›, certifique-se de agrupar a alternância antes de adicionar as âncoras: ‹\A(?:one|two|three)\Z›.

Com sua expressão regular alterada, para corresponder apenas a strings inteiras, você pode utilizar a mesma função preg_match(), tal como descrito na receita anterior.

Perl

O Perl possui somente um operador de correspondência a padrões, que fica satisfeito com correspondências parciais. Caso você queira verificar se sua expressão regular corresponde a toda a string de assunto, adicione a âncora de início de string, ‹\A›, no início de sua expressão regular, e a âncora de final de string, ‹\Z›, no final. Dessa forma, a expressão regular poderá corresponder apenas à string completa, não correspondendo em qualquer outro caso. Se sua expressão regular utiliza alternância, como em ‹one|two|three›, certifique-se de agrupar a alternância antes de adicionar as âncoras: ‹\A(?:one|two|three)\Z›.

Com sua expressão regular alterada para corresponder apenas a strings inteiras, utilize-a como descrito na receita anterior.

Python

A função match() é muito semelhante à função search(), descrita na receita anterior. A diferença fundamental é que match() avalia a expressão regular apenas no início da string de assunto. Se a expressão regular não corresponder ao início da string, match() retorna None imediatamente. A função search(), no entanto, continuará tentando a expressão regular em cada posição sucessiva

na string, até encontrar uma correspondência, ou até atingir o final da string de assunto.

A função match() não exige que a expressão regular corresponda a toda a string. Uma correspondência parcial é aceita, contanto que comeÇe no início da string. Caso queira verificar se a expressão regular pode corresponder à string inteira, anexe a âncora de final de string, ‹\Z›, à sua expressão regular.

Ruby

A classe Regexp do Ruby não possui função para testar se uma expressão regular corresponde totalmente a uma string. A solução é adicionar a âncora de início de string, ‹\A›, no início de sua expressão regular, e a âncora de final de string, ‹\Z›, no final. Dessa forma, a expressão regular poderá corresponder apenas à string completa, não correspondendo em qualquer outro caso. Se sua expressão regular utiliza alternância, como em ‹one|two|three›, certifique-se de agrupar a alternância antes de adicionar as âncoras: ‹\A(?:one|two|three)\Z›.

Com sua expressão regular alterada para corresponder apenas a strings inteiras, você poderá utilizar o mesmo operador =~, tal como descrito na receita anterior.

Veja também:

A receita 2.5 explica, em detalhes, como as âncoras funcionam.

As receitas 2.8 e 2.9 explicam alternância e agrupamento. Se a sua expressão regular usa alternância fora de qualquer grupo, você precisa agrupá-la, antes de adicionar âncoras. Se sua expressão regular não usa alternância, ou se usa alternância apenas dentro de grupos, nenhum agrupamento extra será necessário para fazer as âncoras funcionarem como previsto.

Siga a receita 3.5 quando resultados parciais forem aceitáveis.

3.7 Recuperar o texto correspondido

Problema

Você possui uma expressão regular que corresponde a uma parte do texto de assunto, e você deseja extrair o texto que foi correspondido. Se a expressão regular puder corresponder à string mais de uma vez, você vai querer apenas a primeira correspondência. Por exemplo, ao aplicar a expressão regular ‹\d+› na string Do you like 13 or 42?, 13 deverá ser retornado.

Solução

C#

No caso de correspondências rápidas, você pode usar a chamada estática:

```
string resultString = Regex.Match(subjectString, @"\d+").Value;
```

Se a expressão regular for fornecida pelo usuário final, você deve usar a chamada estática com um tratamento de exceção completo:

```
string resultString = null;
try {
  resultString = Regex.Match(subjectString, @"\d+").Value;
} catch (ArgumentNullException ex) {
  // Não pode passar null como expressão regular ou string de assunto
} catch (ArgumentException ex) {
  // Erro de sintaxe na expressão regular
}
```

Para usar a mesma expressão regular repetidamente, construa um objeto Regex:

```
Regex regexObj = new Regex(@"\d+");
string resultString = regexObj.Match(subjectString).Value;
```

Se a expressão regular for fornecida pelo usuário final, você deve usar o objeto Regex com um tratamento de exceção completo:

```
string resultString = null;
try {
  Regex regexObj = new Regex(@"\d+");
  try {
    resultString = regexObj.Match(subjectString).Value;
  } catch (ArgumentNullException ex) {
```

```
    // Não pode passar null como string de assunto
  }
} catch (ArgumentException ex) {
  // Erro de sintaxe na expressão regular
}
```

VB.NET

No caso de correspondências rápidas, você pode usar a chamada estática:

```
Dim ResultString = Regex.Match(SubjectString, "\d+").Value
```

Se a expressão regular for fornecida pelo usuário final, você deve usar o objeto Regex com um tratamento de exceção completo:

```
Dim ResultString As String = Nothing
Try
  ResultString = Regex.Match(SubjectString, "\d+").Value
Catch ex As ArgumentNullException
  'Não pode passar Nothing como expressão regular ou string de assunto
Catch ex As ArgumentException
  'Erro de sintaxe na expressão regular
End Try
```

Para usar a mesma expressão regular repetidamente, construa um objeto Regex:

```
Dim RegexObj As New Regex("\d+")
Dim ResultString = RegexObj.Match(SubjectString).Value
```

Se a expressão regular for fornecida pelo usuário final, você deve usar o objeto Regex com um tratamento de exceção completo:

```
Dim ResultString As String = Nothing
Try
  Dim RegexObj As New Regex("\d+")
  Try
    ResultString = RegexObj.Match(SubjectString).Value
  Catch ex As ArgumentNullException
    'Não pode passar Nothing como string de assunto
  End Try
  Catch ex As ArgumentException
    'Erro de sintaxe na expressão regular
  End Try
```

Java

Crie um Matcher para executar a pesquisa, e armazene o resultado:

```
String resultString = null;
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
if (regexMatcher.find()) {
  resultString = regexMatcher.group();
}
```

Se a expressão regular for fornecida pelo usuário final, você deve usar um tratamento de exceção completo:

```
String resultString = null;
try {
  Pattern regex = Pattern.compile("\\d+");
  Matcher regexMatcher = regex.matcher(subjectString);
  if (regexMatcher.find()) {
    resultString = regexMatcher.group();
  }
} catch (PatternSyntaxException ex) {
  // Erro de sintaxe na expressão regular
}
```

JavaScript

```
var result = subject.match(/\d+/);
if (result) {
  result = result[0];
} else {
  result = '';
}
```

PHP

```
if (preg_match('/\d+/', $subject, $groups)) {
  $result = $groups[0];
} else {
  $result = '';
}
```

Perl

```
if ($subject =~ m/\d+/) {
  $result = $&;
```

```
} else {
  $result = '';
}
```

Python

No caso de correspondências rápidas, você pode usar a função global:

```
matchobj = re.search("padrão regex", subject)
if matchobj:
  result = matchobj.group()
else:
  result = ""
```

Para usar a mesma regex repetidamente, use um objeto compilado:

```
reobj = re.compile("padrão regex")
matchobj = reobj.search(subject)
if match:
  result = matchobj.group()
else:
  result = ""
```

Ruby

Você pode usar o operador `=~` e sua variável mágica `$&`:

```
if subject =~ /padrão regex/
  result = $&
else
  result = ""
end
```

Alternativamente, você pode chamar o método `match` em um objeto `Regexp`:

```
matchobj = /padrão regex/.match(subject)
if matchobj
  result = matchobj[0]
else
  result = ""
end
```

Discussão

Extrair parte de uma string longa, que se encaixa no padrão, é outra

tarefa primordial das expressões regulares. Todas as linguagens de programação discutidas neste livro fornecem uma maneira fácil de obter a primeira correspondência da expressão regular a partir de uma string. A função tentará a expressão regular no início da string e continuará varrendo os caracteres ao longo da string, até que a expressão regular corresponda.

.NET

A classe Regex do .NET não possui um membro que retorne a string correspondida pela expressão regular. Porém, possui um método Match() que retorna uma instância da classe Match. Este objeto Match possui uma propriedade chamada Value, que contém o texto correspondido pela expressão regular. Se a expressão regular não corresponder, ainda assim ela retornará um objeto Match, mas a propriedade Value armazenará uma string vazia.

Um total de cinco sobrecargas permite que você chame o método Match() de várias maneiras. O primeiro parâmetro é sempre a string que contém o texto de assunto no qual você deseja que a expressão regular encontre uma correspondência. Este parâmetro não deve ser null. Se isso acontecer, Match() lançará uma ArgumentNullException.

Se quiser usar a expressão regular apenas algumas vezes, você poderá utilizar uma chamada estática. O segundo parâmetro será, então, a expressão regular que você deseja utilizar. Você pode passar opções de expressão regular como um terceiro parâmetro opcional. Se sua expressão regular tiver um erro de sintaxe, uma ArgumentException será lançada.

Se desejar utilizar a mesma expressão regular em várias strings, é possível tornar seu código mais eficiente, primeiro construindo um objeto Regex e, então, chamando Match() nesse objeto. O primeiro parâmetro com a string de assunto será, então, o único parâmetro obrigatório. Você pode especificar um segundo parâmetro opcional para indicar o índice do caractere em que a expressão regular deverá começar a busca. Essencialmente, o número que você

passar como segundo parâmetro será o número de caracteres, no início da string de assunto, que a expressão regular deverá ignorar. Isso poderá ser útil no caso de você ter processado a string até certo ponto, desejando, então, pesquisar o restante da string. Se você especificar este número, ele deverá estar entre zero e o comprimento da string de assunto. Caso contrário, IsMatch() lançará uma ArgumentOutOfRangeException.

Caso especifique o segundo parâmetro com a posição inicial, você poderá especificar um terceiro parâmetro, indicando o comprimento da substring em que a expressão regular está autorizada a pesquisar. Esse número deve ser maior ou igual a zero, e não deve exceder o comprimento da string de assunto (primeiro parâmetro) menos o deslocamento inicial (segundo parâmetro). Por exemplo, regexObj.Match("123456", 3, 2) tenta encontrar uma correspondência em "45". Se o terceiro parâmetro for maior do que o comprimento da string de assunto, Match() lançará uma ArgumentOutOfRangeException. Se o terceiro parâmetro não for maior do que o comprimento da string de assunto, mas a soma dos segundo e terceiro parâmetros for maior do que o comprimento da string, então, outra IndexOutOfRangeException será lançada. Se você permitir que o usuário especifique as posições inicial e final, verifique-as antes de chamar Match() ou certifique-se de capturar ambas as exceções do tipo fora-de-intervalo.

As sobrecargas estáticas não permitem o uso dos parâmetros que especificam em qual parte da string a expressão regular poderá pesquisar.

Java

Para obter a parte de uma string correspondida por uma expressão regular, você precisa criar um Matcher, tal como explicado na receita 3.3. Em seguida, chame o método find() em seu Matcher, sem qualquer parâmetro. Se find() retornar true, chame group(), sem qualquer parâmetro para recuperar o texto correspondido por sua expressão regular. Se find() retornar false, você não deve chamar

group(), pois tudo que você vai conseguir é uma IllegalStateException.

Matcher.find() recebe um parâmetro opcional com a posição inicial na string de assunto. Você pode usar esse procedimento para iniciar a busca em uma determinada posição da string. Especifique zero para iniciar a tentativa de correspondência no início da string. Uma IndexOutOfBoundsException é lançada se você definir a posição inicial como um número negativo, ou como um número maior que o comprimento da string de assunto.

Se você omitir o parâmetro, find() iniciará no caractere subsequente à correspondência anterior encontrada por find(). Se estiver chamando find() pela primeira vez após Pattern.matcher() ou Matcher.reset(), então find() começará a pesquisar no início da string.

JavaScript

O método string.match() recebe uma expressão regular como único parâmetro. Você pode passar a expressão regular como literal, um objeto de expressão regular ou como uma string. Se você passar uma string, string.match() cria um objeto regexp temporário.

Quando a tentativa de correspondência falha, string.match() retorna null. Isso permite diferenciar entre uma expressão regular que não encontrou correspondência e uma que encontrou uma correspondência de comprimento zero. Isso significa que você não pode exibir diretamente o resultado, já que poderá aparecer “null” ou um erro relativo a um objeto nulo.

Quando a tentativa de correspondência for bem sucedida, string.match() retornará um array com os detalhes da correspondência. O elemento zero do array é uma string, contendo o texto correspondido pela expressão regular.

Certifique-se de não adicionar a sinalização /g em sua expressão regular. Se o fizer, string.match() vai se comportar de forma diferente, como a receita 3.10 explica.

PHP

A função preg_match(), discutida nas duas últimas receitas, aceita um terceiro parâmetro opcional, que armazenará o texto correspondido pela expressão regular e seus grupos de captura. Quando preg_match() retorna 1, a variável contém um array de strings. O elemento zero, no array, armazena a correspondência geral da expressão regular. Os demais elementos são explicados na receita 3.9.

Perl

Quando o operador de correspondência a padrões m// encontra uma correspondência, ele define diversas variáveis especiais. Uma delas é a $&, que contém a parte da string correspondida pela expressão regular. As demais variáveis especiais são explicadas em receitas posteriores.

Python

A receita 3.5 explica a função search(). Desta vez, armazenamos a instância de MatchObject, retornada por search() em uma variável. Para obter a parte da string correspondida pela expressão regular, chamamos o método group() sem parâmetros no objeto correspondido.

Ruby

A receita 3.8 explica a variável $~ e o objeto MatchData. Em um contexto de string, esse objeto é avaliado como o texto correspondido pela expressão regular. Em um contexto de array, este objeto é avaliado como um array com o elemento zero, contendo a correspondência geral da expressão regular.

$& é uma variável especial somente para leitura. É um apelido para $~[0], que contém uma string com o texto correspondido pela expressão regular.

Veja também:

Receitas 3.5, 3.8, 3.9, 3.10 e 3.11.

3.8 Determinar a posição e o comprimento da correspondência

Problema

Ao invés de extrair a substring correspondida pela expressão regular, como mostrado na receita anterior, você deseja determinar a posição inicial e o comprimento da correspondência. Com essas informações, você pode extrair a correspondência em seu próprio código ou aplicar o processamento que quiser na parte da string original, correspondida pela expressão regular.

Solução

C#

No caso de correspondências rápidas, você pode usar a chamada estática:

```
int matchstart, matchlength = -1;
Match matchResult = Regex.Match(subjectString, @"\d+");
if (matchResult.Success) {
  matchstart = matchResult.Index;
  matchlength = matchResult.Length;
}
```

Para usar a mesma expressão regular repetidamente, construa um objeto Regex:

```
int matchstart, matchlength = -1;
Regex regexObj = new Regex(@"\d+");
Match matchResult = regexObj.Match(subjectString).Value;
if (matchResult.Success) {
  matchstart = matchResult.Index;
  matchlength = matchResult.Length;
}
```

VB.NET

No caso de correspondências rápidas, você pode usar a chamada estática:

```
Dim MatchStart = -1
Dim MatchLength = -1
Dim MatchResult = Regex.Match(SubjectString, "\d+")
If MatchResult.Success Then
  MatchStart = MatchResult.Index
  MatchLength = MatchResult.Length
End If
```

Para usar a mesma expressão regular repetidamente, construa um objeto Regex:

```
Dim MatchStart = -1
Dim MatchLength = -1
Dim RegexObj As New Regex("\d+")
Dim MatchResult = Regex.Match(SubjectString, "\d+")
If MatchResult.Success Then
  MatchStart = MatchResult.Index
  MatchLength = MatchResult.Length
End If
```

Java

```
int matchStart, matchLength = -1;
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
if (regexMatcher.find()) {
  matchStart = regexMatcher.start();
  matchLength = regexMatcher.end() - matchStart;
}
```

JavaScript

```
var matchstart = -1;
var matchlength = -1;
var match = /\d+/.exec(subject);
if (match) {
  matchstart = match.index;
  matchlength = match[0].length;
}
```

PHP

```
if (preg_match('/\d+/', $subject, $groups, PREG_OFFSET_CAPTURE)) {
  $matchstart = $groups[0][1];
  $matchlength = strlen($groups[0][0]);
```

```
}
```

Perl

```
if ($subject =~ m/\d+/g) {
  $matchlength = length($&);
  $matchstart = length($`);
}
```

Python

No caso de correspondências rápidas, você pode usar a função global:

```
matchobj = re.search(r"\d+", subject)
if matchobj:
  matchstart = matchobj.start()
  matchlength = matchobj.end() - matchstart
```

Para usar a mesma expressão regular repetidamente, utilize um objeto compilado:

```
reobj = re.compile(r"\d+")
matchobj = reobj.search(subject)
if matchobj:
  matchstart = matchobj.start()
  matchlength = matchobj.end() - matchstart
```

Ruby

Você pode usar o operador `=~` e sua variável mágica `$~`:

```
if subject =~ /padrão regex/
  matchstart = $~.begin()
  matchlength = $~.end() - matchstart
end
```

Alternativamente, você pode chamar o método `match` em um objeto `Regexp`:

```
matchobj = /padrão regex/.match(subject)
if matchobj
  matchstart = matchobj.begin()
  matchlength = matchobj.end() - matchstart
end
```

Discussão

.NET

Para obter o índice e o comprimento da correspondência, usamos o mesmo método Regex.Match() descrito na receita anterior. Dessa vez, usamos as propriedades Index e Length do objeto Match, retornado por Regex.Match().

O Index é o índice da string de assunto no qual começa a correspondência da expressão regular. Se a correspondência da expressão regular começar no início da string, Index será zero. Se a correspondência começar no segundo caractere da string, Index será um. O valor máximo para Index é o comprimento da string. Isso pode acontecer quando a expressão regular encontrar uma correspondência de comprimento zero no final da string. Por exemplo, a expressão regular constituída exclusivamente da âncora de final de string, ‹\Z›, sempre corresponderá no final da string.

Length indica o número de caracteres correspondidos. É possível que uma correspondência válida tenha um comprimento de zero caractere. Por exemplo, a expressão regular que consista apenas do token de extremidade de palavra, ‹\b›, vai encontrar uma correspondência de comprimento zero, no início da primeira palavra da string.

Se a tentativa de correspondência falhar, Regex.Match() ainda retorna um objeto Match. Suas propriedades Index e Length terão valor zero. Esses valores também podem surgir em uma correspondência bem sucedida. A expressão regular contendo a âncora de início de string, ‹\A›, vai encontrar uma correspondência de comprimento zero no início da string. Assim, não confie em Match.Index, ou em Match.Length, para indicar se a tentativa de correspondência foi bem sucedida. Em vez disso, utilize Match.Success.

Java

Para obter a posição e o comprimento da correspondência, chame Matcher.find(), tal como descrito na receita anterior. Quando find() retornar true, chame Matcher.start(), sem qualquer parâmetro, para

obter o índice do primeiro caractere, que faz parte da correspondência da expressão regular. Chame end(), sem qualquer parâmetro, para obter o índice do primeiro caractere após a correspondência. Subtraia o início do final, para obter o comprimento da correspondência, que pode, inclusive, ser zero. Se você chamar start() e end() sem uma chamada prévia para find(),receberá uma IllegalStateException.

JavaScript

Chame o método exec() em um objeto regexp para obter um array com detalhes sobre a correspondência. Esse array possui algumas propriedades adicionais. A propriedade index armazena a posição na string de assunto onde começa a correspondência da expressão regular. Se a correspondência começar no início da string, index será zero. O elemento zero do array contém uma string com a correspondência global da expressão regular. Obtenha a propriedade length dessa string para determinar o comprimento da correspondência.

Se a expressão regular não puder corresponder à string, regexp.exec() retornará null.

Não use a propriedade lastIndex do array retornado por exec() para determinar a posição final da correspondência. Em uma implementação JavaScript estrita, a propriedade lastIndex não existe no array retornado, existindo só no próprio objeto regexp. Você também não deve usar regexp.lastIndex. Ele não é confiável, devido às diferenças entre navegadores (veja receita 3.11, para mais detalhes). Em vez disso, simplesmente some match.index e match[0].length, para determinar onde a correspondência da expressão regular terminou.

PHP

A receita anterior explica como você pode obter o texto correspondido pela expressão regular passando um terceiro parâmetro para preg_match(). Você pode obter a posição da

correspondência passando a constante PREG_OFFSET_CAPTURE como um quarto parâmetro. Este parâmetro muda o que preg_match() armazena no terceiro parâmetro quando a função retorna 1.

Ao omitir o quarto parâmetro, ou defini-lo como zero, a variável passada como terceiro parâmetro receberá um array de strings. Ao passar PREG_OFFSET_CAPTURE como quarto parâmetro, a variável receberá um array de arrays. O elemento zero, no array global, ainda armazena a correspondência geral (veja a receita anterior), e os elementos um e além ainda estão capturando os grupos um e além (veja a receita a seguir). Mas, em vez de conter uma string com o texto correspondido pela expressão regular ou um grupo de captura, o elemento contém um array com dois valores: o texto correspondido e a posição na string em que foi correspondido.

Para obter os detalhes da correspondência global, o subelemento zero do elemento zero nos dá o texto correspondido pela expressão regular. Nós passamos esse resultado para a função strlen(), para calcular seu comprimento. O subelemento um do elemento zero armazena um número inteiro, com a posição na string de assunto em que começa a correspondência.

Perl

Para obter o comprimento da correspondência, nós simplesmente calculamos o comprimento da variável $&, que armazena a correspondência geral da expressão regular. Para obtermos o início da correspondência, calculamos o comprimento da variável $`, que contém o texto da string anterior à correspondência da expressão regular.

Python

O método start() de MatchObject retorna a posição na string em que começa a expressão regular. O método end() retorna a posição do primeiro caractere após a correspondência. Ambos os métodos retornam o mesmo valor, quando uma correspondência da expressão regular de comprimento zero é encontrada.

Você pode passar um parâmetro para start() e end(), para recuperar o intervalo de texto correspondido por um dos grupos de captura nas expressões regulares. Chame start(1) para o primeiro grupo de captura, end(2) para o segundo grupo, e assim por diante. O Python suporta até 99 grupos de captura. O grupo de número 0 é a correspondência total da expressão regular. Qualquer número diferente de zero, até o número dos grupos de captura na expressão regular (com 99 sendo o limite máximo), fará com que start() e end() levantem uma exceção IndexError. Se o número do grupo for válido, mas o grupo, em si, não participou da correspondência, start() e end() retornarão -1.

Se você deseja armazenar as posições inicial e final em uma tupla, chame o método span() no objeto de correspondência.

Ruby

A receita 3.5 usa o operador =~ para encontrar a primeira correspondência da expressão regular em uma string. Um efeito colateral deste operador é que ele preenche a variável especial $~ com uma instância da classe MatchData. Esta variável é local, tanto em relação à thread quanto em relação ao método. Isso significa que você pode usar o conteúdo dessa variável enquanto seu método não terminar, ou até a próxima vez em que for utilizar o operador =~ dentro de seu método, sem se preocupar se outra thread, ou outro método na mesma thread, irá substituí-la.

Se você quiser manter os detalhes de múltiplas correspondências da expressão regular, chame o método match() em um objeto Regexp. Este método aceita uma string de assunto como único parâmetro. Ele retorna uma instância de MatchData quando uma correspondência puder ser encontrada, ou nil, em caso contrário. Ele também define a variável $~ para a mesma instância de MatchObject, mas não substitui outras instâncias de MatchObject armazenadas em outras variáveis.

O objeto MatchData armazena todas as informações de uma expressão regular. As receitas 3.7 e 3.9 explicam como obter o texto

correspondido pela expressão regular e pelos grupos de captura.

O método begin() retorna a posição na string de assunto onde começa a correspondência da expressão regular. end() retorna a posição do primeiro caractere após a correspondência da expressão regular. offset() retorna um array com as posições inicial e final. Estes três métodos aceitam um parâmetro. Passe 0 para obter as posições da correspondência global da expressão regular, ou passe um número positivo para obter as posições do grupo de captura especificado. Por exemplo, begin(1) retorna o início do primeiro grupo de captura.

Não use length() ou size() para obter o comprimento da correspondência. Ambos os métodos retornam o número de elementos de MatchData, pois, como explicado na receita 3.9, em um contexto de array, MatchData comporta-se como um array.

Veja também:

Receitas 3.5 e 3.9.

3.9 Recuperar parte do texto correspondido

Problema

Tal como na receita 3.7, você possui uma expressão regular que corresponde a uma substring do texto de assunto, mas, dessa vez, você deseja corresponder apenas a uma parte dessa substring. Para isolar a parte desejada, você adicionou um grupo de captura à expressão regular, como descrito na receita 2.9.

Por exemplo, a expressão regular ‹http://([a-z0-9.-]+)› corresponde a http://www.regexcookbook.com na string Please visit http://www.regexcookbook.com for more information. A parte da expressão regular, dentro do primeiro grupo de captura, corresponde a www.regexcookbook.com, e você deseja recuperar em uma variável de

string o nome de domínio capturado pelo primeiro grupo de captura.

Estamos utilizando essa expressão regular simples para ilustrar o conceito de grupos de captura. Consulte o capítulo 7, para ver expressões regulares mais precisas de correspondência a URLs.

Solução

C#

No caso de correspondências rápidas, você pode usar a chamada estática:

```
string resultString = Regex.Match(subjectString,"http://([a-z0-
9.-]+)").Groups[1].Value;
```

Para usar a mesma expressão regular repetidamente, construa um objeto Regex:

```
Regex regexObj = new Regex("http://([a-z0-9.-]+)");
string resultString = regexObj.Match(subjectString).Groups[1].Value;
```

VB.NET

No caso de correspondências rápidas, você pode usar a chamada estática:

```
Dim ResultString = Regex.Match(SubjectString, "http://([a-z0-
9.-]+)").Groups(1).Value
```

Para usar a mesma expressão regular repetidamente, construa um objeto Regex:

```
Dim RegexObj As New Regex("http://([a-z0-9.-]+)")
Dim ResultString = RegexObj.Match(SubjectString).Groups(1).Value
```

Java

```
String resultString = null;
Pattern regex = Pattern.compile("http://([a-z0-9.-]+)");
Matcher regexMatcher = regex.matcher(subjectString);
if (regexMatcher.find()) {
  resultString = regexMatcher.group(1);
}
```

JavaScript

```
var result = "";
var match = /http:\/\/([a-z0-9.-]+)/.exec(subject);
if (match) {
  result = match[1];
} else {
  result = '';
}
```

PHP

```
if (preg_match('%http://([a-z0-9.-]+)%', $subject, $groups)) {
  $result = $groups[1];
} else {
  $result = '';
}
```

Perl

```
if ($subject =~ m!http://([a-z0-9.-]+)!) {
  $result = $1;
} else {
  $result = '';
}
```

Python

No caso de correspondências rápidas, você pode usar a função global:

```
matchobj = re.search("http://([a-z0-9.-]+)", subject)
if matchobj:
  result = matchobj.group(1)
else:
  result = ""
```

Para usar a mesma expressão regular repetidamente, use um objeto compilado:

```
reobj = re.compile("http://([a-z0-9.-]+)")
matchobj = reobj.search(subject)
if match:
  result = matchobj.group(1)
else:
  result = ""
```

Ruby

Você pode usar o operador =~ e suas variáveis mágicas numeradas, como $1:

```
if subject =~ %r!http://([a-z0-9.-]+)!
  result = $1
else
  result = ""
end
```

Alternativamente, você pode chamar o método match em um objeto Regexp:

```
matchobj = %r!http://([a-z0-9.-]+)!.match(subject)
if matchobj
  result = matchobj[1]
else
  result = ""
end
```

Discussão

As receitas 2.10 e 2.21 explicam como você pode usar retrorreferências numeradas, na expressão regular e no texto de substituição, para corresponder ao mesmo texto novamente, ou para inserir parte da correspondência da expressão regular no texto de substituição. Você pode usar, em seu código, os mesmos números de referência para recuperar o texto correspondido por um ou mais grupos de captura.

Em expressões regulares, grupos de captura são numerados a partir de um. As linguagens de programação, tipicamente, iniciam a numeração de arrays e listas em zero. Todas as linguagens de programação discutidas neste livro, que armazenam grupos de captura em um array ou lista, utilizam a mesma numeração para grupos de captura em expressões regulares, iniciando em um. O elemento zero, do array ou lista, é usado para armazenar a correspondência total da expressão regular. Isso significa que, se sua expressão regular tiver três grupos de captura, o array que armazena suas correspondências terá quatro elementos. O

elemento zero contém a correspondência total, e os elementos um, dois e três armazenam o texto correspondido pelos três grupos de captura.

.NET

Para recuperar detalhes sobre os grupos de captura, novamente recorremos à função-membro Regex.Match(), explicada inicialmente na receita 3.7. O objeto Match retornado possui uma propriedade chamada Groups. Ela é uma propriedade de coleção do tipo GroupCollection. A coleção contém os detalhes de todos os grupos de captura em sua expressão regular. Groups[1] contém os detalhes do primeiro grupo de captura, Groups[2], do segundo grupo, e assim por diante.

A coleção Groups contém um objeto Group para cada grupo de captura. A classe Group possui as mesmas propriedades da classe Match, exceto pelas propriedades de Groups. Match.Groups[1].Value retorna o texto correspondido pelo primeiro grupo de captura, da mesma forma que Match.Value retorna a correspondência da expressão regular global. Match.Groups[1].Index e Match.Groups[1].Length retornam a posição inicial e o comprimento do texto correspondido pelo grupo. Veja a receita 3.8, para obter mais detalhes sobre Index e Length.

Groups[0] contém os detalhes da correspondência da expressão regular global, que também ficam armazenadas no objeto de correspondência. Match.Value e Match.Groups[0].Value são equivalentes.

A coleção Groups não lançará uma exceção, se você passar um número de grupo inválido. Por exemplo, Groups[-1] ainda retorna um objeto Group, mas as propriedades desse objeto Group indicam que o grupo de captura fictício, -1, falhou ao corresponder. A melhor maneira de testar isso é utilizando a propriedade Success. Groups[-1].Success retornará false.

Para determinar quantos grupos de captura existem, verifique Match.Groups.Count. A propriedade Count segue a mesma convenção

da propriedade Count dos objetos de coleção no .NET: ela retorna o número de elementos na coleção, que é o índice mais alto permitido, mais um. No nosso exemplo, a coleção Groups possui Groups[0] e Groups[1]. Groups.Count, então, retorna 2.

Java

O código que obtém o texto correspondido por um grupo de captura ou os detalhes da correspondência de um grupo de captura, é praticamente o mesmo que foi usado para obter a correspondência total da expressão regular, tal como mostrado nas duas últimas receitas. group(), start() e end(), métodos da classe Matcher, aceitam um parâmetro opcional. Sem esse parâmetro, ou com este parâmetro definido como zero, você obtém a correspondência ou as posições da correspondência total da expressão regular.

Se você passar um número positivo, obterá os detalhes daquele grupo de captura. Os grupos são numerados a partir de um, assim como as retrorreferências na própria expressão regular. Se você especificar um número maior do que o número de grupos de captura em sua expressão regular, essas três funções lançarão uma IndexOutOfBoundsException. Se o grupo de captura n existe, mas não participou da correspondência, group(n) retornará null, enquanto start(n) e end(n) retornarão -1.

JavaScript

Como explicado na receita anterior, o método exec() de um objeto de expressão regular retorna um array com detalhes sobre a correspondência. O elemento zero do array contém a correspondência global da expressão regular. O elemento um contém o texto correspondido pelo primeiro grupo de captura, o elemento dois armazena a correspondência do segundo grupo etc.

Se a expressão regular não corresponder à string, regexp.exec() retornará null.

PHP

A receita 3.7 explica como você pode obter o texto correspondido pela expressão regular, passando um terceiro parâmetro para preg_match(). Quando preg_match() retorna 1, o parâmetro é preenchido com um array. O elemento zero do array contém uma string, com a correspondência total da expressão regular.

O elemento um contém o texto correspondido pelo primeiro grupo de captura, o elemento dois contém o texto do segundo grupo e assim por diante. O comprimento do array será o número de grupos de captura mais um. Os índices do array correspondem aos números da retrorreferência na expressão regular.

Se você especificar a constante PREG_OFFSET_CAPTURE como o quarto parâmetro, tal como explicado na receita anterior, o comprimento do array ainda será o número de grupos de captura mais um. Mas, em vez de armazenar uma string em cada índice, o array armazenará subarrays com dois elementos cada. O subelemento zero é a string com o texto correspondido pela expressão regular global ou pelo grupo de captura. O subelemento um é um valor inteiro, e indica a posição na string de assunto em que começa o texto correspondido.

Perl

Quando o operador de correspondência a padrões m// encontra uma correspondência, ele define diversas variáveis especiais. Estas incluem as variáveis numeradas $1, $2, $3 etc., que armazenam a parte da string correspondida pelos grupos de captura na expressão regular.

Python

A solução para este problema é quase idêntica à da receita 3.7. Em vez de chamar group(), sem qualquer parâmetro, podemos especificar o número do grupo de captura em que estamos interessados. Chame group(1) para obter o texto correspondido pelo primeiro grupo de captura, group(2) para o segundo grupo e assim por diante. O Python suporta até 99 grupos de captura. O grupo de

número 0 é a correspondência total da expressão regular. Se você passar um número maior que o de grupos de captura em sua expressão regular, group() lançará uma exceção IndexError. Se o número do grupo for válido, mas o grupo, em si, não participou da correspondência da expressão regular, group() retornará None.

Você pode passar vários números de grupo para group(), em uma mesma chamada, para obter o texto correspondido por diversos grupos de captura. O resultado será uma lista de strings.

Se você deseja recuperar uma tupla com o texto correspondido por todos os grupos de captura, poderá chamar o método groups() de MatchObject. A tupla armazenará None, no caso de grupos que não participaram da correspondência. Se você passar um parâmetro para groups(), o valor desse parâmetro será usado no lugar de None para os grupos que não participaram da correspondência.

Se quiser um dicionário, em vez de uma tupla com o texto correspondido pelos grupos de captura, chame groupdict(), ao invés de groups(). Você pode passar um parâmetro a groupdict() para colocar algo diferente de None no dicionário, no caso de grupos que não participaram da correspondência.

Ruby

A receita 3.8 explica a variável $~ e o objeto MatchData. Em um contexto de array, esse objeto é avaliado como um array contendo o texto correspondido a todos os grupos de captura em sua expressão regular. Grupos de captura são numerados a partir de 1, tal como as retrorreferências na expressão regular. O elemento 0 do array contém a correspondência total da expressão regular.

$1, $2 etc. são variáveis especiais apenas para leitura. $1 é um atalho para $~[1], que contém o texto correspondido pelo primeiro grupo de captura. $2 recupera o segundo grupo e assim por diante.

Captura nomeada

Se sua expressão regular utiliza grupos de captura nomeados, você

pode usar o nome do grupo para recuperar a correspondência dele em seu código.

C#

No caso de correspondências rápidas, você pode usar a chamada estática:

```
string resultString = Regex.Match(subjectString,
                      "http://(?<domain>[a-z0-9.-]+)").Groups["domain"].Value;
```

Para usar a mesma expressão regular repetidamente, construa um objeto Regex:

```
Regex regexObj = new Regex("http://(?<domain>[a-z0-9.-]+)");
string resultString = regexObj.Match(subjectString).Groups["domain"].Value;
```

Em C#, não há diferença real no código para se obter o objeto Group de um grupo nomeado, comparado a um grupo numerado. Ao invés de indexar a coleção Groups com um número inteiro, indexe-a com uma string. Também nesse caso, o .NET não lançará uma exceção se o grupo não existir. Match.Groups["nosuchgroup"].Success simplesmente retornará false.

VB.NET

No caso de correspondências rápidas, você pode usar a chamada estática:

```
Dim ResultString = Regex.Match(SubjectString,
                   "http://(?<domain>[a-z0-9.-]+)").Groups("domain").Value
```

Para usar a mesma expressão regular repetidamente, construa um objeto Regex:

```
Dim RegexObj As New Regex("http://(?<domain>[a-z0-9.-]+)")
Dim ResultString = RegexObj.Match(SubjectString).Groups("domain").Value
```

Em VB.NET, não há diferença real no código para obter o objeto Group de um grupo nomeado, comparado a um grupo numerado. Em vez de indexar a coleção Groups com um número inteiro, indexe-a com uma string. Também nesse caso, o .NET não lançará uma exceção se o grupo não existir. Match.Groups["nosuchgroup"].Success simplesmente retorna false.

PHP

```
if (preg_match('%http://(?P<domain>[a-z0-9.-]+)%', $subject, $groups)) {
  $result = $groups['domain'];
} else {
  $result = '';
}
```

Se sua expressão regular nomeou grupos de captura, o array atribuído a $groups será um array associativo. O texto correspondido por cada grupo de captura nomeado será adicionado ao array duas vezes. Você pode recuperar o texto correspondido indexando o array pelo número do grupo ou pelo nome do grupo. Na amostra de código, $groups[0] armazena a correspondência global da expressão regular, enquanto tanto $groups[1], quanto $groups['domain'] armazenam o texto correspondido pelo único grupo de captura da expressão regular.

Perl

```
if ($subject =~ '!http://(?<domain>[a-z0-9.-]+)%!) {
  $result = $+{'domain'};
} else {
  $result = '';
}
```

Perl suporta grupos de captura nomeados a partir da versão 5.10. A variável hash $+ armazena o texto correspondido por todos os grupos de captura nomeados. O Perl numera grupos nomeados junto com os grupos numerados. Neste exemplo, tanto $1 quanto $+{name} armazenam o texto correspondido pelo único grupo de captura da expressão regular.

Python

```
matchobj = re.search("http://(?P<domain>[a-z0-9.-]+)", subject)
if matchobj:
  result = matchobj.group("domain")
else:
  result = ""
```

Se a sua expressão regular possui grupos de captura nomeados,

você pode passar o nome do grupo, em vez de seu número, para o método group().

Veja também:

A receita 2.9 explica os grupos de captura numerados.

A receita 2.11 explica os grupos de captura nomeados.

3.10 Recuperar uma lista de todas as correspondências

Problema

Todas as receitas anteriores, neste capítulo, lidaram apenas com a primeira correspondência que uma expressão regular pode encontrar na string de assunto. Porém, em muitos casos, uma expressão regular que corresponda parcialmente a uma string pode encontrar outra correspondência no restante da string. E pode haver uma terceira correspondência após a segunda, e assim por diante. Por exemplo, a expressão regular ‹\d+› pode encontrar seis correspondências na string de assunto The lucky numbers are 7, 13, 16, 42, 65, and 99: 7, 13, 16, 42, 65, e 99.

Você deseja recuperar a lista de todas as substrings que a expressão regular encontra, quando aplicada repetidamente no restante da string após cada correspondência.

Solução

C#

Você pode usar a chamada estática, caso queira processar apenas um número pequeno de strings com a mesma expressão regular:

```
MatchCollection matchlist = Regex.Matches(subjectString, @"\d+");
```

Construa um objeto Regex se quiser usar a mesma expressão regular com um grande número de strings:

```
Regex regexObj = new Regex(@"\d+");
```

```
MatchCollection matchlist = regexObj.Matches(subjectString);
```

VB.NET

Você pode usar a chamada estática, caso queira processar apenas um pequeno número de strings com a mesma expressão regular:

```
Dim matchlist = Regex.Matches(SubjectString, "\d+")
```

Construa um objeto `Regex`, se quiser usar a mesma expressão regular com um grande número de strings:

```
Dim RegexObj As New Regex("\d+")
Dim MatchList = RegexObj.Matches(SubjectString)
```

Java

```
List<String> resultList = new ArrayList<String>();
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
while (regexMatcher.find()) {
  resultList.add(regexMatcher.group());
}
```

JavaScript

```
var list = subject.match(/\d+/g);
```

PHP

```
preg_match_all('/\d+/', $subject, $result, PREG_PATTERN_ORDER);
$result = $result[0];
```

Perl

```
@result = $subject =~ m/\d+/g;
```

Este procedimento só funciona com expressões regulares que não possuam grupos de captura; então, utilize grupos de não-captura. Veja a receita 2.9 para mais detalhes.

Python

Se você pretende processar apenas um número pequeno de strings com a mesma expressão regular, pode usar a função global:

```
result = re.findall(r"\d+", subject)
```

Para usar a mesma expressão regular repetidamente, use um objeto compilado:

```
reobj = re.compile(r"\d+")
result = reobj.findall(subject)
```

Ruby

```
result = subject.scan(/\d+/)
```

Discussão

.NET

O método Matches() da classe Regex aplica várias vezes a expressão regular na string, até que todas as correspondências sejam encontradas. Ele retorna um objeto MatchCollection, que armazena todas as correspondências. A string de assunto é sempre o primeiro parâmetro. Esta é a string em que a expressão regular tentará encontrar uma correspondência. O primeiro parâmetro não pode ser null. Caso contrário, Matches() lançará uma ArgumentNullException.

Se você deseja obter as correspondências da expressão regular em apenas um pequeno número de strings, poderá usar a sobrecarga estática de Matches(). Passe sua string de assunto como primeiro parâmetro e sua expressão regular como segundo parâmetro. Você pode passar opções de expressão regular como um terceiro parâmetro opcional.

Se for processar muitas strings, construa primeiramente um objeto Regex e use-o para chamar Matches(). A string de assunto é o único parâmetro obrigatório. Você pode especificar um segundo parâmetro opcional, para indicar o índice do caractere em que a expressão regular deverá começar a busca. Essencialmente, o número que você passar como segundo parâmetro será o número de caracteres no início da string de assunto que a expressão regular deverá ignorar. Isso pode ser útil quando você já processou a string até um certo ponto e deseja verificar se o restante deverá ser processado posteriormente. Se você especificar o número, ele deve estar entre

zero e o comprimento da string de assunto. Caso contrário, IsMatch() lança uma ArgumentOutOfRangeException.

As sobrecargas estáticas não permitem o uso do parâmetro que especifica onde a tentativa da expressão regular deverá começar na string. Não há sobrecarga que permita dizer a Matches() para parar antes do final da string. Se quiser fazer isso, poderá chamar Regex.Match("subject", start, stop) em um loop, como mostrado na próxima receita, e adicionar todas as correspondências que ele encontra em uma lista definida por você.

Java

O Java não fornece uma função que recupere a lista de correspondências. Você pode fazer isso facilmente em seu próprio código, adaptando a receita 3.7. Em vez de chamar find() em uma instrução if, faça-a em um loop while.

Para usar as classes List e ArrayList, como no exemplo, insira import java.util.*; no início de seu código.

JavaScript

O código de exemplo chama string.match(), assim como a solução JavaScript da receita 3.7. Há uma pequena, mas muito importante, diferença: a sinalização /g. Sinalizações de expressão regular são explicadas na receita 3.4.

A sinalização /g diz à função match() para iterar todas as correspondências na string e colocá-las em um array. Na amostra de código, list[0] armazenará a primeira correspondência da expressão regular, list[1] a segunda e assim por diante. Verifique list.length para determinar o número de correspondências. Se nenhuma correspondência puder ser encontrada, string.match retornará null, como de costume.

Os elementos do array são strings. Ao utilizar uma expressão regular com a sinalização /g, string.match() não fornecerá mais detalhes sobre a correspondência da expressão regular. Se quiser

obter detalhes de correspondência para todas as correspondências da expressão regular, itere-as, como explicado na receita 3.11.

PHP

Todas as receitas PHP anteriores usavam preg_match(), que encontra a primeira correspondência da expressão regular em uma string. preg_match_all() é muito semelhante. A diferença fundamental é que ela encontrará todas as correspondências na string. Ela retorna um número inteiro, indicando o número de vezes nas quais a expressão regular poderia corresponder.

Os três primeiros parâmetros de preg_match_all() são os mesmos de preg_match(): uma string com sua expressão regular, a string em que você deseja fazer pesquisa e uma variável que receberá um array com os resultados. As únicas diferenças são que o terceiro parâmetro é obrigatório, e o array será sempre multidimensional.

Como quarto parâmetro, especifique a constante PREG_PATTERN_ORDER ou PREG_SET_ORDER. Se você omitir o quarto parâmetro, PREG_PATTERN_ORDER será o padrão.

Se usar PREG_PATTERN_ORDER, você obterá um array que armazena os detalhes da correspondência global no elemento zero, e os detalhes dos grupos de captura um e além, nos elementos um e além. O comprimento do array será o número de grupos de captura mais um. É a mesma ordem utilizada por preg_match(). A diferença está em que, ao invés de cada elemento armazenar uma string com a única correspondência encontrada por preg_match(), cada elemento encontrado por preg_matches() armazenará um subarray com todas as correspondências. O comprimento de cada subarray é o mesmo valor devolvido por preg_matches().

Para obter uma lista de todas as correspondências da expressão regular na string, descartando o texto correspondido por grupos de captura, especifique PREG_PATTERN_ORDER e recupere o elemento zero no array. Se você estiver interessado apenas no texto correspondido por um grupo de captura em particular, use

PREG_PATTERN_ORDER e o número do grupo de captura. Por exemplo, especificar $result[1] após chamar preg_match('%http://([a-z0-9.-]+)%', $subject, $result) lhe dará a lista de nomes de domínio de todas os URLs em sua string de assunto.

PREG_SET_ORDER preenche o array com as mesmas strings, mas de uma maneira diferente. O comprimento do array é o valor retornado por preg_matches(). Cada elemento do array é um subarray, com a correspondência global da expressão regular armazenada no subelemento zero, e os grupos de captura nos elementos um e além. Se você especificar PREG_SET_ORDER, $result[0] armazenará o mesmo array, como se tivesse chamado preg_match().

Você pode combinar PREG_OFFSET_CAPTURE com PREG_PATTERN_ORDER ou PREG_SET_ORDER. Se o fizer, terá o mesmo efeito que passar PREG_OFFSET_CAPTURE como quarto parâmetro de preg_match(). Ao invés de cada elemento no array armazenar uma string, ele armazená um array de dois elementos, contendo a string, e o deslocamento em que ocorre essa string, dentro da string de assunto original.

Perl

A receita 3.4 explica que você precisa adicionar o modificador /g para permitir que sua expressão regular encontre mais de uma correspondência na string de assunto. Se você utilizar uma expressão regular global em um contexto de lista, ela encontrará, e retornará, todas as correspondências. Nesta receita, a variável de lista à esquerda do operador de atribuição fornece o contexto de lista.

Se a expressão regular não possuir grupos de captura, a lista conterá a correspondência da expressão regular global. Se a expressão regular possuir grupos de captura, a lista conterá o texto correspondido por todos os grupos de captura a cada correspondência da expressão regular. A correspondência geral da expressão regular não será incluída, a menos que você coloque um

grupo de captura englobando toda a expressão regular. Se você deseja apenas obter uma lista de correspondências globais da expressão regular, substitua todos os grupos de captura por grupos de não-captura. A receita 2.9 explica os dois tipos de agrupamentos.

Python

A função findall() do módulo re pesquisa repetidamente, ao longo de uma string, para encontrar todas as correspondências da expressão regular. Passe a sua expressão regular como primeiro parâmetro e a string de assunto como segundo. Você pode passar as opções de expressão regular no terceiro parâmetro opcional.

A função re.findall() chama re.compile() e, em seguida, chama o método findall() no objeto de expressão regular compilado. Esse método possui apenas um parâmetro obrigatório: a string de assunto.

O método findall() aceita dois parâmetros opcionais que a função global re.findall() não suporta. Após a string de assunto, você pode passar a posição do caractere na string em que findall() deverá começar sua busca. Se omitir este parâmetro, findall() processa toda a string de assunto. Se especificar uma posição inicial, também poderá especificar uma posição final. Se não especificar uma posição final, a pesquisa terminará no final da string.

Independente de como você chama findall(), o resultado será sempre uma lista com todas as correspondências que poderiam ser encontradas. Se a expressão regular não possuir grupos de captura, você receberá uma lista de strings. Se ela tiver grupos de captura, você receberá uma lista de tuplas, com o texto correspondido por todos os grupos de captura a cada correspondência da expressão regular.

Ruby

O método scan() da classe String aceita uma expressão regular como único parâmetro. Ele itera todas as correspondências da expressão regular na string. Quando chamado sem um bloco, scan() retorna um

array de todas as correspondências da expressão regular.

Se a sua expressão regular não tiver grupos de captura, scan() retornará um array de strings. O array possuirá um elemento para cada correspondência da expressão regular, contendo o texto correspondido.

Quando existirem grupos de captura, scan() retornará um array de arrays. O array possuirá um elemento para cada correspondência da expressão regular. Cada elemento será um array com o texto correspondido, por cada um dos grupos de captura. O subelemento zero conterá o texto correspondido pelo primeiro grupo de captura, o subelemento um conterá o segundo grupo de captura etc. A correspondência total da expressão regular não estará incluída no array. Se você quiser que a correspondência global seja incluída, coloque toda sua expressão regular dentro de um grupo de captura extra.

O Ruby não fornece opção para fazer com que scan() retorne um array de strings quando a expressão regular possuir grupos de captura. Sua única solução será substituir todos os grupos de captura nomeados e numerados por grupos de não-captura.

Veja também:

Receitas 3.7, 3.11 e 3.12.

3.11 Iterar todas as correspondências

Problema

A receita anterior mostra como uma expressão regular poderia ser aplicada repetidamente em uma string para obter uma lista de correspondências. Agora, você deseja iterar todas as correspondências em seu próprio código.

Solução

C#

Você pode usar a chamada estática, quando for processar somente uma pequena quantidade de strings com a mesma expressão regular:

```
Match matchResult = Regex.Match(subjectString, @"\d+");
while (matchResult.Success) {
  // Aqui você pode processar a correspondência armazenada em matchResult
  matchResult = matchResult.NextMatch();
}
```

Contrua um objeto Regex, se quiser usar a mesma expressão regular em uma grande quantidade de strings:

```
Regex regexObj = new Regex(@"\d+");
matchResult = regexObj.Match(subjectString);
while (matchResult.Success) {
  // Aqui você pode processar a correspondência armazenada em matchResult
  matchResult = matchResult.NextMatch();
}
```

VB.NET

Você pode usar a chamada estática, quando for processar somente uma pequena quantidade de strings com a mesma expressão regular:

```
Dim MatchResult = Regex.Match(SubjectString, "\d+")
While MatchResult.Success
  'Aqui você pode processar a correspondência armazenada em matchResult
  MatchResult = MatchResult.NextMatch
End While
```

Contrua um objeto Regex, se quiser usar a mesma expressão regular em uma grande quantidade de strings:

```
Dim RegexObj As New Regex("\d+")
Dim MatchResult = RegexObj.Match(SubjectString)
While MatchResult.Success
  'Aqui você pode processar a correspondência armazenada em matchResult
  MatchResult = MatchResult.NextMatch
End While
```

Java

```
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
while (regexMatcher.find()) {
  // Aqui você pode processar a correspondência armazenada em regexMatcher
}
```

JavaScript

Se a sua expressão regular puder gerar uma correspondência de comprimento zero, ou se você simplesmente não está certo a esse respeito, certifique-se de contornar os problemas de compatibilidade entre navegadores, tratando correspondências de comprimento zero e utilizando exec():

```
var regex = /\d+/g;
var match = null;
while (match = regex.exec(subject)) {
  // Não permita que navegadores como o Firefox entrem em loop infinito
  if (match.index == regex.lastIndex) regex.lastIndex++;
  // Aqui você pode processar a correspondência armazenada na variável match
}
```

Se você souber, com certeza, que sua expressão regular nunca encontrará uma correspondência de comprimento zero, poderá iterar a expressão regular diretamente:

```
var regex = /\d+/g;
var match = null;
while (match = regex.exec(subject)) {
  // Aqui você pode processar a correspondência armazenada na variável match
}
```

PHP

```
preg_match_all('/\d+/', $subject, $result, PREG_PATTERN_ORDER);
for ($i = 0; $i < count($result[0]); $i++) {
  # Texto correspondido = $result[0][$i];
}
```

Perl

```
while ($subject =~ m/\d+/g) {
  # Texto correspondido = $&
}
```

Python

Se você processar apenas uma pequena quantidade de strings com a mesma expressão regular, pode usar a função global:

```
for matchobj in re.finditer(r"\d+", subject):
  # você pode processar a correspondência armazenada na variável matchobj
```

Para usar a mesma expressão regular repetidamente, use um objeto compilado:

```
reobj = re.compile(r"\d+")
for matchobj in reobj.finditer(subject):
  # você pode processar a correspondência armazenada na variável matchobj
```

Ruby

```
subject.scan(/\d+/) {|match|
  # você pode processar a correspondência armazenada na variável match
}
```

Discussão

.NET

A receita 3.7 explica como usar a função-membro Match(), da classe Regex, para recuperar a primeira correspondência da expressão regular na string. Para iterar todas as correspondências na string, mais uma vez chamamos a função Match() para recuperar os detalhes da primeira correspondência. A função Match() retorna uma instância da classe Match, armazenada na variável matchResult. Se a propriedade Success, do objeto matchResult, for igual a true, podemos começar nosso loop.

No início do loop, você pode usar as propriedades da classe Match para processar os detalhes da primeira correspondência. A receita 3.7 explica a propriedade Value, a receita 3.8 explica as propriedades Index e Length, e a receita 3.9 explica a coleção Groups.

Ao terminar de tratar a primeira correspondência, chame a função membro NextMatch() da variável matchResult. Match.NextMatch() retorna uma instância da classe Match, tal como Regex.Match(). A instância

recém-retornada contém os detalhes da segunda correspondência.

Atribuir o resultado de matchResult.NextMatch() à variável matchResult facilita a iteração de todas as correspondências. Temos de verificar matchResult.Success novamente, para ver se NextMatch(), de fato, encontrou outra correspondência. Quando NextMatch() falhar, ela ainda retorna um objeto Match, mas sua propriedade Success será definida como false. Ao usar uma única variável matchResult, podemos combinar o teste inicial, que aguarda o sucesso, e o teste após a chamada a NextMatch(), em uma única instrução while.

Chamar NextMatch() não invalida o objeto Match que o chamou. Se quisesse, você poderia manter o objeto Match completo a cada correspondência da expressão regular.

O método NextMatch() não aceita qualquer parâmetro. Ele utiliza a expressão regular e a string do assunto da mesma forma que elas foram passadas ao método Regex.Match(). O objeto Match mantém as referências à expressão regular e à string de assunto.

Você pode usar a chamada estática Regex.Match(), mesmo quando sua string de assunto possuir um número muito grande de correspondências de expressão regular. Regex.Match() compilará sua expressão regular uma vez, e o objeto Match retornado armazenará uma referência à expressão regular compilada. Match.MatchAgain() utiliza a expressão regular previamente compilada e referenciada pelo objeto Match, mesmo que você tenha usado a chamada estática Regex.Match(). É preciso, apenas, instanciar a classe Regex, se quiser chamar Regex.Match() repetidamente, ou seja, caso queira usar a mesma expressão regular em várias strings.

Java

Iterar todas as correspondências de uma string, no Java, é muito fácil. Basta chamar o método find(), apresentado na receita 3.7, dentro de um loop while. Cada chamada a find() atualiza o objeto Matcher com os detalhes sobre a correspondência e a posição inicial da próxima tentativa de correspondência.

JavaScript

Antes de começar, certifique-se de especificar a sinalização /g, caso queira usar a expressão regular em um loop. Essa sinalização é explicada na receita 3.4. while (regexp.exec()) encontra todos os números na string de assunto, quando regexp = /\d+/g. Se regexp = /\d+/, então while (regexp.exec()) encontrará o primeiro número na string repetidamente, até que seu script trave, ou seja forçosamente encerrado pelo navegador.

Observe que while (/\d+/g.exec()) (repetição de uma expressão regular literal com /g) também ficará preso no mesmo loop infinito, pelo menos com certas implementações do JavaScript, pois a expressão regular é recompilada a cada iteração do loop while. Quando a expressão regular é recompilada, a posição inicial da tentativa de correspondência é ajustada para o início da string. Atribua a expressão regular a uma variável fora do loop, para ter certeza de que ela será compilada apenas uma vez.

As receitas 3.8 e 3.9 explicam o objeto retornado por regexp.exec(). Esse objeto será sempre o mesmo, independentemente de você usar exec() em um loop. Você pode fazer o que quiser com este objeto.

O único efeito de /g é que ela atualiza a propriedade lastIndex do objeto regexp no qual você está chamando exec(). Esse procedimento funciona mesmo quando você estiver usando uma expressão regular literal, como mostra a segunda solução JavaScript para esta receita. Da próxima vez em que você chamar exec(), a tentativa de correspondência começará em lastIndex. Se você atribuir um novo valor para lastIndex, a tentativa de correspondência começará na posição especificada.

Infelizmente, há um grande problema com lastIndex. Se você ler o padrão ECMA-262v3 do JavaScript ao pé da letra, exec() deveria ajustar lastIndex para o primeiro caractere depois da correspondência. Isso significa que, se a correspondência tiver comprimento zero, a próxima tentativa começará na posição da

correspondência recém-encontrada, resultando em um loop infinito.

Todos os mecanismos de expressão regular discutidos neste livro (exceto JavaScript) lidam automaticamente com isso, iniciando a próxima tentativa de correspondência, na string, um caractere adiante, caso a correspondência anterior tenha tido comprimento zero. O Internet Explorer faz isso incrementando lastIndex em um, caso a correspondência tenha comprimento zero. É por isso que a receita 3.7 alega não poder utilizar lastIndex para determinar o final da correspondência; você receberá valores incorretos no Internet Explorer.

Os desenvolvedores do Firefox, no entanto, são inflexíveis em implementar o padrão ECMA-262v3 literalmente, mesmo que isso signifique que regexp.exec() poderá ficar preso em um loop infinito. Esse resultado não é improvável. Por exemplo, você poderia usar re = /^.*$/gm; while (re.exec()) para iterar todas as linhas em uma string de múltiplas linhas, mas, se a string tiver uma linha em branco, o Firefox ficará preso nela.

A solução é incrementar lastIndex em seu próprio código, caso a função exec() ainda não tenha feito isso. A primeira solução JavaScript para esta receita mostra como devemos proceder. Se estiver inseguro, simplesmente cole essa linha de código, e pronto.

Esse problema não existe com string.match() (Receita 3.14), ou quando todas as correspondências são encontradas com string.replace() (Receita 3.10). No caso desses métodos, que usam lastIndex internamente, o padrão ECMA-262v3 diz que lastIndex deverá ser incrementado a cada correspondência de comprimento zero.

PHP

A função preg_match() aceita um quinto parâmetro opcional para indicar a posição na string em que a tentativa de correspondência deverá começar. Você poderia adaptar a receita 3.8 para passar $matchstart + $matchlength como quinto parâmetro na segunda

chamada a preg_match(), para encontrar a segunda correspondência na string, e repetir esse procedimento na terceira e próximas correspondências até preg_match() retornar 0. A receita 3.18 utiliza este método.

Além da necessidade de código extra para calcular o deslocamento inicial a cada tentativa de correspondência, chamar preg_match() repetidamente é ineficiente, porque não existe um jeito de armazenar uma expressão regular compilada em uma variável. preg_match() terá de procurar a expressão regular compilada em seu cache, sempre que você chamá-la.

Uma solução mais fácil e eficiente seria chamar preg_match_all(), como explicado na receita anterior, e iterar o array que contém os resultados da correspondência.

Perl

A receita 3.4 explica que você precisa adicionar o modificador /g para permitir que sua expressão regular encontre mais de uma correspondência na string de assunto. Se você usar uma expressão regular global em um contexto escalar, ela tentará encontrar a próxima correspondência, continuando no final da correspondência anterior. Nesta receita, a instrução while fornece o contexto escalar. Todas as variáveis especiais, como $& (explicada na receita 3.7), estão disponíveis dentro do loop while.

Python

A função finditer(), de re, retorna um iterador que você pode usar para encontrar todas as correspondências da expressão regular. Passe sua expressão regular como primeiro parâmetro e a string de assunto como segundo parâmetro. Você pode passar as opções de expressão regular no terceiro parâmetro opcional.

A função re.finditer() chama re.compile() e, em seguida, chama o método finditer() no objeto de expressão regular compilado. Este método possui apenas um parâmetro obrigatório: a string de

assunto.

O método finditer() aceita dois parâmetros opcionais que a função global re.finditer() não suporta. Após a string de assunto, você pode passar a posição do caractere na string em que finditer() deverá começar sua busca. Se você omitir esse parâmetro, o iterador processará toda a string de assunto. Se você especificar uma posição inicial, também poderá especificar uma posição final. Se não especificar uma posição final, a pesquisa executará até atingir o final da string.

Ruby

O método scan() da classe String aceita uma expressão regular como único parâmetro, iterando todas as correspondências na string. Quando chamado com um bloco, você pode processar as correspondências, conforme elas vão sendo encontradas.

Se sua expressão regular não tiver um ou mais grupos de captura, especifique uma variável de iteração no bloco. Essa variável receberá uma string com o texto correspondido pela expressão regular.

Se sua expressão regular tiver um ou mais grupos de captura, liste uma variável para cada grupo. A primeira variável receberá uma string com o texto correspondido pelo primeiro grupo de captura; a segunda variável receberá o segundo grupo de captura, e assim por diante. Nenhuma variável será preenchida com a correspondência global da expressão regular. Se você quiser que a correspondência global seja incluída, coloque toda a sua expressão regular dentro de um grupo extra de captura:

```
subject.scan(/(a)(b)(c)/) {|a, b, c|
  # a, b e c guardam o texto correspondido pelos três grupos de captura
}
```

Se listar menos variáveis do que grupos de captura em sua expressão regular, você será capaz de acessar apenas os grupos de captura para os quais forneceu as variáveis. Se listar mais variáveis do que grupos de captura, as variáveis extras serão

definidas como nil.

Se listar apenas uma variável iteradora e sua expressão regular tiver um ou mais grupos de captura, a variável será preenchida com um array de strings. O array terá uma string para cada grupo de captura. Se houver apenas um grupo de captura, o array terá um único elemento:

```
subject.scan(/(a)(b)(c)/) {|abc|
  # abc[0], abc[1] e abc[2] guardam o texto
  # correspondido pelos três grupos de captura
}
```

Veja também:

Receitas 3.7, 3.8, 3.10 e 3.12.

3.12 Validar correspondências no código procedural

Problema

A receita 3.10 mostra como você pode recuperar uma lista de todas as correspondências passíveis de serem encontradas em uma expressão regular, quando esta é aplicada repetidamente no restante da string após cada correspondência. Agora, você deseja obter uma lista das correspondências que atendam a certos critérios extras, que você não pode expressar (facilmente) em uma expressão regular. Por exemplo, ao recuperar uma lista de números da sorte, você só quer reter os que sejam um número inteiro múltiplo de 13.

Solução

C#

Você pode usar a chamada estática quando for processar apenas uma pequena quantidade de strings com a mesma expressão regular:

```
StringCollection resultList = new StringCollection();
Match matchResult = Regex.Match(subjectString, @"\d+");
while (matchResult.Success) {
  if (int.Parse(matchResult.Value) % 13 == 0) {
    resultList.Add(matchResult.Value);
  }
  matchResult = matchResult.NextMatch();
}
```

Construa um objeto Regex, se quiser usar a mesma expressão regular em uma grande quantidade de strings:

```
StringCollection resultList = new StringCollection();
Regex regexObj = new Regex(@"\d+");
matchResult = regexObj.Match(subjectString);
while (matchResult.Success) {
  if (int.Parse(matchResult.Value) % 13 == 0) {
    resultList.Add(matchResult.Value);
  }
  matchResult = matchResult.NextMatch();
}
```

VB.NET

Você pode usar a chamada estática quando for processar apenas uma pequena quantidade de strings com a mesma expressão regular:

```
Dim ResultList = New StringCollection
Dim MatchResult = Regex.Match(SubjectString, "\d+")
While MatchResult.Success
  If Integer.Parse(MatchResult.Value) Mod 13 = 0 Then
    ResultList.Add(MatchResult.Value)
  End If
    MatchResult = MatchResult.NextMatch
End While
```

Construa um objeto Regex se quiser usar a mesma expressão regular em uma grande quantidade de strings:

```
Dim ResultList = New StringCollection
Dim RegexObj As New Regex("\d+")
Dim MatchResult = RegexObj.Match(SubjectString)
While MatchResult.Success
  If Integer.Parse(MatchResult.Value) Mod 13 = 0 Then
```

```
    ResultList.Add(MatchResult.Value)
  End If
  MatchResult = MatchResult.NextMatch
End While
```

Java

```
List<String> resultList = new ArrayList<String>();
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
while (regexMatcher.find()) {
  if (Integer.parseInt(regexMatcher.group()) % 13 == 0) {
    resultList.add(regexMatcher.group());
  }
}
```

JavaScript

```
var list = [];
var regex = /\d+/g;
var match = null;
while (match = regex.exec(subject)) {
  // Não permita que navegadores como o Firefox entrem em loop infinito
  if (match.index == regex.lastIndex) regex.lastIndex++;
  // Aqui você pode processar a correspondência armazenada na variável match
  if (match[0] % 13 == 0) {
    list.push(match[0]);
  }
}
```

PHP

```
preg_match_all('/\d+/', $subject, $matchdata, PREG_PATTERN_ORDER);
for ($i = 0; $i < count($matchdata[0]); $i++) {
  if ($matchdata[0][$i] % 13 == 0) {
    $list[] = $matchdata[0][$i];
  }
}
```

Perl

```
while ($subject =~ m/\d+/g) {
  if ($& % 13 == 0) {
    push(@list, $&);
  }
```

```
}
```

Python

Se for processar somente uma pequena quantidade de strings com a mesma expressão regular, você pode usar a função global:

```
list = []
for matchobj in re.finditer(r"\d+", subject):
  if int(matchobj.group()) % 13 == 0:
    list.append(matchobj.group())
```

Para usar a mesma expressão regular repetidamente, use um objeto compilado:

```
list = []
reobj = re.compile(r"\d+")
for matchobj in reobj.finditer(subject):
  if int(matchobj.group()) % 13 == 0:
    list.append(matchobj.group())
```

Ruby

```
list = []
subject.scan(/\d+/) {|match|
  list << match if (Integer(match) % 13 == 0)
}
```

Discussão

As expressões regulares lidam com texto. Embora a expressão regular ‹\d+› corresponda a algo que chamamos de um número, para o mecanismo de expressões regulares trata-se apenas uma string de um ou mais dígitos.

Se quiser encontrar números específicos, tais como os divisíveis por 13, é muito mais fácil escrever uma expressão regular geral que corresponda a todos os números e, em seguida, utilizar um pedaço do código procedural para ignorar as correspondências da expressão regular nas quais não esteja interessado.

As soluções para esta receita são todas baseadas em soluções da receita anterior, que mostra como iterar todas as correspondências. Dentro do loop, podemos converter a correspondência da expressão

regular em um número.

Algumas linguagens fazem isso automaticamente; outras requerem uma chamada de função explícita, para converter a string em um inteiro. Em seguida, verificamos se o número inteiro é divisível por 13. Se for, a correspondência da expressão regular é adicionada à lista. Se não for, a correspondência da expressão regular é ignorada.

Veja também:

Receitas 3.7, 3.10 e 3.11.

3.13 Encontrar uma correspondência dentro de outra

Problema

Você deseja encontrar todas as correspondências de uma determinada expressão regular, mas apenas dentro de determinadas seções da string de assunto. Uma outra expressão regular corresponde a cada uma das seções na string.

Suponha que você tenha um arquivo HTML, no qual diversas passagens sejam marcadas com negrito usando tags . Você deseja encontrar todos os números marcados com negrito. Se algum texto em negrito tiver múltiplos números, você vai querer correspondê-los separadamente. Por exemplo, ao processar a string 1 2 3 4 5 6 7, você deseja encontrar quatro correspondências: 2, 5, 6 e 7.

Solução

C#

```
StringCollection resultList = new StringCollection();
Regex outerRegex = new Regex("<b>(.*?)</b>", RegexOptions.Singleline);
Regex innerRegex = new Regex(@"\d+");
// Encontra a primeira seção
```

```
Match outerMatch = outerRegex.Match(subjectString);
while (outerMatch.Success) {
  // Obtém as correspondências dentro da seção
  Match innerMatch = innerRegex.Match(outerMatch.Groups[1].Value);
  while (innerMatch.Success) {
    resultList.Add(innerMatch.Value);
    innerMatch = innerMatch.NextMatch();
  }
  // Encontra a próxima seção
  outerMatch = outerMatch.NextMatch();
}
```

VB.NET

```
Dim ResultList = New StringCollection
Dim OuterRegex As New Regex("<b>(.*?)</b>", RegexOptions.Singleline)
Dim InnerRegex As New Regex("\d+")
'Encontra a primeira seção
Dim OuterMatch = OuterRegex.Match(SubjectString)
While OuterMatch.Success
  'Obtém as correspondências dentro da seção
  Dim InnerMatch = InnerRegex.Match(OuterMatch.Groups(1).Value)
  While InnerMatch.Success
    ResultList.Add(InnerMatch.Value)
    InnerMatch = InnerMatch.NextMatch
  End While
  'Encontra a próxima seção
  OuterMatch = OuterMatch.NextMatch
End While
```

Java

Iterar usando dois matchers é fácil, e funciona com Java 4 e versões posteriores:

```
List<String> resultList = new ArrayList<String>();
Pattern outerRegex = Pattern.compile("<b>(.*?)</b>", Pattern.DOTALL);
Pattern innerRegex = Pattern.compile("\\d+");
Matcher outerMatcher = outerRegex.matcher(subjectString);
while (outerMatcher.find()) {
  Matcher innerMatcher = innerRegex.matcher(outerMatcher.group());
  while (innerMatcher.find()) {
    resultList.add(innerMatcher.group());
  }
```

```
}
```

O código a seguir é mais eficiente (porque innerMatcher é criado apenas uma vez), mas requer Java 5 ou posterior:

```
List<String> resultList = new ArrayList<String>();
Pattern outerRegex = Pattern.compile("<b>(.*?)</b>", Pattern.DOTALL);
Pattern innerRegex = Pattern.compile("\\d+");
Matcher outerMatcher = outerRegex.matcher(subjectString);
Matcher innerMatcher = innerRegex.matcher(subjectString);
while (outerMatcher.find()) {
 innerMatcher.region(outerMatcher.start(), outerMatcher.end());
 while (innerMatcher.find()) {
  resultList.add(innerMatcher.group());
 }
}
```

JavaScript

```
var result = [];
var outerRegex = /<b>([\s\S]*?)<\/b>/g;
var innerRegex = /\d+/g;
var outerMatch = null;
while (outerMatch = outerRegex.exec(subject)) {
 if (outerMatch.index == outerRegex.lastIndex)
  outerRegex.lastIndex++;
 var innerSubject = subject.substr(outerMatch.index, outerMatch[0].length);
 var innerMatch = null;
 while (innerMatch = innerRegex.exec(innerSubject)) {
  if (innerMatch.index == innerRegex.lastIndex)
     innerRegex.lastIndex++;
  result.push(innerMatch[0]);
 }
}
```

PHP

```
$list = array();
preg_match_all('%<b>(.*?)</b>%s', $subject, $outermatches,
PREG_PATTERN_ORDER);
for ($i = 0; $i < count($outermatches[0]); $i++) {
 if (preg_match_all('/\d+/', $outermatches[0][$i], $innermatches,
PREG_PATTERN_ORDER)) {
  $list = array_merge($list, $innermatches[0]);
 }
```

```
}
```

Perl

```
while ($subject =~ m!<b>(.*?)</b>!gs) {
  push(@list, ($& =~ m/\d+/g));
}
```

Isso só funciona se a expressão regular interna (‹\d+›, neste exemplo) não possuir grupos de captura; então, use grupos de não-captura. Veja a receita 2.9 para mais detalhes.

Python

```
list = []
innerre = re.compile(r"\d+")
for outermatch in re.finditer("(?s)<b>(.*?)</b>", subject):
  list.extend(innerre.findall(outermatch.group(1)))
```

Ruby

```
list = []
subject.scan(/<b>(.*?)<\/b>/m) {|outergroups|
  list += outergroups[0].scan(/\d+/)
}
```

Discussão

As expressões regulares são bem-equipadas para dividir a entrada de dados em tokens, mas elas não o são para analisar a entrada. Dividir em tokens significa identificar as diferentes partes de uma string, tais como números, palavras, símbolos, tags, comentários etc. Trata-se de examinar o texto, da esquerda para a direita, buscando diferentes alternativas e quantidades de caracteres a serem correspondidos. As expressões regulares lidam muito bem com isso.

Analisar significa processar a relação entre esses tokens. Por exemplo, em uma linguagem de programação, combinações desses tokens formam instruções, funções, classes, espaços de nomes etc. É melhor deixar o acompanhamento do significado dos tokens, dentro do contexto maior da entrada de dados, para o código

procedural. Em particular, as expressões regulares não podem acompanhar contextos não-lineares, tais como construções aninhadas[1].

Tentar encontrar um tipo de token dentro de outro tipo de token é uma tarefa que as pessoas, normalmente, tentam resolver com expressões regulares. Um par de tags de negrito, em HTML, é facilmente correspondido com a expressão ‹(.*?)›[2]. Um número é ainda mais facilmente correspondido com a expressão regular ‹\d+›. Mas, se você tentar combinar estas expressões regulares em uma única expressão regular, vai acabar com algo um pouco diferente:

```
\d+(?=(?:(?!<b>).)*</b>)
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Embora a expressão regular apresentada seja uma solução para o problema colocado por esta receita, ela é pouco intuitiva. Mesmo um especialista em expressão regular terá de examiná-la cuidadosamente para determinar o que ela faz, ou talvez recorrer a uma ferramenta para destacar as correspondências. E trata-se, apenas, da combinaçao de duas expressões regulares simples.

Uma solução melhor seria manter as duas expressões regulares como estão e usar o código procedural para combiná-las. O código resultante, embora um pouco mais longo, é muito mais fácil de compreender e manter; criar um código simples é a razão primeira para usar expressões regulares. Uma expressão regular, como ‹(.*?)›, é fácilmente entendida por qualquer pessoa com um mínimo de experiência, e rapidamente faz o que provavelmente levaria muitas linhas de código a mais, algo mais difícil de manter.

Embora as soluções para esta receita estejam entre as mais complexas deste capítulo, elas são muito simples. Duas expressões regulares são usadas. A expressão regular “externa” corresponde às tags de negrito HTML e ao texto entre elas; o texto entre as tags é capturado pelo primeiro grupo de captura. Essa expressão regular é

implementada com o mesmo código mostrado na receita 3.11. A única diferença é que o comentário, dizendo onde usar a correspondência, foi substituído pelo código que permite à expressão regular “interna” fazer seu trabalho.

A segunda expressão regular corresponde a um dígito. Essa expressão regular é aplicada com o mesmo código mostrado na receita 3.10. A única diferença é que, ao invés de processar totalmente a string de assunto, a segunda expressão regular é aplicada apenas à parte da string de assunto correspondida pelo primeiro grupo de captura da expressão regular externa.

Há duas maneiras de restringir uma expressão regular interna ao texto correspondido por (um grupo de captura de) uma expressão regular externa. Algumas linguagens fornecem uma função que permite que a expressão regular seja aplicada em parte de uma string. Isso pode economizar uma cópia extra de string, caso a função de correspondência não preencha automaticamente uma estrutura com o texto correspondido pelos grupos de captura. Sempre será possível apenas recuperar a substring correspondida pelo grupo de captura, aplicando-lhe a expressão regular interna.

De qualquer forma, utilizar duas expressões regulares juntas, em um loop, será mais rápido do que usar uma expressão regular com seus grupos de lookahead aninhados. Esta última exige que o mecanismo de expressão regular faça muitos retrocessos. Em grandes arquivos, será muito mais lento usar apenas uma expressão regular, uma vez que ela precisaria determinar as extremidades da seção (tags de negrito HTML) para cada número na string de assunto, incluindo os números que não estejam entre as tags ‹b›. A solução que utiliza duas expressões regulares nem sequer começa a procurar por números, até que tenha encontrado as extremidades da seção, o que ela faz em tempo linear.

Veja também:

Receitas 3.8, 3.10 e 3.11.

3.14 Substituir todas as correspondências

Problema

Você deseja substituir todas as correspondências da expressão regular ‹before› pelo texto de substituição «after».

Solução

C#

Você pode usar a chamada estática quando for processar apenas uma pequena quantidade de strings com a mesma expressão regular:

```
string resultString = Regex.Replace(subjectString, "before", "after");
```

Se a expressão regular for fornecida pelo usuário final, você deve usar a chamada estática com um tratamento de exceção completo:

```
string resultString = null;
try {
  resultString = Regex.Replace(subjectString, "before", "after");
} catch (ArgumentNullException ex) {
  // Não pode passar null como sendo a expressão regular, a string de assunto
  // ou o texto de substituição
} catch (ArgumentException ex) {
  // Erro de sintaxe na expressão regular
}
```

Construa um objeto Regex, se quiser usar a mesma expressão regular com uma grande quantidade de strings:

```
Regex regexObj = new Regex("before");
string resultString = regexObj.Replace(subjectString, "after");
```

Se a expressão regular for fornecida pelo usuário final, você deve usar o objeto Regex com um tratamento de exceção completo:

```
string resultString = null;
try {
  Regex regexObj = new Regex("before");
  try {
```

```
    resultString = regexObj.Replace(subjectString, "after");
  } catch (ArgumentNullException ex) {
    // Não pode passar null como sendo a string de assunto ou o texto de
substituição
  }
} catch (ArgumentException ex) {
  // Erro de sintaxe na expressão regular
}
```

VB.NET

Você pode usar a chamada estática quando for processar apenas uma pequena quantidade de strings com a mesma expressão regular:

```
Dim ResultString = Regex.Replace(SubjectString, "before", "after")
```

Se a expressão regular for fornecida pelo usuário final, você deve usar a chamada estática com um tratamento de exceção completo:

```
Dim ResultString As String = Nothing
Try
  ResultString = Regex.Replace(SubjectString, "before", "after")
Catch ex As ArgumentNullException
  'Não pode passar null como sendo a expressão regular, a string de assunto
  'ou o texto de substituição
Catch ex As ArgumentException
  'Erro de sintaxe na expressão regular
End Try
```

Construa um objeto Regex, se quiser usar a mesma expressão regular com uma grande quantidade de strings:

```
Dim RegexObj As New Regex("before")
Dim ResultString = RegexObj.Replace(SubjectString, "after")
```

Se a expressão regular for fornecida pelo usuário final, você deve usar o objeto Regex com um tratamento de exceção completo:

```
Dim ResultString As String = Nothing
Try
  Dim RegexObj As New Regex("before")
  Try
    ResultString = RegexObj.Replace(SubjectString, "after")
  Catch ex As ArgumentNullException
    'Não pode passar null como sendo a string de assunto ou o texto de
```

```
substituição
  End Try
Catch ex As ArgumentException
  'Erro de sintaxe na expressão regular
End Try
```

Java

Você pode usar a chamada estática, quando for processar apenas uma string com a mesma expressão regular:

```
String resultString = subjectString.replaceAll("before", "after");
```

Se a expressão regular, ou o texto de substituição, for fornecido pelo usuário final, você deve usar a chamada estática com um tratamento de exceção completo:

```
try {
  String resultString = subjectString.replaceAll("before", "after");
} catch (PatternSyntaxException ex) {
// Erro de sintaxe na expressão regular
} catch (IllegalArgumentException ex) {
  // Erro de sintaxe no texto de substituição (sinais $ não-escapados?)
} catch (IndexOutOfBoundsException ex) {
  // Retrorreferência inexistente utilizou o texto de substituição
}
```

Construa um objeto Matcher, se quiser usar a mesma expressão regular com uma grande quantidade de strings:

```
Pattern regex = Pattern.compile("before");
Matcher regexMatcher = regex.matcher(subjectString);
String resultString = regexMatcher.replaceAll("after");
```

Se a expressão regular, ou o texto de substituição, for fornecido pelo usuário final, você deve usar o objeto Regex com um tratamento de exceção completo:

```
String resultString = null;
try {
  Pattern regex = Pattern.compile("before");
  Matcher regexMatcher = regex.matcher(subjectString);
  try {
    resultString = regexMatcher.replaceAll("after");
  } catch (IllegalArgumentException ex) {
    // Erro de sintaxe no texto de substituição (sinais $ não-escapados?)
```

```
    } catch (IndexOutOfBoundsException ex) {
      // Retrorreferência inexistente utilizou o texto de substituição
    }
  } catch (PatternSyntaxException ex) {
    // Erro de sintaxe na expressão regular
  }
```

JavaScript

```
result = subject.replace(/before/g, "after");
```

PHP

```
$result = preg_replace('/before/', 'after', $subject);
```

Perl

Com a string de assunto contida na variável especial `$_`, armazenando o resultado de volta em `$_`:

```
s/before/after/g;
```

Com a string de assunto contida na variável `$subject`, armazenando o resultado de volta em `$subject`:

```
$subject =~ s/before/after/g;
```

Com a string de assunto contida na variável `$subject`, armazenando o resultado em `$result`:

```
($result = $subject) =~ s/before/after/g;
```

Python

Se você tiver poucas strings para processar, pode usar a função global:

```
result = re.sub("before", "after", subject)
```

Para usar a mesma expressão regular repetidamente, use um objeto compilado:

```
reobj = re.compile("before")
result = reobj.sub("after", subject)
```

Ruby

```
result = subject.gsub(/before/, 'after')
```

Discussão

.NET

No .NET, você sempre usará o método Regex.Replace(), para pesquisar e substituir com uma expressão regular. O método Replace() possui 10 sobrecargas. Metade delas aceita uma string como texto de substituição; estes métodos são discutidos aqui. A outra metade aceita um delegate MatchEvaluator como substituição; estes métodos serão discutidos na receita 3.16.

O primeiro parâmetro esperado por Replace() será sempre a string que armazena o texto de assunto original, que você deseja pesquisar e substituir. Este parâmetro não deve ser null. Caso seja, Replace() lançará uma ArgumentNullException. O valor de retorno de Replace() será sempre a string com as substituições aplicadas.

Se quiser usar a expressão regular apenas algumas vezes, poderá usar uma chamada estática. O segundo parâmetro, então, será a expressão regular que deseja usar. Especifique o texto de substituição como terceiro parâmetro. Você pode passar as opções de expressão regular como um quarto parâmetro opcional. Se sua expressão regular tiver um erro de sintaxe, uma ArgumentException será lançada.

Se quiser usar a mesma expressão regular em várias strings, poderá tornar seu código mais eficiente; primeiro por meio da construção de um objeto Regex e, então, chamando Replace() neste objeto. Passe a string de assunto como primeiro parâmetro e o texto de substituição como segundo parâmetro. Estes são os únicos parâmetros necessários.

Ao chamar Replace() em uma instância da classe Regex, você pode passar parâmetros adicionais, para limitar a pesquisa e a substituição. Se omitir estes parâmetros, todas as correspondências da expressão regular na string de assunto serão substituídas. As sobrecargas estáticas de Replace() não permitem esses parâmetros adicionais; elas sempre substituem todas as correspondências.

Como terceiro parâmetro opcional, após o assunto e a substituição, você pode passar o número de substituições a serem feitas. Se você passar um número maior do que um, este será o máximo de substituições a serem feitas. Por exemplo, Replace(subject, replacement, 3) substitui apenas as três primeiras correspondências da expressão regular. Outras correspondências serão ignoradas. Se houver menos de três correspondências possíveis na string, todas serão substituídas. Você não receberá qualquer indicação de que foram feitas menos substituições do que você pediu. Se você passar zero como terceiro parâmetro, nenhuma substituição será feita, e a string de assunto será retornada inalterada. Se você passar -1, todas as correspondências da expressão regular serão substituídas. Especificar um número menor que -1 fará Replace() lançar uma ArgumentOutOfRangeException.

Se você especificar o terceiro parâmetro com o número de substituições a serem feitas, poderá especificar um quarto parâmetro opcional, para indicar o índice do caractere no qual a expressão regular deverá começar a busca. Essencialmente, o número que você passar como quarto parâmetro será o número de caracteres, no início da string de assunto, que a expressão regular deverá ignorar. Isso pode ser útil quando você já processou a string até um dado ponto, e pretende procurar e substituir apenas em seu restante. Se você especificar um número, ele deverá ficar entre zero e o comprimento da string de assunto. Caso contrário, Replace() lança uma ArgumentOutOfRangeException. Ao contrário de Match(), Replace() não permite que você forneça um parâmetro especificando o comprimento da substring na qual a expressão regular está autorizada a pesquisar.

Java

Se você deseja apenas pesquisar e substituir em uma string com a mesma expressão regular, poderá chamar tanto o método replaceFirst(), quanto replaceAll(), diretamente em sua string. Ambos os métodos aceitam dois parâmetros: uma string com a sua expressão

regular, e uma string com o texto de substituição. Elas são funções convenientes, que chamam Pattern.compile("before").matcher(subjectString).replaceFirst("after") e Pattern.compile("before").matcher(subjectString).replaceAll("after").

Se quiser usar a mesma expressão regular em várias strings, será preciso criar o objeto Matcher, tal como explicado na receita 3.3. Então, chame replaceFirst() ou replaceAll() em seu Matcher, passando o texto de substituição como único parâmetro.

Existem três classes de exceção diferentes com as quais você terá de lidar, caso a expressão regular e o texto de substituição sejam fornecidos pelo usuário final. A classe de exceção PatternSyntaxException é lançada por Pattern.compile(), String.replaceFirst() e String.replaceAll(), caso a expressão regular tenha erro de sintaxe. IllegalArgumentException é lançada por replaceFirst() e replaceAll(), caso haja um erro de sintaxe no texto de substituição. Se o texto de substituição for sintaticamente válido, mas fizer referência a um grupo de captura que não existe, neste caso IndexOutOfBoundsException será lançada.

JavaScript

Para pesquisar e substituir em uma string usando uma expressão regular, chame a função replace(). Passe sua expressão regular como primeiro parâmetro e a string com seu texto de substituição como segundo parâmetro. A função replace() retorna uma nova string com as substituições aplicadas.

Se quiser substituir todas as correspondências da expressão regular na string, defina a sinalização /g ao criar o objeto de expressão regular. A receita 3.4 explica como isso funciona. Se você não usar a sinalização /g, apenas a primeira correspondência será substituída.

PHP

Você pode procurar e substituir facilmente em uma string com

preg_replace(). Passe sua expressão regular como primeiro parâmetro, o texto de substituição como segundo parâmetro e a string de assunto como terceiro parâmetro. O valor de retorno será uma string com as substituições aplicadas.

O quarto parâmetro opcional permite limitar o número de substituições feitas. Se você omitir o parâmetro, ou especificar -1, todas as correspondências da expressão regular serão substituídas. Se você especificar 0, nenhuma substituição será feita. Se você especificar um número positivo, preg_replace() substituirá as correspondências da expressão regular que você especificou. Se houver poucas correspondências, todas são substituídas sem erros.

Se quiser saber quantas substituições foram feitas, poderá adicionar um quinto parâmetro à chamada. Esse parâmetro receberá um número inteiro, com o número de substituições realmente feitas.

Uma característica especial do preg_replace() é que você pode passar arrays, ao invés de strings, para os três primeiros parâmetros. Se você passar como terceiro parâmetro um array de strings, em vez de uma única string, preg_replace() retornará um array com a pesquisa-e-substituição feita em todas as strings.

Se você passar um array de strings de expressões regulares como primeiro parâmetro, preg_replace() utilizará as expressões regulares, uma por uma, para pesquisar e substituir ao longo da string de assunto. Se você passar um array de strings de assunto, todas as expressões regulares serão usadas em todas as strings de assunto. Ao procurar por um array ou por expressões regulares, você pode especificar uma string única como texto de substituição (para ser usada por todas as expressões regulares), ou um array de substituições. Quando utilizamos dois arrays, preg_replace() percorre tanto a expressão regular quanto os arrays de substituição, utilizando um texto de substituição diferente para cada expressão regular. preg_replace() percorre o array conforme ele é armazenado na memória, o que não é, necessariamente, a ordem numérica dos índices no array. Se você não construiu o array em ordem numérica,

chame ksort() com as expressões regulares e textos de substituição, antes de passá-los para preg_replace().

Este exemplo constrói o array $replace na ordem reversa:

```
$regex[0] = '/a/';
$regex[1] = '/b/';
$regex[2] = '/c/';
$replace[2] = '3';
$replace[1] = '2';
$replace[0] = '1';
echo preg_replace($regex, $replace, "abc");
ksort($replace);
echo preg_replace($regex, $replace, "abc");
```

A primeira chamada a preg_replace() exibe 321, que não é o que você espera. Depois de usar ksort(), a substituição retorna 123, como pretendíamos. ksort() modifica a variável que você passa para ela. Não passe seu valor de retorno (verdadeiro ou falso) para preg_replace().

Perl

Em Perl, s/// é, na verdade, um operador de substituição. Se você usar s///, por si só, ele irá procurar e substituir ao longo da variável $_, armazenando o resultado de volta em $_.

Se você quiser usar o operador de substituição em outra variável, use o operador de ligação =~, para associar o operador de substituição à variável. Vincular o operador de substituição a uma string imediatamente executa a pesquisa-e-substituição. O resultado é armazenado na variável que contém a string de assunto.

O operador s/// sempre modifica a variável a ele vinculada. Se quiser armazenar o resultado da pesquisa-e-substituição em uma nova variável, sem alterar a original, primeiro atribua a string original à variável de resultado e, em seguida, vincule à variável o operador de substituição. A solução Perl para esta receita mostra como você pode executar esses dois passos em uma linha de código.

Utilize o modificador /g, explicado na receita 3.4, para substituir

todas as correspondências da expressão regular. Sem ele, o Perl substitui apenas a primeira correspondência.

Python

A função sub(), do módulo re, realiza uma pesquisa-e-substituição usando uma expressão regular. Passe sua expressão regular como primeiro parâmetro, o texto de substituição como segundo parâmetro e a string de assunto como terceiro parâmetro. A função global sub() não aceita um parâmetro com opções de expressão regular.

A função re.sub() chama re.compile() e, em seguida, chama o método sub() no objeto de expressão regular compilado. Este método possui dois parâmetros necessários: o texto de substituição e a string de assunto.

Ambas as formas de sub() retornam uma string com todas as expressões regulares substituídas. Ambas aceitam um parâmetro opcional, que você pode utilizar para limitar o número de substituições. Se você omiti-lo, ou defini-lo, como zero, todas as correspondências da expressão regular serão substituídas. Se você passar um número positivo, este será o número máximo de correspondências a serem substituídas. Se menos correspondências puderem ser encontradas do que a contagem especificada, todas as correspondências serão substituídas sem erros.

Ruby

O método gsub() da classe String faz uma pesquisa-e-substituição usando uma expressão regular. Passe a expressão regular como primeiro parâmetro e uma string com o texto de substituição como segundo parâmetro. O valor de retorno será uma nova string com as substituições aplicadas. Se nenhuma correspondência da expressão regular puder ser encontrada, gsub() retornará a string original.

gsub() não modifica a string na qual você chama o método. Se quiser

que a string original seja modificada, chame gsub!(). Se nenhuma correspondência da expressão regular puder ser encontrada, gsub!() retorna nil. Caso contrário, ele retorna a string em que foi chamado, com as substituições aplicadas.

Veja também:

“Pesquisa e substituição com expressões regulares”, no capítulo 1, e receitas 3.15 e 3.16.

3.15 Substituir correspondências, reutilizando partes da correspondência

Problema

Você deseja executar uma pesquisa-e-substituição que reintroduza partes da correspondência da expressão regular na substituição. As partes que você deseja reinserir foram isoladas em sua expressão regular usando grupos de captura, como descrito na receita 2.9.

Por exemplo, você deseja corresponder a pares de palavras delimitadas por um sinal de igual, e trocar essas palavras na substituição.

Solução

C#

Você pode usar a chamada estática, quando for processar apenas uma pequena quantidade de strings com a mesma expressão regular:

```
string resultString = Regex.Replace(subjectString, @"(\w+)=(\w+)", "$2=$1");
```

Construa um objeto Regex, se quiser usar a mesma expressão regular com uma grande quantidade de strings:

```
Regex regexObj = new Regex(@"(\w+)=(\w+)");
```

```
string resultString = regexObj.Replace(subjectString, "$2=$1");
```

VB.NET

Você pode usar a chamada estática, quando for processar apenas uma pequena quantidade de strings com a mesma expressão regular:

```
Dim ResultString = Regex.Replace(SubjectString, "(\w+)=(\w+)", "$2=$1")
```

Construa um objeto Regex, se quiser usar a mesma expressão regular com uma grande quantidade de strings:

```
Dim RegexObj As New Regex("(\w+)=(\w+)")
Dim ResultString = RegexObj.Replace(SubjectString, "$2=$1")
```

Java

Você pode chamar String.replaceAll(), quando for processar somente uma string com a mesma expressão regular:

```
String resultString = subjectString.replaceAll("(\\w+)=(\\w+)", "$2=$1");
```

Construa um objeto Matcher, se quiser usar a mesma expressão regular com uma grande quantidade de strings:

```
Pattern regex = Pattern.compile("(\\w+)=(\\w+)");
Matcher regexMatcher = regex.matcher(subjectString);
String resultString = regexMatcher.replaceAll("$2=$1");
```

JavaScript

```
result = subject.replace(/(\w+)=(\w+)/g, "$2=$1");
```

PHP

```
$result = preg_replace('/(\w+)=(\w+)/', '$2=$1', $subject);
```

Perl

```
$subject =~ s/(\w+)=(\w+)/$2=$1/g;
```

Python

Se você possui poucas strings para processar, pode usar a função global:

```
result = re.sub(r"(\w+)=(\w+)", r"\2=\1", subject)
```

Para usar a mesma expressão regular repetidamente, use um objeto compilado:

```
reobj = re.compile(r"(\w+)=(\w+)")
result = reobj.sub(r"\2=\1", subject)
```

Ruby

```
result = subject.gsub(/(\w+)=(\w+)/, '\2=\1')
```

Discussão

A expressão regular ‹(\w+)=(\w+)› corresponde ao par de palavras, e captura cada palavra em seu próprio grupo de captura. A palavra antes do sinal de igual é capturada pelo primeiro grupo, e a palavra após o sinal, pelo segundo grupo.

Para a substituição, é necessário especificar que você deseja usar o texto correspondido pelo segundo grupo de captura, seguido por um sinal de igual e do texto correspondido pelo primeiro grupo de captura. Você pode fazer isso com marcadores especiais no texto de substituição. A sintaxe do texto de substituição varia muito entre as diferentes linguagens de programação. “Pesquisa e substituição com expressões regulares”, no capítulo 1, descreve os sabores de texto de substituição, e a receita 2.21 explica como referenciar grupos de captura no texto de substituição.

.NET

No .NET, você pode usar o mesmo método Regex.Replace(), descrito na receita anterior, usando uma string como substituição. A sintaxe para adicionar retrorreferências ao texto de substituição segue o sabor de texto de substituição do .NET, mostrado na receita 2.21.

Java

No Java, você pode usar os mesmos métodos replaceFirst() e replaceAll(), descritos na receita anterior. A sintaxe para adicionar retrorreferências ao texto de substituição segue o sabor Java, descrito neste livro.

JavaScript

No JavaScript, você pode usar o mesmo método string.replace(), descrito na receita anterior. A sintaxe para adicionar retrorreferências ao texto de substituição segue o sabor JavaScript, descrito neste livro.

PHP

No PHP, você pode usar a mesma função preg_replace(), descrita na receita anterior. A sintaxe para adicionar retrorreferências ao texto de substituição segue o sabor PHP, descrito neste livro.

Perl

No Perl, a parte substituição de s/regex/substituição/ é interpretada simplesmente como uma string entre aspas duplas. Você pode usar as variáveis especiais $&, $1, $2 etc., conforme explicado na receita 3.7 e na receita 3.9, na string de substituição. As variáveis são definidas logo depois que a correspondência da expressão regular é encontrada, antes de ser substituída. Você também pode usar essas variáveis em todos os outros códigos Perl. Seus valores persistem até você dizer ao Perl para encontrar outra correspondência de expressão regular.

Todas as outras linguagens de programação, neste livro, fornecem uma chamada de função que interpreta o texto de substituição como uma string. A chamada à função analisa a string para processar retrorreferências, como $1, ou \1. Mas, fora da string de texto de substituição, $1 não significa nada para essas linguagens.

Python

No Python, você pode usar a mesma função sub(), descrita na receita anterior. A sintaxe para adicionar retrorreferências ao texto de substituição segue o sabor Python, descrito neste livro.

Ruby

No Ruby, você pode usar o mesmo método String.gsub(), descrito na receita anterior. A sintaxe para adicionar retrorreferências ao texto de substituição segue o sabor Ruby, descrito neste livro.

Você não pode interpolar variáveis, como $1, no texto de substituição, porque o Ruby faz a interpolação das variáveis antes da chamada a gsub() ser executada. Antes da chamada, gsub() ainda não encontrou nenhuma correspondência. Assim, as retrorreferências não podem ser substituídas. Se você tentar interpolar $1, vai acabar obtendo o texto correspondido pelo primeiro grupo de captura da última correspondência de expressão regular, realizada antes da chamada a gsub().

Ao invés disso, utilize tokens de texto de substituição, como «\1». A função gsub() renova estes tokens, no texto de substituição, a cada correspondência da expressão regular. Recomendo que você use strings entre aspas simples, no texto de substituição. Em strings com aspas duplas, a barra invertida é usada como um escape, e dígitos escapados formam escapes octais. '\1' e "\\1" utilizam o texto correspondido pelo primeiro grupo de captura como substituição, enquanto "\1" substitui o caractere simples literal 0x01.

Captura nomeada

Se você utiliza grupos de captura nomeados em sua expressão regular, poderá referenciá-los por seus nomes, na string de substituição.

C#

Você pode usar a chamada estática, quando for processar apenas uma pequena quantidade de strings com a mesma expressão regular:

```
string resultString = Regex.Replace(subjectString,
                                    @"(?<left>\w+)=(?<right>\w+)", "${right}=${left}");
```

Construa um objeto Regex, se quiser usar a mesma expressão regular com uma grande quantidade de strings:

```
Regex regexObj = new Regex(@"(?<left>\w+)=(?<right>\w+)");
string resultString = regexObj.Replace(subjectString, "${right}=${left}");
```

VB.NET

Você pode usar a chamada estática, quando for processar apenas uma pequena quantidade de strings com a mesma expressão regular:

```
Dim ResultString = Regex.Replace(SubjectString, "(?<left>\w+)=(?<right>\w+)",
"${right}=${left}")
```

Construa um objeto Regex, se quiser usar a mesma expressão regular com uma grande quantidade de strings:

```
Dim RegexObj As New Regex("(?<left>\w+)=(?<right>\w+)")
Dim ResultString = RegexObj.Replace(SubjectString, "${right}=${left}")
```

PHP

```
$result = preg_replace('/(?P<left>\w+)=(?P<right>\w+)/', '$2=$1', $subject);
```

Funções preg do PHP utilizam a biblioteca PCRE, que suporta captura nomeada. As funções preg_match() e preg_match_all() adicionam ao array grupos de captura nomeados com resultados de correspondência. Infelizmente, preg_replace() não fornece uma maneira de usar retrorreferências nomeadas no texto de substituição. Se sua expressão regular possui grupos de captura nomeados, conte os grupos de captura nomeados e os numerados, da esquerda para a direita, para determinar o número da retrorreferência de cada grupo. Use estes números no texto de substituição.

Perl

```
$subject =~ s/(?<left>\w+)=(?<right>\w+)/$+{right}=$+{left}/g;
```

O Perl suporta grupos de captura nomeados a partir da versão 5.10. A variável de hash $+ armazena o texto correspondido por todos os grupos de captura nomeados da expressão regular utilizada por último. Você pode utilizar esse hash na string de texto de substituição, bem como em qualquer outro lugar.

Python

Se você tiver poucas strings para processar, pode utilizar a função global:

```
result = re.sub(r"(?P<left>\w+)=(?P<right>\w+)", r"\g<right>=\g<left>", subject)
```

Para usar a mesma expressão regular repetidamente, use um objeto compilado:

```
reobj = re.compile(r"(?P<left>\w+)=(?P<right>\w+)")
result = reobj.sub(r"\g<right>=\g<left>", subject)
```

Ruby

```
result = subject.gsub(/(?<left>\w+)=(?<right>\w+)/, '\k<left>=\k<right>')
```

Veja também:

O tópico “Pesquisa e substituição com expressões regulares”, no capítulo 1, descreve os sabores de textos de substituição.

A receita 2.21 explica como referenciar grupos de captura no texto de substituição.

3.16 Substituir correspondências com substitutos gerados em código

Problema

Você deseja substituir todas as correspondências de uma expressão regular por uma nova string construída no código procedural. Você deseja ser capaz de substituir cada correspondência por uma string diferente, baseando-se no texto de fato correspondido.

Por exemplo, suponha que você queira substituir todos os números, em uma string, pelo próprio número multiplicado por dois.

Solução

C#

Você pode usar a chamada estática, quando for processar apenas

uma pequena quantidade de strings com a mesma expressão regular:

```
string resultString = Regex.Replace(subjectString, @"\d+",
                                    new MatchEvaluator(ComputeReplacement));
```

Construa um objeto Regex, se quiser usar a mesma expressão regular com uma grande quantidade de strings:

```
Regex regexObj = new Regex(@"\d+");
string resultString = regexObj.Replace(subjectString, new
MatchEvaluator(ComputeReplacement));
```

Ambas as amostras de código chamam a função ComputeReplacement. Você deve adicionar este método à classe em que estiver implementando esta solução:

```
public String ComputeReplacement(Match matchResult) {
  int twiceasmuch = int.Parse(matchResult.Value) * 2;
  return twiceasmuch.ToString();
}
```

VB.NET

Você pode usar a chamada estática, quando for processar apenas uma pequena quantidade de strings com a mesma expressão regular:

```
Dim MyMatchEvaluator As New MatchEvaluator(AddressOf
ComputeReplacement)
Dim ResultString = Regex.Replace(SubjectString, "\d+", MyMatchEvaluator)
```

Construa um objeto Regex, se quiser usar a mesma expressão regular com uma grande quantidade de strings:

```
Dim RegexObj As New Regex("\d+")
Dim MyMatchEvaluator As New MatchEvaluator(AddressOf
ComputeReplacement)
Dim ResultString = RegexObj.Replace(SubjectString, MyMatchEvaluator)
```

Ambas as amostras de código chamam a função ComputeReplacement. Você deve adicionar este método à classe em que estiver implementando esta solução:

```
Public Function ComputeReplacement(ByVal MatchResult As Match) As String
  Dim TwiceAsMuch = Int.Parse(MatchResult.Value) * 2;
  Return TwiceAsMuch.ToString();
```

```
End Function
```

Java

```
StringBuffer resultString = new StringBuffer();
Pattern regex = Pattern.compile("\\d+");
Matcher regexMatcher = regex.matcher(subjectString);
while (regexMatcher.find()) {
  Integer twiceasmuch = Integer.parseInt(regexMatcher.group()) * 2;
  regexMatcher.appendReplacement(resultString, twiceasmuch.toString());
}
regexMatcher.appendTail(resultString);
```

JavaScript

```
var result = subject.replace(/\d+/g,
                             function(match) { return match * 2; }
                             );
```

PHP

Usando uma função callback declarada:

```
$result = preg_replace_callback('/\d+/', compute_replacement, $subject);
function compute_replacement($groups) {
  return $groups[0] * 2;
}
```

Usando uma função callback anônima:

```
$result = preg_replace_callback(
  '/\d+/',
  create_function(
    '$groups',
    'return $groups[0] * 2;'
  ),
  $subject
);
```

Perl

```
$subject =~ s/\d+/$& * 2/eg;
```

Python

Se for processar somente uma pequena quantidade de strings, pode

usar a função global:

```
result = re.sub(r"\d+", computereplacement, subject)
```

Para usar a mesma expressão regular repetidamente, use um objeto compilado:

```
reobj = re.compile(r"\d+")
result = reobj.sub(computereplacement, subject)
```

Ambas as amostras de código chamam a função computereplacement. Esta função precisa ser declarada antes de ser passada a sub().

```
def computereplacement(matchobj):
  return str(int(matchobj.group()) * 2)
```

Ruby

```
result = subject.gsub(/\d+/) {|match|
  Integer(match) * 2
}
```

Discussão

Quando utilizamos uma string como texto de substituição, podemos fazer apenas substituições básicas de texto. Para substituir cada correspondência por algo totalmente diferente, que varie junto com a correspondência sendo substituída, você precisa criar o texto de substituição em seu próprio código.

C#

A receita 3.14 discute as várias maneiras pelas quais você pode chamar o método Regex.Replace(), passando uma string como texto de substituição. Ao utilizar uma chamada estática, a substituição será o terceiro parâmetro, após o assunto e a expressão regular. Se você passou a expressão regular para o construtor Regex(), você pode chamar Replace() no objeto, com a substituição como segundo parâmetro.

Em vez de passar uma string como segundo ou terceiro parâmetro, você pode passar um delegate MatchEvaluator. Esse delegate é uma referência a uma função-membro, que você adiciona à classe em

que estiver fazendo a pesquisa e substituição. Para criar o delegate, use a palavra-chave new, para chamar o construtor MatchEvaluator(). Passe sua função-membro como o único parâmetro de MatchEvaluator().

A função que pretendemos utilizar como delegate deve retornar uma string, e aceita um parâmetro do tipo System.Text.RegularExpressions.Match. Trata-se da mesma classe Match, retornada pelo membro Regex.Match(), usado em quase todas as receitas anteriores deste capítulo.

Quando você chama Replace() com um MatchEvaluator como substituição, sua função será chamada para cada correspondência da expressão regular que precise ser substituída. Sua função precisa retornar o texto de substituição. Você pode usar qualquer uma das propriedades do objeto Match para construir seu texto de substituição. O exemplo mostrado anteriormente usa matchResult.Value para recuperar a string com a correspondência total da expressão regular. Normalmente, você vai usar matchResult.Groups[] para criar seu texto de substituição a partir dos grupos de captura em sua expressão regular.

Se você não deseja substituir determinadas correspondências da expressão regular, sua função deverá retornar matchResult.Value. Se você retornar null ou uma string vazia, a correspondência da expressão regular será substituída por nada (isto é, excluída).

VB.NET

A receita 3.14 discute as várias maneiras como você pode chamar o método Regex.Replace(), passando uma string como texto de substituição. Ao utilizar uma chamada estática, a substituição será o terceiro parâmetro, após o assunto e a expressão regular. Se você usou a palavra-chave Dim para criar uma variável contendo a sua expressão regular, poderá chamar Replace() no objeto, com a substituição como segundo parâmetro.

Em vez de passar uma string como segundo ou terceiro parâmetro,

você poderá passar um objeto MatchEvaluator. Este objeto contém uma referência a uma função que você adiciona à classe em que estiver fazendo a pesquisa-e-substituição. Utilize a palavra-chave Dim, para criar uma nova variável do tipo MatchEvaluator. Passe um parâmetro com a palavra-chave AddressOf, seguida pelo nome da sua função-membro. O operador AddressOf retorna uma referência à função, sem realmente chamá-la naquele momento.

A função que você deseja usar como MatchEvaluator deverá retornar uma string, e deve aceitar um parâmetro do tipo System.Text.RegularExpressions.Match. Trata-se da mesma classe Match, retornada pelo membro Regex.Match(), usado em quase todas as receitas anteriores deste capítulo. O parâmetro será passado por valor; então, você terá que declará-lo com ByVal.

Quando você chama Replace() com um MatchEvaluator como substituição, sua função será chamada para cada correspondência da expressão regular que precise ser substituída. Sua função precisa retornar o texto de substituição. Você pode usar qualquer uma das propriedades do objeto Match para construir seu texto de substituição. O exemplo usa MatchResult.Value para recuperar a string com a correspondência total da expressão regular. Normalmente, você vai usar MatchResult.Groups() para criar seu texto de substituição a partir dos grupos de captura de sua expressão regular.

Se você não deseja substituir determinadas correspondências da expressão regular, sua função deverá retornar MatchResult.Value. Se você retornar Nothing ou uma string vazia, a correspondência da expressão regular será substituída por nada (isto é, excluída).

Java

A solução Java é muito simples. Nós iteramos todas as correspondências da expressão regular, como explicado na receita 3.11. Dentro do loop, chamamos appendReplacement(), em nosso objeto Matcher. Quando find() falhar na tentativa de encontrar qualquer outra correspondência, chamamos appendTail(). Os dois

métodos, appendReplacement() e appendTail(), facilitam o uso de um texto de substituição diferente a cada correspondência da expressão regular.

appendReplacement() aceita dois parâmetros. O primeiro é o StringBuffer, no qual você armazena (temporariamente) o resultado da pesquisa-e-substituição em curso. O segundo é o texto de substituição, a ser utilizado pela última correspondência encontrada por find(). Este texto de substituição pode incluir referências a grupos de captura, como "$1". Se houver um erro de sintaxe no texto de substituição, uma IllegalArgumentException será lançada. Se o texto de substituição referenciar um grupo de captura que não existe, uma IndexOutOfBoundsException será lançada. Se você chamar appendReplacement() sem uma chamada prévia bem sucedida a find(), ele lançará uma IllegalStateException.

Se você chamar appendReplacement() corretamente, ele fará duas coisas. Primeiro, ele copia o texto localizado entre a correspondência da expressão regular anterior e a atual no buffer de string, sem fazer quaisquer modificações ao texto. Se a correspondência em andamento for a primeira, ele copia todo o texto antes dessa correspondência. Depois disso, ele acrescenta seu texto de substituição, substituindo qualquer retrorreferência com o texto correspondido pelos grupos de captura referenciados.

Se quiser excluir uma determinada correspondência, basta substituí-la por uma string vazia. Se quiser deixar inalterada uma correspondência na string, é possível omitir a chamada a appendReplacement() para essa correspondência. Quando digo “correspondência da expressão regular anterior”, refiro-me à correspondência anterior, na qual você chamou appendReplacement(). Se você não chamar appendReplacement() em determinadas correspondências, elas tornam-se parte do texto entre as que você realmente substitui, sendo copiadas na íntegra para o buffer de string-alvo.

Quando você terminar a substituição das correspondências, chame

appendTail() para copiar o texto no final da string, após a última correspondência de expressão regular na qual você chamou appendReplacement().

JavaScript

No JavaScript, uma função é apenas outro objeto que pode ser atribuído a uma variável. Em vez de passar uma string literal ou uma variável que contenha uma string para a função string.replace(), podemos passar uma função que retorne uma string. Esta função é, então, chamada sempre que a substituição precisar ser feita.

Você pode fazer sua função de substituição aceitar um ou mais parâmetros. Se fizer isso, o primeiro parâmetro será definido como o texto correspondido pela expressão regular. Se a sua expressão regular possuir grupos de captura, o segundo parâmetro armazenará o texto correspondido pelo primeiro grupo de captura, o terceiro parâmetro lhe dá o texto do segundo grupo de captura e assim por diante. Você pode configurar esses parâmetros para usar fragmentos da correspondência da expressão regular, para compor a substituição.

A função de substituição, na solução JavaScript desta receita, simplesmente pega o texto correspondido pela expressão regular e o devolve multiplicado por dois. O JavaScript lida implicitamente com as conversões de string para número, e de número para string.

PHP

A função preg_replace_callback() funciona exatamente como a função preg_replace(), descrita na receita 3.14. Ela aceita uma expressão regular, uma substituição, uma string de assunto, o limite de substituição opcional e a contagem de substituição opcional. A expressão regular e a string de assunto podem ser strings simples ou arrays.

A diferença é que preg_replace_callback() não aceita uma string, ou array de strings, como substituição; ela aceita uma função. Você

pode declarar esta função no seu código, ou usar create_function() para criar uma função anônima. A função deve ter um parâmetro e retornar uma string (ou algo que possa ser convertido em uma string).

Sempre que preg_replace_callback() encontrar uma correspondência de expressão regular, ele chamará sua função callback. O parâmetro será preenchido com um array de strings. O elemento zero contém a correspondência total da expressão regular, e os elementos um e além possuem o texto correspondido pelos grupos de captura um e além. Você pode usar este array para construir seu texto de substituição, utilizando o texto correspondido pela expressão regular ou por um ou mais grupos de captura.

Perl

O operador s/// suporta um modificador extra, ignorado pelo operador m//: o modificador /e. O modificador /e, ou "execução", diz ao operador de substituição para executar a parte de substituição como código Perl, ao invés de interpretá-lo como conteúdo de uma string entre aspas duplas. Usando este modificador, podemos, facilmente, obter o texto correspondido com a variável $& e, então, multiplicá-lo por dois. O resultado do código é usado como string de substituição.

Python

A função sub() do Python permite passar o nome de uma função como texto de substituição, em vez de uma string. Essa função é, então, chamada para cada correspondência de expressão regular a ser substituída.

É necessário declarar essa função, antes que você possa fazer referência a ela. Ela deve ter um parâmetro para receber uma instância MatchObject, o mesmo objeto retornado por search(). Você pode usá-la para recuperar (parte da) correspondência da expressão regular, para construir sua substituição. Veja as receitas 3.7 e 3.9 para mais detalhes.

Sua função deverá retornar uma string com o texto de substituição.

Ruby

As duas receitas anteriores chamaram o método gsub() da classe String com dois parâmetros: a expressão regular e o texto de substituição. Esse método também existe em forma de bloco.

Em forma de bloco, gsub() aceita sua expressão regular como único parâmetro. Ele preenche uma variável iteradora com uma string que armazena o texto correspondido pela expressão regular. Se você fornecer variáveis iteradoras adicionais, elas serão definidas como nil, mesmo se a sua expressão regular tiver grupos de captura.

Dentro do bloco, coloque uma expressão avaliada como a string que você pretende utilizar como texto de substituição. Você pode usar as variáveis especiais de correspondência de expressão regular, como $~, $& e $1, dentro do bloco. Seus valores mudam sempre que o bloco fica apto a fazer outra substituição. Veja as receitas 3.7, 3.8 e 3.9, para mais detalhes.

Você não pode usar tokens de texto de substituição, tais como «\1». Eles permanecem como texto literal.

Veja também:

Receitas 3.9 e 3.15.

3.17 Substituir todas as correspondências dentro das correspondências de outra expressão regular

Problema

Você deseja substituir todas as correspondências de uma determinada expressão regular, mas apenas dentro de certas

seções da string de assunto. Outra expressão regular corresponde a cada uma das seções na string.

Digamos que você tenha um arquivo HTML, no qual várias passagens são marcadas com tags de negrito ‹b›. Entre cada par de tags de negrito, você deseja substituir todas as correspondências da expressão regular ‹before› pelo texto de substituição ‹after›. Por exemplo, ao processar a string before first before before before before, você quer terminar com: before first after before after after.

Solução

C#

```
Regex outerRegex = new Regex("<b>.*?</b>", RegexOptions.Singleline);
Regex innerRegex = new Regex("before");
string resultString = outerRegex.Replace(subjectString,
                          new MatchEvaluator(ComputeReplacement));
public String ComputeReplacement(Match matchResult) {
  // Execute a pesquisa-e-substituição interna em cada correspondência da
regex externa
  return innerRegex.Replace(matchResult.Value, "after");
}
```

VB.NET

```
Dim OuterRegex As New Regex("<b>.*?</b>", RegexOptions.Singleline)
Dim InnerRegex As New Regex("before")
Dim MyMatchEvaluator As New MatchEvaluator(AddressOf
ComputeReplacement)
Dim ResultString = OuterRegex.Replace(SubjectString, MyMatchEvaluator)
Public Function ComputeReplacement(ByVal MatchResult As Match) As String
  'Execute a pesquisa-e-substituição interna em cada correspondência da regex
externa
  Return InnerRegex.Replace(MatchResult.Value, "after");
End Function
```

Java

```
StringBuffer resultString = new StringBuffer();
Pattern outerRegex = Pattern.compile("<b>.*?</b>");
```

```
Pattern innerRegex = Pattern.compile("before");
Matcher outerMatcher = outerRegex.matcher(subjectString);
while (outerMatcher.find()) {
  outerMatcher.appendReplacement(resultString,
    innerRegex.matcher(outerMatcher.group()).replaceAll("after"));
}
outerMatcher.appendTail(resultString);
```

JavaScript

```
var result = subject.replace(/<b>.*?<\/b>/g,
                         function(match) {
                            return match.replace(/before/g, "after");
                         } );
```

PHP

```
$result = preg_replace_callback('%<b>.*?</b>%', replace_within_tag, $subject);
function replace_within_tag($groups) {
  return preg_replace('/before/', 'after', $groups[0]);
}
```

Perl

```
$subject =~ s%<b>.*?</b>%($match = $&) =~ s/before/after/g; $match;%eg;
```

Python

```
innerre = re.compile("before")
def replacewithin(matchobj):
  return innerre.sub("after", matchobj.group())
result = re.sub("<b>.*?</b>", replacewithin, subject)
```

Ruby

```
innerre = /before/
result = subject.gsub(/<b>.*?<\/b>/) {|match|
  match.gsub(innerre, 'after')
}
```

Discussão

Esta solução é, mais uma vez, a combinação de duas soluções anteriores, utilizando duas expressões regulares. A expressão regular “externa”, ‹.*?›, corresponde às tags de negrito HTML

e ao texto entre elas. A expressão regular “interna” corresponde a “before”, que iremos substituir por “after”.

A receita 3.16 explica como você pode executar uma pesquisa-e-substituição e construir, em seu próprio código, o texto de substituição para cada correspondência da expressão regular. Aqui, nós fazemos isso com a expressão regular externa. Sempre que ela encontra um par de tags de abertura e fechamento <b>, executamos uma pesquisa-e-substituição usando a expressão regular interna, assim como fizemos na receita 3.14. A string de assunto da pesquisa-e-substituição com a expressão regular interna será o texto correspondido pela expressão regular externa.

Veja também:

Receitas 3.11, 3.13 e 3.16.

3.18 Substituir todas as correspondências entre as correspondências de outra expressão regular

Problema

Você deseja substituir todas as correspondências de uma determinada expressão regular, mas apenas dentro de determinadas seções da string de assunto. Outra expressão regular corresponde ao texto entre as seções. Em outras palavras, você deseja pesquisar-e-substituir ao longo de todas as partes da string de assunto não correspondidas pela outra expressão regular.

Digamos que você tenha um arquivo HTML, no qual deseja substituir as aspas duplas retas por aspas duplas inglesas (smart quotes), mas você só quer substituir as aspas que estejam fora das tags HTML. As aspas dentro de tags HTML devem permanecer

retas simples, no formato ASCII, ou seu navegador web não será capaz de analisar o código HTML. Por exemplo, você deseja transformar "text" "text" "text" em “text”“text” “text”.

Solução

C#

```
string resultString = null;
Regex outerRegex = new Regex("<[^<>]*>");
Regex innerRegex = new Regex("\"([^\"]*)\"");
// Encontre a primeira seção
int lastIndex = 0;
Match outerMatch = outerRegex.Match(subjectString);
while (outerMatch.Success) {
  // Pesquise-e-substitua ao longo do texto entre esta correspondência e a
anterior
  string textBetween = subjectString.Substring(lastIndex, outerMatch.Index -
lastIndex);
  resultString = resultString + innerRegex.Replace(textBetween,
"\u201C$1\u201D");
  lastIndex = outerMatch.Index + outerMatch.Length;
  // Copie o texto da seção, sem alterações
  resultString = resultString + outerMatch.Value;
  // Encontre a próxima seção
  outerMatch = outerMatch.NextMatch();
}
// Pesquise-e-substitua ao longo do que sobrou após a última correspondência
regex
string textAfter = subjectString.Substring(lastIndex, subjectString.Length -
lastIndex);
resultString = resultString + innerRegex.Replace(textAfter, "\u201C$1\u201D");
```

VB.NET

```
Dim ResultString As String = Nothing
Dim OuterRegex As New Regex("<[^<>]*>")
Dim InnerRegex As New Regex("""([^""]*)""")
'Encontre a primeira seção
Dim LastIndex = 0
Dim OuterMatch = OuterRegex.Match(SubjectString)
While OuterMatch.Success
```

```
  'Pesquise-e-substitua ao longo do texto entre esta correspondência e a anterior
  Dim TextBetween = SubjectString.Substring(LastIndex, OuterMatch.Index -
LastIndex);
  ResultString = ResultString + InnerRegex.Replace(TextBetween,
                      ChrW(&H201C) + "$1" + ChrW(&H201D))
  LastIndex = OuterMatch.Index + OuterMatch.Length
  'Copie o texto da seção, sem alterações
  ResultString = ResultString + OuterMatch.Value
  'Encontre a próxima seção
  OuterMatch = OuterMatch.NextMatch
End While
'Pesquise-e-substitua ao longo do que sobrou após a última correspondência
regex
Dim TextAfter = SubjectString.Substring(LastIndex, SubjectString.Length -
LastIndex);
ResultString = ResultString + InnerRegex.Replace(TextAfter, ChrW(&H201C) +
"$1" + ChrW(&H201D))
```

Java

```
StringBuffer resultString = new StringBuffer();
Pattern outerRegex = Pattern.compile("<[^<>]*>");
Pattern innerRegex = Pattern.compile("\"([^\"]*)\"");
Matcher outerMatcher = outerRegex.matcher(subjectString);
int lastIndex = 0;
while (outerMatcher.find()) {
  // Pesquise-e-substitua ao longo do texto entre esta correspondência e a
anterior
  String textBetween = subjectString.substring(lastIndex, outerMatcher.start());
  Matcher innerMatcher = innerRegex.matcher(textBetween);
  resultString.append(innerMatcher.replaceAll("\u201C$1\u201D"));
  lastIndex = outerMatcher.end();
  // Anexe a correspondência regex em si, sem alterações
  resultString.append(outerMatcher.group());
}
// Pesquise-e-substitua ao longo do que sobrou após a última correspondência
regex
String textAfter = subjectString.substring(lastIndex);
Matcher innerMatcher = innerRegex.matcher(textAfter);
resultString.append(innerMatcher.replaceAll("\u201C$1\u201D"));
```

JavaScript

```
var result = "";
```

```
var outerRegex = /<[^<>]*>/g;
var innerRegex = /"([^"]*)"/g;
var outerMatch = null;
var lastIndex = 0;
while (outerMatch = outerRegex.exec(subject)) {
  if (outerMatch.index == outerRegex.lastIndex) outerRegex.lastIndex++;
  // Pesquise-e-substitua ao longo do texto entre esta correspondência e a
anterior
  var textBetween = subject.substring(lastIndex, outerMatch.index);
  result = result + textBetween.replace(innerRegex, "\u201C$1\u201D");
  lastIndex = outerMatch.index + outerMatch[0].length;
  // Anexe a correspondência regex em si, sem alterações
  result = result + outerMatch[0];
}
// Pesquise-e-substitua ao longo do que sobrou após a última correspondência
regex
var textAfter = subject.substr(lastIndex);
result = result + textAfter.replace(innerRegex, "\u201C$1\u201D");
```

PHP

```
$result = '';
$lastindex = 0;
while (preg_match('/<[^<>]*>/', $subject, $groups, PREG_OFFSET_CAPTURE,
$lastindex)) {
  $matchstart = $groups[0][1];
  $matchlength = strlen($groups[0][0]);
  // Pesquise-e-substitua ao longo do texto entre esta correspondência e a
anterior
  $textbetween = substr($subject, $lastindex, $matchstart-$lastindex);
  $result .= preg_replace('/"([^"]*)"/', '"$1"', $textbetween);
  // Anexe a correspondência regex em si, sem alterações
  $result .= $groups[0][0];
  // Mova a posição inicial para a próxima correspondência
  $lastindex = $matchstart + $matchlength;
  if ($matchlength == 0) {
    // Não caia em um loop caso a regex permita correspondências de
comprimento zero
    $lastindex++;
  }
}
// Pesquise-e-substitua ao longo do que sobrou após a última correspondência
regex
$textafter = substr($subject, $lastindex);
```

```
$result .= preg_replace('/"([^"]*)"/', '"$1"', $textafter);
```

Perl

```
use encoding "utf-8";
$result = '';
while ($subject =~ m/<[^<>]*>/g) {
  $match = $&;
  $textafter = $';
  ($textbetween = $`) =~ s/"([^"]*)"/\x{201C}$1\x{201D}/g;
  $result .= $textbetween . $match;
}
$textafter =~ s/"([^"]*)"/\x{201C}$1\x{201D}/g;
$result .= $textafter;
```

Python

```
innerre = re.compile('"([^"]*)"')
result = "";
lastindex = 0;
for outermatch in re.finditer("<[^<>]*>", subject):
  # Pesquise-e-substitua ao longo do texto entre esta correspondência e a
anterior
  textbetween = subject[lastindex:outermatch.start()]
  result += innerre.sub(u"\u201C\\1\u201D", textbetween)
  lastindex = outermatch.end()
  # Anexe a correspondência regex em si, sem alterações
  result += outermatch.group()
# Pesquise-e-substitua ao longo do que sobrou após a última correspondência
regex
textafter = subject[lastindex:]
result += innerre.sub(u"\u201C\\1\u201D", textafter)
```

Ruby

```
result = '';
textafter = ''
subject.scan(/<[^<>]*>/) {|match|
  textafter = $'
  textbetween = $`.gsub(/"([^"]*)"/, '"\1"')
  result += textbetween + match
}
result += textafter.gsub(/"([^"]*)"/, '"\1"')
```

Discussão

A receita 3.13 explica como usar duas expressões regulares para encontrar correspondências (da segunda expressão regular) somente dentro de determinadas seções do arquivo (correspondências da primeira expressão regular). A solução para esta receita usa a mesma técnica de pesquisar e substituir apenas ao longo de algumas partes da string de assunto.

É importante que a expressão regular usada para encontrar as seções continue trabalhando com a string de assunto original. Se você modificar a string original, terá de mudar a posição inicial da expressão regular que localiza a seção, conforme a expressão regular interna adiciona ou apaga caracteres. Mais importante ainda: as alterações podem ter efeitos colaterais indesejados. Por exemplo, se sua expressão regular externa utiliza a âncora ‹^› para corresponder a algo no início de uma linha, e sua expressão regular interna insere uma quebra de linha no final da seção encontrada pela expressão regular externa, ‹^› corresponderá logo após a seção anterior, devido à nova quebra de linha recém-inserida.

Embora as soluções para esta receita sejam bastante longas, elas são muito simples. Duas expressões regulares são usadas. A expressão regular “externa”, ‹<[^<>]*>›, corresponde a um par de colchetes angulares e a qualquer coisa entre eles, exceto os colchetes angulares. Trata-se de uma maneira bruta de corresponder a qualquer tag HTML. Essa expressão regular funciona bem, desde que o arquivo HTML não contenha colchetes angulares literais que não tenham sido codificados (incorretamente) como entidades. Implementamos esta expressão regular com o mesmo código mostrado na receita 3.11. A única diferença é que o comentário naquele código, que nos dizia onde usar a correspondência, foi substituído pelo código real de pesquisa-e-substituição.

A pesquisa-e-substituição dentro do loop segue o código mostrado na receita 3.14. A string de assunto da pesquisa-e-substituição é o

texto entre a correspondência anterior da expressão regular externa, e a correspondência atual. Nós anexamos o resultado da pesquisa-e-substituição interna à string de resultado global. Nós também anexamos a correspondência atual inalterada da expressão regular externa.

Quando a expressão regular externa não encontrar mais correspondências, executamos a pesquisa-e-substituição interna, uma vez mais, no texto após a última correspondência da expressão regular externa.

A expressão regular ‹"([^"]*)"›, utilizada pela pesquisa-e-substituição dentro do loop, corresponde a um par de aspas duplas, e a qualquer coisa entre elas, exceto aspas duplas. O texto entre aspas é capturado no primeiro grupo de captura.

Para o texto de substituição, usamos uma referência ao primeiro grupo de captura, colocado entre duas aspas inglesas. As aspas inglesas ocupam os pontos de código Unicode U+201C e U+201D. Via de regra, você pode simplesmente colar as aspas inglesas direto em seu código-fonte. No entanto, o Visual Studio 2008 insiste em ser esperto, e substitui automaticamente as aspas inglesas por aspas normais.

Em uma expressão regular, você pode corresponder a um ponto de código Unicode com ‹\u201C› ou ‹\x{201C}›, mas nenhuma das linguagens de programação abordadas neste livro suporta esses tokens como parte do texto de substituição. Se um usuário final quiser inserir aspas inglesas no texto de substituição, digitadas por ele em um editor, ele terá de colá-las, literalmente, a partir de um mapa de carateres. Em seu código-fonte, você pode usar escapes Unicode no texto de substituição, caso sua linguagem suporte estes escapes como parte de strings literais. Por exemplo, o C# e o Java suportam \u201C no nível de string, mas o VB.NET não oferece uma maneira de escapar caracteres Unicode em strings. No VB.NET, você pode usar a função ChrW para converter um ponto de código Unicode em um caractere.

Perl e Ruby

As soluções Perl e Ruby utilizam duas variáveis especiais, disponíveis nessas linguagens, que ainda não explicamos. $`(cifrão-crase) contém a parte do texto à esquerda da correspondência de assunto, e $' (cifrão-aspa simples) contém a parte do texto à direita da correspondência de assunto. Em vez de iterar as correspondências na string de assunto original, iniciamos uma nova pesquisa na parte da string após a correspondência anterior. Dessa forma, podemos facilmente recuperar o texto entre a correspondência atual e a anterior, usando $`.

Python

O resultado deste código é uma string Unicode, pois o texto de substituição é especificado como tal. Você pode precisar chamar encode() para poder exibi-lo, por exemplo:

```
print result.encode('1252')
```

Veja também:

Receitas 3.11, 3.13 e 3.16.

3.19 Dividir uma string

Problema

Você deseja dividir uma string usando uma expressão regular. Após a divisão, você terá um array ou lista de strings, contendo o texto entre as correspondências da expressão regular.

Por exemplo, você deseja dividir uma string com tags HTML, utilizando as próprias tags HTML como elementos de divisão. A divisão de I□like□bold□and□italic□fonts deverá resultar em um array de cinco strings: I□like□, bold, □and□, italic e □fonts.

Solução

C#

Você pode usar a chamada estática quando for processar apenas uma pequena quantidade de strings com a mesma expressão regular:

```
string[] splitArray = Regex.Split(subjectString, "<[^<>]*>");
```

Se a expressão regular for fornecida pelo usuário final, você deve usar a chamada estática com um tratamento de exceção completo:

```
string[] splitArray = null;
try {
  splitArray = Regex.Split(subjectString, "<[^<>]*>");
} catch (ArgumentNullException ex) {
  // Não pode passar null como expressão regular ou string de assunto
} catch (ArgumentException ex) {
  // Erro de sintaxe na expressão regular
}
```

Construa um objeto Regex, se quiser usar a mesma expressão regular com uma grande quantidade de strings:

```
Regex regexObj = new Regex("<[^<>]*>");
string[] splitArray = regexObj.Split(subjectString);
```

Se a expressão regular for fornecida pelo usuário final, você deve usar o objeto Regex com um tratamento de exceção completo:

```
string[] splitArray = null;
try {
  Regex regexObj = new Regex("<[^<>]*>");
  try {
    splitArray = regexObj.Split(subjectString);
  } catch (ArgumentNullException ex) {
    // Não pode passar null como string de assunto
  }
} catch (ArgumentException ex) {
  // Erro de sintaxe na expressão regular
}
```

VB.NET

Você pode usar a chamada estática quando for processar apenas uma pequena quantidade de strings com a mesma expressão regular:

```
Dim SplitArray = Regex.Split(SubjectString, "<[^<>]*>")
```

Se a expressão regular for fornecida pelo usuário final, você deve usar a chamada estática com um tratamento de exceção completo:

```
Dim SplitArray As String()
Try
  SplitArray = Regex.Split(SubjectString, "<[^<>]*>")
Catch ex As ArgumentNullException
  'Não pode passar null como expressão regular ou string de assunto
Catch ex As ArgumentException
  'Erro de sintaxe na expressão regular
End Try
```

Construa um objeto `Regex`, se quiser usar a mesma expressão regular com uma grande quantidade de strings:

```
Dim RegexObj As New Regex("<[^<>]*>")
Dim SplitArray = RegexObj.Split(SubjectString)
```

Se a expressão regular for fornecida pelo usuário final, você deve usar o objeto `Regex` com um tratamento de exceção completo:

```
Dim SplitArray As String()
Try
  Dim RegexObj As New Regex("<[^<>]*>")
  Try
    SplitArray = RegexObj.Split(SubjectString)
  Catch ex As ArgumentNullException
    'Não pode passar null como string de assunto
  End Try
Catch ex As ArgumentException
  'Erro de sintaxe na expressão regular
End Try
```

Java

Você pode chamar `String.Split()` diretamente, quando quiser divider apenas uma string com a mesma expressão regular.

```
String[] splitArray = subjectString.split("<[^<>]*>");
```

Se a expressão regular for fornecida pelo usuário final, você deve usar um tratamento de exceção completo:

```
try {
  String[] splitArray = subjectString.split("<[^<>]*>");
```

```
} catch (PatternSyntaxException ex) {
  // Erro de sintaxe na expressão regular
}
```

Construa um objeto Pattern, se quiser usar a mesma expressão regular com uma grande quantidade de strings:

```
Pattern regex = Pattern.compile("<[^<>]*>");
String[] splitArray = regex.split(subjectString);
```

Se a expressão regular for fornecida pelo usuário final, você deve usar o objeto Pattern com um tratamento de exceção completo:

```
String[] splitArray = null;
try {
  Pattern regex = Pattern.compile("<[^<>]*>");
  splitArray = regex.split(subjectString);
} catch (ArgumentException ex) {
  // Erro de sintaxe na expressão regular
}
```

JavaScript

A função string.split() pode dividir uma string utilizando uma expressão regular:

```
result = subject.split(/<[^<>]*>/);
```

Infelizmente, há uma série de questões de compatibilidade entre navegadores, quando utilizamos string.split() com uma expressão regular. Construir a lista em seu próprio código é mais confiável:

```
var list = [];
var regex = /<[^<>]*>/g;
var match = null;
var lastIndex = 0;
while (match = regex.exec(subject)) {
  // Não permita que navegadores como o Firefox caiam em um loop infinito
  if (match.index == regex.lastIndex) regex.lastIndex++;
  // Adicione o texto anterior à correspondência
  list.push(subject.substring(lastIndex, match.index));
  lastIndex = match.index + match[0].length;
}
// Adicione o restante após a última correspondência
list.push(subject.substr(lastIndex));
```

PHP

```
$result = preg_split('/<[^<>]*>/', $subject);
```

Perl

```
@result = split(m/<[^<>]*>/, $subject);
```

Python

Se você possui somente uma pequena quantidade de strings para dividir, use a função global:

```
result = re.split("<[^<>]*>", subject))
```

Para usar a mesma expressão regular repetidamente, use um objeto compilado:

```
reobj = re.compile("<[^<>]*>")
result = reobj.split(subject)
```

Ruby

```
result = subject.split(/<[^<>]*>/)
```

Discussão

Dividir uma string usando uma expressão regular produz, essencialmente, o resultado oposto da receita 3.10. Em vez de recuperar uma lista com todas as correspondências da expressão regular, você obtém uma lista do texto entre as correspondências, incluindo o texto antes da primeira e após a última correspondência. As correspondências da expressão regular, em si, são omitidas da saída da função de divisão.

C# e VB.NET

No .NET, você vai, sempre, usar o método Regex.Split() para dividir uma string com uma expressão regular. O primeiro parâmetro esperado por Split() será sempre a string com o texto de assunto original que você deseja dividir. Este parâmetro não deve ser null. Caso seja, Split() vai lançar uma ArgumentNullException. O valor de retorno de Split() será sempre um array de strings.

Se quiser usar a expressão regular apenas algumas vezes, você pode usar uma chamada estática. O segundo parâmetro, então, será a expressão regular que deseja usar. Você pode passar opções de expressão regular como um terceiro parâmetro opcional. Se sua expressão regular tiver um erro de sintaxe, uma ArgumentException será lançada.

Se quiser usar a mesma expressão regular em muitas strings, pode tornar seu código mais eficiente. Primeiro, construindo um objeto Regex e, em seguida, chamando Split() neste objeto. A string de assunto, então, é o único parâmetro obrigatório.

Ao chamar Split() em uma instância da classe Regex, você pode passar parâmetros adicionais para limitar a operação de divisão. Se omitir esses parâmetros, a string será dividida em todas as correspondências da expressão regular na string de assunto. As versões estáticas de Split() não permitem esses parâmetros adicionais. Elas sempre dividem toda a string em todas as correspondências.

Como um segundo parâmetro opcional, após a string de assunto, você pode passar o número máximo de strings divididas que deseja obter. Por exemplo, ao chamar regexObj.Split(subject, 3), você receberá um array com no máximo três strings. A função Split() tentará encontrar duas correspondências da expressão regular, e retornará um array com o texto antes da primeira correspondência, o texto entre as duas correspondências e o texto após a segunda correspondência. Qualquer possibilidade de novas correspondências, no restante da string de assunto, será ignorada e deixada na última string do array.

Se não houver correspondências da expressão regular o suficiente para chegar a seu limite, Split() vai dividir ao longo de todas as correspondências da expressão regular disponíveis, e retornará um array com menos strings do que você especificou. regexObj.Split(subject, 1) não divide a string, retornando um array com a string original como seu único elemento. regexObj.Split(subject, 0) divide

em todas as correspondências da expressão regular, assim como faz Split(), quando você omite o segundo parâmetro. Especificar um número negativo fará com que Split() lance uma ArgumentOutOfRangeException.

Se especificar o segundo parâmetro com o número máximo de strings do array retornado, você também poderá especificar um terceiro parâmetro opcional, para indicar o índice do caractere em que a expressão regular deverá começar a encontrar correspondências. Essencialmente, o número que você passar como terceiro parâmetro é o número de caracteres, no início da string de assunto, que a expressão regular deverá ignorar. Isso pode ser útil quando você já processou a string até certo ponto, e só deseja dividir o restante da string.

Os caracteres ignorados pela expressão regular ainda serão adicionados ao array retornado. A primeira string no array será toda a substring anterior à primeira correspondência da expressão regular encontrada após a posição inicial especificada, incluindo os caracteres anteriores à posição inicial. Se você especificar o terceiro parâmetro, ele deverá estar entre zero e o comprimento da string do assunto. Caso contrário, Split() lança uma ArgumentOutOfRangeException. Ao contrário de Match(), Split() não permite que você especifique um parâmetro para definir o comprimento da substring na qual a expressão regular está autorizada a pesquisar.

Se ocorrer uma correspondência no início da string de assunto, a primeira string, no array resultante, será vazia. Quando duas correspondências da expressão regular puderem ser encontradas uma ao lado da outra, na string de assunto, sem texto entre elas, uma string vazia será adicionada ao array. Se uma correspondência ocorrer no final da string de assunto, o último elemento no array será uma string vazia.

Java

Se você tiver apenas uma string para dividir, pode chamar o método split() diretamente em sua string de assunto. Passe a expressão regular como único parâmetro. Este método simplesmente chama Pattern.compile("regex").split(subjectString).

Se quiser dividir múltiplas strings, utilize a fábrica Pattern.compile() para criar um objeto Pattern. Dessa forma, a sua expressão regular precisará ser compilada apenas uma vez. Em seguida, chame o método split() em sua instância Pattern, e passe sua string de assunto como parâmetro. Não há necessidade de criar um objeto Matcher. A classe Matcher não possui um método split().

Pattern.split() aceita um segundo parâmetro opcional, mas String.split(), não. Você pode usar o segundo parâmetro para passar o número máximo de strings divididas que deseja obter. Por exemplo, ao chamar Pattern.split(subject, 3), você receberá um array com no máximo três strings. A função split() tentará encontrar duas correspondências da expressão regular, e retornará um array com o texto antes da primeira correspondência, o texto entre as duas correspondências e o texto após a segunda correspondência. Qualquer nova possibilidade de correspondência, no restante da string de assunto, será ignorada e deixada na última string do array. Se não houver correspondências da expressão regular o suficiente para chegar a seu limite, split() vai dividir ao longo de todas as correspondências disponíveis da expressão regular, e retornará um array com menos strings do que você especificou. Pattern.split(subject, 1) não divide a string, retornando um array com a string original como seu único elemento.

Se ocorrer uma correspondência no início da string do assunto, a primeira string, no array resultante, será vazia. Quando duas correspondências da expressão regular puderem ser encontradas uma ao lado da outra na string de assunto, sem texto entre elas, uma string vazia será adicionada ao array. Se ocorrer uma correspondência no final da string de assunto, o último elemento do array será uma string vazia.

No entanto, o Java eliminará as strings vazias no final do array. Se quiser que elas sejam incluídas, passe um número negativo como segundo parâmetro de Pattern.split(). Esse procedimento diz ao Java para dividir a string quantas vezes for possível, deixando qualquer string vazia no final do array. O valor real do segundo parâmetro não faz diferença, quando negativo. Você não pode dizer ao Java para dividir uma string um determinado número de vezes e, ao mesmo tempo, deixar strings vazias no final do array.

JavaScript

No JavaScript, chame o método split() na string que deseja dividir. Passe a expressão regular como único parâmetro, para obter um array com a string dividida, tantas vezes quanto possível. Você pode passar um segundo parâmetro opcional, para especificar o número máximo de strings que deseja obter no array retornado. Este número deverá ser positivo. Se passar zero, terá um array vazio. Se omitir o segundo parâmetro ou passar um número negativo, a string será dividida o quanto for possível. Definir a sinalização /g na expressão regular (Receita 3.4) não faz diferença.

Infelizmente, nenhum dos navegadores atuais implementa todos os aspectos do método split(), conforme especificado na norma do JavaScript. Em particular, alguns navegadores incluem no array o texto correspondido pelos grupos de captura, e outros, não. Aqueles que incluem grupos de captura não manipulam grupos de não-captura de forma consistente. Para evitar tais problemas, use apenas grupos de não-captura (Receita 2.9) em expressões regulares em que for passar a split().

Algumas implementações de JavaScript omitem strings de comprimento zero do array retornado. Strings de comprimento zero deveriam constar no array quando duas correspondências da expressão regular ocorrem uma ao lado da outra, ou quando a expressão regular corresponde ao início, ou ao final, da string que está sendo dividida. Como você não pode resolver essa questão com uma simples mudança de expressão regular, provavelmente é

mais seguro usar a solução de JavaScript mais longa, apresentada para esta receita. Essa solução inclui todas as strings de comprimento zero, mas você pode, facilmente, editá-la para excluir as ditas strings.

A solução longa é uma adaptação da receita 3.12. Ela acrescenta o texto entre as correspondências de expressão regular, e as próprias correspondências da expressão regular, a um array. Para obter o texto entre as correspondências, usamos os detalhes destas, explicados na receita 3.8.

Se quiser uma implementação de String.prototype.split que siga o padrão, e que também funcione em todos os navegadores, Steven Levithan apresenta uma solução em *http://blog.stevenlevithan.com/archives/cross-browser-split*.

PHP

Chame preg_split() para dividir uma string em um array de strings, ao longo das correspondências da expressão regular. Passe a expressão regular como primeiro parâmetro, e a string de assunto como segundo parâmetro. Se omitir o segundo parâmetro, $_ será usado como a string de assunto.

Você pode passar um terceiro parâmetro opcional, para especificar o número máximo de strings divididas. Por exemplo, ao chamar preg_split($regex, $subject, 3), você receberá um array com, no máximo, três strings. A função preg_split() tentará encontrar duas correspondências da expressão regular, e retornará um array com o texto antes da primeira correspondência, o texto entre as duas correspondências e o texto após a segunda correspondência. Uma outra possibilidade de nova correspondência, qualquer que seja, no restante da string de assunto, será ignorada e deixada na última string do array. Se não houver correspondências de expressão regular o suficiente para chegar a seu limite, preg_split() dividirá ao longo de todas as correspondências disponíveis da expressão regular, e retornará um array com menos strings, em relação às

especificadas. Se omitir o terceiro parâmetro, ou defini-lo como -1, a string será dividida tanto quanto possível.

Se ocorrer uma correspondência no início da string de assunto, a primeira string, no array resultante, será vazia. Quando duas correspondências da expressão regular puderem ser encontradas, uma ao lado da outra na string de assunto, sem texto entre elas, uma string vazia será adicionada ao array. Se uma correspondência ocorrer no final da string de assunto, o último elemento do array será uma string vazia. Por padrão, preg_split() inclui as strings vazias no array retornado. Se você quiser strings vazias no array, passe a constante PREG_SPLIT_NO_EMPTY como o quarto parâmetro.

Perl

Chame a função split() para dividir uma string em um array de strings, ao longo das correspondências da expressão regular. Passe um operador de expressão regular como primeiro parâmetro e a string de assunto como segundo parâmetro.

Você pode passar um terceiro parâmetro opcional, para especificar o número máximo de strings divididas. Por exemplo, ao chamar split(/regex/, subject, 3), você receberá um array com, no máximo, três strings. A função split() tentará encontrar duas correspondências da expressão regular, e retornará um array com o texto antes da primeira correspondência, o texto entre as duas correspondências e o texto após a segunda correspondência. Outras possibilidades de correspondências, no restante da string de assunto, serão ignoradas e deixadas na última string do array. Se não houver correspondências da expressão regular o suficiente para chegar a seu limite, split() vai dividir ao longo de todas as correspondências disponíveis da expressão regular, e retornará um array com menos strings do que você especificou.

Se omitir o terceiro parâmetro, o Perl determinará o limite adequado. Se você atribuir o resultado a uma variável de array, como faz a solução para esta receita, a string será dividida tanto quanto

possível. Se você atribuir o resultado a uma lista de variáveis escalares, Perl define o limite como o número de variáveis mais um. Em outras palavras, o Perl tentará preencher todas as variáveis, descartando o restante não dividido. Por exemplo, ($one, $two, $three) = split(/,/) divide $_ com um limite de 4.

Se ocorrer uma correspondência no início da string de assunto, a primeira string no array resultante será vazia. Quando duas correspondências da expressão regular puderem ser encontradas, uma ao lado da outra na string de assunto, sem texto entre elas, uma string vazia será adicionada ao array. Se uma correspondência ocorrer no final da string de assunto, o último elemento do array será uma string vazia.

Python

A função split() do módulo re divide uma string usando uma expressão regular. Passe sua expressão regular como primeiro parâmetro e a string de assunto como segundo parâmetro. A função global split() não aceita um parâmetro com opções de expressão regular.

A função re.split() chama re.compile() e, em seguida, o método split() no objeto de expressão regular compilado. Este método possui apenas um parâmetro obrigatório: a string de assunto.

Ambas as formas de split() retornam uma lista com o texto entre todas as correspondências da expressão regular. Ambas aceitam um parâmetro opcional, que você pode utilizar para limitar o número de divisões aplicadas à string. Se omiti-lo, ou defini-lo como zero, a string será dividida tanto quanto possível. Se passar um número positivo, este será o número máximo de divisões de string nas correspondências da expressão regular. A lista resultante conterá uma string a mais, em relação à contagem especificada por você. A última string contém o resto não dividido da string de assunto após a última correspondência da expressão regular. Se menos correspondências puderem ser encontradas, em relação à

contagem especificada, a string será dividida em todas as correspondências da expressão regular, sem erros.

Ruby

Chame o método split(), na string de assunto, e passe sua expressão regular como primeiro parâmetro, para dividir a string em um array de strings ao longo das correspondências da expressão regular.

O método split() possui um segundo parâmetro opcional, que pode ser usado para indicar o número máximo de strings divididas. Por exemplo, ao chamar subject.split(re, 3), você receberá um array com no máximo três strings. A função split() tentará encontrar duas correspondências da expressão regular, e retornará um array com o texto antes da primeira correspondência, o texto entre as duas correspondências e o texto após a segunda correspondência. Outras possibilidades de correspondências, no restante da string de assunto, serão ignoradas e deixadas na última string do array. Se não houver correspondências da expressão regular o suficiente para chegar ao limite, split() vai dividir ao longo de todas as correspondências da expressão regular disponíveis, e retornará um array com menos strings do que você especificou. split(re, 1) não divide a string, retornando um array com a string original como único elemento.

Se ocorrer uma correspondência no início da string de assunto, a primeira string no array resultante será vazia. Quando duas correspondências da expressão regular puderem ser encontradas, uma ao lado da outra na string de assunto, sem texto entre elas, uma string vazia será adicionada ao array. Se uma correspondência ocorrer no final da string de assunto, o último elemento no array será uma string vazia.

No entanto, o Ruby eliminará strings vazias no final do array. Se quiser que as strings vazias sejam incluídas, passe um número negativo como segundo parâmetro de split(). Isto diz ao Ruby para dividir a string quantas vezes for possível, deixando qualquer string

vazia no final do array. O valor real do segundo parâmetro não faz diferença, quando negativo. Você não pode dizer ao Ruby para dividir uma string um determinado número de vezes e, ao mesmo tempo, deixar strings vazias no final do array.

Veja também:

Receita 3.20.

3.20 Dividir uma string, mantendo as correspondências da expressão regular

Problema

Você deseja dividir uma string usando uma expressão regular. Após a divisão, você terá um array, ou lista de strings, com o texto entre as correspondências da expressão regular, além das próprias correspondências da expressão regular.

Suponha que você queira dividir uma string com tags HTML utilizando as próprias tags HTML como elementos de divisão, mantendo-as. A divisão de I□like□bold□and□italic□fonts resultaria em um array de nove strings: I□like□, , bold, , □and□, , italic, e □fonts.

Solução

C#

Você pode usar a chamada estática, quando for processar apenas uma pequena quantidade de strings com a mesma expressão regular:

```
string[] splitArray = Regex.Split(subjectString, "(<[^<>]*>)");
```

Construa um objeto Regex, se quiser usar a mesma expressão regular com uma grande quantidade de strings:

```
Regex regexObj = new Regex("(<[^<>]*>)");
string[] splitArray = regexObj.Split(subjectString);
```

VB.NET

Você pode usar a chamada estática, quando for processar apenas uma pequena quantidade de strings com a mesma expressão regular:

```
Dim SplitArray = Regex.Split(SubjectString, "(<[^<>]*>)")
```

Construa um objeto Regex, se quiser usar a mesma expressão regular com uma grande quantidade de strings:

```
Dim RegexObj As New Regex("(<[^<>]*>)")
Dim SplitArray = RegexObj.Split(SubjectString)
```

Java

```
List<String> resultList = new ArrayList<String>();
Pattern regex = Pattern.compile("<[^<>]*>");
Matcher regexMatcher = regex.matcher(subjectString);
int lastIndex = 0;
while (regexMatcher.find()) {
  resultList.add(subjectString.substring(lastIndex, regexMatcher.start()));
  resultList.add(regexMatcher.group());
  lastIndex = regexMatcher.end();
}
resultList.add(subjectString.substring(lastIndex));
```

JavaScript

```
var list = [];
var regex = /<[^<>]*>/g;
var match = null;
var lastIndex = 0;
while (match = regex.exec(subject)) {
  // Não deixe que navegadores como o Firefox caiam em um loop infinito
  if (match.index == regex.lastIndex) regex.lastIndex++;
  // Adicione o texto anterior à correspondência, bem como a correspondência
em si
  list.push(subject.substring(lastIndex, match.index), match[0]);
  lastIndex = match.index + match[0].length;
}
// Adicione o restante após a última correspondência
list.push(subject.substr(lastIndex));
```

PHP

```
$result = preg_split('/(<[^<>]*>)/', $subject, -1,
PREG_SPLIT_DELIM_CAPTURE);
```

Perl

```
@result = split(m/(<[^<>]*>)/, $subject);
```

Python

Se você tiver somente uma pequena quantidade de strings para dividir, use a função global:

```
result = re.split("(<[^<>]*>)", subject))
```

Para usar a mesma expressão regular repetidamente, use um objeto compilado:

```
reobj = re.compile("(<[^<>]*>)")
result = reobj.split(subject)
```

Ruby

```
list = []
lastindex = 0;
subject.scan(/<[^<>]*>/) {|match|
  list << subject[lastindex..$~.begin(0)-1];
  list << $&
  lastindex = $~.end(0)
}
list << subject[lastindex..subject.length()]
```

Discussão

.NET

No .NET, o método `Regex.Split()` inclui, no array, o texto correspondido pelos grupos de captura. As versões 1.0 e 1.1 do .NET incluem apenas o primeiro grupo de captura. O .NET 2.0 e posteriores incluem no array todos os grupos de captura como strings distintas. Se quiser incluir a correspondência total da expressão regular no array, coloque toda a expressão regular dentro de um grupo de captura. No caso do .NET 2.0 e posteriores, todos os outros grupos

devem ser grupos de não-captura, ou eles serão incluídos no array.

Os grupos de captura não são incluídos na contagem de string que você pode passar à função Split(). Ao chamar regexObj.Split(subject, 4) com a string de exemplo e a expressão regular desta receita, você vai obter um array com sete strings. Estas serão as quatro strings com o texto antes, entre e após as três primeiras correspondências da expressão regular, mais três strings entre elas com as correspondências da expressão regular, conforme capturado pelo único grupo de captura da expressão regular. Em outras palavras, você terá um array com: I□like□, , bold, , □ and□, e italic□fonts. Se a sua expressão regular tiver 10 grupos de captura e você estiver usando .NET 2.0, ou versão posterior, regexObj.Split(subject, 4) retornará um array com 34 strings.

O .NET não fornece uma opção para excluir os grupos de captura do array. Sua única solução seria substituir todos os grupos nomeados e numerados por grupos de não-captura. Uma maneira fácil de fazer isso, no .NET, é usar RegexOptions.ExplicitCapture e substituir todos os grupos nomeados por grupos normais (ou seja, apenas um par de parênteses) em sua expressão regular.

Java

O método Java Pattern.split() não fornece a opção de adicionar a correspondência da expressão regular ao array resultante. Em vez disso, podemos adaptar a receita 3.12, para adicionar a uma lista o texto entre as correspondências da expressão regular junto com suas próprias correspondências. Para obter o texto entre as correspondências, usamos os detalhes explicados na receita 3.8.

JavaScript

A função string.split() não fornece opção para controlar se as correspondências da expressão regular devem ser adicionadas ao array. De acordo com o padrão JavaScript, todos os grupos de captura devem ter suas correspondências adicionadas ao array. Infelizmente, os navegadores atuais não fazem isso, ou o fazem

inconsistentemente.

Para obter uma solução que funcione com todos os navegadores, podemos adaptar a receita 3.12, para adicionar a uma lista o texto entre as correspondências da expressão regular junto com as próprias correspondências da expressão regular. Para obter o texto entre as correspondências, usamos os detalhes explicados na receita 3.8.

PHP

Passe PREG_SPLIT_DELIM_CAPTURE como o quarto parâmetro de preg_split(), para incluir o texto correspondido pelos grupos de captura no array retornado. Você pode usar o operador | para combinar PREG_SPLIT_DELIM_CAPTURE com PREG_SPLIT_NO_EMPTY.

Os grupos de captura não são incluídos na contagem de strings especificada como terceiro argumento da função preg_split(). Se definir o limite para quatro, usando a string de exemplo e a expressão regular desta receita, você obterá um array com sete strings. Estas serão as quatro strings com o texto antes, entre e após as primeiras três correspondências da expressão regular, além de três strings entre elas com as correspondências da expressão regular, conforme capturado pelo único grupo de captura da expressão regular. Em outras palavras, você obterá um array com: I□like□, , bold, , □ and□, e italic□fonts.

Perl

A função Perl split() inclui o texto correspondido por todos os grupos de captura no array. Se quiser incluir a correspondência total da expressão regular no array, coloque-a, inteira, dentro de um grupo de captura.

Os grupos de captura não estão incluídos na contagem de strings que você pode passar para a função split(). Ao chamar split(/(<[^<>]*>)/, $subject, 4) com a string de exemplo e a expressão regular desta receita, você obterá um array com sete strings. Estas serão as

quatro strings com o texto antes, entre e após as três primeiras correspondências da expressão regular, mais três strings entre elas com as correspondências da expressão regular, conforme capturado pelo único grupo de captura da expressão regular. Em outras palavras, você vai obter um array com: I□like□, , bold, , □ and□, e italic□fonts. Se a sua expressão regular tiver 10 grupos de captura, split($regex, $subject, 4) retornará um array com 34 strings.

O Perl não fornece uma opção para excluir os grupos de captura do array. A única solução seria substituir todos os grupos de captura nomeados e numerados por grupos de não-captura.

Python

A funçãoPython split() inclui no array o texto correspondido por todos os grupos de captura. Se quiser incluir a correspondência total da expressão regular no array, coloque toda a expressão regular dentro de um grupo de captura.

Os grupos de captura não afetam o número de divisões aplicadas à string. Ao chamar split(/(<[^<>]*>)/, $subject, 3) com a string de exemplo e a expressão regular desta receita, você obterá um array com sete strings. A string é dividida três vezes, resultando em quatro fragmentos de texto entre as correspondências, além de três fragmentos de texto correspondidos pelos grupos de captura. Em outras palavras, você vai obter um array com: "I like", "", "bold", "", " and ", "" e "italic fonts". Se a sua expressão regular tiver 10 grupos de captura, split($regex, $subject, 3) retornará um array com 34 strings.

O Python não fornece uma opção para excluir os grupos de captura do array. A única solução seria substituir todos os grupos de captura nomeados e numerados por grupos de não-captura.

Ruby

O método Ruby String.split() não fornece opção para adicionar as correspondências da expressão regular ao array resultante. Em vez

disso, podemos adaptar a receita 3.11, para adicionar a uma lista o texto entre as correspondências da expressão regular junto com as próprias correspondências da expressão regular. Para obter o texto entre as correspondências, usamos os detalhes explicados na receita 3.8.

Veja também:

A receita 2.9 explica os grupos de captura e de não-captura.

A receita 2.11 explica os grupos nomeados.

3.21 Pesquisar linha por linha

Problema

Ferramentas grep tradicionais aplicam a expressão regular em uma linha de texto por vez, exibindo as linhas correspondidas (ou não) pela expressão regular. Você possui um array de strings, ou uma string multilinha, que deseja processar dessa forma.

Solução

C#

Se você possui uma string multilinha, primeiramente a divida em um array de strings, com cada string contendo uma linha de texto:

```
string[] lines = Regex.Split(subjectString, "\r?\n");
```

Então, itere o array lines:

```
Regex regexObj = new Regex("regex pattern");
for (int i = 0; i < lines.Length; i++) {
  if (regexObj.IsMatch(lines[i])) {
    // A regex corresponde a lines[i]
  } else {
    // A regex não corresponde a lines[i]
  }
}
```

VB.NET

Se você possui uma string multilinha, primeiramente a divida em um array de strings, com cada string contendo uma linha de texto:

```
Dim Lines = Regex.Split(SubjectString, "\r?\n")
```

Então, itere o array lines:

```
Dim RegexObj As New Regex("regex pattern")
For i As Integer = 0 To Lines.Length - 1
  If RegexObj.IsMatch(Lines(i)) Then
    'A regex corresponde a lines(i)
  Else
    'A regex não corresponde a lines(i)
  End If
Next
```

Java

Se você possui uma string multilinha, primeiramente a divida em um array de strings, com cada string contendo uma linha de texto:

```
String[] lines = subjectString.split("\r?\n");
```

Então, itere o array lines:

```
Pattern regex = Pattern.compile("regex pattern");
Matcher regexMatcher = regex.matcher("");
for (int i = 0; i < lines.length; i++) {
  regexMatcher.reset(lines[i]);
  if (regexMatcher.find()) {
    // A regex corresponde a lines[i]
  } else {
    // A regex não corresponde a lines[i]
  }
}
```

JavaScript

Se você possui uma string multilinha, primeiramente a divida em um array de strings, com cada string contendo uma linha de texto. Como mencionado na receita 3.19, alguns navegadores excluem as linhas em branco do array.

```
var lines = subject.split(/\r?\n/);
```

Então, itere o array lines:

```
var regexp = /regex pattern/;
```

```
for (var i = 0; i < lines.length; i++) {
  if (lines[i].match(regexp)) {
    // A regex corresponde a lines[i]
  } else {
    // A regex não corresponde a lines[i]
  }
}
```

PHP

Se você possui uma string multilinha, primeiramente a divida em um array de strings, com cada string contendo uma linha de texto:

```
$lines = preg_split('/\r?\n/', $subject)
```

Então, itere o array lines:

```
foreach ($lines as $line) {
  if (preg_match('/regex pattern/', $line)) {
    // A regex corresponde a $line
  } else {
    // A regex não corresponde a $line
  }
}
```

Perl

Se você possui uma string multilinha, primeiramente a divida em um array de strings, com cada string contendo uma linha de texto:

```
@lines = split(m/\r?\n/, $subject)
```

Então, itere o array lines:

```
foreach $line (@lines) {
  if ($line =~ m/regex pattern/) {
    # A regex corresponde a $line
  } else {
    # A regex não corresponde a $line
  }
}
```

Python

Se você possui uma string multilinha, primeiramente a divida em um array de strings, com cada string contendo uma linha de texto:

```
lines = re.split("\r?\n", subject);
```

Então, itere o array lines:

```
reobj = re.compile("regex pattern")
for line in lines[:]:
  if reobj.search(line):
    # A regex corresponde a line
  else:
    # A regex não corresponde a line
```

Ruby

Se você possui uma string multilinha, primeiramente a divida em um array de strings, com cada string contendo uma linha de texto:

```
lines = subject.split(/\r?\n/)
```

Então, itere o array lines:

```
re = /regex pattern/
lines.each { |line|
  if line =~ re
    # A regex corresponde a line
  Else
    # A regex não corresponde a line
}
```

Discussão

Ao trabalhar com dados baseados em linhas, você pode se livrar de muita dor de cabeça, caso os divida em um array de linhas, em vez de tentar trabalhar com uma linha longa, com quebras de linha incorporadas. Assim, você poderá aplicar sua expressão regular em cada string no array, sem se preocupar em corresponder a mais de uma linha. Essa abordagem também ajuda a controlar o relacionamento entre as linhas. Por exemplo, você poderia, facilmente, iterar o array usando uma expressão regular para encontrar uma linha de cabeçalho, e outra para encontrar a linha de rodapé. Com as linhas de delimitação encontradas, você poderia usar uma terceira expressão regular para encontrar as linhas de dados que lhe interessam. Embora possa parecer muito trabalhoso, é tudo muito simples, e resultará em um código que funciona bem.

Tentar criar uma única expressão regular para encontrar o cabeçalho, os dados e o rodapé, de uma só vez, será muito mais complicado e resultará em uma expressão regular muito mais lenta.

O processamento de uma string, linha por linha, também facilita negar uma expressão regular. As expressões regulares não nos fornecem uma maneira fácil de dizer “corresponda a uma linha que não contenha essa ou aquela palavra”. Apenas as classes de caracteres podem ser facilmente negadas. Porém, se você já dividiu sua string em linhas, encontrar as que não contêm uma palavra torna-se tão fácil quanto fazer uma busca de texto literal em todas as linhas, removendo aquelas em que a palavra poderia ser encontrada.

A receita 3.19 mostra como você pode, facilmente, dividir uma string em um array. A expressão regular ‹\r\n› corresponde a um par de caracteres **CR** e **LF**, que delimitam as linhas nas plataformas Microsoft Windows. ‹\n› corresponde a um caractere **LF**, que delimita as linhas em Unix e seus derivados, como o Linux e até o Mac OS X. Como estas duas expressões regulares são, essencialmente, texto simples, você nem precisa usar uma expressão regular. Se a sua linguagem de programação puder dividir strings usando o texto literal, divida-as dessa forma.

Se você não estiver certo sobre qual estilo de quebra de linha seus dados usam, você poderia dividi-los, usando a expressão regular ‹\r?\n›. Ao tornar **CR** opcional, esta expressão regular corresponde a uma quebra de linha Windows **CRLF**, ou a uma quebra de linha Unix **LF**.

Com suas strings no array, você poderá iterá-lo facilmente. Dentro do loop, siga as dicas mostradas na receita 3.5 para verificar quais linhas correspondem, e quais não.

Veja também:

Receitas 3.11 e 3.19.

1 Alguns poucos sabores modernos de expressão regular têm tentado introduzir funcionalidades de correspondência balanceada ou recursiva. No entanto, tais funcionalidades resultam em expressões regulares tão complexas que só acabam por demonstrar nosso argumento de que é melhor deixar a análise para o código procedural.

2 Para permitir que a tag abranja várias linhas, ative o modo “ponto corresponde a quebras de linha”. No caso do JavaScript, utilize ‹([\s\S]*?)›.

CAPÍTULO 4

Validação e formatação

Este capítulo contém receitas para validação e formatação de tipos comuns de entrada de usuário. Algumas das soluções mostram como permitir variações de entradas válidas, como códigos postais (norte-americanos), que podem conter de cinco a nove dígitos. Outras são projetadas para harmonizar ou corrigir os formatos comumente entendidos para elementos como números de telefone, datas e números de cartão de crédito.

Além de ajudar você a concluir o trabalho, eliminando entradas inválidas, essas receitas também podem melhorar a experiência dos usuários de seus aplicativos. Mensagens como “sem espaços ou hífens” ao lado dos campos de telefone, ou do número de cartão de crédito, frequentemente frustram os usuários, ou simplesmente são ignoradas. Felizmente, em muitos casos, as expressões regulares permitem que os usuários insiram seus dados em formatos que eles considerem familiares e confortáveis, com pouco trabalho extra de sua parte.

Certas linguagens de programação fornecem funcionalidades semelhantes a algumas receitas, por meio das suas classes nativas ou bibliotecas. Dependendo de suas necessidades, pode fazer mais sentido utilizar as opções nativas; por isso, iremos apontá-las ao longo do caminho.

4.1 Validar endereços de e-mail

Problema

Você possui um formulário em seu site, ou uma caixa de diálogo em

seu aplicativo, que pede um endereço de e-mail ao usuário. Você deseja usar uma expressão regular para validar este endereço de e-mail, antes de tentar enviar uma mensagem. Isso reduz o número de e-mails não entregues, e devolvidos a você.

Solução

Simples

A primeira solução executa uma verificação muito simples. Ela apenas verifica se o endereço de e-mail possui um sinal de @ e se não possui espaços em branco:

```
^\S+@\S+$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A\S+@\S+\Z
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Simples, com restrição de caracteres

O *nome de domínio*, a parte depois do sinal de @, é restrita aos caracteres permitidos em nomes de domínio. O *nome do usuário*, a parte antes do sinal de @, é restrita aos caracteres comumente usados em nomes de usuários de e-mail, mais restritiva do que aquilo que é aceito pela maioria dos clientes e servidores de e-mail:

```
^[A-Z0-9+_.-]+@[A-Z0-9.-]+$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A[A-Z0-9+_.-]+@[A-Z0-9.-]+\Z
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Simples, com todos os caracteres

Esta expressão regular amplia a anterior, permitindo um conjunto maior de caracteres, raramente usados em nomes de usuário. Nem

todos os softwares de e-mail conseguem tratar esses caracteres, mas incluímos todos os caracteres permitidos pela RFC 2822, que regulamenta o formato de mensagens de correio eletrônico. Entre os caracteres permitidos, estão alguns que apresentarão risco de segurança, caso passem diretamente da entrada de usuário para uma instrução SQL, como a aspa simples (') e o caractere pipe (|). Certifique-se de escapar certos caracteres críticos ao inserir o endereço de e-mail em uma string passada para outro programa, a fim de evitar falhas de segurança, tais como ataques de injeção SQL:

```
^[\w!#$%&'*+/=?`{|}~^.-]+@[A-Z0-9.-]+$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A[\w!#$%&'*+/=?`{|}~^.-]+@[A-Z0-9.-]+\Z
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Sem ponto inicial, final ou pontos consecutivos

Tanto o nome de usuário, quanto o de domínio podem conter um ou mais pontos, mas dois pontos não podem aparecer um ao lado do outro. Além disso, o primeiro e o último caractere do nome de usuário e do nome de domínio não podem ser pontos:

```
^[\w!#$%&'*+/=?`{|}~^-]+(?:\.[\w!#$%&'*+/=?`{|}~^-]+)*@↲
[A-Z0-9-]+(?:\.[A-Z0-9-]+)*$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A[\w!#$%&'*+/=?`{|}~^-]+(?:\.[\w!#$%&'*+/=?`{|}~^-]+)*@↲
[A-Z0-9-]+(?:\.[A-Z0-9-]+)*\Z
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Domínio de nível mais alto possui de duas a seis letras

Esta expressão regular amplia as versões anteriores, especificando que o nome de domínio deve incluir pelo menos um ponto, e que a parte do nome de domínio após o ponto poderá conter apenas letras. Isto é, o domínio deve conter pelo menos dois níveis, como secondlevel.com ou thirdlevel.secondlevel.com. O domínio de nível mais alto, .com, deve ser composto de duas a seis letras. Todos os domínios de nível mais alto, relacionados a códigos de país, possuem duas letras. Os domínios de nível mais alto genéricos têm entre três (.com) e seis letras (.museum):

```
^[\w!#$%&'*+/=?`{|}~^-]+(?:\.[\w!#$%&'*+/=?`{|}~^-]+)*@↵
(?:[A-Z0-9-]+\.)+[A-Z]{2,6}$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A[\w!#$%&'*+/=?`{|}~^-]+(?:\.[\w!#$%&'*+/=?`{|}~^-]+)*@↵
(?:[A-Z0-9-]+\.)+[A-Z]{2,6}\Z
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Discussão

Sobre endereços de e-mail

Se você pensou que algo conceitualmente tão simples, como validar um endereço de e-mail, teria uma solução de expressão regular simples e direta, está completamente errado. Esta receita é um excelente exemplo de que, antes de começar a escrever uma expressão regular, você precisa decidir exatamente ao que deseja corresponder. Não há um acordo universal sobre quais endereços de e-mail são válidos e quais não são. Depende da sua definição de válido.

asdf@asdf.asdf é válido de acordo com a RFC 2822, que define a sintaxe para endereços de e-mail. Mas não é válido se a sua definição especifica de endereço válido de e-mail diga respeito ao tipo que aceite mensagens. Não há domínio top-level asdf.

A resposta mais curta para o problema de validade é que você não

pode saber se john.doe@somewhere.com é um endereço válido, até que tente enviar-lhe um e-mail. E, mesmo assim, você não pode ter certeza se a falta de resposta significa que o domínio somewhere.com está silenciosamente descartando os e-mails enviados para caixas postais inexistentes, ou se John Doe apertou o botão Delete em seu teclado, ou se está com o filtro de spam ativado.

Como, em última instância, você terá de verificar se realmente existe tal endereço para poder enviar o e-mail, poderá decidir usar uma expressão regular mais simples, ou menos rígida. Permitir que endereços inválidos sejam aceitos pode ser melhor do que irritar as pessoas, ao bloquear endereços válidos. Por esta razão, é possível que você queira selecionar a expressão regular “simples, com todos os caracteres”. Embora ela, obviamente, permita outras coisas que não sejam endereços de e-mail, como #$%@.-; a expressão regular é rápida e simples, e nunca bloqueará um endereço de e-mail válido.

Se quiser evitar o envio de e-mails que não possam ser entregues, sem bloquear endereços de e-mail válidos, a expressão regular “Domínio de nível mais alto possui de duas a seis letras”, é uma boa escolha.

Você precisa considerar o nível de complexidade da sua expressão regular. Se estiver validando uma entrada de usuário, é provável que queira uma expressão regular mais complexa, pois o usuário pode digitar qualquer coisa. Mas, se estiver digitalizando arquivos de bancos de dados que contenham apenas endereços de e-mail válidos, é possível usar uma expressão regular muito simples, que apenas separe os endereços de e-mail dos outros dados. Até a solução na subseção anterior pode ser suficiente, nesse caso.

Finalmente, você precisa considerar se sua expressão regular resistirá ao futuro. No passado, fazia sentido restringir o domínio de nível mais alto a apenas combinações de duas letras para os códigos de país, listando exaustivamente os domínios de nível mais alto genéricos, ou seja, ‹com|net|org|mil|edu›. Com novos domínios de nível mais alto sendo adicionados o tempo todo, tais expressões

regulares podem se desatualizar rapidamente.

Sintaxe da Expressão Regular

As expressões regulares apresentadas nesta receita mostram em ação todas as partes básicas da sintaxe de uma expressão regular. Se você já leu a respeito disso no capítulo 2, poderá realizar 90% daqueles trabalhos mais bem resolvidos por meio de expressões regulares.

Todas as expressões regulares exigem que a opção de não-diferenciação entre maiúsculas e minúsculas esteja ativada. Caso contrário, somente caracteres maiúsculos serão permitidos. Ativar esta opção permite que você digite ‹[A-Z]›, em vez de ‹[A-Za-z]›, economizando na digitação. Se você utilizar uma das duas últimas expressões regulares, a opção de não-diferenciação entre maiúsculas e minúsculas será muito útil. Caso contrário, você teria de substituir cada letra ‹X› por ‹[Xx]›.

‹\S› e ‹\w› são classes de caracteres abreviadas, como explica a receita 2.3. ‹\S› corresponde a qualquer caractere que não seja um espaço em branco, enquanto ‹\w› corresponde a um caractere de palavra.

‹@› e ‹\.› correspondem a um sinal literal de @ e a um ponto, respectivamente. Como o ponto é um metacaractere quando utilizado fora de classes de caracteres, ele precisa ser escapado por uma barra invertida. O sinal @ não tem um significado especial em qualquer um dos sabores de expressão regular deste livro. A receita 2.1 fornece uma lista de todos os metacaracteres que precisam ser escapados.

‹[A-Z0-9.-]› e as outras strings entre colchetes são classes de caracteres. Esta classe permite todas as letras de A a Z, todos os dígitos entre 0 e 9, bem como um ponto literal e o hífen. Embora o hífen normalmente crie um intervalo em uma classe de caracteres, ele é tratado como um literal, quando ocorre como o último caractere em uma classe de caracteres.

A receita 2.3 apresenta tudo sobre as classes de caracteres, inclusive as combinando-as com abreviações, como em ‹[\w!#$%&'*+/=?`{|}~^.-]›. Esta classe corresponde a um caractere de palavra, assim como a qualquer um dos 19 caracteres de pontuação listados.

‹+› e ‹*›, quando utilizados fora das classes de caracteres, são quantificadores. O sinal de adição repete uma ou mais vezes o token de expressão regular anterior, enquanto o asterisco repete-o, zero ou mais vezes. Nessas expressões regulares, o token quantificado é geralmente uma classe de caracteres e, às vezes, um grupo. Portanto, ‹[A-Z0-9.-]+› corresponde a uma ou mais letras, dígitos, pontos ou hífens.

Como exemplo do uso de um grupo, ‹(?:[A-Z0-9-]+\.)+› corresponde a uma ou mais letras, dígitos ou hífens, seguido por um ponto literal. O sinal de adição repete este grupo uma ou mais vezes. O grupo deve corresponder pelo menos uma vez, mas pode corresponder tanto quanto possível. A receita 2.12 explica em detalhes a mecânica de construções como esta.

‹(?:group)› é um grupo de não-captura. Use-o para criar um grupo a partir de uma parte da expressão regular, a fim de aplicar um quantificador ao grupo como um todo. O grupo de captura ‹(group)› faz a mesma coisa com uma sintaxe mais limpa, de modo que você poderia substituir ‹(?:› por ‹(› em todas as expressões regulares que utilizamos até agora, sem alterar o resultado geral da correspondência.

Porém, como não estamos interessados em capturar separadamente as partes do endereço de e-mail, o grupo de não-captura é um pouco mais eficiente, embora torne a expressão regular um pouco mais difícil de ler. A receita 2.9 nos apresenta tudo sobre os grupos de captura e de não-captura.

As âncoras ‹^› e ‹$› forçam a expressão regular a encontrar a correspondência no início e no final do texto de assunto, respectivamente. Colocar toda a expressão regular entre esses

caracteres efetivamente obriga a expressão a corresponder a todo o assunto.

Isso é importante ao validar a entrada de usuário. Você não vai querer aceitar drop database; -- joe@server.com haha! como um endereço válido de e-mail. Sem as âncoras, todas as expressões regulares anteriores corresponderão, pois elas encontram joe@server.com no meio do texto. Veja a receita 2.5 para mais detalhes. Essa receita também explica por qual motivo a opção de correspondência "circunflexo e cifrão correspondem em quebras de linha" deve estar desativada.

Em Ruby, o acento circunflexo e o cifrão sempre correspondem em quebras de linha. As expressões regulares que utilizam o acento circunflexo e o cifrão funcionam corretamente no Ruby, mas somente se a string que você estiver tentando validar não contiver quebras de linha. Se a string puder conter quebras de linha, todas as expressões regulares utilizando ‹^› e ‹$› corresponderão ao endereço de e-mail em drop database; -- **LF** joe@server.com **LF** haha!, em que **LF** representa uma quebra de linha.

Para evitar isso, utilize as âncoras ‹\A› e ‹\Z›. Elas correspondem apenas no início e no final da string, independentemente de qualquer opção, em qualquer sabor, discutidos neste livro, exceto o JavaScript. O JavaScript não suporta ‹\A› e ‹\Z›. A receita 2.5 explica essas âncoras.

A questão de ‹^› e ‹$› contra ‹\A› e ‹\Z› aplica-se a todas as expressões regulares que validem entradas. Há uma grande quantidade delas neste livro. Embora façamos o lembrete habitual, não vamos repetir constantemente este conselho, ou mostrar soluções distintas para JavaScript e Ruby a cada receita. Em muitos casos, mostraremos apenas uma solução, usando o acento circunflexo e o cifrão, listando o Ruby como um sabor compatível. Se estiver usando o Ruby, lembre-se de usar ‹\A› e ‹\Z›, caso queira evitar corresponder a uma linha, em uma string com múltiplas linhas.

Construindo uma expressão regular passo a passo

Esta receita ilustra como você pode construir uma expressão regular passo a passo. Esta técnica é particularmente útil se empregada

com um testador de expressão regular interativo, como o RegexBuddy.

Primeiro, carregue um punhado de dados de amostra válidos e inválidos na ferramenta. Neste caso, eles seriam uma lista de endereços de e-mail válidos e uma lista de endereços inválidos.

Em seguida, escreva uma expressão regular simples, que corresponda a todos os endereços válidos de e-mail. Ignore os endereços inválidos, por enquanto. ‹^\S+@\S+$› já define a estrutura básica de um endereço de e-mail: nome de usuário, sinal de arroba e nome de domínio.

Com a estrutura básica do seu padrão de texto definida, você poderá refinar cada parte, até que sua expressão regular não mais corresponda a qualquer um dos dados inválidos. Se sua expressão regular tiver que trabalhar somente com dados previamente existentes, esta poderá ser uma tarefa rápida. Se a sua expressão regular tiver que trabalhar com qualquer entrada de usuário, a tarefa de editá-la, até que ela seja suficientemente restritiva, será muito mais difícil do que fazê-la corresponder apenas aos dados válidos.

Variações

Você não poderá utilizar as âncoras ‹^› e ‹$›, se quiser pesquisar endereços de e-mail em corpos de textos maiores, ao invés de checar se a entrada, como um todo, é um endereço de e-mail. Porém, simplesmente remover as âncoras da expressão regular não é a solução correta. Se fizer isso com a expressão regular final, que restringe o domínio de nível mais alto apenas a letras, ela corresponderá a asdf@asdf.as em asdf@asdf.as99, por exemplo. Em vez de ancorar a correspondência da expressão regular no início e no final do assunto, você terá de especificar que o início do nome de usuário e o domínio de nível mais alto não podem fazer parte de palavras mais longas.

Isto é feito facilmente com um par de extremidades de palavras. Substitua ‹^› e ‹$› por ‹\b›. Por exemplo, ‹^[A-Z0-9+_.-]+@(?:[A-Z0-9-]+\.)+

[A-Z]{2,6}$› torna-se ‹\b[A-Z0-9+_.-]+@(?:[A-Z0-9-]+\.)+[A-Z]{2,6}\b›.

Esta expressão regular combina, de fato, a porção do nome de usuário do subtópico “Simples, com restrição de caracteres”, e a porção do nome do domínio do subtópico “Domínio de nível mais alto possui de duas a seis letras”. Em nossa opinião, esta expressão regular funciona muito bem na prática.

Veja também:

A RFC 2822 define a estrutura e a sintaxe das mensagens de e-mail, incluindo os endereços de e-mail utilizados em mensagens de correio eletrônico. Você pode fazer o download da RFC 2822 em *http://www.ietf.org/rfc/rfc2822.txt.*

4.2 Validar e formatar números de telefone norte-americanos

Problema

Você deseja determinar se um usuário entrou com um número de telefone norte-americano em um formato comum, incluindo o código de área local. Esses formatos incluem 1234567890, 123-456-7890, 123.456.7890, 123 456 7890, (123) 456 7890 e todas as combinações relacionadas. Se o número de telefone for válido, você deseja convertê-lo para o formato padrão, (123) 456-7890, de modo que seus registros fiquem consistentes.

Solução

Uma expressão regular pode facilmente verificar se um usuário entrou com alguma coisa que se pareça com um número válido de telefone. Usando grupos de captura para lembrar cada conjunto de dígitos, a mesma expressão regular pode ser usada para substituir o texto de assunto, exatamente com o formato desejado.

Expressão Regular

```
^\(?([0-9]{3})\)?[-. ]?([0-9]{3})[-. ]?([0-9]{4})$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Substituição

```
($1) $2-$3
```

Sabores de texto de substituição: .NET, Java, JavaScript, Perl, PHP

```
(\1) \2-\3
```

Sabores de texto de substituição: Python, Ruby

C#

```
Regex regexObj = new Regex(@"^\(?([0-9]{3})\)?[-. ]?([0-9]{3})[-. ]?([0-9]{4})$");
if (regexObj.IsMatch(subjectString)) {
  string formattedPhoneNumber = regexObj.Replace(subjectString, "($1) $2-$3");
} else {
  // Número de telefone inválido
}
```

JavaScript

```
var regexObj = /^\(?([0-9]{3})\)?[-. ]?([0-9]{3})[-. ]?([0-9]{4})$/;
if (regexObj.test(subjectString)) {
  var formattedPhoneNumber = subjectString.replace(regexObj, "($1) $2-$3");
} else {
  // Número de telefone inválido
}
```

Outras linguagens de programação

Veja as receitas de 3.5 e 3.15, para ajudá-lo na implementação desta expressão regular com outras linguagens de programação.

Discussão

Esta expressão regular corresponde a três grupos de dígitos. O primeiro grupo pode, opcionalmente, ser colocado entre parênteses, e os dois primeiros grupos podem, opcionalmente, serem seguidos por uma escolha de três separadores (um hífen, um ponto ou um espaço). O layout a seguir quebra a expressão regular em suas

partes individuais, omitindo os grupos de dígitos redundantes:

```
^ # Declara a posição no início da string
\( # Corresponde a um literal "("...
  ? # entre zero e uma vez.
( # Captura a correspondência entre parênteses na retrorreferência 1...
  [0-9] # Corresponde a um dígito...
   {3} # exatamente três vezes.
) # Finaliza o grupo de captura 1.
\) # Corresponde a um literal ")"...
  ? # entre zero e uma vez.
[-. ] # Corresponde a um caractere do conjunto "-. "...
  ? # entre zero e uma vez.
... # [Corresponde aos dígitos remanescentes e ao separador.]
$ # Declara a posição no final da string.
```

Vamos olhar mais de perto cada uma dessas partes.

‹^› e ‹$›, no início e no final da expressão regular, formam um tipo especial de metacaractere, chamado de *âncora* ou *declaração (assertion)*. Em vez de corresponder ao texto, as declarações correspondem a uma posição dentro do texto. Especificamente, ‹^› corresponde no início do texto, e ‹$› no final. Isso garante que a expressão regular dos números de telefone não corresponderá dentro de textos mais longos, tal como `123-456-78901`.

Como temos visto repetidamente parênteses são caracteres especiais nas expressões regulares, mas, neste caso, queremos permitir que um usuário entre com parênteses e que nossa expressão regular os reconheça. Este é um exemplo básico em que precisamos de uma barra invertida como caractere especial de escape, para que a expressão regular trate o caractere como entrada literal. Assim, as sequências ‹\(› e ‹\)›, que colocam o primeiro grupo de dígitos entre parênteses, correspondem a caracteres de parênteses literais. Ambos são seguidos por um ponto de interrogação, tornando-os opcionais. Explicaremos mais sobre o ponto de interrogação, após discutirmos os outros tipos de tokens da expressão regular.

Os parênteses que aparecem sem barras invertidas são grupos de captura, e são usados para lembrar os valores correspondidos

dentro deles, para que o texto correspondido possa ser recuperado mais tarde. Neste caso, são utilizadas retrorreferências aos valores capturados no texto de substituição, para que possamos facilmente reformatar o número de telefone, quando necessário.

Dois outros tipos de tokens, utilizados nesta expressão regular, são as classes de caracteres e os quantificadores. Classes de caracteres nos permitem corresponder a qualquer caractere dentre um conjunto de caracteres. ‹[0-9]› é uma classe de caracteres que corresponde a qualquer dígito. Os sabores de expressões regulares cobertos por este livro incluem a classe de caracteres abreviada ‹\d›, que também corresponde a um dígito, mas, em alguns sabores, ‹\d› corresponde a um dígito de qualquer conjunto ou sistema de escrita de qualquer idioma, o que não queremos aqui. Veja a receita 2.3 para obter mais informações sobre ‹\d›.

‹[-.□]› é uma outra classe de caracteres, que permite qualquer um dos três separadores. É importante que o hífen apareça em primeiro lugar nesta classe de caracteres, pois, se aparecesse entre outros caracteres, ele acabaria criando um intervalo, como acontece com ‹[0-9]›. Outra maneira de garantir que um hífen corresponda a uma versão literal de si mesmo, dentro de uma classe de caracteres, seria utilizando uma barra invertida como caractere de escape. ‹[.\-□]› é, portanto, equivalente.

Finalmente, quantificadores permitem que você repita um símbolo ou grupo. ‹{3}› é um quantificador que faz com que o elemento anterior a ele seja repetido exatamente três vezes. A expressão regular ‹[0-9]{3}› é, portanto, equivalente a ‹[0-9][0-9][0-9]›, mas é mais curta e mais fácil de ler. O ponto de interrogação (mencionado anteriormente) é um quantificador especial que faz o elemento a ele precedente ser repetido zero ou uma vez. Também poderia ser escrito como ‹{0,1}›. Qualquer quantificador que permita a algo ser repetido zero vez, efetivamente, torna esse elemento opcional. Como um ponto de interrogação é utilizado depois de cada separador, os dígitos do número de telefone podem ficar juntos.

Observe que, embora esta receita reivindique lidar com números de telefone norte-americanos, na verdade ela é projetada para trabalhar com números do Plano de Numeração da América do Norte (NANP). NANP é o plano de numeração de telefones para países que compartilham o código de país “1”. Isso inclui os Estados Unidos e seus territórios, Canadá, Bermudas e as 16 nações do Caribe. Exclui o México e os países da América Central.

Variações

Eliminar números de telefone inválidos

Até agora, a expressão regular corresponde a qualquer um dos 10 dígitos. Se quiser limitar as correspondências, para validar números de telefone de acordo com o Plano de Numeração Norte Americano, veja as regras básicas:

- *Códigos de área* começam com um número entre 2 e 9, seguido por 0 e 8 e, então, qualquer terceiro dígito.
- O segundo grupo de três dígitos, conhecido como *central office (escritório central)*, ou *exchange code (código de troca)*, começa com um número entre 2 e 9, seguido por quaisquer dois dígitos.
- Os quatro dígitos finais, conhecidos como *station code (código de estação)*, não têm restrições.

Essas regras podem ser facilmente implementadas com algumas classes de caracteres.

```
^\(?([2-9][0-8][0-9])\)?[-. ]?([2-9][0-9]{2})[-. ]?([0-9]{4})$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Além das regras básicas listadas, há uma variedade de números de telefone reservados, não atribuídos e restritos. A menos que você tenha necessidades muito específicas, que exijam a filtragem da maior quantidade possível de números de telefone, não extrapole tentando eliminar números não utilizados. Novos códigos de área, que se encaixam nas regras listadas anteriormente, estão sendo

disponibilizados regularmente; mesmo se um número de telefone for válido, isso não significa, necessariamente, que foi liberado ou que esteja em uso.

Encontrar números de telefones em documentos

Duas mudanças simples permitirão que a expressão regular anterior corresponda a números de telefone dentro de um texto mais longo:

```
\(?\b([0-9]{3})\)?[-. ]?([0-9]{3})[-. ]?([0-9]{4})\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Aqui, as declarações ‹^› e ‹$›, que limitam a expressão regular ao início e ao final do texto, foram removidas. Em seu lugar, foram adicionados tokens de extremidade de palavra (‹\b›), para garantir que o texto correspondido seja independente, não fazendo parte de um número, ou palavra, maior.

Semelhante a ‹^› e ‹$›, ‹\b› é uma declaração que corresponde a uma posição, ao invés de corresponder ao texto. Especificamente, ‹\b› corresponde à posição entre um caractere de palavra e um caractere que não seja de palavra, ou ao início/final do texto. Letras, números e underscores são todos considerados caracteres de palavras (veja receita 2.6).

Note que o primeiro token de extremidade de palavra aparece após o parêntese opcional de abertura. Isso é importante, pois não há extremidade de palavra a ser correspondida entre dois caracteres que não sejam de palavra, tal como o parêntese de abertura e um caractere de espaço que o preceda. A primeira extremidade de palavra só é relevante quando corresponde a um número sem parênteses, já que a extremidade de palavra sempre corresponde entre o parêntese de abertura e o primeiro dígito de um número de telefone.

Permitir um “1” inicial

Você pode permitir um “1” inicial e opcional para o código do país

(que cubra a região do Plano de Numeração Norte Americano) por meio da inclusão apresentada na seguinte expressão regular:

```
^(?:\+?1[-. ]?)?\(?([0-9]{3})\)?[-. ]?([0-9]{3})[-. ]?([0-9]{4})$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Além dos formatos de números de telefone, mostrados anteriormente, esta expressão regular também corresponderá a strings, como `+1 (123) 456-7890` e `1-123-456-7890`. Ela utiliza um grupo de não-captura, escrito como `‹(?:...)›`. Quando um ponto de interrogação sucede um parêntese esquerdo não escapado, como este, ele não é um quantificador, mas ajuda a identificar o tipo de agrupamento. Grupos de captura padrão exigem que os mecanismos de expressões regulares fiquem de olho nas retrorreferências; por isso, é mais eficiente usar grupos de não-captura, sempre que o texto correspondido por um grupo não precisar ser consultado posteriormente. Outra razão para utilizar um grupo de não-captura é o fato de que você poderá continuar usando a mesma string de substituição, como nos exemplos anteriores. Se adicionássemos um grupo de captura, teríamos que mudar `$1` para `$2` (e assim por diante) no texto de substituição apresentado anteriormente nesta receita.

A inclusão completa dessa versão da expressão regular é `‹(?:\+?1[-. ]?)?›`. O “1”, neste padrão, é precedido por um sinal de adição opcional, que pode, ou não, ser seguido por um dos três separadores (hífen, ponto ou espaço). Todo o grupo de não-captura incluído também é opcional, mas como o “1” é obrigatório dentro do grupo, o sinal de adição precedente e o separador não são permitidos, caso não haja um “1” inicial.

Permitir números de telefone de sete dígitos

Para permitir a correspondência de números de telefone que omitam o código de área local, coloque o primeiro grupo de dígitos entre parênteses, seguido de um separador, em um grupo de não-captura

opcional:

```
^(?:\(?([0-9]{3})\)?[-. ]?)?([0-9]{3})[-. ]?([0-9]{4})$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Como o código de área não é mais necessário como parte da correspondência, simplesmente substituir qualquer correspondência por «($1) $2-$3» pode resultar em algo como () 123-4567, com um conjunto vazio de parênteses. Para resolver este problema, adicione um código fora da expressão regular, que verifica se o grupo 1 correspondeu a qualquer texto, e ajuste o texto de substituição de acordo.

Veja também:

A receita 4.3 mostra como validar números de telefone internacionais.

O Plano de Numeração Norte Americano (NANP) é o plano de numeração telefônico para os Estados Unidos e seus territórios, Canadá, Bermudas e as 16 nações do Caribe. Mais informações estão disponíveis em *http://www.nanpa.com*.

4.3 Validar números de telefone internacionais

Problema

Você deseja validar números de telefones internacionais. Os números devem começar com um sinal de adição, seguido pelo código do país e o número local.

Solução

Expressão regular

```
^\+(?:[0-9] ?){6,14}[0-9]$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

JavaScript

```
function validate (phone) {
  var regex = /^\+(?:[0-9] ?){6,14}[0-9]$/;
  if (regex.test(phone)) {
    // Número de telefone internacional válido
  } else {
    // Número de telefone internacional inválido
  }
}
```

Outras linguagens de programação

Veja a receita 3.5, para ajudá-lo na implementação desta expressão regular com outras linguagens de programação.

Discussão

As regras e convenções utilizadas para imprimir números de telefones internacionais variam significativamente em todo o mundo; por isso, é difícil proporcionar uma validação significativa para um número de telefone internacional, a menos que você adote um formato rígido. Felizmente, existe uma notação simples e padronizada, especificada pela ITU-T E.123. Esta notação exige que os números de telefone internacionais incluam um sinal inicial de adição (conhecido como *símbolo de prefixo internacional*), e permite que somente espaços separem os grupos de dígitos. Embora o caractere til (~) possa aparecer dentro de um número de telefone, para indicar a existência de um tom de discagem adicional, ele foi excluído desta expressão regular, por ser apenas um elemento de procedimento (em outras palavras, não é discado de verdade), sendo raramente usado. Graças ao plano de numeração de telefones internacionais (ITU-T E.164), números de telefone não podem conter mais do que 15 dígitos. Os menores números de telefone internacionais em uso têm sete dígitos.

Com tudo isso em mente, vamos olhar para a expressão regular novamente, após desmembrá-la em suas partes. Como esta versão é escrita utilizando o estilo de espaçamento livre, o caractere literal de espaço foi substituído por ‹\x20›:

```
^ # Declara a posição no início da string.
\+ # Corresponde a um caractere literal "+".
(?: # Agrupa, mas não captura...
  [0-9] # Corresponde a um dígito.
  \x20 # Corresponde a um caractere de espaço...
   ? # entre zero e uma vez.
) # Finaliza o grupo de não-captura.
  {6,14} # Repete o grupo anterior entre 6 e 14 vezes.
[0-9] # Corresponde a um dígito.
$ # Declara a posição no final da string.
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

As âncoras ‹^› e ‹$›, nas extremidades da expressão regular, garantem que ela corresponda a todo o texto de assunto. O grupo de não-captura – colocado dentro de ‹(?:...)› – corresponde a um único dígito, seguido por um caractere opcional de espaço. A repetição desse agrupamento com o quantificador de intervalo ‹{6,14}› impõe a regra relativa às quantidades mínima e máxima de dígitos, permitindo que apareçam separadores de espaços em qualquer lugar dentro do número. A segunda instância da classe de caracteres ‹[0-9]› conclui a regra relativa ao número de dígitos (aumentando de 6 a 14 para 7 a 15), e garante que o número de telefone não termine com um espaço.

Variações

Validar números de telefones internacionais no formato EPP

```
^\+[0-9]{1,3}\.[0-9]{4,14}(?:x.+)?$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Esta expressão regular segue a notação de números de telefones internacionais, especificada pelo Extensible Provisioning Protocol

(EPP; algo como Protocolo de Fornecimento Extensível). EPP é um protocolo relativamente recente (finalizado em 2004), projetado para a comunicação entre os registros de nomes de domínio e seus registradores. Ele é usado por um número crescente de registros de nomes de domínio, incluindo *.com*, *.info*, *.net*, *.org* e *.us*. A importância disso é que os números de telefone internacionais no estilo EPP são cada vez mais utilizados e reconhecidos e, portanto, fornecem um bom formato alternativo de armazenamento (e validação) dos números de telefone internacionais.

Números de telefone no estilo EPP utilizam o formato *+CCC.NNNNNNNNNxEEEE*, em que C é o código do país, de 1 a 3 dígitos, *N* tendo até 14 dígitos, e E sendo a extensão (opcional). O sinal de adição inicial e o ponto após o código do país são obrigatórios. O caractere literal “x” é necessário apenas quando uma extensão é fornecida.

Veja também:

A receita 4.2 fornece mais opções para validar números de telefone norte-americanos. A Recomendação ITU-T E.123 (“Notação para números de telefone nacionais e internacionais, endereços de e-mail e endereços da Web”) pode ser baixada aqui: *http://www.itu.int/rec/T-REC-E.123*.

A Recomendação ITU-T E.164 (“O plano de numeração de telecomunicação pública internacional“) pode ser baixado em *http://www.itu.int/rec/T-REC-E.164*.

Planos de numeração nacionais podem ser baixados em *http://www.itu.int/ITU-T/inr/nnp*.

A RFC 4933 define a sintaxe e a semântica dos identificadores de contato EPP, incluindo números de telefone internacionais. Você pode baixar a RFC 4933 em *http://tools.ietf.org/html/rfc4933*.

4.4 Validar formatos de data

tradicionais

Problema

Você deseja validar datas nos formatos tradicionais mm/dd/aa, mm/dd/aaaa, dd/mm/aa e dd/mm/aaaa. E quer, também, usar uma expressão regular simples, que verifica simplesmente se a entrada se parece com uma data, sem tentar eliminar coisas como 31 de fevereiro.

Solução

Corresponda a qualquer um desses formatos de data, permitindo a omissão de zeros iniciais:

```
^[0-3]?[0-9]/[0-3]?[0-9]/(?:[0-9]{2})?[0-9]{2}$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Corresponda a qualquer um desses formatos de data, exigindo os zeros iniciais:

```
^[0-3][0-9]/[0-3][0-9]/(?:[0-9][0-9])?[0-9][0-9]$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Corresponda a m/d/aa e mm/dd/aaaa, permitindo qualquer combinação de um ou dois dígitos para dia e mês, e dois ou quatro dígitos para ano:

```
^(1[0-2]|0?[1-9])/(3[01]|[12][0-9]|0?[1-9])/(?:[0-9]{2})?[0-9]{2}$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Corresponda a mm/dd/aaaa, exigindo os zeros iniciais:

```
^(1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])/[0-9]{4}$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Corresponda a d/m/aa e dd/mm/aaaa, permitindo qualquer correspondência, de um ou dois dígitos para dia e mês, e dois ou

quatro dígitos para o ano:

```
^(3[01]|[12][0-9]|0?[1-9])/(1[0-2]|0?[1-9])/(?:[0-9]{2})?[0-9]{2}$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Corresponda a dd/mm/aaaa, exigindo os zeros iniciais:

```
^(3[01]|[12][0-9]|0[1-9])/(1[0-2]|0[1-9])/[0-9]{4}$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Corresponda a qualquer um desses formatos de data com maior precisão, permitindo que os zeros iniciais sejam omitidos:

```
^(?:(1[0-2]|0?[1-9])/(3[01]|[12][0-9]|0?[1-9])|↲
(3[01]|[12][0-9]|0?[1-9])/(1[0-2]|0?[1-9]))/(?:[0-9]{2})?[0-9]{2}$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Corresponda a qualquer um desses formatos de data com maior precisão, exigindo os zeros iniciais:

```
^(?:(1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])|↲
(3[01]|[12][0-9]|0[1-9])/(1[0-2]|0[1-9]))/[0-9]{4}$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

A opção de espaçamento livre torna as duas últimas expressões regulares um pouco mais legíveis:

```
^(?:
  # m/d ou mm/dd
  (1[0-2]|0?[1-9])/(3[01]|[12][0-9]|0?[1-9])
|
  # d/m ou dd/mm
  (3[01]|[12][0-9]|0?[1-9])/(1[0-2]|0?[1-9])
)
  # /yy ou /yyyy
  /(?:[0-9]{2})?[0-9]{2}$
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:
```

```
  # mm/dd
  (1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])
|
  # dd/mm
  (3[01]|[12][0-9]|0[1-9])/(1[0-2]|0[1-9])
)
# /yyyy
/[0-9]{4}$
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Discussão

Você poderia pensar que algo conceitualmente tão trivial como uma data deveria ser tarefa fácil para uma expressão regular. Mas não é, por duas razões. Como as datas são algo cotidiano, os seres humanos são muito desleixados em relação a elas. 4/1 pode ser o dia da mentira para você. Para outra pessoa, pode ser o primeiro dia útil do ano, caso o primeiro dia do ano caia em uma sexta-feira. As soluções mostradas correspondem a alguns dos formatos mais comuns de data.

Outra questão é que as expressões regulares não lidam diretamente com números. Você não pode dizer a uma expressão regular para “corresponder a um número entre 1 e 31”, por exemplo. As expressões regulares trabalham caractere por caractere. Usamos ‹3[01]|[12][0-9]|0?[1-9]› para corresponder ao 3, seguido por 0 ou 1, para corresponder a 1 ou 2, seguido de qualquer dígito, ou para corresponder a um 0 opcional, seguido de 1 a 9. Em classes de caracteres, podemos usar intervalos para um dígito único, como ‹[1-9]›. Isto ocorre porque os caracteres dos dígitos de 0 a 9 ocupam posições consecutivas nas tabelas de caracteres ASCII e Unicode. Consulte o capítulo 6, para obter mais detalhes sobre a correspondência a todos os tipos de números com expressões regulares.

Devido a isso, você terá de escolher o nível de simplicidade ou exatidão da sua expressão regular. Se você já sabe que o texto de

assunto não contém datas inválidas, pode usar uma expressão regular trivial, como ‹\d{2}/\d{2}/\d{4}›. O fato de ela corresponder a coisas como 99/99/9999 é irrelevante, se isso não ocorrer no texto de assunto. Você pode construir rapidamente esta expressão regular simples, e ela será rapidamente executada.

As duas primeiras soluções para esta receita são, também, simples e rápidas, e elas também correspondem a datas inválidas, como 0/0/00 e 31/31/2008. Elas usam caracteres literais somente para os delimitadores de data, e classes de caracteres (veja receita 2.3) para os dígitos e o sinal de interrogação (veja receita 2.12), para tornar alguns dígitos opcionais. ‹(?:[0-9]{2})?[0-9]{2}› permite que o ano seja composto por dois ou quatro dígitos. ‹[0-9]{2}› corresponde a exatamente dois dígitos. ‹(?:[0-9]{2})?› corresponde a zero ou dois dígitos. O grupo de não-captura (veja receita 2.9) é necessário, pois o ponto de interrogação deve ser aplicado à combinação da classe de caracteres, com o quantificador ‹{2}›. ‹[0-9]{2}?› corresponde a exatamente dois dígitos, assim como ‹[0-9]{2}›. Sem o grupo, o ponto de interrogação torna o quantificador preguiçoso, o que não teria nenhum efeito, pois ‹{2}› não pode repetir mais que duas vezes, ou menos que duas vezes.

As soluções 3 a 6 restringem o mês para números entre 1 e 12, e o dia para números entre 1 e 31. Usamos alternância (veja receita 2.8) dentro de um grupo para corresponder a vários pares de dígitos, formando um intervalo de números de dois dígitos. Usamos grupos de captura porque, provavelmente, você vai querer capturar os números de dias e meses de qualquer maneira.

As duas últimas soluções são um pouco mais complexas; por isso, estamos apresentando-as tanto na forma condensada, quanto na forma de espaçamento livre. A única diferença entre as duas formas é a legibilidade. O JavaScript não suporta espaçamento livre. As soluções finais permitem todos os formatos de data, assim como os dois primeiros exemplos. A diferença é que os dois últimos usam um nível extra de alternância, para restringir as datas para 12/31 e

31/12, não permitindo meses inválidos, como 31/31.

Variações

Se você deseja pesquisar datas em grandes corpos de texto, ao invés de verificar se a entrada, como um todo, é uma data, você não pode usar as âncoras ‹^› e ‹$›. Apenas remover as âncoras da expressão regular não é a solução certa. Isso permitiria que qualquer uma dessas expressões regulares correspondesse a 12/12/2001 dentro de 9912/12/200199, por exemplo. Ao invés de ancorar a correspondência da expressão regular no início e no final do assunto, você precisa especificar que a data não pode fazer parte de sequências mais longas de dígitos.

Isso é facilmente realizado com um par de extremidades de palavra. Nas expressões regulares, dígitos são tratados como caracteres que podem fazer parte de palavras. Substitua ‹^› e ‹$›, por ‹\b›. Como exemplo:

```
\b(1[0-2]|0[1-9])/(3[01]|[12][0-9]|0[1-9])/[0-9]{4}\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Veja também:

Receitas 4.5, 4.6 e 4.7.

4.5 Validar formatos tradicionais de data com exatidão

Problema

Você deseja validar as datas nos formatos tradicionais mm/dd/aa, mm/dd/aaaa, dd/mm/aa e dd/mm/aaaa. E deseja, também, eliminar datas inválidas, como 31 de fevereiro.

Solução

C#

Mês antes do dia:

```
DateTime foundDate;
Match matchResult = Regex.Match(SubjectString,
  "^(?<month>[0-3]?[0-9])/(?<day>[0-3]?[0-9])/" +
  "(?<year>(?:[0-9]{2})?[0-9]{2})$");
if (matchResult.Success) {
  int year = int.Parse(matchResult.Groups["year"].Value);
  if (year < 50) year += 2000;
  else if (year < 100) year += 1900;
  try {
    foundDate = new DateTime(year,
        int.Parse(matchResult.Groups["month"].Value),
        int.Parse(matchResult.Groups["day"].Value));
  } catch {
    // Data inválida
  }
}
```

Dia antes do mês:

```
DateTime foundDate;
Match matchResult = Regex.Match(SubjectString,
  "^(?<day>[0-3]?[0-9])/(?<month>[0-3]?[0-9])/" +
  "(?<year>(?:[0-9]{2})?[0-9]{2})$");
if (matchResult.Success) {
  int year = int.Parse(matchResult.Groups["year"].Value);
  if (year < 50) year += 2000;
  else if (year < 100) year += 1900;
  try {
    foundDate = new DateTime(year,
        int.Parse(matchResult.Groups["month"].Value),
        int.Parse(matchResult.Groups["day"].Value));
  } catch {
    // Data inválida
  }
}
```

Perl

Mês antes do dia:

```
@daysinmonth = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
$validdate = 0;
```

```
if ($subject =~ m!^([0-3]?[0-9])/([0-3]?[0-9])/((?:[0-9]{2})?[0-9]{2})$!) {
  $month = $1;
  $day = $2;
  $year = $3;
  $year += 2000 if $year < 50;
  $year += 1900 if $year < 100;
  if ($month == 2 && $year % 4 == 0 && ($year % 100 != 0 || $year % 400 == 0))
{
    $validdate = 1 if $day >= 1 && $day <= 29;
  } elsif ($month >= 1 && $month <= 12) {
    $validdate = 1 if $day >= 1 && $day <= $daysinmonth[$month-1];
  }
}
```

Dia antes do mês:

```
@daysinmonth = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
$validdate = 0;
if ($subject =~ m!^([0-3]?[0-9])/([0-3]?[0-9])/((?:[0-9]{2})?[0-9]{2})$!) {
  $day = $1;
  $month = $2;
  $year = $3;
  $year += 2000 if $year < 50;
  $year += 1900 if $year < 100;
  if ($month == 2 && $year % 4 == 0 && ($year % 100 != 0 || $year % 400 == 0))
{
    $validdate = 1 if $day >= 1 && $day <= 29;
  } elsif ($month >= 1 && $month <= 12) {
    $validdate = 1 if $day >= 1 && $day <= $daysinmonth[$month-1];
  }
}
```

Expressão regular pura

Mês antes do dia:

```
^(?:
  # Fevereiro (29 dias todo ano)
  (?<month>0?2)/(?<day>[12][0-9]|0?[1-9])
|
  # meses de 30 dias
  (?<month>0?[469]|11)/(?<day>30|[12][0-9]|0?[1-9])
|
  # meses de 31 dias
  (?<month>0?[13578]|1[02])/(?<day>3[01]|[12][0-9]|0?[1-9])
```

```
)
# Ano
/(?<year>(?:[0-9]{2})?[0-9]{2})$
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET

```
^(?:
  # Fevereiro (29 dias todo ano)
  (0?2)/([12][0-9]|0?[1-9])
|
  # meses de 30 dias
  (0?[469]|11)/(30|[12][0-9]|0?[1-9])
|
  # meses de 31 dias
  (0?[13578]|1[02])/(3[01]|[12][0-9]|0?[1-9])
)
# Ano
/((?:[0-9]{2})?[0-9]{2})$
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:(0?2)/([12][0-9]|0?[1-9])|(0?[469]|11)/(30|[12][0-9]|0?[1-9])|↲
  (0?[13578]|1[02])/(3[01]|[12][0-9]|0?[1-9]))/((?:[0-9]{2})?[0-9]{2})$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Dia antes do mês:

```
^(?:
  # Fev (29 dias todo ano)
  (?<day>[12][0-9]|0?[1-9])/(?<month>0?2)
|
  # meses de 30 dias
  (?<day>30|[12][0-9]|0?[1-9])/(?<month>0?[469]|11)
|
  # meses de 31 dias
  (?<day>3[01]|[12][0-9]|0?[1-9])/(?<month>0?[13578]|1[02])
)
# Ano
/(?<year>(?:[0-9]{2})?[0-9]{2})$
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET

```
^(?:
  # Fevereiro (29 dias todo ano)
  ([12][0-9]|0?[1-9])/(0?2)
|
  # meses de 30 dias
  (30|[12][0-9]|0?[1-9])/([469]|11)
|
  # meses de 31 dias
  (3[01]|[12][0-9]|0?[1-9])/(0?[13578]|1[02])
)
# Ano
/((?:[0-9]{2})?[0-9]{2})$
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:([12][0-9]|0?[1-9])/(0?2)|(30|[12][0-9]|0?[1-9])/([469]|11)|↵
(3[01]|[12][0-9]|0?[1-9])/(0?[13578]|1[02]))/((?:[0-9]{2})?[0-9]{2})$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Existem, essencialmente, duas maneiras de validar datas com uma expressão regular. Um método seria utilizar uma expressão regular simples, que capture apenas grupos de números semelhantes a uma combinação mês/dia/ano e, em seguida, usar código procedural, para verificar se a data está correta. Eu usei a primeira expressão regular da receita anterior, que permite qualquer número entre 0 e 39 para o dia e o mês. Isto facilita a mudança do formato mm/dd/aa para dd/mm/aa, alterando qual grupo de captura será tratado como mês.

A principal vantagem deste método é que você pode incluir restrições adicionais facilmente, como limitar as datas a determinados períodos. Muitas linguagens de programação fornecem suporte específico para o tratamento de datas. A solução C# utiliza a estrutura .NET DateTime, para verificar se a data é válida e para retorná-la em um formato útil, tudo em uma única etapa.

Outro método seria fazer tudo com uma expressão regular. A

solução é viável, se tomarmos a liberdade de tratar cada ano como bissexto. Podemos usar a mesma técnica de explicitar as alternativas, como fizemos nas últimas soluções apresentadas na receita anterior.

O problema com o uso de uma única expressão regular é que ela já não captura perfeitamente o dia e o mês em um único grupo de captura. Agora, temos três grupos de captura para o mês, e três para o dia. Quando a expressão regular corresponde a uma data, apenas três dos sete grupos da expressão regular vão, realmente, capturar algo. Se o mês for fevereiro, os grupos 1 e 2 capturam o mês e o dia. Se o mês tiver 30 dias, os grupos 3 e 4 retornam o mês e o dia. Se o mês tiver 31 dias, os grupos 5 e 6 entram em ação. O grupo 7 sempre captura o ano.

Apenas o sabor de expressão regular .NET nos ajuda nesta situação. O .NET permite que vários grupos de captura nomeados (veja receita 2.11) tenham o mesmo nome, utilizando o mesmo espaço de armazenamento para grupos homônimos. Se você usar a solução exclusiva do .NET com captura nomeada, poderá, simplesmente, recuperar o texto correspondido pelos grupos “month” e “day”, sem se preocupar com quantos dias o mês possui. Todos os outros sabores discutidos neste livro não suportam captura nomeada, ou não permitem que dois grupos tenham o mesmo nome, ou retornam somente o texto capturado pelo último grupo com um determinado nome. No caso destes sabores, a captura nomeada é o único caminho.

A solução da expressão regular pura é interessante somente em situações nas quais ela é tudo que você pode usar, como quando você está usando um aplicativo que oferece uma caixa para digitar uma expressão regular. Quando estiver programando, facilite as coisas com um pouco de código extra. Isso será particularmente útil se você quiser adicionar verificações extras na data mais tarde. Veja uma solução de expressão regular pura, que corresponde a qualquer data entre 2 de maio de 2007 e 29 de agosto de 2008, no

formato dd/mm/aa ou dd/mm/aaaa:

```
  # De 2 de maio de 2007 até 29 de agosto de 2008
  ^(?:
    # De 2 de maio de 2007 até 31 de dezembro de 2007
    (?:
      # De 2 de maio até 31 de maio
      (?<day>3[01]|[12][0-9]|0?[2-9])/(?<month>0?5)/(?<year>2007)
  |
      # De 1 de junho até 31 de dezembro
      (?:
          # meses com 30 dias
          (?<day>30|[12][0-9]|0?[1-9])/(?<month>0?[69]|11)
      |
          # meses com 31 dias
          (?<day>3[01]|[12][0-9]|0?[1-9])/(?<month>0?[78]|1[02])
      )
      /(?<year>2007)
    )
  |
    # De 1 de Janeiro de 2008 até 29 de Agosto de 2008
    (?:
      # De 1 de agosto até 29 de agosto
      (?<day>[12][0-9]|0?[1-9])/(?<month>0?8)/(?<year>2008)
    |
      # De 1 de janeiro até 30 de junho
      (?:
          # Fevereiro
          (?<day>[12][0-9]|0?[1-9])/(?<month>0?2)
      |
          # meses com 30 dias
          (?<day>30|[12][0-9]|0?[1-9])/(?<month>0?[46])
      |
          # meses com 31 dias
          (?<day>3[01]|[12][0-9]|0?[1-9])/(?<month>0?[1357])
      )
      /(?<year>2008)
    )
  )$
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Veja também:

Receitas 4.4, 4.6 e 4.7.

4.6 Validar formatos de horário tradicionais

Problema

Você deseja validar horários em vários formatos tradicionais, como hh:mm e hh:mm:ss, em ambos os formatos de 12 horas e 24 horas.

Solução

Horas e minutos, relógio de 12 horas:

```
^(1[0-2]|0?[1-9]):([0-5]?[0-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Horas e minutos, relógio de 24 horas:

```
^(2[0-3]|[01]?[0-9]):([0-5]?[0-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Horas, minutos e segundos, relógio de 12 horas:

```
^(1[0-2]|0?[1-9]):([0-5]?[0-9]):([0-5]?[0-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Horas, minutos e segundos, relógio de 24 horas:

```
^(2[0-3]|[01]?[0-9]):([0-5]?[0-9]):([0-5]?[0-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Os pontos de interrogação, em todas as expressões regulares anteriores, tornam os zeros iniciais opcionais. Remova os pontos de interrogação, para tornar os zeros iniciais obrigatórios.

Discussão

Validar horários é consideravelmente mais fácil do que validar datas. Cada hora possui 60 minutos, e cada minuto possui 60 segundos. Isso significa que não precisamos de nenhuma alternância complicada na expressão regular. No caso dos minutos e segundos, não usamos alternância. ‹[0-5]?[0-9]› corresponde a um dígito entre 0 e 5, seguido por um dígito entre 0 e 9. Isso corresponde, corretamente, a qualquer número entre 0 e 59. O ponto de interrogação, após a primeira classe de caracteres, torna a alternância opcional. Dessa forma, um único dígito entre 0 e 9 é aceito também como um minuto, ou segundo, válido. Remova o ponto de interrogação, caso os primeiros 10 minutos e segundos necessitem ser escritos como 00 a 09. Veja as receitas 2.3 e 2.12 para obter detalhes sobre classes de caracteres e quantificadores, como o ponto de interrogação.

No caso das horas, utilizamos alternância (veja receita 2.8). O segundo dígito permite diferentes intervalos, dependendo do primeiro. Em um relógio de 12 horas, se o primeiro dígito for 0, o segundo permite todos os 10 dígitos, mas, se o primeiro dígito for 1, o segundo deve ser 0, 1 ou 2. Em uma expressão regular, escrevemos isto como ‹1[0-2]|0?[1-9]›. Em um relógio de 24 horas, se o primeiro dígito for 0 ou 1, o segundo dígito permite todos os 10, mas se o primeiro dígito for 2, o segundo deve estar entre 0 e 3. Na sintaxe da expressão regular, isto pode ser expresso como 2[0-3]|[01]?[0-9]. Novamente, o ponto de interrogação permite que as primeiras 10 horas sejam escritas com um único dígito. Remova o ponto de interrogação para exigir dois dígitos.

Colocamos parênteses entre as partes da expressão regular que correspondam às horas, minutos e segundos. Isso torna mais fácil recuperar os dígitos desses elementos, sem os dois pontos. A receita 2.9 explica como os parênteses criam grupos de captura. A receita 3.9 explica como você pode recuperar o texto correspondido por estes grupos de captura no código procedural.

Os parênteses em torno da parte das horas agrupam duas alternativas. Se você remover os parênteses, a expressão regular não funcionará corretamente. Remover os parênteses em torno dos minutos e dos segundos não terá qualquer efeito, apenas tornará impossível recuperar seus dígitos separadamente.

Variações

Se quiser pesquisar por horas em grandes corpos de texto, ao invés de verificar se a entrada como um todo é um horário, você não poderá usar as âncoras ‹^› e ‹$›. Apenas remover as âncoras da expressão regular não é a solução certa. Isso permitiria que as expressões regulares de horas e minutos correspondessem a 12:12 dentro de 9912:1299, por exemplo. Em vez de ancorar a correspondência da expressão regular no início e no final do assunto, você terá de especificar que o horário não poderá fazer parte de sequências de dígitos mais longas.

Isso é feito facilmente com um par de extremidades de palavra. Em expressões regulares, dígitos são tratados como caracteres que podem fazer parte de palavras. Substitua ‹^› e ‹$› por ‹\b›. Vejamos o exemplo:

```
\b(2[0-3]|[01]?[0-9]):([0-5]?[0-9])\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Extremidades de palavra não proíbem tudo; só não permitem letras, dígitos e underscores. A expressão regular que acabamos de mostrar, que corresponde ao horário e aos minutos em um relógio de 24 horas, corresponde a 16:08 dentro do texto de assunto The time is 16:08:42 sharp. O espaço não é um caractere de palavra, enquanto 1 é. Portanto, a extremidade de palavra corresponde entre eles. O 8 é um caractere de palavra, enquanto o caractere de dois pontos não o é; então ‹\b› também corresponde entre eles.

Se você deseja proibir os dois pontos e os caracteres de palavra, utilize um lookaround (veja receita 2.16). A seguinte expressão

regular não corresponderá a qualquer parte de The time is 16:08:42 sharp. Ela só funciona com sabores que suportem lookbehind:

```
(?<![:\w])(2[0-3]|[01]?[0-9]):([0-5]?[0-9])(?![:\w])
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Veja também:

Receitas 4.4, 4.5, e 4.7.

4.7 Validando datas e horários ISO 8601

Problema

Você deseja corresponder a datas ou horários no formato oficial ISO 8601, a base para muitos formatos padronizados de data e hora. Por exemplo, no esquema XML, os tipos nativos date, time e dateTime são todos baseados na norma ISO 8601.

Solução

A expressão a seguir corresponde a um mês de calendário, como, por exemplo, 2008-08. O hífen é obrigatório:

```
^([0-9]{4})-(1[0-2]|0[1-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-(?<month>1[0-2]|0[1-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
^(?P<year>[0-9]{4})-(?P<month>1[0-2]|0[1-9])$
```

Opções Regex: Nenhuma

Sabores Regex: PCRE, Python

Data de calendário, por exemplo, 2008-08-30. Os hífens são opcionais. Esta expressão regular permite AAAAMMDD e AAAAMM-

DD, que não seguem o padrão ISO 8601:

```
^([0-9]{4})-?(1[0-2]|0[1-9])-?(3[0-1]|0[1-9]|[1-2][0-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-?(?<month>1[0-2]|0[1-9])-?↲
(?<day>3[0-1]|0[1-9]|[1-2][0-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Data de calendário, por exemplo, 2008-08-30. Os hífens são opcionais. Esta expressão regular utiliza uma condicional para excluir AAAA-MMDD e AAAAMM-DD. Existe um grupo extra de captura para o primeiro hífen:

```
^([0-9]{4})(-)?(1[0-2]|0[1-9])(?(2)-)(3[0-1]|0[1-9]|[1-2][0-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE, Perl, Python

Data de calendário, por exemplo, 2008-08-30. Os hífens são opcionais. Esta expressão regular utiliza alternância, para excluir AAAA-MMDD e AAAAMM-DD. Existem dois grupos de captura para o mês:

```
^([0-9]{4})(?:(1[0-2]|0[1-9])|-?(1[0-2]|0[1-9])-?)↲
(3[0-1]|0[1-9]|[1-2][0-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Semana do ano, por exemplo, 2008-W35. O hífen é opcional:

```
^([0-9]{4})-?W(5[0-3]|[1-4][0-9]|0[1-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-?W(?<week>5[0-3]|[1-4][0-9]|0[1-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Data de semana, por exemplo, 2008-W35-6. Os hífens são opcionais:

```
^([0-9]{4})-?W(5[0-3]|[1-4][0-9]|0[1-9])-?([1-7])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-?W(?<week>5[0-3]|[1-4][0-9]|0[1-9])-?(?<day>[1-7])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Data ordinal, por exemplo, 2008-243. O hífen é opcional:

```
^([0-9]{4})-?(36[0-6]|3[0-5][0-9]|[12][0-9]{2}|0[1-9][0-9]|00[1-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>[0-9]{4})-?↲
(?<day>36[0-6]|3[0-5][0-9]|[12][0-9]{2}|0[1-9][0-9]|00[1-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Horas e minutos, por exemplo, 17:21. Os dois pontos são opcionais:

```
^(2[0-3]|[01]?[0-9]):?([0-5]?[0-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<hour>2[0-3]|[01]?[0-9]):?(?<minute>[0-5]?[0-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Horas, minutos e segundos, por exemplo, 17:21:59. Os dois pontos são opcionais:

```
^(2[0-3]|[01]?[0-9]):?([0-5]?[0-9]):?([0-5]?[0-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<hour>2[0-3]|[01]?[0-9]):?(?<minute>[0-5]?[0-9]):?↲
(?<second>[0-5]?[0-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Indicador de fuso horário, por exemplo, Z, +07 ou +07:00. Os dois pontos e os minutos são opcionais:

```
^(Z|[+-](?:2[0-3]|[01]?[0-9])(?::?(?:[0-5]?[0-9]))?)$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Horas, minutos e segundos, com indicador de fuso horário, por exemplo, 17:21:59+07:00. Todos os dois pontos são opcionais. Os minutos também são opcionais no indicador de fuso horário:

```
^(2[0-3]|[01]?[0-9]):?([0-5]?[0-9]):?([0-5]?[0-9])↲
(Z|[+-](?:2[0-3]|[01]?[0-9])(?::?(?:[0-5]?[0-9]))?)$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<hour>2[0-3]|[01]?[0-9]):?(?<minute>[0-5]?[0-9]):?(?<sec>[0-5]?[0-9])↲
(?<timezone>Z|[+-](?:2[0-3]|[01]?[0-9])(?::?(?:[0-5]?[0-9]))?)$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Data com fuso horário opcional, por exemplo, 2008-08-30 ou 2008-08-30+07:00. Hífens são obrigatórios:

Este é o tipo date do esquema XML:

```
^(-?(?:[1-9][0-9]*)?[0-9]{4})-(1[0-2]|0[1-9])-(3[0-1]|0[1-9]|[1-2][0-9])↲
(Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9])?$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>-?(?:[1-9][0-9]*)?[0-9]{4})-(?<month>1[0-2]|0[1-9])-↲
(?<day>3[0-1]|0[1-9]|[1-2][0-9])↲
(?<timezone>Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9])?$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Horário, com microssegundos e fuso horário opcionais, por exemplo, 01:45:36 ou 01:45:36.123+07:00. Este é o tipo dateTime do esquema XML:

```
^(2[0-3]|[0-1][0-9]):([0-5][0-9]):([0-5][0-9])(\.[0-9]+)?↲
(Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9])?$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<hour>2[0-3]|[0-1][0-9]):(?<minute>[0-5][0-9]):(?<second>[0-5][0-9(?<ms>\.[0-9]+)?(?<timezone>Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9])?$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Data e horário, com microssegundos e fuso horário opcionais, por exemplo, 2008-08-30T01:45:36 ou 2008-08-30T01:45:36.123Z. Este é o tipo dateTime do esquema XML:

```
^(-?(?:[1-9][0-9]*)?[0-9]{4})-(1[0-2]|0[1-9])-(3[0-1]|0[1-9]|[1-2][0-9])↵
T(2[0-3]|[0-1][0-9]):([0-5][0-9]):([0-5][0-9])(\.[0-9]+)?↵
(Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9])?$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^(?<year>-?(?:[1-9][0-9]*)?[0-9]{4})-(?<month>1[0-2]|0[1-9])-↵
(?<day>3[0-1]|0[1-9]|[1-2][0-9])T(?<hour>2[0-3]|[0-1][0-9]):↵
(?<minute>[0-5][0-9]):(?<second>[0-5][0-9])(?<ms>\.[0-9]+)?↵
(?<timezone>Z|[+-](?:2[0-3]|[0-1][0-9]):[0-5][0-9])?$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

Discussão

A norma ISO 8601 define uma ampla gama de formatos de data e hora. As expressões regulares, aqui apresentadas, abrangem os formatos mais comuns, mas a maioria dos sistemas que usa o padrão ISO 8601 emprega apenas um subconjunto. Por exemplo, no esquema XML de datas e horários, os hífens e dois pontos são obrigatórios. Para tornar obrigatórios hífens e dois pontos, basta remover os pontos de interrogação que aparecem depois deles. Para não permitir hífens e dois pontos, remova os hífens e os dois pontos, juntamente com o ponto de interrogação, depois deles. Atente para os grupos de não-captura, que usam a sintaxe ‹(?:group)›. Se um ponto de interrogação e um caractere de dois pontos aparecerem depois de um parêntese de abertura, estes três caracteres abrem um grupo de não-captura.

As expressões regulares tornam opcionais os hífens individuais e os

dois pontos, o que não segue exatamente o padrão ISO 8601. Por exemplo, 1733:26 não é um horário ISO 8601 válido, mas será aceito pelas expressões regulares de horário. Exigir que todos os hífens e os dois pontos estejam presentes, ou omitidos, torna sua expressão regular um pouco mais complexa. Nós fizemos isso como um exemplo para a expressão regular de data, mas, na prática, como acontece com os tipos do esquema XML, normalmente os delimitadores são obrigatórios ou proibidos, ao invés de serem opcionais.

Colocamos parênteses em torno de todas as partes de números na expressão regular. Isso facilita recuperar os números para ano, mês, dia, hora, minuto, segundo e fusos horários. A receita 2.9 explica como os parênteses criam grupos de captura. A receita 3.9 explica como você pode recuperar o texto correspondido, no código procedural, por esses grupos de captura.

Para a maioria das expressões regulares, também mostramos uma alternativa que utiliza a captura nomeada. Alguns destes formatos de data e hora podem ser estranhos para você ou seus colegas desenvolvedores. A captura nomeada facilita o entendimento da expressão regular. .NET, PCRE 7, Perl 5,10 e Ruby 1.9 suportam a sintaxe ‹(?<name>group)›. Todas as versões do PCRE e do Python abordadas neste livro suportam a sintaxe alternativa ‹(?P<name>group)›, que adiciona um ‹P›. Veja as receitas 2.11 e 3.9, para mais detalhes.

Os intervalos de números são rígidos em todas as expressões regulares. Por exemplo, o dia de calendário é restrito entre 01 e 31. Você nunca vai acabar com 32 dias ou 13 meses. Nenhuma das expressões regulares, aqui, tenta excluir correspondências inválidas de dia e mês, como 31 de fevereiro. A receita 4.5 explica como você pode lidar com isso.

Embora algumas dessas expressões regulares sejam bastante longas, elas são todas muito simples, e usam as mesmas técnicas explicadas nas receitas 4.4 e 4.6.

Veja também:

Receitas 4.4, 4.5 e 4.6.

4.8 Limitar a entrada a caracteres alfanuméricos

Problema

Seu aplicativo requer que os usuários limitem suas respostas a um ou mais caracteres alfanuméricos do alfabeto inglês.

Solução

Com expressões regulares à disposição, a solução é muito simples. Uma classe de caracteres pode configurar o intervalo permitido de caracteres. Com a inclusão de um quantificador, que repete a classe de caracteres uma ou mais vezes, e as âncoras, que vinculam a correspondência ao início e ao final da string, você está pronto para seguir em frente.

Expressão regular

```
^[A-Z0-9]+$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ruby

```
if subject =~ /^[A-Z0-9]+$/i
  puts "Subject is alphanumeric"
else
  puts "Subject is not alphanumeric"
end
```

Outras linguagens de programação

Veja as receitas 3.4 e 3.5, para ajudá-lo na implementação desta expressão regular com outras linguagens de programação.

Discussão

Vamos olhar as quatro partes da expressão regular, uma por vez:

```
^ # Declara a posição no início da string.
[A-Z0-9] # Corresponde um caractere de "A" a "Z" ou de "0" a "9"...
  + # entre uma e ilimitadas vezes.
$ # Declara a posição no final da string.
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, espaçamento livre.

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

As declarações ‹^› e ‹$›, no início e no final da expressão regular, garantem que toda a string de entrada seja testada. Sem elas, a expressão regular poderia corresponder a qualquer parte de uma string mais longa, permitindo a inclusão de caracteres inválidos. O quantificador ‹+› repete, uma ou mais vezes, o elemento anterior. Se você quiser permitir que a expressão regular corresponda a uma string completamente vazia, pode substituir ‹+›, por ‹*›. O quantificador asterisco ‹*› permite zero ou mais repetições, efetivamente tornando opcional o elemento anterior.

Variações

Limitar a entrada a caracteres ASCII

A seguinte expressão regular limita a entrada aos 128 caracteres da tabela de caracteres ASCII de sete bits. Isso inclui 33 caracteres de controle invisíveis:

```
^[\x00-\x7F]+$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Limitar a entrada a caracteres ASCII não controladores, e a quebras de linhas

Use a seguinte expressão regular para limitar a entrada a caracteres visíveis e espaços em branco da tabela de caracteres ASCII, excluindo os caracteres de controle. Os caracteres de alimentação

de linha e de retorno de carro (posições 0x0A e 0x0D, respectivamente) são os caracteres de controle mais comumente utilizados; então, são incluídos explicitamente, utilizando ‹\n› (alimentação de linha) e ‹\r› (retorno de carro):

```
^[\n\r\x20-\x7E]+$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Limitar a entrada a caracteres compartilhados pela ISO-8859-1 e pelo Windows-1252

ISO-8859-1 e Windows-1252 (muitas vezes referido como ANSI) são duas codificações de caracteres de oito bits, comumente utilizadas, baseadas no padrão Latin-1 (ou, mais formalmente, ISO/IEC 8859-1). No entanto, os caracteres que eles mapeiam para as posições entre 0x80 e 0x9F são incompatíveis. A ISO-8859-1 utiliza essas posições para códigos de controle, enquanto o Windows-1252 as utiliza para uma ampla gama de letras e pontuações.

Essas diferenças, por vezes, levam à dificuldade de exibição de caracteres; especialmente no caso de documentos que não declaram codificação, ou quando o destinatário está usando um sistema que não seja Windows. A seguinte expressão regular pode ser usada para limitar a entrada a caracteres compartilhados pela ISO-8859-1 e pelo Windows-1252 (incluindo caracteres de controle compartilhados):

```
^[\x00-\x7F\xA0-\xFF]+$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

A notação hexadecimal torna essa expressão regular difícil de ler, mas ela funciona como a classe de caracteres ‹[A-Z0-9]›, mostrada anteriormente. Ela corresponde a caracteres em dois intervalos: \x00-\x7F e \xA0-\xFF.

Limitar a entrada a caracteres alfanuméricos em qualquer idioma

Esta expressão regular limita a entrada a letras e números de qualquer idioma ou sistema de escrita. Ela utiliza uma classe de caracteres que inclui propriedades para todos os pontos de código das categorias de letra e número do Unicode:

```
^[\p{L}\p{N}]+$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Ruby 1.9

Infelizmente, as propriedades Unicode não são suportadas por todos os sabores de expressões regulares cobertos por este livro. Especificamente, essa expressão regular não trabalha com JavaScript, Python ou Ruby 1.8. Além disso, utilizar esta expressão regular com o PCRE exige que ele seja compilado com suporte a UTF-8. Propriedades Unicode podem ser usadas com as funções preg do PHP (que se baseiam em PCRE), caso a opção /u esteja anexada à expressão regular.

A expressão regular a seguir mostra uma solução alternativa para o Python:

```
^[^\W_]+$
```

Opções Regex: Unicode

Sabores Regex: Python

Aqui, nós contornamos a falta de propriedades Unicode no Python utilizando a sinalização UNICODE, ou U, ao criar a expressão regular. Ela altera o significado de alguns tokens da expressão regular, fazendo com que eles utilizem a tabela de códigos Unicode. ‹\w› nos leva a uma solução quase completa, já que ele corresponde a caracteres alfanuméricos e ao underscore. Ao usar seu inverso (‹\W›) em uma classe de caracteres negada, podemos remover o underscore deste conjunto. Negativas duplas, como esta, são às vezes bastante úteis em expressões regulares, embora sejam, talvez, de difícil compreensão[1].

Veja também:

A receita 4.9 mostra como limitar o texto por comprimento, em vez do conjunto de caracteres.

4.9 Limitar o comprimento do texto

Problema

Você deseja testar se a string é composta de 1 a 10 letras, de A a Z.

Solução

Todas as linguagens de programação cobertas por este livro fornecem uma maneira simples e eficiente de verificação do tamanho do texto. Por exemplo, strings JavaScript possuem uma propriedade `length`, que contém um inteiro indicando o comprimento da string. No entanto, usar expressões regulares para verificar o tamanho do texto pode ser útil em algumas situações, particularmente quando o comprimento é apenas uma das várias regras que determinam se o texto de assunto encaixa-se no padrão desejado. A seguinte expressão regular garante que o texto possui entre 1 e 10 caracteres e, adicionalmente, limita o texto às letras maiúsculas de A a Z. Você pode modificar a expressão regular para permitir qualquer comprimento de texto, mínimo ou máximo, ou permitir outros caracteres, além do intervalo A-Z.

Expressão regular

```
^[A-Z]{1,10}$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Perl

```
if ($ARGV[0] =~ /^[A-Z]{1,10}$/) {
  print "Input is valid\n";
} else {
  print "Input is invalid\n";
```

```
}
```

Outras linguagens de programação

Veja a receita 3.5, para ajudá-lo na implementação desta expressão regular com outras linguagens de programação.

Discussão

Veja o desmembramento dessa expressão regular simples:

```
^ # Declare a posição no início da string
[A-Z] # Corresponda a uma letra de "A" a "Z"...
  {1,10} # entre 1 e 10 vezes.
$ # Declare a posição no final da string
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

As âncoras ‹^› e ‹$› garantem que a expressão regular corresponda a toda a string de assunto, caso contrário, ela poderia corresponder a 10 caracteres dentro de um texto maior. A classe de caracteres ‹[A-Z]› corresponde a qualquer caractere maiúsculo de A a, e o quantificador de intervalo ‹{1,10}› repete a classe de caracteres de 1 a 10 vezes. Ao combinar o quantificador de intervalo com as âncoras de início e de final da string, a expressão regular não corresponderá, caso o comprimento do texto de assunto esteja fora do intervalo desejado.

Note que a classe de caracteres ‹[A-Z]› permite, explicitamente, apenas letras maiúsculas. Se você também quiser permitir as letras minúsculas de a a z, terá de alterar a classe de caracteres para ‹[A-Za-z]›, ou terá de aplicar a opção de não-diferenciação entre maiúsculas e minúsculas. A receita 3.4 mostra como fazer isso.

Um erro comum, entre os novos usuários de expressões regulares, é tentar salvar alguns caracteres, utilizando o intervalo da classe de caracteres ‹[A-z]›. À primeira vista, pode parecer um truque inteligente, para permitir todas as letras maiúsculas e minúsculas. No entanto, a tabela de caracteres ASCII inclui vários caracteres de pontuação, posicionados entre os intervalos de A a Z, e de a a z.

Assim, ‹[A-z]›, na verdade, equivale a ‹[A-Z[\]^_`a-z]›.

Variações

Limitar o comprimento de um padrão arbitrário

Como os quantificadores do tipo ‹{1,10}› aplicam-se apenas ao elemento imediatamente anterior, devemos ter uma abordagem diferente, ao limitar o número de caracteres que possam ser correspondidos por padrões que incluam mais do que um token simples.

Conforme explicado na receita 2.16, lookaheads (e sua contraparte, lookbehinds) são um tipo especial de declaração que, assim como ‹^› e ‹$›, correspondem em uma posição dentro da string de assunto e não consomem caracteres. Lookaheads podem ser positivos ou negativos, o que significa que podem verificar se um padrão ocorre, ou não, na posição atual da correspondência. O lookahead positivo, escrito como ‹(?=...)›, pode ser usado no início do padrão, para garantir que a string esteja dentro do intervalo-alvo de comprimento. O resto da expressão regular pode validar o padrão desejado, sem se preocupar com o tamanho do texto. Vejamos um exemplo simples:

```
^(?=.{1,10}$).*
```

Opções Regex: Ponto corresponde a quebras de linha

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?=[\S\s]{1,10}$)[\S\s]*
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

É importante que a âncora ‹$› apareça dentro do lookahead, pois o teste de comprimento máximo só funciona se assegurarmos que não existam mais caracteres, após atingirmos o limite. Como o lookahead, no início da expressão regular, assegura o intervalo de comprimento, o padrão que vem após pode, então, aplicar regras de validação adicionais. Nesse caso, o padrão ‹.*› (ou ‹[\S\s]*›, na versão que adiciona suporte a JavaScript) é usado simplesmente

para corresponder a todo texto de assunto sem restrições adicionais.

A primeira expressão regular utiliza a opção “ponto corresponde a quebras de linha” para funcionar corretamente quando sua string de assunto possuir quebras de linha. Veja a receita 3.4, para obter detalhes sobre como aplicar este modificador em sua linguagem de programação. O JavaScript não possui uma opção “ponto corresponde a quebras de linha”, de modo que a segunda expressão regular utiliza uma classe de caracteres que corresponde a qualquer caractere. Consulte “Qualquer caractere, incluindo quebras de linha”, receita 2.4, para mais informações.

Limitar número de caracteres que não sejam espaços em branco

A seguinte expressão regular corresponde a qualquer string que contenha entre 10 e 100 caracteres que não sejam espaços em branco:

```
^\s*(?:\S\s*){10,100}$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Em Java, PCRE, Python e Ruby, ‹\s› corresponde apenas a caracteres ASCII de espaço em branco, e ‹\S› corresponde a todo o resto. Em Python, você pode fazer ‹\s› corresponder a todos os espaços em branco Unicode, passando a sinalização UNICODE, ou U, ao criar a expressão regular. Os desenvolvedores que utilizam Java, PCRE, e Ruby 1.9, e que desejam evitar qualquer contagem de espaços em branco Unicode em seu limite de caracteres, podem mudar para a versão seguinte, que tira proveito das propriedades Unicode (descritas na receita 2.7):

```
^[\p{Z}\s]*(?:[^\p{Z}\s][\p{Z}\s]*){10,100}$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Ruby 1.9

O PCRE deve ser compilado com suporte a UTF-8 para que isso

funcione. No PHP, ative o suporte a UTF-8 com o modificador de padrão /u.

Esta última expressão regular combina a propriedade Separator (Separador) do Unicode, ‹\p{Z}›, à abreviação ‹\s› para espaços em branco. Isso porque os caracteres correspondidos por ‹\p{Z}› e ‹\s› não se sobrepõem completamente. ‹\s› inclui os caracteres nas posições de 0x09 a 0x0D (tabulação horizontal, alimentação de linha, tabulação vertical, alimentação de formulário e retorno de carro), que não são atribuídos pelo padrão Unicode à propriedade Separator. Ao combinar ‹\p{Z}› e ‹\s› em uma classe de caracteres, você garante que todos os caracteres em branco sejam correspondidos.

Em ambas as expressões regulares, o quantificador de intervalo ‹{10,100}› é aplicado ao grupo de não-captura que o antecede, em vez de ser aplicado a um único token. O grupo corresponde a qualquer caractere único que não seja um espaço em branco, seguido de zero ou mais caracteres de espaço em branco. O quantificador de intervalo pode monitorar, com confiança, o número de caracteres que não são espaços em branco e que são correspondidos, pois apenas um caractere que não seja espaço em branco é correspondido durante cada iteração.

Limitar o número de palavras

A seguinte expressão regular é muito semelhante ao exemplo anterior, em que limitamos o número de caracteres que não sejam espaços em branco, exceto que, agora, cada repetição corresponde a uma palavra inteira, e não a um único caractere que não seja espaço em branco. Ela corresponde entre 10 e 100 palavras, ignorando os caracteres que não sejam de palavra, incluindo pontuação e espaços em branco:

```
^\W*(?:\w+\b\W*){10,100}$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Em Java, JavaScript, PCRE e Ruby, o token de caractere de palavra ‹\w› desta expressão regular somente corresponderá aos caracteres ASCII A-Z, a-z, 0-9 e _, portanto, ele não conseguirá contar corretamente as palavras que contenham letras não-ASCII e números. Em .NET e Perl, ‹\w› é baseado na tabela Unicode (assim como seu inverso, ‹\W›, e a extremidade de palavra ‹\b›), e corresponderá a letras e números de todos os sistemas de escrita Unicode. Em Python, você pode escolher se esses tokens devem ser baseados em Unicode ou não, dependendo de você ter passado a sinalização UNICODE, ou U, ao criar a expressão regular.

Se quiser contar as palavras que contenham letras e números não-ASCII, as seguintes expressões regulares fornecem essa capacidade para os sabores de expressão regular adicionais:

```
^[^\p{L}\p{N}_]*(?:[\p{L}\p{N}_]+\b[^\p{L}\p{N}_]*){10,100}$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, Perl

```
^[^\p{L}\p{N}_]*(?:[\p{L}\p{N}_]+(?:[^\p{L}\p{N}_]+|$)){10,100}$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Ruby 1.9

O PCRE deve ser compilado com suporte a UTF-8 para que isso funcione. No PHP, ative o suporte a UTF-8, com o modificador de padrão /u.

Como se observa, a razão para estas expressões regulares diferentes (mas equivalentes) é a manipulação variada dos tokens de caractere de palavra e de extremidade de palavra, explicados em “Caracteres de palavra”, receita 2.6.

As duas últimas expressões regulares usam classes de caracteres que incluem as propriedades Unicode para letras e números separadas (‹\p{L}› e ‹\p{N}›), adicionando manualmente o caractere underscore a cada classe para torná-las equivalentes à expressão regular anterior, que dependia de ‹\w› e ‹\W›.

Cada repetição do grupo de não-captura, nas duas primeiras destas três expressões regulares, corresponde a uma palavra inteira

seguida de zero, ou a mais caracteres que não sejam de palavras. É permitido ao token ‹\W› (ou ‹[^\p{L}\p{N}_]›), dentro do grupo, ser repetido zero vez, no caso da string terminar com um caractere de palavra. No entanto, como isso torna opcional a sequência de caracteres que não sejam de palavra em todo o processo de correspondência, a declaração da extremidade de palavra ‹\b› torna-se necessária entre ‹\w› e ‹\W› (ou ‹[\p{L}\p{N}_]› e ‹[^\p{L}\p{N}_]›), para garantir que cada repetição do grupo realmente corresponda a uma palavra inteira. Sem a extremidade de palavra, seria permitida a uma única repetição corresponder a qualquer parte de uma palavra, com repetições subsequentes correspondendo a partes adicionais.

A terceira versão da expressão regular (que adiciona suporte a PCRE e Ruby 1.9) funciona de um jeito um pouco diferente. Ela usa sinal de adição (um ou mais), ao invés de um quantificador asterisco (zero ou mais), e permite, explicitamente, a correspondência de zero caracteres somente se o processo de correspondência chegar no final da string. Isso nos permite evitar o token de extremidade de palavra, necessário para garantir a precisão, uma vez que ‹\b› não está habilitado para Unicode no PCRE ou no Ruby. ‹\b› está habilitado para Unicode no Java, embora ‹\w› não esteja.

Infelizmente, nenhuma dessas opções permite que o JavaScript ou o Ruby 1.8 trate corretamente as palavras que usem caracteres não-ASCII. Uma solução possível seria reformular a expressão regular para contar espaços em branco, em vez de sequências de caracteres de palavras, como mostrado aqui:

```
^\s*(?:\S+(?:\s+|$)){10,100}$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, Perl, PCRE, Python, Ruby

Em muitos casos, isso irá funcionar como nas soluções anteriores, embora não sejam exatamente equivalentes. Por exemplo, uma diferença é que palavras compostas ligadas por um hífen (como “far-reaching”) passarão a ser contadas como uma palavra, e não duas.

Veja também:

Receitas 4.8 e 4.10.

4.10 Limitar o número de linhas no texto

Problema

Você precisa verificar se uma string é composta de cinco linhas ou menos, sem levar em conta o total de caracteres presentes na string.

Solução

Os caracteres exatos, ou as sequências de caracteres usadas como separadores de linha, podem variar, dependendo da convenção do seu sistema operacional, aplicativo, preferências de usuário etc. Elaborar uma solução ideal, portanto, levanta questões sobre quais convenções devem ser suportadas para indicar o início de uma nova linha. As seguintes soluções suportam o padrão MS-DOS/Windows (‹\r\n›), Mac OS (‹\r›), e as convenções de quebra de linha do Unix/Linux/OS X (‹\n›).

Expressão regular

Os três sabores de expressões regulares específicos, mostrados a seguir, contêm duas diferenças. A primeira expressão regular utiliza grupos atômicos, escritos como ‹(?>...)›, em vez de grupos de não-captura, escritos como ‹(?:...)›, pois, nesse caso, eles têm o potencial de proporcionar uma pequena melhoria na eficiência dos sabores de expressão regular que os suportam. Python e JavaScript não suportam grupos atômicos; então, eles não são usados nesses sabores. Outra diferença fica por conta dos tokens usados para declarar a posição no início e no final da string (‹\A› ou ‹^›, para o início da string, e ‹\z›, ‹\Z› ou ‹$›, para o final). As razões para essa variação serão discutidas em profundidade adiante, nesta receita.

Todos os três sabores específicos de expressão regular correspondem exatamente às mesmas strings:

```
\A(?>(?>\r\n?|\n)?[^\r\n]*){0,5}\z
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Ruby

```
\A(?:(?:\r\n?|\n)?[^\r\n]*){0,5}\Z
```

Opções Regex: Nenhuma

Sabores Regex: Python

```
^(?:(?:\r\n?|\n)?[^\r\n]*){0,5}$
```

Opções Regex: Nenhuma

Sabores Regex: JavaScript

PHP (PCRE)

```
if (preg_match('/\A(?>(?>\r\n?|\n)?[^\r\n]*){0,5}\z/', $_POST['subject'])) {
  print 'Subject contains five or fewer lines';
} else {
  print 'Subject contains more than five lines';
}
```

Outras linguagens de programação

Veja a receita 3.5, para ajudá-lo na implementação destas expressões regulares com outras linguagens de programação.

Discussão

Todas as expressões regulares mostradas até agora, nesta receita, usam um agrupamento que corresponde a uma sequência de quebra de linha do MS-DOS/Windows, Mac OS (legado) ou Unix/Linux/OS X, seguida por qualquer número de caracteres que não sejam quebras de linha. O agrupamento é repetido entre zero e cinco vezes, pois estamos correspondendo a até cinco linhas.

No exemplo seguinte, desmembramos a expressão regular da versão JavaScript em partes individuais. Nós usamos a versão do JavaScript, pois seus elementos são, provavelmente, familiares a uma ampla gama de leitores. Explicaremos, depois, as variantes de

sabores de expressão regular:

```
^ # Declara a posição no início da string
(?: # Agrupa mas não captura...
  (?: # Agrupa mas não captura...
    \r # Corresponde a um retorno de carro (CR, posição ASCII 0x0D).
    \n # Corresponde a uma alimentação de linha (LF, posição ASCII 0x0A)...
       ? # entre zero e uma vez.
    | # ou...
       \n # Corresponde a um caractere de alimentação de linha
    ) # Finaliza o grupo de não captura.
       ? # Repete o grupo anterior entre zero e uma vez.
    [^\r\n] # Corresponde a qualquer caractere simples, exceto CR ou LF...
       * # entre zero e um número ilimitado de vezes.
) # Finaliza o grupo de não-captura.
  {0,5} # Repete o grupo anterior entre zero e cinco vezes.
$ # Declara a posição no final da string.
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

O ‹^› inicial corresponde à posição no início da string. Isso ajuda a garantir que a string não possua mais que cinco linhas, porque, a menos que a expressão regular seja forçada a começar no início da string, ela poderá corresponder a quaisquer cinco linhas dentro de uma string mais longa.

Em seguida, um grupo de não-captura engloba a combinação de uma sequência de quebra de linha e qualquer número de caracteres que não sejam quebras de linhas. O quantificador, que vem na sequencia, permite que este grupo repita entre zero e cinco vezes (zero repetição corresponderia a uma string completamente vazia). Dentro do grupo externo, um subgrupo opcional corresponde a uma sequência de quebra de linha. Em seguida, vem a classe de caracteres que corresponde a qualquer número de caracteres que não sejam quebras de linha.

Dê uma boa olhada na ordem dos elementos do grupo externo (primeiro uma quebra de linha; em seguida, uma sequência que não seja quebra de linha). Se invertêssemos a ordem, de forma que o grupo fosse escrito como ‹(?:[^\r\n]*(?:\r\n?|\n)?)›, uma quinta repetição

permitiria uma quebra de linha final. Efetivamente, tal mudança permitiria uma sexta linha vazia.

O subgrupo permite qualquer uma das três sequências de quebra de linha:

- Um retorno de carro, seguido por uma alimentação de linha (‹\r\n›, a sequência de quebra de linha convencional do MS-DOS/Windows).
- Um retorno de carro independente (‹\r›, caractere de quebra de linha do Mac OS legado).
- Uma alimentação de linha independente (‹\n›, caractere de quebra de linha convencional do Unix/Linux/OS X).

Agora, passemos para as diferenças entre sabores.

A primeira versão da expressão regular (utilizada por todos os sabores, exceto Python e JavaScript) usa grupos atômicos, ao invés de grupos simples de não-captura. Embora em alguns casos o uso de grupos atômicos possa ter um impacto muito mais profundo, neste caso eles simplesmente permitem que o mecanismo de expressão regular evite um pouco de retrocesso desnecessário, que pode ocorrer caso a tentativa de correspondência falhe (veja receita 2.15, para obter mais informações sobre os grupos atômicos).

As outras diferenças entre sabores são os símbolos usados para declarar a posição no início e no final da string. O desmembramento mostrado anteriormente utilizou ‹^› e ‹$› para esses fins. Embora tais âncoras sejam suportadas por todos os sabores de expressão regular discutidos aqui, as expressões regulares alternativas, nesta seção, utilizaram ‹\A›, ‹\Z› e ‹\z›. A explicação mais breve para isso é que o significado desses metacaracteres difere ligeiramente entre os sabores de expressão regular. A explicação longa nos conduz a um pouco de história a respeito das expressões regulares....

Ao utilizar Perl para ler uma linha de um arquivo, a string resultante termina com uma quebra de linha. Por isso, o Perl introduziu um “reforço” para o significado tradicional de ‹$›, copiado pela maioria

dos sabores de expressão regular. Além de corresponder no final absoluto de uma string, o ‹$› do Perl corresponde exatamente antes da quebra de linha em que termina a string. O Perl também introduziu mais duas declarações que correspondem no final de uma string: ‹\Z› e ‹\z›. A âncora ‹\Z› do Perl possui o mesmo significado peculiar de ‹$›, exceto pelo fato de ela não mudar quando a opção de permitir a correspondência de ‹^› e ‹$› em quebras de linha estiver habilitada. ‹\z› sempre corresponde apenas no final absoluto de uma string, sem exceções. Como esta receita lida explicitamente com quebras de linha, a fim de contar as linhas em uma string, ela utiliza a declaração ‹\z› para os sabores de expressão regular que suportam-na, garantindo que uma sexta linha vazia não seja permitida.

A maioria dos outros sabores de expressão regular copia as âncoras de fim-de-linha/string do Perl. .NET, Java, PCRE e Ruby suportam ‹\Z› e ‹\z›, com o mesmo significado que o Perl. O Python inclui apenas ‹\Z› (maiúsculo), mas, confusamente, muda seu significado para corresponder apenas no final absoluto da string, assim como o ‹\z› minúsculo do Perl. O JavaScript não inclui qualquer âncora “z”, mas, ao contrário de todos os outros sabores discutidos aqui, sua âncora ‹$› corresponde apenas no final absoluto da string (quando a opção de permitir a correspondência de ‹^› e ‹$› em quebras de linha não estiver habilitada).

Quanto à ‹\A›, a situação é um pouco melhor. Ela sempre corresponde apenas no início de uma string, e isso significa exatamente a mesma coisa em todos os sabores discutidos aqui, exceto o JavaScript (que não o suporta).

Embora seja lamentável que existam esses tipos de incoerências entre os sabores, um dos benefícios de usar as expressões regulares deste livro é que, normalmente, você não precisa se preocupar com tais incoerências. Detalhes sórdidos, como os que acabamos de descrever, foram incluídos para o caso de você querer ir mais fundo.

Variações

Trabalhar com separadores de linha esotéricos

As expressões regulares mostradas anteriormente limitam o suporte às sequências de caracteres de quebra de linha convencionais do MS-DOS/Windows, Unix/Linux/OS X e Mac OS. No entanto, existem vários caracteres raros de espaços em branco verticais que você pode encontrar, ocasionalmente. As expressões regulares a seguir levam em conta esses caracteres adicionais, enquanto limitam as correspondências a cinco linhas de texto, ou menos.

```
\A(?>\R?\V*){0,5}\z
```

Opções Regex: Nenhuma

Sabores Regex: PCRE 7 (com a opção PCRE_BSR_UNICODE), Perl 5.10

```
\A(?>(?>\r\n?|[\n-\f\x85\x{2028}\x{2029}])?↵
[^\n-\r\x85\x{2028}\x{2029}]*){0,5}\z
```

Opções Regex: Nenhuma

Sabores Regex: PCRE, Perl

```
\A(?>(?>\r\n?|[\n-\f\x85\u2028\u2029])?[^\n-\r\x85\u2028\u2029]*){0,5}\z
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, Ruby

```
\A(?:(?:\r\n?|[\n-\f\x85\u2028\u2029])?[^\n-\r\x85\u2028\u2029]*){0,5}\Z
```

Opções Regex: Nenhuma

Sabores Regex: Python

```
^(?:(?:\r\n?|[\n-\f\x85\u2028\u2029])?[^\n-\r\x85\u2028\u2029]*){0,5}$
```

Opções Regex: Nenhuma

Sabores Regex: JavaScript

Todas essas expressões regulares lidam com os separadores de linha da tabela 4.1, listadas com suas posições e nomes Unicode.

Tabela 4.1 – Separadores de linha

Sequência Unicode	Equivalente na expressão regular	Nome	Quando usado
U+000D U+000A	‹\r\n›	Retorno de carro e alimentação de linha (CRLF)	Arquivos de textos Windows e MS-DOS
U+000A	‹\n›	Alimentação de linha (LF)	Arquivos de textos Unix,

			Linux e OS X
U+000B	‹\v›	Tabulação de linha (tabulação vertical ou VT)	(Raro)
U+000C	‹\f›	Alimentação de formulário (FF)	(Raro)
U+000D	‹\r›	Retorno de carro (CR)	Arquivos de textos Mac OS
U+0085	‹\x85›	Próxima linha (NEL)	Arquivos de textos IBM mainframe (Raro)
U+2028	‹\u2028› ou ‹\x{2028}›	Separador de linha	(Raro)
U+2029	‹\u2029› ou ‹\x{2029}›	Separador de Parágrafo	(Raro)

Veja também:

Receita 4.9.

4.11 Validar respostas afirmativas

Problema

Você precisa verificar uma opção de configuração ou resposta de linha de comando para um valor positivo. Também gostaria de dar maior flexibilidade nas respostas aceitas, de modo que `true`, `t`, `yes`, `y`, `okay`, `ok` e `1` sejam aceitos em qualquer combinação de letras, maiúsculas e minúsculas.

Solução

Usar uma expressão regular que combine todas as formas aceitas nos permite realizar a verificação com um teste simples.

Expressão regular

```
^(?:1|t(?:rue)?|y(?:es)?|ok(?:ay)?)$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

JavaScript

```
var yes = /^(?:1|t(?:rue)?|y(?:es)?|ok(?:ay)?)$/i;
if (yes.test(subject)) {
  alert("Yes");
} else {
```

```
  alert("No");
}
```

Outras linguagens de programação

Veja as receitas 3.4 e 3.5, para ajudá-lo na implementação desta expressão regular com outras linguagens de programação.

Discussão

O desmembramento a seguir mostra as partes individuais da expressão regular. Combinações de tokens, que sejam fáceis de ler em conjunto, são mostrados na mesma linha:

```
^ # Declara a posição no início da string.
(?: # Agrupa, mas não captura...
  1 # Corresponde a um literal "1".
 | # ou...
  t(?:rue)? # Corresponde a "t", opcionalmente seguido por "rue".
 | # ou...
  y(?:es)? # Corresponde a "y", opcionalmente seguido por "es".
 | # ou...
  ok(?:ay)? # Corresponde a "ok", opcionalmente seguido por "ay".
) # Finaliza o grupo de não-captura.
$ # Declara a posição no final da string.
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Esta expressão regular é, essencialmente, um teste simples para um dos sete valores literais, sem distinção de maiúsculas e minúsculas. Ela poderia ser escrita de várias maneiras. Por exemplo, ‹^(?:[1ty]|true|yes|ok(?:ay)?)$› é uma abordagem igualmente boa. Simplesmente alternar entre os sete valores, como em ‹^(?:1|t|true|y|yes|ok|okay)$›, também funcionaria bem, embora, por motivos de desempenho, seja melhor reduzir a quantidade de alternâncias que utilizem o operador pipe ‹|›, em favor de classes de caracteres e sufixos opcionais (utilizando o quantificador ‹?›). Neste caso, a diferença de desempenho, provavelmente, não será mais do que alguns poucos microssegundos. Porém, é uma boa ideia ter

sempre em mente as questões de desempenho de expressão regular. Às vezes, a diferença entre essas abordagens pode surpreendê-lo.

Todos estes exemplos envolvem valores de correspondência em potencial com um grupo de não-captura, para limitar o alcance dos operadores de alternância. Se omitirmos o agrupamento e, em vez disso, utilizarmos algo como ‹^true|yes$›, o mecanismo de expressão regular vai procurar pelo “início da string seguido de ‘true’”, ou por “‘yes’ seguido pelo final da string”. ‹^(?:true|yes)$› diz ao mecanismo de expressão regular para encontrar o início da string e, em seguida, “true” ou “yes”, para, então, encontrar o final da string.

Veja também:

Receitas 5.2 e 5.3.

4.12 Validar números de Previdência Social

Problema

Você precisa verificar se alguém digitou um texto no formato de um número de previdência social válido.

Solução

Se você precisa, simplesmente, garantir que uma string siga o formato de números da previdência social, e que números inválidos sejam eliminados, a expressão regular a seguir fornece, para tanto, uma solução simples. Se você precisar de uma solução mais rigorosa que verifique, junto à administração da previdência social, se o número pertence a uma pessoa viva, consulte os links na seção “Veja também”, desta receita.

Expressão regular

```
^(?!000|666)(?:[0-6][0-9]{2}|7(?:[0-6][0-9]|7[0-2]))-↲
```

```
(?!00)[0-9]{2}-(?!0000)[0-9]{4}$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Python

```
if re.match(r"^(?!000|666)(?:[0-6][0-9]{2}|7(?:[0-6][0-9]|7[0-2]))-↲
(?!00)[0-9]{2}-(?!0000)[0-9]{4}$", sys.argv[1]):
  print "SSN is valid"
else:
  print "SSN is invalid"
```

Outras linguagens de programação

Veja a receita 3.5, para ajudá-lo na implementação desta expressão regular com outras linguagens de programação.

Discussão

Os números da previdência social, nos Estados Unidos, possuem nove dígitos, no formato AAA-GG-SSSS:

- Os três primeiros dígitos são atribuídos pela região geográfica, e são chamados de *números de área*. O número de área não pode ser 000 ou 666 e, no momento em que escrevo isto, nenhum número válido da previdência social contém um número de área acima de 772.
- Os dígitos quatro e cinco são chamados de *números de grupo*, e vão de 01 a 99.
- Os últimos quatro dígitos são *números seriais*, de 0001 a 9999.

Esta receita segue todas as regras listadas. Aqui está a expressão regular, desta vez explicada parte por parte:

```
^ # Declara a posição no início da string.
(?!000|666) # Declara que nem "000", nem "666", podem ser correspondidos
aqui.
(?: # Agrupa, mas não captura...
  [0-6] # Corresponde a um caractere no intervalo entre "0" e "6".
  [0-9]{2} # Corresponde a um dígito, exatamente duas vezes.
 | # ou...
  7 # Corresponde a um literal "7".
```

```
  (?: # Agrupa, mas não captura...
    [0-6] # Corresponde a um caractere no intervalo entre "0" e "6".
    [0-9] # Corresponde a um dígito.
   | # ou...
    7 # Corresponde a um literal "7".
    [0-2] # Corresponde a um caractere no intervalo entre "0" e "2".
  ) # Finaliza o grupo de não captura.
) # Finaliza o grupo de não captura.
- # Corresponde a um literal "-".
(?!00) # Declara que "00" não pode ser correspondido aqui.
[0-9]{2} # Corresponde a um dígito, exatamente duas vezes.
- # Corresponde a um literal "-".
(?!0000) # Declara que "0000" não pode ser correspondido aqui.
[0-9]{4} # Corresponde a um dígito, exatamente quatro vezes.
$ # Declara a posição no final da string.
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Além dos tokens ‹^› e ‹$›, que declaram a posição no início e no final da string, esta expressão regular pode ser dividida em três grupos de dígitos, separados por hífens. O primeiro grupo é o mais complexo. O segundo e o terceiro simplesmente correspondem a qualquer número de dois ou quatro dígitos, respectivamente, mas utilizam um lookahead negativo anterior, para excluir a possibilidade de corresponder a zeros em todas as posições.

O primeiro grupo de dígitos é muito mais complexo e mais difícil de ler, em comparação com os outros, porque ele corresponde a um intervalo numérico. Primeiro, ele usa o lookahead negativo, ‹(?!000|666)› para excluir os valores específicos “000” e “666”. Em seguida, vem a tarefa de eliminar qualquer número superior a 772.

Como as expressões regulares lidam com texto, em vez de números, nós temos de desmembrar o intervalo numérico, caractere por caractere. Primeiro, sabemos que é possível corresponder a qualquer número de três dígitos, de 0 a 6, porque o lookahead negativo anterior já descartou os números inválidos 000 e 666. Esta primeira parte é facilmente realizada usando um par de classes de caracteres e um quantificador: ‹[0-6][0-9]{2}›. Como precisamos

oferecer uma alternativa para os números que começam com 7, o padrão que acabamos de construir é colocado em um agrupamento, como ‹(?:[0-6][0-9]{2}|7)›, a fim de limitar o alcance do operador de alternância.

Números que comecem com 7 são permitidos somente se eles se situarem entre 700 e 772; assim, o próximo passo será dividir ainda mais qualquer número iniciado por 7, com base no segundo dígito. Se estiver entre 0 e 6, qualquer terceiro dígito será permitido. Se o segundo dígito for 7, o terceiro deve estar entre 0 e 2. Ao colocarmos essas regras para números que comecem com 7, obtemos ‹7(?:[0-6][0-9]|7[0-2])›, que corresponde ao número 7, seguido por uma das duas opções para o segundo e terceiro dígitos.

Finalmente, insira isso no agrupamento externo do primeiro conjunto de dígitos, e você obterá ‹(?:[0-6][0-9]{2}|7(?:[0-6][0-9]|7[0-2]))›. É isso aí. Você criou, com êxito, uma expressão regular que corresponde a um número de três dígitos entre 000 e 772.

Variações

Encontrar números de Previdência Social em documentos

Se estiver procurando por números da previdência social em um documento maior ou em uma string de entrada, substitua as âncoras ‹^› e ‹$› por extremidades de palavra. Mecanismos de expressão regular consideram todos os caracteres alfanuméricos e o underscore como caracteres de palavra.

```
\b(?!000|666)(?:[0-6][0-9]{2}|7(?:[0-6][0-9]|7[0-2]))-↲
(?!00)[0-9]{2}-(?!0000)[0-9]{4}\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Veja também

O site da Administração da Previdência Social, em

http://www.socialsecurity.gov, fornece respostas a perguntas comuns, bem como listas atualizadas sobre quais áreas e números de grupo foram atribuídos.

O Serviço de Verificação de Números da Previdência Social (SSNVS), em *http://www.socialsecurity.gov/employer/ssnv.htm*, oferece duas maneiras de verificação pela Internet, se os nomes e números de Previdência Social correspondem aos registros da Administração da Previdência Social.

Uma discussão mais aprofundada sobre correspondência de intervalos numéricos, incluindo exemplos de correspondência com um número variável de dígitos, poderá ser encontrada na receita 6.5.

4.13 Validando ISBNs

Problema

Você precisa verificar a validade de um Número de Livro, no Padrão Internacional (International Standard Book Number, ou ISBN), que pode estar no formato mais velho (ISBN-10), ou no formato atual (ISBN-13). Você quer permitir um identificador ISBN no início, além de fazer com que as partes ISBN possam ser, opcionalmente, separadas por hífens ou espaços. ISBN 978-0-596-52068-7, ISBN-13: 978-0-596-52068-7, 978 0 596 52068 7, 9780596520687, ISBN-10 0-596-52068-9 e 0-596-52068-9 são exemplos válidos de entrada.

Solução

Você não pode validar um ISBN utilizando somente uma expressão regular, porque o último dígito é calculado utilizando-se um algoritmo de checksum (soma verificadora). As expressões regulares, nesta seção, validam o formato ISBN, enquanto os exemplos de código subsequentes incluem uma verificação de validade para o último dígito.

Expressões regulares

ISBN-10:

```
^(?:ISBN(?:-10)?:?␣)?(?=[-0-9X␣]{13}$|[0-9X]{10}$)[0-9]{1,5}[-␣]?↲
(?:[0-9]+[-␣]?){2}[0-9X]$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

ISBN-13:

```
^(?:ISBN(?:-13)?:?␣)?(?=[-0-9␣]{17}$|[0-9]{13}$)97[89][-␣]?[0-9]{1,5}↲
[-␣]?(?:[0-9]+[-␣]?){2}[0-9]$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

ISBN-10 or ISBN-13:

```
^(?:ISBN(?:-1[03])?:?␣)?(?=[-0-9␣]{17}$|[-0-9X␣]{13}$|[0-9X]{10}$)↲
(?:97[89][-␣]?)?[0-9]{1,5}[-␣]?(?:[0-9]+[-␣]?){2}[0-9X]$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

JavaScript

```
// `regex` verifica o formato ISBN-10 ou ISBN-13
var regex = /^(?:ISBN(?:-1[03])?:? )?(?=[-0-9 ]{17}$|[-0-9X ]{13}$|↲
[0-9X]{10}$)(?:97[89][- ]?)?[0-9]{1,5}[- ]?(?:[0-9]+[- ]?){2}[0-9X]$/;
if (regex.test(subject)) {
  // Remova os dígitos que não façam parte do ISBN, então divida em um array
  var chars = subject.replace(/[^0-9X]/g, "").split("");
  // Remova o dígito ISBN final de `chars`, e atribua-o a `last`
  var last = chars.pop();
  var sum = 0;
  var digit = 10;
  var check;
  if (chars.length == 9) {
    // Calcule o dígito verificador ISBN-10
    for (var i = 0; i < chars.length; i++) {
        sum += digit * parseInt(chars[i], 10);
        digit -= 1;
    }
```

```
    check = 11 - (sum % 11);
    if (check == 10) {
        check = "X";
    } else if (check == 11) {
        check = "0";
    }
  } else {
    // Calcule o dígito verificador ISBN-13
    for (var i = 0; i < chars.length; i++) {
        sum += (i % 2 * 2 + 1) * parseInt(chars[i], 10);
    }
    check = 10 - (sum % 10);
    if (check == 10) {
        check = "0";
    }
  }
  if (check == last) {
    alert("Valid ISBN");
  } else {
    alert("Invalid ISBN check digit");
    }
  } else {
    alert("Invalid ISBN");
}
```

Python

```
import re
import sys
# `regex` verifica o formato ISBN-10 ou ISBN-13
regex = re.compile("^(?:ISBN(?:-1[03])?:? )?(?=[-0-9 ]{17}$|↲
[-0-9X ]{13}$|[0-9X]{10}$)(?:97[89][- ]?)?[0-9]{1,5}[- ]?↲
(?:[0-9]+[- ]?){2}[0-9X]$")
subject = sys.argv[1]
if regex.search(subject):
  # Remova os dígitos que não façam parte do ISBN, então divida em um array
  chars = re.sub("[^0-9X]", "", subject).split("")
  # Remova o dígito ISBN final de `chars`, e atribua-o a `last`
  last = chars.pop()
  if len(chars) == 9:
    # Calcule o dígito verificador ISBN-10
    val = sum((x + 2) * int(y) for x,y in enumerate(reversed(chars)))
```

```
    check = 11 - (val % 11)
    if check == 10:
        check = "X"
    elif check == 11:
        check = "0"
  else:
    # Calcule o dígito verificador ISBN-13
    val = sum((x % 2 * 2 + 1) * int(y) for x,y in enumerate(chars))
    check = 10 - (val % 10)
    if check == 10:
        check = "0"
  if (str(check) == last):
    print "Valid ISBN"
  else:
    print "Invalid ISBN check digit"
else:
  print "Invalid ISBN"
```

Outras linguagens de programação

Veja a receita 3.5, para ajudá-lo na implementação desta expressão regular com outras linguagens de programação.

Discussão

O ISBN é um identificador único para livros comerciais e produtos similares. O formato ISBN de 10 dígitos foi publicado como um padrão internacional, ISO 2108, em 1970. Todos os ISBNs atribuídos desde 1 de janeiro de 2007 possuem 13 dígitos.

Números ISBN-10 e ISBN-13 são divididos em quatro ou cinco elementos, respectivamente. Três dos elementos possuem comprimento variável, os elementos restantes (um ou dois, dependendo do formato) são de comprimento fixo. Todas as cinco partes, geralmente, são separadas com hífens ou espaços. Segue uma breve descrição de cada elemento:

- ISBNs de 13 dígitos começam com o prefixo 978 ou 979.
- O *identificador de grupo* verifica o grupo de países que compartilham o mesmo idioma. Ele varia de um a cinco dígitos.
- O *identificador de editora* varia em comprimento, e é atribuído

pela agência ISBN nacional.

- O *identificador de título* também varia em comprimento, e é selecionado pela editora.
- O caractere final é chamado de *dígito verificador*, sendo calculado mediante um algoritmo de checksum. Um dígito verificador ISBN-10 pode ser tanto um número de 0 a 9, ou a letra X (numeral romano para 10), enquanto um dígito verificador ISBN-13 varia de 0 a 9. Os caracteres permitidos são diferentes, pois os dois tipos ISBN usam algoritmos checksum diferentes.

As partes da expressão regular “ISBN-10 ou ISBN-13” são mostradas no desmembramento, a seguir.

Como esta expressão regular foi escrita no modo de espaçamento livre, os caracteres literais de espaço foram escapados com barras invertidas. O Java exige que mesmo espaços dentro das classes de caracteres sejam escapados no modo de espaçamento livre:

```
^ # Declara a posição no início da string.
(?: # Agrupa, mas não captura...
  ISBN # Corresponde ao texto "ISBN".
  (?:-1[03])? # Opcionalmente corresponde ao texto "-10" ou "-13".
  :? # Opcionalmente corresponde a um literal ":".
  \ # Corresponde a um caractere de espaço (escapado).
)? # Repete o grupo entre zero e uma vez.
(?= # Declara que o seguinte pode ser correspondido aqui...
  [-0-9\ ]{17}$ # Corresponde a 17 hífens, dígitos e espaços, seguidos pelo final
 | # da string. Ou...
  [-0-9X\ ]{13}$ # Corresponde a 13 hífens, dígitos, Xs e espaços, seguidos pelo
 | # final da string. Ou...
  [0-9X]{10}$ # Corresponde a 10 dígitos e Xs, seguidos pelo final da string.
) # Finaliza o lookahead positivo.
(?: # Agrupa, mas não captura...
  97[89] # Corresponde ao texto "978" ou "979".
  [-\ ]? # Opcionalmente corresponde a um hífen ou espaço.
)? # Repete o grupo entre zero e uma vez.
[0-9]{1,5} # Corresponde a um dígito entre uma e cinco vezes.
[-\ ]? # Opcionalmente corresponde a um hífen ou espaço.
(?: # Agrupa, mas não captura...
  [0-9]+ # Corresponde a um dígito entre uma vez e um número ilimitado de
vezes.
```

```
  [-\ ]? # Opcionalmente corresponde a um hífen ou espaço.
)){2} # Repete o grupo exatamente duas vezes.
[0-9X] # Corresponde a um dígito ou "X".
$ # Declara a posição no final da string.
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

O ‹(?:ISBN(?:-1[03])?:?)?›, no início, possui três elementos opcionais que permitem a correspondência com qualquer uma das sete strings (todas, exceto a opção da string vazia, incluem um caractere de espaço no final):

- ISBN
- ISBN-10
- ISBN-13
- ISBN:
- ISBN-10:
- ISBN-13:
- *A string vazia (sem prefixo)*

Em seguida, o lookahead positivo ‹(?=[-0-9]{17}$|[-0-9X]{13}$|[0-9X]{10}$)› impõe uma das três opções (separadas pelo operador de alternância ‹|›) para o comprimento e o conjunto de caracteres do restante da correspondência. Todas as três opções (mostradas a seguir) terminam com a âncora ‹$›, que garante que não haja qualquer texto à direita que não se encaixe em um dos padrões:

‹[-0-9]{17}$›

Permite um ISBN-13 com quatro separadores (17 caracteres no total).

‹[-0-9X]{13}$›

Permite um ISBN-13 sem separadores, ou um ISBN-10 com três separadores (13 caracteres no total).

‹[0-9X]{10}$›

Permite um ISBN-10 sem separadores (10 caracteres no total).

Depois que o lookahead positivo valida o comprimento e o conjunto

de caracteres, podemos corresponder aos elementos ISBN individuais, sem nos preocuparmos com seus comprimentos combinados. ‹(?:97[89][-□]?)?› corresponde ao prefixo “978” ou “979”, exigido pelo ISBN-13. O grupo de não-captura é opcional, pois ele não corresponderá dentro de uma string de assunto ISBN-10. ‹[0-9]{1,5}[-□]?› corresponde ao identificador de grupo, de um a cinco dígitos, e a um separador opcional. ‹(?:[0-9]+[-□]?){2}› corresponde aos identificadores de editora e título, que possuem comprimento variável, juntamente com seus separadores opcionais. Finalmente, ‹[0-9X]$› corresponde ao dígito verificador no final da string.

Apesar de uma expressão regular poder verificar que o dígito final utiliza um caractere válido (dígito ou X), ela não pode determinar se ele está correto em relação checksum do ISBN. Um dos dois algoritmos de checksum (determinado pelo fato de você estar trabalhando com um número ISBN-10 ou ISBN-13) é utilizado para fornecer algum nível de garantia de que os dígitos ISBN não foram, acidentalmente, transpostos ou digitados incorretamente. Os códigos de exemplo para JavaScript e Python, mostrados anteriormente, implementaram ambos os algoritmos. As seções seguintes descrevem as regras de cada checksum, a fim de ajudá-lo na implementação destes algoritmos com outras linguagens de programação.

Checksum do ISBN-10

O dígito verificador, para um número ISBN-10, varia de 0 a 10 (com o número romano X utilizado no lugar do 10). É computado desta maneira:

1. Multiplique cada um dos 9 primeiros dígitos por um número, na sequência decrescente, de 10 a 2, e some os resultados.
2. Divida a soma por 11.
3. Subtraia o resto (não o quociente) de 11.
4. Se o resultado for 11, use o número 0; se for 10, use a letra X. Vejamos um exemplo de como derivar o dígito de verificação

ISBN-10 para 0-596-52068-?:

Passo 1:
sum = 10×0 + 9×5 + 8×9 + 7×6 + 6×5 + 5×2 + 4×0 + 3×6 + 2×8
= 0 + 45 + 72 + 42 + 30 + 10 + 0 + 18 + 16
= 233
Passo 2:
233 ÷ 11 = 21, remainder 2
Passo 3:
11 − 2 = 9
Passo 4:
9 [Nào há a necessidade de substituição]

O dígito verificador é 9; então, a sequência completa é ISBN 0-596-52068-9.

Checksum do ISBN-13

Um dígito verificador ISBN-13 varia de 0 a 9, e é computado utilizando passos similares.

1. Multiplique cada um dos 12 primeiros dígitos por 1 ou 3 – alternadamente, conforme movimento da esquerda para a direita – e some os resultados.
2. Divida a soma por 10.
3. Subtraia o resto (não o quociente) de 10.
4. Se o resultado for 10, use o número 0.

Por exemplo, o dígito de verificação ISBN-13 para 978-0-596-52068-? é calculado da seguinte forma:

Passo 1:
sum = 1×9 + 3×7 + 1×8 + 3×0 + 1×5 + 3×9 + 1×6 + 3×5 + 1×2 + 3×0 + 1×6 + 3×8
= 9 + 21 + 8 + 0 + 5 + 27 + 6 + 15 + 2 + 0 + 6 + 24
= 123
Passo 2:
123 ÷ 10 = 12, sobra 3
Passo 3:
10 − 3 = 7
Passo 4:
7 [Nào há a necessidade de substituição]

O dígito verificador é 7, e a sequência completa é ISBN 978-0-596-

52068-7.

Variações

Encontrar ISBNs em documentos

Esta versão da expressão regular “ISBN-10 ou ISBN-13” utiliza extremidades de palavra, ao invés de âncoras, para ajudá-lo a encontrar ISBNs dentro de textos mais longos, assegurando que eles sejam válidos. O identificador “ISBN” também se tornou uma string obrigatória nesta versão, por duas razões. Em primeiro lugar, exigi-lo ajudará a eliminar falsos positivos (sem ele, a expressão regular pode, potencialmente, corresponder a qualquer um dos 10 ou 13 dígitos); em segundo lugar, ISBNs devem, oficialmente, usar este identificador quando impressos:

```
\bISBN(?:-1[03])?:?□(?=[-0-9□]{17}$|[-0-9X□]{13}$|[0-9X]{10}$)↲
(?:97[89][-□]?)?[0-9]{1,5}[-□]?(?:[0-9]+[-□]?){2}[0-9X]\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Eliminar identificadores ISBN incorretos

Uma limitação das expressões regulares anteriores é que elas permitem uma correspondência a um número ISBN-10 precedido pelo identificador “ISBN-13” e vice-versa. A expressão regular a seguir usa condicionais (veja receita 2.17) para garantir que um identificador “ISBN-10” ou “ISBN-13” seja seguido pelo tipo ISBN adequado. Ela permite ambos os números ISBN-10 e ISBN-13, quando o tipo não for especificado explicitamente. Tal expressão regular é um exagero na maioria das circunstâncias, porque o mesmo resultado poderia ser alcançado com a utilização das expressões regulares específicas ISBN-10 e ISBN-13, mostradas anteriormente, uma de cada vez. Ela está incluída aqui apenas para demonstrar um uso interessante das expressões regulares:

```
^
(?:ISBN(-1(?:(0)|3))?:?\ )?
```

```
(?(1)
  (?(2)
    (?=[-0-9X ]{13}$|[0-9X]{10}$)
    [0-9]{1,5}[- ]?(?:[0-9]+[- ]?){2}[0-9X]$
   |
    (?=[-0-9 ]{17}$|[0-9]{13}$)
    97[89][- ]?[0-9]{1,5}[- ]?(?:[0-9]+[- ]?){2}[0-9]$
  )
 |
  (?=[-0-9 ]{17}$|[-0-9X ]{13}$|[0-9X]{10}$)
  (?:97[89][- ]?)?[0-9]{1,5}[- ]?(?:[0-9]+[- ]?){2}[0-9X]$
)
$
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, PCRE, Perl, Python

Veja também:

A versão mais atualizada do manual do usuário ISBN pode ser encontrada no site da Agência Internacional ISBN em *http://www.isbn-international.org*.

A Lista Numérica de Grupos de Identificadores oficial, em *http://www.isbn-international.org/en/identifiers/allidentifiers.html*, pode ajudá-lo na identificação do país de origem ou da área de um livro, tendo por base os primeiros 1 a 5 dígitos de seu ISBN.

4.14 Validar códigos postais

Problema

Você precisa validar um código postal (norte-americano), permitindo ambos os formatos de 5 e 9 dígitos (*ZIP + 4*). A expressão regular deverá corresponder a 12345 e 12345-6789, mas não a 1234, 123456, 123456789 ou 1234-56789.

Solução

Expressão regular

```
^[0-9]{5}(?:-[0-9]{4})?$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

VB.NET

```
If Regex.IsMatch(subjectString, "^[0-9]{5}(?:-[0-9]{4})?$") Then
  Console.WriteLine("Valid ZIP code")
Else
  Console.WriteLine("Invalid ZIP code")
End If
```

Outras linguagens de programação

Veja a receita 3.5, para ajudá-lo na implementação desta expressão regular com outras linguagens de programação.

Discussão

Segue um desmembramento da expressão regular de código postal:

```
^ # Declara a posição no início da string.
[0-9]{5} # Corresponde a um dígito exatamente cinco vezes.
(?: # Agrupa, mas não captura...
  - # Corresponde a um literal "-".
  [0-9]{4} # Corresponde a um dígito exatamente quatro vezes.
) # Finaliza o grupo de não-captura.
  ? # Repete o grupo anterior entre zero e uma vez.
$ # Declara a posição no final da string.
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Esta expressão regular é bem simples, portanto não há muito a acrescentar. Uma mudança simples, que permitiria encontrar códigos postais dentro de uma string de entrada mais longa, seria substituir as âncoras ‹^› e ‹$› por extremidades de palavra, resultando em ‹\b[0-9]{5}(?:-[0-9]{4})?\b›.

Veja também:

Receitas 4.15, 4.16 e 4.17.

4.15 Validar códigos postais canadenses

Problema

Você deseja verificar se uma string é um código postal canadense.

Solução

^(?!.*[DFIOQU])[A-VXY][0-9][A-Z] [0-9][A-Z][0-9]$

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

O lookahead negativo, no início desta expressão regular, impede que os caracteres D, F, I, O, Q ou U apareçam em qualquer parte da string de assunto. A classe de caracteres ‹[A-VXY]› ainda impede que W ou Z apareçam como primeiro caractere. Fora essas duas exceções, os códigos postais canadenses utilizam uma sequência alternada de seis caracteres alfanuméricos, com um espaço no meio. Por exemplo, a expressão regular corresponderá a K1A 0B1, código postal canadense para a sede postal de Ottawa.

Veja também:

Receitas 4.14, 4.16 e 4.17.

4.16 Validar códigos postais do Reino Unido

Problema

Você precisa de uma expressão regular que corresponda a um código postal do Reino Unido.

Solução

^[A-Z]{1,2}[0-9R][0-9A-Z]? [0-9][ABD-HJLNP-UW-Z]{2}$

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Os códigos postais do Reino Unido (ou *postcodes*, como são chamados) são compostos de cinco a sete caracteres alfanuméricos, separados por um espaço. As regras, que abrangem quais caracteres podem aparecer em determinadas posições, são bastante complicadas e cheias de exceções. Esta expressão regular prende-se às regras básicas.

Veja também:

British Standard BS7666, disponível em *http://www.govtalk.gov.uk/gdsc/html/frames/PostCode.htm*, descreve as regras dos códigos postais do Reino Unido.

Receitas 4.14, 4.15 e 4.17.

4.17 Encontrar endereços com caixas postais

Problema

Você deseja capturar endereços que contenham uma caixa postal (P.O. box), para avisar aos usuários que suas informações de envio devem conter um endereço de uma rua.

Solução

Expressão regular

^(?:Post (?:Office)?|P[.]?O\.?)?Box\b

Opções Regex: Não diferenciar maiúsculas de minúsculas, ^ e $ correspondem em quebras de linha

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

C#

```
Regex regexObj = new Regex(
  @"^(?:Post (?:Office )?|P[. ]?O\.? )?Box\b",
  RegexOptions.IgnoreCase | RegexOptions.Multiline
);
if (regexObj.IsMatch(subjectString) {
  Console.WriteLine("The value does not appear to be a street address");
} else {
  Console.WriteLine("Good to go");
}
```

Outras linguagens de programação

Veja a receita 3.5, para ajudá-lo na implementação desta expressão regular com outras linguagens de programação.

Discussão

A explicação a seguir está escrita no modo de espaçamento livre, assim, os caracteres de espaço significativos foram escapados com uma barra invertida:

```
^ # Declara a posição no início de uma linha.
(?: # Agrupa, mas não captura...
  Post\ # Corresponde a "Post ".
  (?:Office\ )? # Opcionalmente corresponde a "Office ".
 | # ou...
  P[.\ ]? # Corresponde a "P" e a um ponto ou caractere de espaço opcional.
  O\.?\ # Corresponde a "O", um ponto opcional, e um caractere de espaço.
)? # Repete o grupo entre zero e uma vez.
Box # Corresponde a "Box".
\b # Declara a posição em uma extremidade de palavra.
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, ^ e $ correspondem em quebras de linha

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Esta expressão regular corresponde a todas as strings de exemplo a seguir, quando elas aparecem no início de uma linha:

- Post Office Box
- post box
- P.O. box

- P O Box
- Po. box
- PO Box
- Box

Apesar das precauções aqui tomadas, você pode encontrar alguns falsos positivos, ou falsos negativos, porque muitas pessoas estão acostumadas com carteiros bem flexíveis, capazes de decifrar endereços. Para mitigar este risco, é melhor informar, logo no início, que caixas postais não são permitidas. Se você obtiver uma correspondência usando essa expressão regular, avise o usuário que, aparentemente, ele informou uma caixa postal, além de dar a opção de manter a entrada.

Veja também:

Receitas 4.14, 4.15 e 4.16.

4.18 Reformatar nomes no formato “Nome Sobrenome” para “Sobrenome, Nome”

Problema

Você deseja converter nomes de pessoas do formato “Nome Sobrenome” para o formato “Sobrenome, Nome”, para uso em uma lista em ordem alfabética. Além disso, você deseja considerar outras partes do nome, para que possa, por exemplo, converter “Nome NomesDoMeio Partículas Sobrenome Sufixo” para “Sobrenome, Nome NomesDoMeio Partículas Sufixo”.

Solução

Infelizmente, não é possível analisar nomes de forma confiável usando uma expressão regular. As expressões regulares são rígidas, enquanto nomes são tão flexíveis que até nós, seres

humanos, nos confundimos. Determinar a estrutura de um nome, ou como ele deveria ser listado em ordem alfabética, muitas vezes exige considerar convenções tradicionais e nacionais, ou até mesmo preferências pessoais. No entanto, se você estiver disposto a fazer certas suposições sobre seus dados, e puder lidar com um nível moderado de erro, uma expressão regular pode fornecer uma solução rápida.

A seguinte expressão regular foi mantida simples deliberadamente. Ele não tenta levar em conta casos extremos.

Expressão regular

```
^(.+?) ([^\s,]+)(, ?(?:[JS]r\.?|III?|IV))?$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Substituição

```
$2, $1$3
```

Sabores de texto de substituição: .NET, Java, JavaScript, Perl, PHP

```
\2, \1\3
```

Sabores de texto de substituição: Python, Ruby

JavaScript

```
function formatName (name) {
  return name.replace(/^(.+?) ([^\s,]+)(,? (?:[JS]r\.?|III?|IV))?$/i, "$2, $1$3");
}
```

Outras linguagens de programação

Veja a receita 3.5, para ajudá-lo na implementação desta expressão regular com outras linguagens de programação.

Discussão

Primeiro, vamos dar uma olhada nesta expressão regular, parte por parte. Comentários mais aprofundados serão fornecidos mais tarde, para ajudar a explicar quais partes de um nome estão sendo

correspondidas pelos vários segmentos da expressão regular. Como a expressão regular está escrita no modo de espaçamento livre, os caracteres literais de espaço foram escapados com barras invertidas:

```
^ # Declara a posição no início da string.
( # Captura a correspondência englobada na retrorreferência 1...
  .+? # Corresponde a um ou mais caracteres, um mínimo de vezes possível.
) # Finaliza o grupo de captura.
\ # Corresponde um caractere literal de espaço.
( # Captura a correspondência englobada na retrorreferência 2...
  [^\s,]+ # Corresponde a um ou mais caracteres que não sejam espaços em branco
          # ou vírgulas.
) # Finaliza o grupo de captura.
( # Captura a correspondência englobada na retrorreferência 3...
  ,?\ # Corresponde a ", " ou " ".
  (?: # Agrupa, mas não captura...
    [JS]r\.? # Corresponde a "Jr", "Jr.", "Sr" ou "Sr.".
   | # ou...
    III? # Corresponde a "II" ou "III".
   | # ou...
    IV # Corresponde a "IV".
  ) # Finaliza o grupo de não-captura.
)? # Repete o grupo entre zero e uma vez.
$ # Declara a posição no final da string.
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Esta expressão regular faz as seguintes suposições sobre o dado de assunto:

- Ele contém pelo menos um nome e um sobrenome (outras partes do nome são opcionais).
- O nome é listado antes do sobrenome.
- Se o nome contiver um sufixo, será um dos valores “Jr”, “Jr.”, “Sr”, “Sr.”, “II”, “III” ou “IV”, com uma vírgula precedente opcional.

Algumas questões a mais, para considerar:

- A expressão regular não consegue identificar sobrenomes

compostos que não usem hífens. Por exemplo, Sacha Baron Cohen seria substituído por Cohen, Sacha Baron, em vez da listagem correta, Baron Cohen, Sacha.

- Ela não mantém as partículas à frente do nome de família; embora isso seja ocasionalmente usado por convenção ou preferência pessoal (por exemplo, a listagem alfabética correta de “Charles de Gaulle” é “de Gaulle, Charles” de acordo com o *Manual de Estilos de Chicago*, 15ª Edição, que contradiz o *Dicionário Biográfico Merriam-Webster*, em relação a este nome em particular).
- Por causa das âncoras ‹^› e ‹$›, que vinculam a correspondência ao início e ao final da string, nenhuma substituição poderá ser feita se o texto de assunto total não se encaixar no padrão. Portanto, se nenhuma correspondência for encontrada (por exemplo, se o texto de assunto possuir somente um nome), o nome ficará inalterado.

Quanto ao modo de funcionamento da expressão regular, ela utiliza três grupos de captura para dividir o nome. Os pedaços são reagrupados na ordem desejada, por meio de retrorreferências no texto de substituição. O grupo de captura 1 utiliza o padrão máximo de flexibilidade, ‹.+?›, para pegar o primeiro nome, junto com qualquer quantidade de nomes do meio e partículas de sobrenome, como o alemão “von”, ou o “de”, do francês, português e espanhol. Estas partes do nome são tratadas em conjunto, porque elas são listadas sequencialmente na saída. Juntar o primeiro e o segundo nomes também ajuda a evitar erros, pois a expressão regular não pode distinguir entre um nome composto, como “Mary Lou” ou “Norma Jeane”, e um primeiro nome mais o nome do meio. Mesmo nós não conseguimos fazer a distinção exata apenas pelo exame visual.

O grupo de captura 2 corresponde ao último nome utilizando ‹[^\s,]+›. Tal como o ponto utilizado no grupo de captura 1, a flexibilidade desta classe de caracteres permite corresponder aos caracteres

acentuados e a quaisquer outros caracteres não-latinos. O grupo de captura 3 corresponde a um sufixo opcional, como “Jr.” ou “III”, a partir de uma lista predefinida de valores possíveis. O sufixo é tratado separadamente do sobrenome, porque ele deve continuar a aparecer no final do nome reformatado.

Vamos voltar, por um minuto, para o grupo de captura 1. Por que o ponto dentro do grupo 1 foi seguido pelo quantificador preguiçoso ‹+?›, se a classe de caracteres, no grupo 2, foi seguida pelo quantificador mesquinho ‹+›? Porque se o grupo 1 (que lida com um número variável de elementos e, portanto, precisa ir tão longe quanto puder dentro do nome) usasse um quantificador mesquinho, o grupo de captura 3 (que tenta corresponder a um sufixo) não teria chance de participar da correspondência. O ponto do grupo 1 corresponderia até o final da string e, como o grupo de captura 3 é opcional, o mecanismo de expressão regular só iria retroceder o suficiente para encontrar uma correspondência para o grupo 2, antes de declarar sucesso. O grupo de captura 2 pode utilizar um quantificador mesquinho, porque sua classe de caracteres mais restritiva só permite corresponder a um nome.

A tabela 4.2 mostra alguns exemplos de como certos nomes seriam formatados usando esta expressão regular e o texto de substituição.

Tabela 4.2 – Nomes formatados

Entrada	Saída
Robert Downey, Jr.	Downey, Robert, Jr.
John F. Kennedy	Kennedy, John F.
Scarlett O’Hara	O’Hara, Scarlett
Pepé Le Pew	Pew, Pepé Le
J.R.R. Tolkien	Tolkien, J.R.R.
Catherine Zeta-Jones	Zeta-Jones, Catherine

Variações

Listar partículas de sobrenomes no início do nome

Um segmento adicional, na expressão regular a seguir, permite exibir partículas de sobrenome, selecionadas a partir de uma lista predefinida, na frente do sobrenome. Especificamente, esta expressão regular leva em conta os valores “De”, “Du”, “La”, “Le”, “St”, “St.”, “Ste”, “Ste.”, “Van” e “Von”. Qualquer quantidade destes valores é permitida em sequência (por exemplo, “de la”):

```
^(.+?) ((?:(?:D[eu]|L[ae]|Ste?\.?|V[ao]n) )*[^\s,]+)↵
(,? (?:[JS]r\.?|III?|IV))?$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
$2, $1$3
```

Sabores de texto de substituição: .NET, Java, JavaScript, Perl, PHP

```
\2, \1\3
```

Sabores de texto de substituição: Python, Ruby

4.19 Validar números de cartão de crédito

Problema

Suponha que lhe foi dado o trabalho de implementar um formulário de pedido para uma empresa que aceita pagamentos por cartão de crédito. Como a administradora do cartão de crédito cobra por tentativa de transação, incluindo as que falham, você quer usar uma expressão regular para excluir os números de cartões de crédito obviamente inválidos.

Ao fazer isso, você também melhorará a experiência do usuário. Uma expressão regular pode, instantaneamente, detectar erros óbvios, assim que o cliente finalizar o preenchimento do campo no formulário web. Em comparação, uma viagem de ida e volta até a administradora do cartão de crédito pode levar, facilmente, de 10 a 30 segundos.

Solução

Remover espaços e hífens

Recupere o número do cartão de crédito inserido pelo cliente, e o armazene em uma variável. Antes de realizar a verificação de um número válido, faça uma pesquisa-e-substituição, para remover espaços e hífens. Substitua esta expressão regular, globalmente, por um texto de substituição vazio:

```
[ -]
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

A receita 3.14 mostra como realizar esta substituição inicial.

Validar o número

Com os espaços e hífens retirados da entrada de dados, a expressão regular verifica se o número do cartão de crédito utiliza o formato de qualquer uma das seis principais empresas de cartões de crédito. Ela utiliza captura nomeada, para detectar o tipo de cartão de crédito que o cliente possui:

```
^(?:
(?<visa>4[0-9]{12}(?:[0-9]{3})?) |
(?<mastercard>5[1-5][0-9]{14}) |
(?<discover>6(?:011|5[0-9][0-9])[0-9]{12}) |
(?<amex>3[47][0-9]{13}) |
(?<diners>3(?:0[0-5]|[68][0-9])[0-9]{11}) |
(?<jcb>(?:2131|1800|35\d{3})\d{11})
)$
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
^(?:
(?P<visa>4[0-9]{12}(?:[0-9]{3})?) |
(?P<mastercard>5[1-5][0-9]{14}) |
(?P<discover>6(?:011|5[0-9][0-9])[0-9]{12}) |
(?P<amex>3[47][0-9]{13}) |
(?P<diners>3(?:0[0-5]|[68][0-9])[0-9]{11}) |
(?P<jcb>(?:2131|1800|35\d{3})\d{11})
```

```
)$
```

Opções Regex: Espaçamento livre

Sabores Regex: PCRE, Python

Java, Perl 5.6, Perl 5.8 e Ruby 1.8 não suportam capturas nomeadas. Em vez disso, você pode usar capturas numeradas. O grupo 1 capturará os cartões Visa; o grupo 2 capturará cartões MasterCard e assim sucessivamente, até o grupo 6 para JCB:

```
^(?:
(4[0-9]{12}(?:[0-9]{3})?) | # Visa
(5[1-5][0-9]{14}) | # MasterCard
(6(?:011|5[0-9][0-9])[0-9]{12}) | # Discover
(3[47][0-9]{13}) | # AMEX
(3(?:0[0-5]|[68][0-9])[0-9]{11}) | # Diners Club
((?:2131|1800|35\d{3})\d{11}) # JCB
)$
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

O JavaScript não suporta espaçamento livre. Removendo espaços em branco e comentários, obtemos:

```
^(?:(4[0-9]{12}(?:[0-9]{3})?)|(5[1-5][0-9]{14})|↲
(6(?:011|5[0-9][0-9])[0-9]{12})|(3[47][0-9]{13})|↲
(3(?:0[0-5]|[68][0-9])[0-9]{11})|((?:2131|1800|35\d{3})\d{11}))$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Se você não precisa determinar qual o tipo de cartão, pode remover os grupos de captura desnecessários:

```
^(?:
4[0-9]{12}(?:[0-9]{3})? | # Visa
5[1-5][0-9]{14} | # MasterCard
6(?:011|5[0-9][0-9])[0-9]{12} | # Discover
3[47][0-9]{13} | # AMEX
3(?:0[0-5]|[68][0-9])[0-9]{11} | # Diners Club
(?:2131|1800|35\d{3})\d{11} # JCB
)$
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Ou, no caso do JavaScript:

```
^(?:4[0-9]{12}(?:[0-9]{3})?|5[1-5][0-9]{14}|6(?:011|5[0-9][0-9])[0-9]{12}|↲
3[47][0-9]{13}|3(?:0[0-5]|[68][0-9])[0-9]{11}|(?:2131|1800|35\d{3})\d{11})$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Siga a receita 3.6, para adicionar essa expressão regular a seu formulário de pedido e validar o número do cartão. Se você for usar diferentes administradoras para diferentes cartões, ou se deseja apenas manter algumas estatísticas, pode usar a receita 3.9, para verificar qual grupo de captura, nomeado ou numerado, contém a correspondência. Isso vai lhe dizer o tipo de cartão de crédito que o cliente possui.

Exemplo de página web com JavaScript

```
<html>
<head>
<title>Teste de Cartão de Crédito</title>
</head>
<body>
<h1>Teste de Cartão de Crédito</h1>
<form>
<p>Por favor, entre com o número de seu cartão de crédito:</p>
<p><input type="text" size="20" name="cardnumber"
  onkeyup="validatecardnumber(this.value)"></p>
<p id="notice">(nenhum número de cartão digitado)</p>
</form>
<script>
function validatecardnumber(cardnumber) {
  // Remove espaços e hífens
  cardnumber = cardnumber.replace(/[ -]/g, '');
  // Veja se o cartão é válido
  // A expressão regular vai capturar o número em um dos grupos de captura
  var match = /^(?:(4[0-9]{12}(?:[0-9]{3})?)|(5[1-5][0-9]{14})|↲
(6(?:011|5[0-9][0-9])[0-9]{12})|(3[47][0-9]{13})|(3(?:0[0-5]|[68][0-9])↲
[0-9]{11})|((?:2131|1800|35\d{3})\d{11}))$/.exec(cardnumber);
  if (match) {
    // Lista de tipos de cartões, na mesma ordem dos grupos de captura da regex
```

```
    var types = ['Visa', 'MasterCard', 'Discover', 'American Express',
                 'Diners Club', 'JCB'];
    // Encontre o grupo de captura que correspondeu
    // Pule o elemento zero do array correspondido (a correspondência global)
    for (var i = 1; i < match.length; i++) {
      if (match[i]) {
        // Exiba o tipo de cartão para aquele grupo
        document.getElementById('notice').innerHTML = types[i - 1];
        break;
      }
    }
  } else {
    document.getElementById('notice').innerHTML = '(número de cartão inválido)';
  }
}
</script>
</body>
</html>
```

Discussão

Remover espaços e hífens

Em um cartão de crédito, os dígitos em relevo do número do cartão são, geralmente, colocados em grupos de quatro. Isso faz com que o número do cartão seja mais fácil de ler. Naturalmente, muitas pessoas vão tentar inserir o número do cartão dessa mesma forma nos pedidos, incluindo os espaços.

Escrever uma expressão regular que valide o número do cartão, permitindo espaços, hífens e outros adereços, é muito mais difícil do que escrever uma expressão regular que permita apenas dígitos. Assim, se você não quiser irritar o cliente com a redigitação do número do cartão sem espaços ou hífens, faça uma rápida pesquisa-e-substituição, para retirá-los antes de validar o número do cartão e enviá-lo para a administradora.

A expressão regular ‹[-]› corresponde a um caractere que seja um espaço ou um hífen. Substituir todas as correspondências desta expressão regular por nada, efetivamente elimina todos os espaços

e hífens.

Números de cartão de crédito só podem conter dígitos. Em vez de usar ‹[-]› para remover apenas espaços e hífens, você poderia utilizar a classe de caracteres abreviada ‹\D› para retirar todos os caracteres que não sejam dígitos.

Validar o número

Cada empresa de cartão de crédito usa um formato de número diferente. Vamos explorar essa diferença, para permitir aos usuários inserir um número, sem especificar a empresa; esta poderá ser reconhecida a partir do número. O formato para cada empresa é:

Visa
: 13 ou 16 dígitos, começando com 4.

MasterCard
: 16 dígitos, começando com 51, e indo a 55.

Discover
: 16 dígitos, começando com 6011 ou 65.

American Express
: 15 dígitos, começando com 34 ou 37.

Diners Club
: 14 dígitos, começando com 300, e indo a 305; 36 ou 38.

JCB
: 15 dígitos, começando com 2131 ou 1800; ou 16 dígitos, começando com 35.

Se você for aceitar apenas algumas bandeiras de cartões de crédito, poderá excluir da expressão regular os cartões não aceitos. Ao excluir o JCB, certifique-se de apagar também o último ‹|›, remanescente na expressão regular. Se você acabar com ‹||›, ou ‹|)›, em sua expressão regular, ela vai aceitar uma string vazia como um número de cartão válido.

Por exemplo, para aceitar somente Visa, MasterCard e AMEX, você pode usar:

```
^(?:
4[0-9]{12}(?:[0-9]{3})? | # Visa
5[1-5][0-9]{14} | # MasterCard
3[47][0-9]{13} # AMEX
)$
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Alternativamente:

```
^(?:4[0-9]{12}(?:[0-9]{3})?|5[1-5][0-9]{14}|3[47][0-9]{13})$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Se estiver buscando números de cartões de crédito em um corpo de texto mais longo, substitua as âncoras por extremidades de palavra ‹\b›.

Incorporando a solução em uma página web

O exemplo em “Exemplo de página web com JavaScript”, mostra como você poderia adicionar essas duas expressões regulares em seu formulário de pedido. O campo de entrada do número do cartão de crédito possui um manipulador de evento onkeyup, que chama a função validatecardnumber(). Essa função recupera o número do cartão a partir do campo de entrada, remove os espaços e hífens e, em seguida, valida o número, usando a expressão regular com grupos de captura numerados. O resultado da validação é exibido, substituindo-se o texto no último parágrafo da página.

Se a expressão regular falhar na correspondência, regexp.exec() retorna null, e (número de cartão inválido) será exibido. Se a expressão regular corresponder, regexp.exec() retorna um array de strings. O elemento zero contém a correspondência total. Os elementos de 1 a 6 armazenam o texto correspondido pelos seis grupos de captura.

A nossa expressão regular possui seis grupos de captura, separados por alternância. Isso significa que apenas um grupo de

captura vai participar da correspondência, armazenando o número do cartão. Os outros grupos ficarão vazios (undefined, ou com uma string vazia, dependendo de seu navegador.) A função verifica os seis grupos de captura, um por um. Quando encontrar um que não esteja vazio, o número do cartão é reconhecido e exibido.

Validação extra com o algoritmo de Luhn

Há uma verificação de validação adicional, que você pode fazer no número do cartão de crédito antes de processar o pedido. O último dígito do cartão de crédito é um checksum, calculado de acordo com o *algoritmo de Luhn*. Como este algoritmo requer aritmética básica, não é possível implementá-lo com uma expressão regular.

Você pode adicionar a verificação de Luhn ao exemplo da página web desta receita, inserindo a chamada luhn(cardnumber); antes da linha “else” na função validatecardnumber(). Dessa forma, a verificação de Luhn será feita somente se a expressão regular encontrar uma correspondência válida, e depois de determinar a bandeira do cartão. No entanto, determinar a bandeira do cartão de crédito não é necessário para a verificação de Luhn. Todos os cartões de crédito utilizam o mesmo método.

Em JavaScript, você pode codificar a função de Luhn como demonstrado a seguir:

```
function luhn(cardnumber) {
  // Construa um array com os dígitos do número do cartão
  var getdigits = /\d/g;
  var digits = [];
  while (match = getdigits.exec(cardnumber)) {
    digits.push(parseInt(match[0], 10));
  }
  // Execute o algoritmo de Luhn no array
  var sum = 0;
  var alt = false;
  for (var i = digits.length - 1; i >= 0; i--) {
    if (alt) {
        digits[i] *= 2;
        if (digits[i] > 9) {
```

```
            digits[i] -= 9;
          }
        }
        sum += digits[i];
        alt = !alt;
      }
      // O número do cartão é inválido de qualquer forma
      if (sum % 10 == 0) {
        document.getElementById("notice").innerHTML += '; verificação de Luhn
    passou';
      } else {
        document.getElementById("notice").innerHTML += ';verificação de Luhn
    falhou;
      }
    }
```

Essa função recebe uma string com o número do cartão de crédito como parâmetro. O número do cartão deverá ser composto apenas de dígitos. Em nosso exemplo, validatecardnumber() já retirou os espaços e hífens e determinou que o número do cartão possui o número correto de dígitos.

Em primeiro lugar, a função utiliza a expressão regular ‹\d›, para iterar todos os dígitos na string. Observe o modificador /g. Dentro do loop, match[0] recupera os dígitos correspondidos. Dado que expressões regulares tratam somente com texto (strings), chamamos parseInt() para nos certificarmos de que a variável será armazenada como um número inteiro, e não como string. Se não fizermos isso, a variável sum vai acabar como uma concatenação de uma string de dígitos, ao invés de um valor inteiro, com a adição dos números.

O algoritmo é executado sobre o array, calculando um checksum. Se a operação “soma módulo 10” for igual a zero, significa que o número do cartão é válido. Se não, o número é inválido.

4.20 Números VAT europeus

Problema

Suponha que lhe foi dado o trabalho de implementar um formulário de pedido on-line para uma empresa na União Europeia.

A legislação fiscal europeia estipula que, quando uma empresa registrada no VAT (seu cliente), localizada em um país da União Europeia (UE), efetua uma compra de um fornecedor (sua empresa) em outro país da UE, o fornecedor não deverá cobrar o VAT (Imposto sobre Valor Agregado). Se o comprador não for registrado no VAT, o fornecedor deve cobrar o VAT e remeter o imposto à administração fiscal local. O fornecedor deve usar o número de identificação VAT do comprador como prova, para a administração fiscal, de que nenhum imposto é devido. Isso significa que, para o fornecedor, é muito importante validar o número VAT do comprador, antes de proceder com a venda com isenção de imposto.

A causa mais comum, em relação a números inválidos VAT, são simples erros de digitação por parte do cliente. Para tornar o processo de compra mais rápido e amigável, você deve usar uma expressão regular para validar imediatamente o número VAT, enquanto o cliente preenche o formulário on-line. Você pode fazer isso com algum código JavaScript no lado do cliente, ou no script CGI em seu servidor web que receberá o formulário. Se o número não corresponder à expressão regular, o cliente poderá corrigir o erro de digitação imediatamente.

Solução

Remover espaços em branco e pontuação

Recupere o número VAT inserido por parte do cliente e o armazene em uma variável. Antes de realizar a verificação de número válido, realize uma pesquisa-e-substituição para substituir esta expressão regular, globalmente, por um texto de substituição vazio:

```
[-. □]
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

A receita 3.14 mostra como realizar essa substituição inicial. Estamos supondo que o cliente não entrará com qualquer pontuação, exceto hífens, pontos e espaços. Quaisquer outros caracteres estranhos serão capturados pela verificação programada.

Validar o número

Com espaços e pontuação retirados, esta expressão regular verifica se o número VAT é válido para qualquer um dos 27 países da UE:

```
^(
(AT)?U[0-9]{8} | # Áustria
(BE)?0?[0-9]{9} | # Bélgica
(BG)?[0-9]{9,10} | # Bulgária
(CY)?[0-9]{8}L | # Chipre
(CZ)?[0-9]{8,10} | # República Checa
(DE)?[0-9]{9} | # Alemanha
(DK)?[0-9]{8} | # Dinamarca
(EE)?[0-9]{9} | # Estônia
(EL|GR)?[0-9]{9} | # Grécia
(ES)?[0-9A-Z][0-9]{7}[0-9A-Z] | # Espanha
(FI)?[0-9]{8} | # Finlândia
(FR)?[0-9A-Z]{2}[0-9]{9} | # França
(GB)?([0-9]{9}([0-9]{3})?|[A-Z]{2}[0-9]{3}) | # Reino Unido
(HU)?[0-9]{8} | # Hungria
(IE)?[0-9]S[0-9]{5}L | # Irlanda
(IT)?[0-9]{11} | # Itália
(LT)?([0-9]{9}|[0-9]{12}) | # Lituânia
(LU)?[0-9]{8} | # Luxemburgo
(LV)?[0-9]{11} | # Latvia
(MT)?[0-9]{8} | # Malta
(NL)?[0-9]{9}B[0-9]{2} | # Holanda
(PL)?[0-9]{10} | # Polônia
(PT)?[0-9]{9} | # Portugal
(RO)?[0-9]{2,10} | # Romênia
(SE)?[0-9]{12} | # Suécia
(SI)?[0-9]{8} | # Eslovênia
(SK)?[0-9]{10} # Eslováquia
)$
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Esta expressão regular utiliza o modo de espaçamento livre, para facilitar sua edição posterior. De vez em quando, novos países aderem à União Europeia e países-membros mudam suas regras para os números VAT. Infelizmente, o JavaScript não suporta espaçamento livre. Neste caso, você deve colocar tudo em uma linha:

```
^((AT)?U[0-9]{8}|(BE)?0?[0-9]{9}|(BG)?[0-9]{9,10}|(CY)?[0-9]{8}L|↲
(CZ)?[0-9]{8,10}|(DE)?[0-9]{9}|(DK)?[0-9]{8}|(EE)?[0-9]{9}|↲
(EL|GR)?[0-9]{9}|(ES)?[0-9A-Z][0-9]{7}[0-9A-Z]|(FI)?[0-9]{8}|↲
(FR)?[0-9A-Z]{2}[0-9]{9}|(GB)?([0-9]{9}([0-9]{3})?|[A-Z]{2}[0-9]{3})|↲
(HU)?[0-9]{8}|(IE)?[0-9]S[0-9]{5}L|(IT)?[0-9]{11}|↲
(LT)?([0-9]{9}|[0-9]{12})|(LU)?[0-9]{8}|(LV)?[0-9]{11}|(MT)?[0-9]{8}|↲
(NL)?[0-9]{9}B[0-9]{2}|(PL)?[0-9]{10}|(PT)?[0-9]{9}|(RO)?[0-9]{2,10}|↲
(SE)?[0-9]{12}|(SI)?[0-9]{8}|(SK)?[0-9]{10})$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Siga a receita 3.6, para adicionar esta expressão regular a seu formulário de pedido.

Discussão

Retirar espaços em branco e pontuação

Para fazer com que os números VAT sejam mais fáceis de ler, as pessoas muitas vezes os digitam com pontuação extra, para separar os dígitos em grupos. Por exemplo, um cliente alemão poderá digitar seu número VAT DE123456789 como DE 123.456.789.

Uma expressão regular simples, que corresponda a números VAT de 27 países em qualquer notação imaginável, é uma tarefa impossível. Uma vez que a pontuação serve apenas para facilitar a leitura, é muito mais fácil retirar toda a pontuação em primeiro lugar e, em seguida, validar o número VAT resultante.

A expressão regular ‹[-.]› corresponde a um caractere que seja hífen, ponto ou espaço. Substituir todas as correspondências desta

expressão regular por nada, efetivamente exclui os caracteres de pontuação comumente usados em números VAT.

Números VAT consistem apenas em letras e algarismos. Em vez de usar ‹[-.]› para remover somente a pontuação comum, você pode usar ‹[^A-Z0-9]› para remover todos os caracteres inválidos.

Validar o número

As duas expressões regulares para validar o número são idênticas. A única diferença é que a primeira usa a sintaxe de espaçamento livre, para tornar a expressão regular mais legível e para indicar os países. O JavaScript não suporta espaçamento livre, mas os outros sabores nos dão essa escolha.

A expressão regular usa alternâncias para acomodar os números VAT de todos os 27 países. Os formatos essenciais são:

País	Formato(s)
Austria	U*99999999*
Bélgica	*999999999* ou *0999999999*
Bulgária	*999999999* ou *9999999999*
Chipre	*99999999*L
República Checa	*99999999*, *999999999* ou *9999999999*
Alemanha	*999999999*
Dinamarca	*99999999*
Estônia	*999999999*
Grécia	*999999999*
Espanha	X*9999999*X
Finlândia	*99999999*
França	XX*999999999*
Reino Unido	*999999999*, *999999999999* ou XX*999*
Hungria	*99999999*
Irlanda	*9*S*99999*L
Itália	*99999999999*
Lituânia	*999999999* ou *99999999999*
Luxemburgo	*99999999*

País	Formato(s)
Latvia	*99999999999*
Malta	*99999999*
Holanda	*999999999*B*99*
Polônia	*999999999*
Portugal	*999999999*
Romênia	*99*, *999*, *9999*, *99999*, *999999*, *9999999*, *99999999*, *999999999* ou *9999999999*
Suécia	*99999999999*
Eslovênia	*99999999*
Eslováquia	*999999999*

Estritamente falando, o código de duas letras faz parte do número VAT. No entanto, as pessoas muitas vezes o omitem, pois o endereço de cobrança já indica o país. A expressão regular aceitará os números VAT com ou sem o código do país. Se quiser que o código do país seja obrigatório, remova todos os pontos de interrogação da expressão regular. Se fizer isso, diga ao usuário, na mensagem de erro apresentada, que você exige a digitação do código do país.

Se você aceitar pedidos apenas de determinados países, poderá deixar de fora os que não aparecem na seleção de seu formulário de pedido. Ao excluir uma alternativa, certifique-se, também, de eliminar o operador ‹|›, que separa essa alternativa das alternativas posterior ou anterior. Se não fizer isso, vai acabar com ‹||› em sua expressão regular. ‹||› Insere uma alternativa que corresponde à string vazia, o que significa que seu formulário de pedido aceitará a omissão de um número VAT como um número VAT válido.

As 27 alternativas estão agrupadas. O grupo é colocado entre um acento circunflexo e um cifrão, ancorando a expressão regular no início e no final da string que estiver validando. A entrada completa deverá ser validada como um número VAT.

Se estiver procurando números VAT em um corpo maior do texto, substitua as âncoras por extremidades de palavra ‹\b›.

Variações

A vantagem de utilizar uma expressão regular para verificar todos os 27 países é que você só precisa adicionar uma validação de expressão regular em seu formulário de pedido. Você pode melhorar seu formulário utilizando 27 expressões regulares separadas. Primeiro, verifique o país que o cliente especificou no endereço de faturamento. Em seguida, procure a expressão regular apropriada, de acordo com o país:

País	Expressão regular
Áustria	‹^(AT)?U[0-9]{8}$›
Bélgica	‹^(BE)?0?[0-9]{9}$›
Bulgária	‹^(BG)?[0-9]{9,10}$›
Chipre	‹^(CY)?[0-9]{8}L$›
República Checa	‹^(CZ)?[0-9]{8,10}$›
Alemanha	‹^(DE)?[0-9]{9}$›
Dinamarca	‹^(DK)?[0-9]{8}$›
Estônia	‹^(EE)?[0-9]{9}$›
Grécia	‹^(EL|GR)?[0-9]{9}$›
Espanha	‹^(ES)?[0-9A-Z][0-9]{7}[0-9A-Z]$›
Finlândia	‹^(FI)?[0-9]{8}$›
França	‹^(FR)?[0-9A-Z]{2}[0-9]{9}$›
Reino Unido	‹^(GB)?([0-9]{9}([0-9]{3})?|[A-Z]{2}[0-9]{3})$›
Hungria	‹^(HU)?[0-9]{8}$›
Irlanda	‹^(IE)?[0-9]S[0-9]{5}L$›
Itália	‹^(IT)?[0-9]{11}$›
Lituânia	‹^(LT)?([0-9]{9}|[0-9]{12})$›
Luxemburgo	‹^(LU)?[0-9]{8}$›
Latvia	‹^(LV)?[0-9]{11}$›
Malta	‹^(MT)?[0-9]{8}$›
Holanda	‹^(NL)?[0-9]{9}B[0-9]{2}$›
Polônia	‹^(PL)?[0-9]{10}$›
Portugal	‹^(PT)?[0-9]{9}$›
Romênia	‹^(RO)?[0-9]{2,10}$›
Suécia	‹^(SE)?[0-9]{12}$›
Eslovênia	‹^(SI)?[0-9]{8}$›
Eslováquia	‹^(SK)?[0-9]{10}$›

Implemente a receita 3.6, para validar o número VAT em relação à expressão regular selecionada. Isso vai lhe dizer se o número é válido no país em que o cliente diz residir.

A principal vantagem de usar expressões regulares em separado é que você pode garantir que o número VAT inicie com o código correto do país, sem pedir que o cliente o digite. Quando a expressão regular corresponder ao número fornecido, verifique o conteúdo do primeiro grupo de captura. A receita 3.9 explica como fazer isso. Se o primeiro grupo de captura estiver vazio, o cliente não digitou o código do país no início do número VAT. Você pode, então, adicionar o código do país, antes de armazenar o número validado em sua base de dados de pedidos.

Números VAT gregos permitem dois códigos de país. EL é, tradicionalmente, utilizado para os números VAT gregos, mas GR é o código ISO para a Grécia.

Veja também:

A expressão regular verifica apenas se o número se parece com um número VAT válido. Isso é suficiente para eliminar os erros honestos. Obviamente, uma expressão regular não pode verificar se o número VAT está atribuído à empresa que faz o pedido. A União Europeia disponibiliza uma página web, em *http://ec.europa.eu/taxation_customs/vies/vieshome.do*, onde você pode verificar a qual empresa pertence um dado número VAT, caso haja algum a ela atribuído.

As técnicas utilizadas na expressão regular são discutidas nas receitas 2.3, 2.5 e 2.8.

---

1 Para ter ainda mais diversão (pelo menos se você tiver uma definição distorcida de diversão), tente criar níveis negativos triplos, quádruplos ou ainda maiores, lançando lookarounds negativos (veja a receita 2.16) e fazendo subtração de classes de caracteres (veja “Funcionalidades específicas de cada sabor” na página49, receita 2.3).

CAPÍTULO 5

Palavras, linhas e caracteres especiais

Este capítulo possui receitas que lidam com descoberta e manipulação de texto em uma variedade de contextos. Algumas das receitas mostram como construir os recursos que você deseja encontrar em um mecanismo avançado de pesquisa, tais como procurar uma grande variedade de palavras, ou encontrar palavras que apareçam próximas umas das outras. Outros exemplos ajudam-no a encontrar linhas inteiras que contenham palavras específicas, a remover palavras repetidas ou a escapar metacaracteres de expressões regulares.

O tema central deste capítulo é mostrar uma variedade de construções e técnicas de expressões regulares em ação. Lê-lo é como passar por um treinamento em um grande número de funcionalidades da sintaxe de expressão regular. Ele vai ajudá-lo na aplicação das expressões regulares nos problemas que você encontrar. Em muitos casos, o que buscamos é simples, mas os modelos que oferecemos nas soluções permitem que você os personalize para problemas específicos que estiver enfrentando.

5.1 Encontrar uma palavra específica

Problema

Você foi incumbido da simples tarefa de localizar todas as ocorrências da palavra “cat”, sem distinção entre maiúsculas e minúsculas. O problema é que ela deve aparecer como uma palavra completa. Você não deseja encontrar partes de palavras mais

longas, como hellcat , application ou Catwoman.

Solução

Os tokens de extremidade de palavra resolvem facilmente este problema:

```
\bcat\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

A receita 3.7 mostra como você pode usar esta expressão regular para encontrar todas as correspondências. A receita 3.14 mostra como você pode substituir as correspondências por outro texto.

Discussão

As extremidades de palavra, em ambas as extremidades da expressão regular, garantem que cat seja correspondido apenas quando aparecer como uma palavra completa. Mais precisamente, as extremidades de palavra exigem que a palavra cat seja separada de outros textos, por meio do início ou do final da string, de espaços em branco, pontuação ou de outros caracteres que não sejam palavras.

Mecanismos de expressão regular consideram letras, números e sublinhados como caracteres de palavras. Extremidades de palavra são discutidos, em maior profundidade, na receita 2.6.

Poderá ocorrer um problema ao trabalharmos com texto internacional em JavaScript, PCRE e Ruby, uma vez que esses sabores de expressão regular só consideram as letras da tabela ASCII para criar uma extremidade de palavra. Ou seja, as extremidades de palavra são encontradas apenas nas posições entre uma correspondência de ‹^|[^A-Za-z0-9_]› e ‹[A-Za-z0-9_]›, ou entre ‹[A-Za-z0-9_]› e ‹[^A-Za-z0-9_]|$›. Isso também é válido em Python, quando a sinalização UNICODE, ou U, não está definida. Isso impede que ‹\b› seja útil para uma pesquisa do tipo “apenas palavras inteiras”, em textos que contenham letras acentuadas ou em

palavras que utilizem sistemas de escrita diferentes do latino. Por exemplo, em JavaScript, PCRE e Ruby, ‹\büber\b› encontrará uma correspondência em darüber, mas não em dar über. Na maioria dos casos, isso é exatamente o oposto do que você deseja. O problema ocorre porque ü não é considerado um caractere de palavra e, por isso, uma extremidade de palavra é encontrada entre os dois caracteres rü. Nenhuma extremidade de palavra é encontrada entre um caractere de espaço e ü, porque eles criam uma sequência contígua de caracteres que não fazem parte de palavras.

Você pode lidar com esse problema usando um lookahead e um lookbehind (coletivamente, *lookaround*), ao invés de extremidades de palavras. Tal como as extremidades de palavras, o lookaround corresponde a uma posição de comprimento zero. Em PCRE (quando compilado com suporte a UTF-8) e Ruby 1.9, você pode emular extremidades de palavras baseadas em Unicode, utilizando, por exemplo, ‹(?<=\P{L}|^)cat(?=\P{L}|$)›. Esta expressão regular também utiliza tokens com a propriedade Letter (letra), do Unicode, negada (‹\P{L}›); tal propriedade foi discutida na receita 2.7. Lookaround é discutido na receita 2.16. Se quiser que os lookarounds tratem também os números e underscores como caracteres de palavra (assim como ‹\b›), substitua as duas instâncias de ‹\P{L}› pela classe de caracteres ‹[^\p{L}\p{N}_]›.

JavaScript e Ruby 1.8 não suportam lookbehind, nem propriedades Unicode. Você pode contornar esta falta de suporte a lookbehind, correspondendo a todo caractere que não seja de palavra e que apareça imediatamente antes de cada correspondência, para, então, removê-lo da correspondência, ou colocá-lo de volta na string, ao substituir as correspondências (veja exemplos de como utilizar partes de uma correspondência em uma string de substituição na receita 3.15). A falta de suporte adicional para corresponder a propriedades Unicode (juntamente com o fato dos tokens ‹\w› e ‹\W›, em ambas as linguagens de programação, corresponderem apenas a caracteres ASCII) significa que você deve se contentar com uma solução restritiva. Pontos de código, na

categoria de letras, estão espalhados ao longo de todo o conjunto de caracteres Unicode; assim, levaria milhares de caracteres para emular ‹\p{L}› utilizando sequências de escape Unicode e intervalos de classe de caracteres. Um bom meio-termo poderia ser ‹[A-Za-z\xAA\xB5\xBA\xC0-\xD6\xD8-\xF6\xF8-\xFF]›, que corresponde a todos os caracteres de letras Unicode em um espaço de endereçamento de 8 bits – ou seja, os primeiros 256 pontos de código Unicode, entre as posições 0x0 e 0xFF (veja a figura 5.1, para a lista de caracteres correspondidos). Esta classe de caracteres permite a correspondência (ou, no formato de negação, exclui a correspondência) de uma variedade de caracteres acentuados comumente utilizados e fora do espaço de endereçamento ASCII de 7 bits.

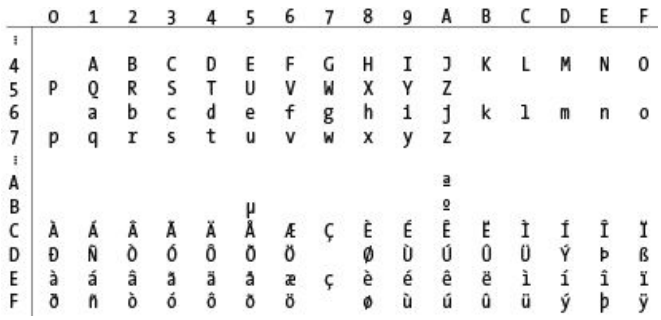


Figura 5.1 – Caracteres de letras Unicode no espaço de endereçamento de 8 bits.

A seguir, temos um exemplo de como substituir todas as instâncias da palavra “cat” por “dog”, em JavaScript. Levamos em consideração os caracteres acentuados, de modo que ëcat não é alterado. Para fazer isso, você precisa construir sua própria classe de caracteres, em vez de depender do ‹\b› ou ‹\w› nativos:

```
// Caracteres de letras de 8-bits
var L = 'A-Za-z\xAA\xB5\xBA\xC0-\xD6\xD8-\xF6\xF8-\xFF';
var pattern = '([^{L}]|^)cat([^{L}]|$)'.replace(/{L}/g, L);
var regex = new RegExp(pattern, 'gi');

// Substitui cat por dog, e coloca de volta qualquer
// caractere adicional correspondido
subject = subject.replace(regex, '$1dog$2');
```

Note que strings literais JavaScript utilizam \x*HH* (em que *HH* é um número hexadecimal de dois dígitos) para inserir caracteres

especiais. Assim, a variável L, passada para a expressão regular, na verdade acaba contendo as versões literais dos caracteres. Se você quisesse que as metassequências \x*HH* fossem passadas por meio da expressão regular em si, você teria de escapar as barras invertidas na string literal (por exemplo, "\\x*HH*"). No entanto, em nosso caso, isso não importa, já que não vai mudar aquilo a que a expressão regular corresponderá.

Veja também:

Receitas 5.2, 5.3 e 5.4.

5.2 Encontrar uma palavra entre várias

Problema

Você deseja encontrar qualquer palavra dentro de uma lista de palavras, sem ter de pesquisar várias vezes na string de assunto.

Solução

Usando alternância

A solução mais simples é alternar entre as palavras que deseja corresponder:

```
\b(?:one|two|three)\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Exemplos mais complicados, envolvendo correspondência de palavras similares, são mostrados na receita 5.3.

Exemplo de solução JavaScript

```
var subject = 'One times two plus one equals three.';
var regex = /\b(?:one|two|three)\b/gi;
subject.match(regex);
```

```
// Retorna um array com quatro correspondências: [ 'One', 'two', 'one', 'three']
// Esta função faz a mesma coisa, mas aceita um array de palavras a serem
// correspondidas. Qualquer metacaractere de expressão regular dentro das palavras
// aceitas é escapado com uma barra invertida antes da pesquisa.
function match_words (subject, words) {
var regex_metachars = /[(){}[\]*+?.\\^$|,\-]/g;
for (var i = 0; i < words.length; i++) {
  words[i] = words[i].replace(regex_metachars, '\\$&');
}
var regex = new RegExp('\\b(?:' + words.join('|') + ')\\b', 'gi');
return subject.match(regex) || [];
}
match_words(subject, ['one','two','three']);
// Retorna um array com quatro correspondências: [ 'One', 'two', 'one', 'three']
```

Discussão

Usando alternância

Há três partes nesta expressão regular: as extremidades de palavra, em ambas as pontas, o grupo de não-captura e a lista de palavras (separadas pelo operador de alternância ‹|›). As extremidades de palavra asseguram que a expressão regular não corresponderá a uma parte de uma palavra mais longa. O grupo de não-captura limita o alcance dos operadores de alternância; caso contrário, você precisaria escrever ‹\bone\b|\btwo\b|\bthree\b› para obter o mesmo efeito. Cada uma das palavras simplesmente corresponde a si mesma.

Como o mecanismo de expressão regular tenta corresponder a cada palavra na lista, da esquerda para a direita, você poderá sentir um pequeno ganho de desempenho ao colocar mais próximo do início da lista as palavras com mais probabilidade de serem encontradas no texto de assunto. Desde que as palavras estejam cercadas por extremidades de palavras, elas podem aparecer em qualquer ordem. No entanto, sem as extremidades de palavra, pode ser mais importante colocar as palavras mais longas em primeiro lugar; caso

contrário, você nunca encontraria “awesome”, ao buscar por ‹awe|awesome›. A expressão regular sempre corresponderia a “awe”, no início da palavra.

Note que esta expressão regular é utilizada para demonstrar, genericamente, uma correspondência de uma lista de palavras. Como ‹two› e ‹three›, nesse exemplo, começam com a mesma letra, você pode guiar mais eficientemente o mecanismo de expressão regular reescrevendo a expressão regular como ‹\b(?:one|t(?:wo|hree))\b›. Veja a receita 5.3 para mais exemplos sobre como corresponder, com mais eficiência, a uma palavra dentre uma lista de palavras.

Exemplo de solução em JavaScript

O exemplo em JavaScript corresponde à mesma lista de palavras, de duas maneiras diferentes. A primeira abordagem é, simplesmente, criar a expressão regular e pesquisar a string de assunto utilizando o método match, disponível para strings em JavaScript. Quando o método match é passado a uma expressão regular que usa a sinalização /g (“global”), ele retorna um array de todas as combinações encontradas na string, ou null, caso nenhuma correspondência seja encontrada.

A segunda abordagem utiliza uma função (match_words), que aceita a string de assunto onde será feita a pesquisa, e um array de palavras para pesquisar. A primeira função escapa qualquer metacaractere de expressão regular que possa existir nas palavras fornecidas e, então, divide a lista de palavras em uma nova expressão regular, utilizada para pesquisar na string. A função retorna um array de todas as correspondências encontradas, ou um array vazio, caso a expressão regular gerada não corresponda à string. As palavras desejadas podem ser correspondidas em qualquer combinação de letras minúsculas e maiúsculas, graças à utilização da flag de não-diferenciação entre maiúsculas e minúsculas (/i).

Veja também:

Receitas 5.1, 5.3 e 5.4.

5.3 Pesquisar palavras similares

Problema

Você possui diversos problemas, neste caso:

- Você deseja encontrar todas as ocorrências de color e colour em uma string.
- Você deseja encontrar qualquer uma das três palavras que terminam com “at”: bat, cat ou rat.
- Você deseja encontrar alguma palavra que termine com phobia.
- Você deseja encontrar variações comuns do nome “Steven”: Steve, Steven e Stephen.
- Você deseja corresponder a qualquer forma comum do termo “regular expression”.

Solução

As expressões regulares, que resolvem cada um dos problemas listados, são mostradas uma por vez. Todas estas soluções são listadas com a opção de não-diferenciação entre maiúsculas e minúsculas.

Color ou colour

```
\bcolou?r\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Bat, cat ou rat

```
\b[bcr]at\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Palavras terminadas em “phobia”

```
\b\w*phobia\b
```

Opções Regex: Sem distinção entre maiúsculas e minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Steve, Steven ou Stephen

```
\bSte(?:ven?|phen)\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Variações de “regular expression”

```
\breg(?:ular expressions?|ex(?:ps?|e[sn])?)\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Usar as extremidades de palavra para corresponder a palavras completas

Todas as cinco expressões regulares utilizam extremidades de palavra (‹\b›) para garantir que correspondam apenas a palavras completas. Os padrões utilizam várias abordagens diferentes, para permitir variações nas palavras correspondidas por eles.

Vamos dar uma olhada em cada uma.

Color ou colour

Esta expressão regular corresponderá a color ou colour, mas não corresponderá dentro de colorblind. Ela utiliza o quantificador ‹?› para tornar o “u” opcional. Quantificadores como ‹?› não funcionam como os curingas, com os quais as pessoas estão familiarizadas. Em vez disso, eles se conectam ao elemento imediatamente anterior, que pode ser tanto um token único (neste caso, o caractere literal “u”) ou um grupo de tokens envoltos em parênteses. O quantificador ‹?› repete o elemento precedente zero ou uma vez. O mecanismo de expressão regular tenta corresponder, primeiramente, ao elemento

vinculado ao quantificador e, se isso não funcionar, o mecanismo move-se para frente, sem correspondê-lo. Qualquer quantificador que permita zero repetição efetivamente torna o elemento anterior opcional, exatamente o que queremos aqui.

Bat, cat ou rat

Esta expressão regular utiliza uma classe de caracteres para corresponder a “b”, “c” ou “r”, seguido pelos caracteres literais “at”. Você poderia fazer a mesma coisa usando ‹\b(?:b|c|r)at\b›, ‹\b(?:bat|cat|rat)\b› ou ‹\bbat\b|\bcat\b|\brat\b›. No entanto, sempre que a diferença entre as correspondências permitidas for a escolha de um caractere, dentre uma lista, é melhor utilizar uma classe de caracteres. Não só as classes de caracteres fornecem uma sintaxe mais compacta e legível (por serem capazes de eliminar todas as barras verticais e intervalos de uso, tais como A-Z), a maioria dos mecanismos de expressão regular também proporciona uma otimização muito superior para as classes de caracteres. Alternar utilizando barras verticais exige que o mecanismo utilize o algoritmo de retrocesso, algo caro em termos computacionais, enquanto as classes de caracteres utilizam uma abordagem de pesquisa mais simples.

É preciso ter alguma cautela, no entanto. Classes de caracteres estão entre as funcionalidades mais incorretamente usadas em expressões regulares. É possível que elas nem sempre sejam bem documentadas, ou que talvez apenas alguns leitores prestem atenção aos detalhes. Independentemente das razões, não cometa os mesmos erros dos novatos. As classes de caracteres são capazes de corresponder a somente um caractere por vez, dentre os caracteres nelas especificados – sem exceções.

Seguem duas das formas mais comuns de má utilização das classes de caracteres:

Colocar as palavras em classes de caracteres

Claro, algo como ‹[cat]{3}› corresponderá a cat, mas também

corresponderá a act, ttt e a qualquer outra combinação de três caracteres, dentre os listados. A mesma reflexão aplica-se às classes de caracteres negadas, como ‹[^cat]›, que corresponde a qualquer caractere único que não seja c, a ou t.

Tentar utilizar o operador de alternância em classes de caracteres

Por definição, as classes de caracteres permitem escolha entre os caracteres especificados dentro delas. ‹[a|b|c]› corresponde a um único caractere do conjunto “abc|”, o que, provavelmente, não é o que você quer. E, mesmo que seja, a classe contém uma barra vertical redundante.

Veja a receita 2.3, para todos os detalhes necessários ao uso correto e eficaz das classes de caracteres.

Palavras terminadas em “phobia”

Como a expressão regular anterior, esta também utiliza um quantificador para fornecer a variação nas strings às quais ela corresponde. Essa expressão regular corresponde, por exemplo, a arachnophobia e hexakosioihexekontahexaphobia, e como ‹*› permite zero repetição, ela também corresponde a phobia, isoladamente. Se você deseja obrigar que pelo menos um caractere apareça antes do sufixo “phobia”, altere ‹*› para ‹+›.

Steve, Steven ou Stephen

Esta expressão regular combina algumas das funcionalidades utilizadas nos exemplos anteriores. Um grupo de não-captura, escrito como ‹(?:...)›, limita o alcance do operador de alternância ‹|›. O quantificador ‹?›, utilizado dentro da opção da primeira alternância do grupo, torna opcional o caractere anterior ‹n›. Isso melhora a eficiência (e brevidade) em relação ao equivalente ‹\bSte(?:ve|ven|phen)\b›. O mesmo princípio explica por que a string literal ‹Ste› aparece na frente da expressão regular, ao invés de ser repetida três vezes com ‹\b(?:Steve|Steven|Stephen)\b› ou ‹\bSteve\b|\bSteven\b|\bStephen\b›. Alguns mecanismos de retrocesso de

expressões regulares não são inteligentes o suficiente para perceber que qualquer texto correspondido por estas últimas expressões regulares deve começar com Ste. Em vez disso, quando o mecanismo penetra na string de assunto à procura de uma correspondência, ele encontrará, primeiro, uma extremidade de palavra; em seguida, verificará o caractere seguinte, para ver se é um S. Se não for, o mecanismo deverá tentar todos os caminhos alternativos, ao longo da expressão regular, antes que possa seguir em frente e começar tudo de novo, na próxima posição da string. Embora seja fácil para um ser humano enxergar que isto seria um desperdício de tempo (uma vez que todos os caminhos alternativos, ao longo da expressão regular, começam com “Ste”), o mecanismo não sabe disso. Se, diferentemente, você escrever a expressão regular como ‹\bSte(?:ven?|phen)\b›, o mecanismo imediatamente perceberá que não pode corresponder a uma string que não comece com esses caracteres.

Para uma visão mais aprofundada de um mecanismo de retrocesso de expressões regulares, consulte a receita 2.13.

Variações de “regular expression”

O último exemplo desta receita mistura alternância, classes de caracteres e quantificadores para corresponder a qualquer variação comum do termo “regular expression”. Como a expressão regular pode ser um pouco difícil de ler, vamos dividir e analisar cada uma de suas partes.

A expressão regular é escrita utilizando a opção de espaçamento livre, não disponível em JavaScript. Como espaços em branco são ignorados no modo de espaçamento livre, o caractere literal de espaço foi escapado com uma barra invertida:

```
\ b # Declara a posição em uma extremidade de palavra.
reg # Corresponde a "reg".
(?: # Agrupa, mas não captura ...
  ular \ # Corresponde a "ular".
  expressions? # Corresponde a "expression" ou "expressions".
 | # ou ...
```

```
  ex # Corresponde a "ex".
  (?: # Agrupa, mas não captura ...
    PS? # Corresponde a "p" ou "ps".
   | # ou ...
    e # Corresponde a "e".
    [sn] # Corresponde a um caractere do conjunto "sn".
  ) # Fim do grupo não-captura.
    ? # Repete o grupo anterior zero ou uma vez.
) # Fim do grupo não-captura.
\ b # Declara a posição em uma extremidade de palavra.
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex:. NET, Java, PCRE, Perl, Python, Ruby

Esse padrão corresponde a qualquer uma das seguintes strings:

- regular expressions
- regular expression
- regexps
- regexp
- regexes
- regexen
- regex

Veja também:

Receitas 5.1, 5.2 e 5.4.

5.4 Encontrar todas, exceto uma palavra específica

Problema

Você deseja usar uma expressão regular para corresponder a qualquer palavra completa, exceto cat. Catwoman e outras palavras que contenham as letras “cat” deverão ser correspondidas – com exceção da palavra cat.

Solução

Um lookahead negativo pode ajudá-lo a excluir palavras específicas, e é fundamental para a expressão regular a seguir:

```
\b(?!cat\b)\w+
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex:. NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Embora uma classe de caracteres negada (escrita como ‹[^...]›) facilite a correspondência de qualquer coisa, exceto um caractere específico, você não pode simplesmente escrever ‹[^cat]› para corresponder a qualquer coisa, exceto à palavra cat. ‹[^cat]› é uma expressão regular válida, mas corresponde a qualquer caractere, exceto c, a ou t. Assim, embora ‹\b[^cat]+\b› evitaria corresponder à palavra cat, ela também não corresponderia à palavra cup, porque ela contém a letra proibida c. A expressão regular ‹\b[^c][^a][^t]\w*› não é boa, também, porque rejeitaria qualquer palavra que tenha c como primeira letra, a como segunda letra, ou t como terceira. Além disso, ela não restringe as três primeiras letras a caracteres de palavra, e corresponde apenas a palavras com pelo menos três caracteres, uma vez que nenhuma das classes de caracteres negadas é opcional.

Com tudo isso em mente, vejamos como a expressão regular, mostrada no início desta receita, resolve o problema:

```
\ b # Declara a posição em uma extremidade de palavra.
(?! # Declara que a expr. regular a seguir não pode ser correspondida,
começando aqui ...
  cat # Corresponde a "cat".
  \ b # Declara a posição em uma extremidade de palavra.
) # Fim do lookahead negativo.
\w+ # Corresponde a um ou mais caracteres de palavras.
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex:. NET, Java, PCRE, Perl, Python, Ruby

A chave para este padrão é um lookahead negativo, que se pareça com ‹(?!...)›. O lookahead negativo proíbe a sequência cat, seguida por uma extremidade de palavra, sem impedir o uso das letras quando elas não aparecem nessa sequência exata, ou quando elas aparecem como parte de uma palavra mais longa ou mais curta. Não há extremidade de palavra no final da expressão regular, pois ela não mudaria aquilo a que a expressão regular corresponderia. O quantificador ‹+›, em ‹\w+›, repete o token de caractere de palavra tanto quanto possível, o que significa que sempre corresponderá até a próxima extremidade de palavra.

Quando aplicada à string de assunto categorically match any word except cat, a expressão regular encontrará cinco correspondências: categorically, match, any, word e except.

Variações

Encontrar palavras que não contenham outra palavra

Se, ao invés de tentar corresponder a qualquer palavra que não seja cat, você estiver tentando corresponder a qualquer palavra que *não contenha* cat, é necessário uma abordagem um pouco diferente:

```
\b(?:(?!cat)\w)+\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex:. NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Na seção anterior desta receita, a extremidade de palavra, no início da expressão regular, forneceu uma âncora conveniente; ela nos permitiu colocar o lookahead negativo no início da palavra. A solução utilizada aqui não é tão eficiente, mas é uma construção comumente utilizada, que permite corresponder a algo diferente de uma determinada palavra ou padrão. Ela procede repetindo um grupo que contém um lookahead negativo e um único caractere de palavra. Antes de corresponder a cada caractere, o mecanismo de expressão regular garante que a palavra cat não poderá ser

correspondida a partir da posição atual.

Ao contrário da expressão regular anterior, esta exige uma extremidade de palavra no final. Caso contrário, poderia acabar correspondendo apenas à primeira parte de uma palavra, até o ponto em que cat aparece dentro dela.

Veja também:

A receita 2.16, que inclui uma discussão mais aprofundada sobre lookaround (termo coletivo para lookaheads e lookbehinds, positivos e negativos).

Receitas 5.1, 5.5, 5.6 e 5.11.

5.5 Localizar qualquer palavra não seguida por uma palavra específica

Problema

Você deseja corresponder a qualquer palavra que não seja imediatamente seguida por cat, ignorando quaisquer espaços, pontuação ou outros caracteres que não façam parte de palavras e que apareçam entre elas.

Solução

Lookahead negativo é o ingrediente secreto para esta expressão regular:

```
\b\w+\b(?!\W+cat\b)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex:. NET, Java, JavaScript, PCRE, Perl, Python, Ruby

As receitas 3.7 e 3.14 mostram exemplos de como você pode implementar esta expressão regular em código.

Discussão

Tal como acontece com muitas outras receitas deste capítulo, as

extremidades de palavra (‹\b›) e o token de caracteres de palavra (‹\w›) trabalham em conjunto para corresponder a uma palavra completa. Você pode encontrar descrições mais detalhadas desses recursos na receita 2.6.

O ‹(?!...)›, em torno da segunda parte da expressão regular, é um lookahead negativo. O lookahead diz ao mecanismo de expressão regular para, temporariamente, dar um passo à frente na string, a fim de verificar se o padrão dentro do lookahead pode ser correspondido logo após a posição atual. Ele não consome qualquer um dos caracteres correspondidos no interior do lookahead. Ao invés disso, ele se limita a afirmar se uma correspondência é possível. Como estamos utilizando um lookahead negativo, o resultado da afirmação é invertido. Em outras palavras, se o padrão dentro do lookahead puder ser combinado logo adiante, a tentativa de correspondência falha, e o mecanismo de expressão regular avança para tentar tudo de novo, a partir do próximo caractere da string de assunto. Você pode encontrar mais detalhes sobre o lookahead (e sua contraparte, o lookbehind) na receita 2.16.

Quanto ao padrão dentro do lookahead, o ‹\W+› corresponde a um ou mais caracteres que não sejam de palavra e que apareçam antes de ‹cat›; a extremidade de palavra, no final, garante que ignoraremos apenas as palavras não seguidas de cat como uma palavra completa, no lugar de ignorar as palavras seguidas por qualquer outra palavra que comece com cat.

Note que esta expressão regular também corresponde à palavra cat, contanto que a palavra seguinte também não seja cat. Se quiser evitar a correspondência a cat, você pode combinar esta expressão regular com a receita 5.4, para obter ‹\b(?!cat\b)\w+\b(?!\W+cat\b)›.

Variações

Se quiser corresponder apenas às palavras seguidas por cat (sem incluir cat e os caracteres precedentes que não sejam de palavra, que, fatalmente, farão parte do texto correspondido), mude o

lookahead de negativo para positivo, e melhore seu astral:

```
\b\w+\b(?=\W+cat\b)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex:. NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Veja também:

A receita 2.16, que inclui uma discussão mais aprofundada sobre lookaround (termo coletivo para lookaheads e lookbehinds positivos e negativos).

Receitas 5.4 e 5.6.

5.6 Localizar qualquer palavra que não seja precedida por uma palavra específica

Problema

Você deseja corresponder a qualquer palavra que não seja imediatamente precedida da palavra `cat`, ignorando quaisquer espaços, pontuação ou caracteres que não sejam de palavras e que ocorram entre ambas.

Solução

Olhe atrás de você

O lookbehind permite verificar se o texto aparece antes de uma determinada posição. Funciona instruindo o mecanismo de expressão regular a, temporariamente, dar um passo para trás na string, verificando se algo pode ser encontrado, e terminando na posição em que você colocou o lookbehind. Veja a receita 2.16, caso precise refrescar os detalhes do lookbehind.

As três expressões regulares a seguir utilizam lookbehind negativo, que se parece com `‹(?<!...)›`. Infelizmente, os sabores de expressão

regular estudados neste livro diferem quanto aos tipos de padrões que eles permitem colocar dentro do lookbehind. Como resultado, as soluções acabam funcionando de maneira diferente em cada caso. Certifique-se de verificar a “Discussão” desta receita, para mais detalhes.

Palavras não precedidas por “cat”

```
(?<!\bcat\W+)\b\w+
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Regex sabor:. NET

```
(?<!\bcat\W{1,9})\b\w+
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex:. NET, Java, PCRE

```
(?<!\bcat)(?:\W+|^)(\w+)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex:. NET, Java, PCRE, Perl, Python, Ruby 1.9

Simular o lookbehind

JavaScript e Ruby 1.8 não suportam lookbehind, mesmo que suportem lookahead. No entanto, como o lookbehind aparece logo no início da expressão regular, para este problema, é perfeitamente possível simular o lookbehind dividindo a expressão regular em duas partes, como demonstrado no exemplo a seguir, em JavaScript:

```
var subject = 'My cat is furry.',
  main_regex = /\b\w+/g,
  lookbehind = /\bcat\W+$/i,
  lookbehind_type = false, // lookbehind negativo
  matches = [],
  match,
  left_context;
while (match = main_regex.exec(subject)) {
  left_context = subject.substring(0, match.index);
  if (lookbehind_type == lookbehind.test(left_context)) {
    matches.push(match[0]);
  } else {
```

```
    main_regex.lastIndex = match.index + 1;
  }
}
// corresponde a: ['My','cat','furry']
```

Discussão

Lookbehind de comprimento fixo, finito e infinito

A primeira expressão regular utiliza o lookbehind negativo ‹(?<!\bcat\W+)›. Como o quantificador ‹+›, utilizado dentro do lookbehind, não possui um limite superior de quantos caracteres pode corresponder, esta versão só funciona com o sabor de expressão regular .NET. Todos os outros sabores de expressão regular abrangidos por este livro exigem um comprimento fixo ou máximo (finito) para padrões de lookbehind.

A segunda expressão regular substitui o ‹+›, dentro do lookbehind, por ‹{1,9}›. Como resultado, ele pode ser usado com .NET, Java e PCRE, todos os quais suportam lookbehinds de comprimento variável, quando há um limite superior conhecido de quantos caracteres podem ser correspondidos dentro deles. Eu tenho escolhido, arbitrariamente, um comprimento máximo de nove caracteres que não sejam caracteres de palavra, para separar as palavras. Isso permite um pouco de pontuação e algumas linhas em branco para separar as palavras. A menos que você esteja trabalhando com um texto incomum, isso provavelmente funcionará exatamente como a solução.NET anterior. No entanto, mesmo em .NET, estabelecer um limite razoável de repetições para qualquer quantificador dentro do lookbehind pode tornar sua expressão regular mais eficiente, mesmo que seja apenas por reduzir a quantidade de retrocessos não-antecipados que possam, potencialmente, ocorrer dentro do lookbehind.

A terceira expressão regular foi reestruturada de forma que permita ao lookbehind testar uma string de comprimento fixo, adicionando, assim, suporte a mais sabores de expressão regular. Para tanto, a

classe de caracteres abreviada (‹\W›), relacionada a caracteres que não sejam de palavra, foi movida para fora do lookbehind. Isso significa que os caracteres que não sejam de palavra (como pontuação e espaços em branco), precedentes às palavras que esteja procurando, farão parte da string correspondida (e retornada), de fato, pela expressão regular. Para poder ignorar mais facilmente essa parte de cada correspondência, um grupo de captura foi adicionado em torno da sequência final de caracteres que não sejam de palavra. Com um pouco de código adicional, você pode ler apenas o valor da retrorreferência 1, em vez de toda a correspondência, dando-lhe o mesmo resultado que você obteria a partir das expressões regulares anteriores. A receita 3.9 mostra o código necessário para trabalhar com retrorreferências.

Simular o lookbehind

O JavaScript não suporta lookbehind, mas o código de exemplo em JavaScript mostra como você pode simular o lookbehind que aparece no início de uma expressão regular utilizando duas expressões regulares. Ele não impõe restrições sobre o comprimento do texto correspondido pelo lookbehind (simulado).

Começamos por dividir a expressão regular ‹(?<!\bcat\W+)\b\w+› da solução original em duas partes: o padrão dentro do lookbehind (‹\bcat\W+›) e o padrão posterior (‹\b\w+›). Acrescente um ‹$› no final da expressão regular do lookbehind. Se precisar usar a opção “circunflexo e cifrão correspondem em quebras de linha” (/m), na expressão regular lookbehind, use ‹$(?!\s)›, em vez de ‹$›, para garantir que ela possa corresponder apenas no final do seu texto de assunto. A variável lookbehind_type indica se estamos emulando um lookbehind positivo ou negativo, utilizando true para positivo e false para negativo.

Com as variáveis definidas, utilizamos main_regex e o método exec para iterar a string de assunto (veja a receita 3.11, para uma descrição deste processo). Quando uma correspondência é encontrada, a parte do texto de assunto, antes da correspondência,

é copiada para uma nova variável de string (left_context) e, assim, testamos se a expressão regular lookbehind corresponde àquela string. Por causa da âncora acrescentada ao final da expressão regular lookbehind, ela efetivamente coloca a segunda correspondência imediatamente à esquerda da primeira. Ao comparar o resultado do teste de lookbehind com lookbehind_type, podemos determinar se a correspondência satisfaz os critérios completos de uma correspondência bem-sucedida.

Finalmente, escolhemos uma de duas rotas. Se tivermos uma correspondência bem-sucedida, anexamos o texto correspondido ao array matches. Se não, mudamos a posição na qual continuaremos buscando uma correspondência (utilizando main_regex.lastIndex) para um caractere após a posição inicial da última correspondência do objeto main_regex, em vez de deixar que a próxima iteração do método exec inicie no final da correspondência em andamento.

Ufa! Terminamos.

Este é um truque avançado que tira proveito da propriedade lastIndex, dinamicamente atualizada por expressões regulares em JavaScript que utilizem a sinalização /g (“global”). Normalmente, atualizar e redefinir lastIndex é algo que acontece “automagicamente”. Aqui, nós utilizamos a propriedade para assumir o controle do trajeto da expressão regular ao longo da string de assunto, movendo-se para frente e para trás, conforme necessário. Este truque só permite emular um lookbehind que apareça no início de uma expressão regular. Com algumas mudanças, este código também pode ser usado para emular um lookbehind no final de uma expressão regular. No entanto, ele não serve como substituto completo para suporte ao lookbehind. Devido à interação entre lookbehind e retrocesso, essa abordagem não pode ajudá-lo a emular, com precisão, o comportamento de um lookbehind que apareça no meio de uma expressão regular.

Variações

Se você deseja apenas corresponder a palavras precedidas por cat (sem incluir cat e os caracteres seguintes que não sejam de palavra, e que fatalmente farão parte do texto correspondido), altere o lookbehind negativo para um lookbehind positivo:

```
(?<=\bcat\W+)\b\w+
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Regex sabor:. NET

```
(?<=\bcat\W{1,9})\b\w+
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex:. NET, Java, PCRE

```
(?<=\bcat)(?:\W+|^)(\w+)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex:. NET, Java, PCRE, Perl, Python, Ruby 1.9

Veja também:

A receita 2.16, que inclui uma discussão mais aprofundada sobre lookaround (termo coletivo para lookaheads e lookbehinds positivos e negativos).

Receitas 5.4 e 5.5.

5.7 Encontrar palavras próximas umas das outras

Problema

Você deseja emular uma pesquisa NEAR utilizando uma expressão regular. Para leitores não familiarizados com o termo, algumas ferramentas de busca que utilizam operadores booleanos, como NOT e OR, também possuem um operador chamado NEAR. Buscar por “word1 NEAR word2” encontra word1 e word2 em qualquer ordem, contanto que elas ocorram dentro de uma certa distância uma da outra.

Solução

Se estiver pesquisando duas palavras diferentes, poderá combinar duas expressões regulares – uma que corresponda a word1 antes de word2, e outra que inverta a ordem das palavras. A seguinte expressão regular permite até cinco palavras, para podermos separar as duas que estejamos pesquisando:

```
\b(?:word1\W+(?:\w+\W+){0,5}?word2|word2\W+(?:\w+\W+){0,5}?word1)\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
\b(?:
  word1 # primeiro termo
  \W+ (?:\w+\W+){0,5}? # até cinco palavras
  word2 # segundo termo
| # ou o mesmo padrão ao contrário...
  word2 # segundo termo
  \W+ (?:\w+\W+){0,5}? # até cinco palavras
  word1 # primeiro termo
)\b
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

A segunda expressão regular utiliza a opção de espaçamento livre, e adiciona espaços em branco e comentários. De resto, as duas expressões regulares são idênticas. O JavaScript não suporta o modo de espaçamento livre, mas os outros sabores de expressões regulares listados permitem que você faça essa escolha. As receitas 3.5 e 3.7 mostram exemplos de como adicionar estas expressões regulares em seu formulário de busca ou em outro código.

Discussão

Esta expressão regular coloca, lado a lado, duas cópias invertidas do mesmo padrão e, então, as engloba com extremidades de palavra. O primeiro subpadrão corresponde a word1, seguido de zero a cinco palavras, para, em seguida, corresponder a word2. O segundo subpadrão corresponde à mesma coisa, com a ordem das palavras invertida.

O quantificador preguiçoso ‹{0,5}?› aparece em ambos os subpadrões. Isso faz com que a expressão regular corresponda à menor quantidade de palavras possível entre os dois termos que estiver pesquisando. Se você executar a expressão regular no texto de assunto word1 word2 word2, ela corresponderá a word1 word2, pois esta correspondência contém menos palavras (zero) entre os pontos inicial e final. Para configurar a distância permitida entre as palavras-alvos, mude os valores 0 e 5, nos dois quantificadores, para seus valores preferidos. Por exemplo, se você os alterou para ‹{1,15}?›, isso permitiria até 15 palavras entre as duas que estiver procurando, exigindo que sejam separadas por, pelo menos, uma outra palavra.

As classes de caracteres abreviadas, usadas para corresponder a caracteres de palavra e a caracteres que não sejam de palavra (‹\w› e ‹\W›, respectivamente), seguem a definição peculiar de expressões regulares, em que as palavras são compostas apenas de letras, números e underscores.

Variações

Usar uma condicional

Há muitas maneiras de escrever a mesma expressão regular. Neste livro, tentamos equilibrar as permutas entre portabilidade, brevidade, eficiência e outras considerações. No entanto, por vezes, as soluções que não são as ideais ainda podem ser educativas. As próximas duas expressões regulares ilustram abordagens alternativas para encontrar palavras próximas uma da outra. Não recomendamos utilizá-las porque, embora correspondam ao mesmo texto, normalmente demorarão um pouco mais. Elas também funcionam com menos sabores de expressões regulares.

A primeira expressão regular utiliza uma condicional para determinar se corresponde a word1 ou a word2 no final da expressão regular, em vez de simplesmente juntar os padrões reversos. A condicional verifica se o grupo de captura 1 participou da correspondência, o

que significaria que a correspondência começa com word2.

```
\b(?:word1|(word2))\W+(?:\w+\W+){0,5}?(?(1)word1|word2)\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, PCRE, Perl, Python

Esta próxima versão, mais uma vez, utiliza a condicional para determinar qual palavra deve ser correspondida no final. Porém, ela adiciona mais duas funcionalidades de expressão regular na mistura:

```
\b(?:(?<w1>word1)|(?<w2>word2))\W+(?:\w+\W+){0,5}?(?(w2)(?&w1)|(?&w2))\b
```

Opções Regex: Nenhuma

Sabores Regex: PCRE 7, Perl 5.10

Aqui, grupos de captura nomeados, escritos como ‹(?<name>...)›, envolvem as primeiras instâncias de ‹word1› e ‹word2›. Isso permite que você utilize a sintaxe de subrotina ‹(?&name)›, para reutilizar um subpadrão chamado pelo nome. Este procedimento não funciona do mesmo modo que uma retrorreferência a um grupo nomeado. Uma retrorreferência nomeada, como ‹\k<name>› (.NET, PCRE 7, Perl 5.10) ou ‹(?P=name)› (PCRE 4 e posterior, Perl 5.10 e Python), permite que você corresponda, novamente, ao texto que já tenha sido correspondido por um grupo de captura nomeado. Uma subrotina do tipo ‹(?&name)› permite reutilizar o padrão real, contido dentro do grupo correspondente. Você não pode usar uma retrorreferência aqui; isso só permitiria corresponder novamente a palavras que já tenham sido correspondidas. As subrotinas dentro da condicional, no final da expressão regular, correspondem à palavra (dentre as duas opções providenciadas) que ainda *não tenha* sido correspondida, sem que seja preciso soletrar novamente. Isso significa que há apenas um lugar que precisa ser atualizado na expressão regular, caso você tenha de reutilizá-la para corresponder a palavras diferentes.

Corresponder a três ou mais palavras, próximas umas das outras

Permutações em crescimento exponencial. Corresponder a duas palavras, próximas uma da outra, é tarefa bem simples. Afinal, só existem duas maneiras possíveis de ordená-las. Porém, e se você quiser combinar três palavras em qualquer ordem? Existem seis ordens possíveis (veja figura 5.2). O número de maneiras a partir das quais você pode dispor um determinado conjunto de palavras é *n*!, ou o produto de inteiros consecutivos de 1 a *n* (“n fatorial”). Com quatro palavras, existem 24 possibilidades de ordenação. Quando você chegar a 10 palavras, o número de arranjos explodirá em milhões. Simplesmente não é viável corresponder a mais do que algumas palavras próximas umas das outras, utilizando as técnicas de expressão regular discutidas até agora.

A solução feia. Uma maneira de resolver este problema seria a repetição de um grupo que corresponda às palavras obrigatórias ou a qualquer outra palavra (depois que uma palavra é correspondida pela primeira vez), e, em seguida, utilizar condicionais para impedir que uma tentativa de correspondência tenha êxito, até que todas as palavras obrigatórias sejam correspondidas. A seguir, temos um exemplo de como corresponder a três palavras em qualquer ordem, com até cinco outras palavras separando-as:

```
\b(?:(?>(word1)|(word2)|(word3)|(?(1)|(?(2)|(?(3)|(?!)))))\w+)\b\W*?){3,8}↲
(?(1)(?(2)(?(3)|(?!))|(?!))|(?!))
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, PCRE, Perl

```
Dois valores:
  [ 12, 21 ]
  = 2 arranjos possíveis
Três valores:
  [ 123, 132,
  213, 231,
  312, 321 ]
  = 6 arranjos possíveis
Quatro valores:
  [ 1234, 1243, 1324, 1342, 1423, 1432,
  2134, 2143, 2314, 2341, 2413, 2432,
  3124, 3142, 3214, 3241, 3412, 3421,
  4123, 4132, 4213, 4231, 4312, 4321 ]
  = 24 arranjos possíveis
```

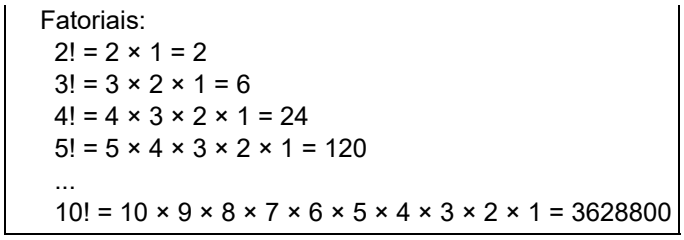


Figura 5.2 – Várias maneiras de dispor um conjunto

Aqui vemos a expressão regular novamente com o grupo atômico (veja a receita 2.14), substituído por um grupo-padrão de não-captura, a fim de adicionar suporte ao Python:

```
\b(?:(?:(word1)|(word2)|(word3)|(?(1)|(?(2)|(?(3)|(?!)))))\w+)\b\W*?){3,8}↵
(?(1)(?(2)(?(3)|(?!))|(?!))|(?!))
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, PCRE, Perl, Python

Os quantificadores ‹{3,8}›, nas expressões regulares mostradas, representam as três palavras obrigatórias e, assim, permite de zero a cinco palavras entre elas. Os lookaheads negativos vazios, que se parecem com ‹(?!)›, nunca corresponderão, e são usados para bloquear certos caminhos ao longo da expressão regular, até ocorrer a correspondência de uma ou mais palavras obrigatórias. A lógica que controla esses caminhos é implementada utilizando dois conjuntos de condicionais aninhadas. O primeiro conjunto impede a correspondência de qualquer palavra antiga usando ‹\w+›, até que pelo menos uma das palavras obrigatórias tenha sido correspondida. O segundo conjunto de condicionais, no final, força o mecanismo de expressão regular a retroceder ou a falhar, a menos que todas as palavras obrigatórias tenham sido correspondidas.

Essa é a visão geral sobre como isso funciona, mas, em vez de se aprofundar e descrever como adicionar mais palavras obrigatórias, vamos dar uma olhada em uma implementação aprimorada, que adiciona suporte para mais sabores de expressão regular, envolvendo alguns truques.

Explorando retrorreferências vazias. A solução feia funciona, mas ela, provavelmente, venceria um concurso de expressões regulares

confusas, por ser tão difícil de ler e gerenciar. E ela só pioraria, caso você adicionasse mais palavras obrigatórias à mistura.

Felizmente, há um truque que você pode utilizar, tornando-a muito mais fácil de acompanhar, ao mesmo tempo adicionando suporte para Java e Ruby (nenhum dos quais suporta condicionais).

O comportamento descrito nesta seção deve ser usado com cautela em aplicações de produção. Estamos elevando as expectativas a respeito do comportamento das expressões regulares a níveis não documentados na maioria das bibliotecas de expressões regulares.

```
\b(?:(?>word1()|word2()|word3()|(?>\1|\2|\3)\w+)\b\W*?){3,8}\1\2\3
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Ruby

```
\b(?:(?:word1()|word2()|word3()|(?:\1|\2|\3)\w+)\b\W*?){3,8}\1\2\3
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Utilizando esta construção, fica fácil adicionar mais palavras obrigatórias. A seguir, temos um exemplo que permite que quatro palavras apareçam em qualquer ordem, com um total de até cinco outras palavras entre elas:

```
\b(?:(?>word1()|word2()|word3()|word4()|↲
(?>\1|\2|\3|\4)\w+)\b\W*?){4,9}\1\2\3\4
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Ruby

```
\b(?:(?:word1()|word2()|word3()|word4()|↲
(?:\1|\2|\3|\4)\w+)\b\W*?){4,9}\1\2\3\4
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Essas expressões regulares, intencionalmente, utilizam grupos de captura vazios após cada uma das palavras obrigatórias. Uma vez que qualquer tentativa de corresponder a uma retrorreferência como ‹\1› falhará, se o grupo de captura correspondente ainda não tiver participado da correspondência, retrorreferências a grupos vazios podem ser usadas para controlar o caminho de um mecanismo de expressão regular ao longo de um padrão, tal como as condicionais

verbosas que mostramos anteriormente. Se o grupo correspondente já participou da tentativa de correspondência quando o mecanismo atinge a retrorreferência, ele simplesmente corresponderá a uma string vazia e seguirá em frente.

Aqui, o agrupamento ‹(?>\1|\2|\3)› impede a correspondência de uma palavra por meio de um ‹\w+›, até que pelo menos uma das palavras obrigatórias seja correspondida. As retrorreferências são repetidas no final do padrão, para evitar que qualquer correspondência obtenha êxito até que todas as palavras obrigatórias sejam encontradas.

O Python não suporta grupos atômicos, assim, mais uma vez, os exemplos que listam o Python entre os sabores de expressão regular substituem esses grupos pelos grupos-padrão de não-captura. Embora isso faça com que as expressões regulares sejam menos eficientes, suas correspondências não mudam. O agrupamento mais externo não pode ser atômico em qualquer sabor, pois, para que isso funcione, o mecanismo de expressão regular deve ser capaz de retroceder ao grupo externo, caso as retrorreferências, no final do padrão, não consigam corresponder.

O JavaScript usa retrorreferências de acordo com suas próprias regras. Mesmo que o JavaScript suporte toda a sintaxe utilizada nas versões Python deste padrão, ele possui duas regras de comportamento que impedem que o truque funcione. A primeira questão está relacionada ao que é correspondido pelas retrorreferências a grupos de captura que ainda não participaram de uma correspondência. A especificação do JavaScript determina que tais retrorreferências correspondem a uma string vazia, ou, em outras palavras, que elas sempre corresponderão com êxito. Em quase todos os outros sabores de expressão regular, o oposto é verdadeiro: elas nunca correspondem e, como resultado, elas forçam o retrocesso do mecanismo de expressão regular, até que toda a correspondência falhe ou até que os grupos referenciados por elas participem da correspondência, criando, assim, a

possibilidade de que as retrorreferências também consigam corresponder.

A segunda diferença do sabor JavaScript envolve o valor lembrado pelos grupos de captura aninhados dentro de um grupo exterior repetido, por exemplo, ‹((a)|(b))+›. Na maioria dos sabores de expressão regular, o valor lembrado por um grupo de captura, dentro de um agrupamento repetido, será qualquer coisa que o grupo tenha correspondido na última vez em que participou da correspondência. Assim, após um ‹(?:(a)|(b))+› ser usado para corresponder a ab, o valor da retrorreferência 1 seria a. No entanto, de acordo com a especificação do JavaScript, o valor das retrorreferências a grupos aninhados é redefinida sempre que o grupo externo é repetido. Assim, ‹(?:(a)|(b))+› ainda corresponderia a ab, mas a retrorreferência 1 referenciaria um grupo de captura não-participante após a correspondência estar completa, que em JavaScript corresponderia a uma string vazia dentro da própria expressão regular e que seria retornada como undefined, por exemplo, no array retornado pelo método RegExp.prototype.exec.

Qualquer uma destas diferenças de comportamento, encontradas no sabor de expressão regular do JavaScript, é o suficiente para impedir a emulação de condicionais utilizando grupos de captura vazios, tal como descrito aqui.

Palavras múltiplas a qualquer distância umas das outras

Se você deseja, simplesmente, testar se uma lista de palavras pode ser encontrada em qualquer lugar de uma string de assunto, sem levar em conta sua proximidade, o lookahead positivo fornece uma maneira de fazê-lo, utilizando uma operação de busca.

Em muitos casos, é mais simples, e mais eficiente, realizar pesquisas distintas para cada termo que estiver procurando, ficando de olho para ver se todos os testes retornam positivos.

```
\A(?=.*?\bword1\b)(?=.*?\bword2\b).*\Z
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, ponto

corresponde a quebras de linha

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?=[\s\S]*?\bword1\b)(?=[\s\S]*?\bword2\b)[\s\S]*$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas (“^ e $ correspondem em quebras de linhas” não deve estar ativado)

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Essas expressões regulares correspondem à string completa, caso todas as palavras sejam encontradas dentro dela; caso contrário, elas não encontrarão nenhuma correspondência. Programadores JavaScript não podem utilizar a primeira versão, pois o JavaScript não suporta as âncoras ‹\A› e ‹\Z›, nem a opção “ponto corresponde a quebras de linha”.

Você pode aplicar estas expressões regulares seguindo o código da receita 3.6. Basta mudar os termos ‹word1› e ‹word2› pelos termos que estiver procurando. Se estiver verificando mais de duas palavras, poderá adicionar tantos lookaheads à frente da expressão regular quanto necessitar. Por exemplo, ‹\A(?=.*?\bword1\b)(?=.*?\bword2\b)(?=.*?\bword3\b).*\Z› pesquisa por três palavras.

Veja também:

Receitas 5.5 e 5.6.

5.8 Encontrar palavras repetidas

Problema

Você está editando um documento, e gostaria de verificar se existem palavras incorretamente repetidas. Você deseja encontrar estas palavras duplicadas, independentemente das diferenças entre maiúsculas e minúsculas, como em “The the”. Você também deseja permitir diferentes quantidades de espaços em branco entre as palavras, mesmo que isso faça com que as palavras se estendam por mais de uma linha.

Solução

Uma retrorreferência corresponde a algo correspondido antes e, portanto, fornece o ingrediente-chave para esta receita:

```
\b([A-Z]+)\s+\1\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Se você deseja usar essa expressão regular para manter a primeira palavra, mas remover palavras subsequentes duplicadas, substitua todas as correspondências pela retrorreferência 1. Outra abordagem seria destacar as correspondências, envolvendo-as com outros caracteres (como uma tag HTML), para que possam ser mais facilmente identificadas durante uma inspeção posterior. A receita 3.15 mostra como utilizar retrorreferências em seu texto de substituição, o que você precisará fazer para implementar qualquer uma dessas abordagens.

Se quiser apenas encontrar palavras repetidas, de modo que você possa, manualmente, verificar se elas precisam ser corrigidas, a receita 3.7 mostra o código necessário. Um editor de texto ou uma ferramenta tipo grep, como as mencionadas em “Ferramentas para trabalhar com expressões regulares”, no capítulo 1, podem ajudá-lo a encontrar palavras repetidas, enquanto fornecem o contexto necessário para determinar se as palavras em questão estão sendo usadas corretamente.

Discussão

Existem duas coisas necessárias para corresponder a algo previamente correspondido: um grupo de captura e uma retrorreferência. Coloque o que você deseja corresponder mais de uma vez dentro de um grupo de captura e, então, corresponda-o novamente, usando uma retrorreferência. Isso funciona diferentemente de uma simples repetição de um token, ou de um grupo usando um quantificador. Considere a diferença entre as expressões regulares simplificadas ‹(\w)\1› e ‹\w{2}›. A primeira expressão regular utiliza um grupo de captura e uma retrorreferência

para corresponder ao mesmo caractere de palavra duas vezes, enquanto a segunda utiliza um quantificador para corresponder a quaisquer dois caracteres de palavra. A receita 2.10 discute a magia das retrorreferências em maior profundidade.

De volta para o problema em mãos. Esta receita só encontra palavras repetidas compostas por letras de A a Z, e de a a z (desde que a opção de não-diferenciação entre maiúsculas e minúsculas esteja habilitada). Para também permitir letras acentuadas e letras de outros sistemas de escrita, você pode utilizar a propriedade Letter (‹\p{L}›) do Unicode, caso seu sabor de expressão regular a suporte. (veja “Propriedade ou categoria Unicode”).

Entre o grupo de captura e a retrorreferência, ‹\s+› corresponde a todos os caracteres em branco, como espaços, tabulações ou quebras de linha. Se quiser restringir a espaços horizontais os caracteres que podem separar palavras repetidas (ou seja, sem quebras de linha), substitua o ‹\s› por ‹[^\S\r\n]›. Isso impede a correspondência de palavras repetidas que apareçam em várias linhas. PCRE 7 e Perl 5.10 incluem a classe de caracteres abreviada ‹\h›, que, talvez, você prefira usar aqui, uma vez que ela é projetada especificamente para corresponder a espaços horizontais.

Por fim, as extremidades de palavra, no início e no final da expressão regular, garantem que ela não corresponderá dentro de outras palavras, como, por exemplo, em “this thistle”.

Repare que o uso de palavras repetidas nem sempre é errado; então, simplesmente removê-las sem uma avaliação é potencialmente perigoso. Por exemplo, as construções “that that” e “had had” são, geralmente, aceitas no inglês coloquial. Homônimos, nomes, palavras onomatopeicas (como “oink oink” ou “ha ha”) e outras construções também resultam, ocasionalmente, em palavras repetidas de maneira intencional. Portanto, na maioria dos casos, você terá de examinar visualmente cada correspondência.

Veja também:

Receita 2.10, que discute as retrorreferências com mais profundidade.

Receita 5.9, que mostra como corresponder a linhas de texto repetidas.

5.9 Remover linhas duplicadas

Problema

Você possui um arquivo de log, uma saída de consulta a um banco de dados, ou algum outro tipo de arquivo, ou string, com linhas duplicadas. Você precisa remover todas, exceto uma, de cada linha em duplicata, utilizando um editor de texto ou ferramenta similar.

Solução

Existem vários softwares (incluindo o utilitário de linha de comando do Unix, `uniq`, e o cmdlet `Get-Unique`, do Windows PowerShell) que podem ajudá-lo a remover linhas duplicadas em um arquivo ou em uma string. As seções seguintes possuem três abordagens de expressão regular que podem ser especialmente úteis, quando tentamos realizar essa tarefa em um editor de texto sem recursos de criação de scripts, mas com suporte a pesquisa-e-substituição por expressões regulares.

Ao programar, as opções dois e três devem ser evitadas, pois são ineficientes em comparação a outros métodos disponíveis, como usar um objeto hash para manter um registro de linhas únicas. No entanto, a primeira opção (que exige a classificação das linhas com antecedência, a menos que você queira apenas remover duplicatas adjacentes) pode ser uma abordagem aceitável, por ser rápida e fácil.

Opção 1: Ordenar linhas e remover duplicatas adjacentes

Se você for capaz de ordenar as linhas no arquivo ou na string em

que estiver trabalhando, para que todas as linhas duplicadas apareçam uma ao lado da outra, você deve fazê-lo, a menos que a ordem das linhas deva ser preservada. Esta opção permitirá uma utilização mais simples, e mais eficiente, da operação busca-e-substituição para remover as duplicatas.

Após a classificação das linhas, utilize a expressão regular e as strings de substituição mostradas a seguir para se livrar das duplicatas:

```
^(.*)(?:(?:\r?\n|\r)\1)+$
```

Opções Regex: ^ e $ corresponde em quebras de linhas (“ponto corresponde a quebras de linha” não deve estar ativado)

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Substitua por:

```
$1
```

Sabores de texto de substituição: .NET, Java, JavaScript, Perl, PHP

```
\1
```

Sabores de texto de substituição: Python, Ruby

Esta expressão regular utiliza um grupo de captura e uma retrorreferência (entre outros ingredientes) para corresponder a duas ou mais linhas duplicadas sequenciais. Uma retrorreferência é utilizada, no texto de substituição, para colocar a primeira linha de volta. A receita 3.15 mostra um exemplo de código que pode ser remodelado para implementar isso.

Opção 2: Manter a última ocorrência de cada linha duplicada em um arquivo não-ordenado

Se estiver usando um editor de texto que não tenha a capacidade interna de ordenar linhas, ou se for importante preservar a ordem original das linhas, a seguinte solução permite remover duplicatas, mesmo quando separadas por outras linhas:

```
^([^\r\n]*)(?:\r?\n|\r)(?=.*^\1$)
```

Opções Regex: Ponto corresponde a quebras de linha, ^ e $ correspondem em quebras de linha

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

A seguir, temos a mesma coisa como uma expressão regular compatível com JavaScript, sem a exigência da opção “ponto corresponde a quebras de linha”:

```
^(.*)(?:\r?\n|\r)(?=[\s\S]*^\1$)
```

Opções Regex: ^ e $ correspondem em quebras de linha (“ponto corresponde a quebras de linha” não deve estar definido)

Sabores Regex: JavaScript

Substitua por:

(A string vazia, ou seja, nada.)

Sabores de texto de substituição: Não se aplica

Opção 3: Manter a primeira ocorrência de cada linha duplicada em um arquivo não-ordenado

Se quiser preservar a primeira ocorrência de cada linha duplicada, precisará usar uma abordagem diferente. Primeiro, vejamos a expressão regular e a string de substituição que usaremos:

```
^([^\r\n]*)$(.*?)(?:(?:\r?\n|\r)\1$)+
```

Opções Regex: Ponto corresponde a quebras de linhas, ^ e $ correspondem em quebras de linhas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Mais uma vez, precisamos fazer algumas alterações para tornar isso compatível com expressões regulares do sabor JavaScript, uma vez que esta linguagem não possui uma opção “ponto corresponde a quebras de linha”.

```
^(.*)$([\s\S]*?)(?:(?:\r?\n|\r)\1$)+
```

Opções Regex: ^ e $ correspondem em quebras de linhas(“ponto corresponde a quebras de linha” não deve estar definido)

Sabores Regex: JavaScript

Substitua por:

```
$1$2
```

Sabores de texto de substituição: .NET, Java, JavaScript, Perl, PHP

```
\1\2
```

Sabores de texto de substituição: Python, Ruby

Ao contrário das opções de expressão regular 1 e 2, esta versão não remove todas as linhas duplicadas com uma operação de busca-e-substituição. Você precisará aplicar continuamente "substituir tudo", até que a expressão regular não corresponda mais a sua string, o que significa que não existirão mais duplicatas a serem removidas. Consulte a seção "Discussão", desta receita, para mais detalhes.

Discussão

Opção 1: Ordenar linhas e remover duplicatas adjacentes

Esta expressão regular remove todas, menos a primeira, das linhas duplicadas que aparecem umas próximas às outras. Ela não remove as duplicatas separadas por outras linhas. Vamos percorrer o processo.

Primeiro, o acento circunflexo (‹^›), na frente da expressão regular, corresponde ao início de uma linha. Normalmente, ele só corresponderia no início da string de assunto, por isso você precisa se certificar de que a opção "^ e $ correspondem em quebras de linha" esteja habilitada (a receita 3.4 mostra como definir opções de expressão regular). Em seguida, o ‹.*› dentro dos parênteses de captura corresponde a todo o conteúdo de uma linha (mesmo que esteja em branco), e o valor é armazenado como sendo a retrorreferência 1. Para que isso funcione corretamente, a opção "ponto corresponde a quebras de linha" não deve estar definida; caso contrário, a correspondência ponto-asterisco corresponderia até o final da string.

Dentro de um grupo de não-captura exterior, utilizamos ‹(?:\r?\n|\r)› para corresponder a um separador de linha usado em arquivos de texto Windows (‹\r\n›), Unix/Linux/OS X (‹\n›) ou Mac OS (‹\r›). A retrorreferência ‹\1›, então, tenta corresponder à linha que acabamos

de corresponder. Se a mesma linha não for encontrada nessa posição, a tentativa de corresponder falha, e o mecanismo de expressão regular segue em frente. Se corresponder, repetimos o grupo (composto por uma sequência de quebra de linha e pela retrorreferência 1), usando o quantificador ‹+› para tentar corresponder a linhas duplicadas adicionais.

Finalmente, usamos o sinal de cifrão, no final da expressão regular, para declarar a posição no final da linha. Isso garante que só correspondamos a linhas idênticas, e não a linhas que apenas comecem com os mesmos caracteres de uma linha anterior.

Como estamos fazendo uma pesquisa-e-substituição, cada correspondência completa (incluindo a linha original e os separadores de linha) é removida da string. Nós substituímos a correspondência pela retrorreferência 1, para reinserir a linha original.

Opção 2: Manter a última ocorrência de cada linha duplicada em um arquivo não-ordenado

Aqui, há várias alterações em relação à opção 1 de expressão regular no início desta receita, que só encontra linhas duplicadas quando elas aparecem uma ao lado da outra. Em primeiro lugar, na versão não-JavaScript da opção 2 de expressão regular, o ponto dentro do grupo de captura foi substituído por ‹[^\r\n]› (qualquer caractere, exceto uma quebra de linha), e a opção “ponto corresponde a quebras de linha” foi habilitada. Isso porque um ponto é utilizado mais tarde, na expressão regular, para corresponder a qualquer caractere, incluindo quebras de linha. Em segundo lugar, um lookahead foi adicionado, para fazer a varredura de linhas duplicadas em qualquer outra posição ao longo da string. Como o lookahead não consome quaisquer caracteres, o texto correspondido pela expressão regular será sempre uma linha única (junto com sua quebra de linha), que sabemos que aparecerá novamente, mais tarde, na string. Substituir todas as

correspondências pela string vazia remove as linhas duplicadas, deixando para trás apenas a última ocorrência de cada uma delas.

Opção 3: Manter a primeira ocorrência de cada linha duplicada em um arquivo não-ordenado

Como o lookbehind não é tão amplamente suportado quanto o lookahead (e, dependendo de onde é suportado, pode ainda não ser capaz de olhar tão para trás quanto é preciso), a opção 3 de expressão regular é significativamente diferente da opção 2. Em vez de corresponder às linhas repetidas anteriormente na string (o que seria comparável à tática da opção 2), esta expressão regular corresponde a uma linha, à primeira duplicata desta linha, que ocorre mais tarde na string, e a todas as linhas entre elas. A linha original é armazenada como a retrorreferência 1, e as linhas entre elas (se houver) como a retrorreferência 2. Ao substituir cada correspondência por ambas as retrorreferências 1 e 2, você reinsere as partes que deseja manter, deixando de fora a linha duplicada e sua quebra de linha anterior.

Essa abordagem alternativa apresenta alguns problemas. Primeiro, como cada correspondência de um conjunto de linhas duplicadas pode incluir outras linhas entre elas, é possível que existam duplicatas de valor diferente em seu texto correspondido, e elas serão ignoradas durante a operação “substituir tudo”. Segundo, se uma linha é repetida mais de duas vezes, a expressão regular corresponderá, primeiramente, às duplicatas um e dois, mas, depois disso, ela tomará outro conjunto de duplicatas para fazer com que a expressão regular corresponda novamente, na medida em que avança ao longo da string. Assim, uma única ação “substituir tudo”, na melhor das hipóteses, apenas removerá uma duplicata sim, outra não, de qualquer linha específica.

Para resolver esses problemas, e garantir que todas as duplicatas sejam removidas, você precisa aplicar, continuamente, a operação busca-e-substituição em toda a string de assunto, até que a

expressão regular não mais corresponda dentro dela. Pense em como essa expressão regular funcionará, quando aplicada à seguinte string de assunto:

```
value1
value2
value2
value3
value3
value1
value2
```

São necessários três passos para remover todas as linhas duplicadas desta string. A tabela 5.1 nos mostra o resultado de cada passo.

Tabela 5.1 – Passos de substituição

Passo um	Passo dois	Passo três	String final
value1	value1	value1	value1
value2	value2	value2	value2
value2	~~value2~~	value3	value3
value3	value3	~~value2~~	
value3	~~value3~~		
~~value1~~	value2		
value2			
Uma correspondência/ substituição	*Duas correspondências/ substituições*	*Uma correspondências/ substituição*	*Nenhuma duplicata permanece*

Veja também:

Receita 2.10, que discute retrorreferências com mais detalhes.

Receita 5.8, que mostra como corresponder a palavras repetidas.

5.10 Corresponder a linhas inteiras que contenham uma determinada palavra

Problema

Você deseja corresponder a todas as linhas que contenham a palavra ninja, em qualquer lugar dentro delas.

Solução

```
^.*\bninja\b.*$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, ^ e $ correspondem em quebras de linhas (“ponto corresponde a quebras de linha” não deve estar ativado)

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Muitas vezes, é útil corresponder a linhas completas, a fim de coletá-las ou removê-las. Para corresponder a qualquer linha que contenha a palavra ninja, começamos com a expressão regular ‹\bninja\b›. Os tokens de extremidade de palavra, em ambos os lados, garantem que só corresponderemos à palavra “ninja”, quando ela aparecer como uma palavra completa, de acordo com o explicado na receita 2.6.

Para ampliar a expressão regular, fazendo-a corresponder a uma linha completa, adicione ‹.*› nas duas extremidades. As sequências ponto-asterisco correspondem a zero ou mais caracteres na linha atual. Os quantificadores asterisco são mesquinhos; então, eles corresponderão a tanto texto quanto possível. O primeiro ponto-asterisco corresponderá até a última ocorrência de “ninja” na linha, e o segundo ponto-asterisco corresponderá a qualquer caractere que não seja de quebra de linha e que ocorra depois dele.

Por último, coloque as âncoras de acento circunflexo e cifrão no início e no final da expressão regular, respectivamente, para assegurar que as correspondências contenham uma linha completa. Estritamente falando, a âncora de cifrão, no final, é redundante, uma vez que o ponto-asterisco mesquinho sempre corresponde até o final da linha. No entanto, não faz mal adicioná-la, para tornar a expressão regular um pouco mais autoexplicativa. A adição de

âncoras de linha ou de strings às suas expressões regulares, quando apropriado, pode ajudá-lo a evitar problemas inesperados; portanto, não é um mau hábito a se criar. Note que, ao contrário do cifrão, o acento circunflexo no início da expressão regular não é necessariamente redundante, pois ele assegura que a expressão regular corresponderá apenas a linhas completas, mesmo que a pesquisa comece no meio de uma linha, por alguma razão.

Lembre-se de que os três metacaracteres-chaves, usados para restringir as correspondências a uma única linha (as âncoras ‹^› e ‹$›, e o ponto), não possuem significados fixos. Para torná-los todos orientados a linhas, você precisa ativar a opção que permite ‹^› e ‹$› corresponderem a quebras de linha, além de se certificar da não habilitação da opção que permite que o ponto corresponda a quebras de linha. A receita 3.4 mostra como aplicar estas opções em seu código. Se estiver usando JavaScript ou Ruby, existe uma opção a menos com que se preocupar, pois o JavaScript não tem uma opção para deixar o ponto corresponder a quebras de linha, e as âncoras de acento circunflexo e cifrão do Ruby sempre correspondem em quebras de linha.

Variações

Para pesquisar por linhas que contenham qualquer uma das várias palavras, use a alternância:

```
^.*\b(one|two|three)\b.*$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, ^ e $ correspondem em quebras de linhas (“ponto corresponde a quebras de linha” não deve estar ativado)

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

A expressão regular mostrada corresponde a qualquer linha que contenha pelo menos uma das palavras “one”, “two”, ou “three”. Os parênteses em torno das palavras servem para dois propósitos. Primeiro, eles limitam o alcance da alternância e, segundo, eles capturam a palavra que realmente aparece na linha para dentro da retrorreferência 1. Se a linha possuir mais de uma das palavras

dadas, a retrorreferência conterá a que ocorre mais à direita. Isso ocorre porque o quantificador asterisco que aparece antes dos parênteses é mesquinho, e ampliará o ponto para corresponder a tanto texto quanto possível. Se você tornar o asterisco preguiçoso, como em ‹^.*?\b(one|two|three)\b.*$›, a retrorreferência 1 conterá a palavra de sua lista que aparecer mais à esquerda.

Para encontrar linhas que deverão conter várias palavras, use um lookahead:

```
^(?=.*?\bone\b)(?=.*?\btwo\b)(?=.*?\bthree\b).+$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, ^ e $ correspondem em quebras de linhas (“ponto corresponde a quebras de linha” não deve estar ativado)

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Esta expressão regular utiliza um lookahead positivo para corresponder a linhas que possuam três palavras obrigatórias em qualquer lugar dentro destas linhas. O ‹.+›, no final, é utilizado para corresponder à linha, depois de os lookaheads terem determinado que a linha atende aos requisitos.

Veja também:

A receita 5.11 mostra como corresponder a linhas completas que não contenham uma palavra específica.

5.11 Corresponder a linhas completas que não contenham determinada palavra

Problema

Você deseja corresponder a linhas completas que não contenham a palavra ninja.

Solução

```
^(?:(?!\bninja\b).)*$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, ^ e $ correspondem em quebras de linhas (“ponto corresponde a quebras de linha” não deve estar ativado)

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

A fim de corresponder a uma linha que *não* contenha algo, utilize um lookahead negativo (descrito na receita 2.16). Observe que, nesta expressão regular, um lookahead negativo e um ponto são repetidos em conjunto, utilizando um grupo de não-captura. Isso garante que a expressão regular ‹\bninja\b› falhe em qualquer posição da linha. As âncoras ‹^› e ‹$› são colocadas nas extremidades da expressão regular, para garantir que uma linha completa será correspondida.

As opções aplicadas a esta expressão regular determinam se ela tentará corresponder à string de assunto inteira, ou a apenas uma linha de cada vez. Com a opção para deixar ‹^› e ‹$› corresponderem em quebras de linha habilitada, e a opção de deixar o ponto corresponder a quebras de linha desabilitada, esta expressão regular funciona como descrito, correspondendo linha por linha. Se você inverter o estado das duas opções, a expressão regular corresponderá a qualquer string que não contenha a palavra “ninja”.

Testar um lookahead negativo, em cada posição de uma linha ou string, é bastante ineficiente. Esta solução destina-se a ser utilizada apenas em situações nas quais uma expressão regular é tudo o que pode ser usado, como quando utilizamos um aplicativo que não possa ser programado. Quando estiver programando, a receita 3.21 apresenta uma solução muito mais eficiente.

Veja também:

A receita 5.10 mostra como corresponder a linhas completas que contenham uma palavra específica.

5.12 Remover espaços em branco iniciais e finais

Problema

Você deseja remover os espaços em branco iniciais e finais de uma string.

Solução

Para manter as coisas simples e rápidas, a melhor solução é usar duas substituições, uma para remover espaços em branco iniciais, e outra para remover os espaços em branco finais:

```
^\s+
```

Opções Regex: Nenhuma (“^ e $ correspondem em quebras de linha” não deve estar ativado)

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
\s+$
```

Opções Regex: Nenhuma (“^ e $ correspondem em quebras de linha” não deve estar ativado)

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Basta substituir por strings vazias as correspondências encontradas, utilizando as duas expressões regulares. A receita 3.14 mostra como fazer isso. Usando ambas as expressões regulares, você só precisa substituir a primeira correspondência encontrada, já que elas correspondem a todos os espaços em branco iniciais e finais, de uma só vez.

Discussão

Remover espaços em branco iniciais e finais é uma tarefa simples e bem comum. As duas expressões regulares mostradas são compostas por três partes, cada uma: uma âncora para declarar a posição no início ou no final da string (‹^› e ‹$›, respectivamente), a classe de caracteres abreviada, que corresponderá a qualquer caractere em branco (‹\s›), e os quantificadores que repetem a classe, uma ou mais vezes (‹+›).

Muitas linguagens de programação fornecem uma função, geralmente chamada trim ou strip, que consegue remover os espaços

em branco iniciais ou finais. A tabela 5.2 mostra como usar esse método (ou função) nativo em uma variedade de linguagens de programação.

Tabela 5.2 – Funções para remover espaços em branco iniciais, ou finais

Linguagem de programação	Exemplo de uso
C#, VB.NET	String.Trim([*chars*])
Java	string.trim()
PHP	trim($*string*)
Python, Ruby	string.strip()

JavaScript e Perl não possuem função equivalente em suas bibliotecas-padrão, mas você pode, facilmente, criar suas próprias funções.

Em Perl:

```
sub trim {
  my $string = shift;
  $string =~ s/^\s+//;
  $string =~ s/\s+$//;
  return $string;
}
```

Em JavaScript:

```
function trim (string) {
  return string.replace(/^\s+/, '').replace(/\s+$/, '');
}
// Alternativamente, use isto para transformar trim em um método de todas as
strings:
String.prototype.trim = function () {
  return this.replace(/^\s+/, '').replace(/\s+$/, '');
};
```

Tanto em Perl, quanto em JavaScript, ‹\s› corresponde a qualquer caractere definido como espaço em branco pelo padrão Unicode, além do espaço, tabulação, alimentação de linha e caracteres de retorno de carro, mais comumente considerados espaços em branco.

Variações

Há, de fato, várias formas de escrever uma expressão regular para ajudá-lo a cortar uma string. No entanto, elas são invariavelmente mais lentas do que usar duas substituições simples, quando se

trabalha com strings longas (nos casos em que desempenho é mais importante). A seguir, vejamos algumas das soluções alternativas mais comuns. Elas estão todas escritas em JavaScript, e, como o JavaScript não possui uma opção “ponto corresponde a quebras de linha”, as expressões regulares utilizam ‹[\s\S]› para corresponder a qualquer caractere individual, incluindo quebras de linha. Em outras linguagens de programação, use um ponto, e habilite a opção “ponto corresponde a quebras de linha”.

```
string.replace(/^\s+|\s+$/g, '');
```

Esta é, provavelmente, a solução mais comum. Ela combina duas expressões regulares simples por meio de alternância (veja a receita 2.8), e usa a opção /g (global) para substituir todas as correspondências, ao invés de substituir apenas a primeira (ela corresponderá duas vezes, se houver espaços em branco iniciais e finais na string). Esta abordagem não é terrível, mas é mais lenta do que usar duas substituições simples, quando trabalhamos com strings longas.

```
string.replace(/^\s*([\s\S]*?)\s*$/, '$1')
```

Esta expressão regular funciona correspondendo a toda a string e capturando a sequência do primeiro ao último caractere que não seja espaço (se houver) na retrorreferência 1. Ao substituir a string inteira pela retrorreferência 1, você acabará com uma versão cortada dela.

Esta abordagem é conceitualmente simples, mas o quantificador preguiçoso, dentro do grupo de captura, faz o mecanismo de expressão regular realizar um monte de trabalho extra e, portanto, tende a tornar esta opção muito lenta, no caso de strings longas. Após o mecanismo de expressão regular entrar no grupo de captura, durante o processo de correspondência, o quantificador preguiçoso exige que a classe de caracteres ‹[\s\S]› seja repetida um mínimo de vezes. Assim, o mecanismo de expressão regular corresponderá a um caractere por vez, parando depois de cada caractere para tentar corresponder ao padrão restante (‹\s*$›). Se isso falhar, por restarem caracteres

que não sejam espaços em branco, em algum lugar após a posição atual na string, o mecanismo corresponderá a mais um caractere e, em seguida, tentará o restante do padrão novamente.

```
string.replace(/^\s*([\s\S]*\S)?\s*$/, '$1')
```

Isto é semelhante à última expressão regular, mas substitui o quantificador preguiçoso por um mesquinho, por questões de desempenho. A fim de garantir que o grupo de captura ainda corresponderá somente até o último caractere que não seja um espaço em branco, usamos um ‹\S› final obrigatório. No entanto, como a expressão regular precisa ser capaz de corresponder a strings que contenham apenas espaços em branco todo o grupo de captura torna-se opcional com a inclusão de um quantificador de ponto de interrogação no final.

Vamos voltar um pouco, para ver como isso realmente funciona. Aqui, o asterisco mesquinho, em ‹[\s\S]*›, repete o padrão “qualquer caractere” até o final da string. Então, o mecanismo de expressão regular retrocede a partir do final da string, um caractere por vez, até que seja capaz de corresponder ao ‹\S› seguinte, ou até retroceder ao primeiro caractere correspondido dentro do grupo de captura. A menos que haja mais espaços em branco no final do que texto até este ponto, isso geralmente acaba sendo mais rápido do que a solução anterior, que utilizou um quantificador preguiçoso. Ainda assim, não se compara com o desempenho de duas substituições simples.

```
string.replace(/^\s*(\S*(?:\s+\S+)*)\s*$/, '$1')
```

Esta é uma abordagem relativamente comum; por isso, é incluída aqui como um aviso. Não há nenhuma boa razão para usar isso, pois ela é mais lenta do que todas as outras soluções aqui mostradas. É semelhante às duas últimas expressões regulares, na medida em que corresponde à string inteira e a substitui pela parte que você deseja manter, mas, como o grupo interno de não-captura corresponde a apenas uma palavra por vez, há uma série de passos discretos que o mecanismo de expressão regular

deve tomar. O impacto no desempenho provavelmente será imperceptível ao se cortar strings curtas, mas, no caso de strings muito longas, que contenham uma grande quantidade de palavras, esta expressão regular pode se tornar um gargalo no desempenho.

Algumas implementações de expressões regulares possuem otimizações inteligentes, que alteram os processos internos de correspondência descritos aqui e, portanto, fazem com que algumas destas opções tenham desempenho um pouco melhor, ou pior, do que o sugerido. De qualquer forma, a simplicidade na utilização de duas substituições fornece um desempenho consistente e respeitável com diferentes comprimentos e conteúdos de strings, configurando-se ainda, portanto, como a melhor solução.

Veja também:

Receita 5.13.

5.13 Substituir espaços em branco repetidos por um único espaço

Problema

Como parte de uma rotina de limpeza da entrada de usuário ou de outros dados, você deseja substituir caracteres em branco repetidos por um único espaço. Quaisquer tabulações, quebras de linha ou outros espaços em branco também deverão ser substituídos por um espaço.

Solução

Para implementar qualquer uma das seguintes expressões regulares, simplesmente substitua todas as correspondências por um único caractere de espaço. A receita 3.14 mostra o código para tanto.

Limpar quaisquer caracteres de espaço em branco

```
\s+
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Limpar caracteres de espaço em branco horizontais

```
[ \t]+
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Uma rotina comum de limpeza de texto é substituir os caracteres de espaço em branco repetidos por um único espaço. Em HTML, por exemplo, espaços em branco repetidos são simplesmente ignorados quando uma página é processada (com algumas exceções). Então, remover espaços em branco repetidos pode ajudar a reduzir o tamanho de arquivo das páginas sem qualquer efeito negativo.

Limpar quaisquer caracteres de espaço em branco

Nesta solução, qualquer sequência de caracteres em branco (quebras de linha, tabulações, espaços etc) é substituída por um único espaço. Como o quantificador ‹+› repete a classe de espaços em branco (‹\s›) uma ou mais vezes, até mesmo um único caractere de tabulação, por exemplo, será substituído por um espaço. Se você substituir o ‹+› por ‹{2,}›, apenas sequências de dois ou mais caracteres de espaço em branco serão substituídas. Isso poderia resultar em menos substituições, aumentando, assim, o desempenho, mas também poderia deixar para trás caracteres de tabulação ou quebras de linha, que, de outra forma, seriam substituídos por caracteres de espaço. A melhor abordagem,

portanto, depende do que você está tentando realizar.

Limpar caracteres de espaço em branco horizontais

Funciona exatamente como a solução anterior, exceto por não mexer nas quebras de linha. Apenas tabulações e espaços são substituídos.

Veja também:

Receita 5.12.

5.14 Escapar metacaracteres de expressão regular

Problema

Você deseja usar uma string literal, fornecida por um usuário ou por alguma outra fonte, como parte de uma expressão regular. No entanto, você deseja escapar todos os metacaracteres de expressão regular dentro da string antes de colocá-la em sua expressão regular, para evitar consequências inesperadas.

Solução

Ao adicionar uma barra invertida antes de quaisquer caracteres que, potencialmente, possuam um significado especial dentro de uma expressão regular, você poderá usar com segurança o padrão resultante para corresponder a uma sequência literal de caracteres. Das linguagens de programação estudadas neste livro, todas, exceto o JavaScript, possuem uma função interna ou método para realizar esta tarefa (listadas na tabela 5.3). No entanto, para sermos exaustivos, vamos mostrar como conseguir isso usando sua própria expressão regular, mesmo nas linguagens que possuam uma solução pronta.

Soluções nativas

A tabela 5.3 lista as funções nativas projetadas para resolver este problema.

Tabela 5.3 – Soluções nativas para escapar metacaracteres de expressão regular

Linguagem	Função
C#, VB.NET	Regex.Escape(str)
Java	Pattern.quote(str)
Perl	quotemeta(str)
PHP	preg_quote(str, [delimiter])
Python	re.escape(str)
Ruby	Regexp.escape(str)

Notadamente ausente da lista está o JavaScript, que não possui função nativa projetada para esta finalidade.

Expressão regular

Embora seja melhor utilizar uma solução nativa, se estiver disponível, você pode fazer isso por conta própria, utilizando a seguinte expressão regular, juntamente com a string de substituição adequada (mostrada a seguir). Apenas certifique-se de substituir todas as correspondências, ao invés de substituir apenas a primeira. A receita 3.15 mostra o código que substitui as correspondências por uma string contendo uma retrorreferência. Você vai precisar de uma retrorreferência para trazer de volta o caractere especial correspondido, juntamente com uma barra invertida precedente:

```
[[\]{}()*+?.\\|^$\-,&#\s]
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Substituição

As strings de substituição, a seguir, possuem um caractere literal de barra invertida. As strings são mostradas sem as barras invertidas extras, que podem ser necessárias para escapar barras invertidas, quando utilizamos strings literais em algumas linguagens de

programação. Veja a receita 2.19, para obter mais detalhes sobre os sabores de texto de substituição.

```
\$&
```
Sabores de texto de substituição: .NET, JavaScript

```
\\$&
```
Sabores de texto de substituição: Perl

```
\\$0
```
Sabores de texto de substituição: Java, PHP

```
\\\0
```
Sabores de texto de substituição: PHP, Ruby

```
\\\&
```
Sabores de texto de substituição: Ruby

```
\\\g<0>
```
Sabores de texto de substituição: Python

Função de exemplo para JavaScript

Vejamos um exemplo sobre como você pode utilizar a expressão regular, e a string de substituição, para criar um método estático chamado RegExp.escape, em JavaScript:

```
RegExp.escape = function (str) {
  return str.replace(/[[\]{}()*+?.\\|^$\-,&#\s]/g, "\\$&");
};
// Testando a função...
var str = "Hello.World?";
var escaped_str = RegExp.escape(str);
alert(escaped_str == "Hello\\.World\\?"); // -> true
```

Discussão

Esta receita de expressão regular coloca todos os metacaracteres de expressão regular dentro de uma única classe de caracteres. Vamos dar uma olhada em cada um desses caracteres, e examinar por que eles precisam ser escapados. Alguns são menos óbvios do que outros:

```
[] {} ()
```

‹[› e ‹]› criam classes de caracteres. ‹{› e ‹}› criam quantificadores de intervalo e, também, são utilizados em outras construções especiais, como propriedades Unicode. ‹(› e ‹)› são utilizados para agrupamento, captura e outras construções especiais.

* + ?

Estes três caracteres são quantificadores que repetem o elemento precedente zero ou mais vezes, uma ou mais vezes, ou entre zero e uma vez, respectivamente. O ponto de interrogação é usado também após um parêntese de abertura, para a criação de agrupamentos especiais e outras construções (a mesma função é vista no asterisco do Perl 5.10 e do PCRE 7).

. \ |

Um ponto corresponde a qualquer caractere dentro de uma linha ou string, uma barra invertida escapa um caractere especial ou transforma um caractere especial em caractere literal, e a barra vertical alterna entre várias opções.

^ $

Os símbolos de acento circunflexo e cifrão são âncoras que correspondem ao início ou ao final de uma linha ou string. O acento circunflexo também pode negar uma classe de caracteres.

Os caracteres restantes, correspondidos pela expressão regular, são especiais apenas em circunstâncias adequadas. Eles foram incluídos por uma questão de prudência.

-

Um hífen cria um intervalo dentro de uma classe de caracteres. É escapado, aqui, para evitar, inadvertidamente, a criação de intervalos ao incorporar o texto no meio de uma classe de caracteres. Tenha em mente que, se você inserir um texto dentro de uma classe de caracteres, a expressão regular resultante não corresponderá à string incorporada, mas, sim, a qualquer um dos caracteres da string incorporada.

,

A vírgula é usada dentro de um quantificador de intervalo, como em ‹{1,5}›. Como a maioria dos sabores de expressão regular trata as chaves como caracteres literais, caso elas não formem um quantificador válido, é possível (embora pouco improvável) criar um quantificador onde não havia nenhum, ao inserir texto literal em uma expressão regular sem escapar as vírgulas.

&

O caractere de conjunção & foi incluído na lista porque são utilizados dois && adjacentes, para a intersecção de classes de caracteres em Java. Em outras linguagens de programação, é seguro remover o & da lista de caracteres que precisam ser escapados, mas não faz mal algum mantê-lo.

e espaço em branco

O sinal # e o espaço em branco (correspondido por ‹\s›) serão metacaracteres somente se a opção de espaçamento livre estiver habilitada. Novamente, não faz mal algum escapá-los.

Quanto ao texto de substituição, um dos cinco tokens («$&», «\&», «$0», «\0», ou «\g<0>») é utilizado para restaurar o caractere correspondido, juntamente com uma barra invertida o precedendo. Em Perl, $& é realmente uma variável, e utilizá-la com qualquer expressão regular impõe uma penalidade global de desempenho em todas as expressões regulares. Se $& for utilizado em outro lugar em seu programa Perl, não haverá problemas, pois você já pagou o preço por ela. Caso contrário, provavelmente seria melhor encapsular toda a expressão regular em um grupo de captura, e utilizar $1, em vez de $&, na substituição.

Variações

Como explicado em “Escape em bloco”, receita 2.1, você pode criar uma sequência de escape em bloco, dentro de uma expressão regular, utilizando ‹\Q...\E›. No entanto, escapes em bloco só são suportados por Java, PCRE e Perl e, mesmo nestas linguagens, escapes em bloco não são infalíveis. Para ter segurança completa, você precisaria escapar qualquer ocorrência de \E dentro da string

que você pretende incorporar, em sua expressão regular. Na maioria dos casos, provavelmente é mais fácil usar apenas a abordagem de compatibilidade entre linguagens, escapando todos os metacaracteres da expressão regular.

Veja também:

A receita 2.1 discute como corresponder a caracteres literais. Nela, a lista de caracteres que precisam ser escapados é menor, uma vez que ela não se preocupa com os caracteres que talvez precisem ser escapados, no modo de espaçamento livre, ou que são jogados dentro de um padrão mais longo e arbitrário.

CAPÍTULO 6

Números

As expressões regulares são projetadas para lidar com texto. Elas não entendem os significados que nós, humanos, atribuímos a sequências de dígitos. Para uma expressão regular, 56 não é o número cinquenta e seis, mas uma string de dois caracteres, exibida como os dígitos 5 e 6. O mecanismo de expressão regular sabe que são dígitos, pois a classe de caracteres abreviada ‹\d› corresponde a eles (veja receita 2.3). Mas é só isso. Ela não sabe que 56 possui um significado maior, assim como não sabe que :-) representa algo mais do que três caracteres de pontuação correspondidos por ‹\p{P}{3}›.

Porém, os números formam algumas das entradas mais importantes com as quais, provavelmente, você lidará e, às vezes você precisará processá-los dentro de uma expressão regular, e não apenas passá-los para uma linguagem de programação convencional, quando quiser respostas a perguntas como: “Será que este número está dentro do intervalo de 1 a 100?”. Então, dedicamos um capítulo inteiro para correspondências a todos os tipos de números usando expressões regulares. Começamos com algumas receitas que podem parecer triviais, mas, na verdade, são importantes para explicar conceitos básicos. As receitas posteriores, que lidam com expressões regulares mais complexas, assumem que você absorveu estes conceitos básicos.

6.1 Números inteiros

Problema

Você deseja encontrar vários tipos de números decimais inteiros em

um corpo de texto maior, ou verificar se a variável de string contém um número decimal inteiro.

Solução

Encontre um número decimal inteiro positivo em um corpo de texto maior:

```
\b[0-9]+\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Verifique se uma string de texto contém um número decimal inteiro positivo:

```
\A[0-9]+\Z
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^[0-9]+$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Encontre qualquer número decimal inteiro positivo que apareça isolado em um corpo de texto maior:

```
(?<=^|\s)[0-9]+(?=$|\s)
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby 1.9

Encontre qualquer número decimal inteiro positivo que apareça isolado em um corpo de texto maior, permitindo que espaços em branco iniciais sejam incluídos na correspondência da expressão regular:

```
(^|\s)([0-9]+)(?=$|\s)
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Encontre qualquer número decimal com um sinal opcional de adição, ou de subtração, a sua esquerda:

```
[+-]?\b[0-9]+\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Verifique se uma string de texto contém apenas um número decimal inteiro com sinal opcional:

```
\A[+-]?[0-9]+\Z
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^[+-]?[0-9]+$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Encontre qualquer número decimal inteiro com sinal opcional, permitindo espaços em branco entre o número e o sinal, mas sem permitir espaços em branco à esquerda quando não houver um sinal:

```
([+-] *)?\b[0-9]+\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Um número inteiro é uma série contígua de um ou mais dígitos, cada um entre zero e nove. Nós podemos, facilmente, representar isso com uma classe de caracteres (Receita 2.3) e um quantificador (Receita 2.12): ‹[0-9]+›.

> Preferimos utilizar o intervalo explícito ‹[0-9]›, em vez da abreviação ‹\d›. Em .NET e Perl, ‹\d› corresponde a qualquer dígito, em qualquer sistema de escrita, mas ‹[0-9]› sempre corresponde apenas aos 10 dígitos da tabela ASCII. Se você sabe que seu texto de assunto não inclui um dígito que não seja ASCII, poderá economizar digitações e usar ‹\d›, em vez de ‹[0-9]›.

> Caso não saiba se o assunto vai incluir dígitos fora da tabela ASCII, você precisa pensar sobre o que pretende fazer com as correspondências da expressão regular e quais são as expectativas do usuário, a fim de decidir se deve usar ‹\d› ou ‹[0-9]›. Se você pretende converter o texto correspondido pela expressão regular em um número inteiro, verifique se a função de conversão de string-para-inteiro, da sua linguagem de programação, pode interpretar dígitos que não sejam ASCII. Usuários escrevendo documentos em seus sistemas de escrita nativos esperam que o software reconheça dígitos em seus sistemas nativos.

Além de ser uma série de dígitos, o número também deve estar

isolado. A4 é um tamanho de papel, não um número. Há várias maneiras de ter certeza de que sua expressão regular corresponderá somente a números puros.

Se quiser verificar se sua string não contém nada além de um número, basta colocar as âncoras início-de-string e final-de-string em torno de sua expressão regular. ‹\A› e ‹\Z› são suas melhores opções, porque seu significado não muda. Infelizmente, o JavaScript não as suporta. Em JavaScript, use ‹^› e ‹$›, e certifique-se de não especificar a sinalização /m, que faz o acento circunflexo e o cifrão corresponderem em quebras de linha. Em Ruby, o acento circunflexo e o cifrão sempre correspondem em quebras de linha, então você não pode utilizá-los de forma confiável, para forçar que sua expressão regular corresponda a toda a string.

Ao procurar por números dentro de um conjunto maior de texto, as extremidades de palavra (receita 2.6) são uma solução fácil. Ao colocá-las antes ou depois de um símbolo de expressão regular que corresponda a um dígito, a extremidade de palavra garante que não haja nenhum caractere de palavra antes ou depois do dígito correspondido. Por exemplo, ‹4› corresponde a 4, em A4. ‹4\b› também, porque não há nenhum caractere de palavra após o 4. ‹\b4› e ‹\b4\b› não correspondem a A4, porque ‹\b› falha entre os dois caracteres de palavra, A e 4. Em expressões regulares, caracteres de palavra incluem letras, dígitos e underscores.

Se você incluir caracteres que não sejam de palavra, como sinais de adição e subtração ou espaços em branco, em sua expressão regular, precisará ter cuidado com a colocação das extremidades de palavra. Para corresponder a +4, excluindo +4B, utilize ‹\+4\b›, em vez de ‹\b\+4\b›. Este último não corresponde a +4, pois não há nenhum caractere de palavra antes do sinal de adição, na string de assunto, para satisfazer a extremidade de palavra. Além disso, ‹\b\+4\b› corresponde a +4 no texto 3+4, porque 3 é um caractere de palavra, e + não.

‹\+4\b› só precisa de uma extremidade de palavra. O primeiro ‹\b›, em

‹\+\b4\b›, é supérfluo. Quando esta expressão regular corresponde, o primeiro ‹\b› está sempre entre um + e um 4, e assim nunca exclui nada. O primeiro ‹\b› torna-se importante quando o sinal de adição é facultativo. ‹\+?\b4\b› não corresponde a 4 em A4, enquanto ‹\+?4\b›, sim.

Extremidades de palavra nem sempre são a solução correta. Considere o texto de assunto $123,456.78. Se você iterar essa string com a expressão regular ‹\b[0-9]+\b›, ela corresponderá a 123, 456 e 78. O sinal de cifrão, a vírgula e o ponto decimal não são caracteres de palavra, de modo que a extremidade de palavra corresponde entre um dígito e qualquer um desses caracteres. Às vezes, é isso o que você quer; outras vezes, não.

Se você apenas deseja encontrar inteiros cercados por espaços em branco, ou o início/final de uma string, precisará usar um lookaround, em vez de extremidades de palavra. ‹(?=$|\s)› corresponde no final da string ou antes de um caractere que seja um espaço em branco (espaços em branco incluem quebras de linha). ‹(?<=^|\s)› corresponde no início da string, ou depois de um caractere que seja um espaço em branco. Você pode substituir ‹\s› por uma classe de caracteres que corresponda a qualquer um dos caracteres que deseja permitir antes ou depois do número. Veja a receita 2.16 para aprender como funciona um lookaround.

JavaScript e Ruby 1.8 não suportam lookbehind. Você pode usar um grupo normal, no lugar de um lookbehind, para verificar se o número ocorre no início da string ou se é precedido por espaços em branco. A desvantagem é que o caractere de espaço em branco será incluído na correspondência global da expressão regular, se o número não ocorrer no início da string. Uma solução fácil seria colocar a parte da expressão regular que corresponde ao número dentro de um grupo de captura. A quinta expressão regular, na seção “Solução”, captura o caractere de espaço em branco no primeiro grupo de captura, e o inteiro correspondido no segundo grupo de captura.

Veja também:

Receitas 2.3 e 2.12.

6.2 Números hexadecimais

Problema

Você deseja encontrar números hexadecimais em um corpo de texto maior, ou verificar se uma variável de string contém um número hexadecimal.

Solução

Encontrar qualquer número hexadecimal em um corpo de texto maior:

```
\b[0-9A-F]+\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
\b[0-9A-Fa-f]+\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Verificar se uma string de texto contém apenas um número hexadecimal:

```
\A[0-9A-F]+\Z
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^[0-9A-F]+$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Encontrar um número hexadecimal com um prefixo `0x`:

```
\b0x[0-9A-F]+\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Encontrar um número hexadecimal com um prefixo `&H`:

```
&H[0-9A-F]+\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Encontrar um número hexadecimal com um sufixo H:

```
\b[0-9A-F]+H\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Encontrar um valor de byte hexadecimal, ou um número de 8 bits:

```
\b[0-9A-F]{2}\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Encontrar um valor de palavra hexadecimal, ou um número de 16 bits:

```
\b[0-9A-F]{4}\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Encontrar um valor de palavra dupla hexadecimal, ou um número de 32 bits:

```
\b[0-9A-F]{8}\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Encontre um valor de palavra quádrupla hexadecimal, ou um número de 64 bits:

```
\b[0-9A-F]{16}\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Encontrar uma string de bytes hexadecimais (ou seja, um número par de dígitos hexadecimais):

```
\b(?:[0-9A-F]{2})+\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

As técnicas para corresponder a inteiros hexadecimais com uma expressão regular são as mesmas utilizadas para corresponder a inteiros decimais. A única diferença é que a classe de caracteres que corresponde a um único dígito agora precisa incluir as letras de A a F. Você precisa considerar se as letras deverão ser maiúsculas ou minúsculas, ou se é permitido misturá-las. As expressões regulares mostradas aqui permitem a mistura de maiúsculas e minúsculas.

Por padrão, expressões regulares diferenciam maiúsculas de minúsculas. ‹[0-9a-f]› permite apenas dígitos hexadecimais minúsculos, e ‹[0-9A-F]› permite apenas dígitos hexadecimais maiúsculos. Para permitir maiúsculas e minúsculas, use ‹[0-9a-fA-F]› ou ative a opção para tornar suas expressões regulares indiferentes a maiúsculas e minúsculas. A receita 3.4 explica como fazer isso nas linguagens de programação estudadas neste livro. A primeira expressão regular, na solução, é mostrada duas vezes, empregando as duas maneiras diferentes de fazê-la diferenciar maiúsculas e minúsculas. As outras expressões regulares mostradas utilizam apenas o segundo método.

Se você deseja permitir apenas letras maiúsculas em números hexadecimais, utilize as expressões regulares com a opção de não-diferenciação entre maiúsculas e minúsculas desabilitada. Para permitir somente letras minúsculas, desabilite a não-diferenciação de maiúsculas e minúsculas, e substitua ‹A-F› por ‹a-f›.

‹(?:[0-9A-F]{2})+› corresponde a um número par de dígitos hexadecimais. ‹[0-9A-F]{2}› corresponde a exatamente dois dígitos hexadecimais. ‹(?:[0-9A-F]{2})+› faz isso uma ou mais vezes. O grupo de não-captura (veja receita 2.9) é necessário, porque o sinal de adição precisa repetir a combinação da classe de caracteres e o quantificador ‹{2}›. ‹[0-9]{2}+› não configura erro de sintaxe em Java, PCRE e Perl 5.10, mas não faz o que você quer. O ‹+› extra torna o ‹{2}› possessivo. Tal recurso não tem efeito, porque, de qualquer

maneira, ‹{2}› não pode repetir menos do que duas vezes.

Várias das soluções apresentadas demonstram como exigir que o número hexadecimal tenha um dos prefixos, ou sufixos, comumente usados para identificar números hexadecimais. Eles são usados para diferenciar entre números decimais e hexadecimais que consistam apenas em dígitos decimais. Por exemplo, 10 pode ser o número decimal entre 9 e 11, ou o número hexadecimal entre F e 11.

A maioria das soluções é apresentada com extremidades de palavra (Receita 2.6). Use extremidades de palavra, como mostrado, para encontrar números dentro de um corpo maior de texto. Observe que a expressão regular utilizando o prefixo &H não possui uma extremidade de palavra no início, pois o sinal de conjunção (&) não é uma extremidade de palavra. Se colocássemos uma extremidade de palavra no início dessa expressão regular, ela encontraria somente números hexadecimais imediatamente após um caractere de palavra.

Se quiser verificar se a sua string contém apenas números hexadecimais, basta colocar âncoras de início-de-string e final-de-string em torno de sua expressão regular. ‹\A› e ‹\Z› são suas melhores opções, pois seus significados não mudam. Infelizmente, o JavaScript não as suporta. Em JavaScript, use ‹^› e ‹$›, e certifique-se de não especificar a sinalização /m, que faz o acento circunflexo e o sinal de cifrão corresponderem em quebras de linhas. Em Ruby, o circunflexo e o cifrão sempre correspondem em quebras de linha; então, você não pode usá-los de forma confiável, para forçar sua expressão regular a corresponder a toda a string.

Veja também:

Receitas 2.3 e 2.12.

6.3 Números binários

Problema

Você deseja encontrar números binários em um corpo de texto maior, ou verificar se uma variável de string contém um número binário.

Solução

Encontrar um número binário em um corpo de texto maior:

```
\b[01]+\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Verificar se uma string de texto contém apenas um número binário:

```
\A[01]+\Z
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^[01]+$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Encontrar um número binário com um sufixo B:

```
\b[01]+B\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Encontrar um valor de byte binário, ou um número de 8 bits:

```
\b[01]{8}\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Encontrar um valor de palavra binária, ou um número de 16 bits:

```
\b[01]{16}\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Encontrar uma string de bytes (por exemplo, um múltiplo de oito bits):

```
\b(?:[01]{8})+\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Todas essas expressões regulares utilizam as técnicas explicadas nas duas receitas anteriores. A maior diferença é que cada dígito, agora, é um 0 ou um 1. Correspondemos facilmente a isso com uma classe de caracteres que inclua apenas estes dois caracteres: ‹[01]›.

Veja também:

Receitas 2.3 e 2.12.

6.4 Remover zeros à esquerda

Problema

Você deseja corresponder a um número inteiro, e quer retornar o número sem os zeros à esquerda, ou eliminar os zeros à esquerda.

Solução

Expressão regular

```
\b0*([1-9][0-9]*|0)\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Substituição

```
$1
```

Sabores de texto de substituição: .NET, Java, JavaScript, PHP, Perl

```
\1
```

Sabores de texto de substituição: PHP, Python, Ruby

Obtendo os números em Perl

```
while ($subject =~ m/\b0*([1-9][0-9]*|0)\b/g) {
  push(@list, $1);
```

```
}
```

Removendo zeros à esquerda em PHP

```
$result = preg_replace('/\b0*([1-9][0-9]*|0)\b/', '$1', $subject);
```

Discussão

Utilizamos um grupo de captura para separar um número de zeros que apareçam à sua esquerda. Antes do grupo, ‹0*› corresponde aos zeros à esquerda, se houver. Dentro do grupo, ‹[1-9][0-9]*› corresponde a um número constituído por um ou mais dígitos, com o primeiro dígito não sendo zero. O número pode começar com um zero somente se ele, em si, for zero. As extremidades de palavra asseguram que não corresponderemos a números parciais, como explicado na receita 6.1.

Para obter uma lista de todos os números sem os zeros à esquerda no texto de assunto, itere as correspondências da expressão regular, como explicado na receita 3.11. Dentro do loop, recupere o texto correspondido pelo primeiro (e único) grupo de captura, como explicado na receita 3.9. A solução desta receita mostra como você poderia fazer isso em Perl.

Remover os zeros à esquerda é fácil com uma pesquisa-e-substituição. Nossa expressão regular possui um grupo de captura que separa o número de seus zeros à esquerda. Se substituirmos a correspondência global da expressão regular (o número, incluindo os zeros à esquerda) pelo texto correspondido pelo primeiro grupo de captura, efetivamente removeremos os zeros à esquerda. A solução mostra como fazer isso em PHP. A receita 3.15 mostra como fazê-lo em outras linguagens de programação.

Veja também:

Receitas 3.15 e 6.1.

6.5 Números dentro de um certo

intervalo

Problema

Você deseja corresponder a um número inteiro dentro de um certo intervalo de números, e quer que a expressão regular especifique o intervalo com precisão, ao invés de apenas limitar a quantidade de dígitos.

Solução

1 a 12 (hora ou mês):

```
^(1[0-2]|[1-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

1 a 24 (hora):

```
^(2[0-4]|1[0-9]|[1-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

1 a 31 (dia do mês):

```
^(3[01]|[12][0-9]|[1-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

1 a 53 (semana do ano):

```
^(5[0-3]|[1-4][0-9]|[1-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

0 a 59 (minuto ou segundo):

```
^[1-5]?[0-9]$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

0 a 100 (porcentagem):

```
^(100|[1-9]?[0-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

1 a 100:

```
^(100|[1-9][0-9]?)$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

32 a 126 (códigos ASCII imprimíveis):

```
^(12[0-6]|1[01][0-9]|[4-9][0-9]|3[2-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

0 a 127 (byte sinalizado não-negativo):

```
^(12[0-7]|1[01][0-9]|[1-9]?[0-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

-128 a 127 (byte sinalizado):

```
^(12[0-7]|1[01][0-9]|[1-9]?[0-9]|-(12[0-8]|1[01][0-9]|[1-9]?[0-9]))$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

0 a 255 (byte não-sinalizado):

```
^(25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9]?[0-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

1 a 366 (dia do ano):

```
^(36[0-6]|3[0-5][0-9]|[12][0-9]{2}|[1-9][0-9]?)$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

1900 a 2099 (ano):

```
^(19|20)[0-9]{2}$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

0 a 32767 (palavra sinalizada não-negativa):

```
^(3276[0-7]|327[0-5][0-9]|32[0-6][0-9]{2}|3[01][0-9]{3}|[12][0-9]{4}|↲
[1-9][0-9]{1,3}|[0-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

-32768 a 32767 (palavra sinalizada):

```
^(3276[0-7]|327[0-5][0-9]|32[0-6][0-9]{2}|3[01][0-9]{3}|[12][0-9]{4}|↲
[1-9][0-9]{1,3}|[0-9]|-(3276[0-8]|327[0-5][0-9]|32[0-6][0-9]{2}|↲
3[01][0-9]{3}|[12][0-9]{4}|[1-9][0-9]{1,3}|[0-9]))$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

0 a 65535 (palavra não-sinalizada):

```
^(6553[0-5]|655[0-2][0-9]|65[0-4][0-9]{2}|6[0-4][0-9]{3}|[1-5][0-9]{4}|↲
[1-9][0-9]{1,3}|[0-9])$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

As receitas anteriores correspondiam a inteiros com qualquer número de dígitos, ou com um determinado número de dígitos. Elas permitiam o intervalo completo para todos os dígitos do número. Tais expressões regulares são muito simples.

Corresponder um número a um intervalo específico – por exemplo, um número entre 0 e 255 – não é uma tarefa simples para fazer com expressões regulares. Você não pode escrever ‹[0-255]›. Bem, você poderia, mas não corresponderia a um número entre 0 e 255. Tal classe de caracteres, equivalente a ‹[0125]›, corresponde a um único caractere que seja um 0, 1, 2, ou 5.

Como estas expressões regulares são um pouco mais longas, todas as soluções utilizam âncoras para tornar a expressão regular adequada a verificar se uma string, como uma entrada de usuário, consiste em um único número aceitável. A receita 6.1 explica como você pode usar extremidades de palavra ou lookarounds, em vez de âncoras, para outros fins. Na discussão, nós mostramos as expressões regulares sem as âncoras, mantendo o foco sobre como lidar com intervalos numéricos. Se quiser usar qualquer uma dessas expressões regulares, terá de adicionar âncoras ou extremidades de palavra, para garantir que sua expressão regular não corresponda a dígitos que façam parte de um número mais longo.

Expressões regulares trabalham caractere a caractere. Se quisermos corresponder a um número que tenha mais de um dígito, teremos de enunciar todas as combinações possíveis dos dígitos. Os blocos de construção essenciais são as classes de caracteres (receita 2.3) e a alternância (receita 2.8).

Em classes de caracteres, podemos utilizar intervalos de um dígito, como ‹[0-5]›, porque os caracteres para os dígitos de 0 a 9 ocupam posições consecutivas nas tabelas ASCII e Unicode. ‹[0-5]› corresponde de um a seis caracteres, como ‹[j-o]› e ‹[\x09-\x0E]› correspondem a diferentes intervalos de seis caracteres.

Quando um intervalo numérico é representado como texto, ele consiste em uma certa quantidade de posições. Cada posição permite um certo intervalo de dígitos. Algumas faixas possuem uma quantidade fixa de posições, como de 12 a 24. Outros possuem uma quantidade variável de posições, como de 1 a 12. A gama de dígitos permitidos em cada posição pode ser interdependente, ou independente, dos dígitos em outras posições. No intervalo de 40 a 59, as posições são independentes. No intervalo de 44 a 55, as posições são interdependentes.

Os intervalos mais fáceis são aqueles com uma quantidade fixa de posições independentes, como de 40 a 59. Para codificá-los como uma expressão regular, tudo que precisa fazer é reunir um grupo de classes de caracteres. Utilize uma classe de caracteres para cada posição, especificando a série de dígitos permitidos naquela posição.

```
[45][0-9]
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

O intervalo de 40 a 59 exige um número com dois dígitos. Assim, temos duas classes de caracteres. O primeiro dígito deve ser um 4 ou 5. A classe de caracteres ‹[45]› corresponde a um desses dois dígitos. O segundo dígito pode ser qualquer um dos 10 dígitos. ‹[0-9]› faz o serviço.

Também poderíamos ter usado a abreviação ‹\d›, em vez de ‹[0-9]›. Usamos o intervalo explícito ‹[0-9]› por uma questão de consistência, em relação às outras classes de caracteres, para ajudar a manter a legibilidade. Reduzir o número de barras invertidas em suas expressões regulares também é muito útil se você estiver trabalhando com uma linguagem de programação como Java, que exige que barras invertidas sejam escapadas em strings literais.

Os números no intervalo de 44 a 55 também precisam de duas posições, mas eles não são independentes. O primeiro dígito deverá ser 4 ou 5. Se o primeiro dígito for 4, o segundo deverá estar entre 4 e 9, que cobre os números de 44 a 49. Se o primeiro dígito for 5, o segundo deverá estar entre 0 e 5. Isso abrange os números de 50 a 55. Para criar nossa expressão regular, podemos simplesmente usar uma alternância para combinar os dois intervalos:

```
4[4-9]|5[0-5]
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ao utilizar a alternância, estamos dizendo ao mecanismo de expressão regular para corresponder a ‹4[4-9]›, ou a ‹5[0-5]›. O operador de alternância tem a prioridade mais baixa entre todos os operadores de expressão regular e, assim, não precisamos agrupar os dígitos, como em ‹(4[4-9])|(5[0-5)›.

Você pode reunir, usando alternâncias, tantos intervalos quanto quiser. O intervalo de 34 a 65 também possui duas posições interdependentes. O primeiro dígito deve estar entre 3 e 6. Se o primeiro dígito for 3, o segundo deverá estar entre 4 e 9. Se o primeiro for 4 ou 5, o segundo poderá ser qualquer dígito. Se o primeiro for 6, o segundo deverá estar entre 0 e 5:

```
3[4-9]|[45][0-9]|6[0-5]
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Assim como utilizamos a alternância para dividir os intervalos com posições interdependentes em vários intervalos com posições independentes, podemos utilizá-la para dividir os intervalos, com uma quantidade variável de posições, em vários intervalos com uma quantidade fixa de posições. O intervalo de 1 a 12 possui números

com uma ou duas posições. Faremos essa divisão no intervalo de 1 a 9, com uma posição, e no intervalo de 10 a 12, com duas posições. As posições de cada um desses dois intervalos são independentes, de modo que não precisamos dividi-los ainda mais:

```
1[0-2]|[1-9]
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Listamos o intervalo com dois dígitos antes do intervalo com um único dígito. O uso desse recurso é intencional, porque o mecanismo de expressão regular é *ansioso*. Ele examina as alternativas, da esquerda para a direita, para logo corresponder a uma das alternativas. Se seu texto for 12, ‹1[0-2|[1-9]› corresponderá a 12, enquanto ‹[1-9]|1[0-2]› corresponderá apenas a ‹1›. A primeira alternativa, ‹[1-9]›, é tentada primeiro. Como essa alternativa fica feliz em corresponder a apenas 1, o mecanismo de expressão regular nunca tenta verificar se ‹1[0-2]› pode oferecer uma solução “melhor”.

Alguns mecanismos de expressão regular não são ansiosos

Mecanismos de expressão regular compatíveis com POSIX e mecanismos de expressão regular DFA não seguem essa regra. Eles tentam todas as alternativas, e devolvem a que encontrar a correspondência mais longa. No entanto, todos os sabores discutidos neste livro são os mecanismos NFA, que não fazem o trabalho extra exigido pelo POSIX. Todos eles dirão que ‹[1-9]|1[0-2]› corresponde a 1 em 12.

Na prática, normalmente você vai utilizar âncoras ou extremidades de palavra em sua lista de alternativas. Então, a ordem das alternativas realmente não importa. ‹^([1-9]|1[0-2])$› e ‹^(1[0-2]|[1-9])$› correspondem a 12 em 12, com todos os sabores de expressão regular deste livro, bem como em expressões regulares “estendidas” do POSIX e mecanismos DFA. As âncoras exigem que a expressão regular corresponda a toda a string, ou não corresponda a nada. DFA e NFA são definidos na barra lateral “História do Termo ‘Expressão Regular’”, capítulo 1.

O intervalo de 85 a 117 inclui números de dois comprimentos diferentes. O intervalo de 85 a 99 possui duas posições, e o de 100 a 117, três posições. As posições, nestes intervalos, são interdependentes e, por isso, precisamos dividi-los ainda mais. No caso do intervalo de dois dígitos, se o primeiro for 8, o segundo deverá estar entre 5 e 9. Se o primeiro for 9, o segundo pode ser qualquer dígito. No caso do intervalo de três dígitos, a primeira

posição permite apenas o dígito 1. Se a segunda posição contiver o dígito 0, a terceira permitirá qualquer um. Porém, se o segundo dígito for 1, o terceiro deverá estar entre 0 e 7. Isso nos dá quatro intervalos no total: de 85 a 89, de 90 a 99, de 100 a 109 e de 110 a 117. Embora as coisas estejam ficando excessivamente longas, a expressão regular continua tão simples quanto as anteriores:

```
8[5-9]|9[0-9]|10[0-9]|11[0-7]
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

O que apresentamos é tudo o que existe para correspondência de intervalos numéricos com expressões regulares: basta dividir o intervalo, até ter intervalos com uma quantidade fixa de posições com dígitos independentes. Dessa forma, você sempre obterá uma expressão regular correta que seja fácil de ler e manter, mesmo que ela possa ficar excessivamente longa.

Existem algumas técnicas extras que permitem uma expressão regular mais curta. Por exemplo, utilizando o sistema anterior, o intervalo de 0 a 65535 exigiria esta expressão regular:

```
6553[0-5]|655[0-2][0-9]|65[0-4][0-9][0-9]|6[0-4][0-9][0-9][0-9]|↲
[1-5][0-9][0-9][0-9][0-9]|[1-9][0-9][0-9][0-9]|[1-9][0-9][0-9]|↲
[1-9][0-9]|[0-9]
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Essa expressão regular funciona perfeitamente, e você não será capaz de ter uma expressão regular que funcione consideravelmente mais rápido. Qualquer otimização a ser feita (por exemplo, existem várias alternativas começando com um 6) já foi realizada pelo mecanismo, quando ele compila sua expressão regular. Não há nenhuma necessidade de desperdiçar seu tempo para tornar sua expressão regular mais complicada, na esperança de torná-la mais rápida. Porém, você pode encurtar sua expressão regular para reduzir a quantidade de digitação, enquanto a mantém legível.

Várias das alternativas possuem classes de caracteres idênticas, próximas umas das outras. Você poderia eliminar as duplicações utilizando quantificadores. A receita 2.12 diz tudo sobre eles.

```
6553[0-5]|655[0-2][0-9]|65[0-4][0-9]{2}|6[0-4][0-9]{3}|[1-5][0-9]{4}|↲
[1-9][0-9]{3}|[1-9][0-9]{2}|[1-9][0-9]|[0-9]
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

A parte ‹[1-9][0-9]{3}|[1-9][0-9]{2}|[1-9][0-9]› da expressão regular possui três alternativas muito similares, e todas possuem o mesmo par de classes de caracteres. A única diferença é o número de vezes em que a segunda classe é repetida. Podemos, facilmente, combinar isto em ‹[1-9][0-9]{1,3}›.

```
6553[0-5]|655[0-2][0-9]|65[0-4][0-9]{2}|6[0-4][0-9]{3}|[1-5][0-9]{4}|↲
[1-9][0-9]{1,3}|[0-9]
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Qualquer outro truque prejudicará a leitura. Por exemplo, você poderia isolar o 6 à esquerda das primeiras quatro alternativas:

```
6(?:553[0-5]|55[0-2][0-9]|5[0-4][0-9]{2}|[0-4][0-9]{3})|[1-5][0-9]{4}|↲
[1-9][0-9]{1,3}|[0-9]
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

No entanto, esta expressão regular, na verdade, acaba ficando um caractere mais longo, pois tivemos que adicionar um grupo de não-captura, para isolar as alternativas com o 6 inicial das outras opções. Você não terá ganho de desempenho em qualquer um dos sabores de expressão regular discutidos neste livro. Todos fazem essa otimização internamente.

Veja também:

Receitas 2.8, 4.12 e 6.1.

6.6 Números hexadecimais dentro de um certo intervalo

Problema

Você deseja corresponder a um número hexadecimal em um determinado intervalo de números, e quer que a expressão regular especifique o intervalo com precisão, ao invés de apenas limitar a quantidade de dígitos.

Solução

1 a C (1 a 12: hora ou mês):

```
^[1-9a-c]$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

1 a 18 (1 a 24: hora):

```
^(1[0-8]|[1-9a-f])$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

1 a 1F (1 a 31: dia do mês):

```
^(1[0-9a-f]|[1-9a-f])$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

1 a 35 (1 a 53: semana do ano):

```
^(3[0-5]|[12][0-9a-f]|[1-9a-f])$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

0 a 3B (0 a 59: minuto, ou segundo):

```
^(3[0-9a-b]|[12]?[0-9a-f])$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

0 a 64 (0 a 100: porcentagem):

```
^(6[0-4]|[1-5]?[0-9a-f])$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

1 a 64 (1 a 100):

```
^(6[0-4]|[1-5][0-9a-f]|[1-9a-f])$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

20 a 7E (32 a 126: códigos ASCII imprimíveis):

```
^(7[0-9a-e]|[2-6][0-9a-f])$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

0 a 7F (0 a 127: número de 7 bits):

```
^[1-7]?[0-9a-f]$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

0 a FF (0 a 255: número de 8 bits):

```
^[1-9a-f]?[0-9a-f]$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

1 a 16E (1 a 366: dia do ano):

```
^(16[0-9a-e]|1[0-5][0-9a-f]|[1-9a-f][0-9a-f]?)$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

76C a 833 (1900 a 2099: ano):

```
^(83[0-3]|8[0-2][0-9a-f]|7[7-9a-f][0-9a-f]|76[c-f])$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

0 a 7FFF: (0 a 32767: número de 15 bits):

```
^([1-7][0-9a-f]{3}|[1-9a-f][0-9a-f]{1,2}|[0-9a-f])$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

0 a FFFF: (0 a 65535: número de 16 bits):

```
^([1-9a-f][0-9a-f]{1,3}|[0-9a-f])$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Não há nenhuma diferença entre corresponder, com uma expressão regular, intervalos numéricos decimais e intervalos numéricos hexadecimais. Como explicado na receita anterior, divida o intervalo em várias partes, até que cada um deles contenha uma quantidade fixa de posições com dígitos hexadecimais independentes. Depois, basta usar uma classe de caracteres para cada posição, correspondendo aos intervalos por meio de alternância.

Como as letras e os algarismos ocupam áreas separadas nas tabelas de caracteres ASCII e Unicode, você não pode usar a classe de caracteres ‹[0-F]› para corresponder a qualquer um dos 16 dígitos hexadecimais. Embora tal classe de caracteres realmente faça isso, ela também corresponderá aos símbolos de pontuação situados entre os dígitos e as letras, na tabela ASCII. Ao invés disso, coloque dois intervalos de caracteres na classe de caracteres: [0-9A-F].

Outra questão que entra em jogo é a não-distinção entre maiúsculas e minúsculas. Por padrão, expressões regulares diferenciam maiúsculas de minúsculas. ‹[0-9A-F]› corresponde apenas a caracteres maiúsculos, e ‹[0-9a-f]› corresponde somente a caracteres minúsculos. ‹[0-9A-Fa-f]› corresponde a ambos.

Precisar digitar, explicitamente, ambos os intervalos de maiúsculas e minúsculas, em cada classe de caracteres, é algo que se torna entediante com muita rapidez. Ativar a opção de não-diferenciação entre maiúsculas e minúsculas é muito mais fácil. Veja a receita 3.4, para aprender a fazer isso em sua linguagem de programação favorita.

Veja também:

Receitas 2.8 e 6.2.

6.7 Números de ponto flutuante

Problema

Você deseja corresponder a um número de ponto flutuante e especificar se as partes de sinal, valor inteiro, valor fracionário e expoente do número são obrigatórios, opcionais ou proibidos. Você não quer utilizar a expressão regular para restringir os números a um intervalo específico, deixando isso para o código procedural, como explicado na receita 3.12.

Solução

Sinal, parte inteira, parte fracionária e expoente são obrigatórios:

```
^[-+][0-9]+\.[0-9]+[eE][-+]?[0-9]+$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Sinal, parte inteira e parte fracionária são obrigatórios, mas sem expoente:

```
^[-+][0-9]+\.[0-9]+$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Sinal opcional, parte inteira e parte fracionária são obrigatórios, mas sem expoente:

```
^[-+]?[0-9]+\.[0-9]+$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Sinal e parte inteira opcionais, parte fracionária obrigatória, mas sem expoente:

```
^[-+]?[0-9]*\.[0-9]+$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Sinal, parte inteira e parte fracionária opcionais. Se a parte inteira for omitida, a fração é obrigatória. Se a fração for omitida, o ponto decimal também deve ser omitido; sem expoente.

```
^[-+]?([0-9]+(\.[0-9]+)?|\.[0-9]+)$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Sinal, parte inteira e parte fracionária opcionais. Se a parte inteira for omitida, a fração é obrigatória. Se a fração for omitida, o ponto decimal também é opcional; sem expoente.

```
^[-+]?([0-9]+(\.[0-9]*)?|\.[0-9]+)$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Sinal, parte inteira e parte fracionária opcionais. Se a parte inteira for omitida, a fração é obrigatória. Se a fração for omitida, o ponto decimal também deve ser omitido; expoente opcional.

```
^[-+]?([0-9]+(\.[0-9]+)?|\.[0-9]+)([eE][-+]?[0-9]+)?$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Sinal, parte inteira e parte fracionária opcionais. Se a parte inteira for omitida, a fração é obrigatória. Se a fração for omitida, o ponto decimal é opcional; expoente opcional.

```
^[-+]?([0-9]+(\.[0-9]*)?|\.[0-9]+)([eE][-+]?[0-9]+)?$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

A expressão regular anterior, editada para encontrar o número em um corpo de texto maior:

```
[-+]?(\b[0-9]+(\.[0-9]*)?|\.[0-9]+)([eE][-+]?[0-9]+\b)?
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Todas as expressões regulares são colocadas entre âncoras (receita

2.5), para garantir a verificação de que toda entrada é um número de ponto flutuante, ao contrário de um número de ponto flutuante que ocorra em uma string maior. Você poderia utilizar extremidades de palavra ou lookarounds, como explicado na receita 6.1, se quisesse encontrar números de ponto flutuante em um conjunto de texto maior.

Soluções sem partes opcionais são muito simples: elas simplesmente soletram as partes da esquerda para a direita. Classes de caracteres (receita 2.3) correspondem ao sinal, aos dígitos e ao e. Os quantificadores + e ? (receita 2.12) permitem qualquer quantidade de dígitos e um sinal de expoente opcional.

Tornar opcionais apenas as partes do sinal e do valor inteiro é fácil. O ponto de interrogação, após a classe de caracteres com os símbolos de sinais, torna essa parte opcional. Usar um asterisco, em vez de um sinal de adição, para repetir os dígitos do número inteiro, permitirá zero ou mais dígitos, ao invés de um ou mais dígitos.

As complicações surgem quando o sinal, o valor inteiro e o valor fracionário forem todos opcionais. Embora sejam opcionais por natureza, eles não podem ser todos opcionais ao mesmo tempo, e a string vazia não é um número válido de ponto flutuante. A solução ingênua, ‹[-+]?[0-9]*\.?[0-9]*›, corresponde a todos os números válidos de ponto flutuante, mas também corresponde à string vazia. E como omitimos as âncoras, esta expressão regular corresponderá à string de comprimento zero entre dois caracteres quaisquer em seu texto de assunto. Se você executar uma pesquisa-e-substituição com esta expressão regular, e a substituição «{$&}» em 123abc456, você obterá {123}{}a{}b{}c{456}{}. A expressão regular corresponde a 123 e 456 corretamente, mas ela também encontra uma correspondência de comprimento zero em todas as outras tentativas de correspondência.

Ao criar uma expressão regular em uma situação na qual tudo é opcional, é muito importante considerar se todo o resto continua a

ser opcional, caso uma parte seja omitida. Números de ponto flutuante devem conter pelo menos um dígito.

As soluções para esta receita indicam claramente que, quando as partes inteira e fracionária forem opcionais, uma delas ainda é necessária. Elas também apontam se 123. é um número de ponto flutuante com um ponto decimal, ou se é um número inteiro seguido por um ponto que não faz parte do número. Por exemplo, em uma linguagem de programação, aquele ponto final poderia ser um operador de concatenação, ou o primeiro ponto em um operador de intervalo especificado por dois pontos.

Para implementar o requisito de que os inteiros e fracionários não podem ser omitidos ao mesmo tempo, usamos uma alternância (receita 2.8) dentro de um grupo (receita 2.9) para, simplesmente, enunciar as duas situações. ‹[0-9]+(\.[0-9]+)?› corresponde a um número com uma parte inteira obrigatória e uma parte fracionária opcional. ‹\.[0-9]+› corresponde apenas a um número fracionário.

Combinados, ‹[0-9]+(\.[0-9]+)?|\.[0-9]+› abrangem todas as três situações. A primeira alternativa abrange números com partes inteira e fracionária, bem como números sem uma fração. A segunda corresponde apenas à fração. Como o operador de alternância possui a menor prioridade dentre todos os operadores, precisamos colocar essas duas alternativas em um grupo, antes que possamos acrescentá-las a uma expressão regular mais longa.

‹[0-9]+(\.[0-9]+)?|\.[0-9]+› exige que o ponto decimal seja omitido quando a fração for omitida. Se o ponto decimal puder ocorrer mesmo sem dígitos fracionários, vamos poder utilizar ‹[0-9]+(\.[0-9]*)?|\.[0-9]+›. Na primeira alternativa desta expressão regular, a parte fracionária ainda está agrupada com o quantificador de ponto de interrogação, o que a torna opcional. A diferença é que os dígitos fracionários, em si, agora são opcionais. Mudamos o sinal de adição (um ou mais) para um asterisco (zero ou mais). O resultado é que a primeira alternativa, nesta expressão regular, corresponde a um número inteiro com a parte fracionária opcional, em que a fração pode ser

um ponto decimal seguido por dígitos ou apenas um ponto decimal. A segunda alternativa na expressão regular permanece inalterada.

Este último exemplo é interessante, porque temos uma mudança de requisito sobre um elemento, mas mudamos o quantificador em outra parte da expressão regular. A mudança de requisito é relativa ao ponto ser facultativo por si só, ao invés de ser opcional em conjunto com os dígitos fracionários. Nós conseguimos isso alterando o quantificador na classe de caracteres dos dígitos fracionários. Tal recurso funciona, pois o ponto decimal e a classe de caracteres já estavam dentro de um grupo que tornou ambos opcionais ao mesmo tempo.

Veja também:

Receitas 2.3, 2.8, 2.9 e 2.12.

6.8 Números com separadores de milhar

Problema

Você deseja corresponder a números que utilizem a vírgula como separador de milhar e o ponto como separador decimal.

Solução

Partes inteira e fracionária obrigatórias:

```
^[0-9]{1,3}(,[0-9]{3})*\.[0-9]+$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Parte inteira obrigatória e fração opcional. O ponto decimal deve ser omitido, caso a fração seja omitida.

```
^[0-9]{1,3}(,[0-9]{3})*(\.[0-9]+)?$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Partes inteira e fracionária opcionais. O ponto decimal deve ser omitido, caso a fração seja omitida.

```
^([0-9]{1,3}(,[0-9]{3})*(\.[0-9]+)?|\.[0-9]+)$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

A expressão regular anterior, editada para encontrar o número em um corpo de texto maior:

```
\b[0-9]{1,3}(,[0-9]{3})*(\.[0-9]+)?\b|\.[0-9]+\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Como todas são expressões regulares que correspondem a números de ponto flutuante, elas utilizam as mesmas técnicas da receita anterior. A única diferença é que, no lugar de, simplesmente, corresponder à parte inteira com ‹[0-9]+›, agora usamos ‹[0-9]{1,3}(,[0-9]{3})*›. Esta expressão regular corresponde entre 1 e 3 dígitos, seguidos de zero ou mais grupos, consistindo em uma vírgula e 3 dígitos.

Não podemos utilizar ‹[0-9]{0,3}(,[0-9]{3})*› para tornar a parte inteira opcional, porque isso corresponderia a números com uma vírgula à esquerda, por exemplo ,123. É a mesma armadilha de tornar tudo opcional, explicada na receita anterior. Para tornar a parte inteira opcional, não mudamos a parte da expressão regular relativa ao valor inteiro, mas, em vez disso, a tornamos facultativa em sua totalidade. As duas últimas expressões regulares, na solução, fazem isso usando alternância. A expressão regular, para um valor inteiro obrigatório e para uma parte fracionária opcional, é alternada com uma expressão regular que corresponde à fração sem o valor inteiro. Disso resulta uma expressão regular em que as partes inteira e fracionária são opcionais, mas não ao mesmo tempo.

Veja também:

Receitas 2.3, 2.9 e 2.12.

6.9 Numerais romanos

Problema

Você deseja corresponder a numerais romanos, como IV, XIII e MVIII.

Solução

Numerais romanos sem validação:

```
^[MDCLXVI]+$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Numerais romanos modernos estritos:

```
^(?=[MDCLXVI])M*(C[MD]|D?C{0,3})(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Numerais romanos modernos flexíveis:

```
^(?=[MDCLXVI])M*(C[MD]|D?C*)(X[CL]|L?X*)(I[XV]|V?I*)$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Numerais romanos simples:

```
^(?=[MDCLXVI])M*D?C{0,4}L?X{0,4}V?I{0,4}$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Algarismos romanos são escritos usando as letras M, D, C, L, X, V e I, representando os valores 1000, 500, 100, 50, 10, 5 e 1, respectivamente. A primeira expressão regular corresponde a qualquer string composta por essas letras, sem verificar se as letras aparecem na ordem ou quantidade necessárias para formar um

algarismo romano válido.

Nos tempos modernos (ou seja, durante os últimos cem anos), os algarismos romanos eram escritos, geralmente, seguindo um rigoroso conjunto de regras. Estas regras produzem exatamente um numeral romano por número. Por exemplo, 4 é sempre escrito como IV, nunca como IIII. A segunda expressão regular, na solução, corresponde apenas a algarismos romanos que sigam tais regras modernas.

Cada dígito que não seja zero no número decimal é escrito em separado no numeral romano. 1999 é escrito como MCMXCIX, em que M é 1000, CM é 900, XC é 90 e IX é 9. Não escrevemos MIM, ou IMM.

Os milhares são fáceis: um M para cada milhar, facilmente correspondidos por ‹M*›.

Existem 10 variações para as centenas, às quais correspondemos utilizando duas alternativas. ‹C[MD]› corresponde a CM e CD, que representam 900 e 400. ‹D?C{0,3}› corresponde a DCCC, DCC, DC, D, CCC, CC, C e à string vazia, o que representa 800, 700, 600, 500, 300, 200, 100 e nada. Isso nos dá todos os 10 dígitos para as centenas.

Correspondemos às dezenas com ‹X[CL]|L?X{0,3}› e às unidades com ‹I[XV]|V?I{0,3}›. Elas usam a mesma sintaxe, mas com letras diferentes.

Todas as quatro partes da expressão regular permitem que tudo seja opcional, pois cada um dos dígitos pode ser zero. Os romanos não tinham um símbolo, ou mesmo uma palavra para representar o zero. Assim, o zero não é escrito em numerais romanos. Enquanto cada parte da expressão regular deve realmente ser opcional, elas não são todas opcionais ao mesmo tempo. É preciso ter certeza de que nossas expressões regulares não permitirão correspondências de comprimento zero. Para tanto, colocamos o lookahead ‹(?=[MDCLXVI])› no início da expressão regular. Esse lookahead, como a receita 2.16 explica, garante que haja pelo menos uma letra na

correspondência da expressão regular. O lookahead não consome a letra correspondida por ele, de modo que a letra poderá ser correspondida novamente pelo resto da expressão regular.

A terceira expressão regular é um pouco mais flexível. Ela também aceita numerais, como IIII, enquanto ainda aceita IV.

A quarta expressão regular permite apenas numerais escritos sem o uso de subtrações e, portanto, todas as letras deverão estar em ordem decrescente; 4 deverá ser escrito como IIII, em vez de IV. Os próprios romanos, geralmente, escreviam números dessa forma.

Todas as expressões regulares são colocadas entre âncoras (receita 2.5), para termos certeza de que toda a entrada é um numeral romano, ao contrário de um número ocorrendo em uma string maior. Você pode substituir ‹^› e ‹$› por extremidades de palavra ‹\b›, caso queira encontrar algarismos romanos em um corpo de texto maior.

Converter numerais romanos para decimais

Esta função em Perl utiliza a expressão regular “estrita”, mostrada nesta receita, para verificar se a entrada é um numeral romano válido. Se for, ela utiliza a expressão regular ‹[MDLV]|C[MD]?|X[CL]?|I[XV]?›, para iterar todas as letras no numeral, somando seus valores:

```
sub roman2decimal {
  my $roman = shift;
  if ($roman =~
  m/^(?=[MDCLXVI])
    (M*) # 1000
    (C[MD]|D?C{0,3}) # 100
    (X[CL]|L?X{0,3}) # 10
    (I[XV]|V?I{0,3}) # 1
    $/ix)
  {
    # Numeral romano encontrado
    my %r2d = ('I' => 1, 'IV' => 4, 'V' => 5, 'IX' => 9,
                    'X' => 10, 'XL' => 40, 'L' => 50, 'XC' => 90,
                    'C' => 100, 'CD' => 400, 'D' => 500, 'CM' => 900,
                    'M' => 1000);
    my $decimal = 0;
    while ($roman =~ m/[MDLV]|C[MD]?|X[CL]?|I[XV]?/ig) {
       $decimal += $r2d{uc($&)};
```

```
    }
    return $decimal;
  } else {
    # Não é um numeral romano
    return 0;
  }
}
```

Veja também:

Receitas 2.3, 2.8, 2.9, 2.12, 2.16, 3.9 e 3.11.

CAPÍTULO 7

URLs, paths e endereços de Internet

Assim como os números, tema do capítulo anterior, outro tópico importante, que diz respeito a uma ampla gama de programas, são os vários caminhos e localizadores utilizados para encontrar dados:

- URLs, URNs e strings relacionadas.
- Nomes de domínios.
- Endereços IP.
- Nomes de pastas e arquivos do Microsoft Windows.

O formato URL, em particular, tem se mostrado tão flexível e útil que acabou sendo adotado por um vasto leque de recursos sem qualquer relação com a World Wide Web. A caixa de ferramentas de análise de expressões regulares deste capítulo será útil em uma variedade surpreendente de situações.

7.1 Validar URLs

Problema

Você deseja verificar se um determinado pedaço de texto é uma URL válida para seus propósitos.

Solução

Permitir quase todas as URLs:

```
^(https?|ftp|file)://.+$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A(https?|ftp|file)://.+\Z
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Requer um nome de domínio, e não permite um nome de usuário ou senha:

```
\A # Âncora
(https?|ftp):// # Protocolo
[a-z0-9-]+(\.[a-z0-9-]+)+ # Domínio
([/?].*)? # Path e/ou parâmetros
\Z # Âncora
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^(https?|ftp)://[a-z0-9-]+(\.[a-z0-9-]+)+↲
([/?].+)?$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Requer um nome de domínio, e não permite nome de usuário ou senha. Permite que o protocolo (*http* ou *ftp*) seja omitido, caso ele possa ser deduzido do subdomínio (*www* ou *ftp*):

```
\A # Âncora
((https?|ftp)://|(www|ftp)\.) # Protocolo ou subdomínio
[a-z0-9-]+(\.[a-z0-9-]+)+ # Domínio
([/?].*)? # Path e/ou parâmetros
\Z # Âncora
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^((https?|ftp)://|(www|ftp)\.)[a-z0-9-]+(\.[a-z0-9-]+)+([/?].*)?$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Requer um nome de domínio e um caminho que apontem para um arquivo de imagem. Não permite nome de usuário, senha ou parâmetros:

```
\A # Âncora
```

```
(https?|ftp):// # Protocolo
[a-z0-9-]+(\.[a-z0-9-]+)+ # Domínio
(/[\w-]+)* # Path
/[\w-]+\.(gif|png|jpg) # Arquivo
\Z # Âncora
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^(https?|ftp)://[a-z0-9-]+(\.[a-z0-9-]+)+(/[\w-]+)*/[\w-]+\.(gif|png|jpg)$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Discussão

Você não pode criar uma expressão regular que corresponda a todas as URLs válidas sem acabar correspondendo a URLs inválidas. A razão é que praticamente qualquer coisa pode ser uma URL válida, em protocolos que ainda estão por ser inventados.

Validar URLs tem utilidade apenas quando sabemos que seu contexto é válido. Podemos limitar as URLs aceitas pelos protocolos suportados pelo programa que estamos usando. Todas as expressões regulares desta receita são para URLs usadas por navegadores web. Tais URLs usam o formato:

```
scheme://user:password@domain.name:80/path/file.ext?
param=value&param2=value2#fragment
```

Todas essas partes são, na verdade, opcionais. Uma URL `file:` possui apenas um caminho. URLs `http:` só precisam de um nome de domínio.

A primeira expressão regular, na solução, verifica se a URL começa com um dos protocolos mais comuns usados por navegadores web: `http`, `https`, `ftp` e `file`. O acento circunflexo ancora a expressão regular no início da string (receita 2.5). A alternância (receita 2.8) é utilizada para enunciar a lista dos protocolos. `‹https?›` é uma maneira inteligente de dizer `‹http|https›`.

Como a primeira expressão regular permite protocolos muito

diferentes, como http e file, ela não tenta validar o texto após o protocolo. ‹.+$› simplesmente pega tudo até o final da string, desde que ela não contenha caracteres de quebra de linha.

Por padrão, o ponto (receita 2.4) corresponde a todos os caracteres, exceto os de quebra de linha, e o cifrão (receita 2.5) não corresponde em quebras de linha embutidas. O Ruby é a exceção aqui. Em Ruby, o acento circunflexo e o cifrão sempre correspondem em quebras de linha embutidas e, por isso, temos de usar ‹\A› e ‹\Z› (receita 2.5). Estritamente falando, você teria de fazer a mesma alteração, no Ruby, para todas as outras expressões regulares mostradas nesta receita. Você deveria... se sua entrada pudesse consistir em várias linhas, e você pretende evitar corresponder a uma URL que ocupe uma linha dentro de várias linhas de texto.

As próximas duas expressões regulares são as versões normal e com espaçamento livre (receita 2.18) da mesma expressão regular. A expressão regular com espaçamento livre é mais fácil de ler, enquanto a versão normal é mais rápida para digitar. O JavaScript não suporta expressões regulares com espaçamento livre.

Essas duas expressões regulares aceitam somente URLs web e FTP, e requerem que o protocolo HTTP ou FTP seja seguido por algo que se pareça com um nome de domínio válido. O nome de domínio deve usar a tabela de caracteres ASCII. Domínios internacionalizados (IDNs) não são aceitos. O domínio pode ser seguido por um caminho ou uma lista de parâmetros, separados do domínio por uma barra ou um ponto de interrogação. Como o ponto de interrogação está dentro de uma classe de caracteres (receita 2.3), não precisamos escapá-lo. O ponto de interrogação é um caractere comum da classe de caracteres, e a barra é um caractere comum em qualquer lugar na expressão regular. Se você encontrá-la escapada, no código-fonte, é porque o Perl e outras linguagens de programação utilizam barras para delimitar expressões regulares literais.

Nenhuma tentativa é feita para validar o caminho ou os parâmetros. ‹.*› simplesmente corresponde a qualquer coisa que não inclua quebras de linhas. Como o caminho e os parâmetros são ambos opcionais, ‹[/?].*› é colocado dentro de um grupo, feito opcional por meio de um ponto de interrogação (receita 2.12).

Essas expressões regulares, e as que seguem, não permitem especificar nome de usuário ou senha como parte da URL. Colocar informações do usuário na URL é considerada uma prática ruim, por razões de segurança.

A maioria dos navegadores web aceitam URLs que não especifiquem o protocolo, e o deduzem corretamente por meio do nome de domínio. Por exemplo, www.regexbuddy.com é uma abreviação de http://www.regexbuddy.com. Para permitir tais URLs, simplesmente expandimos a lista de protocolos permitidos pela expressão regular para incluir os subdomínios www.e ftp.

‹(https?|ftp)://|(www|ftp)\.› faz isso muito bem. Esta lista possui duas alternativas, e cada uma delas começa com duas alternativas. A primeira permite ‹https?› e ‹ftp›, que devem ser seguidos por ‹://›. A segunda permite ‹www› e ‹ftp›, que devem ser seguidos por um ponto. Você pode editar facilmente ambas as listas, para alterar os protocolos e subdomínios que a expressão regular deve aceitar.

As duas últimas expressões regulares exigem um protocolo, um nome de domínio em ASCII, um caminho e um nome de arquivo para um arquivo de imagem GIF, PNG ou JPEG. O caminho e o nome do arquivo permitem todas as letras e dígitos de qualquer sistema de escrita, bem como underscores e hífens. A classe de caracteres abreviada ‹\w› inclui tudo isso, exceto os hífens (receita 2.3).

Qual destas expressões regulares você deve usar? Isso realmente depende do que estiver tentando fazer. Em muitas situações, a resposta pode ser não usar nenhuma expressão regular. Simplesmente tente resolver a URL. Se ela retornar um conteúdo válido, aceite-a. Se obtiver um erro 404, ou qualquer outro, rejeite-a.

Em última análise, este é o único teste real para ver se a URL é válida.

Veja também:

Receitas 2.3, 2.8, 2.9 e 2.12.

7.2 Encontrar URLs dentro de um texto completo

Problema

Você deseja encontrar URLs em um corpo de texto maior. URLs podem, ou não, ser encapsuladas dentro de pontuações (como parênteses) que não façam parte da URL.

Solução

URL sem espaços:

```
\b(https?|ftp|file)://\S+
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

URL sem espaços ou pontuação final:

```
\b(https?|ftp|file)://[-A-Z0-9+&@#/%?=~_|$!:,.;]*↲
[A-Z0-9+&@#/%=~_|$]
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

URL sem espaços ou pontuação final. URLs que iniciem com o subdomínio www ou ftp podem omitir o protocolo:

```
\b((https?|ftp|file)://|(www|ftp)\.)[-A-Z0-9+&@#/%?=~_|$!:,.;]*↲
[A-Z0-9+&@#/%=~_|$]
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Dado o texto:

> Visit http://www.somesite.com/page, where you will find more information.

O que é a URL?

Antes que você diga http://www.somesite.com/page, pense nisso: pontuação e espaços são caracteres válidos em URLs. Vírgulas, pontos e até mesmo espaços não precisam ser escapados, como %20. Espaços literais são perfeitamente válidos. Algumas ferramentas WYSIWYG de autoria web até facilitam a colocação de espaços em nomes de arquivos e pastas, e incluem estes espaços literalmente em links para aqueles arquivos.

Isso significa que, se usarmos uma expressão regular que permita todas as URLs válidas, ela encontrará esta URL no texto anterior:

> http://www.somesite.com/page, where you will find more information.

São pequenas as chances de que a pessoa que digitou essa frase tivesse a intenção de incluir os espaços na URL, uma vez que os espaços não-escapados em URLs são raros. A primeira expressão regular, na solução, exclui os espaços usando a classe de caracteres abreviada ‹\S›, que inclui todos os caracteres que não sejam espaços em branco. Embora a expressão regular especifique a opção de “não-diferenciação entre maiúsculas e minúsculas”, o S deve ser maiúsculo, porque ‹\S› não é o mesmo que ‹\s›. Na verdade, eles são exatamente o oposto. A receita 2.3 mostra todos os detalhes dessa questão.

A primeira expressão regular ainda está bem crua. Ela incluirá a vírgula do texto de exemplo na URL. Embora não seja incomum que uma URL inclua vírgulas e outros sinais a pontuação raramente ocorre em seu final.

A próxima expressão regular usa duas classes de caracteres, ao invés da abreviação simples ‹\S›. A primeira classe de caracteres inclui mais pontuação que a segunda, que exclui os caracteres que possam aparecer como pontuação do idioma inglês após uma URL, quando esta é colocada em uma frase em inglês. A primeira classe de caracteres possui o quantificador asterisco (receita 2.12), para

permitir URLs de qualquer tamanho. A segunda não possui um quantificador, exigindo que a URL termine com um caractere dessa classe. As classes de caracteres não incluem letras minúsculas; a opção de “não-diferenciação entre maiúsculas e minúsculas” cuida disso. Veja a receita 3.4, para aprender a configurar tais opções em sua linguagem de programação.

A segunda expressão regular funcionará incorretamente em determinadas URLs que usem pontuações estranhas, correspondendo a elas apenas parcialmente. Porém, ela resolve o problema muito comum de uma vírgula ou ponto final logo após uma URL, enquanto ainda permite vírgulas e pontos.

A maioria dos navegadores aceitam URLs que não especificam o protocolo, deduzindo-o, corretamente, a partir do nome de domínio. Por exemplo, www.regexbuddy.com é a abreviação de http://www.regexbuddy.com. Para permitir essas URLs, a expressão regular final amplia a lista de protocolos permitidos, para incluir os subdomínios www. e ftp..

‹(https?|ftp)://|(www|ftp)\.› faz isso muito bem. Essa lista possui duas alternativas, cada uma das quais começa com duas alternativas. A primeira permite ‹https?› e ‹ftp›, que devem ser seguidos por ‹://›. A segunda permite ‹www› e ‹ftp›, que devem ser seguidos por um ponto. Você pode editar facilmente ambas as listas, para mudar os protocolos e subdomínios que a expressão regular deve aceitar.

Veja também:

Receitas 2.3 e 2.6.

7.3 Encontrar URLs entre aspas no texto completo

Problema

Você deseja encontrar URLs em um corpo de texto maior. URLs

podem, ou não, ser englobadas pela pontuação que integra o corpo de texto maior, ao invés de tal pontuação fazer parte da URL. Você quer dar aos usuários a opção de colocar URLs entre aspas, para que eles possam indicar explicitamente se a pontuação, ou mesmo os espaços, devem fazer parte da URL.

Solução

```
\b(?:(?:https?|ftp|file)://|(www|ftp)\.)[-A-Z0-9+&@#/%?=~_|$!:,.;]*
                                [-A-Z0-9+&@#/%=~_|$]
|"(?:(?:https?|ftp|file)://|(www|ftp)\.)[^"\r\n]+"
|'(?:(?:https?|ftp|file)://|(www|ftp)\.)[^'\r\n]+'
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas, ponto corresponde a quebras de linhas, âncoras correspondem em quebras de linhas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

A receita anterior explica o problema de misturar URLs com texto em inglês e como diferenciar entre a pontuação em inglês, e os caracteres da URL. Embora a solução da receita anterior seja muito útil e acerte na maioria das vezes, nenhuma expressão regular dá certo sempre.

Se a sua expressão regular será usada em um texto a ser escrito no futuro, você pode dispor a seus usuários uma maneira de colocarem suas URLs entre aspas. A solução que apresentamos permite a colocação de um par de aspas simples ou um par de aspas duplas ao redor da URL. Quando uma URL é colocada entre aspas, ela deve começar com um dos vários protocolos: ‹https?|ftp|file›, ou um dos dois subdomínios ‹www|ftp›. Depois do protocolo ou subdomínio, a expressão regular permite que a URL inclua qualquer caractere, exceto quebras de linhas e a aspa delimitadora.

A expressão regular, como um todo, é dividida em três alternativas. A primeira é a expressão regular da receita anterior, que corresponde a uma URL que não esteja entre aspas, tentando diferenciar entre a pontuação em inglês e os caracteres da URL. A

segunda corresponde a uma URL entre aspas duplas. A terceira alternativa corresponde a uma URL entre aspas simples. Usamos duas alternativas, em vez de uma única, com um grupo de captura em torno da aspa de abertura e uma retrorreferência para a aspa de fechamento, porque não podemos usar uma retrorreferência, no interior da classe de caracteres negada, que exclua o caractere de aspa da URL.

Optamos por usar aspas simples e duplas, pois é assim que as URLs geralmente aparecem em arquivos HTML e XHTML. Colocar aspas em URLs dessa maneira é natural para pessoas que trabalham na Web, mas você pode editar facilmente a expressão regular, permitindo diferentes pares de caracteres para delimitar as URLs.

Veja também:

Receitas 2.8 e 2.9.

7.4 Encontrar URLs entre parênteses no texto completo

Problema

Você quer encontrar URLs em um corpo de texto maior. URLs podem ser, ou não, englobadas pela pontuação que faz parte do corpo de texto maior, em vez de tal pontuação fazer parte da URL. Você deseja corresponder corretamente a URLs que incluam pares de parênteses como parte da URL, sem corresponder aos parênteses colocados a seu redor.

Solução

```
\b(?:(?:https?|ftp|file)://|www\.|ftp\.)
  (?:\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)|[-A-Z0-9+&@#/%=~_|$?!:,.])*
  (?:\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)|[A-Z0-9+&@#/%=~_|$])
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
\b(?:(?:https?|ftp|file)://|www\.|ftp\.)(?:\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)↲
|[-A-Z0-9+&@#/%=~_|$?!:,.])*(?:\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)|↲
[A-Z0-9+&@#/%=~_|$])
```

Opções Regex: não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Praticamente, qualquer caractere é válido em URLs, incluindo os parênteses. No entanto, parênteses são muito raros em URLs, por isso não os incluímos em nenhuma das expressões regulares nas receitas anteriores. Alguns sites importantes, porém, estão começando a usá-los:

http://en.wikipedia.org/wiki/PC_Tools_(Central_Point_Software)
http://msdn.microsoft.com/en-us/library/aa752574(VS.85).aspx

Uma solução seria exigir que seus usuários coloquem aspas nessas URLs. A outra seria melhorar sua expressão regular, para que ela as aceite. A parte difícil é determinar se um parêntese de fechamento faz parte da URL ou se é usado como pontuação em seu entorno, como neste exemplo:

RegexBuddy's web site (em http://www.regexbuddy.com) is really cool.

Como é possível que um dos parênteses esteja adjacente à URL, enquanto o outro não, não podemos usar a técnica de colocar expressões regulares entre aspas, mostrada na receita anterior. A solução mais simples seria permitir parênteses em URLs apenas quando eles ocorrerem em pares não aninhados de abertura e fechamento. As URLs da Wikipédia e da Microsoft satisfazem essa exigência.

As duas expressões regulares na solução são idênticas. A primeira usa o modo de espaçamento livre, para torná-la um pouco mais fácil de ler.

Estas expressões regulares são essencialmente iguais à última expressão regular na solução da receita 7.2. Existem três partes em

todas essas expressões regulares: a lista de protocolos, seguida do corpo da URL, que usa o quantificador asterisco para permitir URLs de qualquer tamanho, e o final da URL, que não possui quantificador (ou seja, deverá ocorrer uma vez). Na expressão regular original, na receita 7.2, tanto o corpo da URL, quanto seu final, consistiam de apenas uma classe de caracteres.

As soluções desta receita substituem as duas classes de caracteres por elementos mais elaborados. A classe de caracteres do meio:

```
[-A-Z0-9+&@#/%=~_|$?!:,.]
```

tornou-se:

```
\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)|[-A-Z0-9+&@#/%=~_|$?!:,.]
```

A classe de caracteres final:

```
[A-Z0-9+&@#/%=~_|$]
```

tornou-se:

```
\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)|[A-Z0-9+&@#/%=~_|$]
```

Ambas as classes de caracteres foram substituídas por algo envolvendo alternância (receita 2.8). Como a alternância tem a prioridade mais baixa entre todos os operadores de expressão regular, usamos grupos de não-captura (receita 2.9) para manter as duas alternativas reunidas.

Em ambas as classes de caracteres, adicionamos a alternativa ‹\([-A-Z0-9+&@#/%=~_|$?!:,.]*\)›, deixando a classe de caracteres original como a outra opção. A nova alternativa corresponde a um par de parênteses em torno de qualquer quantidade, de quaisquer caracteres permitidos em uma URL.

Foi dada a mesma alternativa para a classe de caracteres final, o que permite que a URL termine com o texto entre parênteses ou com um caractere simples que, provavelmente, não seja pontuação do idioma inglês.

Combinadas, elas resultam em uma expressão regular que corresponde a URLs com qualquer número de parênteses, incluindo URLs sem parênteses e até URLs que consistam em nada mais do

que parênteses, contanto que eles ocorram aos pares.

No caso do corpo da URL, nós colocamos o quantificador asterisco em torno de todo o grupo de não-captura. Isso permite que qualquer número de pares de parênteses ocorra na URL. Como temos o asterisco em torno do grupo de não-captura, já não precisamos de um asterisco diretamente na classe de caracteres original. Na verdade, temos de nos certificar de não incluir o asterisco.

A expressão regular, na solução, possui a forma ‹(ab*c|d)*› no meio, em que ‹a› e ‹c› são os parênteses literais e ‹b› e ‹d› são as classes de caracteres. Escrever ‹(ab*c|d*)*› seria um erro. Pode parecer lógico à primeira vista, porque permitimos qualquer número de caracteres de ‹d›, mas o ‹*› externo repete ‹d›. Se adicionarmos um asterisco interno diretamente em ‹d›, a complexidade da expressão regular se torna exponencial. ‹(d*)*› pode corresponder a dddd de várias formas. Por exemplo, o asterisco externo poderia repetir quatro vezes, repetindo o asterisco interno uma vez a cada repetição sua. O asterisco externo poderia repetir três vezes, com o asterisco interno fazendo 2-1-1, 1-2-1 ou 1-1-2. O asterisco externo poderia repetir duas vezes, com o asterisco interno fazendo 2-2, 1-3, ou 3-1. Você pode imaginar que, conforme cresce o comprimento da string, o número de combinações rapidamente explode. Chamamos isso de retrocesso catastrófico, termo introduzido na receita 2.15. Esse problema surgirá quando a expressão regular não conseguir encontrar uma correspondência válida, por exemplo, porque você anexou algo à expressão regular para localizar URLs que terminem com, ou contenham algo específico às suas necessidades.

Veja também:

Receitas 2.8 e 2.9.

7.5 Transformar URLs em links

Problema

Você possui um corpo de texto que pode ter URLs, e deseja converter as URLs contidas no texto em links, colocando tags HTML de âncora em torno das URLs. A URL, em si, será tanto o destino do link quanto o texto do link.

Solução

Para localizar as URLs no texto, utilize uma das expressões regulares das receitas 7.2 ou 7.4. Como texto de substituição, use:

```
<a href="$&">$&</a>
```

Sabores de texto de substituição: .NET, JavaScript, Perl

```
<a href="$0">$0</a>
```

Sabores de texto de substituição: .NET, Java, PHP

```
<a href="\0">\0</a>
```

Sabores de texto de substituição: PHP, Ruby

```
<a href="\&">\&</a>
```

Sabor de texto de substituição: Ruby

```
<a href="\g<0>">\g<0></a>
```

Sabor de texto de substituição: Python

Ao programar, você pode implementar esta pesquisa-e-substituição, como explicado na receita 3.15.

Discussão

A solução para este problema é muito simples. Nós devemos usar uma expressão regular que corresponda a uma URL para, em seguida, substituí-la por «URL», em que *URL* representa a URL correspondida. Diferentes linguagens de programação usam sintaxes diferentes para o texto de substituição, daí a longa lista de soluções para este problema. Porém, todas elas fazem exatamente a mesma coisa. A receita 2.20 explica a sintaxe do texto de substituição.

Veja também:

Receitas 2.21, 3.15, 7.2 e 7.4.

7.6 Validar URNs

Problema

Você deseja verificar se uma string representa um Nome de Recurso Uniforme (Uniform Resource Name, ou URN) válido, conforme especificado na RFC 2141, ou encontrar URNs em um corpo de texto maior.

Solução

Verificar se uma string consiste inteiramente em um URN válido:

```
\Aurn:
# Identificador do espaço de nomes
[a-z0-9][a-z0-9-]{0,31}:
# String específica do espaço de nomes
[a-z0-9()+,\-.:=@;$_!*'%/?#]+
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^urn:[a-z0-9][a-z0-9-]{0,31}:[a-z0-9()+,\-.:=@;$_!*'%/?#]+$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Encontrar um URN em um corpo de texto maior:

```
\burn:
# Identificador do espaço de nomes
[a-z0-9][a-z0-9-]{0,31}:
# String específica do espaço de nomes
[a-z0-9()+,\-.:=@;$_!*'%/?#]+
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
\burn:[a-z0-9][a-z0-9-]{0,31}:[a-z0-9()+,\-.:=@;$_!*'%/?#]+
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Encontrar um URN em um corpo de texto maior, assumindo que a pontuação, em seu final, faça parte do texto (em inglês) no qual ele é citado, em vez de fazer parte do próprio URN:

```
\burn:
# Identificador do espaço de nomes
[a-z0-9][a-z0-9-]{0,31}:
# String específica do espaço de nomes
[a-z0-9()+,\-.:=@;$_!*'%/?#]*[a-z0-9+=@$/]
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
\burn:[a-z0-9][a-z0-9-]{0,31}:[a-z0-9()+,\-.:=@;$_!*'%/?#]*[a-z0-9+=@$/]
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Um URN consiste em três partes. A primeira parte é composta pelos quatro caracteres `urn:`, que podemos acrescentar literalmente à expressão regular.

A segunda parte é o Identificador de Namespace (NID). Ele possui entre 1 e 32 caracteres de comprimento. O primeiro caractere deve ser uma letra ou um dígito. Os caracteres restantes podem ser letras, dígitos e hífens. Correspondemos a isso usando duas classes de caracteres (receita 2.3): a primeira corresponde a uma letra ou a um dígito, e a segunda corresponde entre 0 e 31 letras, dígitos e hífens. O NID deve ser delimitado por um sinal de dois pontos (:), que, novamente, adicionamos literalmente à expressão regular.

A terceira parte do URN é a String Específica do Namespace (Namespace Specific String, ou NSS). Ela pode ser de qualquer tamanho, e pode incluir alguns caracteres de pontuação, além de letras e algarismos. Correspondemos a isso facilmente com outra classe de caracteres. O sinal de adição após a classe de caracteres repete-a uma ou mais vezes (receita 2.12).

Se quiser verificar se uma string representa um URN válido, tudo o que resta é adicionar âncoras, no início e no final da expressão regular, que correspondam ao início e ao final da string. Podemos fazer isso com ‹^› e ‹$› em todos os sabores, exceto o Ruby, e com ‹\A› e ‹\Z› em todos os sabores, exceto o JavaScript. A receita 2.5 tem todos os detalhes sobre estas âncoras.

As coisas ficam um pouco mais complicadas se você quiser encontrar URNs em um corpo de texto maior. A questão da pontuação em URLs, discutida na receita 7.2, também existe no caso dos URNs. Suponha que você tenha o texto:

```
The URN is urn:nid:nss, isn't it?
```

A questão é se a vírgula faz parte do URN. URNs terminados em vírgulas são sintaticamente válidos, mas qualquer pessoa que leia esta frase em inglês veria a vírgula como uma pontuação da língua inglesa, não como parte do URN. A última expressão regular na seção “Solução” resolve esse problema, por ser um pouco mais rigorosa do que a RFC 2141. Ela restringe o último caractere do URN a um caractere que seja válido para a parte da NSS e que, provavelmente, não aparecerá como pontuação em inglês dentro de uma frase que mencione um URN.

Fazemos facilmente esta tarefa substituindo o quantificador de adição (um ou mais) por um asterisco (zero ou mais), e adicionando uma segunda classe de caracteres ao caractere final. Se adicionássemos a classe de caracteres sem alterar o quantificador, exigiríamos que a NSS tivesse, pelo menos, dois caracteres de comprimento, e não é isso o que queremos.

Veja também:

Receitas 2.3 e 2.12.

7.7 Validar URLs genéricas

Problema

Você deseja verificar se um certo fragmento de texto é uma URL válida, de acordo com a RFC 3986.

Solução

```
\A
(# Scheme
 [a-z][a-z0-9+\-.]*:
 (# Autoridade e path
  //
  ([a-z0-9\-._~%!$&'()*+,;=]+@)? # Usuário
  ([a-z0-9\-._~%]+ # Host nomeado
  |\[[a-f0-9:.]+\] # Host IPv6
  |\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # Host IPvFuture
  (:[0-9]+)? # Porta
  (/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/? # Path
 |# Path sem autoridade
  (/?[a-z0-9\-._~%!$&'()*+,;=:@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)?
 )
|# URL relativa (sem protocolo ou autoridade)
 (# Path relativo
  [a-z0-9\-._~%!$&'()*+,;=@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?
 |# Path absoluto
  (/[a-z0-9\-._~%!$&'()*+,;=:@]+)+/?
 )
)
# Consulta
(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
# Fragmento
(\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
\A
(# Scheme
 (?<scheme>[a-z][a-z0-9+\-.]*):
 (# Autoridade e path
  //
  (?<user>[a-z0-9\-._~%!$&'()*+,;=]+@)? # Usuário
  (?<host>[a-z0-9\-._~%]+ # Host nomeado
  | \[[a-f0-9:.]+\] # Host IPv6
```

```
  | \[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # Host IPvFuture
  (?<port>:[0-9]+)? # Porta
  (?<path>(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?) # Path
 |# Path sem autoridade
  (?<path>/?[a-z0-9\-._~%!$&'()*+,;=:@]+
         (/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)?
 )
|# URL relativa (sem protocolo, ou autoridade)
 (?<path>
  # Path relativo
  [a-z0-9\-._~%!$&'()*+,;=@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?
 |# Path absoluto
  (/[a-z0-9\-._~%!$&'()*+,;=:@]+)+/?
 )
)
# Consulta
(?<query>\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
# Fragmento
(?<fragment>\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET

```
\A
(# Scheme
 (?<scheme>[a-z][a-z0-9+\-.]*):
 (# Autoridade e path
  //
  (?<user>[a-z0-9\-._~%!$&'()*+,;=]+@)? # Usuário
  (?<host>[a-z0-9\-._~%]+ # Host nomeado
  | \[[a-f0-9:.]+\] # Host IPv6
  | \[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # Host IPvFuture
  (?<port>:[0-9]+)? # Porta
  (?<hostpath>(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?) # Path
 |# Path sem autoridade
  (?<schemepath>/?[a-z0-9\-._~%!$&'()*+,;=:@]+
            (/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)?
  )
|# URL relativa (sem protocolo, ou autoridade)
 (?<relpath>
  # Path relativo
  [a-z0-9\-._~%!$&'()*+,;=@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?
```

```
 |# Path absoluto
  (/[a-z0-9\-._~%!$&'()*+,;=:@]+)+/?
 )
)
# Consulta
(?<query>\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
# Fragmento
(?<fragment>\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\A
(# Scheme
 (?P<scheme>[a-z][a-z0-9+\-.]*):
 (# Autoridade e path
  //
  (?P<user>[a-z0-9\-._~%!$&'()*+,;=]+@)? # Usuário
  (?P<host>[a-z0-9\-._~%]+ # Host nomeado
  | \[[a-f0-9:.]+\] # Host IPv6
  | \[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # Host IPvFuture
  (?P<port>:[0-9]+)? # Porta
  (?P<hostpath>(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?) # Path
 |# Path sem autoridade
  (?P<schemepath>/?[a-z0-9\-._~%!$&'()*+,;=:@]+
                (/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)?
 )
|# URL relativa (sem protocolo, ou autoridade)
 (?P<relpath>
  # Path relativo
  [a-z0-9\-._~%!$&'()*+,;=@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?
 |# Path absoluto
  (/[a-z0-9\-._~%!$&'()*+,;=:@]+)+/?
 )
)
# Consulta
(?P<query>\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
# Fragmento
(?P<fragment>\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: PCRE 4 e superior, Perl 5.10, Python

```
^([a-z][a-z0-9+\-.]*:(\/\/([a-z0-9\-._~%!$&'()*+,;=]+@)?([a-z0-9\-._~%]+|↲
\[[a-f0-9:.]+\]|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\])(:[0-9]+)?↲
(\/[a-z0-9\-._~%!$&'()*+,;=:@]+)*\/?|(\/?[a-z0-9\-._~%!$&'()*+,;=:@]+↲
(\/[a-z0-9\-._~%!$&'()*+,;=:@]+)*\/?)?)|([a-z0-9\-._~%!$&'()*+,;=@]+↲
(\/[a-z0-9\-._~%!$&'()*+,;=:@]+)*\/?|(\/[a-z0-9\-._~%!$&'()*+,;=:@]+)↲
+\/?))
(\?[a-z0-9\-._~%!$&'()*+,;=:@\/?]*)?(#[a-z0-9\-._~%!$&'()*+,;=:@\/?]*)?$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Discussão

A maioria das receitas anteriores, neste capítulo, lidaram com URLs, e as expressões regulares dessas receitas lidaram com tipos específicos de URLs. Algumas das expressões regulares foram adaptadas para fins específicos, como determinar se a pontuação faz parte da URL ou do texto que cita a URL.

A expressão regular desta receita lida com URLs genéricas. Elas não são destinadas à busca de URLs em textos maiores; são feitas para validar strings que supostamente contenham URLs, e para dividir URLs em várias partes. Elas realizam essas tarefas em qualquer tipo de URL, mas, na prática, é provável que você queira construir expressões regulares mais específicas. As receitas posteriores mostram exemplos de expressões regulares nesse sentido.

A RFC 3986 descreve a aparência de uma URL válida. Ela abrange todas as URLs possíveis, incluindo as relativas e as feitas para protocolos ainda não inventados. Como resultado, a RFC 3986 é muito ampla, e uma expressão regular que a implemente é bastante longa. As expressões regulares, nesta receita, implementam somente o básico. Elas são o suficiente para dividir a URL em suas várias partes de maneira confiável, mas não para validar cada uma destas partes. De qualquer forma, validar todas as partes exigiria conhecimentos específicos de cada protocolo de URL.

A RFC 3986 não cobre todas as URLs que você poderá encontrar. Por exemplo, muitos navegadores e servidores web aceitam URLs com espaços literais, mas a RFC 3986 exige que espaços sejam escapados como %20.

Uma URL absoluta deve começar com um protocolo, como http: ou ftp:. O primeiro caractere do protocolo deve ser uma letra. Os caracteres seguintes podem ser letras, números e alguns caracteres de pontuação específicos. Podemos corresponder a isso facilmente com duas classes de caracteres: ‹[a-z][a-z0-9+\-.]*›.

Muitos protocolos de URL exigem que a RFC 3986 invoque uma “autoridade”. A autoridade é o nome de domínio ou endereço IP do servidor, opcionalmente precedido por um nome de usuário, e opcionalmente seguido por um número de porta.

O nome de usuário pode ser constituído por letras, dígitos e algumas pontuações. Ele deve ser separado do nome de domínio ou endereço IP por um sinal @. ‹[a-z0-9\-._~%!$&'()*+,;=]+@› corresponde ao nome de usuário e ao delimitador.

A RFC 3986 é bastante liberal no que diz respeito ao nome de domínio. A receita 7.15 explica o que é geralmente permitido para domínios: letras, dígitos, hífens e pontos. A RFC 3986 também permite til e qualquer outro caractere por meio da notação percentual. O nome de domínio deve ser convertido para UTF-8; qualquer byte que não seja letra, dígito, hífen ou til deve ser codificado como %*FF*, em que *FF* é a representação hexadecimal do byte.

Para mantermos nossa expressão regular simples, não vamos verificar se cada sinal de porcentagem é seguido por exatamente dois dígitos hexadecimais. É melhor fazer essa validação após a separação das várias partes da URL. Então, correspondemos ao nome do host apenas com ‹[a-z0-9\-._~%]+›, que também corresponde a endereços IPv4 (permitidos pela RFC 3986).

Em vez de um nome de domínio ou endereço IPv4, o host também pode ser especificado como um endereço IPv6 entre colchetes, ou

mesmo como uma versão futura de endereços IP. Nós correspondemos aos endereços IPv6 com ‹\[[a-f0-9:.]+\]›, e aos endereços futuros com ‹\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]›. Embora não possamos validar endereços IP usando uma versão de IP ainda não definida, poderíamos ser mais rigorosos com os endereços IPv6. Mas, novamente, é melhor deixarmos isso para uma segunda expressão regular, após a extração do endereço da URL. A receita 7.17 mostra que validar endereços IPv6 está longe de ser trivial.

O número da porta, se especificado, é simplesmente um número decimal separado do nome do host por um sinal de dois pontos. ‹:[0-9]+› é tudo de que precisamos.

Se uma autoridade é especificada, ela deve ser seguida por um caminho absoluto ou não apresentar um caminho. Um caminho absoluto começa com uma barra, seguida por um ou mais segmentos delimitados por barras. Um segmento é composto por uma ou mais letras, dígitos ou caracteres de pontuação. Não pode haver barras consecutivas. O caminho pode terminar com uma barra. ‹(/[a-z0-9._~%!$&'()*+,;=:@]+)*/?› corresponde a tais caminhos.

Se a URL não especificar uma autoridade, o caminho pode ser absoluto, relativo ou omitido. Caminhos absolutos começam com uma barra, enquanto caminhos relativos, não. Como a barra inicial agora é opcional, precisamos de uma expressão regular um pouco mais longa para corresponder a ambos os caminhos, absoluto e relativo: ‹/?[a-z0-9\-._~%!$&'()*+,;=:@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?›.

URLs relativas não especificam um protocolo e, portanto, não especificam uma autoridade. O caminho torna-se obrigatório, e pode ser absoluto ou relativo. Se a URL não especificar um protocolo, o primeiro segmento de um caminho relativo não pode conter um sinal de dois pontos. Caso contrário, esses dois pontos seriam vistos como os delimitadores do protocolo. Então, precisamos de duas expressões regulares para corresponder ao caminho de uma URL relativa. Correspondemos aos caminhos relativos com ‹[a-z0-9\-._~%!$&'()*+,;=@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?›, de maneira muito

semelhante à expressão regular feita para caminhos que tenham um protocolo, mas que não tenham uma autoridade. As únicas diferenças são a barra opcional no início, que está faltando, e a primeira classe de caracteres, que não inclui os dois pontos. Nós correspondemos a caminhos absolutos com ‹(/[a-z0-9\-._~%!$&'()*+,;=:@]+)+/?›. Esta é a mesma expressão regular feita para caminhos em URLs que especificam um protocolo e uma autoridade, exceto pelo fato de que o asterisco que repete os segmentos do caminho tornou-se um sinal de adição. URLs relativas exigem pelo menos um segmento de caminho.

A parte de consulta da URL é opcional. Se estiver presente, ela deve começar com um ponto de interrogação. A consulta é executada até o primeiro sinal de hash (#) na URL, ou até o final da URL. Como o sinal de hash não está entre os caracteres de pontuação válidos para a parte de consulta da URL, podemos corresponder facilmente a isso com ‹\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*›. Ambos os pontos de interrogação, nesta expressão regular, são caracteres literais. O primeiro fica fora de uma classe de caracteres, e deve ser escapado. O segundo fica dentro de uma classe de caracteres, onde ele é sempre um caractere literal.

A parte final de uma URL é o fragmento, também opcional. Ele começa com um sinal de hash e vai até o final da URL. ‹\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*› corresponde a isso.

Para facilitar o trabalho com as diversas partes da URL, usamos os grupos de captura nomeados. A receita 2.11 explica como a captura nomeada funciona nos diferentes sabores de expressão regular, discutidos neste livro. O .NET é o único sabor que consegue tratar múltiplos grupos com um mesmo nome, como se fossem um único grupo. Tal recurso é muito útil nesta situação, porque nossa expressão regular possui várias formas de corresponder ao caminho da URL, se o protocolo ou a autoridade forem especificados. Se dermos o mesmo nome para estes três grupos podemos simplesmente, consultar o grupo “path” para obter o caminho,

independentemente da URL ter um protocolo/autoridade.

Os outros sabores não suportam esse comportamento de captura nomeada do .NET, embora a maioria suporte a mesma sintaxe. No caso dos outros sabores, os três grupos de captura do caminho possuem nomes diferentes. Apenas um deles conterá, realmente, o caminho da URL, quando a correspondência for encontrada. Os outros dois não terão participado da correspondência.

Veja também:

Receitas 2.3, 2.8, 2.9 e 2.12.

7.8 Extrair o protocolo de uma URL

Problema

Você deseja extrair o protocolo da URL a partir de uma string que contenha uma URL. Por exemplo, você deseja extrair http de http://www.regexcookbook.com.

Solução

Extrair o protocolo de uma URL válida

```
^([a-z][a-z0-9+\-.]*):
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Extrair o protocolo enquanto valida a URL

```
\A
([a-z][a-z0-9+\-.]*):
(# Autoridade e caminho
 //
 ([a-z0-9\-._~%!$&'()*+,;=]+@)? # Usuário
 ([a-z0-9\-._~%]+ # Host nomeado
 |\[[a-f0-9:.]+\] # Host IPv6
 |\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # Host IPvFuture
 (:[0-9]+)? # Porta
 (/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/? # Caminho
```

```
|# Caminho sem autoridade
 (/?[a-z0-9\-._~%!$&'()*+,;=:@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)?
)
# Consulta
(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
# Fragmento
(\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?
\Z
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^([a-z][a-z0-9+\-.]*):(//([a-z0-9\-._~%!$&'()*+,;=]+@)?([a-z0-9\-._~%]+|↲
\[[a-f0-9:.]+\]|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\])(:[0-9]+)?↲
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?|(/?[a-z0-9\-._~%!$&'()*+,;=:@]+↲
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?)?)(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?↲
(#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Discussão

Extrair o protocolo de uma URL é fácil, se você já sabe que o texto de assunto é uma URL válida. Um protocolo de URL sempre ocorre no início da URL. O acento circunflexo (receita 2.5) especifica este requisito na expressão regular. O protocolo começa com uma letra, que pode ser seguido por outras letras, algarismos, sinais de adição, hífens e pontos. Correspondemos a isso com duas classes de caracteres ‹[a-z][a-z0-9+\-.]*› (receita 2.3).

O protocolo está separado do resto da URL por um sinal de dois pontos. Nós adicionamos estes dois pontos, na expressão regular, para garantirmos que corresponderemos ao protocolo apenas se a URL realmente começar com um. URLs relativas não começam com um protocolo. A sintaxe de URL especificada na RFC 3986 garante que URLs relativas não possuam sinal de dois pontos, a menos que esses dois pontos sejam precedidos por caracteres não permitidos em protocolos. Foi por isso que tivemos de excluir os dois pontos de uma das classes de caracteres para corresponder ao caminho, na

receita 7.7. Se você usar as expressões regulares desta receita em uma URL válida, mas relativa, elas não encontrarão correspondência.

Como a expressão regular corresponde a mais do que apenas o protocolo em si (que inclui os dois pontos), adicionamos um grupo de captura à expressão regular. Quando a expressão regular encontra uma correspondência, você pode recuperar o texto correspondido pelo primeiro (e único) grupo de captura, para obter o protocolo sem os dois pontos. A receita 2.9 informa tudo sobre grupos de captura. Veja a receita 3.9 para aprender a recuperar texto correspondido por grupos de captura em sua linguagem de programação favorita.

Se você ainda não sabe se o texto de assunto é uma URL válida, é possível usar uma versão simplificada da expressão regular da receita 7.7. Como queremos extrair o protocolo, podemos excluir URLs relativas, que não especificam um protocolo. Isso simplifica um pouco a expressão regular.

Como esta expressão regular corresponde a toda a URL, adicionamos um grupo de captura extra em torno da parte da expressão regular que corresponde ao protocolo. Recupere o texto correspondido pelo grupo de captura 1, para obter o protocolo da URL.

Veja também:

Receitas 2.9, 3.9 e 7.7.

7.9 Extrair o usuário de uma URL

Problema

Você deseja extrair o usuário de uma string que contenha uma URL. Por exemplo, você deseja extrair jan de ftp://jan@www.regexcookbook.com.

Solução

Extrair o usuário de uma URL válida

```
^[a-z0-9+\-.]+://([a-z0-9\-._~%!$&'()*+,;=]+)@
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Extrair o usuário enquanto valida a URL

```
\A
[a-z][a-z0-9+\-.]*:// # Scheme
([a-z0-9\-._~%!$&'()*+,;=]+)@ # Usuário
([a-z0-9\-._~%]+ # Host nomeado
|\[[a-f0-9:.]+\] # Host IPv6
|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # Host IPvFuture
(:[0-9]+)? # Porta
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/? # Caminho
(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)? # Consulta
(\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)? # Fragmento
\Z
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^[a-z][a-z0-9+\-.]*://([a-z0-9\-._~%!$&'()*+,;=]+)@([a-z0-9\-._~%]+|↲
\[[a-f0-9:.]+\]|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\])(:[0-9]+)?↲
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?↲
(#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Discussão

Extrair o usuário de uma URL é fácil, se você já sabe que o texto de assunto é uma URL válida. O nome do usuário, estando presente na URL, ocorre logo após o protocolo e as duas barras que dão início à parte da “autoridade” na URL. O nome de usuário está separado do nome do host, que o segue por um sinal @. Como os sinais @ não são válidos em nomes de host, podemos ter a certeza de estarmos extraindo a parte do nome de usuário em uma URL, caso

encontremos um sinal @ após as duas barras e antes da próxima barra na URL. Barras não são válidas em nomes de usuário, portanto não é necessário fazer qualquer verificação especial para elas.

Todas essas regras significam que podemos extrair facilmente o nome do usuário, se soubermos que a URL é válida. Apenas pulamos o protocolo com ‹[a-z0-9+\-.]+› e ://. Então, pegamos o nome do usuário. Se pudermos corresponder ao sinal @, sabemos que os caracteres anteriores formam o nome do usuário. A classe de caracteres ‹[a-z0-9\-._~%!$&'()*+,;=]› lista todos os caracteres válidos em nomes de usuário.

Esta expressão regular encontrará uma correspondência somente se a URL especificar um usuário. Caso isso aconteça, a expressão regular corresponderá às partes do protocolo e do usuário na URL. Portanto, adicionamos um grupo de captura à expressão regular. Quando a expressão regular encontra uma correspondência, você pode recuperar o texto correspondido pelo primeiro (e único) grupo de captura, para obter o nome do usuário sem qualquer tipo de delimitador ou outra parte da URL. A receita 2.9 informa tudo sobre grupos de captura. Veja também a receita 3.9, para aprender a recuperar texto correspondido por grupos de captura em sua linguagem de programação favorita.

Se você ainda não sabe se o texto do assunto é uma URL válida, pode usar uma versão simplificada da expressão regular da receita 7.7. Como queremos extrair o usuário, podemos excluir URLs que não especifiquem uma autoridade. A expressão regular na solução realmente corresponde apenas a URLs que especifiquem uma autoridade que inclua um nome de usuário. Exigir a parte da autoridade na URL torna a expressão regular um pouco mais simples. É ainda mais simples do que a utilizada na receita 7.8.

Como esta expressão regular corresponde a toda a URL, adicionamos um grupo extra de captura em torno da parte da expressão regular que corresponde ao usuário. Recupere o texto

correspondido pelo grupo de captura 1, para obter o usuário da URL.

Se você quiser uma expressão regular que corresponda a qualquer URL válida, inclusive àquelas que não especifiquem o usuário, pode utilizar uma das estudadas na receita 7.7. Naquela receita, a primeira expressão regular captura o usuário, se presente, no terceiro grupo de captura. O grupo de captura incluirá o símbolo @. Você pode adicionar um grupo de captura extra à expressão regular, se quiser capturar o nome do usuário sem o símbolo @.

Veja também:

Receitas 2.9, 3.9 e 7.7.

7.10 Extrair o host de uma URL

Problema

Você deseja extrair o host de uma string que contenha uma URL. Por exemplo, você deseja extrair www.regexcookbook.com de http://www.regexcookbook.com/.

Solução

Extrair o host de uma URL válida

```
\A
[a-z][a-z0-9+\-.]*:// # Scheme
([a-z0-9\-._~%!$&'()*+,;=]+@)? # Usuário
([a-z0-9\-._~%]+ # Host nomeado ou IPv4
|\[[a-z0-9\-._~%!$&'()*+,;=:]+\]) # Host IPv6+
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^[a-z][a-z0-9+\-.]*://([a-z0-9\-._~%!$&'()*+,;=]+@)?([a-z0-9\-._~%]+|↲
\[[a-z0-9\-._~%!$&'()*+,;=:]+\])
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Extrair o host enquanto valida a URL

```
\A
[a-z][a-z0-9+\-.]*:// # Scheme
([a-z0-9\-._~%!$&'()*+,;=]+@)? # Usuário
([a-z0-9\-._~%]+ # Host nomeado
|\[[a-f0-9:.]+\] # Host IPv6
|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # Host IPvFuture
(:[0-9]+)? # Porta
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/? # Caminho
(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)? # Consulta
(\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)? # Fragmento
\Z
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^[a-z][a-z0-9+\-.]*://([a-z0-9\-._~%!$&'()*+,;=]+@)?([a-z0-9\-._~%]+|↲
\[[a-f0-9:.]+\]|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\])(:[0-9]+)?↲
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?↲
(#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)?$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Discussão

Extrair o host de uma URL é fácil, se você sabe que seu texto de assunto é uma URL válida. Usamos ‹\A› ou ‹^› para ancorar a correspondência no início da string. ‹[a-z][az0-9+\-.]*://› pula o protocolo, e ‹([a-z0-9\-._~%!$&'()*+,;=]+@)?› pula o usuário opcional. O nome do host vem logo em seguida.

A RFC 3986 permite duas notações diferentes para o host. Nomes de domínio e endereços IPv4 são especificados sem colchetes, enquanto endereços IPv6 e endereços futuros de IP são especificados com colchetes. Temos de tratá-los separadamente, porque a notação com colchetes permite mais pontuações do que a sem colchetes. Em particular, sinais de dois pontos são permitidos entre colchetes, mas não em nomes de domínio ou endereços IPv4. Os dois pontos também são utilizados para separar o nome do host

(com ou sem colchetes) do número da porta.

‹[a-z0-9\-._~%]+› corresponde a nomes de domínio e a endereços IPv4. ‹\[[a-z0-9\-._~%!$&'()*+,;=:]+\]› trata a versão IPv6 e posterior. Combinamos ambos os padrões usando alternância (receita 2.8) em um grupo. O grupo de captura também nos permite extrair o nome do host.

Esta expressão regular encontrará uma correspondência somente se a URL especificar um host. Quando isso acontece, a expressão regular corresponderá às partes do protocolo, do usuário e do host, na URL. Quando a expressão regular encontra uma correspondência, você pode recuperar o texto correspondido pelo segundo grupo de captura, para obter o nome do host sem delimitadores ou outras partes da URL. O grupo de captura incluirá os colchetes, no caso de endereços IPv6. A receita 2.9 informa tudo sobre a captura de grupos. Veja a receita 3.9, para saber como recuperar o texto correspondido por grupos de captura em sua linguagem de programação favorita.

Se você ainda não sabe se o texto de assunto é uma URL válida, pode usar uma versão simplificada da expressão regular da receita 7.7. Como queremos extrair o host, podemos excluir URLs que não especifiquem autoridade. Isso torna a expressão regular um pouco mais simples. Trata-se de procedimento muito semelhante ao empregado na receita 7.9. A única diferença é que agora a parte do usuário da autoridade é, mais uma vez, opcional, como era na receita 7.7.

Esta expressão regular também usa alternância para as diferentes notações do host, que são mantidas em conjunto por um grupo de captura. Obtenha o texto correspondido pelo grupo de captura 2 para conseguir o host da URL.

Se quiser uma expressão regular que corresponda a qualquer URL válida, inclusive aquelas que não especificam o usuário, você pode usar uma das expressões regulares da receita 7.7. A primeira expressão regular naquela receita captura o host, se presente, no

quarto grupo de captura.

Veja também

Receitas 2.9, 3.9 e 7.7

7.11 Extrair a porta de uma URL

Problema

Você deseja extrair o número da porta de uma string que contenha uma URL. Por exemplo, você deseja extrair 80 de http://www.regexcookbook.com:80/.

Solução

Extrair a porta de uma URL válida

```
\A
[a-z][a-z0-9+\-.]*:// # Scheme
([a-z0-9\-._~%!$&'()*+,;=]+@)? # Usuário
([a-z0-9\-._~%]+ # Host nomeado ou IPv4
|\[[a-z0-9\-._~%!$&'()*+,;=:]+\]) # Host IPv6+
:(?<port>[0-9]+) # Número da porta
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^[a-z][a-z0-9+\-.]*://([a-z0-9\-._~%!$&'()*+,;=]+@)?↵
([a-z0-9\-._~%]+|\[[a-z0-9\-._~%!$&'()*+,;=:]+\]):([0-9]+)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Extrair o host enquanto valida a URL

```
\A
[a-z][a-z0-9+\-.]*:// # Scheme
([a-z0-9\-._~%!$&'()*+,;=]+@)? # Usuário
([a-z0-9\-._~%]+ # Host nomeado
|\[[a-f0-9:.]+\] # Host IPv6
|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]+\]) # Host IPvFuture
```

```
:([0-9]+) # Porta
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/? # Caminho
(\?[a-z0-9\-._~%!$&'()*+,;=:@/?]*)? # Consulta
(\#[a-z0-9\-._~%!$&'()*+,;=:@/?]*)? # Fragmento
\Z
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^[a-z][a-z0-9+\-.]*:\/\/([a-z0-9\-._~%!$&'()*+,;=]+@)?↲
([a-z0-9\-._~%]+|\[[a-f0-9:.]+\]|\[v[a-f0-9][a-z0-9\-._~%!$&'()*+,;=:]↲
+\]):([0-9]+)(\/[a-z0-9\-._~%!$&'()*+,;=:@]+)*\/?↲
(\?[a-z0-9\-._~%!$&'()*+,;=:@\/?]*)?(#[a-z0-9\-._~%!$&'()*+,;=:@\/?]*)?$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Discussão

Extrair o número da porta de uma URL é fácil, se você já sabe que o texto de assunto é uma URL válida. Usamos ‹\A› ou ‹^› para ancorar a correspondência ao início da string. ‹[a-z][a-z0-9+\-.]*://› pula o protocolo, e ‹([a-z0-9\-._~%!$&'()*+,;=]+@)?› pula o usuário opcional. ‹([a-z0-9\-._~%]+|\[[a-z0-9\-._~%!$&'()*+,;=:]+\])› pula o nome do host.

O número da porta é separado do nome do host por um sinal de dois pontos, que adicionamos como um caractere literal à expressão regular. O número da porta, em si, é simplesmente uma string de dígitos, facilmente correspondida por ‹[0-9]+›.

Essa expressão regular encontra correspondência apenas se a URL especificar um número de porta. Quando isso acontece, a expressão regular corresponde às partes de protocolo, usuário, host e número de porta da URL. Quando a expressão regular encontra uma correspondência, você pode recuperar o texto correspondido pelo terceiro grupo de captura, para obter o número da porta sem delimitadores ou outras partes da URL.

Os outros dois grupos são usados para tornar opcional o nome do usuário, e para manter reunidas as duas alternativas para o nome do host. A receita 2.9 fala tudo sobre os grupos de captura. Veja a

receita 3.9, para aprender como recuperar o texto correspondido pelos grupos de captura em sua linguagem de programação favorita.

Se você ainda não sabe que o texto de assunto é uma URL válida, pode usar uma versão simplificada da expressão regular da receita 7.7. Como queremos extrair o host, podemos excluir URLs que não especifiquem autoridade. Isso torna a expressão regular um pouco mais simples. Trata-se de procedimento muito semelhante ao usado na receita 7.10.

A única diferença é que, dessa vez, o número da porta não é opcional, e movemos o grupo de captura do número da porta para excluir o sinal de dois pontos que separa o número da porta e o nome do host. O número do grupo de captura é 3.

Se quiser uma expressão regular que corresponda a qualquer URL válida, inclusive as que não especificam a porta, você pode usar uma das expressões estudadas na receita 7.7. Naquela receita, a primeira expressão regular captura a porta, se presente, no quinto grupo de captura.

Veja também:

Receitas 2.9, 3.9 e 7.7.

7.12 Extrair o caminho de uma URL

Problema

Você deseja extrair o caminho de uma string que contenha uma URL. Por exemplo, você deseja extrair /index.html de http://www.regexcookbook.com/index.html ou de /index.html#fragment.

Solução

Vamos extrair o caminho de uma string que contém uma URL válida. A seguinte expressão regular localiza uma correspondência para todas as URLs, mesmo para aquelas que não contenham um

caminho:

```
\A
# Pule o scheme e a autoridade, caso existam
([a-z][a-z0-9+\-.]*:(//[^/?#]+)?)?
# Caminho
([a-z0-9\-._~%!$&'()*+,;=:@/]*)
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^([a-z][a-z0-9+\-.]*:(//[^/?#]+)?)?([a-z0-9\-._~%!$&'()*+,;=:@/]*)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Vamos extrair o caminho de uma string que sabemos possuir uma URL válida. Só corresponda a URLs que realmente possuam um caminho:

```
\A
# Pule o scheme e a autoridade, caso existam
([a-z][a-z0-9+\-.]*:(//[^/?#]+)?)?
# Caminho
(/?[a-z0-9\-._~%!$&'()*+,;=@]+(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?|/)
# Consulta, fragmento, ou final da URL
([#?]|\Z)
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^([a-z][a-z0-9+\-.]*:(//[^/?#]+)?)?(/?[a-z0-9\-._~%!$&'()*+,;=@]+↵
(/[a-z0-9\-._~%!$&'()*+,;=:@]+)*/?|/)([#?]|$)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Vamos extrair o caminho de uma string que contenha uma URL válida. Use um agrupamento atômico, para corresponder apenas às URLs que realmente possuam um caminho:

```
\A
# Pule o scheme e a autoridade, caso existam
(?>([a-z][a-z0-9+\-.]*:(//[^/?#]+)?)?)
# Caminho
```

([a-z0-9\-._~%!$&'()*+,;=:@/]+)

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Ruby

Discussão

Você pode usar uma expressão regular muito mais simples para extrair o caminho, se você já sabe que o texto de assunto é uma URL válida. Embora a expressão regular genérica, na receita 7.7, possua três maneiras diferentes de corresponder ao caminho, dependendo de a URL especificar um protocolo, ou uma autoridade, a expressão regular específica que extrai o caminho de uma URL válida deverá corresponder ao caminho apenas uma vez.

Começamos com ‹\A› ou ‹^› para ancorar a correspondência ao início da string. ‹[a-z][a-z0-9+\-.]*:› pula o protocolo, e ‹//[^/?#]+› pula a autoridade. Podemos usar essa expressão regular muito simples para a autoridade, porque já sabemos que ela é válida, e não estamos interessados em extrair o usuário, host ou a porta de dentro da autoridade.

A autoridade começa com duas barras, e vai até o início do caminho (barra), consulta (ponto de interrogação) ou fragmento (símbolo de hash). A classe de caracteres negada corresponde a tudo até a primeira barra, ponto de interrogação ou hash (receita 2.3).

Como a autoridade é opcional, podemos colocá-la em um grupo seguido pelo quantificador ponto de interrogação: ‹(//[^/?#]+)?›. O protocolo também é opcional. Se o protocolo for omitido, a autoridade também deverá ser omitida. Para corresponder a isso, colocamos as partes da expressão regular relacionadas ao protocolo e à autoridade opcional em outro grupo, também tornada opcional graças a um ponto de interrogação.

Como sabemos que a URL é válida, podemos facilmente corresponder ao caminho com uma única classe de caracteres ‹[a-z0-9\-._~%!$&'()*+,;=:@/]*›, que inclui a barra. Não precisamos verificar

a existência de barras consecutivas, não permitidas em caminhos de URLs.

Nós realmente usamos um asterisco, em vez de um sinal de adição, como o quantificador na classe de caracteres do caminho. Pode parecer estranho tornar opcional o caminho em uma expressão regular que só existe para extrair justamente isso em uma URL. Na verdade, tornar o caminho opcional é essencial, por causa dos atalhos que tomamos ao pular o protocolo e a autoridade.

Na expressão regular genérica para URLs, na receita 7.7, temos três maneiras diferentes de corresponder ao caminho, de acordo com a presença do protocolo ou da autoridade na URL. Isso garante que o protocolo não será correspondido acidentalmente como caminho.

Agora, vamos tentar manter as coisas simples, usando apenas uma classe de caracteres para o caminho. Considere a URL http://www.regexcookbook.com, que possui um protocolo e uma autoridade, mas nenhum caminho. A primeira parte de nossa expressão regular corresponderá sem problemas ao protocolo e à autoridade. O mecanismo de expressão regular, então, tenta corresponder à classe de caracteres do caminho, mas não sobraram caracteres. Se o caminho for opcional (usando o quantificador asterisco), o mecanismo de expressão regular ficará perfeitamente feliz em não corresponder a qualquer caractere relativo ao caminho. Ele atinge o final da expressão regular e declara que uma correspondência total foi encontrada.

Porém, se a classe de caracteres do caminho não for opcional, o mecanismo de expressão regular retrocederá (veja a receita 2.13, se não estiver familiarizado com o mecanismo de retrocesso). Ele lembrou que as partes da autoridade e do protocolo de nossa expressão regular são opcionais; então, é como se o mecanismo dissesse: “Vamos tentar novamente, sem permitir que ‹(//[^/?#]+)?› corresponda a alguma coisa”. ‹[a-z0-9\-._~%!$&'()*+,;=:@/]+›, então, corresponde a //www.regexcookbook.com como se fosse o caminho,

claramente o que não queremos. Se usássemos uma expressão regular mais precisa para o caminho, que desautorizasse as barras duplas, o mecanismo de expressão regular simplesmente retrocederia novamente, e fingiria que a URL não possui um protocolo. Com uma expressão regular mais precisa, corresponderíamos a http como sendo o caminho. Para evitar isso, também teríamos de acrescentar uma verificação adicional, para garantir que o caminho seja seguido pela consulta, pelo fragmento ou por nada. Se fizermos tudo isso, acabaremos com a expressão regular indicada como "só corresponda a URLs que realmente possuam um caminho", na "Solução" desta receita. Estas expressões regulares são um pouco mais complicadas em relação às duas primeiras; tudo apenas para fazer a expressão regular não corresponder a URLs sem um caminho.

Se o seu sabor de expressão regular suporta agrupamentos atômicos, há uma maneira mais fácil. Todos os sabores discutidos neste livro, exceto o JavaScript e o Python, suportam agrupamento atômico (veja receita 2.15). Essencialmente, um grupo atômico diz ao mecanismo de expressão regular para não retroceder. Se colocarmos as partes do protocolo de autoridade de nossa expressão regular dentro de um grupo atômico, o mecanismo de expressão regular será forçado a manter a correspondência das partes do protocolo e da autoridade que tenham sido correspondidos, mesmo que isso não dê espaço à classe de caracteres para corresponder ao caminho. Esta solução é tão eficiente quanto tornar o caminho opcional.

Independentemente da expressão regular que você escolher, nesta receita, o terceiro grupo de captura conterá o caminho. O terceiro grupo de captura pode retornar a string vazia, ou null, no JavaScript, caso você use uma das duas primeiras expressões regulares que permitam o caminho opcional.

Caso ainda não saiba se o texto de assunto é uma URL válida, você poderá usar a expressão regular da receita 7.7. Se estiver usando o

.NET, poderá usar a expressão regular específica para o .NET, que inclui três grupos denominados “path”, para capturar as três partes da expressão regular que poderiam corresponder ao caminho da URL. Se estiver usando outro sabor que suporte capturas nomeadas, um dos três grupos terá capturado o caminho: “hostpath”, “schemepath” ou “relpath”. Como apenas um dos três grupos realmente vai capturar alguma coisa, um truque simples para obter o caminho é concatenar as strings retornadas pelos três grupos. Dois deles retornarão a string vazia, por isso nenhuma concatenação real é feita.

Se o seu sabor não suporta capturas nomeadas, você pode usar a primeira expressão regular da receita 7.7. Ela captura o caminho no grupo 6, 7 ou 8. Você pode usar o mesmo truque para concatenar o texto capturado por estes três grupos, pois dois deles retornarão a string vazia. Em JavaScript, no entanto, isso não vai funcionar. O JavaScript retorna undefined para grupos que não participaram da correspondência.

A receita 3.9 tem mais informações sobre como recuperar texto correspondido por grupos de captura numerados e nomeados, levando em conta sua linguagem de programação favorita.

Veja também:

Receitas 2.9, 3.9 e 7.7.

7.13 Extrair a consulta de uma URL

Problema

Você deseja extrair a consulta (query) de uma string que contenha uma URL. Por exemplo, você deseja extrair param=value de http://www.regexcookbook.com?param=value ou de /index.html?param=value.

Solução

```
^[^?#]+\?([^#]+)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Extrair a consulta de uma URL é algo trivial, se você souber que o texto de assunto é uma URL válida. A consulta é separada da parte da URL anterior a ela por um ponto de interrogação. Esse é o primeiro ponto de interrogação permitido em qualquer lugar dentro de URLs. Deste modo, podemos facilmente pular para o primeiro ponto de interrogação com ‹^[^?#]+\?›. O ponto de interrogação é um metacaractere somente fora das classes de caractere, não dentro; por isso, escapamos o ponto de interrogação literal fora da classe de caracteres. O primeiro ‹^› é uma âncora (receita 2.5), enquanto o segundo ‹^› nega a classe de caracteres (receita 2.3).

Pontos de interrogação podem aparecer em URLs como parte (opcional) do fragmento, após a consulta. Então, precisamos usar ‹^[^?#]+\?›, ao invés de apenas ‹\?›, para garantir que obteremos o primeiro ponto de interrogação da URL, e para garantir que ele não faça parte do fragmento, em uma URL sem consulta.

A consulta vai até o início do fragmento, ou até o final da URL, se não houver fragmento separado do resto da URL por um sinal de hash. Como os sinais de hash não são permitidos em nenhum lugar, exceto no fragmento, ‹[^#]+› é tudo que precisamos para corresponder à parte da consulta. A classe de caracteres negada corresponde a tudo até o primeiro sinal de hash, ou até o final do assunto, caso não haja sinais de hash.

Esta expressão regular encontrará correspondência apenas para URLs que contenham consulta. Quando ela corresponde a uma URL, a correspondência inclui tudo, desde o início da URL, por isso colocamos a parte ‹[^#]+› da expressão regular, que corresponde à parte da consulta, dentro de um grupo de captura. Quando a expressão regular encontra uma correspondência, você pode recuperar o texto correspondido pelo primeiro (e único) grupo de

captura, para obter a consulta sem delimitadores ou outras partes da URL. A receita 2.9 informa tudo sobre captura de grupos. Veja a receita 3.9, para aprender como recuperar texto correspondido por grupos de captura em sua linguagem de programação favorita.

Se você não sabe se o texto de assunto é uma URL válida, pode usar uma das expressões regulares da receita 7.7. A primeira expressão regular naquela receita captura a consulta, caso exista, no grupo de captura número 12.

Veja também:

Receitas 2.9, 3.9 e 7.7.

7.14 Extrair o fragmento de uma URL

Problema

Você deseja extrair o fragmento de uma string que contenha uma URL. Por exemplo, você deseja extrair top de http://www.regexcookbook.com#top ou de /index.html#top.

Solução

```
#(.+)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Extrair o fragmento de uma URL é algo trivial, se você souber que o texto de assunto é uma URL válida. A consulta é separada da parte da URL anterior a ela por um sinal de hash (#). O fragmento é a única parte das URLs em que os sinais de hash (#) são permitidos; o fragmento é sempre a última parte da URL. Assim podemos, facilmente extrair o fragmento, encontrando o primeiro sinal de hash e pegando tudo até o final da string. ‹#.+› faz isso muito bem. Certifique-se de desativar o modo de espaçamento livre, caso

contrário, você precisará escapar o sinal de hash literal com uma barra invertida.

Esta expressão regular encontrará uma correspondência somente no caso das URLs que contenham um fragmento. A correspondência consiste apenas no fragmento, mas inclui o sinal de hash (#), que separa o dito fragmento do resto da URL. A solução possui um grupo de captura extra para recuperar apenas o fragmento, sem o delimitador #.

Se você ainda não sabe se o texto de assunto é uma URL válida, pode usar uma das expressões regulares da receita 7.7. A primeira expressão regular naquela receita captura o fragmento, se estiver presente na URL, no grupo de captura número 13.

Veja também:

Receitas 2.9, 3.9 e 7.7.

7.15 Validar nomes de domínio

Problema

Você deseja verificar se uma string é um nome de domínio totalmente qualificado e válido, ou encontrar nomes de domínio em textos mais longos.

Solução

Verificar se uma string se parece com um nome de domínio válido:

```
^([a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,}$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A([a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,}\Z
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Encontrar nomes de domínio válidos em textos mais longos:

```
\b([a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,}\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Verificar se as partes do domínio não são maiores do que 63 caracteres:

```
\b((?=[a-z0-9-]{1,63}\.)[a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,63}\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Permitir nomes de domínio internacionalizados, usando a notação punycode:

```
\b((xn--)?[a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,}\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Verificar se as partes do domínio não têm mais do que 63 caracteres, permitindo nomes de domínio internacionalizados com notação punycode:

```
\b((?=[a-z0-9-]{1,63}\.)(xn--)?[a-z0-9]+(-[a-z0-9]+)*\.)+[a-z]{2,63}\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Um nome de domínio possui o formato `domínio.tld`, `subdomínio.domínio.tld`, ou qualquer quantidade de subdomínios adicionais. O domínio de nível mais alto (top-level domain, ou `tld`) é constituído por duas, ou mais, letras. Esta é a parte mais fácil da expressão regular: `‹[a-z]{2,}›`.

O domínio e os subdomínios são compostos por letras, números e hífens. Hífens não podem aparecer aos pares, e não podem aparecer como o primeiro, ou o último, caractere no domínio. Tratamos dessa questão com a expressão regular `‹[a-z0-9]+(-[a-z0-9]+)*›`. Esta expressão regular permite qualquer número de letras e dígitos, opcionalmente seguidos por qualquer número de grupos constituídos por um hífen seguido por outra sequência de letras e algarismos. Lembre-se de que o hífen é um metacaractere dentro

das classes de caracteres (receita 2.3), mas é um caractere comum fora das classes de caracteres; por isso, não precisamos escapar hífens nesta expressão regular.

O domínio e os subdomínios são delimitados por um ponto literal, que correspondemos com ‹\.› em uma expressão regular. Como podemos ter qualquer quantidade de subdomínios, além do domínio, colocamos a parte do nome de domínio da expressão regular e o ponto literal em um grupo que será repetido: ‹([a-z0-9]+(-[a-z0-9]+)*\.)+›. Como os subdomínios seguem a mesma sintaxe do domínio, este grupo trata ambos.

Se quiser verificar se uma string representa um nome de domínio válido, tudo o que resta é adicionar âncoras no início e no fim da expressão regular, que correspondam ao início e ao final da string. Podemos fazer isso com ‹^› e ‹$› em todos os sabores, exceto o Ruby, e com ‹\A› e ‹\Z› em todos os sabores, exceto o JavaScript. A receita 2.5 tem todos os detalhes.

Se quiser encontrar nomes de domínio em um corpo de texto maior, você pode adicionar extremidades de palavra (‹\b›; veja a receita 2.6).

O nosso primeiro conjunto de expressões regulares não verifica se cada parte do domínio possui mais do que 63 caracteres. Não podemos fazer isso facilmente, pois nossa expressão regular relativa a cada parte do domínio, ‹[a-z0-9]+(-[a-z0-9]+)*›, possui três quantificadores. Não há nenhuma maneira de dizer ao mecanismo de expressão regular para fazê-los somarem 63.

Poderíamos usar ‹[-a-z0-9]{1,63}›, para corresponder a uma parte do domínio que tenha entre 1 e 63 caracteres de comprimento, ou ‹\b([-a-z0-9]{1,63}\.)+[a-z]{2,63}› para todo o nome de domínio. Dessa forma, já não estaremos excluindo os domínios com hífens nos lugares errados.

O que podemos fazer é usar um lookahead, para corresponder ao mesmo texto duas vezes. Reveja a receita 2.16 primeiro, caso não esteja familiarizado com lookaheads. Usamos a mesma expressão

regular, ‹[a-z0-9]+(-[a-z0-9]+)*\.›, para corresponder ao nome de domínio com hífens válidos, e adicionamos ‹[-a-z0-9]{1,63}\.› dentro de um lookahead, para verificar se ele também possui 63 caracteres de comprimento ou menos. O resultado é ‹(?=[-a-z0-9]{1,63}\.)[a-z0-9]+(-[a-z0-9]+)*\.›.

O lookahead ‹(?=[-a-z0-9]{1,63}\.)› primeiro verifica se existem de 1 a 63 letras, dígitos e hífens, até o próximo ponto. É importante incluir o ponto no lookahead. Sem ele, os domínios com mais de 63 caracteres ainda assim satisfariam o requisito do lookahead. Somente com a colocação do ponto literal, dentro do lookahead, cumpriremos o requisito de obtermos, no máximo, 63 caracteres.

O lookahead não consome o texto ao qual corresponde. Assim, se o lookahead for bem-sucedido, ‹[a-z0-9]+(-[a-z0-9]+)*\.› será aplicado ao texto já correspondido por ele. Já confirmamos que não há mais de 63 caracteres; agora, testamos para ver se eles formam a combinação correta de hífens e não hífens.

Nomes de domínio internacionalizados (IDNs), em teoria, podem conter praticamente qualquer caractere. A lista atual de caracteres depende do registro que gerencia o domínio de nível mais alto. Por exemplo, .es permite nomes de domínio com caracteres em espanhol.

Na prática, os nomes de domínio internacionalizados são, frequentemente, codificados usando um protocolo chamado *punycode*[1]. Embora o algoritmo punycode seja bastante complicado, o que importa, aqui, é que ele resulta em nomes de domínio que formam combinação de letras, dígitos e hífens, seguindo as regras já tratadas, em nossa expressão regular, para nomes de domínio. A única diferença é que o nome de domínio produzido pelo punycode é prefixado com xn--. Para adicionar suporte a esses tipos de domínios, em nossa expressão regular, basta adicionar ‹(xn--)?› ao grupo que corresponda às partes do nome de domínio.

Veja também:

Receitas 2.3, 2.12, e 2.16.

7.16 Corresponder a endereços IPv4

Problema

Você deseja verificar se uma determinada string representa um endereço IPv4 válido, na notação 255.255.255.255. Opcionalmente, você deseja converter este endereço em um número inteiro de 32 bits.

Solução

Expressão regular

Expressão regular simples para verificar um endereço IP:

```
^(?:[0-9]{1,3}\.){3}[0-9]{1,3}$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Expressão regular precisa, para verificar um endereço IP:

```
^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↵
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Expressão regular simples, para extrair endereços IP de um texto mais longo:

```
\b(?:[0-9]{1,3}\.){3}[0-9]{1,3}\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Expressão regular precisa, para extrair endereços IP de textos mais longos:

```
\b(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↵
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Expressão regular simples, que captura as quatro partes do endereço IP:

```
^([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Expressão regular precisa, que captura as quatro partes do endereço IP:

```
^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.↲
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.↲
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.↲
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Perl

```
if ($subject =~ m/^([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})/)
{
  $ip = $1 << 24 | $2 << 16 | $3 << 8 | $4;
}
```

Discussão

Um endereço IP, versão 4, geralmente é escrito na forma 255.255.255.255, em que cada um dos quatro números deve estar entre 0 e 255. Corresponder a esses endereços IP com uma expressão regular é muito simples.

Na solução, apresentamos seis expressões regulares. Três delas são taxadas como “simples”, enquanto as outras três são marcadas como “precisas”.

As expressões regulares simples utilizam {[0-9]{1,3}}, para corresponder a cada um dos quatro blocos de dígitos no endereço IP. Essas expressões regulares, na verdade, permitem números de 0 a 999, e não de 0 a 255. As expressões regulares simples são mais eficientes quando você já sabe que sua entrada conterá

apenas endereços IP válidos: você só precisa separar os endereços IP das outras coisas.

Expressões regulares precisas utilizam ‹25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?› para corresponder a cada um dos quatro números do endereço IP. Esta expressão regular corresponde, precisamente, a um número no intervalo de 0 a 255, com um zero à esquerda opcional para números entre 10 e 99, e dois zeros à esquerda opcionais para números entre 0 e 9. ‹25[0-5]› corresponde de 250 a 255, ‹2[0-4][0-9]› corresponde de 200 a 249, e ‹[01]?[0-9][0-9]?› cuida de 0 a 199, incluindo os zeros opcionais à esquerda. A receita 6.5 explica, em detalhes, como corresponder a intervalos numéricos com uma expressão regular.

Se quiser verificar se uma string representa um endereço IP válido em sua totalidade, utilize uma das expressões regulares que começam com um sinal de acento circunflexo e terminam com um cifrão. Eles formam as âncoras de início-de-string e de final-de-string, explicadas na receita 2.5. Se quiser encontrar endereços IP dentro de um texto mais longo, utilize uma das expressões regulares que começa e termina com extremidades de palavra ‹\b› (receita 2.6).

As quatro primeiras expressões regulares utilizam o formato ‹(?:number\.){3}number›. Os três primeiros números, no endereço IP, são correspondidos por um grupo de não-captura (receita 2.9), repetido três vezes (receita 2.12). O grupo corresponde a um número e a um ponto literal, dos quais há três em um endereço IP. A última parte da expressão regular corresponde ao número final, no endereço IP. Utilizando o grupo de não-captura, e repetindo-o três vezes, nossa expressão regular torna-se mais curta e mais eficiente.

Para converter a representação textual do endereço IP em um número inteiro, é preciso capturar os quatro números separadamente. As duas últimas expressões regulares, na solução, fazem isso. Em vez de usar o truque de repetir um grupo três vezes, elas possuem quatro grupos de captura, um para cada número.

Soletrar, assim, é a única forma de conseguirmos capturar separadamente todos os quatro números do endereço IP.

Capturados os números, combiná-los em um número de 32 bits é fácil. Em Perl, as variáveis especiais $1, $2, $3 e $4 contêm o texto correspondido pelos quatro grupos de captura da expressão regular. A receita 3.9 explica como recuperar grupos de captura em outras linguagens de programação. Em Perl, as variáveis de string, para os grupos de captura, são automaticamente convertidas em números, quando aplicamos nelas o operador bitwise de deslocamento à esquerda (<<). Em outras linguagens, talvez você tenha de chamar String.toInteger(), ou algo similar, antes de deslocar os números e combiná-los com uma operação or bitwise.

Veja também:

Receitas 2.3, 2.8, 2.9 e 2.12.

7.17 Corresponder a endereços IPv6

Problema

Você deseja verificar se uma string representa um endereço IPv6 válido, utilizando as notações padrão, compacta ou mista.

Solução

Notação padrão

Corresponda a um endereço IPv6, na notação padrão, que consiste em oito palavras de 16 bits usando notação hexadecimal, delimitadas por sinais de dois pontos (por exemplo: 1762:0:0:0:0:B03:1:AF18). Zeros à esquerda são opcionais.

Verificar se todo o texto de assunto é um endereço IPv6 utilizando notação padrão:

```
^(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

```
\A(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}\Z
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Encontrar um endereço IPv6, utilizando a notação padrão dentro de uma coleção de texto maior:

```
(?<![:.\w])(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}(?![:.\w])
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby 1.9

JavaScript e Ruby 1.8 não suportam lookbehind. Temos de remover a verificação no início da expressão regular que a impede de encontrar endereços IPv6 dentro de sequências mais longas de dígitos hexadecimais e sinais de dois pontos. Uma extremidade de palavra realiza parte do teste:

```
\b(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Notação mista

Corresponda a um endereço IPv6, na notação mista, que consista em seis palavras de 16 bits usando a notação hexadecimal, seguido por quatro bytes, usando notação decimal. As palavras são delimitadas por sinais de dois pontos, e os bytes são delimitados por pontos. Um sinal de dois pontos separa as palavras dos bytes. Zeros à esquerda são opcionais, para as palavras hexadecimais e para os bytes decimais. Esta notação é utilizada em situações nas quais IPv4 e IPv6 são misturados, e os endereços IPv6 são extensões dos endereços IPv4. 1762:0:0:0:0:B03:127.32.67.15 é um exemplo de um endereço IPv6 na notação mista.

Verificar se todo o texto de assunto é um endereço IPv6 usando notação mista:

```
^(?:[A-F0-9]{1,4}:){6}(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↵
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Encontrar o endereço IPv6, usando notação mista dentro de uma coleção de texto maior:

```
(?<![:.\w])(?:[A-F0-9]{1,4}:){6}↲
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↲
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)(?![:.\w])
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

JavaScript e Ruby 1.8 não suportam lookbehind. Temos de remover a verificação no início da expressão regular que a impede de encontrar endereços IPv6 dentro de sequências mais longas de dígitos hexadecimais e sinais de dois pontos. Uma extremidade de palavra realiza parte do teste:

```
\b(?:[A-F0-9]{1,4}:){6}(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↲
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Notação padrão ou mista

Corresponda a um endereço IPv6 utilizando notação padrão ou mista.

Verificar se todo o texto de assunto representa um endereço IPv6 utilizando notação padrão ou mista:

```
\A # Início da string
(?:[A-F0-9]{1,4}:){6} # 6 palavras
(?:[A-F0-9]{1,4}:[A-F0-9]{1,4} # 2 palavras
| (?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3} # ou 4 bytes
  (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
)\Z # Fim da string
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:[A-F0-9]{1,4}:){6}(?:[A-F0-9]{1,4}:[A-F0-9]{1,4}|↲
```

```
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↲
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?))$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Encontrar um endereço IPv6 utilizando notação padrão ou mista dentro de uma coleção de texto maior:

```
(?<![:.\w]) # Ancora o endereço
(?:[A-F0-9]{1,4}:){6} # 6 palavras
(?:[A-F0-9]{1,4}:[A-F0-9]{1,4} # 2 palavras
| (?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3} # ou 4 bytes
  (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
)(?![:.\w]) # Ancora o endereço
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby 1.9

JavaScript e Ruby 1.8 não suportam lookbehind. Temos de remover a verificação no início da expressão regular que a impede de encontrar endereços IPv6 dentro de sequências mais longas de dígitos hexadecimais e sinais de dois pontos. Uma extremidade de palavra realiza parte do teste:

```
\b # Extremidade de palavra
(?:[A-F0-9]{1,4}:){6} # 6 palavras
(?:[A-F0-9]{1,4}:[A-F0-9]{1,4} # 2 palavras
| (?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3} # ou 4 bytes
  (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
)\b # Extremidade de palavra
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
\b(?:[A-F0-9]{1,4}:){6}(?:[A-F0-9]{1,4}:[A-F0-9]{1,4}| ↲
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3} ↲
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?))\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Notação compacta

Corresponda a um endereço IPv6, usando notação compacta. Notação compacta é o mesmo que notação padrão, exceto pelo fato de que uma sequência de uma ou mais palavras iguais a zero podem ser omitidas, deixando apenas os sinais de dois pontos, antes e depois dos zeros omitidos. Endereços utilizando notação compacta podem ser reconhecidos pela ocorrência de dois sinais de dois pontos adjacentes no endereço. Somente uma sequência de zeros pode ser omitida; caso contrário, seria impossível determinar quantas palavras foram omitidas em cada sequência. Se a sequência omitida ocorrer no início ou no final do endereço IP, ele vai começar ou terminar com dois sinais de dois pontos. Se todos os números forem zero, o endereço IPv6 compacto consistirá apenas em dois sinais de dois pontos, sem dígitos.

Por exemplo, 1762::B03:1:AF18 é a forma compacta de 1762:0:0:0:0:B03:1:AF18. As expressões regulares, nesta seção, corresponderão tanto ao formato compacto quanto ao formato padrão do endereço IPv6.

Verificar se todo o texto de assunto é um endereço IPv6, usando notação padrão ou compacta:

```
\A(?:
 # Padrão
 (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}
 # Compacta com, no máximo, 7 sinais de dois pontos
|(?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}
   \Z) # e ancorada
 # e com, no máximo, um sinal duplo de dois pontos
 (([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)
)\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}|↲
(?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}$)(([0-9A-F]{1,4}:){1,7}|:)↲
((:[0-9A-F]{1,4}){1,7}|:))$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Encontrar um endereço IPv6 utilizando notação padrão ou compacta dentro de uma coleção de texto maior:

```
(?<![:.\w])(?:
 # Padrão
 (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}
 # Compacta com, no máximo, 7 sinais de dois pontos
|(?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}
    (?![:.\w])) # e ancorada
 # e com, no máximo, um sinal duplo de dois pontos
 (([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)
)(?![:.\w])
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby 1.9

JavaScript e Ruby 1.8 não suportam lookbehind, por isso temos de remover a verificação no início da expressão regular que a impede de encontrar endereços IPv6 dentro de sequências mais longas de dígitos hexadecimais e dois pontos. Não podemos usar uma extremidade de palavra, pois o endereço pode começar com um sinal de dois pontos, que não é um caractere de palavra:

```
(?:
 # Padrão
 (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}
 # Compacta com, no máximo, 7 sinais de dois pontos
|(?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}
    (?![:.\w])) # e ancorada
 # e com, no máximo, um sinal duplo de dois pontos
 (([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)
)(?![:.\w])
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
(?:(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}|(?=(?:[A-F0-9]{0,4}:){0,7}↲
[A-F0-9]{0,4}(?![:.\w]))(([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4})↲
{1,7}|:))(?![:.\w])
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Notação mista compacta

Corresponda a um endereço IPv6, usando notação mista compacta. Notação mista compacta é o mesmo que notação mista, exceto pelo fato de que uma sequência de uma ou mais palavras iguais a zero podem ser omitidas, deixando apenas os sinais de dois pontos antes e após os zeros omitidos. Os 4 bytes decimais devem ser todos especificados, mesmo que sejam zero. Endereços utilizando notação mista compacta podem ser reconhecidos pela ocorrência de dois sinais de dois pontos adjacentes, na primeira parte do endereço, e os três pontos na segunda parte. Apenas uma sequência de zeros pode ser omitida; caso contrário, seria impossível determinar quantas palavras foram omitidas em cada sequência. Se a sequência omitida estiver no início do endereço IP, ele vai começar com um sinal duplo de dois pontos, e não com um dígito.

Por exemplo, o endereço IPv6 1762::B03:127.32.67.15 é o formato compacto de 1762:0:0:0:0:B03:127.32.67.15. As expressões regulares, nesta seção, corresponderão a endereços IPv6 compactos e não-compactos que utilizam notação mista.

Verificar se o texto de assunto total compõe um endereço IPv6, utilizando notação mista compacta ou não-compacta:

```
\A
(?:
 # Não-compacta
 (?:[A-F0-9]{1,4}:){6}
 # Compacta com, no máximo, 6 sinais de dois pontos
|(?=(?:[A-F0-9]{0,4}:){0,6}
    (?:[0-9]{1,3}\.){3}[0-9]{1,3} # e 4 bytes
    \Z) # e ancorada
 # e com, no máximo, um sinal duplo de dois pontos
 (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)
)
# 255.255.255.
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
```

```
# 255
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:(?:[A-F0-9]{1,4}:){6}|(?=(?:[A-F0-9]{0,4}:){0,6}(?:[0-9]{1,3}\.)↵
{3}[0-9]{1,3}$)(([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:))↵
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4]↵
[0-9]|[01]?[0-9][0-9]?)$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Encontrar endereço IPv6, utilizando notação mista compacta, ou notação não-compacta, dentro de uma coleção de texto maior:

```
(?<![:.\w])
(?:
 # Não-compacta
 (?:[A-F0-9]{1,4}:){6}
 # Compacta com, no máximo, 6 sinais de dois pontos
|(?=(?:[A-F0-9]{0,4}:){0,6}
    (?:[0-9]{1,3}\.){3}[0-9]{1,3} # e 4 bytes
    (?![:.\w])) # e ancorada
 # e com, no máximo, um sinal duplo de dois pontos
 (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)
)
# 255.255.255.
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
# 255
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
(?![:.\w])
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby 1.9

JavaScript e Ruby 1.8 não suportam lookbehind, por isso temos de remover a verificação no início da expressão regular que a impede de encontrar endereços IPv6 dentro de sequências mais longas de dígitos hexadecimais e dois pontos. Não podemos usar uma

extremidade de palavra, pois o endereço pode começar com um sinal de dois pontos, que não é caractere de palavra:

```
(?:
 # Não-compacta
 (?:[A-F0-9]{1,4}:){6}
 # Compacta com, no máximo, 6 sinais de dois pontos
|(?=(?:[A-F0-9]{0,4}:){0,6}
    (?:[0-9]{1,3}\.){3}[0-9]{1,3} # e 4 bytes
    (?![:.\w])) # e ancorada
 # e com, no máximo, um sinal duplo de dois pontos
 (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)
)
# 255.255.255.
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
# 255
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
(?![:.\w])
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
(?:(?:[A-F0-9]{1,4}:){6}|(?=(?:[A-F0-9]{0,4}:){0,6}(?:[0-9]{1,3}\.){3}↲
[0-9]{1,3}(?![:.\w]))(([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:))↲
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↲
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)(?![:.\w])
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Notação padrão, mista ou compacta

Corresponda a um endereço IPv6, utilizando qualquer uma das notações explicadas anteriormente: padrão, mista, compacta e mista compacta.

Verificar se o texto de assunto total compõe um endereço IPv6:

```
\A(?:
# Mista
 (?:
  # Não-compacta
  (?:[A-F0-9]{1,4}:){6}
  # Compacta com, no máximo, 6 sinais de dois pontos
```

```
 |(?=(?:[A-F0-9]{0,4}:){0,6}
     (?:[0-9]{1,3}\.){3}[0-9]{1,3} # e 4 bytes
     \Z) # e ancorada
  # e com, no máximo, um sinal duplo de dois pontos
  (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)
 )
 # 255.255.255.
 (?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
 # 255
 (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
|# Padrão
 (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}
|# Compacta com, no máximo, 7 sinais de dois pontos
 (?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}
    \Z) # e ancorada
 # e com, no máximo, um sinal duplo de dois pontos
 (([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)
)\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:(?:(?:[A-F0-9]{1,4}:){6}|(?=(?:[A-F0-9]{0,4}:){0,6}(?:[0-9]{1,3}\.){3}↲
[0-9]{1,3}$)(([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:))↲
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}↲
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)|(?:[A-F0-9]{1,4}:){7}↲
[A-F0-9]{1,4}|(?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}$)↲
(([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:))$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Encontrar um endereço IPv6, utilizando notação padrão ou mista dentro de uma coleção de texto maior:

```
(?<![:.\w])(?:
# Mista
 (?:
  # Não-compacta
  (?:[A-F0-9]{1,4}:){6}
  # Compacta com, no máximo, 6 sinais de dois pontos
 |(?=(?:[A-F0-9]{0,4}:){0,6}
     (?:[0-9]{1,3}\.){3}[0-9]{1,3} # e 4 bytes
```

```
    (?![:.\w])) # e ancorada
  # e com, no máximo, um sinal duplo de dois pontos
  (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)
 )
 # 255.255.255.
 (?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
 # 255
 (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
|# Padrão
 (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}
|# Compacta com, no máximo, 7 sinais de dois pontos
 (?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}
    (?![:.\w])) # e ancorada
 # e com, no máximo, um sinal duplo de dois pontos
 (([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)
)(?![:.\w])
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby 1.9

JavaScript e Ruby 1.8 não suportam lookbehind, por isso temos de remover a verificação no início da expressão regular que a impede de encontrar endereços IPv6 dentro de sequências mais longas de dígitos hexadecimais e dois pontos. Não podemos usar uma extremidade de palavra, pois o endereço pode começar com um sinal de dois pontos, que não é caractere de palavra:

```
(?:
 # Mista
 (?:
  # Não-compacta
  (?:[A-F0-9]{1,4}:){6}
  # Compacta com, no máximo, 6 sinais de dois pontos
 |(?=(?:[A-F0-9]{0,4}:){0,6}
    (?:[0-9]{1,3}\.){3}[0-9]{1,3} # e 4 bytes
    (?![:.\w])) # e ancorada
 # e com, no máximo, um sinal duplo de dois pontos
 (([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:)
 )
 # 255.255.255.
 (?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
 # 255
```

```
 (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
|# Padrão
 (?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}
|# Compacta com, no máximo, 7 sinais de dois pontos
 (?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}
    (?![:.\w])) # e ancorada
 # e com, no máximo, um sinal duplo de dois pontos
 (([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:)
)(?![:.\w])
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
(?:(?:(?:[A-F0-9]{1,4}:){6}|(?=(?:[A-F0-9]{0,4}:){0,6}(?:[0-9]{1,3}\.){3}↲
[0-9]{1,3}(?![:.\w]))(([0-9A-F]{1,4}:){0,5}|:)((:[0-9A-F]{1,4}){1,5}:|:))↲
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|↲
[01]?[0-9][0-9]?)|(?:[A-F0-9]{1,4}:){7}[A-F0-9]{1,4}|↲
(?=(?:[A-F0-9]{0,4}:){0,7}[A-F0-9]{0,4}(?![:.\w]))↲
(([0-9A-F]{1,4}:){1,7}|:)((:[0-9A-F]{1,4}){1,7}|:))(?![:.\w])
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Por causa das diferentes notações, corresponder a um endereço IPv6 não é tão simples quanto corresponder a um endereço IPv4. As notações que você pretende aceitar terão grande impacto na complexidade de sua expressão regular. Basicamente, há duas notações: padrão e mista. Você pode optar por permitir apenas uma das duas notações, ou ambas. Isso nos dá três conjuntos de expressões regulares.

Ambas as notações, padrão e mista, possuem um formato compacto que omite zeros. Permitir a notação compacta nos dá mais três conjuntos de expressões regulares.

Você vai precisar de expressões regulares um pouco diferentes, dependendo do que quiser fazer: verificar se uma string é um endereço IPv6 válido, ou encontrar endereços IP em um conjunto

maior de texto. Para validar o endereço IP, usamos âncoras, como a receita 2.5 explica. O JavaScript utiliza as âncoras ‹^› e ‹$›, enquanto o Ruby utiliza ‹\A› e ‹\Z›. Todos os outros sabores suportam ambos os tipos. O Ruby também suporta ‹^› e ‹$›, mas ele permite que essas âncoras também correspondam em quebras de linhas incorporadas na string. Você deverá usar o acento circunflexo e o cifrão, no Ruby, somente se você souber que sua string não possui qualquer quebra de linha incorporada.

Para localizar endereços IPv6 dentro de um texto maior, usamos o lookbehind negativo ‹(?<![:.\w])› e o lookahead negativo ‹(?![:.\w])›, para garantir que o endereço não seja precedido ou seguido por um caractere de palavra (letra, dígito ou underscore), por um ponto ou por um sinal de dois pontos. Isso garante que não corresponderemos a partes de sequências mais longas de dígitos e dois pontos. A receita 2.16 explica como o lookbehind e o lookahead trabalham. Se o lookaround não estiver disponível, as extremidades de palavra podem verificar se o endereço não é precedido, ou seguido, por um caractere de palavra, mas apenas se você tiver certeza de que o primeiro e o último caractere no endereço são dígitos (hexadecimais). A notação compacta permite endereços que comecem e terminem com um sinal de dois pontos. Se tivéssemos de colocar uma extremidade de palavra antes, ou depois, do sinal de dois pontos, seria necessário uma letra ou dígito adjacente, algo que não queremos. A receita 2.6 explica tudo sobre extremidades de palavras.

Notação padrão

A notação IPv6 padrão é bastante simples de ser tratada com uma expressão regular. Precisamos corresponder a oito palavras em notação hexadecimal, separadas por sete sinais de dois pontos. ‹[A-F0-9]{1,4}› corresponde de 1 a 4 caracteres hexadecimais, o que precisamos para uma palavra de 16 bits com zeros à esquerda opcionais. A classe de caracteres (receita 2.3) lista apenas as letras maiúsculas. O modo de correspondência de não-diferenciação entre

maiúsculas e minúsculas cuida das letras minúsculas. Veja a receita 3.4, para aprender como definir modos de correspondência em sua linguagem de programação.

O grupo de não-captura ‹(?:[A-F0-9]{1,4}:){7}› corresponde a uma palavra hexadecimal, seguida por um sinal de dois pontos literal. O quantificador repete o grupo sete vezes. O primeiro sinal de dois pontos, nesta expressão regular, faz parte da sintaxe para grupos de não-captura, como a receita 2.9 explica; o segundo representa um sinal de dois pontos literal. O sinal de dois pontos não é um metacaractere, nas expressões regulares, salvo em algumas situações muito específicas, como quando faz parte de um token de expressão regular maior. Portanto, não precisamos usar barras invertidas literais para escapar os sinais de dois pontos em nossa expressão regular. Poderíamos escapá-los, mas isso só tornaria a expressão regular mais difícil de ler.

Notação mista

A expressão regular para a notação IPv6 mista consiste em duas partes. ‹(?:[A-F0-9]{1,4}:){6}› corresponde a seis palavras hexadecimais, cada uma seguida por um sinal de dois pontos literal, assim como temos uma sequência de sete palavras, na expressão regular, para a notação IPv6 padrão.

Ao invés de ter mais uma palavra hexadecimal no final, temos, agora, um endereço IPv4 completo no final. Nós correspondemos a isso usando a expressão regular “precisa”, mostrada na receita 7.16.

Notação padrão ou mista

Permitir ambas as notações, padrão e mista, exige uma expressão regular um pouco mais longa. As duas notações diferem apenas na representação dos últimos 32 bits do endereço IPv6. A notação padrão usa duas palavras de 16 bits, enquanto a notação mista usa 4 bytes decimais, como acontece no IPv4.

A primeira parte da expressão regular corresponde a seis palavras

hexadecimais, como na expressão regular que suporta somente notação mista. A segunda parte consiste em um grupo de não-captura, com duas alternativas para os últimos 32 bits. Como a receita 2.8 explica, o operador de alternância (barra vertical) possui a prioridade mais baixa dentre todos os operadores de expressão regular. Assim, precisamos do grupo de não-captura para excluir as seis palavras da alternância.

A primeira alternativa, localizada à esquerda da barra vertical, corresponde a duas palavras hexadecimais com um sinal de dois pontos literal entre elas. A segunda alternativa corresponde a um endereço IPv4.

Notação compacta

As coisas ficam um pouco mais complicadas quando permitimos a notação compacta. A razão disso é que a notação compacta permite omitir uma quantidade variável de zeros. `1:0:0:0:0:6:0:0`, `1::6:0:0` e `1:0:0:0:0:6::` são três maneiras de escrever o mesmo endereço IPv6. O endereço deve ter, no máximo, oito palavras, mas não precisa ter nenhuma. Com menos de oito palavras, ele deverá ter uma sequência de dois sinais de dois pontos, representando os zeros omitidos.

Repetição variável é fácil de fazer em expressões regulares. Se um endereço IPv6 tiver um sinal duplo de dois pontos, poderá haver, no máximo, sete palavras antes e depois do sinal duplo. Poderíamos facilmente escrever isso assim:

```
(
  ([0-9A-F]{1,4}:){1,7} # entre 1 e 7 palavras à esquerda
| : # ou um sinal duplo de dois pontos no início
)
(
  (:[0-9A-F]{1,4}){1,7} # entre 1 e 7 palavras à direita
| : # ou um sinal duplo de dois pontos no final
)
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Esta expressão regular, e as que seguem nesta discussão, também funcionam no JavaScript, caso você elimine os comentários e os espaços em branco extras. O JavaScript suporta todos os recursos utilizados nestas expressões regulares, exceto o espaçamento livre, que usamos, aqui, para tornar as expressões regulares fáceis de entender.

Essa expressão regular corresponde a todos os endereços IPv6 compactos, mas não corresponde a nenhum endereço que utilize a notação padrão não-compacta.

Essa expressão regular é bastante simples. A primeira parte corresponde de 1 a 7 palavras, seguidas por um sinal de dois pontos, ou apenas a um sinal de dois pontos, no caso dos endereços que não tenham palavra à esquerda do sinal duplo de dois pontos.

A segunda parte corresponde de 1 a 7 palavras, precedidas por um sinal de dois pontos, ou apenas a um sinal de dois pontos, no caso de endereços que não tenham nenhuma palavra à direita do sinal duplo de dois pontos. Reunidas, as correspondências válidas são: o sinal duplo de dois pontos por si só, o sinal duplo de dois pontos com 1 a 7 palavras à esquerda, apenas, o sinal duplo de dois pontos duplos com 1 a 7 palavras à direita apenas, e o sinal duplo de dois pontos com 1 a 7 palavras em ambos os lados.

Esta última parte é problemática. A expressão regular permite de 1 a 7 palavras, tanto à esquerda como à direita, como deveria, mas não especifica que a soma total de palavras, à esquerda e à direita, deve ser 7 ou menos. Um endereço IPv6 possui 8 palavras. O sinal duplo de dois pontos indica que estamos omitindo pelo menos uma palavra; então, no máximo 7 permanecem.

Expressões regulares não fazem cálculos. Elas podem contar se algo ocorre entre 1 e 7 vezes. Mas elas não podem contar se duas coisas ocorrem um total de 7 vezes, dividindo estas 7 vezes entre as duas coisas, em qualquer combinação.

Para entender melhor este problema, vamos fazer uma analogia simples. Digamos que você deseja corresponder a algo no formato

aaaaxbbb. A string deverá ter entre 1 e 8 caracteres de comprimento e ter de 0 a 7 vezes o caractere a, exatamente um x e de 0 a 7 vezes o b.

Há duas maneiras de resolver este problema com uma expressão regular. A primeira seria soletrar todas as alternativas. A próxima seção, que discute a notação mista compacta, emprega essa via. Isso pode resultar em uma expressão regular bem grande, mas fácil de entender.

```
\A(?:a{7}x
 | a{6}xb?
 | a{5}xb{0,2}
 | a{4}xb{0,3}
 | a{3}xb{0,4}
 | a{2}xb{0,5}
 | axb{0,6}
 | xb{0,7}
)\Z
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Esta expressão regular possui uma alternativa para cada uma das quantidades possíveis de letras a. Cada alternativa soletra o tanto de letras b permitidas, depois que uma determinada quantidade de letras a e o x foram correspondidos. A outra solução seria usar um lookahead. Este é o método utilizado pela expressão regular dentro da seção “Solução”, que corresponde a um endereço IPv6 usando notação compacta. Se não estiver familiarizado com lookaheads, leia a receita 2.16 primeiro. Usando um lookahead, podemos, essencialmente, corresponder ao mesmo texto duas vezes, verificando-o em duas condições.

```
\A
  (?=[abx]{1,8}\Z)
  a{0,7}xb{0,7}
\Z
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

O ‹\A›, no início da expressão regular, ancora o início do texto de assunto. Então, o lookahead positivo entra em jogo. Ele verifica se uma série de 1 a 8 letras – ‹a›, ‹b› ou ‹x› – podem ser correspondidas, e se o final da string é alcançado quando estas letras são correspondidas. O ‹\Z› dentro do lookahead é fundamental. A fim de limitar a expressão regular a strings de oito caracteres ou menos, o lookahead deve verificar se não há mais caracteres após os que foram correspondidos.

Em um cenário diferente, você poderia usar outro tipo de delimitador, em vez de ‹\A› e ‹\Z›. Se quisesse fazer uma pesquisa “somente palavras completas” para aaaaxbbb e amigos, você usaria extremidades de palavras. Porém, para restringir a correspondência da expressão regular ao comprimento certo, você teria de usar algum tipo de delimitador, além de colocar o delimitador que corresponde ao final da string, tanto dentro do lookahead quanto no final da expressão regular. Se você não fizer isso, a expressão regular corresponderá, parcialmente, a uma string que tenha caracteres demais.

Depois que o lookahead satisfaz seus requisitos, ele desiste dos caracteres aos quais correspondeu. Assim, no momento em que o mecanismo de expressão regular tenta ‹a{0,7}›, ele está de volta ao início da string. O fato de o lookahead não consumir o texto correspondido por ele é a diferença fundamental entre um lookahead e um grupo de não-captura, e é o que nos permite aplicar dois padrões a um único fragmento de texto.

Embora ‹a{0,7}xb{0,7}›, por conta própria, possa corresponder a até 15 letras, neste caso ela pode corresponder a apenas 8, porque o lookahead já se certificou de que há apenas 8 letras. Tudo que ‹a{0,7}xb{0,7}› tem a fazer é verificar se elas aparecem na ordem certa. Na verdade, ‹a*xb*› teria exatamente o mesmo efeito de ‹a{0,7}xb{0,7}›, nesta expressão regular.

O segundo ‹\Z›, no final da expressão regular, também é essencial. Assim como o lookahead precisa ter certeza de que não existam

muitas letras, o segundo teste após o lookahead precisa se certificar de que todas as letras estejam na ordem correta. Isso garante que não corresponderemos a algo como axba, mesmo que isso satisfaça o lookahead, por estar entre 1 e 8 caracteres de comprimento.

Notação mista compacta

A notação mista pode ser compacta, tal como a notação padrão. Embora os quatro bytes no final sempre devam ser especificados, mesmo quando forem zero, a quantidade de palavras hexadecimais antes deles, novamente, torna-se variável. Se todas as palavras hexadecimais forem zero, o endereço IPv6 pode acabar parecendo um endereço IPv4, com um sinal duplo de dois pontos antes dele.

Criar uma expressão regular para a notação mista compacta envolve resolver as mesmas questões da notação padrão compacta. A seção anterior explica tudo isso.

A principal diferença entre a expressão regular para notação mista compacta e a expressão regular feita para notação padrão compacta é que a primeira precisa verificar o endereço IPv4 após as seis palavras hexadecimais. Fazemos essa verificação no final da expressão regular, utilizando a mesma expressão regular para endereços IPv4 precisos, da receita 7.16, que usamos nesta receita para a notação mista não-compacta.

Temos de corresponder à parte do endereço IPv4 no final da expressão regular, mas também temos de verificá-la dentro do lookahead, que garante não termos mais de seis sinais de dois pontos, ou seis palavras hexadecimais no endereço IPv6. Como estamos fazendo um teste preciso no final da expressão regular, o lookahead se satisfaz com uma verificação simples do IPv4. O lookahead não precisa validar a parte IPv4, pois a expressão regular principal já faz isso. Porém, ela precisa corresponder à parte IPv4, de modo que a âncora de final-de-string, no final do lookahead, possa fazer seu trabalho.

Notação padrão, mista ou compacta

O conjunto final de expressões regulares junta todas as notações. Elas correspondem a um endereço IPv6 em qualquer notação: padrão ou mista, compacta ou não.

Estas expressões regulares são formadas alternando notação mista compacta e notação padrão compacta. Estas expressões regulares já utilizam a alternância, para corresponder às variedades compacta e não-compacta da notação IPv6 suportada por elas.

O resultado é uma expressão regular com três alternativas no nível mais alto, com a primeira consistindo em duas alternativas próprias. A primeira alternativa corresponde a um endereço IPv6 usando a notação mista, compacta ou não-compacta. A segunda corresponde a um endereço IPv6, usando a notação padrão. A terceira abrange a notação padrão compacta.

Temos três alternativas no nível mais alto, em vez de duas alternativas contendo, cada uma, suas próprias duas alternativas, pois não há nenhuma razão especial para agrupar alternativas nas notações padrão e compacta. No caso da notação mista, nós mantivemos juntas as alternativas compacta e não-compacta, porque isso nos salva de precisar soletrar duas vezes a parte IPv4.

Essencialmente, combinamos esta expressão regular:

```
^(6words|compressed6words)ip4$
```

e esta expressão regular:

```
^(8words|compressed8words)$
```

em:

```
^((6words|compressed6words)ip4|8words|compressed8words)$
```

em vez de:

```
^((6words|compressed6words)ip4|(8words|compressed8words))$
```

Veja também:

Receitas 2.16 e 7.16.

7.18 Validar caminhos do Windows

Problema

Você deseja verificar se uma string se parece com um caminho válido para uma pasta ou arquivo no sistema operacional Microsoft Windows.

Solução

Caminhos com letra de unidade

```
\A
[a-z]:\\ # Unidade
(?:[^\\/:*?"<>|\r\n]+\\)* # Pasta
[^\\/:*?"<>|\r\n]* # Arquivo
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^[a-z]:\\(?:[^\\/:*?"<>|\r\n]+\\)*[^\\/:*?"<>|\r\n]*$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Caminhos UNC e com letra de unidade

```
\A
(?:[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\ # Unidade
(?:[^\\/:*?"<>|\r\n]+\\)* # Pasta
[^\\/:*?"<>|\r\n]* # Arquivo
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\(?:[^\\/:*?"<>|\r\n]+\\)*↲
[^\\/:*?"<>|\r\n]*$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Caminhos relativos, UNC e com letra de unidade

```
\A
```

```
(?:(?:[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\| # Unidade
  \\?[^\\/:*?"<>|\r\n]+\\?) # Caminho relativo
(?:[^\\/:*?"<>|\r\n]+\\)* # Pasta
[^\\/:*?"<>|\r\n]* # Arquivo
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:(?:[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\|\\?[^\\/:*?"<>|\r\n]+\\?)↵
(?:[^\\/:*?"<>|\r\n]+\\)*[^\\/:*?"<>|\r\n]*$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Discussão

Caminhos com letra de unidade

Corresponder a um caminho completo para um arquivo ou pasta, em uma unidade que tenha uma letra, é muito simples. A unidade é indicada por uma letra simples, seguida por um sinal de dois pontos e uma barra invertida. Correspondemos a isso facilmente com ‹[a-z]:\\›. A barra invertida é um metacaractere em expressões regulares e, por isso, precisamos escapá-la com outra barra invertida, para fazer a correspondência literalmente.

Pastas e nomes de arquivos, no Windows, podem conter todos os caracteres, exceto estes: \/:*?"<>|. Quebras de linha também não são permitidas. Podemos corresponder facilmente a uma sequência de todos os caracteres, exceto esses, com a classe de caracteres negada ‹[^\\/:*?"<>|\r\n]+›. A barra invertida também é um metacaractere em classes de caracteres, por isso a escapamos. ‹\r› e ‹\n› são os dois caracteres de quebra de linha. Veja a receita 2.3, para saber mais sobre classes de caracteres (negadas). O quantificador + (receita 2.12) especifica que queremos um ou mais caracteres.

As pastas são delimitadas por barras invertidas. Podemos corresponder a uma sequência de zero ou mais pastas com ‹(?:

[^\\/:*?"<>|\r\n]+\\)*›, que coloca a expressão regular para o nome da pasta e uma barra invertida literal dentro de um grupo de não-captura (receita 2.9), repetido zero ou mais vezes com o asterisco (receita 2.12).

Para corresponder ao nome do arquivo, usamos ‹[^\\/:*?"<>|\r\n]*›. O asterisco torna o nome do arquivo opcional, para permitir caminhos que terminem com uma barra invertida. Se você não quer permitir esses caminhos, altere o último ‹*› na expressão regular para ‹+›.

Caminhos UNC e com letra de unidade

Caminhos para arquivos em unidades de rede que não estejam mapeados como letras de unidades podem ser acessados se utilizarmos caminhos da Convenção de Nomeação Universal (UNC). Caminhos UNC possuem o formato *servidor**compartilhamento**pasta**arquivo*.

Podemos adaptar, facilmente, a expressão regular que trata caminhos com letra de unidade, de forma a suportar caminhos UNC. Tudo o que temos a fazer é substituir a parte ‹[a-z]:›, que corresponde à letra de unidade, por algo que corresponda a uma letra de unidade ou a um nome de servidor.

‹(?:[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)› faz isso. A barra vertical é o operador de alternância (receita 2.8). Ele nos permite a escolha entre uma letra de unidade correspondida com ‹[a-z]:› ou um nome de servidor, mais um nome compartilhado, correspondidos com ‹\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+›. O operador de alternância tem a prioridade mais baixa dentre todos os operadores de expressão regular. Para agrupar as duas alternativas, podemos utilizar um grupo de não-captura. Como a receita 2.9 explica, os caracteres ‹(?:› formam o complicado delimitador de abertura de um grupo de não-captura. O ponto de interrogação não tem seu sentido habitual após um parêntese.

O resto da expressão regular pode permanecer igual. O nome do compartilhamento, em caminhos UNC, será correspondido pela

parte da expressão regular que corresponde a nomes de pastas.

Caminhos relativos, UNC e com letra de unidade

Um caminho relativo é aquele que começa com um nome de pasta (talvez a pasta especial .., para selecionar a pasta-mãe) ou que consiste apenas em um nome de arquivo. Para suportar caminhos relativos, podemos acrescentar uma terceira alternativa para a porção “unidade” em nossa expressão regular. Essa alternativa corresponderia ao início de um caminho relativo, em vez de uma letra de unidade ou nome de servidor.

‹\\?[^\\/:*?"<>|\r\n]+\\?› corresponde ao início do caminho relativo. O caminho pode começar com uma barra invertida, mas isso não é obrigatório. ‹\\?› corresponde à barra invertida, se estiver presente, ou a nada. ‹[^\\/:*?"<>|\r\n]+› corresponde a uma pasta, ou a um nome de arquivo. Se o caminho relativo consistir apenas em um nome de arquivo, o ‹\\?› final não corresponderá a nada, muito menos às partes “pasta” e “arquivo” da expressão regular, ambas opcionais. Se o caminho relativo especificar uma pasta, o ‹\\?› final corresponderá à barra invertida, que separa a primeira pasta, no caminho relativo, do resto do caminho. A parte “pasta”, então, corresponderá às pastas restantes, se houver, e a parte “arquivo” corresponderá ao nome do arquivo.

A expressão regular para corresponder a caminhos relativos já não usa partes claramente distintas da expressão regular para corresponder a partes distintas do texto de assunto. A parte da expressão regular rotulada como “caminho relativo” realmente corresponderá a uma pasta, ou a um nome de arquivo, se o caminho for relativo. Se o caminho relativo especificar uma ou mais pastas, a parte “caminho relativo” corresponderá à primeira pasta, e as partes “pasta” e “arquivo” corresponderão ao que sobrou. Se o caminho relativo for apenas um nome de arquivo, ele será correspondido pela parte “caminho relativo”, não deixando nada para as partes “pasta” e “arquivo”. Como estamos interessados em validar o caminho, isso não importa. Os comentários na expressão

regular são apenas rótulos que nos ajudam a entendê-la.

Se quiséssemos extrair partes do caminho em grupos de captura, precisaríamos ter mais cuidado para corresponder à unidade, à pasta e ao nome do arquivo separadamente. A próxima receita lida com esse problema.

Veja também:

Receitas 2.3, 2.8, 2.9 e 2.12.

7.19 Dividir caminhos do Windows em suas partes constituintes

Problema

Você deseja verificar se uma string aparenta ser um caminho válido para uma pasta ou para um arquivo, no sistema operacional Microsoft Windows. Se a string realmente possuir um caminho válido no Windows, você também desejará extrair separadamente as partes do caminho relativas à unidade, à pasta e ao nome do arquivo.

Solução

Caminhos com letra de unidade

```
\A
(?<drive>[a-z]:)\\
(?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?<file>[^\\/:*?"<>|\r\n]*)
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\A
(?P<drive>[a-z]:)\\
(?P<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?P<file>[^\\/:*?"<>|\r\n]*)
```

```
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: PCRE 4 e superior, Perl 5.10, Python

```
\A
([a-z]:)\\
((?:[^\\/:*?"<>|\r\n]+\\)*)
([^\\/:*?"<>|\r\n]*)
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^([a-z]:)\\((?:[^\\/:*?"<>|\r\n]+\\)*)([^\\/:*?"<>|\r\n]*)$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Caminhos UNC e com letra de unidade

```
\A
(?<drive>[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\
(?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?<file>[^\\/:*?"<>|\r\n]*)
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\A
(?P<drive>[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\
(?P<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?P<file>[^\\/:*?"<>|\r\n]*)
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: PCRE 4 e superior, Perl 5.10, Python

```
\A
([a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\
((?:[^\\/:*?"<>|\r\n]+\\)*)
([^\\/:*?"<>|\r\n]*)
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^([a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\((?:[^\\/:*?"<>|\r\n]+\\)*)↲
([^\\/:*?"<>|\r\n]*)$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Caminhos relativos, UNC e com letra de unidade

Estas expressões regulares podem corresponder à string vazia. Veja a discussão para mais detalhes e uma solução alternativa.

```
\A
(?<drive>[a-z]:\\|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+\\|\\?)
(?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?<file>[^\\/:*?"<>|\r\n]*)
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\A
(?P<drive>[a-z]:\\|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+\\|\\?)
(?P<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
(?P<file>[^\\/:*?"<>|\r\n]*)
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: PCRE 4 e superior, Perl 5.10, Python

```
\A
([a-z]:\\|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+\\|\\?)
((?:[^\\/:*?"<>|\r\n]+\\)*)
([^\\/:*?"<>|\r\n]*)
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^([a-z]:\\|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+\\|\\?)↲
((?:[^\\/:*?"<>|\r\n]+\\)*)([^\\/:*?"<>|\r\n]*)$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

Discussão

As expressões regulares, nesta receita, são muito similares às da receita anterior. Esta discussão assume que você já leu e compreendeu a discussão da receita anterior.

Caminhos com letra de unidade

Fizemos apenas uma mudança, nas expressões regulares, para os caminhos com letra de unidade, em comparação com as expressões da receita anterior. Adicionamos três grupos de captura, que você pode usar para recuperar as várias partes do caminho: ‹drive›, ‹folder› e ‹file›. Você pode usar estes nomes se o seu sabor de expressão regular suportar captura nomeada (receita 2.11). Se não, terá de fazer referência a grupos de captura pelos números: 1, 2 e 3. Veja a receita 3.9, para saber como obter o texto correspondido pelos grupos nomeados/numerados em sua linguagem de programação favorita.

Caminhos UNC e com letra de unidade

Adicionamos os mesmos três grupos de captura para as expressões regulares de caminhos UNC.

Caminhos relativos, UNC e letra de unidade

As coisas ficam um pouco mais complicadas se quisermos, também, permitir caminhos relativos. Na receita anterior, poderíamos apenas acrescentar uma terceira alternativa para a parte da unidade, na expressão regular, para corresponder ao início do caminho relativo. Nós não podemos fazer isso aqui. No caso de um caminho relativo, o grupo de captura da unidade deve permanecer vazio.

Em vez disso, a barra invertida literal que ficava depois do grupo de captura da unidade, na expressão regular da seção “Caminhos UNC e com letra de unidade”, mudou-se para dentro daquele grupo de

captura. Nós o adicionamos ao final das alternativas relativas à letra de unidade e ao compartilhamento de rede. Nós adicionamos uma terceira alternativa, com uma barra invertida opcional, para caminhos relativos que possam ou não começar com uma barra invertida. Como a terceira alternativa é opcional, todo o grupo relativo à unidade é essencialmente opcional.

A expressão regular resultante corresponde, corretamente, a todos os caminhos do Windows. O problema é que, tornando a parte da unidade opcional, agora temos uma expressão regular em que tudo é opcional. As partes da pasta e do arquivo já eram opcionais nas expressões regulares que suportavam apenas caminhos absolutos. Em outras palavras: a nossa expressão regular corresponderá à string vazia.

Se quisermos nos certificar de que a expressão regular não corresponderá a strings vazias, será preciso acrescentar outras alternativas, para lidar com caminhos relativos que especifiquem uma pasta (caso em que o nome do arquivo é opcional), e com caminhos relativos que não especifiquem uma pasta (caso em que o nome do arquivo é obrigatório):

```
\A
(?:
  (?<drive>[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\
  (?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
  (?<file>[^\\/:*?"<>|\r\n]*)
| (?<relativefolder>\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
  (?<file2>[^\\/:*?"<>|\r\n]*)
| (?<relativefile>[^\\/:*?"<>|\r\n]+)
)
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, PCRE 7, Perl 5.10, Ruby 1.9

```
\A
(?:
  (?P<drive>[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\
  (?P<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
```

```
  (?P<file>[^\\/:*?"<>|\r\n]*)
| (?P<relativefolder>\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
  (?P<file2>[^\\/:*?"<>|\r\n]*)
| (?P<relativefile>[^\\/:*?"<>|\r\n]+)
)
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: PCRE 4 e superior, Perl 5.10, Python

```
\A
(?:
  ([a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\
  ((?:[^\\/:*?"<>|\r\n]+\\)*)
  ([^\\/:*?"<>|\r\n]*)
| (\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
  ([^\\/:*?"<>|\r\n]*)
| ([^\\/:*?"<>|\r\n]+)
)
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

```
^(?:([a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\((?:[^\\/:*?"<>|\r\n]+\\)*)↲
([^\\/:*?"<>|\r\n]*)|(\\?(?:[^\\/:*?"<>|\r\n]+\\)+)([^\\/:*?"<>|\r\n]*)|↲
([^\\/:*?"<>|\r\n]+))$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python

O preço que pagamos para excluir strings de comprimento zero é que, agora, temos seis grupos de captura para capturar as três diferentes partes do caminho. Você terá de analisar o cenário em que deseja usar estas expressões regulares, para determinar se é mais fácil fazer uma verificação extra para strings vazias, antes de usar a expressão regular, ou concentrar seus esforços para lidar com múltiplos grupos de captura, depois que uma correspondência foi encontrada.

Se estiver usando o sabor de expressão regular .NET, você poderá dar o mesmo nome a vários grupos nomeados. O sabor .NET é o

único que trata grupos homônimos como se fossem um único grupo de captura. Com esta expressão regular .NET, você pode simplesmente pegar a correspondência do grupo da pasta ou do arquivo, sem se preocupar com qual dos dois grupos de pasta, ou dos três grupos de arquivo, realmente participaram da correspondência da expressão regular:

```
\A
  (?:
  (?<drive>[a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)\\
  (?<folder>(?:[^\\/:*?"<>|\r\n]+\\)*)
  (?<file>[^\\/:*?"<>|\r\n]*)
| (?<folder>\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
  (?<file>[^\\/:*?"<>|\r\n]*)
| (?<file>[^\\/:*?"<>|\r\n]+)
)
\Z
```

Opções Regex: Espaçamento livre, não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET

Veja também:

Receitas 2.9, 2.11, 3.9 e 7.18.

7.20 Extrair a letra da unidade de um caminho Windows

Problema

Você possui uma string que contém um caminho (sintaticamente) válido para um arquivo ou pasta em um PC com Windows local ou em rede. Você deseja extrair a letra de unidade, caso haja, de dentro do caminho. Por exemplo, você deseja extrair c de c:\folder\file.ext.

Solução

```
^([a-z]):
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Extrair a letra de unidade de uma string que contenha um caminho válido é algo trivial, mesmo que você não saiba se o caminho realmente começa com uma letra de unidade. O caminho poderia ser relativo ou UNC.

O sinal de dois pontos é um caractere inválido em caminhos do Windows, exceto para delimitar a letra da unidade. Assim, se tivermos uma letra seguida de dois pontos no início da string, sabemos que a letra é a letra da unidade.

A âncora ‹^› corresponde no início da string (receita 2.5). Não importa o fato de o acento circunflexo também corresponder às quebras de linha incorporadas no Ruby, porque caminhos válidos do Windows não incluem quebras de linha. A classe de caracteres ‹[a-z]› corresponde a uma única letra (receita 2.3). Nós colocamos a classe de caracteres entre um par de parênteses (que formam um grupo de captura), para que você possa obter a letra da unidade sem os dois pontos, também correspondidos pela expressão regular. Nós adicionamos o sinal de dois pontos à expressão regular, para nos certificarmos de que extrairemos a letra da unidade, ao invés de a primeira letra em um caminho relativo.

Veja também:

A receita 2.9 fala tudo sobre grupos de captura.

Veja a receita 3.9, para aprender a recuperar texto correspondido por grupos de captura em sua linguagem de programação favorita.

Siga a receita 7.19, caso não saiba de antemão se sua string contém um caminho válido do Windows.

7.21 Extrair o servidor e o

compartilhamento de um caminho UNC

Problema

Você possui uma string que contém um caminho (sintaticamente) válido para um arquivo ou pasta, em um PC com Windows local ou em rede. Se o caminho for UNC, você deseja extrair o nome do servidor de rede e o compartilhamento, no servidor, para o qual o caminho aponta. Por exemplo, você deseja extrair server e share, de \\server\share\folder\file.ext.

Solução

```
^\\\\([a-z0-9_.$]+)\\([a-z0-9_.$]+)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Extrair o servidor de rede e o compartilhamento de uma string que contenha um caminho válido é fácil, mesmo que você não saiba se o caminho é UNC. O caminho pode ser relativo, ou usar uma letra de unidade.

Caminhos UNC começam com duas barras invertidas. Duas barras invertidas consecutivas não são permitidas em caminhos do Windows, exceto para iniciar um caminho UNC. Assim, se um caminho válido conhecido começar com duas barras invertidas, sabemos que o servidor e o nome do compartilhamento devem segui-las.

A âncora ‹^› corresponde no início da sequência (receita 2.5). O fato de que o acento circunflexo também corresponde às quebras de linha incorporadas no Ruby não importa, pois caminhos válidos do Windows não incluem quebras de linha. ‹\\\\› corresponde a duas barras literais invertidas. Como a barra invertida é um metacaractere em expressões regulares, temos de escapar uma barra invertida

com outra barra invertida, se quisermos fazer a correspondência com um caractere literal. A primeira classe de caracteres, ‹[a-z0-9_.$]+›, corresponde ao nome do servidor de rede. A segunda, após outra barra literal invertida, corresponde ao nome do compartilhamento. Colocamos ambas as classes de caracteres entre um par de parênteses, formando um grupo de captura. Dessa forma, você pode obter o nome do servidor isolado no primeiro grupo de captura, e o nome do compartilhamento isolado no segundo grupo de captura. A correspondência global da expressão regular será \\server\share.

Veja também:

A receita 2.9 fala tudo sobre grupos de captura.

Veja a receita 3.9, para aprender a recuperar texto correspondido por grupos de captura em sua linguagem de programação favorita.

Siga a receita 7.19, caso você não saiba de antemão se a string contém um caminho válido do Windows.

7.22 Extrair a pasta de um caminho Windows

Problema

Você possui uma string que contém um caminho (sintaticamente) válido para um arquivo ou pasta, em um PC com Windows local ou em rede, e deseja extrair a pasta do caminho. Por exemplo, você deseja extrair \folder\subfolder\ de c:\folder\subfolder\file.ext ou de \\server\share\folder\subfolder\file.ext.

Solução

```
^([a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)?((?:\\|^)↲
(?:[^\\/:*?"<>|\r\n]+\\)+)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Extrair a pasta de um caminho do Windows é algo um pouco complicado se quisermos suportar caminhos UNC, pois não podemos simplesmente pegar a parte do caminho entre barras invertidas. Se fizéssemos isso, acabaríamos pegando, também, o servidor e o compartilhamento de caminhos UNC.

A primeira parte da expressão regular, ‹^([a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)?›, pula a letra de unidade, ou os nomes do servidor de rede e do compartilhamento de rede, no início do caminho. Essa parte da expressão regular é constituída por um grupo de captura com duas alternativas. A primeira alternativa corresponde à letra de unidade, como na receita 7.20, e a segunda corresponde ao servidor e ao compartilhamento em caminhos UNC, como na receita 7.21. A receita 2.8 explica o operador de alternância.

O ponto de interrogação, depois do grupo, o torna opcional. Isso nos permite suportar caminhos relativos, que não possuam letra de unidade ou compartilhamento de rede.

As pastas são facilmente correspondidas com ‹(?:[^\\/:*?"<>|\r\n]+\\)+›. A classe de caracteres corresponde a um nome de pasta. O grupo de não-captura corresponde a um nome de pasta, seguido por uma barra invertida literal, que tem como função separar as pastas, umas das outras e do nome do arquivo. Repetimos este grupo uma ou mais vezes. Isso significa que nossa expressão regular corresponderá somente aos caminhos que realmente especifiquem uma pasta. Caminhos que especifiquem apenas um nome de arquivo, unidade ou compartilhamento de rede não serão correspondidos.

Se o caminho começa com uma letra de unidade, ou com um compartilhamento de rede, ele deverá ser seguido por uma barra invertida. Um caminho relativo pode, ou não, começar com uma barra invertida. Assim, precisamos adicionar uma barra invertida opcional, no início do grupo que corresponde à parte da pasta do caminho. Como iremos somente usar a nossa expressão regular em

caminhos sabidamente válidos, não precisamos ser rigorosos, e exigirmos a barra invertida, no caso de uma letra de unidade ou compartilhamento de rede. Temos apenas que permitir tal ocorrência.

Como exigimos que a expressão regular corresponda a pelo menos uma pasta, devemos ter certeza de que nossa expressão regular não corresponderá a e\ como se fosse a pasta em \\server\share\. É por isso que usamos ‹(\\|^)›, em vez de ‹\\?›, para adicionarmos a barra invertida opcional no início do grupo de captura da pasta.

Se você estiver se perguntando por que \\server\shar poderia ser correspondido como a unidade, e e\ como a pasta, revise a receita 2.13. Mecanismos de expressão regular retrocedem. Suponha esta expressão regular:

```
^([a-z]:|\\\\[a-z0-9_.$]+\\[a-z0-9_.$]+)?↵
((?:\\?(?:[^\\/:*?"<>|\r\n]+\\)+)
```

Tal expressão regular, assim como a da solução, exige pelo menos um caractere que não seja barra invertida, além de uma barra invertida para o caminho. Se a expressão regular corresponde a \\server\share como a unidade em \\server\share e, depois, não consegue corresponder ao grupo da pasta, ela não desistirá; tentará diferentes variantes da expressão regular.

Neste caso, o mecanismo lembrou que a classe de caracteres ‹[a-z0-9_.$]+›, que corresponde ao compartilhamento da rede, não precisa corresponder a todos os caracteres disponíveis. Um caractere é o suficiente para satisfazer o ‹+›. O mecanismo retrocede, forçando a classe de caracteres a desistir de um dado caractere, e então tenta continuar.

Ao continuar, o mecanismo encontra dois caracteres remanescentes, na string de assunto, para corresponder à pasta: e\. Estes dois caracteres são o suficiente para satisfazer ‹(?:[^\\/:*?"<>|\r\n]+\\)+›; assim, temos uma correspondência total para a expressão regular. Porém, não é a correspondência que queríamos.

Usar ‹(\\|^)›, em vez de ‹\\?›, resolve isso. Tal recurso ainda permite

uma barra invertida opcional, mas quando a barra invertida estiver faltando, ele exigirá que a pasta comece no início da string. Isso significa que, se uma unidade tiver sido correspondida e, portanto, o mecanismo de expressão regular tiver avançado além do início da string, a barra invertida é necessária. O mecanismo de expressão regular ainda tentará retroceder, se não puder corresponder a nenhuma pasta, mas vai fazê-lo em vão, porque ‹(\\|^)› falhará na correspondência. O mecanismo de expressão regular recuará, até que ele retorne ao início da string. O grupo de captura da letra de unidade e do compartilhamento de rede é opcional; assim, o mecanismo de expressão regular será bem-vindo para tentar corresponder à pasta no início da string. Embora ‹(\\|^)› corresponda neste caso, o mesmo não acontecerá com o resto da expressão regular, porque ‹(?:[^\\/:*?"<>|\r\n]+\\)+› não permite o sinal de dois pontos seguinte à letra de unidade, ou a barra dupla invertida do compartilhamento de rede.

Se estiver se perguntando por que esta técnica não foi usada nas receitas 7.18 e 7.19, é porque aquelas expressões regulares não necessitam de uma pasta. Como tudo é opcional, após a parte que corresponde à unidade nessas expressões regulares, o mecanismo de expressão regular nunca fará qualquer retrocesso. Claro, tornar as coisas opcionais pode levar a problemas diferentes, como discutido na receita 7.19.

Quando esta expressão regular encontra uma correspondência, o primeiro grupo de captura conterá a letra de unidade ou o compartilhamento de rede; o segundo grupo de captura conterá a pasta. O primeiro grupo estará vazio, no caso de um caminho relativo. O segundo grupo sempre conterá pelo menos uma pasta. Se você usar essa expressão regular em um caminho que não especifique uma pasta, ela não encontrará uma correspondência.

Veja também:

A receita 2.9 fala tudo sobre grupos de captura.

Veja a receita 3.9, para aprender como recuperar texto correspondido por grupos de captura em sua linguagem de programação favorita.

Siga a receita7.19, caso não saiba de antemão se sua string contém um caminho válido do Windows.

7.23 Extrair o nome do arquivo de um caminho do Windows

Problema

Você possui uma string que contém um caminho (sintaticamente) válido para um arquivo ou pasta, em um PC com Windows local ou em rede, e deseja extrair o nome do arquivo, caso haja, de dentro do caminho. Por exemplo, você deseja extrair file.ext de c:\folder\file.ext.

Solução

```
[^\\/:*?"<>|\r\n]+$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Extrair o nome do arquivo de uma string que possua um caminho válido é algo trivial, mesmo que você não saiba se o caminho realmente termina com um nome de arquivo.

O nome do arquivo ocorre sempre no final da string. Ele não pode conter sinais de dois pontos ou barras invertidas; por isso, ele não pode ser confundido com pastas, letras de unidade ou compartilhamentos de rede, que utilizam barras invertidas e/ou dois pontos.

A âncora ‹$› corresponde no final da string (receita 2.5). O fato de o cifrão também corresponder em quebras de linhas incorporadas no Ruby não importa, pois caminhos válidos do Windows não incluem

quebras de linhas. A classe de caracteres negada ‹[^\\/:*?"<>|\r\n]+› (receita 2.3) corresponde aos caracteres que possam ocorrer em nomes de arquivos. Embora o mecanismo de expressão regular percorra a string da esquerda para a direita, a âncora, no final da expressão regular, garante que somente a última carreira de caracteres de nome de arquivo na string seja correspondida, retornando nosso nome de arquivo.

Se a string acabar com uma barra invertida, como em caminhos que não especificam um nome de arquivo, a expressão regular não corresponderá. Quando houver uma correspondência, ela corresponderá somente ao nome do arquivo; assim, não precisamos utilizar quaisquer grupos de captura para separar o nome do arquivo do resto do caminho.

Veja também:

Veja a receita 3.7, para aprender como recuperar um texto correspondido pela expressão regular em sua linguagem de programação favorita.

Siga a receita 7.19, caso não saiba se sua string contém um caminho válido do Windows.

7.24 Extrair a extensão de arquivo de um caminho Windows

Problema

Você possui uma string que contém um caminho (sintaticamente) válido para um arquivo ou pasta, em um PC com Windows local ou em rede, e deseja extrair a extensão de arquivo de dentro do caminho, se houver. Por exemplo, você deseja extrair .ext de c:\folder\file.ext.

Solução

```
\.[^.\\/:*?"<>|\r\n]+$
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Para extrair a extensão do arquivo, podemos utilizar a mesma técnica usada para extrair o nome completo do arquivo, na receita 7.23.

A única diferença está na forma como tratamos os pontos. A expressão regular, na receita 7.23, não inclui quaisquer pontos. A classe de caracteres, negada naquela expressão regular, simplesmente corresponderá a quaisquer pontos que possam estar no nome do arquivo. Uma extensão de arquivo deve começar com um ponto. Assim, adicionamos ‹\.› para corresponder a um ponto literal no início da expressão regular.

Nomes de arquivo como `Version 2.0.txt` podem conter múltiplos pontos. O último ponto é o que delimita a extensão do nome de arquivo. A extensão em si não deve conter nenhum ponto. Especificamos isso na expressão regular colocando um ponto dentro da classe de caracteres. O ponto é, simplesmente, um caractere literal dentro de classes de caracteres, então não precisamos escapá-lo. A âncora ‹$›, no final da expressão regular, nos certifica de que corresponderemos a `.txt`, em vez de `.0`.

Se a string terminar com uma barra invertida, ou com um nome de arquivo que não inclua um ponto, a expressão regular não corresponderá. Quando ela corresponder de fato, corresponderá à extensão, incluindo o ponto que separa a extensão e o nome do arquivo.

Veja também:

Siga a receita 7.19, caso não saiba se sua string contém um caminho válido do Windows.

7.25 Retirar caracteres inválidos de

nomes de arquivos

Problema

Você deseja retirar uma sequência de caracteres inválidos em nomes de arquivos do Windows. Por exemplo, você possui uma string com o título de um documento que deseja usar como nome de arquivo padrão, quando o usuário clicar no botão Salvar pela primeira vez.

Solução

Expressão regular

```
[\\/:"*?<>|]+
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Substituição

Deixe o texto de substituição em branco

Sabores de texto de substituição: .NET, Java, JavaScript, PHP, Perl, Python, Ruby

Discussão

Os caracteres `\/:"*?<>|` não são válidos em nomes de arquivos do Windows. Estes caracteres são usados para delimitar unidades e pastas, para citar caminhos ou para especificar curingas e redirecionamentos na linha de comando.

Podemos corresponder facilmente a estes caracteres com a classe de caracteres `‹[\\/:"*?<>|]›`. A barra invertida é um metacaractere dentro de classes de caracteres; então, precisamos escapá-la com outra barra invertida. Todos os outros caracteres serão sempre caracteres literais, dentro de classes de caracteres.

Repetimos a classe de caracteres com um `‹+›`, por uma questão de eficiência. Dessa forma, se a string possuir uma sequência de caracteres inválidos, toda a sequência será apagada de uma só vez,

e não caractere por caractere. Você não notará diferença no desempenho quando estiver lidando com strings muito curtas, como nomes de arquivos, mas é uma boa técnica para se ter em mente, quando estiver trabalhando com conjuntos de dados propensos a ter carreiras de caracteres mais longas do que você deseja apagar.

Como queremos apenas apagar os caracteres ofensivos, executamos uma pesquisa-e-substituição com a string vazia como texto de substituição.

Veja também:

A receita 3.14 explica como executar uma pesquisa-e-substituição com um texto de substituição fixo em sua linguagem de programação favorita.

---

1 Punycode é um protocolo de programação no qual uma cadeia de caracteres Unicode pode ser traduzida para a codificação de caracteres mais limitada, permitida para nomes de domínio. Referência: *http://pt.wikipedia.org/wiki/Punycode*.

CAPÍTULO 8

Marcação e intercâmbio de dados

Este último capítulo concentra-se em tarefas rotineiras, que surgem quando trabalhamos com uma variedade de linguagens de marcação e formatos comuns: HTML, XHTML, XML, CSV e INI. Embora nele assumamos que você tenha pelo menos alguma familiaridade com estas tecnologias, uma breve descrição de cada uma delas é incluída, no início do capítulo, para nos certificarmos de que estamos falando a mesma língua, antes de mergulharmos de cabeça nos problemas aqui apresentados. As descrições concentram-se nas regras básicas de sintaxe, necessárias para uma correta busca por meio das estruturas de dados de cada formato. Outros detalhes serão introduzidos conforme encontrarmos questões relevantes.

Embora nem sempre aparentes na superfície, alguns destes formatos podem ser surpreendentemente complexos para processar e manipular com precisão, pelo menos quando se usa expressões regulares. Geralmente, é melhor usar analisadores dedicados e APIs, em vez de expressões regulares, ao realizar muitas das tarefas deste capítulo; especialmente se a precisão for crítica (por exemplo, se o seu processamento tiver implicações de segurança). Todavia, estas receitas mostram técnicas úteis, que podem ser usadas em muitas tarefas rápidas de processamento.

Então, vamos dar uma olhada no que estamos enfrentando. Muitas das dificuldades que vamos encontrar, ao longo deste capítulo, envolvem a forma como devemos lidar com os casos que se afastam das regras.

Linguagem de Marcação de Hipertexto (HTML)

> O HTML é usado para descrever a estrutura, a semântica e a aparência de bilhões de páginas da web e de outros documentos. É comum querer processar HTML usando expressões regulares, mas você deve saber, de antemão, que a linguagem é pouco adequada para a rigidez e a precisão das expressões regulares. Isso é especialmente verdadeiro com o HTML mal escrito, comum em muitas páginas web, graças, em parte, à extrema tolerância que os navegadores possuem em relação a HTMLs mal construídos. Neste capítulo, vamos nos concentrar nas regras necessárias para processar os componentes-chave de um HTML bem formado: elementos (e os atributos contidos neles), referências de caractere, comentários e declarações de tipo de documento. Este livro cobre o HTML 4.01, finalizado em 1999 e que continua sendo a última versão finalizada do padrão, até o momento em que escrevemos esse livro.

Os blocos básicos de construção do HTML são chamados de *elementos*. Os elementos são escritos usando *tags*, colocadas entre colchetes angulares. Os elementos são classificados como nível de bloco (por exemplo, parágrafos, títulos, listas, tabelas e formulários) ou interno (por exemplo, hiperlinks, citações, itálicos e controles de entrada de formulários). Elementos, geralmente, possuem tanto uma tag de abertura (por exemplo, `<html>`) como uma tag de fechamento (`</html>`). A tag inicial de um elemento pode conter *atributos*, que serão descritos mais tarde. Entre as tags fica o *conteúdo* do elemento, que pode ser composto de texto e de outros elementos, ou pode ser deixado em branco. Elementos podem ser aninhados, mas não podem se sobrepor (por exemplo, `<div><div></div></div>` está correto, mas não `<div><span></div></span>`). No caso de alguns elementos (como `<p>`, que marca um parágrafo), a tag de fechamento é opcional. Elementos com uma tag de fechamento opcional são fechados automaticamente pelo início de um novo bloco de elementos. Alguns elementos (incluindo `<br>`, que termina uma linha) não podem ter conteúdo próprio, e nunca usam tag de

fechamento. No entanto, um elemento vazio ainda pode conter atributos. Nomes de elementos HTML começam com uma letra de A-Z. Todos os elementos válidos utilizam apenas letras e números em seus nomes. Os nomes dos elementos não fazem distinção entre maiúsculas e minúsculas.

Os elementos <script> e <style> merecem consideração especial: eles deixam você inserir código de linguagem de script e folhas de estilo em seu documento. Estes elementos terminam após a primeira ocorrência de </style> ou </script>, mesmo que apareçam dentro de um comentário, ou de uma string inserida no estilo ou na linguagem de script.

Atributos aparecem dentro da tag de abertura do elemento após seu nome, e são separados por um ou mais caracteres em branco. A maioria dos atributos é escrita como pares nome-valor. Assim, o exemplo a seguir mostra um elemento <a> (âncora), com dois atributos e o conteúdo “Click me!”:

```
<a href="http://www.regexcookbook.com"
  title = 'Regex Cookbook'>Click me!</a>
```

Como mostrado, o nome de um atributo e o valor são separados por um sinal de igual e por espaços em branco opcionais. O valor é delimitado por aspas simples ou duplas. Para usar o tipo de aspa que engloba o valor dentro do próprio valor, você deve usar uma referência de caractere (descrita a seguir). Os caracteres de aspas que englobam o valor não são necessários, se o dito valor contiver apenas os caracteres A-Z, a-z, 0-9, underscore, ponto, sinal de dois pontos e hífen (escritos na forma de expressão regular ficariam ‹^[-.0-9:A-Z_a-z]+$›). Alguns atributos (como selected e checked, utilizados em alguns elementos de formulários) afetam o elemento que os contém, simplesmente pela sua presença, e não exigem um valor. Nestes casos, o sinal de igual, que separa um nome de atributo e seu valor, também é omitido. Alternativamente, estes atributos “minimizados” podem reutilizar seu nome como sendo o valor (por exemplo, selected="selected"). Os nomes de atributos começam com uma letra de A-Z. Todos os atributos válidos usam

somente letras e hífens em seus nomes. Atributos podem aparecer em qualquer ordem, e seus nomes não fazem diferença entre maiúsculas e minúsculas.

A versão 4 do HTML define 252 *referências de entidades de caractere* e mais de um milhão de *referências de caracteres numéricos* (que chamaremos, coletivamente, de *referências de caracteres*). As referências númericas referem-se a um caractere por seu ponto de código Unicode, e utilizam o formato &#*nnnn*; ou &#x*hhhh*;, em que *nnnn* representa um ou mais dígitos decimais, de 0 a 9, e *hhhh* representa um ou mais dígitos hexadecimais, de 0 a 9 e de A a F (sem distinção entre maiúsculas e minúsculas). Referências de entidades de caractere são escritas como &*nomeentidade*; (com distinção entre maiúsculas e minúsculas, diferentemente da maioria dos outros aspectos do HTML), e são especialmente úteis ao introduzirem caracteres literais sensíveis em alguns contextos, como colchetes angulares (< e >), aspas duplas ("), e o sinal de conjunção (&).

Também comum é a entidade   (espaço sem quebra de linha, posição 0xA0), que é particularmente útil, pois todas as ocorrências desta natureza são processadas na tela, mesmo quando aparecem em sequência. Espaços, tabulações e quebras de linha são, normalmente, processados como um único caractere de espaço, mesmo que vários deles estejam inseridos em sequência. O caractere de conjunção (&) não pode ser utilizado fora das referências de caracteres.

Comentários HTML possuem a seguinte sintaxe:

```
<!-- isso é um comentário -->
<!-- isso também, mas este comentário
  abrange mais de uma linha -->
```

O conteúdo dentro dos comentários não possui nenhum significado especial, ficando escondido da maioria dos usuários. Espaço em branco é permitido entre o fechamento -- e >. Por uma questão de compatibilidade com os antigos (anteriores a 1995) navegadores, algumas pessoas englobam o conteúdo dos elementos <script> e

<style> com um comentário HTML. Navegadores modernos ignoram esses comentários, e processam o conteúdo do script ou do estilo normalmente.

Por fim, documentos HTML normalmente começam com uma *declaração de tipo do documento* (informalmente, um “DOCTYPE”), que identifica uma especificação legível para o computador do conteúdo permitido e proibido no documento. O DOCTYPE se parece um pouco com um elemento HTML, como mostrado na linha a seguir, usada em documentos que desejam ficar em conformidade com a definição exata do HTML 4.01:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
```

Então, resumidamente, essa é a estrutura física de um documento HTML. Esteja ciente de que no mundo real o HTML é, muitas vezes, repleto de desvios a essas regras, desvios estes que a maioria dos navegadores ficam felizes em acolher. Para além destes princípios, cada elemento possui restrições sobre o conteúdo e os atributos que podem aparecer dentro dele, a fim de que um documento HTML seja considerado válido. As regras desses conteúdos estão fora do escopo deste livro, mas o livro da O’Reilly *HTML & XHTML: The Definitive Guide*, escrito por Chuck Musciano e Bill Kennedy, é uma boa fonte, caso você precise de mais informações.

Como a estrutura do HTML é muito semelhante ao XHTML e XML (ambos descritos a seguir), muitas expressões regulares, neste capítulo, são escritas para suportar todas as três linguagens de marcação.

Linguagem Extensível de Marcação de Hipertexto (XHTML)

O XHTML foi concebido como o sucessor do HTML 4.01, e migra o HTML de sua herança SGML para uma fundação XML. No entanto, o desenvolvimento do HTML continua separadamente e, portanto, o XHTML é, de maneira mais precisa, considerado como uma alternativa ao HTML. Este livro cobre o XHTML 1.0 e 1.1. Embora essas versões da norma sejam compatíveis, na maior parte, com o HTML, existem algumas diferenças fundamentais em relação à estrutura HTML que acabamos de

descrever:

- Documentos XHTML começam com uma declaração XML como <?xml version="1.0" encoding="UTF-8"?>.
- Elementos não-vazios devem ter uma tag de fechamento. Elementos vazios devem usar uma tag de fechamento, ou terminar com />.
- Nomes de elementos e atributos utilizam minúsculas, e fazem distinção entre maiúsculas e minúsculas.
- Devido ao uso de prefixos de espaços de nome XML, nomes de elementos e atributos podem incluir um sinal de dois pontos, além dos caracteres encontrados em nomes HTML.
- Valores de atributos que não estejam entre aspas não são permitidos. Valores de atributos devem estar entre aspas simples ou duplas.
- Atributos devem ter um valor que os acompanhe.

Há várias outras diferenças entre HTML e XHTML que afetam, principalmente, os casos extremos e a manipulação de erros. Porém, geralmente, elas não afetam as expressões regulares neste capítulo. Para mais informações sobre as diferenças entre HTML e XHTML, consulte *http://www.w3.org/TR/xhtml1/# diffs e http://wiki.whatwg.org/wiki/HTML_vs._XHTML*.

Como a sintaxe do XHTML é muito semelhante à do HTML, sendo formada a partir do XML, muitas expressões regulares, neste capítulo, são escritas para suportar todas as três linguagens de marcação. As receitas que se referem a "(X)HTML" lidam com HTML e XHTML da mesma forma. Você, normalmente, não pode depender de um documento usando somente convenções HTML ou XHTML, já que as misturas são comuns e os navegadores web, geralmente, não se importam com isso.

Linguagem Extensível de Marcação (XML)

XML é uma linguagem de uso geral, projetada primeiramente para compartilhar dados estruturados. É usada como base para a criação de uma grande variedade de linguagens de marcação, incluindo o XHTML, que acabamos de discutir. Este livro cobre as versões 1.0 e 1.1 do XML. Uma descrição completa de seus recursos e sua gramática do XML está além do escopo deste

livro, mas, para os nossos propósitos, vale indicar que existem algumas diferenças fundamentais em relação à estrutura HTML, que já descrevemos:

- Documentos XML podem começar com uma declaração XML, como <?xml version="1.0" encoding="UTF-8"?>, e podem conter outras instruções de processamento, formatadas de maneira semelhante. Por exemplo, <?xml-stylesheet type="text/xsl" href="transform.xslt"?> especifica que o arquivo de transformação XSL transform.xslt deverá ser aplicado ao documento.
- O DOCTYPE pode incluir declarações de marcação internas dentro de colchetes. Por exemplo:

```
<!DOCTYPE example [
  <!ENTITY copy "&#169;">
  <!ENTITY copyright-notice "Copyright &copy; 2008, O'Reilly Media">
]>
```

- Seções CDATA são usadas para escapar blocos de texto. Elas começam com a string <![CDATA[e terminam na primeira ocorrência de]]>.
- Elementos não-vazios devem ter uma tag de fechamento. Elementos vazios devem usar uma tag de fechamento, ou terminar com />.
- Nomes XML (que regem as regras de nomeação de elementos, atributos e referências de entidade) fazem distinção entre maiúsculas e minúsculas, e podem usar um grupo grande de caracteres Unicode. Os caracteres permitidos incluem A-Z, a-z, dois pontos (:), e underscore (_), assim como 0-9, hífen (-) e ponto (.), após o primeiro caractere. Veja a receita 8.4, para informações mais precisas.
- Valores de atributos que não estejam entre aspas não são permitidos. Valores de atributos devem estar entre aspas simples ou duplas.
- Atributos devem ter um valor que os acompanhe.

Há muitas outras regras que devem ser respeitadas para escrever

documentos XML bem formados, ou para escrever o próprio analisador XML. No entanto, as regras que acabamos de descrever (mais as estruturas já delineadas para documentos HTML) são, geralmente, o suficiente para pesquisas de expressão regular simples.

Como a estrutura do XML é muito semelhante ao HTML, constituindo a base do XHTML, muitas expressões regulares, neste capítulo, são escritas para suportar todas as três linguagens de marcação. As receitas que se referem ao "estilo XML" de marcação manipulam XML, XHTML e HTML da mesma forma.

Valores Separados por Vírgulas (CSV)

CSV (Comma-Separated Values) é um formato de arquivo velho, mas ainda muito comum, usado em planilhas de dados. O formato CSV é suportado pela maioria das planilhas e sistemas de gerenciamento de banco de dados, e é especialmente popular para troca de dados entre aplicações. Embora não exista uma especificação CSV oficial, uma tentativa de definição comum foi publicada em outubro de 2005 como a RFC 4180, e registrado na IANA como o tipo MIME "text/csv". Antes da publicação desta RFC, as convenções CSV usadas pelo Microsoft Excel foram estabelecidas mais ou menos como um padrão de fato. Como o RFC especifica regras muito semelhantes às utilizadas pelo Excel, esse não é um grande problema.

Este capítulo aborda os formatos CSV especificados pelo RFC 4180, e usados pelo Microsoft Excel 2003 e posteriores.

Como o nome sugere, arquivos CSV possuem uma lista de valores, ou *campos*, separados por vírgulas. Cada linha, ou *registro*, aparece em sua própria linha. O último campo em um registro não é seguido por uma vírgula. O último registro em um arquivo pode, ou não, ser seguido por uma quebra de linha. Ao longo de todo o arquivo, cada registro deve ter o mesmo número de campos.

O valor de cada campo CSV pode estar sem adornos, ou pode estar colocado entre aspas duplas. Os campos também podem ficar inteiramente vazios. Qualquer campo que contenha vírgulas, aspas duplas ou quebras de linha deve ser colocado entre aspas duplas.

Uma aspa dupla que apareça dentro de um campo é escapada, sendo precededida por outra aspa dupla.

O primeiro registro em um arquivo CSV é, por vezes, usado como um cabeçalho, representando os nomes de cada coluna. Isso não pode ser determinado por meio de programação, a partir do conteúdo de um arquivo CSV; por isso, algumas aplicações solicitam que o usuário decida como a primeira linha deverá ser tratada.

A RFC 4180 especifica que os espaços iniciais e finais de um campo fazem parte do valor. Algumas versões mais antigas do Excel ignoravam estes espaços, mas o Excel 2003 acompanhou a RFC neste ponto. A RFC não especifica o tratamento de erros para aspas duplas não escapadas, ou para praticamente qualquer outra coisa. O tratamento do Excel pode ser imprevisível em casos extremos; por isso, é importante garantir que as aspas duplas estejam escapadas, que os campos contendo aspas duplas estejam eles próprios fechados com aspas duplas, e que os campos entre aspas não contenham espaços iniciais, ou finais, fora das aspas.

O exemplo CSV, a seguir, demonstra muitas das regras que discutimos. Ele contém dois registros com três campos cada um:

```
aaa,b b,"""c"" cc"
1,,"333, three,
still more threes"
```

A tabela 8.1 mostra como este conteúdo CSV seria exibido em uma tabela.

Tabela 8.1 – Exemplo de saída CSV

aaa	b b	“c” cc
1	(empty)	333, three, still more threes

Embora tenhamos descrito as regras CSV, observadas pelas receitas deste capítulo, há uma boa quantidade de variação a respeito de como os diferentes programas leem e escrevem arquivos CSV. Muitos aplicativos até permitem que arquivos com a extensão “csv” utilizem qualquer delimitador, e não apenas vírgulas.

Outras variações comuns incluem a forma como as vírgulas (ou outros delimitadores de campo), as aspas duplas e as quebras de linha são embutidas nos campos, e se os espaços em branco iniciais e finais de campos sem aspas serão ignorados, ou tratados como texto literal.

Arquivos de inicialização (INI)

O formato leve do arquivo INI é, comumente, usado para arquivos de configuração. Ele é mal definido e, como consequência, há muita variação em como o formato é interpretado por diferentes programas e sistemas. As expressões regulares, neste capítulo, aderem às convenções mais comuns do arquivo INI, que iremos descrever aqui.

Parâmetros de arquivos INI são pares nome-valor, separados por um sinal de igual, e espaços ou tabulações opcionais. Os valores podem ser colocados entre aspas simples ou duplas, o que lhes permite conter espaços em branco à esquerda e à direita, e outros caracteres especiais.

Parâmetros podem ser agrupados em *seções*, que começam com o nome da seção entre colchetes em sua própria linha. As seções continuam até a próxima declaração de seção ou até o final do arquivo. Seções não podem ser aninhadas.

Um ponto-e-vírgula marca o início de um *comentário*, que continua até o final da linha. Um comentário pode aparecer, na mesma linha, como um parâmetro ou declaração da seção. Conteúdo dentro de comentários não tem nenhum significado especial.

A seguir, temos um exemplo de arquivo INI com um comentário introdutório (tomando nota da última vez em que o arquivo foi modificado), duas seções (“user” e “post”) e um total de três parâmetros (“name”, “title” e “content”):

```
; última modificação: 2008-12-25
[user]
name=J. Random Hacker
```

```
[post]
title = Regular Expressions Rock!
content = "Let me count the ways..."
```

8.1 Encontrar tags no estilo XML

Problema

Você deseja corresponder a tags HTML, XHTML ou XML em uma string, a fim de remover, modificar, contar ou manipulá-las de qualquer forma.

Solução

A solução mais adequada depende de vários fatores, incluindo os níveis de precisão, eficiência e tolerância a marcações erradas. Depois de determinar a abordagem que atenda as suas necessidades, há uma série de coisas que você pode querer fazer com os resultados. Porém, independentemente de querer remover as tags, fazer buscas dentro delas, adicionar ou remover atributos, ou substituí-las por uma marcação alternativa, o primeiro passo é encontrá-las.

Fique avisado: esta será uma receita longa, cheia de sutilezas, exceções e variações. Se estiver procurando uma solução rápida, e não estiver com muita disposição para determinar a melhor solução para suas necessidades, pode pular para a seção “Tags (X)HTML (permissiva)” desta receita, que oferece uma mistura razoável de tolerância e precaução.

Rápida e rasteira

Esta primeira solução é simples e mais comumente usada do que se poderia esperar. Ela está aqui inclusa, principalmente, para comparação e análise de suas falhas. Porém, pode ser boa o bastante, quando você sabe exatamente com o tipo de conteúdo que está lidando, e não está excessivamente preocupado com as

consequências de uma manipulação incorreta. Esta expressão regular começa correspondendo a <; então, simplesmente continua até o primeiro >:

```
<[^>]*>
```

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Permitir > em valores de atributos

Esta próxima expressão regular é, mais uma vez, bastante simplista, e não trata todos os casos corretamente. No entanto, ela pode funcionar bem para suas necessidades, se usada para processar apenas trechos válidos de (X)HTML. Sua vantagem sobre a expressão regular anterior é que ela passa corretamente por cima de caracteres › que apareçam dentro de valores de atributos:

```
<(?:[^>'"]|"[^"]*"|'[^']*')*>
```

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Aqui está a mesma expressão regular, com espaços em branco e comentários adicionados, para melhorar a legibilidade:

```
<
(?: [^>'"] # Caractere não colocado entre aspas, ou...
  | "[^"]*" # Valor de atributo entre aspas duplas, ou...
  | '[^']*' # Valor de atributo entre aspas simples
)*
>
```

Opções Regex: Espaçamento livre
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

As duas expressões regulares mostradas trabalham de forma idêntica, então você pode usar a que preferir. No entanto, programadores JavaScript estão presos à primeira opção, pois ela não possui opção de espaçamento livre.

Tags (X)HTML (permissiva)

Além de suportar caracteres › embutidos em valores de atributos, a próxima expressão regular emula as regras permissivas para nomes

de tags (X)HTML que os navegadores, em geral, implementam. Isso permite que a expressão regular evite conteúdos que não se pareçam com uma tag, incluindo comentários, DOCTYPEs e caracteres <, dentro do texto, que não estejam codificados. Ela usa o mesmo tratamento de atributos e outros caracteres perdidos que possam aparecer dentro de uma tag, tal qual a expressão regular anterior, mas adiciona um tratamento especial ao nome da tag. Especificamente, é necessário que o nome começe com uma letra da Língua Inglesa. O nome da tag é capturado na retrorreferência 1, no caso de você precisar referenciá-la:

```
</?([A-Za-z][^\s>/]*)(?:[^>'"]|"[^"]*"|'[^']*')*>
```

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

E no modo de espaçamento livre:

```
<
/? # Permite tags de fechamento
([A-Za-z][^\s>/]*) # Captura o nome da tag na retrorreferência 1
(?: [^>'"] # Caractere não colocado entre aspas, ou...
  | "[^"]*" # Valor de atributo entre aspas duplas, ou...
  | '[^']*' # Valor de atributo entre aspas simples
)*
>
```

Opções Regex: Espaçamento livre
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

As duas últimas expressões regulares trabalham de forma idêntica, embora a última não possa ser usada em JavaScript, pois ele carece de uma opção para espaçamento livre.

Tags (X)HTML (estrita)

Esta expressão regular é mais complicada do que as vistas nesta receita, porque, na verdade, ela segue as regras de tags (X)HTML, explicadas na parte introdutória deste capítulo. Isso nem sempre é desejável, uma vez que os navegadores não aderem estritamente a essas regras. Em outras palavras, essa expressão regular vai evitar a correspondência de conteúdo que não se pareça com tags

(X)HTML válidas, ao custo de, eventualmente, não corresponder a algum conteúdo que os navegadores, de fato, interpretariam como uma tag (por exemplo, se sua marcação usar um nome de atributo que inclua caracteres não contabilizados, ou se os atributos forem incluídos em uma tag de fechamento). As regras de tags do HTML e do XHTML são tratadas em conjunto, pois é comum que suas convenções sejam misturadas. O nome da tag é capturado na retrorreferência 1 ou 2 (dependendo se for uma tag de abertura ou de fechamento), no caso de precisar referenciá-la:

```
<(?:([A-Z][-:A-Z0-9]*)(?:\s+[A-Z][-:A-Z0-9]*(?:\s*=\s*(?:"[^"]*"|↲
'[^']*'|[-.:\w]+))?)*\s*/?|/([A-Z][-:A-Z0-9]*)\s*)>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Para torná-la um pouco menos obscura, aqui está a mesma expressão regular no modo de espaçamento livre, com comentários:

```
< #
(?: # Seção das tags de abertura...
  ([A-Z][-:A-Z0-9]*) # Captura o nome da tag de abertura na retrorreferência 1
  (?: # Permite zero ou mais atributos...
    \s+ # ...separados por espaços em branco
    [A-Z][-:A-Z0-9]* # Nome de atributo
    (?: #
      \s*=\s* # Delimitador do par nome-valor do atributo
      (?: "[^"]*" # Valores de atributo entre aspas duplas
        | '[^']*' # Valores de atributo entre aspas simples
        | [-.:\w]+ # Valor de atributo sem aspas (HTML)
      ) #
  )? # Permite atributos sem um valor (HTML)
 )* #
 \s* # Permite espaços em branco iniciais
 /? # Permite tags com autofechamento (XHTML)
 | # Seção das tags de fechamento...
  / #
  ([A-Z][-:A-Z0-9]*) # Captura a tag de fechamento na retrorreferência 2
  \s* # Permite espaços em branco iniciais
) #
> #
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, espaçamento livre
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Tags XML (estrita)

XML é uma linguagem muito bem especificada. Ela exige dos agentes de usuário aderência e imposição estritas de suas regras. Trata-se de uma mudança gritante em relação ao HTML, e aos navegadores resignados que o processam:

```
<(?:([_:A-Z][-.:\w]*)(?:\s+[_:A-Z][-.:\w]*\s*=\s*(?:"[^"]*"|'[^']*'))*\s*↲
/?|/([_:A-Z][-.:\w]*)\s*)>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Mais uma vez, vejamos a mesma expressão regular no modo de espaçamento livre, com comentários acrescentados:

```
< #
(?: # Seção das tags de abertura...
  ([_:A-Z][-.:\w]*) # Captura o nome da tag de abertura na retrorreferência 1
  (?: # Permite zero, ou mais atributos...
    \s+ # ...separados por espaços em branco
    [_:A-Z][-.:\w]* # Nome de atributo
    \s*=\s* # Delimitador do par nome-valor do atributo
    (?: "[^"]*" # Valores de atributo entre aspas duplas
      | '[^']*' # Valores de atributo entre aspas simples
    ) #
  )* #
  \s* # Permite espaços em branco iniciais
  /? # Permite tags com autofechamento
 | # Seção das tags de fechamento...
  / #
  ([_:A-Z][-.:\w]*) # Captura a tag de fechamento na retrorreferência 2
  \s* # Permite espaços em branco iniciais
) #
> #
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, espaçamento livre
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Assim como nas duas soluções anteriores, mostradas para as tags (X)HTML, estas expressões regulares capturam o nome da tag na retrorreferência 1 ou 2, se uma tag de abertura, ou fechamento, for correspondida. A expressão regular da tag XML é um pouco menor em comparação com as versões (X)HTML, pois ela não precisa lidar

com a sintaxe HTML (atributos minimizados e valores que não estejam entre aspas). Ela também permite usar uma gama mais ampla de caracteres para nomes de elementos e atributos.

Discussão

Algumas palavras de cautela

Embora seja comum querer corresponder a tags no estilo XML usando expressões regulares, fazer isso com segurança requer equilibrar perdas e ganhos, além de pensar cuidadosamente a respeito dos dados com os quais estiver trabalhando. Devido a essas dificuldades, algumas pessoas optam por abandonar o uso de expressões regulares em qualquer tipo de processamento XML ou (X)HTML, em favor de analisadores e APIs especializados. Esta é uma abordagem que você deveria considerar fortemente, uma vez que tais instrumentos são tipicamente otimizados para executar tais tarefas de forma rápida, e eles incluem a detecção robusta ou tratamento de marcações incorretas. No universo dos navegadores, por exemplo, é melhor tirar proveito da árvore do Modelo de Objetos do Documento (DOM) para as suas necessidades de manipulação e pesquisa HTML. Em outros ambientes, você estaria bem servido com um analisador SAX ou com o XPath. No entanto, você pode, ocasionalmente, encontrar locais nos quais as soluções baseadas em expressão regular fazem muito sentido e funcionam perfeitamente.

Com esse aviso, vamos examinar as expressões regulares já vistas nesta receita. As duas primeiras soluções são demasiadamente simplistas, na maioria dos casos, mas lidam com as linguagens de marcação no estilo XML de forma igual. As últimas três seguem regras mais rigorosas e são adaptadas às suas respectivas linguagens de marcação. Mesmo nas últimas soluções, no entanto, convenções de tags HTML e XHTML são tratadas em conjunto, uma vez que é comum serem misturadas, muitas vezes inadvertidamente. Por exemplo, um autor pode fazer uso de uma tag

de autofechamento
, em um documento HTML, ou usar incorretamente um nome de elemento em letras maiúsculas dentro de um documento com um DOCTYPE XHTML.

Rápida e rasteira

A vantagem desta solução é a sua simplicidade, o que a torna mais fácil de lembrar e digitar, além de ter uma execução rápida. A desvantagem é que ela manipula incorretamente certas construções XML e (X)HTML, válidas e inválidas. Se estiver trabalhando com uma marcação que você mesmo escreveu, e sabe que esses casos nunca vão aparecer em seu texto de assunto, ou se você não está preocupado com as consequências, esta desvantagem pode até ser boa. Outro exemplo de onde esta solução poderia ser boa ocorre quando você está trabalhando com um editor de texto que permite a visualização prévia das correspondências de uma expressão regular.

A expressão regular começa encontrando um caractere literal ‹<› (o começo de uma tag). Em seguida, ela usa uma classe de caracteres negada e o quantificador asterisco mesquinho ‹[^>]*›, para corresponder a zero ou mais caracteres que não sejam um >. Com isso, cuidamos de corresponder ao nome da tag, aos atributos e a uma barra (/) inicial ou final. Poderíamos ter usado um quantificador preguiçoso (‹[^>]*?›), mas isso não mudaria nada, a não ser tornar a expressão regular um pouco mais lenta, já que causaria mais retrocessos (a receita 2.13 explica o porquê). Em seguida, para finalizar a tag, a expressão regular corresponde a um ‹›› literal.

Se você preferir usar um ponto, no lugar da classe de caracteres negada ‹[^>]›, vá em frente. Um ponto funcionará bem, desde que você também use um asterisco preguiçoso (‹.*?›) junto com ele, e certifique-se de habilitar a opção “ponto corresponde a quebras de linha” (em JavaScript, você poderia usar ‹[\s\S]*?›). Um ponto com um asterisco mesquinho (gerando o padrão completo ‹<.*>›) mudaria o significado da expressão regular, fazendo-a corresponder incorretamente, do primeiro < até o último ›, na string de assunto,

mesmo que, para tanto, a expressão regular tenha de engolir várias tags ao longo do caminho.

É hora de alguns exemplos. Essa expressão regular corresponde a cada uma das seguintes linhas na íntegra:

```
<div>
</div>
<div class="box">
<div id="pandoras-box" class="box" />
<!-- comment -->
<!DOCTYPE html>
<< < w00t! >
<>
```

Observe que o padrão corresponde a mais do que apenas tags. Pior, não corresponderá corretamente a todas as tags nas strings de assunto `<input type="button" value=">>">`, ou `<input type="button" onclick="alert(2 > 1)">`. Ao invés disso, ele só corresponderá até o primeiro `>`, que aparece dentro do valor de atributo. Ele terá problemas semelhantes com comentários, seções CDATA do XML, DOCTYPEs, código dentro de elementos `<script>`, e qualquer outra coisa que contenha símbolos `>` incorporados.

Se estiver processando qualquer coisa que vá além das marcações mais básicas, especialmente se o texto de assunto for proveniente de fontes mistas ou desconhecidas, você estará mais bem servido por soluções mais robustas, mostradas mais adiante nesta receita.

Permitir > em valores de atributos

Semelhante à expressão regular rápida e rasteira que acabamos de descrever, esta foi incluída principalmente para contrastar com as últimas soluções, mais robustas. No entanto, ela aborda os conceitos básicos necessários para corresponder a tags no estilo XML, e assim ela poderia servir bem a suas necessidades, caso seja usada para processar trechos de marcação válidos que incluam apenas elementos e texto. A diferença, em relação à expressão regular anterior, é que esta passa por cima dos caracteres › que apareçam dentro de valores de atributos. Por exemplo, ela

corresponderá corretamente a todas as tags <input>, nos exemplos de strings de assunto mostrados anteriormente: <input type="button" value=">>"> e <input type="button" onclick="alert(2 > 1)">.

Como antes, a expressão regular usa caracteres de colchetes angulares literais, nas extremidades da expressão regular, para corresponder ao início e ao final de uma tag. No meio, ela repete um grupo de não captura contendo três alternativas, cada uma delas separada pelo metacaractere de alternância ‹|›.

A primeira alternativa é a classe de caracteres negada ‹[^>"']›, que corresponde a qualquer caractere que não seja um colchete angular direito (que fecha a tag), uma aspa dupla ou uma aspa simples (ambas as aspas indicam o início de um valor de atributo). Esta primeira alternativa é responsável pela correspondência da tag e dos nomes de atributos, bem como de quaisquer outros caracteres fora dos valores entre aspas. A ordem das alternativas é intencional e escrita tendo o desempenho em mente. Mecanismos de expressão regular tentam caminhos alternativos ao longo de uma expressão regular, da esquerda para a direita; as tentativas de corresponder a esta primeira opção, provavelmente, obterão sucesso com mais frequência, em comparação com as alternativas para os valores entre aspas (especialmente porque ela corresponde a apenas um caractere por vez).

Em seguida, temos as alternativas que correspondem a valores de atributos entre aspas duplas e simples (‹"[^"]*"› e ‹'[^']*'›). O uso que elas fazem de classes de caracteres negadas permite-lhes continuar correspondendo após quaisquer caracteres › inclusos, quebras de linha e qualquer outra coisa que não seja uma aspa de encerramento.

Note que esta solução não possui um tratamento especial que permita excluir ou corresponder devidamente a comentários, e a outros nós especiais, em seus documentos. Certifique-se de estar familiarizado com o tipo de conteúdo com o qual está trabalhando, antes de colocar esta expressão regular em funcionamento.

Uma otimização (segura) da eficiência

Após a leitura dessa seção, é possível que você pense que a expressão regular poderia ser um pouco mais rápida, bastando, para tanto, adicionar um quantificador ‹*› ou ‹+› depois da classe de caracteres negada (‹[^>'"]›) inicial. Nas posições dentro da string de assunto, nas quais a expressão regular encontra correspondências, você estaria certo. Ao corresponder a mais de um caractere por vez, você deixaria o mecanismo de expressão regular pular muitos passos desnecessários para uma correspondência bem-sucedida.

O que não é facilmente perceptível é a consequência negativa que uma alteração desse tipo poderia ter em lugares nos quais o mecanismo de expressão regular só encontra uma correspondência parcial. Quando a expressão regular corresponde a um caractere < de abertura, mas não existe um > seguinte, que permita a conclusão bem-sucedida da tentativa de correspondência, você entrará no problema de “retrocesso catastrófico”, descrito na receita 2.15. Tal ocorrência deve-se ao grande número de formas a partir das quais o novo quantificador interno poderia ser combinado com o quantificador externo (após o grupo de não-captura), para corresponder ao texto após o <; todas essas formas teriam de ser tentadas pelo mecanismo, antes dele desistir da tentativa de correspondência. Cuidado!

No caso de sabores de expressão regular que suportem quantificadores possessivos ou grupos atômicos (JavaScript e Python não possuem nenhum dos dois), é possível evitar este problema, e ainda ganhar vantagem no desempenho, ao corresponder de uma só vez a mais de um caractere que não esteja entre aspas. Na verdade, podemos ir além, e reduzir o retrocesso em potencial em outras partes da expressão regular. Se o sabor de expressão regular que você está usando oferece suporte a ambos os recursos, os quantificadores possessivos (exibidos, aqui, na segunda expressão regular) são sua melhor opção, uma vez que eles mantêm a expressão regular mais curta e legível.

Com grupos atômicos:

```
<(?>(?:(?>[^>'"]+)|"[^"]*"|'[^']*')*)>
```

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, PCRE, Perl, Ruby

Com quantificadores possessivos:

```
<(?:[^>'"]++|"[^"]*"|'[^']*')*+>
```

Opções Regex: Nenhuma
Sabores Regex: Java, PCRE, Perl 5.10, Ruby 1.9

Tags (X)HTML (permissiva)

Por meio de uma simples inclusão, esta expressão regular fica muito mais perto de emular as regras permissivas que os navegadores web usam para identificar tags (X)HTML no código-fonte. Ela se torna, assim, uma boa solução para os casos em que você tenta copiar o comportamento do navegador, e não se importa se as tags correspondidas realmente seguem todas as regras de uma marcação válida. Tenha em mente, porém, o seguinte: é possível

criar HTMLs horrivelmente inválidos, intratáveis mesmo por essa expressão regular, e apesar de aceitos por alguns navegadores, posto que navegadores analisam casos extremos de marcação errada cada um a sua maneira.

A diferença mais significativa dessa expressão regular, em relação à solução anterior, é que ela requer que o caractere após o colchete angular esquerdo de abertura (<) seja uma letra de A-Z ou a-z, opcionalmente precedida por / (para tags de fechamento). Esta restrição exclui a correspondência de caracteres < perdidos, não codificados no texto, bem como de comentários, DOCTYPEs, declarações XML, instruções de processamento, seções CDATA etc. Isso não protege a expressão regular de corresponder a algo que se pareça com uma tag dentro de comentários, códigos de linguagem de script, conteúdo dos elementos <textarea> e assim por diante. A próxima seção, “Pular seções (X)HTML e XML complicadas”, dá uma solução para este problema. Mas, primeiro, vejamos como funciona a expressão regular.

A correspondência começa com um colchete angular esquerdo ‹<›. O ‹/?›, que se segue, permite uma barra opcional para tags de fechamento. Em seguida vem o grupo de captura ‹([A-Za-z][^\s>/]*)›, que corresponde ao nome da tag e a armazena na retrorreferência 1. Se não precisar referenciar o nome da tag (por exemplo, se estiver simplesmente removendo todas as tags), você pode remover os parênteses de captura (apenas não se livre do padrão dentro deles). Dentro do grupo estão duas classes de caracteres. A primeira, ‹[A-Za-z]›, define a regra do caractere inicial para nomes de tags. A próxima classe, ‹[^\s>/]›, permite que quase todos os caracteres sigam-no, como parte do nome. As únicas exceções são o espaço em branco (‹\s›, que separa o nome da tag de quaisquer atributos seguintes), › (que finaliza a tag) e / (usado antes da tag de fechamento >, em tags singulares no estilo XHTML). Quaisquer outros caracteres (incluindo aspas) são tratados como parte do nome da tag. Este procedimento pode parecer excessivamente permissivo, mas é como a maioria dos navegadores opera. Tags

falsas podem não ter efeito sobre a forma como uma página é processada, mas, de qualquer forma, elas tornam-se acessíveis por meio da árvore DOM e não são desenhadas na tela como texto, apesar do fato de que qualquer conteúdo dentro delas será exibido.

Após o nome da tag vem a manipulação dos atributos, que ocorre de maneira direta, fora da expressão regular anterior: ‹(?:[^>'"]|"[^"]*"|'[^']*')*›. Adicione um colchete angular direito para finalizar a tag, e pronto!

As expressões regulares a seguir mostram como este modelo pode ser adaptado para corresponder somente a tags de abertura, de fechamento ou singulares (autofechamento):

Tags de abertura

```
<([A-Za-z][^\s>/]*)(?:[^>'"/]|"[^"]*"|'[^']*')*>
```

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Esta versão adiciona uma barra (/) na primeira classe de caracteres negada do grupo de não-captura, para evitar a ocorrência de barras em qualquer lugar que não seja dentro dos valores de atributos entre aspas.

Tags singulares (singleton)

```
<([A-Za-z][^\s>/]*)(?:[^>'"]|"[^"]*"|'[^']*')*/>
```

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Aqui, nós adicionamos uma barra obrigatória, imediatamente antes do colchete angular direito de fechamento.

Tags de abertura e singulares

```
<([A-Za-z][^\s>/]*)(?:[^>'"]|"[^"]*"|'[^']*')*>
```

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Não há inclusões neste caso. Em vez disso, a ‹/?›, que surgiu após a ‹<› de abertura na expressão regular original, foi removida.

Tags de fechamento

```
</([A-Za-z][^\s>/]*)(?:[^>"']|"[^"]*"|'[^']*')*>
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Aqui, a barra após o colchete angular esquerdo de abertura tornou-se parte obrigatória da correspondência. Note que estamos, deliberadamente, permitindo atributos dentro de tags de fechamento, já que esta expressão regular é baseada na solução “permissiva”. Embora os navegadores não usem atributos que ocorram em tags de fechamento, eles não se importam se esses atributos existirem.

A barra lateral “Uma otimização (segura) da eficiência”, mostrou como melhorar o desempenho, ao corresponder a tags por meio da utilização de grupos atômicos ou quantificadores possessivos.

Dessa vez, o potencial de melhora de desempenho é ainda maior, pois os caracteres que podem ser correspondidos pela classe de caracteres ‹[^\s>/]› se sobrepõem à última parte da expressão regular, proporcionando, assim, muitas combinações de padrão a serem tentadas antes que o mecanismo de expressão regular desista de uma correspondência parcial.

Se os grupos atômicos ou quantificadores possessivos estiverem disponíveis no sabor de expressão regular que estiver usando, será possível obter um desempenho significativamente melhor, com o aproveitamento deles. As seguintes alterações também podem ser transferidas para as expressões regulares específicas às tags de abertura/fechamento/singulares mostradas:

```
</?([A-Za-z](?>[^\s>/]*))(?>(?:(?>[^>"']+)|"[^"]*"|'[^']*')*)>
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, PCRE, Perl, Ruby

```
</?([A-Za-z][^\s>/]*+)(?:[^>"']++|"[^"]*"|'[^']*')*+>
```

Opções Regex: Nenhuma

Sabores Regex: Java, PCRE, Perl 5.10, Ruby 1.9

Tags (X)HTML (estrita)

Ao dizer que esta solução é estrita, queremos dizer que ela tenta

seguir as regras de sintaxe HTML e XHTML explicadas na parte introdutória deste capítulo, ao invés de imitar as regras que os navegadores realmente usam, ao analisar o código-fonte de um documento. Este rigor adiciona as seguintes regras em relação às expressões regulares anteriores:

- Nomes e atributos de tags devem começar com uma letra de A-Z ou a–z, e seus nomes só podem utilizar caracteres de A-Z, a-z, 0-9, hífen e o sinal de dois pontos (no formato de expressão regular ficaria ‹^[-:A-Za-z0-9]+$›).
- Caracteres perdidos e impróprios não são permitidos após o nome da tag. Apenas espaços em branco, atributos (com ou sem um valor que os acompanhe) e, opcionalmente, uma barra final (/) podem aparecer após o nome da tag.
- Valores de atributos que não estejam entre aspas somente podem usar os caracteres A-Z, a-z, 0-9, underscore, hífen, ponto e sinal de dois pontos (no formato de expressão regular ficaria ‹^[-.:A-Za-z0-9_]+$›).
- Tags de fechamento não podem incluir atributos.

Como o padrão é dividido em duas ramificações (a primeira para as tags de abertura e singulares, e a segunda, para as tags de fechamento), o nome da tag é capturado na retrorreferência 1 ou 2, dependendo do tipo de tag correspondida. Ambos os conjuntos de parênteses de captura podem ser eliminados, se você não tiver necessidade de referenciar os nomes das tags.

Nos exemplos a seguir, as duas ramificações do padrão são separadas em suas próprias expressões regulares. Ambas capturam o nome da tag, na retrorreferência 1:

Tags de abertura e singulares

```
<([A-Z][-:A-Z0-9]*)(?:\s+[A-Z][-:A-Z0-9]*(?:\s*=\s*↲
(?:"[^"]*"|'[^']*'|[-.:\w]+))?)*\s*/?>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

O ‹/?›, que aparece pouco antes do ‹>› de fechamento, é o

elemento que permite esta expressão regular corresponder a tags de abertura e singulares. Remova-o para corresponder apenas a tags de abertura. Remova apenas o quantificador ponto de interrogação, e ela corresponderá apenas a tags singulares.

Tags de fechamento

```
</([A-Z][-:A-Z0-9]*)\s*>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Nas duas últimas seções, mostramos como obter um ganho de desempenho potencial, adicionando grupos atômicos ou quantificadores possessivos. Os caminhos estritamente definidos ao longo desta expressão regular significam que não existe potencial para corresponder às mesmas strings em mais de uma maneira e, portanto, há um menor potencial de retrocesso com que se preocupar. Esta expressão regular não *depende* de retrocessos, então, se você quisesse, poderia transformar cada ‹*›, ‹+›, e ‹?› final em um quantificador possessivo (ou atingir o mesmo efeito usando grupos atômicos). Apesar de sabermos que tal ocorrência só ajudaria, vamos pular tais variações nesta (e na próxima) expressão regular, para tentar manter sob controle a quantidade maluca de opções desta receita.

Veja “Pular seções (X)HTML e XML complicadas”, para estudar uma forma de evitar correspondência a tags dentro de comentários, tags <script> e assim por diante.

Tags XML (estrito)

O XML elimina a necessidade de uma solução “permissiva” por meio de sua especificação precisa, e do requisito de que analisadores conformes não deverão processar marcação que não seja bem formada. Embora você possa usar as expressões regulares anteriores para processar documentos XML, a simplicidade não lhe dará a vantagem de realmente fornecer uma pesquisa mais confiável, já que não há a necessidade de emular o comportamento

permissivo de agentes de usuário XML.

Esta expressão regular é, basicamente, uma versão mais simples da expressão regular “(X)HTML (estrito)”, já que podemos remover o suporte a duas características HTML que não são permitidas em XML: valores de atributos que não estejam entre aspas e atributos minimizados (atributos sem um valor que os acompanhe). Outra diferença são os caracteres permitidos como parte dos nomes de tags e atributos. Na verdade, as regras para nomes XML (que regem os requisitos para nomes de tags e atributos) são mais permissivas do que o mostrado aqui, permitindo milhares de caracteres Unicode adicionais. Se você precisa permitir estes caracteres em sua busca, pode substituir as três ocorrências de ‹[_:A-Z][-.:\w]*› por um dos padrões encontrados na receita 8.4. Note que a lista de caracteres permitidos varia de acordo com a versão do XML em uso.

Tal como acontece com as expressões regulares (X)HTML, o nome da tag é capturado na retrorreferência 1 ou 2, dependendo se for tag de abertura/singular, ou tag de fechamento. E, mais uma vez, você pode remover os parênteses de captura, se não precisar fazer referência aos nomes de tag.

Nos exemplos seguintes, as duas ramificações do padrão são separadas em suas próprias expressões regulares. Como resultado, ambas as expressões regulares capturam o nome da tag na retrorreferência 1:

Tags de abertura e únicas

```
<([_:A-Z][-.:\w]*)(?:\s+[_:A-Z][-.:\w]*\s*=\s*↲
(?:"[^"]*"|'[^']*'))*\s*/?>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

O ‹/?›, que aparece pouco antes do ‹>› de fechamento, é o elemento que permite que esta expressão regular coresponda a tags de abertura e singulares. Remova-o, para corresponder apenas a tags de abertura. Remova apenas o quantificador ponto

de interrogação, e ela corresponderá apenas a tags singulares.

Tags de fechamento

```
</([_:A-Z][-.:\w]*)\s*>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Veja a seção “Pular seções (X)HTML e XML complicadas”, para estudar uma forma de evitar correspondência a tags dentro de comentários, seções CDATA e DOCTYPEs.

Pular seções (X)HTML e XML complicadas

Ao tentar corresponder a tags no estilo XML, dentro de um arquivo-fonte ou string, grande parte da batalha é evitar conteúdo que se pareça com uma tag, mesmo que sua colocação ou outro contexto impeça que seja interpretado como tag. As expressões regulares específicas (X)HTML e XML, mostradas nesta receita, evitam conteúdo problemático, ao restringirem o caractere inicial do nome de um elemento. Algumas foram ainda mais longe, exigindo que as tags se conformassem às regras de sintaxe (X)HTML ou XML. Ainda assim, uma solução robusta exige que nós, também, evitemos conteúdo que apareça nos comentários, códigos de linguagem de script (que podem usar os símbolos maior que e menor que, em operações matemáticas), seções CDATA de XML e várias outras construções. Podemos resolver este problema pesquisando esses trechos problemáticos, para, em seguida, pesquisar por tags somente no conteúdo fora dessas correspondências.

A receita 3.18 mostra como escrever o código que faz uma busca entre as correspondências de outra expressão regular. São necessários dois padrões: uma expressão regular interna e uma expressão regular externa. As soluções já delineadas servirão como nossa expressão regular interna. A expressão regular externa é mostrada a seguir, com padrões distintos para (X)HTML e XML. Esta abordagem retira as seções problemáticas da linha de visão da expressão regular interna e, assim, nos permite manter as coisas

relativamente simples.

Expressão regular externa para (X)HTML. A seguinte expressão regular corresponde a comentários, bem como a elementos <script>, <style>, <textarea>, e <xmp>[1] (incluindo seu conteúdo):

```
<!--.*?--\s*>|<(script|style|textarea|xmp)\b(?:[^>"']|"[^"]*"|↲
'[^']*')*?(?:/>|>.*?</\1\s*>)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, ponto corresponde a quebras de linha

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Caso esta não seja a linha de código mais legível que você já tenha visto, aqui vai a expressão regular, novamente no modo de espaçamento livre, com alguns comentários adicionados:

```
# Comentário
<!-- .*? --\s*>
|
# Elemento especial e seu conteúdo
<( script | style | textarea | xmp )\b
   (?: [^>"'] # Caractere que não esteja entre aspas
     | "[^"]*" # Valor entre aspas duplas
     | '[^']*' # Valor entre aspas simples
   )*?
(?: # Tag singular
   />
  | # Caso não seja, inclui o conteúdo do elemento e a tag de fechamento
correspondente
   > .*? </\1\s*>
)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, ponto corresponde a quebras de linha, espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Nenhuma dessas expressões regulares funciona corretamente em JavaScript, pois ela não possui as opções “ponto corresponde a quebras de linha” e “espaçamento livre”. A seguinte expressão regular volta a ser ilegível e substitui os pontos por ‹[\s\S]›, para que possa ser usada no JavaScript:

```
<!--[\s\S]*?--\s*>|<(script|style|textarea|xmp)\b(?:[^>"']|"[^"]*"|↲
'[^']*')*?(?:/>|>[\s\S]*?</\1\s*>)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: JavaScript

Estas expressões regulares apresentam um dilema: como elas correspondem a tags <script>, <style>, <textarea> e <xmp>, estas tags nunca são correspondidas pela segunda expressão regular (interior), embora, supostamente, estejamos pesquisando por todas as tags. No entanto, trata-se apenas de adicionar um pouco de código, para lidar com essas tags. Estas expressões regulares (exteriores) já capturam os nomes de tags na retrorreferência 1, que pode servir como boa verificação para saber se elas corresponderam a um comentário ou a uma tag (se for uma tag, você saberá qual).

Expressão regular externa para XML. Esta expressão regular corresponde a comentários, seções CDATA e DOCTYPEs. Cada um dessses casos é correspondido mediante um padrão distinto; todos são combinados em uma única expressão regular, usando o metacaractere de alternância ‹|›:

```
<!--.*?--\s*>|<!\[CDATA\[.*?]]>|<!DOCTYPE\s(?:[^<>"']|"[^"]*"|
'[^']*'|<!(?:[^>"']|"[^"]*"|'[^']*')*>)*>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, ponto corresponde a quebras de linha
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Vejamos a expressão, novamente, no modo de espaçamento livre:

```
# Comentário
<!-- .*? --\s*>
|
# Seção CDATA
<!\[CDATA\[ .*? ]]>
|
# Declaração de tipo de documento
<!DOCTYPE\s
   (?: [^<>"'] # Caracteres não-especiais
     | "[^"]*" # Valor entre aspas duplas
     | '[^']*' # Valor entre aspas simples
     | <!(?:[^>"']|"[^"]*"|'[^']*')*> # Declaração da marcação
  )*
>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, ponto corresponde a quebras de linha, espaçamento livre
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

E aqui está uma versão que funciona em JavaScript (sem as opções “ponto corresponde a quebras de linha” e “espaçamento livre”):

```
<!--[\s\S]*?--\s*>|<!\[CDATA\[[\s\S]*?]]>|<!DOCTYPE\s(?:[^<>'"]|"[^"]*"|↲
'[^']*'|<!(?:[^>'"]|"[^"]*"|'[^']*')*>)*>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: JavaScript

Variações

Corresponder a tags HTML 4 válidas

Ocasionalmente, você pode querer limitar sua pesquisa a elementos HTML válidos, especialmente em se tratando de pesquisa de tags em documentos não-HTML, na qual a precaução extra contra falsos positivos pode ser importante. A expressão regular, a seguir, corresponde apenas aos 91 elementos válidos do HTML 4. Esta lista não inclui o HTML fora do padrão, como as tags proprietárias <blink>, <bgsound>, <embed› e <nobr>. Ela também não inclui elementos XHTML 1.1 (o XHTML 1.0 não adiciona novas tags), ou novos elementos previstos para o HTML 5:

```
</?(a|abbr|acronym|address|applet|area|b|base|basefont|bdo|big|blockquote|↲
body|br|button|caption|center|cite|code|col|colgroup|dd|del|dfn|dir|div|↲
dl|dt|em|fieldset|font|form|frame|frameset|h1|h2|h3|h4|h5|h6|head|hr|html|↲
i|iframe|img|input|ins|isindex|kbd|label|legend|li|link|map|menu|meta|↲
noframes|noscript|object|ol|optgroup|option|p|param|pre|q|s|samp|script|↲
select|small|span|strike|strong|style|sub|sup|table|tbody|td|textarea|↲
tfoot|th|thead|title|tr|tt|u|ul|var)\b(?:[^>'"]|"[^"]*"|'[^']*')*>
```

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Podemos tornar essa expressão regular mais rápida, reduzindo o número de alternativas separadas pelo metacaractere ‹|›. Ao invés disso, vamos usar classes de caracteres e sufixos opcionais, sempre que possível. Estas mudanças podem reduzir drasticamente

a quantidade de retrocesso exigido pelo mecanismo de expressão regular. Considere o que acontece quando o mecanismo encontra `<0` no texto de assunto. Esse ponto não pode ser o começo de uma tag, uma vez que nenhum nome de tag começa com o número `0`, mas, antes que o mecanismo de expressão regular possa excluir tal elemento como correspondência, ele deve verificar se qualquer uma das 91 alternativas começa com um `0` literal. Ao reduzirmos o número de alternativas para o mínimo (uma para cada letra que possa iniciar uma tag), reduzimos o número de etapas que devem ser tentadas após a descoberta de um colchete angular esquerdo, de 91 para apenas 19. Veja como fica tal expressão regular:

```
</?(a(?:bbr|cronym|ddress|pplet|rea)?|b(?:ase(?:font)?|do|ig|lockquote|↲
ody|r|utton)?|c(?:aption|enter|ite|o(?:de|l(?:group)?))|d(?:[dlt]|el|fn|↲
i[rv])|em|f(?:ieldset|o(?:nt|rm)|rame(?:set)?)|h(?:[1-6r]|ead|tml)|↲
i(?:frame|mg|n(?:put|s)|sindex)?|kbd|l(?:abel|egend|i(?:nk)?)|m(?:ap|↲
e(?:nu|ta))|no(?:frames|script)|o(?:bject|l|p(?:tgroup|tion))|p(?:aram|↲
re)?|q|s(?:amp|cript|elect|mall|pan|t(?:rike|rong|yle)|u[bp])?|t(?:able|↲
body|[dhrt]|extarea|foot|head|itle)|ul?|var)\b(?:[^>'"]|"[^"]*"|'[^']*')*>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Certamente, é mais difícil de ler, mas o modo de espaçamento livre pode ajudar um pouco:

```
<
/? # Permite tags de fechamento
( # Captura o nome da tag na retrorreferência 1
  a(?:bbr|cronym|ddress|pplet|rea)?|
  b(?:ase(?:font)?|do|ig|lockquote|ody|r|utton)?|
  c(?:aption|enter|ite|o(?:de|l(?:group)?))|
  d(?:[dlt]|el|fn|i[rv])|
  em|
  f(?:ieldset|o(?:nt|rm)|rame(?:set)?)|
  h(?:[1-6r]|ead|tml)|
  i(?:frame|mg|n(?:put|s)|sindex)?|
  kbd|
  l(?:abel|egend|i(?:nk)?)|
  m(?:ap|e(?:nu|ta))|
  no(?:frames|script)|
```

```
  o(?:bject|l|p(?:tgroup|tion))|
  p(?:aram|re)?|
  q|
  s(?:amp|cript|elect|mall|pan|t(?:rike|rong|yle)|u[bp])?|
  t(?:able|body|[dhrt]|extarea|foot|head|itle)|
  ul?|
  var
) \b # Não permite correspondências parciais de nomes
(?: [^>'"] # Qualquer caractere, exceto >, " ou '
  | "[^"]*" # Valor de atributo entre aspas duplas
  | '[^']*' # Valor de atributo entre aspas simples
)*
>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Se você quiser enlouquecer quanto ao uso de espaços em branco, veja outra forma de escrever a mesma expressão regular, no modo de espaçamento livre, que pode facilitar ainda mais a leitura:

```
<
/? # Permite tags de fechamento
( # Captura o nome da tag na retrorreferência 1
  a (?: bbr #
      | cronym #
      | ddress #
      | pplet #
      | rea #
    )?| # Grupo opcional (permite <a>)
  b (?: ase (?:font)? # <base>, <basefont>
      | do #
      | ig #
      | lockquote #
      | ody #
      | r #
      | utton #
    )?| # Grupo opcional (permite <b>)
  c (?: aption #
      | enter #
      | ite #
      | o (?:de|l(?:group)?) # <code>, <col>, <colgroup>
    ) | #
  d (?: [dlt] # <dd>, <dl>, <dt>
```

```
    | el #
    | fn #
    | i[rv] # <dir>, <div>
  ) | #
em | #
f (?: ieldset #
    | o (?:nt|rm) # <font>, <form>
    | rame (?:set)? # <frame>, <frameset>
  ) | #
h (?: [1-6r] # <h1>, <h2>, <h3>, <h4>, <h5>, <h6>, <hr>
    | ead #
    | tml #
  ) | #
i (?: frame #
    | mg #
    | n (?:put|s) # <input>, <ins>
    | sindex #
  )?| # Grupo opcional (permite <i>)
kbd | #
l (?: abel #
    | egend #
    | i (?:nk)? # <li>, <link>
  ) | #
m (?: ap #
    | e (?:nu|ta) # <menu>, <meta>
  ) | #
no (?: frames #
    | script #
  ) | #
o (?: bject #
    | l #
    | p (?:tgroup|tion) # <optgroup>, <option>
  ) | #
p (?: aram #
    | re #
  )?| # Optional group (allow <p>)
q | #
s (?: amp #
    | cript #
    | elect #
    | mall #
    | pan #
    | t (?:rike|rong|yle) # <strike>, <strong>, <style>
```

```
      | u[bp] # <sub>, <sup>
    )?| # Grupo opcional (permite <s>)
  t (?: able #
      | body #
      | [dhrt] # <td>, <th>, <tr>, <tt>
      | extarea #
      | foot #
      | head #
      | itle #
    ) | #
  ul? | # <u>, <ul>
  var #
) \b # Não permite correspondências parciais de nomes
(?: [^>"'] # Qualquer caractere, exceto >, " ou '
  | "[^"]*" # Valor de atributo entre aspas duplas
  | '[^']*' # Valor de atributo entre aspas simples
)*
>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Se estiver trabalhando com XHTML, observe que, embora o XHTML 1.0 não adicione novas tags, ele remove as 14 seguintes: <applet>, <basefont>, , <dir>, <font>, <frame>, <frameset>, <iframe>, <isindex>, <menu>, <noframes>, <s>, <strike> e <u>.

O XHTML 1.1 mantém todos os elementos do XHTML 1.0 e adiciona seis novos (todos relacionados com o texto em Ruby, para idiomas asiáticos): <rb>, <rbc>, <rp>, <rt>, <rtc> e <ruby>. Criar expressões regulares dedicadas a corresponder apenas a elementos XHTML 1.0 ou 1.1 válidos é um exercício que vamos deixar para você.

Veja também:

Corresponder a todas as tags pode ser útil, mas também é comum querer corresponder a uma, ou a algumas tags dentre todas as existentes; a receita 8.2 mostra como realizar essas duas tarefas.

A receita 8.4 descreve os caracteres que podem ser utilizados em nomes de elementos e atributos XML válidos.

8.2 Substituir tags <b> por

Problema

Você deseja substituir, em uma string, todas as tags <b> de abertura e fechamento pelas tags correspondentes, preservando os atributos existentes.

Solução

Esta expressão regular corresponde a tags <b> de abertura e fechamento, com ou sem atributos:

```
<(/?)b\b((?:[^>"']|"[^"]*"|'[^']*')*)>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

No modo de espaçamento livre:

```
< #
(/?) # Captura a barra inicial opcional na retrorreferência 1
b \b # Nome completo da tag, com extremidade de palavra
( # Captura quaisquer atributos etc. na retrorreferência 2
    (?: [^>"'] # Qualquer caractere, exceto >, " ou '
      | "[^"]*" # Valor de atributo entre aspas duplas
      | '[^']*' # Valor de atributo entre aspas simples
   )* #
) #
> #
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, espaçamento livre
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Para preservar todos os atributos, ao alterar o nome da tag, utilize o seguinte texto de substituição:

```
<$1strong$2>
```

Sabores de texto de substituição: .NET, Java, JavaScript, Perl, PHP

```
<\1strong\2>
```

Sabores de texto de substituição: Python, Ruby

Se quiser descartar todos os atributos no mesmo processo, omita a retrorreferência 2, no texto de substituição:

```
<$1strong>
```

Sabores de texto de substituição: .NET, Java, JavaScript, Perl, PHP

<\1strong>

Sabores de texto de substituição: Python, Ruby

A receita 3.15 mostra o código necessário para implementar esta ocorrência.

Discussão

A receita anterior incluía uma discussão detalhada das maneiras de se corresponder a *qualquer* tag no estilo XML. Esse fato libera esta receita para focar em uma abordagem simples, que busque um tipo específico de tag. <b> e sua substituição são oferecidos como exemplos, mas você pode substituir estes nomes de tags por quaisquer outros.

A expressão regular começa correspondendo a um ‹<› literal – o primeiro caractere de qualquer tag. Em seguida, opcionalmente, corresponde à barra encontrada nas tags de fechamento, usando ‹/?› dentro de parênteses de captura. Capturar o resultado deste padrão (que será uma string vazia, ou uma barra) nos permite, facilmente, restaurar a barra na string de substituição, sem qualquer lógica condicional.

Em seguida, correspondemos ao nome da tag em si, <b>. Você poderia usar qualquer outro nome de tag, se quisesse. Estamos usando a opção de não-distinção entre maiúsculas e minúsculas, para nos certificarmos de que também corresponderemos a um B maiúsculo.

A extremidade de palavra (‹\b›), que segue o nome da tag, é fácil de esquecer, mas é uma das partes mais importantes desta expressão regular. A extremidade de palavra nos permite corresponder apenas a tags <b>, e não a
, <body>, <blockquote> ou a quaisquer outras tags que comecem com a letra “b”. Poderíamos, alternativamente, corresponder a um token de espaço em branco (‹\s›) após o nome, como uma proteção contra este problema, mas isso não funcionaria para tags que não possuam atributos e que, portanto, não podem ter

nenhum espaço em branco após seu nome de tag. A extremidade de palavra resolve este problema de forma simples e elegante.

Ao trabalhar com o XML e o XHTML, esteja ciente de que o sinal de dois pontos usado em namespaces, bem como hífens e alguns outros caracteres permitidos como partes de nomes XML, criam uma extremidade de palavra. Por exemplo, a expressão regular pode acabar correspondendo a algo como <b-sharp>. Se estiver preocupado com isso, use um lookahead ‹(?=[\s/>])›, em vez de uma extremidade de palavra. Ele chega ao mesmo resultado, ao assegurar que não corresponderemos a nomes parciais de tags, e o faz de forma mais confiável.

Após o nome da tag, o padrão ‹((?:[^>'"]|"[^"]*"|'[^']*')*)› é usado para corresponder a qualquer coisa que tenha sobrado dentro da tag, até o colchete angular direito. Encapsular esse padrão em um grupo de captura, como fizemos aqui, nos permite facilmente trazer de volta quaisquer atributos e outros caracteres (como a barra final de tags singulares), em nossa string de substituição. Dentro dos parênteses de captura, o padrão repete um grupo de não captura com três alternativas. A primeira, ‹[^>'"]›, corresponde a qualquer caractere único, exceto >, " ou '. As duas alternativas restantes correspondem a uma string inteira, colocada entre aspas duplas ou simples, o que nos permite corresponder aos valores dos atributos que contenham colchetes angulares direitos, sem fazer a expressão regular pensar ter encontrado o final da tag.

Variações

Substituir uma lista de tags

Se quiser corresponder a qualquer tag de uma lista de nomes de tags, será necessário fazer uma mudança simples. Coloque todos os nomes de tags desejados dentro de um grupo e alterne entre eles. Colocar os nomes em um grupo limita o alcance do metacaractere de alternância (‹|›).

A expressão regular a seguir corresponde às tags de abertura e fechamento de <b>, <i>, e <big>. O texto de substituição, mostrado posteriormente, substitui todas elas por uma tag ou correspondente, preservando os atributos:

```
<(/?)([bi]|em|big)\b((?:[^>'"]|"[^"]*"|'[^']*')*)>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Vejamos a mesma expressão regular no modo de espaçamento livre:

```
< #
(/?) # Captura a barra inicial opcional na retrorreferência 1
([bi]|em|big) \b # Captura o nome da tag na retrorreferência 2
( # Captura quaisquer atributos etc. na retrorreferência 3
  (?: [^>'"] # Qualquer caractere, exceto >, " ou '
    | "[^"]*" # Valor de atributo entre aspas duplas
    | '[^']*' # Valor de atributo entre aspas simples
  )* #
) #
> #
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, espaçamento livre
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Nós usamos a classe de caracteres ‹[bi]› para corresponder tanto a tags <b> quanto a tags <i>, ao invés de separá-las com o metacaractere de alternância (|), como fizemos em e <big>. Classes de caracteres executam mais rápido do que as de alternância, já que elas não utilizam retrocesso para conseguir realizar o trabalho. Quando a diferença entre as duas opções for um único caractere, utilize uma classe de caracteres.

Nós também adicionamos um grupo de captura para o nome da tag, deslocando o grupo que corresponde aos atributos, e o fazendo armazenar a correspondência na retrorreferência 3. Embora não haja a necessidade de se referir ao nome da tag, caso você só esteja substituindo todas as correspondências por tags , armazenar o nome da tag em sua própria retrorreferência pode ajudá-lo a verificar o tipo de tag que foi correspondida, quando necessário.

Para preservar todos os atributos ao substituir o nome da tag, utilize o seguinte texto de substituição:

```
<$1strong$3>
```

Sabores de texto de substituição: .NET, Java, JavaScript, Perl, PHP

```
<\1strong\3>
```

Sabores de texto de substituição: Python, Ruby

Omita a retrorreferência 3, na string de substituição, se quiser descartar os atributos das tags correspondidas como parte do mesmo processo:

<$1strong>

Sabores de texto de substituição: .NET, Java, JavaScript, Perl, PHP

<\1strong>

Sabores de texto de substituição: Python, Ruby

Veja também:

A receita 8.1 mostra como corresponder a todas as tags no estilo XML, enquanto equilibra vantagens e desvantagens, incluindo a tolerância a marcações inválidas.

A receita 8.3 é o oposto desta receita, e mostra como corresponder a todas as tags, exceto por uma seleta lista.

8.3 Remover todas as tags de estilo XML, exceto e

Problema

Você deseja remover todas as tags em uma string, exceto e .

Em um caso separado, você não só deseja remover todas as tags que não sejam e , mas também as tags e que contenham atributos.

Solução

Este é um cenário perfeito para colocar em uso o lookahead negativo (explicada na receita 2.16). Aplicado a este problema, o lookahead negativo permite que você corresponda a algo que se pareça com uma tag, *exceto* quando certas palavras vêm logo após o <, ou o </ de abertura. Se você substituir todas as correspondências por uma string vazia (a receita 3.14 mostra como),

somente as tags aprovadas serão deixadas para trás.

Solução 1: Corresponder a tags, exceto e

```
</?(?!(?:em|strong)\b)[a-z](?:[^>'"]|"[^"]*"|'[^']*')*>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

No modo espaçamento livre:

```
< /? # Permite tags de fechamento
(?! # Lookahead negativo
    (?: em | strong ) # Lista de tags em que deve se evitar a correspondência
    \b # A extremidade de palavra evita correspondências parciais de palavras
) #
[a-z] # Caractere inicial do nome da tag deve ser a-z
(?: [^>'"] # Qualquer caractere, exceto >, " ou '
  | "[^"]*" # Valor de atributo entre aspas duplas
  | '[^']*' # Valor de atributo entre aspas simples
)* #
> #
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, espaçamento livre
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Solução 2: Corresponder a tags, exceto e , e a quaisquer tags que contenham atributos

Com uma mudança (substituir o ‹\b› por ‹\s*>›), você pode fazer a expressão regular corresponder também a qualquer tag e que contenha atributos:

```
</?(?!(?:em|strong)\s*>)[a-z](?:[^>'"]|"[^"]*"|'[^']*')*>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Mais uma vez, a mesma expressão regular no modo espaçamento livre:

```
< /? # Permite tags de fechamento
(?! # Lookahead negativo
    (?: em | strong ) # Lista de tags em que deve se evitar a correspondência
    \s* > # Evite as tags apenas se não contiverem atributos
```

```
) #
[a-z] # inicial do nome da tag deve ser a-z
(?: [^>'"] # Qualquer caractere, exceto >, " ou '
  | "[^"]*" # Valor de atributo entre aspas duplas
  | '[^']*' # Valor de atributo entre aspas simples
)* #
> #
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, espaçamento livre
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Discussão

As expressões regulares desta receita possuem muito em comum com as que incluímos no início deste capítulo, para a correspondência de tags em estilo XML. Independentemente do lookahead negativo adicionado para evitar que algumas tags sejam corrrespondidas, estas expressões regulares são quase equivalentes às expressões regulares “tags (X)HTML (permissiva)”, da receita 8.1. A outra diferença principal, aqui, é que não estamos capturando o nome da tag na retrorreferência 1.

Então, vamos olhar mais de perto o que há de novo nesta receita. A solução 1 nunca corresponde a tags ou , independentemente de ter algum atributo, mas corresponde a todas as outras tags. A solução 2 corresponde a todas as mesmas tags da solução 1 e, adicionalmente, corresponde às tags e , que contenham um ou mais atributos. A tabela 8.2 mostra alguns exemplos de strings de assunto que ilustram essa questão.

Tabela 8.2 – Alguns exemplos de strings de assunto

String de assunto	Solução 1	Solução 2
	Corresponde	Corresponde
	Corresponde	Corresponde
	Corresponde	Corresponde
	Não corresponde	Não corresponde
	Não corresponde	Não corresponde
	Não corresponde	Corresponde

Como o objetivo destas expressões regulares é substituir as

correspondências por strings vazias (ou seja, remover as tags), a solução 2 é menos propensa ao abuso das tags e permitidas, que forneceriam uma formatação inesperada ou outras bobagens.

Esta receita tem (até agora) intencionalmente evitado o termo “lista branca” (whitelist) ao descrever como apenas algumas tags são deixadas no lugar, uma vez que essa palavra possui conotações de segurança. Há uma enorme variedade de maneiras de contornar as restrições deste padrão, utilizando strings especialmente criadas com HTML malicioso. Se você estiver preocupado com HTML malicioso e ataques de scripting entre-sites (XSS), a melhor aposta seria converter todos os caracteres <, > e & para suas referências de entidade de caracteres correspondentes (<, > e &) e, em seguida, trazer de volta as tags que sabemos serem seguras (desde que não contenham atributos, ou apenas usem os que estejam dentro de uma lista seleta de atributos aprovados). style é um exemplo de um atributo inseguro, uma vez que alguns navegadores permitem que você insira código de linguagem de script em seu CSS. Por exemplo, para trazer as tags de volta, depois de substituir <,>, e & por referências de entidade, você pode fazer a pesquisa usando a expressão regular ‹<(/?)em>› e substituir as correspondências por «<$1em>» (ou, em Python e Ruby, «<\1em>»).

Variações

Atributos específicos na lista branca

Considere estas novas necessidades: você precisa corresponder a todas as tags, exceto <a>, e , com duas exceções. Qualquer tag <a> que possua atributos que não sejam href ou title deve ser correspondida; se as tags ou possuirem qualquer atributo, corresponda a elas também. Todas as strings correspondidas serão removidas.

Em outras palavras, você deseja remover todas as tags, com exceção das que estão em sua lista branca (<a>, , e). Os únicos atributos na lista branca são href e title, e são permitidos somente dentro de tags <a>. Se um atributo fora da lista branca aparecer em qualquer tag, a tag inteira deverá ser removida.

Vejamos uma expressão regular que pode fazer o trabalho, mostrada com e sem o modo livre-spacing:

```
<(?!(?:em|strong|a(?:\s+(?:href|title)\s*=\s*(?:"[^"]*"|'[^']*'))*)\s*>)↲
[a-z](?:[^>'"]|"[^"]*"|'[^']*')*>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
< /? # Permite tags de fechamento
(?! # Lookahead negativo
    (?: em # Não corresponde a <em>...
      | strong # ou <strong>...
      | a # ou <a>...
        (?: # Evitar corresponder a tags <a> apenas quando se
                    # limitarem aos...
          \s+ # atributos href e/ou title
          (?:href|title)
          \s*=\s*
          (?:"[^"]*"|'[^']*') # Valor de atributo entre aspas simples ou duplas
        )*
    )
    \s* > # Evitar corresponder a essas tags apenas quando se
                    # limitarem aos...
) # atributos listados acima
[a-z] # inicial do nome da tag deve ser a-z
(?: [^>"'] # Qualquer caractere, exceto >, " ou '
  | "[^"]*" # Valor de atributo entre aspas duplas
  | '[^']*' # Valor de atributo entre aspas simples
)*
>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, espaçamento livre
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Este procedimento nos leva ao limite de até onde faz sentido usar uma expressão regular tão complicada como essa. Se suas regras ficarem mais complexas, provavelmente seria melhor escrever um código baseado nas receitas 3.11 ou 3.16, que verifique o valor de cada tag correspondida, para determinar a forma de processá-la (com base no nome da tag, nos atributos incluídos ou no que mais for necessário).

Veja também:

A receita 8.1 mostra como corresponder a todas as tags de estilo XML, enquanto equilibra vantagens e desvantagens, incluindo a tolerância a marcações inválidas.

A receita 8.2 é o oposto desta receita, e mostra como corresponder a uma seleta lista de tags.

8.4 Corresponder a nomes XML

Problema

Você deseja verificar se uma string é um nome XML legítimo (uma construção sintática comum). O XML prevê regras precisas para os caracteres que possam ocorrer em um nome, e reutiliza essas regras para nomes de elementos, de atributos e de entidades, alvos de instrução de processamento. Os nomes devem ser compostos por uma letra, underscore ou sinal de dois pontos como primeiro caractere, seguido por qualquer correspondência de letras, dígitos, underscores, dois pontos, hífens e pontos. Trata-se, realmente, de uma descrição aproximada, mas bem próxima. A lista exata de caracteres permitidos depende da versão de XML em uso.

Alternativamente, você pode querer dividir um padrão, para corresponder a nomes válidos dentro de outras expressões regulares que tratem XML, quando uma precisão extra justifica a complexidade adicionada.

A seguir estão alguns exemplos de nomes válidos:

- thing
- _thing_2_
- :Российские-Вещь
- fantastic4:the.thing
- 日本の物

Note que letras de sistemas de escrita não-latinos são permitidas, mesmo incluindo caracteres ideográficos, no último exemplo. Da mesma forma, qualquer dígito Unicode será permitido após o primeiro caractere, não apenas os algarismos arábicos 0-9.

Para comparação, vejamos alguns exemplos de nomes inválidos, que não devem ser combinados com a expressão regular:

- thing!
- thing with spaces

- .thing.with.a.dot.in.front
- -thingamajig
- 2nd_thing

Solução

Tal como identificadores em muitas linguagens de programação, existe um conjunto de caracteres que pode ocorrer em um nome XML, e um subconjunto que pode ser usado como o primeiro caractere. Essas listas de caracteres são radicalmente diferentes para o XML 1.0 Quarta Edição (e anteriores), e para o XML 1.1 e 1.0 Quinta Edição. Essencialmente, nomes XML 1.1 podem usar todos os caracteres permitidos no XML 1.0 Quarta Edição, além de quase um milhão a mais. No entanto, a maioria dos caracteres adicionais é, apenas, posições na tabela Unicode. A maioria ainda não tem um caractere atribuído, sendo permitida para compatibilidade futura, na medida em que o banco de dados de caracteres Unicode se expande.

Por razões de concisão, as referências ao XML 1.0, nesta receita, descrevem da primeira à quarta edição do XML 1.0. Quando falamos em nomes XML 1.1, também estamos descrevendo as regras do XML 1.0 Quinta Edição. A quinta edição só se tornou uma recomendação oficial do W3C no final de novembro de 2008, quase cinco anos após o XML 1.1.

Expressão regulares, nesta receita, são mostradas com as âncoras iniciais e finais de string (usando ‹^...$› ou ‹\A...\Z›). Elas fazem com que sua string de assunto seja correspondida em totalidade, ou que não seja correspondida de forma alguma. Se você deseja incorporar esses padrões em uma expressão regular mais longa, que lide com a correspondência de, digamos, elementos XML, certifique-se de remover as âncoras no início e no final dos padrões exibidos aqui. Âncoras são explicadas na receita 2.5.

Nomes XML 1.0 (aproximada)

```
\A[:_\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nl}][:_\-.\p{L}\p{M}\p{Nd}\p{Nl}]*\Z
```

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, PCRE, Perl, Ruby 1.9

O PCRE deve ser compilado com suporte a UTF-8, para que as propriedades Unicode (‹\p{.}›) funcionem. No PHP, ative o suporte a

UTF-8 com o modificador de padrão /u.

Propriedades Unicode não são suportadas em JavaScript, Python ou Ruby 1.8. As expressões regulares para nomes XML 1.1, mostradas a seguir, não se baseiam em propriedades Unicode e, portanto, poderiam ser uma boa alternativa, caso você esteja usando uma dessas linguagens de programação. Veja o tópico “Discussão” desta receita, para obter detalhes sobre os motivos pelos quais seria melhor usar a solução baseada em XML 1.1, mesmo que seu sabor de expressão regular suporte propriedades Unicode.

Nomes XML 1.1 (exata)

A seguir, mostramos três versões de uma mesma expressão regular, devido a diferenças de sabor. A única diferença, entre as duas primeiras, são as âncoras utilizadas no início e no final dos padrões. A terceira versão utiliza ‹\x{.}›, em vez de ‹\u›, para especificar os pontos de código Unicode maiores que o hexadecimal FF (decimal 255).

```
\A[:_A-Za-z\xC0-\xD6\xD8-\xF6\xF8-\u02FF\u0370-\u037D\u037F-
\u1FFF\u200C↲
\u200D\u2070-\u218F\u2C00-\u2FEF\u3001-\uD7FF\uF900-\uFDCF\uFDF0-
\uFFFD]↲
[:_\-.A-Za-z0-9\xB7\xC0-\xD6\xD8-\xF6\xF8-\u036F\u0370-\u037D\u037F-
\u1FFF↲
\u200C\u200D\u203F\u2040\u2070-\u218F\u2C00-\u2FEF\u3001-
\uD7FF\uF900-↲
\uFDCF\uFDF0-\uFFFD]*\Z
```

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, Python, Ruby 1.9

```
^[:_A-Za-z\xC0-\xD6\xD8-\xF6\xF8-\u02FF\u0370-\u037D\u037F-
\u1FFF\u200C↲
\u200D\u2070-\u218F\u2C00-\u2FEF\u3001-\uD7FF\uF900-\uFDCF\uFDF0-
\uFFFD]↲
[:_\-.A-Za-z0-9\xB7\xC0-\xD6\xD8-\xF6\xF8-\u036F\u0370-\u037D\u037F-
\u1FFF↲
\u200C\u200D\u203F\u2040\u2070-\u218F\u2C00-\u2FEF\u3001-
```

\uD7FF\uF900-↲
\uFDCF\uFDF0-\uFFFD]*$

Opções Regex: Nenhuma (“^ e $ correspondem em quebras de linha” não deve estar definida)

Sabores Regex: .NET, Java, JavaScript, Python

```
\A[:_A-Za-z\xC0-\xD6\xD8-\xF6\xF8-\x{2FF}\x{370}-\x{37D}\x{37F}-\x{1FFF}↲
\x{200C}\x{200D}\x{2070}-\x{218F}\x{2C00}-\x{2FEF}\x{3001}-\x{D7FF}↲
\x{F900}-\x{FDCF}\x{FDF0}-\x{FFFD}][:_\-.A-Za-z0-9\xB7\xC0-\xD6\xD8-\xF6↲
\xF8-\x{36F}\x{370}-\x{37D}\x{37F}-\x{1FFF}\x{200C}\x{200D}\x{203F}↲
\x{2040}\x{2070}-\x{218F}\x{2C00}-\x{2FEF}\x{3001}-\x{D7FF}\x{F900}-↲
\x{FDCF}\x{FDF0}-\x{FFFD}]*\Z
```

Opções Regex: Nenhuma

Sabores Regex: PCRE, Perl

O PCRE deve ser compilado com suporte a UTF-8, para que as metassequências ‹\x{.}› funcionem com valores superiores ao hexadecimal FF. No PHP, ative o suporte a UTF-8 com o modificador de padrão /u.

O Ruby 1.8 não oferece suporte a expressões regulares Unicode; consulte a seção “Variações” desta receita, para uma solução alternativa possível, e menos precisa.

Embora tenhamos reivindicado que essas expressões regulares sigam exatamente as regras de nomes XML 1.1, tal exigência só é verdadeira para os caracteres com até 16 bits (posições entre 0x0 e 0xFFFF). Além destes, o XML 1.1 permite que os 917.503 pontos de código, entre as posições 0x10000 e 0xEFFFF, ocorram após o caractere inicial do nome. No entanto, apenas PCRE, Perl e Python são capazes de referenciar pontos de código além de 0xFFFF, e é improvável que você encontre qualquer um deles na vida real (para começar, a maioria das posições desse intervalo nem receberam a atribuição de um caractere real). Se precisar adicionar suporte para esses pontos extras de código, em PCRE e Perl você poderia adicionar ‹\x{10000}-\x{EFFFF}› no final da segunda classe de caracteres, e, em Python, você poderia adicionar ‹\U00010000-\U000EFFFF› (note o U maiúsculo, que deve ser seguido por oito

dígitos hexadecimais). Porém, mesmo sem acrescentar este intervalo maciço, a lista de caracteres do XML 1.1, que acabamos de mostrar, é muito mais permissiva do que a do XML 1.0.

Discussão

Como muitas das expressões regulares neste capítulo lidam com correspondências de elementos XML, grande parte desta receita serve para fornecer uma ampla discussão a respeito dos padrões que podem ser usados, caso você queira ser bastante específico sobre como nomes de tags e atributos são correspondidos. Em outros lugares, nos atemos principalmente aos padrões mais simples e menos precisos, tendo por interesse a clareza e a eficiência.

Então, vamos cavar um pouco mais fundo, estudando as regras que existem por trás desses padrões.

Nomes XML 1.0

A especificação XML 1.0 utiliza uma abordagem de lista branca (whitelist) para suas regras de nome, listando explicitamente todos os caracteres permitidos. O caractere inicial do nome pode ser um sinal de dois pontos (:), underscore (_), ou qualquer caractere nas seguintes categorias Unicode:

- Letra minúscula (Ll).
- Letra maiúscula (Lu).
- Letra de capitalização (Lt).
- Letra sem variação de maiúscula e minúscula (Lo).
- Letra-numeral (Nl).

Após o caractere inicial, hífens (-), pontos (.), qualquer outro caractere, nas seguintes categorias, é permitido, além dos já mencionados:

- Marca (M), que combina as subcategorias de marca que não ocupam espaço (Mn), marca de combinação de espaço (Mc) e marca de encapsulamento (Me).

- Modifier letter (Lm).
- Decimal digit (Nd).

Estas regras nos levam à expressão regular mostrada na “Solução” desta receita.

Vamos vê-la, novamente, no modo de espaçamento livre:

```
\A # Início da string
[:_\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nl}] # Caractere inicial do nome
[:_\-.\p{L}\p{M}\p{Nd}\p{Nl}]* # Caracteres do nome (zero ou mais)
\Z # Fim da string
```

Opções Regex: Espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Ruby 1.9

Novamente, o PCRE deve ser compilado com suporte a UTF-8. No PHP, ative o suporte a UTF-8 com o modificador de padrão `/u`.

Observe que, na segunda classe de caracteres, todas as subcategorias de letra (Ll, Lu, Lt, Lo e LM) foram combinadas em sua categoria de base, usando `‹\p{L}›`.

Anteriormente, observamos que as regras aqui descritas são aproximadas. Existem algumas razões para isso. Primeiramente, a especificação XML 1.0 (lembre-se de que não estamos falando da quinta edição, e versão posterior) enumera uma série de exceções a estas regras. Em segundo lugar, as listas de caracteres do XML 1.0 foram derivadas do Unicode 2.0, lançado em 1996. Versões posteriores do padrão Unicode adicionaram suporte a uma variedade de novos sistemas de escrita, cujos caracteres não são permitidos pelas regras do XML 1.0. No entanto, desacoplar a expressão regular de qualquer versão Unicode que seu mecanismo de expressão regular utilize, de forma a restringir as correspondências a caracteres Unicode 2.0, transformaria esse padrão em uma monstruosidade do tamanho de uma página, preenchida com centenas de intervalos e pontos de código. Se você realmente quiser criar esse monstro, consulte o XML 1.0 Quarta Edição (*http://www.w3.org/TR/2006/REC-xml-20060816*), seção 2.3, “Common Syntactic Constructs”, e Apêndice B, “Character Classes”.

A seguir, apresentamos várias maneiras específicas, de cada sabor, para encurtar a expressão regular já vista.

Perl e PCRE permitem combinar as subcategorias de letras minúsculas (Ll), letras maiúsculas (Lu) e letras de capitalização (Lt), na categoria especial “letras com diferenciação entre maiúsculas e minúsculas” (L&). Estes sabores de expressão regular também permitem que você omita as chaves na sequência de escape ‹\p{...}›, se apenas uma letra for usada internamente. Nós aproveitamos esse fato na seguinte expressão regular, utilizando ‹\pL\pM›, em vez de ‹\p{L}\p{M}›:

```
\A[:_\p{L&}\p{Lo}\p{Nl}][:_\-.\pL\pM\p{Nd}\p{Nl}]*\Z
```

Opções Regex: Nenhuma
Sabores Regex: PCRE, Perl

O .NET suporta a subtração de classes de caracteres, usada aqui, na primeira classe de caracteres, para subtrair a subcategoria Lm de L, ao invés de listar explicitamente todas as outras subcategorias de letras:

```
\A[:_\p{L}\p{Nl}-[\p{Lm}]][:_\-.\p{L}\p{M}\p{Nd}\p{Nl}]*\Z
```

Opções Regex: Nenhuma
Sabores Regex: .NET

O Java, assim como o PCRE e o Perl, permite que você omita as chaves em torno de categorias Unicode de uma letra. A seguinte expressão regular também tira proveito da versão Java mais complicada para a subtração da classe de caracteres (implementada por meio de uma intersecção com uma classe negada), com o intuito de subtrair a subcategoria Lm de L:

```
\A[:_\pL\p{Nl}&&[^\p{Lm}]][:_\-.\pL\pM\p{Nd}\p{Nl}]*\Z
```

Opções Regex: Nenhuma
Sabores Regex: Java

JavaScript, Python e Ruby 1.8 não suportam categorias Unicode. O Ruby 1.9 não possui as funcionalidades que acabamos de descrever, mas suporta a versão mais portátil destas expressões regulares, mostrada na “Solução”desta receita.

Nomes XML 1.1

O XML 1.0 cometeu o erro de se vincular explicitamente ao Unicode 2.0. Versões posteriores do padrão Unicode adicionaram suporte para mais caracteres, alguns dos quais são de sistemas de escrita que, anteriormente, não haviam sido contabilizados (por exemplo, Cherokee, Etíope e Mongol). Como o XML quer ser considerado um formato universal, ele tentou corrigir este problema com o XML 1.1 e 1.0 Quinta Edição. Estas versões posteriores mudaram de uma abordagem de lista branca para uma de lista negra, a fim de suportarem não apenas os caracteres adicionados desde o Unicode 2.0, mas também os que poderão ser adicionados no futuro.

A nova estratégia, permitindo qualquer coisa que não seja explicitamente proibida, melhora a compatibilidade futura, e também torna mais fácil e menos verboso seguir as regras. É por isso que as expressões regulares nomeadas como XML 1.1 são rotuladas como exatas, ao passo que a expressão regular XML 1.0 é aproximada.

Variações

Em algumas das receitas deste capítulo (por exemplo, receita 8.1), os segmentos de padrão que lidam com nomes XML não empregam restrições, ou proibições, de sistemas de escrita estrangeiros e de outros caracteres que sejam, de fato, perfeitamente válidos. Assim procedemos para manter as coisas simples. No entanto, se você quiser permitir sistemas de escrita estrangeiros, enquanto continua a fornecer um nível básico de restrições (você não precisa da validação de nomes mais precisa das expressões regulares no início desta receita), as expressões regulares a seguir podem fazer esse truque.

Deixamos as âncoras de início e final da string fora destas expressões regulares, pois tais expressões não foram feitas para serem usadas por conta própria, mas como partes de padrões mais longos.

Esta primeira expressão regular simplesmente evita a correspondência dos caracteres usados como separadores e delimitadores dentro de tags XML e, além disso, impede a

correspondência de um dígito como primeiro caractere:

```
[^\d\s"'/<=>][^\s"'/<=>]*
```

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

A seguir temos outra expressão regular, uma maneira mais curta de fazer a mesma coisa. Em vez de usar duas classes de caracteres distintas, ela usa um lookahead negativo, para proibir um dígito como caractere inicial. Esta proibição aplica-se somente ao primeiro caractere correspondido, mesmo que o quantificador ‹+›, após a classe de caracteres, permita que a expressão regular corresponda a um número ilimitado de caracteres:

```
(?!\d)[^\s"'/<=>]+
```

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Veja também:

John Cowan, um dos editores da especificação XML 1.1, explica quais caracteres são proibidos em nomes XML 1.1, e por que são proibidos, em um artigo no endereço *http://recycledknowledge.blogspot.com/2008/02/which-characters-are-excluded-in-xml.html*.

O documento “Background to Changes in XML 1.0, 5th Edition”, em *http://www.w3.org/XML/2008/02/xml10_5th_edition_background.html* , discute a justificativa para retroportar as regras de nomes do XML 1.1 para o XML 1.0 Quinta Edição.

8.5 Converter texto simples em HTML adicionando tags <p> e

Problema

Dada uma string de texto simples, como um valor multilinha enviado por meio de um formulário, você deseja convertê-la em um fragmento HTML, para exibi-la em uma página web. Parágrafos,

separados por duas quebras de linha sucessivas, devem ser cercados por ... . Quebras de linha adicionais devem ser substituídas por tags
.

Solução

Este problema pode ser resolvido em quatro etapas simples. Na maioria das linguagens de programação, apenas as duas etapas intermediárias beneficiam-se das expressões regulares.

Passo 1: Substituir caracteres especiais HTML por referências a entidades de caractere

Como estamos convertendo texto simples em HTML, o primeiro passo é converter os três caracteres especiais HTML &, < e › em referências a entidades de caractere (Tabela 8.3). Caso contrário, a marcação resultante poderá ter resultados indesejados, ao ser exibida em um navegador web.

Tabela 8.3 – Substituições de caracteres especiais HTML

Busque por	Substitua por
‹&›	‹‹&››
‹<›	‹‹<››
‹>›	‹‹>››

Os sinais de conjunção (&) devem ser substituídos primeiro, uma vez que você incluirá sinais de conjunção adicionais à string de assunto, como parte das referências a entidades de caractere.

Passo 2: Substituir todas as quebras de linha por

Pesquise por:

\r\n?|\n

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

\R

Opções Regex: Nenhuma

Sabores Regex: PCRE 7, Perl 5.10

Substitua por:

```
<br>
```

Sabores de texto de substituição: .NET, Java, JavaScript, Perl, PHP, Python, Ruby

Passo 3: Substituir tags duplas
 por

Pesquise por:

```
<br>\s*<br>
```

Opções Regex: Nenhuma

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Substitua por:

```
</p><p>
```

Sabores de texto de substituição: .NET, Java, JavaScript, Perl, PHP, Python, Ruby

Passo 4: Envolver toda a string com ...

Este passo é uma simples concatenação de strings, e não requer expressões regulares.

Exemplo JavaScript

Ao juntarmos todos os quatro passos, criamos uma função JavaScript chamada `html_from_plaintext`. Esta função aceita uma string, processa-a, utilizando os passos que acabamos de descrever e, em seguida, retorna a nova string HTML:

```
function html_from_plaintext (subject) {
  // passo 1 (pesquisas por texto simples)
  subject = subject.replace(/&/g, "&amp;").
                    replace(/</g, "&lt;").
                    replace(/>/g, "&gt;");

  // passo 2
  subject = subject.replace(/\r\n?|\n/g, "<br>");

  // passo 3
  subject = subject.replace(/<br>\s*<br>/g, "</p><p>");

  // passo 4
  subject = "<p>" + subject + "</p>";

  return subject;
```

```
}
/*
html_from_plaintext("Test.") -> "<p>Test.</p>"
html_from_plaintext("Test.\n") -> "<p>Test.<br></p>"
html_from_plaintext("Test.\n\n") -> "<p>Test.</p><p></p>"
html_from_plaintext("Test1.\nTest2.") -> "<p>Test1.<br>Test2.</p>"
html_from_plaintext("Test1.\n\nTest2.") -> "<p>Test1.</p><p>Test2.</p>"
html_from_plaintext("< AT&T >") -> "<p>&lt; AT&amp;T &gt;</p>"
*/
```

Vários exemplos estão incluídos no final do trecho de código, mostrando sua saída, quando esta função é aplicada a várias strings de assunto. Se o JavaScript for um estranho para você, observe que o modificador `/g`, anexado a cada uma das expressões regulares literais, faz o método `replace` substituir todas as ocorrências do padrão, em vez de apenas a primeira. A metassequência `\n`, nas strings-exemplo de assunto, insere um caractere de alimentação de linha (posição ASCII 0x0A) em uma string literal do JavaScript.

Discussão

Passo 1: Substituir caracteres especiais HTML por referências a entidades de caractere

A maneira mais fácil de concluir esta etapa é usando três operações distintas de pesquisa-e-substituição (consulte a tabela 8.3, mostrada anteriormente, para ver a lista de substituições). O JavaScript sempre utiliza expressões regulares para operações globais de pesquisa-e-substituição, mas em outras linguagens de programação você, normalmente, obtém melhor desempenho em substituições de texto simples.

Passo 2: Substituir todas as quebras de linha por

Nesta etapa, usamos a expressão regular `‹\r\n?|\n›` para encontrar as quebras de linha que seguem as convenções do Windows/MS-DOS (CRLF), Unix/Linux/OS X (LF) e do Mac OS (CR). Usuários de Perl

5.10 e PCRE 7 podem usar o token dedicado ‹\R› (observe o R maiúsculo), em vez de corresponder a estas e outras sequências de quebra de linha.

Substituir todas as quebras de linha por
, antes de adicionar tags de parágrafo na próxima etapa, mantém as coisas mais simples, já que isso lhe dá a opção de adicionar espaço entre suas tags em substituições posteriores. Fazer isso pode ajudar a manter seu código HTML legível, pois ele não ficará tão compacto.

Se preferir usar tags singulares no estilo XHTML, utilize «<br□/>», em vez de «
» como texto de substituição. Você também vai precisar alterar a expressão regular, na Etapa 3, para corresponder a esta mudança.

Passo 3: Substituir tags
 duplas por

Duas quebras de linha em sequência indicam o fim de um parágrafo e o início de outro; então, nosso texto de substituição, para esta etapa, é uma tag de fechamento , seguida por uma tag de abertura . Se o texto de assunto contiver apenas um parágrafo (ou seja, duas quebras de linha nunca aparecem em sequência), nenhuma substituição será feita. A etapa 2 já substituiu várias convenções de quebra de linha (deixando apenas tags
 no lugar dessas quebras), de modo que este passo poderia ser tratado por um texto de substituição simples. No entanto, usar uma expressão regular aqui facilita quando formos levar as coisas um passo adiante, ignorando os espaços em branco que aparecerem entre as quebras de linha. De qualquer maneira, caracteres de espaços extras não são apresentados em um documento HTML.

Se você estiver gerando XHTML e, portanto, quebras de linha substituídas por «<br□/>», em vez de «
», precisará ajustar a expressão regular dessa etapa para ‹<br□/>\s*<br□/>›.

Passo 4: Envolver toda a string com ...</ p›

O terceiro passo apenas adicionou marcação entre os parágrafos.

Agora, você precisa adicionar uma tag no início da string de assunto e uma tag de fechamento no final. Isso conclui o processo, independente de haver 1 ou 100 parágrafos de texto.

Veja também:

A receita 4.10 inclui mais informações sobre o token ‹\R› do Perl e do PCRE, e mostra como corresponder manualmente a separadores de linha esotéricos adicionais, suportados por ‹\R›.

8.6 Encontrar um atributo específico em tags no estilo XML

Problema

Você deseja encontrar tags dentro de um arquivo (X)HTML ou XML que contenham um atributo específico, como id.

Esta receita cobre muitas variações a respeito do mesmo problema. Suponha que você queira corresponder a cada um dos seguintes tipos de strings, usando expressões regulares em separado:

- Tags que contenham o atributo id.
- Tags que contenham o atributo id.
- Tags que contenham o atributo id com o valor my-id.
- Tags que contenham my-class dentro do valor do atributo class (classes são separadas por espaços em branco).

Solução

Tags que contenham um atributo id (rápido e rasteiro)

Se quiser fazer uma busca rápida em um editor de texto que permita pré-visualizar os resultados, a seguinte expressão regular (excessivamente simplista) pode fazer esse truque:

```
<[^>]+\sid\b[^>]*>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Veja um desmembramento da expressão regular, no modo de espaçamento livre:

```
< # Início da tag
[^>]+ # Nome da tag, atributos etc.
\s id \b # O nome do atributo-alvo, como uma palavra inteira
[^>]* # O restante da tag, o valor do atributo id
> # Final da tag
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, espaçamento livre
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Tags que contenham um atributo id (mais confiável)

Ao contrário da expressão regular mostrada anteriormente, a próxima abordagem do mesmo problema suporta valores de atributos entre aspas que contenham caracteres literais ›; ela não corresponde a tags que simplesmente contenham a palavra id dentro de um de seus valores de atributos:

```
<(?:[^>'"]|"[^"]*"|'[^']*')+?\sid\s*=\s*("[^"]*"|'[^']*')↵
(?:[^>'"]|"[^"]*"|'[^']*')*>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

No modo espaçamento livre:

```
< #
(?: [^>'"] # Nome da tag e dos atributos etc.
  | "[^"]*" # ...e valores de atributos entre aspas
  | '[^']*' #
)+? #
\s id # O nome do atributo-alvo, como uma palavra inteira
\s* = \s* # Separador do par nome/valor do atributo
( "[^"]*" | '[^']*' ) # Captura o valor do atributo na retrorreferência 1
(?: [^>'"] # Quaisquer caracteres remanescentes
  | "[^"]*" # ...e valores de atributos entre aspas
  | '[^']*' #
)* #
> #
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Esta expressão regular captura o valor do atributo id e as aspas em torno do valor na retrorreferência 1. Isso permite que você use o valor em códigos fora da expressão regular ou em uma substituição de string. Se você não precisar reutilizar o valor, pode mudar para um grupo de não-captura ou substituir toda a sequência ‹\s*=\s*("[^"]*"|'[^']*')› por ‹\b›. O restante da expressão regular preenche a lacuna e corresponde ao valor do atributo id.

Tags que contenham um atributo id

Para procurar um tipo de tag específica, você precisa adicionar o nome dela ao início da expressão regular, e fazer pequenas alterações na expressão regular anterior. No exemplo a seguir, nós adicionamos ‹div\s› após a tag de abertura ‹<›. O símbolo ‹\s› (espaço em branco) garante que não corresponderemos a tags cujos nomes apenas comecem com as três letras “div”. Sabemos que haverá um caractere de espaço em branco após o nome da tag, porque as tags que procuramos têm, pelo menos, um atributo (id). Além disso, a sequência ‹+?\sid› foi alterada para ‹*?\bid›, de modo que a expressão regular funcione quando o id for o primeiro atributo dentro da tag, não existindo caracteres separadores adicionais (além do espaço inicial) após o nome da tag:

```
<div\s(?:[^>"']|"[^"]*"|'[^']*')*?\bid\s*=\s*("[^"]*"|'[^']*')↲
(?:[^>"']|"[^"]*"|'[^']*')*>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Vejamos a mesma coisa, no modo de espaçamento livre:

```
<div \s # Nome da tag e caracteres de espaço em branco subsequentes
(?: [^>"'] # Nomes e atributos de tag etc.
  | "[^"]*" # ...e valores de atributos entre aspas
  | '[^']*' #
)*? #
\b id # O nome do atributo-alvo, como uma palavra inteira
\s* = \s* # Separador do par nome/valor do atributo
( "[^"]*" | '[^']*' ) # Captura o valor do atributo na retrorreferência 1
(?: [^>"'] # Quaisquer caracteres remanescentes
```

```
  | "[^"]*" # ... e valores de atributos entre aspas
  | '[^']*' #
)* #
> #
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, espaçamento livre
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Tags que contenham um atributo id com o valor “my-id”

Em comparação com a expressão regular entitulada “Tags que contenham um atributo id (mais confiável)”, desta vez removeremos o grupo de captura em torno do valor do atributo id, pois você já sabe o valor com antecedência. Especificamente, o subpadrão ‹(" [^"]*"|'[^']*')› foi substituído por ‹(?:"my-id"|'my-id')›:

```
<(?:[^>"']|"[^"]*"|'[^']*')+?\sid\s*=\s*(?:"my-id"|'my-id')↵
(?:[^>"']|"[^"]*"|'[^']*')*>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

E seria essa a versão de espaçamento livre:

```
< #
(?: [^>"'] # Nomes e atributos de tag etc.
  | "[^"]*" # ...e valores de atributos entre aspas
  | '[^']*' #
)+? #
\s id # O nome do atributo-alvo, como uma palavra inteira
\s* = \s* # Separador do par nome/valor do atributo
(?: "my-id" # O valor do atributo-alvo
  | 'my-id' ) # ...colocado entre aspas simples ou duplas
(?: [^>"'] # Quaisquer caracteres remanescentes
  | "[^"]*" # ...e valores de atributos entre aspas
  | '[^']*' #
)* #
> #
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, espaçamento livre
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Voltando ao subpadrão ‹(?:"my-id"|'my-id')› por um segundo, alternativamente você poderia evitar a repetição de ”my-id“ (ao custo de alguma eficiência), usando ‹(["'])my-id\1›. Tal procedimento usa um

grupo de captura e uma retrorreferência, para garantir que o valor começará e terminará com o mesmo tipo de aspa.

Tags que contenham “my-class” dentro do valor do atributo class

Se as expressões regulares anteriores ainda não haviam atravessado o limiar, este é o lugar no qual fica óbvio que estamos forçando o limite do que pode ser razoavelmente alcançado utilizando uma única expressão regular. Dividir o processo em várias expressões regulares ajuda um pouco, então vamos dividir essa pesquisa em três partes. A primeira expressão regular corresponderá a tags, a próxima encontrará o atributo class dentro da tag (e armazenará seu valor dentro de uma retrorreferência) e, finalmente, pesquisaremos dentro do valor my-class.

Encontrar tags:

```
<(?:[^>'"]|"[^"]*"|'[^']*')+>
```

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

A receita 8.1 é dedicada a corresponder a tags no estilo XML. Ela explica como a expressão regular mostrada recentemente funciona, fornecendo algumas alternativas com diferentes graus de complexidade e precisão.

Em seguida, siga o código da receita 3.13 para pesquisar, dentro de cada correspondência, por um atributo class usando a seguinte expressão regular:

```
^(?:[^>'"]|"[^"]*"|'[^']*')+?\sclass\s*=\s*("[^"]*"|'[^']*')
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Esse procedimento captura o valor completo de class e suas aspas dentro da retrorreferência 1. Tudo que venha antes do atributo class é correspondido utilizando ‹^(?:[^>'"]|"[^"]*"|'[^']*')+?›, que corresponde aos valores entre aspas em etapas, para evitar encontrar a palavra “class” dentro de outro valor de atributo. No lado direito do padrão, a correspondência termina assim que chegamos ao final do valor do atributo class. Nada depois disso é relevante para a nossa pesquisa;

portanto, não há nenhuma razão para corresponder a tudo, até o final da tag em que estiver fazendo a pesquisa.

O acento circunflexo, no início da expressão regular, a âncora ao início da string de assunto. Isso não muda o que é correspondido, mas existe para certificar que, caso o mecanismo de expressão regular não possa encontrar uma correspondência no início da string, ele não tentará (e, inevitavelmente, falhará) novamente a cada posição de caractere subsequente.

Finalmente, se ambas as expressões regulares anteriores corresponderem com sucesso, você vai querer pesquisar o seguinte padrão dentro da retrorreferência 1 das correspondências da segunda expressão regular:

```
(?:^|\s)my-class(?:\s|$)
```

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Como as classes são separadas por espaços em branco, `my-class` deve ser delimitada, em ambas as extremidades, por um espaço em branco ou por nada. Se não fosse pelo fato de nomes de classe poderem incluir hífens, você poderia, neste caso, usar tokens de extremidade de palavra, em vez dos dois grupos de não-captura. No entanto, hífens criam extremidades de palavra e, assim, `‹\bmy-class\b›` corresponderia dentro de `not-my-class`.

Discussão

A “Solução” desta receita já aborda os detalhes de como funcionam estas expressões regulares, por isso vamos evitar refazer tudo. Lembre-se de que as expressões regulares, muitas vezes, não são a solução ideal para pesquisas de marcação, especialmente as que atinjam uma complexidade como a descrita nesta receita. Antes de utilizar estas expressões regulares, considere se não estaria mais bem servido por uma solução alternativa, como XPath, um analisador SAX ou um DOM. Nós incluímos estas expressões regulares porque não é incomum que as pessoas tentem fazer este tipo de coisa. Porém, não diga que não avisamos. Esperamos que o

estudo pelo menos ajude a mostrar algumas das questões envolvidas em pesquisas de marcação, ajudando-o a evitar soluções ainda mais ingênuas.

Veja também:

A receita 8.7 é o inverso conceitual desta receita, e encontra tags que não contenham um atributo específico.

8.7 Adicionar um atributo cellspacing em tags <table> que ainda não o incluam

Problema

Você deseja pesquisar ao longo de um arquivo (X)HTML, adicionando cellspacing="0" a todas as tabelas que ainda não incluam um atributo cellspacing.

Esta receita serve como um exemplo de como adicionar um atributo a tags no estilo XML que ainda não tenham tal atributo. Você pode trocar o nome da tag, o nome do atributo e o valor aqui apresentados.

Solução

Expressão regular 1: solução simplista

Você pode usar um lookahead negativo, para corresponder a tags <table> que não contenham a palavra cellspacing, como segue:

```
<table\b(?![^>]*?\scellspacing\b)([^>]*)>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Vejamos, novamente, a expressão regular no modo de espaçamento livre:

```
<table \b # Corresponde a "<table", seguido de uma extremidade de palavra
(?! # Declara que a regex abaixo não pode ser correspondida aqui
```

```
  [^>] # Corresponde a qualquer caractere, exceto ">"...
  *? # zero ou mais vezes, o mínimo de vezes possível (preguiçoso)
  \s cellspacing \b # Corresponde a "cellspacing" como uma palavra completa
) #
( # Captura a regex abaixo na retrorreferência 1
  [^>] # Corresponde a qualquer caractere, exceto ">"...
  * # zero ou mais vezes, o máximo de vezes possível (mesquinho)
) #
> # Corresponde a um ">"literal no final da tag
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Expressão regular 2: solução mais confiável

A expressão regular seguinte substitui as duas instâncias da classe de caracteres negada ‹[^>]›, da solução simplista, por ‹(?:[^>"']|"[^"]*"|'[^']*')›. Dessa forma, melhoramos a confiabilidade da expressão regular de duas maneiras. Primeiro, ela adiciona suporte para valores de atributos entre aspas que contenham caracteres “>” literais. Segundo, ela garante que não impediremos a correspondência de tags que apenas contenham a palavra “cellspacing” no valor de um atributo.

Veja como fica a expressão regular com a mudança que acabamos de descrever:

```
<table\b(?!(?:[^>"']|"[^"]*"|'[^']*')*?\scellspacing\b)↵
((?:[^>"']|"[^"]*"|'[^']*')*)>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

E, agora, no modo espaçamento livre:

```
<table \b # Corresponde a "<table", seguido de uma extremidade de palavra
(?! # Declara que a regex abaixo não pode ser correspondida aqui
  (?: [^>"'] # Corresponde a qualquer caractere, exceto >, ", or '
    | "[^"]*" # Ou a um valor entre aspas duplas
    | '[^']*' # Ou a um valor entre aspas simples
  )*? # Zero ou mais vezes, o mínimo de vezes possível (preguiçoso)
  \s cellspacing \b # Corresponde a "cellspacing" como uma palavra completa
) #
( # Captura a regex abaixo na retrorreferência 1
  (?: [^>"'] # Corresponde a qualquer caractere, exceto >, ", or '
```

```
    | "[^"]*" # Ou a um valor entre aspas duplas
    | '[^']*' # Ou a um valor entre aspas simples
  )* # Zero ou mais vezes, o máximo de vezes possível (mesquinho)
) #
> #
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Inserir o novo atributo

Todas as expressões regulares mostradas nesta receita podem usar as mesmas strings de substituição, pois todas as expressões regulares capturam atributos dentro das tags <table> correspondidas (se houver) na retrorreferência 1. Esse processo permite trazer de volta esses atributos como parte de um valor de substituição, enquanto adicionamos o novo atributo cellspacing. Aqui estão as strings de substituição necessárias:

```
<table cellspacing="0"$1>
```

Sabores de texto de substituição: .NET, Java, JavaScript, Perl, PHP

```
<table cellspacing="0"\1>
```

Sabores de texto de substituição: Python, Ruby

A receita 3.15 mostra como realizar substituições que utilizem uma retrorreferência no texto de substituição.

Discussão

A fim de examinar como funcionam estas expressões regulares, vamos desmembrar a primeira solução simplista. Como você vai ver, ela tem quatro partes lógicas.

A primeira parte, ‹<table\b›, corresponde aos caracteres literais <table, seguidos por uma extremidade de palavra (‹\b›). A extremidade de palavra impede a correspondência de nomes de tags que apenas comecem com “table”. Embora possa parecer desnecessária, quando estiver trabalhando com (X)HTML (já que não há elementos válidos denominados “tablet”, “tableau”, ou “tablespoon”, por exemplo), essa é uma boa prática e pode ajudá-lo a evitar erros, quando for adaptar a expressão regular para pesquisar por outras

tags.

A segunda parte da expressão regular, ‹(?![^>]*?\scellspacing\b)›, é um lookahead negativo. Ele não consome qualquer tipo de texto como parte da correspondência, mas declara que a tentativa de correspondência deverá falhar, se a palavra cellspacing ocorrer em qualquer lugar dentro da tag de abertura. Como vamos adicionar o atributo cellspacing a todas as correspondências, não queremos corresponder às tags que já o contenham.

Como o lookahead dá uma espiada além da posição atual na tentativa de correspondência, ele utiliza o ‹[^>]*?› inicial para poder pesquisar o quanto quiser, até o ponto em que se presume ser o final do tag (a primeira ocorrência de >). O restante do subpadrão do lookahead (‹\scellspacing\b›) simplesmente corresponde aos caracteres “cellspacing” literais como uma palavra completa. Correspondemos a um caractere inicial de espaço em branco (‹\s›), pois um espaço em branco deve, sempre, separar o nome de um atributo do nome da tag, ou de seus atributos anteriores. Nós correspondemos a uma extremidade de palavra à direita, em vez de outro caractere de espaço em branco, pois uma extremidade de palavra atende a nossa necessidade de corresponder a cellspacing como uma palavra completa, e ainda assim funciona, se o atributo não tiver nenhum valor ou se o nome do atributo for imediatamente seguido por um sinal de igual.

Continuando, chegamos à terceira parte da expressão regular: ‹([^>]*)›. Trata-se de uma classe de caracteres negada e de um quantificador “zero ou mais” subsequente, ambos encapsulados em um grupo de captura. Capturar esta parte da correspondência nos permite facilmente trazer de volta, como parte da string de substituição, os atributos contidos em cada tag correspondida. E, ao contrário do lookahead negativo, essa parte realmente adiciona os atributos dentro da tag à string correspondida pela expressão regular.

Finalmente, a expressão regular corresponde ao caractere literal ‹>›

para finalizar a tag.

A expressão regular 2, a chamada versão mais confiável, funciona exatamente da mesma forma que a expressão regular que acabamos de descrever, exceto pelo fato de que ambas as instâncias da classe de caracteres negada ‹[^>]› são substituídas por ‹(?:[^>'"]|"[^"]*"|'[^']*')›. Este padrão mais longo ignora os valores de atributos entre aspas simples ou duplas em uma só etapa.

Quanto às strings de substituição, elas trabalham com ambas as expressões regulares, substituindo cada tag ‹table› correspondida por uma nova tag com cellspacing como primeiro atributo, seguido pelos atributos ocorridos dentro da tag original (retrorreferência 1).

Veja também:

A receita 8.6 é o inverso conceitual desta receita, e encontra tags que contenham um atributo específico.

8.8 Remover comentários no estilo XML

Problema

Você deseja remover os comentários de um documento (X)HTML, ou XML. Por exemplo, você deseja remover os comentários de uma página web antes de ser servida aos navegadores web, a fim de reduzir o tamanho de arquivo da página e, assim, reduzir o tempo de carregamento para pessoas com conexões à Internet lentas.

Solução

Encontrar comentários não é uma tarefa difícil, graças à disponibilidade dos quantificadores preguiçosos. Veja a expressão regular utilizada:

```
<!--.*?-->
```

Opções Regex: Ponto corresponde a quebras de linha
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Esse trabalho é bastante simples. Como de costume, porém, a falta da opção “ponto corresponde a quebras de linha” no JavaScript significa que você terá de substituir o ponto por uma classe de caracteres que inclua tudo, para que a expressão regular corresponda aos comentários abrangendo mais de uma linha. A seguir, temos uma versão que funciona com JavaScript:

```
<!--[\s\S]*?-->
```

Opções Regex: Nenhuma
Sabores Regex: JavaScript

Para remover os comentários, substitua todas as correspondências pela string vazia (ou seja, nada).

A receita 3.14 lista o código para substituir todas as correspondências de uma expressão regular.

Discussão

Como isso funciona

No início e no final desta expressão regular estão as sequências de caracteres literais ‹<!--› e ‹-->›. Como nenhum desses caracteres são especiais na sintaxe da expressão regular (exceto dentro de classes de caracteres, nas quais hífens criam intervalos), eles não precisam ser escapados. Isso nos deixa apenas com o ‹.*?› ou ‹[\s\S]*?›, no meio da expressão regular, para serem examinados mais a fundo.

Graças à opção “ponto corresponde a quebras de linha”, o ponto na expressão regular mostrada inicialmente corresponde a qualquer caractere único. Na versão do JavaScript, a classe de caracteres ‹[\s\S]› toma o seu lugar. No entanto, as duas expressões regulares são exatamente iguais. ‹\s› corresponde a qualquer caractere de espaço em branco, e ‹\S› corresponde a todo o resto. Combinadas, elas correspondem a qualquer caractere.

O quantificador preguiçoso ‹*?› repete o elemento “qualquer caractere” anterior zero ou mais vezes, o mínimo de vezes possível. Assim, o símbolo anterior é repetido somente até a primeira ocorrência de -->, ao invés de corresponder a tudo até o final da

string de assunto, para depois retroceder até o último -->. Veja a receita 2.13, para saber mais sobre como funciona o retrocesso com quantificadores preguiçosos e mesquinhos. Essa estratégia simples funciona bem, já que comentários no estilo XML não podem ser aninhados uns dentro dos outros. Em outras palavras, eles sempre terminam na primeira ocorrência (mais à esquerda) de -->.

Quando comentários não podem ser removidos

A maioria dos desenvolvedores web está familiarizada com o uso de comentários HTML dentro de elementos <script› e <style›, para compatibilidade com navegadores antigos. Nos dias atuais, esta é apenas uma superstição sem sentido, mas sua utilização sobrevive, graças, em parte, à codificação do tipo copiar-e-colar. Nós supomos que, ao remover os comentários de um documento (X)HTML, você não vai querer remover JavaScript e CSS incorporados. Você provavelmente também vai querer deixar sozinho o conteúdo dos elementos <textarea>, seções CDATA e os valores de atributos dentro de tags.

Anteriormente, nós dissemos que remover comentários não era uma tarefa difícil. Como podemos constatar, isso só é verdade se você ignorar algumas das áreas mais difíceis do (X)HTML ou XML, em que as regras de sintaxe mudam. Em outras palavras, se você ignorar as partes difíceis do problema, tudo fica fácil.

Claro que, em alguns casos, você pode avaliar a marcação com a qual esteja lidando e decidir que está tudo bem quanto a ignorar esses casos problemáticos, talvez porque você mesmo tenha escrito a marcação e sabe o que esperar dela. Em outra situação positiva, pode ser que você esteja fazendo uma pesquisa-e-substituição em um editor de texto, sendo capaz de inspecionar, manualmente, cada correspondência antes de removê-la.

Porém, voltando à forma de contornar esses problemas, em “Pular seções (X)HTML e XML complicadas”, discutimos algumas dessas questões no contexto da correspondência de tags, no estilo XML.

Podemos utilizar uma linha de ataque semelhante ao pesquisar por comentários. Use o código da receita 3.18, para, primeiro, pesquisar seções complicadas usando a expressão regular mostrada a seguir, e depois substituir os comentários encontrados entre as correspondências pela string vazia (ou seja, para remover os comentários):

```
<(script|style|textarea|xmp)\b(?:[^>'"]|"[^"]*"|'[^']*')*?↵
(?:/>|>.*?</\1\s*>)|<[a-z](?:[^>'"]|"[^"]*"|'[^']*')*>|<!\[CDATA\[.*?]]>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, ponto corresponde a quebras de linha

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Adicionar alguns espaços em branco e alguns comentários à expressão regular, no modo de espaçamento livre, a torna muito mais fácil de acompanhar:

```
# Elemento especial: tag e conteúdo
<( script | style | textarea | xmp )\b
  (?: [^>'"] # Corresponde a qualquer nome de atributo
    | "[^"]*" # ...e a qualquer valor
    | '[^']*' #
)*?
(?: # Tag singular
/>
  | # De outra forma, inclua o conteúdo e a tag de fechamento correspondente
ao elemento
  > .*? </\1\s*>
)
|
# Elemento padrão: tag apenas
<[a-z] # Caractere inicial do nome da tag
  (?: [^>'"] # Corresponde ao resto do nome da tag
    | "[^"]*" # ...junto com nomes e valores
    | '[^']*' # ...de atributos
  )*
>
|
# Seção CDATA
<!\[CDATA\[ .*? ]]>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, ponto corresponde a quebras de linha, espaçamento livre

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Vejamos uma versão equivalente para o JavaScript, que não possui as opções “ponto corresponde a quebras de linha” e “espaçamento livre”:

```
<(script|style|textarea|xmp)\b(?:[^>'"]|"[^"]*"|'[^']*')*?↲
(?:/>|>[\s\S]*?</\1\s*>)|<[a-z](?:[^>'"]|"[^"]*"|'[^']*')*>|<!\[CDATA\[↲
[\s\S]*?]]>
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: JavaScript

Variações

Encontrar comentários válidos no estilo XML

Há, de fato, algumas poucas regras de sintaxe para comentários (X)HTML e XML, que vão além de simplesmente começar com <!-- e terminar com -->. Especificamente:

- Dois hífens não podem aparecer em uma linha dentro de um comentário. Por exemplo, <!--com--ment --> é inválido por causa dos dois hífens no meio.
- O delimitador de fechamento não pode ser precedido por um hífen que faça parte do comentário. Por exemplo, <!-- comment ---> é inválido, mas o comentário completamente vazio <!----> é permitido.
- Espaços em branco podem ocorrer entre o -- e o > de fechamento. Por exemplo, <!—comment -- > é um comentário válido e completo.

Não é difícil tratar essas regras em uma expressão regular:

```
<!--[^-]*(?:-[^-]+)*--\s*>
```

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Observe que tudo entre os delimitadores de abertura e fechamento de um comentário ainda é opcional, então a expressão corresponde ao comentário completamente vazio <!---->. No entanto, se um hífen ocorrer entre os delimitadores, ele deve ser seguido por pelo menos

um caractere que não seja um hífen. Como a parte interna da expressão regular não pode mais corresponder a dois hífens em sequência, o quantificador preguiçoso das expressões regulares, no início desta receita, foi substituído por quantificadores mesquinhos. Quantificadores preguiçosos ainda poderiam funcionar bem, mas utilizá-los aqui resultaria em retrocessos desnecessários (veja a receita 2.13).

Alguns leitores poderão olhar para esta nova expressão regular, querendo saber por que a classe de caracteres negada ‹[^-]› é utilizada duas vezes, no lugar de apenas tornar o hífen opcional dentro do grupo de não-captura (ou seja, ‹<!--(?:-?[^-]+)*--\s*>›). Há uma boa razão para isso, que nos traz de volta à discussão do "retrocesso catastrófico", da receita 2.15.

Os chamados *quantificadores aninhados* sempre exigem atenção e cuidados redobrados, para garantir que você não esteja criando a possibilidade de um retrocesso catastrófico. Um quantificador é aninhado quando ocorre dentro de um agrupamento que é, em si, repetido por um quantificador. Por exemplo, o padrão ‹(?:-?[^-]+)*› contém dois quantificadores aninhados: o ponto de interrogação após o hífen e o sinal de adição após a classe de caracteres negada.

No entanto, aninhar quantificadores não é o que realmente torna esse exercício perigoso, no sentido de desempenho. Pelo contrário: há um número potencialmente enorme de maneiras pelas quais o quantificador externo ‹*› pode ser combinado com os quantificadores internos, enquanto tenta corresponder a uma string. Se o mecanismo de expressão regular não conseguir encontrar --> no final de uma correspondência parcial (como é exigido quando você conecta este segmento de padrão dentro da expressão regular de correspondência a comentários), ele deverá tentar todas as combinações possíveis de repetição, antes de falhar na tentativa de correspondência e seguir em frente. Este número de opções aumenta muito rapidamente com cada caractere adicional que o

mecanismo deverá tentar corresponder. No entanto, não há nada de perigoso nos quantificadores aninhados, se esta situação for evitada. Por exemplo, o padrão ‹(?:-[^-]+)*› não representa um risco, mesmo que ele contenha um quantificador aninhado ‹+›, pois, agora que exatamente um hífen deverá ser correspondido por repetição do grupo, o número potencial de pontos de retrocesso aumenta linearmente com o comprimento da string de assunto.

Outra forma de evitar o problema de retrocesso em potencial, que acabamos de descrever, é usando um grupo atômico. A expressão regular seguinte é equivalente à primeira expressão regular apresentada nesta seção, mas tem alguns caracteres a menos e não é suportada pelo JavaScript ou Python:

```
<!--(?>-?[^-]+)*--\s*>
```

Opções Regex: Nenhuma
Sabores Regex: .NET, Java, PCRE, Perl, Ruby

Veja a receita 2.14, para obter detalhes sobre como grupos atômicos (e sua contraparte, quantificadores possessivos) funcionam.

Encontre comentários no estilo da linguagem C

O mesmo tipo de padrão, como mostrado em comentários no estilo XML, funciona bem com outros comentários não-aninhados de várias linhas. Os comentários no estilo da linguagem C começam com /* e terminam com a primeira ocorrência de */, ou começam com // e continuam até o fim da linha. A expressão regular seguinte corresponde a esses dois tipos de comentários, ao combinar um padrão para cada tipo usando uma barra vertical:

```
/\*[\s\S]*?\*/|//.*
```

Opções Regex: Nenhuma (“ponto corresponde a quebras de linha” não deve estar ativado)
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Veja também:

A receita 8.9 mostra como encontrar palavras específicas, quando

elas ocorrem dentro de comentários no estilo XML.

8.9 Encontrar palavras dentro de comentários no estilo XML

Problema

Você deseja encontrar todas as ocorrências da palavra TODO (“pendência”) dentro de comentários (X)HTML ou XML. Por exemplo, você deseja combinar somente o texto sublinhado dentro da seguinte string:

```
Este "TODO" não está dentro de um comentário, mas o próximo está. <!--
TODO: ↲
Inventar um comentário mais bacana para este exemplo. -->
```

Solução

Há pelo menos duas abordagens para este problema, e ambas têm suas vantagens. A primeira tática, descrita como “Abordagem de duas etapas”, é encontrar comentários com uma expressão regular externa e, em seguida, pesquisar dentro de cada correspondência, usando uma expressão regular separada ou até mesmo uma busca de texto simples. O procedimento funciona melhor se você estiver escrevendo código para realizar o trabalho, uma vez que separar a tarefa em duas etapas mantém as coisas simples e rápidas. No entanto, se estiver pesquisando nos arquivos usando um editor de texto ou uma ferramenta grep, dividir a tarefa em duas fases não vai funcionar, a menos que sua ferramenta de escolha ofereça uma opção especial para pesquisar dentro de correspondências encontradas por outra expressão regular[2].

Se precisar encontrar palavras dentro de comentários usando uma única expressão regular, você pode empregar um lookaround. O segundo método é mostrado na seção “Abordagem de etapa única”, que virá mais adiante.

Abordagem de duas etapas

Quando for uma opção viável, a melhor solução é dividir a tarefa em duas: pesquisar por comentários e, em seguida, pesquisar por TODO dentro desses comentários.

Veja como você pode encontrar comentários:

```
<!--.*?-->
```

Opções Regex: Ponto corresponde a quebras de linha
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

O JavaScript não tem a opção “ponto corresponde a quebras de linha”, mas você pode usar uma classe de caracteres que inclua tudo no lugar do ponto, da seguinte forma:

```
<!--[\s\S]*?-->
```

Opções Regex: Nenhuma
Sabor Regex: JavaScript

Para cada comentário que encontrar usando uma das expressões regulares mostradas, você pode pesquisar, dentro do texto correspondido, pelos caracteres literais ‹TODO›. Se preferir, pode fazer disso uma expressão regular sem distinção entre maiúsculas e minúsculas, com extremidades de palavra de cada lado, para se certificar de que apenas a palavra completa TODO será correspondida. Assim:

```
\bTODO\b
```

Opções Regex: Não diferenciar maiúsculas de minúsculas
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

A receita 3.13 mostra como fazer uma busca dentro de correspondências de uma expressão regular externa.

Abordagem de etapa única

O lookahead (descrito na receita 2.16) nos permite resolver este problema com uma única expressão regular, embora seja menos eficiente. Na seguinte expressão regular, um lookahead positivo é usado para garantir que a palavra TODO seja seguida pelo delimitador de comentário final -->. Por si só, isso não significa que a

palavra apareça dentro de um comentário ou que ela seja, simplesmente, seguida de um comentário; assim, um lookahead negativo aninhado é usada para garantir que o delimitador do comentário de abertura <!-- não apareça antes de -->:

```
\bTODO\b(?=(?:(?!<!--).)*?-->)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, ponto corresponde a quebras de linha

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Como o JavaScript não tem uma opção “ ponto corresponde a quebras de linha “, use ‹[\s\S]› no lugar do ponto:

```
\bTODO\b(?=(?:(?!<!--)[\s\S])*?-->)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas

Sabor Regex: JavaScript

Discussão

Abordagem em duas etapas

A receita 3.13 mostra o código necessário para fazer uma pesquisa dentro de correspondências de outra expressão regular. É preciso utilizar uma expressão regular interna e outra externa. A expressão regular de comentários serve como expressão regular externa, e ‹\bTODO\b› como expressão regular interna. A principal coisa a notar aqui é o quantificador preguiçoso ‹*?›, que segue o ponto ou classe de caracteres na expressão regular de comentários. Conforme explicado na receita 2.13, isso nos permite corresponder com o primeiro --> (aquele que termina o comentário), ao invés da última ocorrência de --> em sua string de assunto.

Abordagem de etapa única

Esta solução é mais complexa e mais lenta. Olhando pelo lado positivo, ela combina as duas etapas da abordagem anterior em uma expressão regular. Assim, ela pode ser usada quando trabalhamos com um editor de texto, IDE ou com outra ferramenta que não nos permita pesquisar dentro de correspondências de outra expressão regular.

Vamos quebrar essa expressão regular no modo de espaçamento livre, dando uma olhada em cada uma das suas partes constituintes:

```
\b TODO \b # Corresponde aos caracteres "TODO", como uma palavra completa
(?= # Declara que a regex abaixo pode corresponder aqui
  (?: # Agrupa, mas não captura...
    (?! <!-- ) # Declara que "<!--" não pode corresponder aqui
    . # Corresponde a qualquer caractere único
  )*? # Repete zero ou mais vezes, o mínimo de vezes possível (preguiçoso)
  --> # Corresponde aos caracteres "-->"
) #
```

Opções Regex: Ponto corresponde a quebras de linha, espaçamento livre
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Esta versão comentada da expressão regular não funciona em JavaScript, uma vez que ela carece tanto do modo de “espaçamento livre”, quanto do modo “ponto corresponde a quebras de linha”.

Observe que a expressão regular possui um lookahead negativo aninhado dentro de um lookahead positivo externo. Isso nos permite exigir que qualquer correspondência de TODO seja seguida por --> e que <!-- não ocorra entre eles.

Se estiver claro para você como tudo isso funciona em conjunto, ótimo: você pode pular o restante desta seção. Mas, no caso de ainda estar um pouco vago, vamos dar um passo para trás e construir o lookahead positivo externo desta expressão regular, passo a passo.

Digamos que, por um momento, nós simplesmente queiramos corresponder às ocorrências da palavra TODO seguidas, em algum lugar na string, por -->. Isso nos dá a expressão regular ‹\bTODO\b(?=.*?-->)› (com “ponto corresponde a quebras de linha” habilitado), que corresponde muito bem ao texto sublinhado <!--TODO-->. Precisamos do ‹.*?› no início do lookahead, pois, caso contrário, a expressão regular corresponderia apenas quando TODO fosse *imediatamente* seguido por -->, sem caracteres entre eles. O quantificador ‹*?› repete o ponto zero ou mais vezes, num mínimo de vezes possível, o que é muito bom, já que queremos corresponder apenas até o primeiro -->.

Fora dessas reflexões, a expressão regular poderia ser reescrita como ‹\bTODO(?=.*?-->)\b› – com o segundo ‹\b› colocado após o lookahead – sem qualquer efeito sobre o texto correspondido. Isso ocorre porque, tanto a extremidade de palavra, como o lookahead são declarações de comprimento zero (veja “Lookaround”). No entanto, é melhor colocar primeiro a extremidade de palavra, por uma questão de legibilidade e eficiência. No meio de uma correspondência parcial, o mecanismo de expressão regular pode testar mais rapidamente uma extremidade de palavra, falhar e avançar, para tentar novamente no próximo caractere da string, sem ter de gastar tempo testando o lookahead quando isso não é necessário, pois TODO não é uma palavra completa.

Então, a expressão regular ‹\bTODO\b(?=.*?-->)› parece funcionar bem, até agora; mas, e quando ela é aplicada à string de assunto TODO <!-- comentário separado -->? A expressão regular ainda corresponde a TODO, já que é seguida por -->, apesar de TODO não estar dentro de um comentário. Assim, precisamos mudar o ponto dentro do lookahead; ele não corresponderá a qualquer caractere, mas a qualquer caractere que não faça parte da string <!--, pois isso indicaria o início de um novo comentário. Não podemos usar uma classe de caracteres negada como ‹[^<!-]›, pois queremos permitir os caracteres <, ! e - que não estejam agrupados na sequência exata <!--.

É neste momento que o lookahead negativo aninhado entra. ‹(?!<!--).› corresponde a qualquer caractere único que não faça parte de um delimitador de comentário de abertura. Colocar esse padrão dentro de um grupo de não-captura, como ‹(?:(?!<!--).)›, nos permite repetir toda a sequência com o quantificador preguiçoso ‹*?›, que tínhamos aplicado anteriormente apenas para o ponto.

Juntando tudo, temos a expressão regular final, listada como a solução para este problema: ‹\bTODO\b(?=(?:(?!<!--).)*?-->)›. Em JavaScript, que não possui a opção necessária “ponto corresponde a quebras de linha”, ‹\bTODO\b(?=(?:(?!<!--)[\s\S])*?-->)› é equivalente .

Variações

Embora a expressão regular “abordagem de passo único” assegure que qualquer correspondência a TODO seja seguida de -->, sem <!-- ocorrendo entre eles, ela não verifica o contrário: o fato de que a palavra-alvo também é precedida por <!--, sem --> entre eles. Há várias razões para deixarmos esta regra de fora:

- Normalmente, você não precisa fazer esta verificação dupla, especialmente porque a expressão regular de passo único foi feita para ser usada com editores de texto e coisas do gênero, onde é possível verificar visualmente os resultados.
- Verificar menos coisas significa menos tempo gasto realizando a verificação (ou seja, é mais rápido abandonar a verificação extra).
- O mais importante: como você não sabe o quão anteriormente o comentário pode ter iniciado, olhar para trás exige um lookbehind de comprimento infinito, suportado apenas pelo sabor de expressão regular .NET.

Se estiver trabalhando com .NET e deseja incluir esta verificação, use a seguinte expressão regular:

```
(?<=<!--(?:(?!-->).)*?)\bTODO\b(?=(?:(?!<!--).)*?-->)
```

Opções Regex: Não diferenciar maiúsculas de minúsculas, ponto corresponde a quebras de linha

Sabor Regex: .NET

Esta expressão regular, mais rigorosa e exclusiva do .NET, acrescenta um lookbehind positivo no início, que funciona exatamente como o lookahead no final, mas em sentido reverso. Como o lookbehind funciona para frente, a partir da posição em que ele encontra o <!--, o lookbehind contém um lookahead negativo aninhado, que o permite corresponder a qualquer caractere que não faça parte da sequência -->.

Já que o lookahead à esquerda, e o lookbehind à direita são ambos declarações de comprimento zero, a correspondência final será apenas a palavra TODO. As strings correspondidas dentro dos

lookarounds não se tornam parte do texto correspondido final.

Veja também:

A receita 8.8 inclui uma discussão detalhada sobre como corresponder a comentários no estilo XML.

8.10 Mudar o delimitador usado em arquivos CSV

Problema

Você deseja mudar todas as vírgulas delimitadoras de campos em um arquivo CSV para tabulações. Vírgulas que ocorram dentro de valores com aspas duplas devem permanecer inalteradas.

Solução

A seguinte expressão regular corresponde a um campo CSV individual, juntamente com seu delimitador anterior, se houver. O delimitador anterior é geralmente uma vírgula, mas também pode ser uma string vazia (ou seja, nada), quando corresponde ao primeiro campo do primeiro registro, ou uma quebra de linha, quando corresponde ao primeiro campo de qualquer registro subsequente. Sempre que for encontrada uma correspondência, o próprio campo, incluindo as aspas duplas que podem envolvê-lo, é capturado na retrorreferência 2, e seu delimitador anterior é capturado na retrorreferência 1.

As expressões regulares, nesta receita, são projetadas para funcionar corretamente somente com arquivos CSV válidos, de acordo com as regras de formato sob o termo “Valores separados por vírgula (CSV)”, discutido anteriormente neste livro.

```
(,|\r?\n|^)([^",\r\n]+|"(?:[^"]|"")*")?
```

Opções Regex: Nenhuma (“^ e $ correspondem em quebras de linha” não deve estar definido)

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Vejamos a mesma expressão regular no modo de espaçamento livre:

```
( , | \r?\n | ^ ) # Grupo de captura 1 corresponde a delimitadores de campos
                  # ou ao início da string
( # Grupo de captura 2 corresponde a um único campo:
  [^",\r\n]+ # Um campo não colocado entre aspas
| # ou...
  " (?:[^"]|"")* " # um campo entre aspas (pode conter aspas duplas escapadas)
)? # O grupo é opcional, porque os campos podem estar vazios
```

Opções Regex: Espaçamento livre (“^ e $ correspondem em quebras de linha” não deve estar definido)

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Usando esta expressão regular e o código da receita 3.11, você pode iterar seu arquivo CSV e verificar o valor da retrorreferência 1, após cada correspondência. A string de substituição necessária para cada correspondência depende do valor desta retrorreferência. Se for uma vírgula, substitua-a por um caractere de tabulação. Se a retrorreferência estiver vazia ou contiver uma quebra de linha, deixe o valor no lugar (ou seja, não faça nada, ou coloque-a de volta como parte de uma string de substituição). Como os campos CSV são capturados na retrorreferência 2, como parte de cada correspondência, você também terá de colocar isso de volta como parte de cada string de substituição. As únicas coisas que você está realmente substituindo são as vírgulas capturadas na retrorreferência 1.

Exemplo JavaScript

O código a seguir é uma página web completa, que inclui dois campos de texto com várias linhas de entrada, com um botão chamado Replace entre eles. Clicar no botão pega qualquer string que você colocar na primeira caixa de texto (rotulada de Input), converte qualquer tipo de delimitador de vírgula para tabulações, com a ajuda da expressão regular mostrada e, então, coloca a nova string na segunda caixa de texto (chamada de Output). Se você usar conteúdo CSV válido como sua entrada, ele deverá aparecer na segunda caixa de texto, com todos os delimitadores de vírgula substituídos por tabulações. Para testá-lo, salve este código em um arquivo com a extensão “.html”, e o abra em seu navegador

preferido:

```
<html>
<head>
<title>Muda os delimitadores CSV de vírgulas para tabulações</title>
</head>
<body>
<p>Input:</p>
<textarea id="input" rows="5" cols="75"></textarea>
<p><input type="button" value="Replace" onclick="commas_to_tabs()"></p>
<p>Output:</p>
<textarea id="output" rows="5" cols="75"></textarea>
<script>
function commas_to_tabs () {
  var input = document.getElementById('input'),
    output = document.getElementById('output'),
    regex = /(,|\r?\n|^)([^",\r\n]+|"(?:[^"]|"")*")?/g,
    result = '',
    match;
  while (match = regex.exec(input.value)) {
    // Verifica o valor na retorreferência 1
    if (match[1] == ',') {
        // Adiciona uma tabulação (no lugar da vírgula correspondida)
        // e a retrorreferência 2 ao resultado.
        // Se a retrorreferência 2 for undefined (porque o segundo grupo de captura
        // opcional não participou da correspondência), use uma string vazia.
        result += '\t' + (match[2] || '');
    } else {
        // Adiciona toda a correspondência ao resultado
        result += match[0];
    }
    // Evita que alguns navegadores fiquem presos em um loop infinito
    if (match.index == regex.lastIndex) regex.lastIndex++;
  }
  output.value = result;
}
</script>
</body>
</html>
```

Discussão

A abordagem prescrita por esta receita permite que você pule cada campo CSV completo (incluindo quebras de linha incorporadas, aspas duplas escapadas e vírgulas), um de cada vez. Cada correspondência, então, começa exatamente antes do próximo delimitador de campo.

O primeiro grupo de captura da expressão regular, ‹(,|\r?\n|^)›, corresponde a uma vírgula, a uma quebra de linha ou à posição no início da string de assunto. Como o mecanismo de expressão regular tentará alternativas da esquerda para a direita, estas opções são listadas na ordem em que elas ocorrem com mais frequência no arquivo CSV. Este grupo de captura é a única parte da expressão regular que, obrigatoriamente, deverá corresponder. Portanto, é possível que a expressão regular completa corresponda a uma string vazia, já que a âncora ‹^› pode sempre corresponder uma vez. O valor correspondido por este primeiro grupo de captura deve ser verificado via código, fora da expressão regular que substitui vírgulas pelo delimitador substituto (por exemplo, um tab).

Ainda não analisamos toda a expressão regular, mas a abordagem descrita até agora já é um tanto complicada. Você pode estar se perguntando por que a expressão regular não foi escrita *apenas* para corresponder às vírgulas que devem ser substituídas por guias. Se você pudesse fazer isso, uma simples substituição de todo o texto correspondido evitaria a necessidade de código fora da expressão regular, para verificar se o grupo de captura 1 correspondeu a uma vírgula ou a alguma outra string. Afinal de contas, deveria ser possível utilizar um lookahead e um lookbehind para determinar se uma vírgula está dentro ou fora de um campo CSV entre aspas, certo?

Infelizmente, para que tal abordagem consiga determinar, com precisão, quais vírgulas estão fora de campos entre aspas duplas, você precisaria de um lookbehind de comprimento infinito, disponível somente no sabor de expressão regular .NET (veja “Diferentes níveis de lookbehind”, para uma discussão sobre as

limitações do lookbehind). Mesmo os desenvolvedores .NET deveriam evitar uma abordagem baseada em lookaround, uma vez que acrescentaria enorme complexidade e, também, tornaria a expressão regular mais lenta.

Voltando ao funcionamento da expressão regular, a maior parte do padrão aparece dentro do próximo conjunto de parênteses: o grupo de captura 2. Este segundo grupo corresponde a um único campo CSV, incluindo aspas duplas que possam estar envolvendo o campo. Ao contrário do grupo de captura anterior, este é opcional, a fim de permitir a correspondência de campos vazios.

Repare que o grupo 2 da expressão regular contém dois padrões alternativos, separados pelo metacaractere ‹|›. A primeira alternativa, ‹[^",\r\n]+›, é uma classe de caracteres negada, seguida de um ou mais quantificadores (‹+›) que, juntos, correspondem a um campo inteiro que não esteja entre aspas. Para que ela corresponda, o campo não pode conter aspas duplas, vírgulas ou quebras de linha.

A segunda alternativa dentro do grupo 2, ‹"(?:[^"]|"")*"›, corresponde a um campo colocado entre aspas duplas. Mais precisamente, ela corresponde a um caractere de aspa dupla, seguido por zero ou mais caracteres que não sejam aspas duplas e aspas duplas repetidas (escapadas), seguidos por uma aspa dupla de fechamento.

O quantificador ‹*›, no final do grupo de não-captura interior, repete as duas opções internas tanto quanto possível, até chegar a uma aspa dupla que não se repita e que, portanto, termina o campo.

Supondo que você esteja trabalhando com um arquivo CSV válido, a primeira correspondência encontrada por esta expressão regular deverá ocorrer no início da string de assunto, e cada correspondência subsequente deverá ocorrer imediatamente após o final da última correspondência.

Veja também:

A receita 8.11 descreve como reutilizar a expressão regular nesta receita, para extrair campos CSV de uma coluna específica.

8.11 Extrair campos CSV de uma coluna específica

Problema

Você deseja extrair todos os campos da terceira coluna de um arquivo CSV.

Solução

As expressões regulares da receita 8.10 podem ser reutilizadas aqui para iterar cada campo em uma string de assunto CSV. Com um pouco de código extra, você pode contar o número de campos, da esquerda para a direita, em cada linha, ou *registro*, e extrair os campos na posição em que estiver interessado.

A seguinte expressão regular (mostrada com, e sem, a opção de espaçamento livre) corresponde a um único campo CSV, e a seu delimitador anterior em dois grupos de captura separados. Como as quebras de linha podem aparecer dentro de campos entre aspas duplas, não seria muito preciso, simplesmente, pesquisar a partir do início de cada linha em sua string CSV. Ao corresponder e pular os campos, um por um, você pode, facilmente, determinar quais quebras de linha aparecem fora de campos entre aspas duplas, e que, portanto, iniciam um novo registro.

As expressões regulares, nesta receita, são projetadas para funcionar corretamente somente com arquivos CSV válidos, de acordo com as regras discutidas em “Valores separados por vírgulas (CSV)”.

```
(,|\r?\n|^)([^",\r\n]+|"(?:[^"]|"")*")?
```

Opções Regex: Nenhuma (“^ e $ correspondem em quebras de linha” não deve estar definido)

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
( , | \r?\n | ^ ) # Grupo de captura 1 corresponde aos delimitadores de campos
                  # ou ao início da string
```

```
( # Grupo de captura 2 corresponde a um único campo:
  [^",\r\n]+ # um campo que não esteja entre aspas
| # ou...
  " (?:[^"]|"")* " # um campo entre aspas (pode conter aspas duplas escapadas)
)? # O grupo é opcional porque o campo pode estar vazio
```

Opções Regex: Espaçamento livre (“^ e $ correspondem em quebras de linha” não deve estar ativado)

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Essas expressões regulares são exatamente as mesmas da receita 8.10, e também podem ser reutilizadas para muitas outras tarefas de processamento CSV. O código de exemplo, a seguir, demonstra como você pode usar a versão sem a opção de espaçamento livre para ajudá-lo a extrair uma coluna CSV.

Exemplo JavaScript

O código seguinte é uma página web completa, que inclui dois campos de texto, com várias linhas, e um botão entre eles, chamado Extract Column 3. Clicar no botão pega qualquer string que você coloque na caixa de texto de entrada, extrai o valor do terceiro campo, em cada registro, com a ajuda da expressão regular mostrada anteriormente e, então, insere a coluna inteira (com cada valor separado por uma quebra de linha) no campo *Output*. Para testá-lo, salve este código em um arquivo com a extensão “.html”, e o abra em seu navegador preferido:

```
<html>
<head>
<title>Extrai a terceira coluna de uma string CSV</title>
</head>
<body>
<p>Input:</p>
<textarea id="input" rows="5" cols="75"></textarea>
<p><input type="button" value="Extract Column 3"
  onclick="display_csv_column(2)"></p>
<p>Output:</p>
<textarea id="output" rows="5" cols="75"></textarea>
<script>
function display_csv_column (index) {
```

```
  var input = document.getElementById('input'),
    output = document.getElementById('output'),
    column_fields = get_csv_column(input.value, index);
  if (column_fields.length > 0) {
    // Exibe cada registro em sua própria linha, separadas por uma alimentação
de linha (\n)
    output.value = column_fields.join('\n');
  } else {
    output.value = '[No data found to extract]';
  }
}
// Retorna um array de campos CSV relativo ao índice fornecido (baseado em
zero)
function get_csv_column (csv, index) {
  var regex = /(,|\r?\n|^)([^",\r\n]+|"(?:[^"]|"")*")?/g,
    result = [],
    column_index = 0,
    match;
  while (match = regex.exec(csv)) {
    // Verifica o valor da retrorreferência 1. Se for uma vírgula,
    // incrementa column_index. Caso contrário, redefina-o para zero.
    if (match[1] == ',') {
        column_index++;
    } else {
        column_index = 0;
    }
    if (column_index == index) {
        // Adiciona o campo (retroreferência 2) no final do array result
        result.push(match[2]);
    }
    // Evita que alguns navegadores fiquem presos em um loop infinito
    if (match.index == regex.lastIndex) regex.lastIndex++;
  }
  return result;
}
</script>
</body>
</html>
```

Discussão

Como estas expressões regulares são versões modificadas da

receita 8.10, não repetiremos a explicação detalhada sobre como elas funcionam. No entanto, esta receita inclui novos exemplos de código JavaScript, que usam a expressão regular para extrair campos em um índice específico de cada registro, na string de assunto CSV.

No código apresentado, a função get_csv_column trabalha para iterar a string de assunto, uma correspondência por vez. Depois de cada correspondência, a retrorreferência 1 é examinada, para verificar se ela contém uma vírgula. Se contiver, você correspondeu a algo diferente do primeiro campo de uma fileira; então, a variável column_index é incrementada, para ficar de olho na coluna em que você está. Se a retrorreferência 1 for outra coisa que não uma vírgula (ou seja, uma sequência vazia, ou uma quebra de linha), você correspondeu ao primeiro campo em uma nova linha, e column_index é redefinido para zero.

O próximo passo, no código, é verificar se o contador column_index atingiu o índice que você pretende extrair. Sempre que isso ocorre, o valor da retrorreferência 2 (tudo, depois do delimitador inicial) é empurrado para o array result. Depois que você iterou toda a string de assunto, a função get_csv_column retorna um array, contendo todos os valores da coluna especificada (neste exemplo, a terceira coluna). A lista de correspondências é, então, despejada na segunda caixa de texto da página, com cada valor separado por um caractere de alimentação de linha (\n).

Uma melhoria simples seria permitir que o usuário especificasse o índice de coluna que deveria ser extraído, por meio de um prompt ou um campo de texto adicional. A função get_csv_column, que já discutimos, foi escrita com esta funcionalidade em mente, permitindo que você especifique a coluna desejada como um valor inteiro, baseado em zero, por meio do segundo parâmetro (index).

Variações

Embora utilizar o código para iterar um campo CSV por vez em uma

string permita uma flexibilidade extra, se estiver usando um editor de texto para fazer esse trabalho, você poderá se limitar a operações de busca-e-substituição. Nesta situação, é possível conseguir um resultado semelhante, correspondendo a cada registro completo e substituindo-o pelo valor do campo no índice da coluna que estiver procurando (usando uma retrorreferência). As seguintes expressões regulares ilustram esta técnica para índices de coluna específicos, substituindo cada registro pelo campo de uma coluna específica.

Em todas estas expressões regulares, se um registro não contiver, no mínimo, uma quantidade de campos igual ao índice da coluna procurada, este registro não será correspondido, e será deixado no lugar.

Corresponder a um registro CSV e capturar o campo da coluna 1, na retrorreferência 1

```
^([^",\r\n]+|"(?:[^"]|"")*")?(?:,(?:[^",\r\n]+|"(?:[^"]|"")*")?)*
```

Opções Regex: ^ e $ correspondem em quebras de linha
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Corresponder a um registro CSV e capturar o campo da coluna 2, na retrorreferência 1

```
^(?:[^",\r\n]+|"(?:[^"]|"")*")?,([^",\r\n]+|"(?:[^"]|"")*")?↲
(?:,(?:[^",\r\n]+|"(?:[^"]|"")*")?)*
```

Opções Regex: ^ e $ correspondem em quebras de linha
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Corresponder a um registro CSV, e capturar o campo da coluna 3, ou superior, na retrorreferência 1

```
^(?:[^",\r\n]+|"(?:[^"]|"")*")?(?:,(?:[^",\r\n]+|"(?:[^"]|"")*")?){1},↲
([^",\r\n]+|"(?:[^"]|"")*")?(?:,(?:[^",\r\n]+|"(?:[^"]|"")*")?)*
```

Opções Regex: ^ e $ correspondem em quebras de linha
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Incremente o número dentro do quantificador ‹{1}›, para fazer esta

última expressão regular trabalhar para qualquer elemento maior do que a coluna 3. Por exemplo, mude-a para ‹{2}› para capturar os campos da coluna 4, ‹{3}› para a coluna 5 e assim por diante. Se estiver trabalhando com a coluna 3, você pode simplesmente remover o ‹{1}›, se preferir, já que aqui ele não tem efeito.

String de substituição

A mesma string de substituição (retrorreferência 1) é usada em todas estas expressões regulares. O ato de substituir cada correspondência pela retrorreferência 1 deverá deixá-lo apenas com os campos que estiver procurando.

```
$1
```

Sabores de texto de substituição: .NET, Java, JavaScript, Perl, PHP

```
\1
```

Sabores de texto de substituição: Python, Ruby

8.12 Corresponder a cabeçalhos de seção INI

Problema

Você deseja corresponder a todos os cabeçalhos de seção em um arquivo INI.

Solução

Esta é fácil. Cabeçalhos de seção INI aparecem no início de uma linha, e são designados colocando-se um nome entre colchetes (por exemplo, [Section1]). Essas regras são simples de traduzir em uma expressão regular:

```
^\[[^\]\r\n]+]
```

Opções Regex: ^ e $ correspondem em quebras de linha
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Discussão

Não existem muitas partes nesta expressão regular, então é fácil

desmembrá-la:

- O ‹^› inicial corresponde à posição no início de uma linha, pois a opção “^ e $ correspondem em quebras de linha” está habilitada.
- ‹\[› corresponde a um caractere [literal. Ele é escapado com uma barra, para evitar que [inicie uma classe de caracteres.
- ‹[^\]\r\n]› é uma classe de caracteres negada que corresponde a qualquer caractere, exceto um], um retorno de carro (\r) ou uma alimentação de linha (\n). O quantificador ‹+›, imediatamente a seguir, deixa a classe corresponder a um ou mais caracteres, o que nos leva ao próximo item.
- O ‹]›, à direita, corresponde a um caractere] literal para finalizar o cabeçalho de seção. Não há necessidade de escapar este caractere com uma barra, pois ele não ocorre dentro de uma classe de caracteres.

Se voce quiser apenas encontrar uma cabeçalho de seção específico, o processo fica ainda mais fácil. A expressão regular seguinte corresponde ao cabeçalho de uma seção chamada Section1:

```
^\[Section1]
```

Opções Regex: ^ e $ correspondem em quebras de linha
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Neste caso, a única diferença de uma pesquisa de texto simples para “[Section1]” é que a correspondência deve ocorrer no início de uma linha. Isso impede a correspondência a cabeçalhos de seção comentados (precedidos por um ponto-e-vírgula), ou a algo que se pareça com um cabeçalho, mas que, na verdade, faça parte do valor de um parâmetro (por exemplo, Item1=[Value1]).

Veja também:

A receita 8.13 descreve como corresponder a blocos de seção INI.

A receita 8.14 procede da mesma maneira com pares nome-valor INI.

8.13 Corresponder a blocos de seção INI

Problema

Você precisa corresponder a cada bloco de seção INI completo (ou seja, um cabeçalho de seção e todos os seus pares parâmetro-valor), a fim de dividir um arquivo INI ou para processar cada bloco separadamente.

Solução

A receita 8.12 mostrou como corresponder a um cabeçalho de seção INI. Para corresponder a uma seção inteira, vamos começar com o mesmo padrão nela apresentado, mas continuaremos correspondendo até alcançarmos o final da string, ou um caractere `[` que ocorra no início de uma linha (pois isso indica o início de uma nova seção):

```
^\[[^\]\r\n]+](?:\r?\n(?:[^[\r\n].*)?)*
```

Opções Regex: ^ e $ correspondem em quebras de linha (“ponto corresponde a quebras de linha” não deve estar definido)

Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Ou, em modo espaçamento livre:

```
^ \[ [^\]\r\n]+ ] # Corresponde a um cabeçalho de seção
(?: # Seguido pelo resto da seção...
  \r?\n # Corresponde a uma sequência de quebra de linha
  (?: # Após o início de cada linha, corresponda...
    [^[\r\n] # Qualquer caractere, exceto "[", ou um caractere de quebra de linha
    .* # Corresponde ao restante da linha
  )? # O grupo é opcional, para pemitir a correspondência de linhas vazias
)* # Continua até o final da seção
```

Opções Regex: ^ e $ correspondem em quebras de linha, espaçamento livre (“ponto corresponde a quebras de linha” não deve estar definido)

Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Discussão

Esta expressão regular começa correspondendo a um cabeçalho de

seção INI com o padrão ‹^\[[^\]\r\n]+]›, e continua correspondendo uma linha por vez, enquanto as linhas não começarem com [. Considere o seguinte texto de assunto:

```
[Section1]
Item1=Value1
Item2=[Value2]
; [SectionA]
; O cabeçalho SectionA foi comentado
ItemA=ValueA ; ItemA não está comentado, e faz parte de Section1
[Section2]
Item3=Value3
Item4 = Value4
```

Dada a string que acabamos de exibir, esta expressão regular encontra duas correspondências. A primeira correspondência se estende do início da string, até, e inclusive, a linha vazia anterior a “[Section2]”. A segunda correspondência se estende do início do cabeçalho Section2 até o final da string.

Veja também:

A receita 8.12 mostra como corresponder a cabeçalhos de seção INI.

A receita 8.14 procede da mesma maneira, para pares nome-valor INI.

8.14 Corresponder a pares nome-valor INI

Problema

Você deseja corresponder a pares nome-valor de parâmetros INI (por exemplo, Item1=Value1), separando cada correspondência em duas partes e utilizando grupos de captura. A retrorreferência 1 deverá conter o nome do parâmetro (Item1), e a retrorreferência 2, o valor (Value1).

Solução

Vejamos a expressão regular para realizar esse trabalho (a segunda é mostrada com o modo de espaçamento livre):

```
^([^=;\r\n]+)=([^;\r\n]*)
```

Opções Regex: ^ e $ correspondem em quebras de linha
Sabores Regex: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

```
^ # Início de uma linha
( [^=;\r\n]+ ) # Captura o nome na retrorreferência 1
= # Delimitador nome-valor
( [^;\r\n]* ) # Captura o nome na retrorreferência 2
```

Opções Regex: ^ e $ correspondem em quebras de linha, espaçamento livre
Sabores Regex: .NET, Java, PCRE, Perl, Python, Ruby

Discussão

Tal como as outras receitas INI, neste capítulo, estamos trabalhando com ingredientes bastante simples. A expressão regular começa com ‹^›, para corresponder à posição no início de uma linha (certifique-se de que a opção “^ e $ correspondem em quebras de linha” esteja habilitada). Isso é importante porque, sem a garantia de que as correspondências realmente comecem no início de uma linha, você poderia corresponder a parte de uma linha de comentário.

Em seguida, a expressão regular utiliza um grupo de captura, que contém uma classe de caracteres negada (‹[^=;\r\n]›) seguida por um ou mais quantificadores (‹+›), para corresponder ao nome do parâmetro e lembrá-lo na retrorreferência 1. A classe negada corresponde a qualquer caractere, exceto aos quatro seguintes: sinal de igual, ponto-e-vírgula, retorno de carro (‹\r›) e alimentação de linha (‹\n›). Os caracteres de retorno de carro e alimentação de linha são utilizados para terminar um parâmetro INI, um ponto-e-vírgula marca o início de um comentário e um sinal de igual separa o nome e o parâmetro de um valor.

Depois de corresponder ao nome do parâmetro, a expressão regular corresponde a um sinal de igual literal (o delimitador nome-valor) e,

então, ao valor do parâmetro. O valor é correspondido usando um segundo grupo de captura, semelhante ao padrão empregado para corresponder ao nome do parâmetro, mas com duas restrições a menos. Em primeiro lugar, este segundo subpadrão permite corresponder a sinais de igual como sendo parte do valor (ou seja, existe um caractere a menos sendo negado na classe de caracteres). Em segundo, ele usa um quantificador ‹*›, para eliminar a necessidade de corresponder a pelo menos um caractere (uma vez que, ao contrário de nomes, valores podem estar vazios).

E, assim, terminamos.

Veja também:

A receita 8.12 explica como combinar cabeçalhos de seção INI.

A receita 8.13 dá os detalhes dos blocos da seção INI.

---

1 <xmp> é um elemento pouco conhecido, mas amplamente suportado, semelhante ao <pre>. Como o <pre>, ele preserva todos os espaços em branco e utiliza uma fonte de largura fixa por padrão, mas dá um passo adiante, e mostra todo seu conteúdo (incluindo tags HTML) como texto simples. O <xmp> foi tornado obsoleto no HTML 3.2, e inteiramente retirado do HTML 4.

2 O PowerGREP, descrito em “Ferramentas para se trabalhar com expressões regulares”, no capítulo 1, é uma ferramenta capaz de pesquisar dentro de correspondências.

Sobre os autores

Expressões Regulares Cookbook foi escrito por **Jan Goyvaerts** e **Steven Levithan**, dois dos maiores especialistas do mundo em expressões regulares.

Jan Goyvaerts comanda a Just Great Software, onde projeta e desenvolve alguns dos softwares mais populares relativos a expressões regulares. Seus produtos incluem o RegexBuddy, único editor de expressão regular no mundo que emula as peculiaridades de 15 sabores de expressões regulares, e o PowerGREP, a mais completa ferramenta grep para Microsoft Windows.

Steven Levithan é um dos principais peritos em expressão regular JavaScript, e comanda um blog sobre expressões regulares em *http://blog.stevenlevithan.com*. Expandir seu conhecimento sobre este sabor de expressões regulares e leitura ao ar livre (library landscape) têm sido alguns de seus hobbies, nestes últimos anos.

Informações finais

A imagem na capa do *Expressões Regulares Cookbook* é um musaranho-almiscareiro (gênero *Crocidura*, família *Muridae*). Existem vários tipos de musaranhos-almiscareiros, incluindo musaranhos-de-dentes-brancos, musaranhos-de-dentes-vermelhos, musaranhos-almiscareiros cinzas e musaranhos-almiscareiros vermelhos. O roedor é nativo da África do Sul e da Índia.

Apesar das várias características físicas que distinguem um tipo de musaranho de outro, todos compartilham alguns atributos. Por exemplo, os musaranhos são tidos como os menores insetívoros do mundo; todos têm pernas curtas, cinco garras em cada pé e um focinho alongado, com pelos tácteis. As diferenças incluem variações de cor entre seus dentes (de forma mais notória entre os apropriadamente denominados musaranhos-de-dentes-brancos e musaranhos-de-dentes-vermelhos) e na cor da pele, que varia do vermelho, para o castanho e o cinza.

Embora o musaranho geralmente cace insetos, ele também ajuda os agricultores a manter certas pragas sob controle, comendo ratos, ou outros pequenos roedores, em seus campos.

Muitos musaranhos-almiscareiros desprendem um forte odor de almíscar (daí o nome), que usam para marcar território. Antigamente, havia rumores de que o cheiro do musaranho-almiscareiro era tão forte que impregnaria as garrafas de vinho ou de cerveja por onde passasse, dando assim às bebidas um certo odor almiscarado, mas esse boato tem se provado uma farsa.

A imagem da capa é da *Lydekker's Royal Natural History*.


5ª EDIÇÃO
Revisada e ampliada
EXPRESSÕES REGULARES
UMA ABORDAGEM DIVERTIDA
(bb|[^b]{2})
[ER]
novatec
Aurelio Marinho Jargas
www.aurelio.net

Expressões Regulares - 5ª edição

Jargas, Aurelio Marinho
9788575224755
248 páginas

Você procura uma sigla em um texto longo, mas não lembra direito quais eram as letras. Só lembra que era uma sigla de quatro letras. Simples, procure por [A-Z]{4}. Revisando aquela tese de mestrado, você percebe que digitou errado o nome daquele pesquisador alemão famoso. E foram várias vezes. Escreveu Miller, Mueller e Müler, quando na verdade era Müller. Que tal corrigir todos de uma vez? Fácil, use a expressão M(i|ue|ü)ll?er. Que tal encontrar todas as palavras repetidas repetidas em seu texto? Ou garantir que há um espaço em branco após todas as vírgulas e os pontos finais? Se você é programador, seria bom validar dados em um único passo, não? Endereço de e-mail, número IP,

telefone, data, CEP, CPF... Chega de percorrer vetores e fazer checagens "na mão". Estes são exemplos de uso das Expressões Regulares, que servem para encontrar rapidamente trechos de texto informando apenas o seu formato. Ou ainda pesquisar textos com variações, erros ortográficos e muito mais! Visite o site do livro: www.piazinho.com.br

O'REILLY

Padrões para Kubernetes

Elementos reutilizáveis no design de aplicações nativas de nuvem

Bilgin Ibryam
Roland Huß

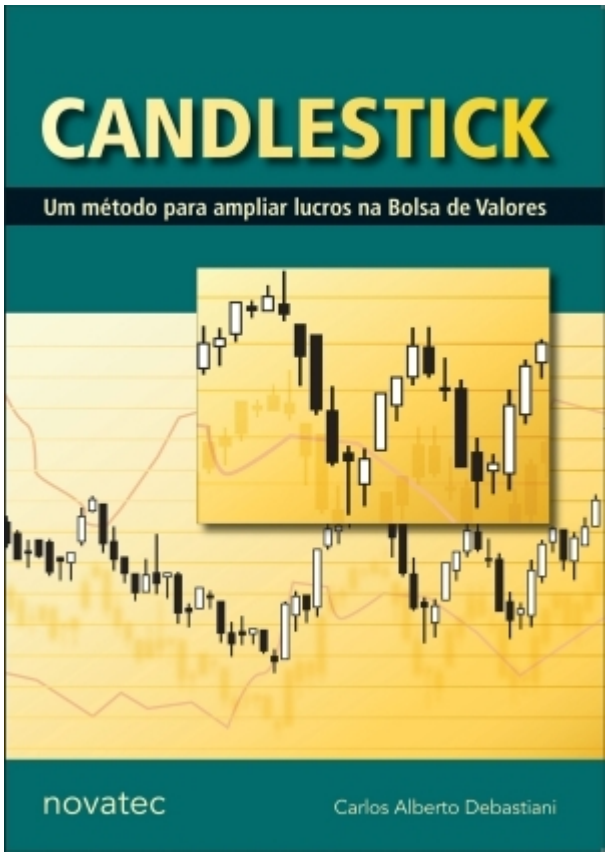
novatec

Padrões para Kubernetes

Ibryam, Bilgin
9788575228159
272 páginas

O modo como os desenvolvedores projetam, desenvolvem e executam software mudou significativamente com a evolução dos microsserviços e dos contêineres. Essas arquiteturas modernas oferecem novas primitivas distribuídas que exigem um conjunto diferente de práticas, distinto daquele com o qual muitos desenvolvedores, líderes técnicos e arquitetos estão acostumados. Este guia apresenta padrões comuns e reutilizáveis, além de princípios para o design e a implementação de aplicações nativas de nuvem no Kubernetes. Cada padrão inclui uma descrição do problema e uma solução específica no Kubernetes. Todos os padrões acompanham e são demonstrados por

exemplos concretos de código. Este livro é ideal para desenvolvedores e arquitetos que já tenham familiaridade com os conceitos básicos do Kubernetes, e que queiram aprender a solucionar desafios comuns no ambiente nativo de nuvem, usando padrões de projeto de uso comprovado. Você conhecerá as seguintes classes de padrões: • Padrões básicos, que incluem princípios e práticas essenciais para desenvolver aplicações nativas de nuvem com base em contêineres. • Padrões comportamentais, que exploram conceitos mais específicos para administrar contêineres e interações com a plataforma. • Padrões estruturais, que ajudam você a organizar contêineres em um Pod para tratar casos de uso específicos. • Padrões de configuração, que oferecem insights sobre como tratar as configurações das aplicações no Kubernetes. • Padrões avançados, que incluem assuntos mais complexos, como operadores e escalabilidade automática (autoscaling).

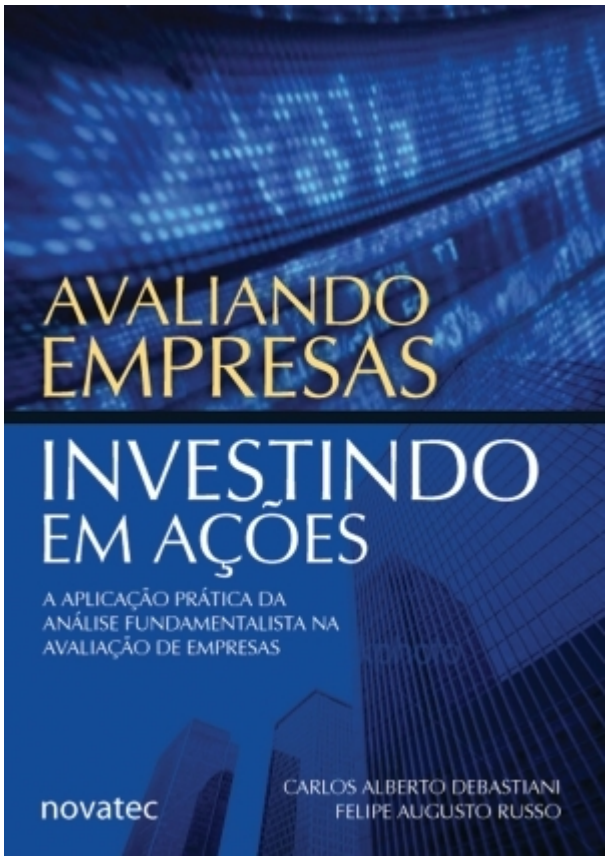

CANDLESTICK
Um método para ampliar lucros na Bolsa de Valores
novatec
Carlos Alberto Debastiani

Candlestick

Debastiani, Carlos Alberto
9788575225943
200 páginas

A análise dos gráficos de Candlestick é uma técnica amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos. Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com

sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.


AVALIANDO
EMPRESAS
INVESTINDO
EM AÇÕES
A APLICAÇÃO PRÁTICA DA
ANÁLISE FUNDAMENTALISTA NA
AVALIAÇÃO DE EMPRESAS
novatec
CARLOS ALBERTO DEBASTIANI
FELIPE AUGUSTO RUSSO

Avaliando Empresas, Investindo em Ações

Debastiani, Carlos Alberto
9788575225974
224 páginas

Avaliando Empresas, Investindo em Ações é um livro destinado a investidores que desejam conhecer, em detalhes, os métodos de análise que integram a linha de trabalho da escola fundamentalista, trazendo ao leitor, em linguagem clara e acessível, o conhecimento profundo dos elementos necessários a uma análise criteriosa da saúde financeira das empresas, envolvendo indicadores de balanço e de mercado, análise de liquidez e dos riscos pertinentes a fatores setoriais e conjunturas econômicas nacional e internacional. Por meio de exemplos práticos e ilustrações, os autores exercitam os conceitos teóricos abordados, desde os

fundamentos básicos da economia até a formulação de estratégias para investimentos de longo prazo.


Marcos Abe
MANUAL DE
ANÁLISE
TÉCNICA
ESSÊNCIA E ESTRATÉGIAS AVANÇADAS
TUDO O QUE UM INVESTIDOR PRECISA SABER PARA
PROSPERAR NA BOLSA DE VALORES ATÉ EM TEMPOS DE CRISE
novatec

Manual de Análise Técnica

Abe, Marcos
9788575227022
256 páginas

Este livro aborda o tema Investimento em Ações de maneira inédita e tem o objetivo de ensinar os investidores a lucrarem nas mais diversas condições do mercado, inclusive em tempos de crise. Ensinará ao leitor que, para ganhar dinheiro, não importa se o mercado está em alta ou em baixa, mas sim saber como operar em cada situação. Com o Manual de Análise Técnica o leitor aprenderá: - os conceitos clássicos da Análise Técnica de forma diferenciada, de maneira que assimile não só os princípios, mas que desenvolva o raciocínio necessário para utilizar os gráficos como meio de interpretar os movimentos da massa de investidores do mercado; - identificar oportunidades para lucrar na bolsa de valores, a

longo e curto prazo, até mesmo em mercados baixistas; um sistema de investimentos completo com estratégias para abrir, conduzir e fechar operações, de forma que seja possível maximizar lucros e minimizar prejuízos; - estruturar e proteger operações por meio do gerenciamento de capital. Destina-se a iniciantes na bolsa de valores e investidores que ainda não desenvolveram uma metodologia própria para operar lucrativamente.