Kapitola 3

Programování s regulárními výrazy

Programovací jazyky a regulární výrazy

V této kapitole si vysvětlíme, jak regulární výrazy zapracovat do vámi zvoleného programovacího jazyka. V receptech zde uvedených předpokládáme, že již máte k dispozici fungující regulární výraz. V tomto ohledu vám mohou pomoci předcházející kapitoly. Nyní před vámi stojí úkol začlenit tento regulární výraz do zdrojového kódu, aby mohl konečně něco vykonávat.

V této kapitole se snažíme vysvětlit, jak který úsek kódu pracuje, a proč je tomu tak. S ohledem na úroveň podrobnosti, kterou tato kapitola vykazuje, by mohlo být čtení celého textu poněkud otravné. Jestli pročítáte *Kuchařku regulárních výrazů* poprvé, doporučujeme vám si tuto kapitolu zběžně projít, abyste si udělali představu, co vše je zapotřebí. Až budete později implementovat regulární výraz z následujících kapitol, vraťte se. Naučíte se zaintegrovat regulární výrazy do vámi zvoleného programovacího jazyka.

V kapitolách 4 až 8 budeme popisovat regulární výrazy používané k řešení skutečných problémů. Tyto kapitoly se zaměřují na samotné regulární výrazy a v mnoha receptech není zobrazen vůbec žádný zdrojový kód. Chcete-li, aby vám některý z regulárních výrazů uvedený v těchto kapitolách fungoval, zapojte je jednoduše do úryvků kódu nacházejících se v této kapitole.

Protože se další kapitoly soustředí na regulární výrazy, předvádí spíše řešení konkrétních typů regulárních výrazů než řešení v konkrétních programovacích jazycích. Regulární výrazy s programovacími jazyky přímo nekorespondují. Skriptovací jazyky mívají zabudovány své vlastní typy regulárních výrazů, zatímco ostatní jazyky se ve věci podpory regulárních výrazů spoléhají na knihovny. Některé knihovny jsou přístupné ve více jazycích, ovšem některé jazyky pracují s více knihovnami, které v jiných jazycích nefungují.

Oddíl "Regulární výrazy v mnoha příchutích" na straně 14 popisuje všechny druhy regulárních výrazů, kterým se v knize věnujeme. Na straně 17 uvádíme v oddíle Náhradní text ve více příchutích" seznam druhů nahrazovacího textu užívaných při vyhledávání a nahrazování prostřednictvím regulárního výrazu. Všechny programovací jazyky, jež v této kapitole zmiňujeme, používají některý z těchto druhů.

Programovací jazyky této kapitoly

Tato kapitola popisuje sedm programovacích jazyků. Každý recept obsahuje samostatná řešení ve všech sedmi programovacích jazycích a mnohé recepty ve všech těchto jazycích nabízí i oddělené popisy řešení. Je-li způsob řešení aplikovatelný na více než jeden jazyk, budeme ho v popisu daného jazyku opakovat. To proto, abyste mohli bezpečně přeskakovat popisy řešení v programovacích jazycích, které vás nezajímají:

C#

Jazyk C# pracuje s rozhraním Microsoft .NET. Třídy `System.Text.RegularExpressions` používají regulární výrazy a nahrazovací texty .NET. V knize popisujeme jazyk C# až do verze 3.5, nebo Visual Studio 2002 až do verze 2008.

VB.NET

V knize používáme jazyků VB.NET a Visual Basic.NET při odkazování na Visual Basic 2002 a pozdější, abychom tyto verze rozeznali od verze Visual Basic 6 a dřívějších. Visual Basic nyní pracuje s rozhraním Microsoft .NET. Třídy `System.Text.RegularExpressions` používají regulární výrazy a nahrazovací texty .NET. V knize popisujeme verze Visual Basic 2002 až 2008.

Java

Java 4 je prvním vydáním jazyka Java, které díky balíku `java.util.regex` nabízí zabudovanou podporu regulárních výrazů. Balík `java.util.regex` používá „javovské“ regulární výrazy a nahrazování textu. V knize se věnujeme verzím 4, 5 a 6.

JavaScript

Tento druh regulárních výrazů se používá v programovacím jazyce, který je běžně znám pod názvem JavaScript. Implementují ho všechny moderní webové prohlížeče: Internet Explorer (od verze 5.5), Firefox, Opera, Safari a Chrome. Mnoho dalších aplikací používá JavaScript jako skriptovací jazyk.

V této knize, stručně řečeno, používáme termín *JavaScript* tehdy, když chceme označit programovací jazyk definovaný ve třetí verzi standardu ECMA-262. Tento standard definuje programovací jazyk ECMAScript, který je v různých webových prohlížečích lépe znám díky svým implementacím JavaScript a JScript.

PHP

PHP nabízí regulárním výrazům tři sady funkcí. Důrazně doporučujeme funkce `preg`. Proto se v této knize budeme věnovat právě pouze těmto funkcím, které jsou do PHP zabudovány již od verze 4.2.0. My zde pokryjeme verze 4 a 5. Funkce `preg` jsou obaly jazyka PHP okolo knihovny PCRE. S regulárními výrazy druhu PCRE v knize pracujeme jako s „PCRE“. PCRE neobsahuje funkce pro vyhledávání a nahrazování textu, takže si vývojáři jazyka PHP vytvořili pro funkci `preg_replace` svou vlastní syntaxi nahrazování textu. Tento druh nahrazování textu v knize označujeme jako „PHP“.

Funkce `mb_ereg` jsou součástí „vícebajtových“ funkcí jazyka PHP, jež jsou navrženy tak, aby pracovaly dobře s jazyky tradičně kódovanými vícebajtovými znakovými sadami, jako například japonština či čínština. Ve verzi PHP 5 používají funkce `mb_ereg` knihovnu regulárních výrazů Oniguruma, která byla původně vyvíjena pro Ruby. V naší knize označujeme druh Oniguruma jako „Ruby 1.9“. Používat funkce mb_ereg se doporučuje pouze tehdy, když máte při práci s vícebajtovými kódovými stránkami zvláštní požadavky, a když jste již s funkcemi `mb_` v PHP seznámeni.

Skupina funkcí ereg je v jazyce PHP nejstarší sadou funkcí regulárních výrazů a od verze PHP 5.3.0 jsou oficiálně prohlášeny za zastaralé. Tyto funkce nejsou závislé na externích knihovnách, implementují druh POSIX ERE. Tato skupina nabízí pouze omezenou sadu funkcí a v knize se jí nevěnujeme. POSIX ERE je přímou podmnožinou sad Ruby 1.9 a PCRE. Regulární výraz lze převzít z libovolného volání funkce `ereg` a použít ho spolu s funkcemi `mb_ereg` či `preg`. Ve funkci `preg` je třeba přidat ještě perlovské oddělovače (recept 3.1).

Perl

Zabudovaná podpora regulárních výrazů je v Perlu hlavním důvodem, proč jsou regulární výrazy dnes oblíbené. O skupině regulárních výrazů a nahrazování textu, která pracuje s Perl operátory `m//` a `s///` v této knize hovoříme jako o jazyce „Perl“. Popsány jsou verze 5.6, 5.8 a 5.10.

Python

Jazyk Python podporuje regulární výrazy prostřednictvím svého modulu `re`. Skupiny regulárních výrazů a nahrazování textu, které tento modul používá, v knize označujeme jako jazyk „Python“. V knize popisujeme verze Python 2.4 a 2.5.

Ruby

Jazyk Ruby má podporu regulárních výrazů zabudovanou. V knize se věnujeme verzím Ruby 1.8 a Ruby 1.9. Tyto dvě verze jazyka Ruby používají implicitně odlišné stroje regulárních výrazů. Ruby 1.9 pracuje se strojem Oniguruma, jenž nabízí více funkcí regulárních výrazů než klasický stroj verze Ruby 1.8. Na straně 15 v oddíle “Příchuti regulárních výrazů“ najdete další podrobnosti.

V této kapitole nebudeme o rozdílech mezi Ruby 1.8 a Ruby 1.9 příliš hovořit. Regulární výrazy v této kapitole jsou základními regulárními výrazy a nové funkce z verze Ruby 1.9 nepoužívají. Protože je podpora regulárních výrazů zabudována i do samotného jazyka Ruby, bude kód Ruby, kterým regulární výrazy implementujete, stejný, a nebude přitom záležet na tom, zda jste Ruby kompilovali pomocí klasického stroje regulárních výrazů, nebo prostřednictvím stroje Oniguruma. Budete-li funkce stroje Oniguruma potřebovat ve verzi Ruby 1.8, můžete ji překompilovat.

Další programovací jazyky

Programovací jazyky uvedené na seznamu níže nejsou v této knize popsány, ale používají jeden z druhů regulárních výrazů, které v knize postiženy jsou. Jestli v jednom z těchto jazyků pracujete, můžete tuto kapitolu přeskočit, ale všechny ostatní kapitoly se vám budou hodit:

ActionScript

ActionScript je implementace standardu ECMA-262 od společnosti Adobe. Od verze 3.0 má jazyk ActionScript plnou podporu regulárních výrazů standardu ECMA-262v3. Tuto skupinu v knize označujeme jako jazyk „JavaScript“. Jazyk ActionScript je současně i velice blízko jazyku JavaScript. Příklady jazyka JavaScript uvedené v této kapitole byste měli být schopni upravit pro jazyk ActionScript.

C

Jazyk C může používat širokou škálu knihoven regulárních výrazů. Otevřená (open source) knihovna PCRE bude ze všech druhů popsaných v této knize zřejmě nejlepší volbou. Kompletní zdrojový kód jazyka C si můžete stáhnout na adrese *http://www.pcre.org*. Kód je napsán tak, aby kompiloval v široké škále kompilátorů na velkém množství platforem.

C++

Jazyk C++ umí pracovat s řadou knihoven regulárních výrazů. Otevřená (open source) knihovna PCRE bude ze všech druhů popsaných v této knize zřejmě nejlepší volbou. Buď můžete použít přímo rozhraní API jazyka C, nebo baliče (wrappery) třídy C++ obsažené v samotném stáhnutelném PCRE (viz adresa *http://www.pcre.org*.)

V systému Windows můžete importovat objekt RegExp COM jazyka VBScript 5.5, jak si později vysvětlíme na jazyce Visual Basic 6. To se může hodit k udržení konzistence regulárního výrazu mezi serverovým systémem (back-endem) v jazyce C++ a klientem (frontendem) v jazyce JavaScript.

Delphi for Win 32

V době, kdy byla psána tato kniha, nenabízel jazyk Delphi ve verzi Win32 žádnou podporu regulárních výrazů. Knihovna VCL má mnoho dostupných součástí, které nabízí podporu regulárních výrazů. Doporučil bych vám vybrat si takovou, jež je založena na PCRE. Delphi umí do aplikací napojovat objektové soubory jazyka C a mnohé baliče VCL pro prostředí PCRE tyto souborové soubory využívají. Můžete tak aplikaci uchovávat v podobě jediného souboru `.exe`.

Na adrese *http://www.regexp.info/del phi.html* si můžete stáhnout mou vlastní komponentu `TPerlRegEx`. Jedná se o VCL komponentu, která se nainstaluje do palety komponent, takže ji můžete snadno přetahovat do formulářů. Dalším oblíbeným PCRE baličem pro prostředí Delphi je třída `TJclRegEx`, což je součást knihovny JCL, kterou si můžete stáhnout na adrese *http://www.delphi-jedi.org*. Třída `TJclRegEx` vychází ze třídy `TObject`, takže ji do formuláře přenášet nemůžete.

Obě knihovny jsou otevřené a šíří se pod licencí Mozilla Public License.

Delphi Prism

V jazyce Delphi Prism můžete využívat podpory regulárních výrazů, kterou nabízí rozhraní .NET. Jednoduše vložíte třídu `System.Text.RegularExpressions` do klauzule `uses` libovolné jednotky v jazyce Delphi Prism, v níž chcete regulární výrazy používat.

Jakmile to uděláte, můžete používat stejné principy, jaké uvidíte v této kapitole ve výtažcích kódu v jazycích C# a VB.NET.

Groovy

Díky balíku `java.util.regex` můžete v jazyce Groovy používat regulární výrazy stejně jako v jazyce Java. Vlastně i všechna řešení v jazyce Java uvedená v této kapitole fungují současně i v jazyce Groovy. Vlastní syntaxe regulárních výrazů jazyka Groovy nabízí pouze notační zkratky. Literální výraz členěný lomítky je instancí třídy `java.lang.String` a operátor =~ vytváří instanci třídy `java.util.regex.Matcher`. Syntaxi jazyka Groovy lze volně míchat se standardní syntaxí jazyka Java – třídy a objekty jsou shodné.

PowerShell

PowerShell je prostředí příkazového řádku se skriptovacím jazykem založený na rozhraní .NET. V jazyce PowerShell zabudované operátory `-match` a `-replace` používají regulární výrazy a nahrazování textu typu .NET, jak je v knize popsáno.

R

Jazyk R Project regulární výrazy podporuje prostřednictvím funkcí `grep`, `sub` a `regexpr` obsažených v balíku `base`. Všechny tyto funkce pojímají argument označený jako `perl`. Když ho neuvedete, bude mít hodnotu `FALSE`. Když ho nastavíte na hodnotu `TRUE`, budete moci regulární výrazy PCRE používat tak, jak si popisujeme v této knize. Regulární výrazy uvedené u jazyka PCRE 7 fungují v jazyce R ve verzi 2.5.0 a novějších. U dřívějších verzí jazyka R používejte regulární výrazy, které v této knize označujeme jako „PCRE 4 a novější“. „Základní“ a „rozšířené“ druhy, které jazyk R podporuje, jsou starší a omezenější a v naší knize se jim nevěnujeme.

REALbasic

REALbasic má zabudovanou třídu `RegEx`. Interně užívá tato třída UTF-8 verzi knihovny PCRE. To znamená, že můžete využít podpory Unicode, kterou PCRE nabízí, ale při konvertování ne-ASCII textu do kódování UTF-8 budete muset před předáním textu třídě `RegEx` použít třídu jazyka REALbasic, `TextConverter`.

Všechny regulární výrazy v jazyce PCRE 6, které v knize uvádíme, budou fungovat i v jazyce REALbasic. Je však třeba varovat, že v jazyce REALbasic jsou volby „nerozlišování malých a velkých znaků“ a „znaku stříšky a dolaru zastupujících zalomení řádku“ („víceřádkový režim“) nastaveny implicitně. Když budete chtít použít regulární výraz uvedený v této knize, který neříká, jak tyto režimy hledání shody aktivovat, musíte je v jazyce REALbasic aktivovat explicitně.

Scala

Jazyk Scala nabízí podporu regulárních výrazů zabudovanou prostřednictvím balíku `scala.util.matching`. Tato podpora se zakládá na stroji regulárních výrazů obsaženém v Java balíku `java.util.regex`. Typy regulárních výrazů a nahrazovacích textů používaných jazyky Java a Scala označujeme v této knize jako „Java“.

Visual Basic 6

Visual Basic 6 je poslední verzí jazyka Visual Basic, která nevyžaduje rozhraní .NET. Rovněž to však znamená, že Visual Basic 6 nemůže využívat vynikající podporu, jež toto rozhraní nabízí. Vzorky kódu VB.NET nebudou v jazyce VB6 vůbec fungovat.

Visual Basic 6 velmi usnadňuje použití funkcí poskytovaných knihovnami ActiveX a COM. Jednou z těchto knihoven je skriptovací knihovna VBScript od Microsoftu, která již do verze 5.5 umí s regulárními výrazy zacházet velice slušně. Tato skriptovací knihovna implementuje stejný druh regulárních výrazů, které se užívají v jazyce JavaScript, a které jsou uvedeny ve standardu ECMA-262v3. Je součástí prohlížeče Internet Explorer 5.5 a jeho novějších verzí. Dostupná je na všech počítačích běžících pod systémy Windows XP či Vista, ale i na starších systémech Windows, pakliže je na nich přítomen prohlížeč IE 5.5 nebo novější. Spadají sem tedy téměř všechny počítače se systémem Windows, jež se připojují k internetu.

Když budete tuto knihovnu ve své aplikaci v jazyce Visual Basic použít, vyhledejte a označte v nabídce položku **Project|References**. Posuňte se v seznamu níže a najděte položku **“Microsoft VBScript Regular Expressions 5.5“**, která se nachází přímo pod položkou **“Microsoft VBScript Regular Expressions 1.0“**. Dejte si pozor, abyste označili verzi 5.5, ne 1.0. Verze 1.0 je uvedena jen z důvodů zpětné kompatibility a její funkce rozhodně nejsou dostačující.

Jakmile odkaz vložíte, uvidíte, které třídy a členy tříd knihovna nabízí. V nabídce vyberte položku **Select View|Object Browser**. V prohlížeči objektů (Object Browser) označte v rozbalovací nabídce v levém horním rohu položku **“VBScript_RegExp_55“**.

3.1 Literální regulární výrazy ve zdrojovém kódu

Úloha

Jako řešení problému máte k dispozici regulární výraz `<[$"'\n\d/\\]>`. Tento regulární výraz sestává z jediné znakové třídy odpovídající znaku dolaru, dvojité uvozovce, jednoduché uvozovce, symbol pro posun řádku, libovolnou číslici v rozmezí 0 až 9, lomítko nebo zpětné lomítko. Vy budete tento regulární výraz do zdrojového kódu zakódovávat napevno v podobě konstanty řetězce či operátoru regulárního výrazu.

Řešení

C#

Jako normální řetězec:

```
"[$\"'\n\\d/\\\\]"
```

Jako doslovný řetězec:

```
@"[$""'\n\d/\\]"
```

VB.NET

```
"[$""'\n\d/\\]"
```

Java

```
"[$\"'\n\\d/\\\\]"
```

JavaScript

```
/[$"'\n\d\/\\]/
```

PHP

```
'%[$"\'\n\d/\\\\]%'
```

Perl

Operátor porovnání podle vzorů

```
/[\$"'\n\d\/\\]/
m![\$"'\n\d/\\]!
```

Substituční operátor:

```
s![\$"'\n\d/\\]!!
```

Python

Surový řetězec se třemi uvozovkami:

```
r"""[$"'\n\d/\\]"""
```

Normální řetězec:

```
"[$\"'\n\\d/\\\\]"
```

Ruby

Literální regulární výraz oddělený lomítky:

```
/[$"'\n\d\/\\]/
```

Literální regulární výraz oddělený vámi zvolenými interpunkčními znaky:

```
%r![$"'\n\d/\\]!
```

Popis řešení

Když vám v knize předvádíme samotný regulární výraz (ve srovnání s částí delšího úryvku zdrojového kódu), vždy vám ho ukazujeme bez všelijakých pozlátek. Tento výraz tvoří jedinou

výjimku. Při práci s testerem regulárních výrazů, jakým jsou RegexBuddy nebo RegexPal, byste zadávali regulární výraz takto. Přijímá-li aplikace regulární výraz ve formě uživatelského vstupu, uživatel by ho zadal v této podobě.

Když ale budete chtít regulární výraz do zdrojového kódu vložit napevno, budete muset udělat ještě něco. Při bezstarostném kopírování a vkládání regulárních výrazů z testeru do zdrojového kódu – anebo obráceně – si často budete nervózně vjíždět prsty do vlasů a budete se divit, jak je možné, že regulární výraz v nástroji funguje, ale ve zdrojovém kódu ne, případně proč tester selhává u regulárního výrazu, který jste zkopírovali z něčího kódu. Všechny programovací jazyky, které si v knize popisujeme, vyžadují, aby byly literální regulární výrazy nějakým způsobem uvozeny. Některé jazyky vyžadují řetězce, zatímco jiné požadují speciální konstanty regulárního výrazu. Jestli se ve vašem regulárním výrazu již nějaké oddělovače, nebo jiné znaky se zvláštním významem, nachází, budete k nim muset přidat uvozovací znaky.

Nejčastěji používaným uvozovacím znakem je zpětné lomítko. Proto ve většině řešení úlohy vidíte mnohem více zpětných lomítek, než ta čtyři, která se nachází v původním regulárním výrazu.

C#

V jazyce C# lze literální regulární výrazy předávat konstruktoru `Regex()` a mnoha členským funkcím ve třídě `Regex`. Parametr, který regulární výraz přijímá, je vždy deklarován jako řetězec.

C# podporuje dva druhy řetězcových literálů. Nejběžnější je řetězec uvozený dvojitými uvozovkami, který je dobře známý z jazyků, jakými jsou C++ a Java. V takových řetězcích je nutné uvozovky a zpětná lomítka uvodit zpětným lomítkem. Podporu mají v řetězcích i uvození netisknutelných znaků, například `<\n>`. Při práci se třídou `RegexOptions.IgnorePatternWhitespace` (viz recept 3.4) je při aktivaci režimu free-space (přehlížení bílých znaků) mezi řetězci `"\n"` a `"\\n"` rozdíl, viz recept 2.18. `"\n"` je řetězec s literálním zalomením řádku, který je přehlížen jako bílý znak. `"\\n"` je řetězec s tokenem regulárního výrazu `<\n>`, jenž odpovídá novému řádku.

Doslovné řetězce začínají znakem zavináče a dvojitou uvozovkou a končí samotnou dvojitou uvozovkou. Chcete-li dvojitou uvozovku použít jako doslovný řetězec, zdvojte ji. Zpětná lomítka uvozovat nutné není, navíc tím docílíte toho, že bude regulární výraz mnohem snáze čitelný. Řetězec `@"\n"` vždy představuje token `<\n>` odpovídající novému řádku, dokonce i v režimu přehlížení bílých znaků (free-spacing). Doslovné řetězce na této úrovni token `<\n>` nepodporují, ale mohou se rozpínat na mnoha řádcích. Proto jsou doslovné řetězce pro regulární výrazy, které přehlíží bílé znaky, ideální.

Správná volba je zřejmá: při zadávání regulárních výrazů do zdrojového kódu v jazyce C# používejte doslovné řetězce.

VB.NET

V jazyce VB.NET lze literální regulární výrazy předávat konstruktoru `Regex()` a mnoha členským funkcím ve třídě `Regex`. Parametr, který regulární výraz přijímá, je vždy deklarován jako řetězec.

Jazyk Visual Basic používá dvojité uvozovky. Dvojité uvozovky v řetězci musí být zdvojené. Žádné jiné znaky není třeba uvozovat.

Java

V jazyce Java lze literální regulární výrazy předávat faktorii třídy `Pattern.compile()` a dalším různým funkcím třídy `String`. Parametr, který zachytává regulární výraz, je vždy deklarován jako řetězec.

Java pracuje s řetězci uvozenými dvojitými uvozovkami. V těchto řetězcích, musí být dvojité uvozovky a zpětná lomítka uvozeny zpětným lomítkem. Uvození netisknutelných znaků, jakými je `<\n>`, a uvození standardu Unicode, například `<\uFFFF>`, jsou v řetězcích podporována také.

Při práci se třídou `Pattern.COMMENTS` (viz recept 3.4) se při aktivaci režimu přehlížení bílých znaků (free-space) výrazy `"\n"` a `"\\n"` liší, což je vysvětleno v receptu 2.18. `"\n"` je řetězcem s literálním zalomením řádku, které je přehlíženo stejně jako bílý znak. `"\\n"` je řetězcem s tokenem regulárního výrazu `<\n>`, který značí nový řádek.

JavaScript

V jazyce JavaScript se regulární výrazy vytváří nejlépe s použitím speciální syntaxe určené k deklarování literálních regulárních výrazů. Jednoduše umístíte regulární výraz mezi dvě lomítka. Když se někde v regulárním výrazu vyskytne samostatné lomítko, uvedete ho zpětným lomítkem.

I když je možné objekt `RegExp` vytvořit z řetězce, používat notaci řetězce v kódu pro účely literálních regulárních výrazů by bylo skoro nesmyslné. Uvozovky byste museli uvést zpětnými lomítky a obyčejně byste tak dali vyrůst celému lesu zpětných lomítek.

PHP

Literální regulární výrazy určené k použití s funkcí `preg` jazyka PHP jsou zvláštní pomůckou. Na rozdíl od jazyků JavaScript a Perl nemá PHP žádný nativní typ regulárních výrazů. Regulární výrazy je vždy třeba citovat ve formě řetězců. Totéž platí i o funkcích `ereg` a `mb_ereg`. Ve snaze napodobit Perl však vývojáři PHP balících funkcí pro PCRE přidali ještě jeden požadavek.

V řetězci je třeba regulární výraz citovat jako perlovský literální regulární výraz. To znamená, že tam, kde byste v jazyce Perl zapsali výraz `/regex/`, se řetězci funkce `preg` v PHP dostává podoby `'/regex/'`. Stejně jako v jazyce Perl, i zde můžete na místě oddělovačů používat různé páry interpunkčních znamének. Když se v regulárním výrazu objeví oddělovač, musí být uvozen zpětným lomítkem. Chcete-li se tomu vyhnout, zvolte si oddělovač, který se v regulárním výrazu neobjevuje. V tomto receptu jsem použil znak procenta, protože lomítko se ve výrazu objevuje, zatímco znak procenta nikoliv. Kdyby se v regulárním výrazu lomítko nevyskytovalo, použijte ho, v jazyce Perl se jedná o nejběžněji užívaný oddělovač a v jazycích JavaScript a Ruby je dokonce vyžadován.

Jazyk PHP podporuje řetězce uvozené jednoduchými i dvojitými uvozovkami. U obou je nutné uvozovky (jednoduché i dvojité) a zpětné lomítko v regulárním výrazu uvést zpětným lomítkem. V řetězcích s dvojitými uvozovkami je nutné uvozovat i znak dolaru. Jestli opravdu nechcete proměnné v regulárním výrazu interpolovat, měli byste používat řetězce s jednoduchými uvozovkami.

Perl

V jazyce Perl se literální regulární výrazy používají s operátory porovnání podle vzorů a se substitučními operátory. Operátor porovnání sestává ze dvou lomítek ohraničujících regulární výraz. Lomítka v regulárním výrazu musí být uvozena zpětným lomítkem. Další znaky, s výjimkou znaků `$` a `@`, jak je vysvětleno na konci tohoto pododdílu, není třeba uvozovat.

Alternativní notace operátoru porovnání umístí regulární výraz mezi pár interpunkčních znamének, kterým bude předcházet písmeno `m`. Použijete-li jako oddělovače libovolná otevírací a zavírací znaménka (kulaté závorky, složené závorky), budou se muset shodovat: například `m{regulární_výraz}`. Jestli použijete odlišnou interpunkci, zadejte jednoduše stejný znak dvakrát. V řešení tohoto receptu používáme vykřičník. Nemusíme tak v regulárním výrazu uvozovat literální lomítko. Zpětným lomítkem je třeba uvodit pouze zavírací oddělovač.

Substituční operátor je podobný operátoru porovnání podle vzorů. Namísto písmene `m` začíná písmenem `s` a přidává nahrazovací text. Když jako oddělovače používáte závorky či podobná

interpunkční znaménka, budete potřebovat dva páry: `s[regulární_výraz][náhrada]`. U všech ostatních interpunkčních znaků použijete oddělovač třikrát: `s/regulární_výraz/náhrada/`.

Perl substituční operátory a operátory porovnání trasuje jako řetězce s dvojitými uvozovkami. Když zadáte `m/I am $name/` a proměnná `$name` obsahuje řetězec `"Jan"`, dostanete regulární výraz `<I•am•Jan>`. `$"` je v Perlu proměnnou také, takže literální znak dolaru jsme v regulárním výrazu v našem receptu museli uvést znakovou třídou.

Znak dolaru, který zamýšlíte použít jako záložku, nikdy neuvozujte (viz recept 2.5). Uvozený znak dolaru je vždy literálem. Perl je natolik chytrý, že umí rozlišit mezi znaky dolaru, které slouží jako odkazy, a znaky dolaru používané k interpolaci proměnných, protože záložky lze smysluplně nasadit pouze na konci skupiny, na konci celého regulárního výrazu, či před zalomením řádku. Jestli chcete kontrolovat, zda se regulární výraz („regex“) shoduje se zdrojovým řetězcem zcela, neměli byste ve výrazu `<m/^regex$/>` znak dolaru uvozovat.

Zavináč nemusí mít v regulárních výrazech žádný zvláštní význam, ale Perl ho používá při interpolaci proměnných. V kódu v jazyce Perl je ho třeba v literálních regulárních výrazech uvozovat, stejně jako řetězce s dvojitými uvozovkami.

Python

Funkce v modulu `re` jazyka Python předpokládají, že se literální regulární výrazy předávají ve formě řetězců. Ty je možné uvozovat kterýmkoliv z mnohých způsobů, jež Python nabízí. V závislosti na znacích, které se v regulárním výrazu nachází, mohou různé typy uvozování snížit počet znaků, jež bude třeba uvodit zpětnými lomítky.

Obecně jsou nejlepším řešením surové řetězce. Ty není v Pythonu třeba nijak uvozovat. Při práci se surovými řetězci nebudete muset zpětná lomítka v regulárním výrazu zdvojovat. Výraz `r"\d+"` se čte snáze než výraz `"\\d+"`, obzvláště pak v delších regulárních výrazech.

Jediným případem, kdy nejsou surové řetězce ideálním řešením, je ten, kdy se v regulárním výrazu nachází jak jednoduché, tak i dvojité uvozovky. Tehdy totiž není možné surový řetězec uvozovat párem jednoduchých či dvojitých uvozovek, protože neexistuje způsob, jak uvozovkami ohraničit i uvozovky nacházející se ve výrazu. V takovém případě lze použít surový řetězec s trojím uvozením, stejně jako jsme to v tomto receptu učinili v jazyce Python. Pro srovnání uvádíme i normální řetězec.

Budete-li ve svých regulárních výrazech chtít používat funkce standardu Unicode, které jsou vysvětleny v receptu 2.7, musíte pracovat s řetězci Unicode. Řetězec lze převést na řetězec Unicode tak, že před něj umístíte znak `u`.

Surové řetězce nepodporují netisknutelné znaky, jakým je například `\n`. S uvozovacími sekvencemi se zachází jako s literálním textem. U modulu `re` to žádné komplikace nezpůsobí. Modul tato uvození podporuje coby součást syntaxe regulárního výrazu a nahrazovacího textu. V regulárních výrazech a nahrazovacích textech se literál `\n` bude interpretovat jako zalomení řádku.

Při práci se třídou `re.VERBOSE` (viz recept 3.4) je při zapínání režimu přehlížení bílých znaků, který je popsán v receptu 2.18, rozdíl mezi řetězcem `"\n"` na jedné straně a řetězcem `"\\n"` a surovým řetězcem `r"\n"` na straně druhé. `"\n"` je řetězcem s literálním zalomením řádku, které je coby bílý znak ignorováno. `"\\n"` a `r"\n"` jsou oba řetězce s tokenem regulárního výrazu `<\n>`, jenž odpovídá zalomení řádku.

Při práci s režimem přehlížení bílých znaků (free-space) jsou nejlepším řešením třikrát uvozené surové řetězce, jako například `r"""\n"""`, protože mohou zasahovat na více řádků. Token `<\n>`

není navíc vykládán na úrovni řetězce, takže může být interpretován na úrovni regulárního výrazu a může značit zalomení řádku.

Ruby

V Ruby se regulární výrazy vytváří nejlépe prostřednictvím speciální syntaxe určené k deklarování literálních regulárních výrazů. Jednoduše svůj regulární výraz umístíte mezi dvě lomítka. Když se v samotném regulárním výrazu nějaká lomítka objeví, uvedete je zpětným lomítkem.

Jestli nechcete lomítka v regulárním výrazu uvozovat, můžete před regulární výraz zadat znaky `%r` a jako oddělovač pak použít libovolný interpunkční znak.

I když je z řetězce možné vytvořit objekt `Regexp`, užívat v kódu notaci řetězce na literální regulární výrazy moc rozumné není. Uvozovky byste totiž pak museli uvozovat zpětnými lomítky a ta by se vám obyčejně rozrostla do celého lesa zpětných lomítek.

V tomto ohledu se Ruby značně podobá jazyku JavaScript. Ovšem název třídy je v Ruby pouze jednoslovný, `Regexp`, zatímco v jazyce JavaScript je zkratkou dvou slov, `RegExp`.

Další informace

Recept 2.3 vysvětluje fungování znakových tříd a také to, proč jsou v regulárním výrazu zapotřebí dvě zpětná lomítka, když ve znakové třídě chcete použít jediné zpětné lomítko.

V receptu 3.4 vysvětlujeme, jak nastavit volby regulárního výrazu, což se v některých programovacích jazycích provádí v rámci literálního regulárního výrazu.

3.2 Import knihovny regulárních výrazů

Úloha

Abyste ve své aplikaci mohli používat regulární výrazy, budete do zdrojového kódu importovat knihovnu regulárních výrazů nebo jmenný prostor.

Zbytek vzorků zdrojového kódu, který se v knize nachází, předpokládá, že jste to již v případě potřeby provedli.

Řešení

C#

```
using System.Text.RegularExpressions;
```

VB.NET

```
Imports System.Text.RegularExpressions
```

Java

```
import java.util.regex.*;
```

Python

```
import re
```

Popis řešení

Některé programovací jazyky mají regulární výrazy zabudované. V těchto jazycích není podporu regulárních výrazů třeba nijak zprovozňovat. Ostatní jazyky umožňují funkce regulárních výrazů prostřednictvím knihovny, kterou je třeba do zdrojového kódu vložit pomocí příkazu. Některé jazyky nenabízí regulárním výrazům podporu vůbec žádnou. V nich budete muset tuto podporu zkompilovat a odkázat na ni sami.

C#

Když na začátek zdrojového kódu v jazyce C# umístíte příkaz `using`, můžete odkazovat na třídy, které funkce regulárních výrazů poskytují přímo, bez potřeby plné kvalifikace. Namísto funkce `System.Text.RegularExpressions.Regex()` můžete například zadat `Regex()`.

VB.NET

Když na začátek zdrojového kódu v jazyce C# umístíte příkaz `Imports`, můžete odkazovat na třídy, které funkce regulárních výrazů poskytují přímo, bez potřeby plné kvalifikace. Namísto funkce `System.Text.RegularExpressions.Regex()` můžete například zadat `Regex()`.

Java

Abyste mohli používat zabudovanou knihovnu regulárních výrazů v jazyce Java, budete muset do aplikace importovat balík `java.util.regex`.

JavaScript

Podpora regulárních výrazů je v jazyce JavaScript zabudována a je vždy dostupná.

PHP

Funkce `preg` jsou v jazyce PHP ve verzi 4.2.0 a vyšších zabudovány a vždy dostupné.

Python

Abyste mohli využívat funkcí regulárních výrazů v jazyce Python, budete muset importovat modul `re`.

Ruby

Podpora regulárních výrazů je v jazyce Ruby zabudována a vždy dostupná.

3.3 Vytváříme objekty regulárních výrazů

Úloha

Máte za úkol vytvořit z objektu regulárního výrazu instanci, anebo jiným způsobem zkompilovat regulární výraz, abyste ho v aplikaci mohli účinně používat.

Řešení

C#

Když víte, že má regulární výraz správnou podobu:

```
Regex regexObj = new Regex("regex pattern");
```

Když je regulární výraz poskytován koncovým uživatelem (a řetězcovou proměnnou představuje třída `UserInput`):

```
try {
    Regex regexObj = new Regex(UserInput);
} catch (ArgumentException ex) {
    // Regulární výraz má chybnou syntaxi
}
```

VB.NET

Když víte, že má regulární výraz správnou podobu:

```
Dim RegexObj As New Regex("regex pattern")
```

Když je regulární výraz poskytován koncovým uživatelem (a řetězcovou proměnnou představuje třída `UserInput`):

```
Try
    Dim RegexObj As New Regex(UserInput)
Catch ex As ArgumentException
    'Regulární výraz má chybnou syntaxi
End Try
```

Java

Když víte, že má regulární výraz správnou podobu:

```
Pattern regex = Pattern.compile("regex pattern");
```

Když je regulární výraz poskytován koncovým uživatelem (a řetězcovou proměnnou představuje třída `UserInput`):

```
try {
    Pattern regex = Pattern.compile(userInput);
} catch (PatternSyntaxException ex) {
    //Regulární výraz má chybnou syntaxi
}
```

Abyste mohli svůj regulární výraz používat na řetězec, vytvořte třídu `Matcher`:

```
Matcher regexMatcher = regex.matcher(subjectString);
```

Chcete-li regulární výraz používat na jiném řetězci, můžete vytvořit novou třídu `Matcher` tak, jak jsme si právě ukázali, anebo můžete znovu použít již vytvořenou třídu:

```
regexMatcher.reset(anotherSubjectString);
```

JavaScript

Literální regulární výraz v kódu:

```
var myregexp = /regex pattern/;
```

Regulární výraz získaný z uživatelského vstupu v podobě řetězce uloženého v proměnné `userinput`:

```
var myregexp = new RegExp(userinput);
```

Perl

```
$myregex = qr/regex pattern/
```

Regulární výraz získaný z uživatelského vstupu v podobě řetězce uloženého v proměnné `$userinput`:

```
$myregex = qr/$userinput/
```

Python

```
reobj = re.compile("regex pattern")
```

Regulární výraz získaný z uživatelského vstupu v podobě řetězce uloženého v proměnné `userinput`:

```
reobj = re.compile(userinput)
```

Ruby

Literální regulární výraz v kódu:

```
myregexp = /regex pattern/;
```

Regulární výraz získaný z uživatelského vstupu v podobě řetězce uloženého v proměnné `userinput`:

```
myregexp = Regexp.new(userinput);
```

Popis řešení

Než bude moci stroj regulárního výrazu srovnat regulární výraz s řetězcem, musí být regulární výraz zkompilován. Kompilace se provádí za běhu aplikace. Konstruktor regulárního výrazu či kompilační funkce trasuje řetězec, který uchovává regulární výraz, a převádí ho na stromovou strukturu, či na stavový stroj. Funkce, jež provádí samotné srovnání se vzorem, tento strom či stavový stroj při skenování řetězce přeskočí. Programovací jazyky s podporou literálních regulárních výrazů tuto kompilaci provádí tehdy, když se při provádění dosáhne operátoru regulárního výrazu.

.NET

V jazycích C#, VB.NET a .NET uchovává třída `System.Text.RegularExpressions.Regex` jeden zkompilovaný regulární výraz. Nejjednodušší konstruktor přijímá pouze jeden parametr: řetězec, který obsahuje váš regulární výraz.

Když se v regulárním výrazu vyskytne chybná syntaxe, konstruktor `Regex()` vrátí výjimku `ArgumentException`. Text výjimky vám přesně řekne, která chyba byla nalezena. Je-li regulární výraz poskytován uživatelem aplikace, je důležité tuto výjimku zachytit. Nechte si text výjimky zobrazit a poprosťe uživatele, aby regulární výraz opravil. Jestliže je regulární výraz pevně zakódovaným literálním řetězcem, můžete od zachycení výjimky upustit v případě, že použijete nástroj pro kontrolu kódu, díky němuž se ujistíte, že se řádek provedl, aniž by vyvolal výjimku. Stav či režim, které mohou zapříčinit, že se literální regulární výraz v jedné situaci zkompiluje, ovšem ve druhé nikoliv, nelze měnit. Mějte na paměti, že když má regulární výraz chybnou syntaxi, objeví se výjimka při běhu aplikace, ne při její kompilaci.

Jestli chcete regulární výraz používat ve smyčce, nebo v rámci aplikace opakovaně, měli byste vytvořit objekt `Regex`. Sestavení objektu regulárního výrazu si nevyžádá žádné další náklady. Statičtí členové třídy `Regex`, kteří pojímají regulární výraz jako parametr řetězce, vytvoří interně objekt `Regex` tak jako tak, takže to můžete udělat v kódu a zachovat si na objekt odkaz.

Jestli plánujete, že regulární výraz použijete pouze jednou či několikrát, můžete použít statické členy třídy `Regex` a s jejich pomocí uložit řádek kódu. Statičtí členové třídy `Regex` interně vytvořené

objekty regulárního výrazu ihned neodstraňují. Namísto toho uchovají v paměti 15 naposledy použitých regulárních výrazů. Velikost vyrovnávací paměti lze upravit vlastností `Regex.CacheSize`. Vyrovnávací paměť se pak prohledává tak, že se v ní vyhledá řetězec vašeho regulárního výrazu. Hlavně to však s vyrovnávací pamětí nepřeženěte. Jestli potřebujete větší množství objektů regulárních výrazů často, vytvořte si vlastní vyrovnávací paměť, ve které lze vyhledávat účinněji než pomocí řetězců.

Java

V jazyce Java uchovává jeden zkompilovaný regulární výraz třída `Pattern`. Prostřednictvím faktorie třídy `Patttern.compile()`, která vyžaduje pouhý jeden parametr – řetězec s regulárním výrazem – můžete vytvářet objekty této třídy.

Je-li syntaxe regulárního výrazu chybná, faktorie `Pattern.compile()` vypíše výjimku `PatternSyntaxException`. Text výjimky přesně určí, k jaké chybě došlo. Je-li regulární výraz poskytován uživatelem aplikace, je důležité tuto výjimku zachytit. Nechte si text výjimky zobrazit a poproste uživatele, aby regulární výraz opravil. Jestliže je regulární výraz pevně zakódovaným literálním řetězcem, můžete od zachycení výjimky upustit v případě, že použijete nástroj pro kontrolu kódu, díky němuž se ujistíte, že se řádek provedl, aniž by vyvolal výjimku. Stav či režim, které mohou zapříčinit, že se literální regulární výraz v jedné situaci zkompiluje a ve druhé nikoliv, nelze měnit. Mějte na paměti, že když má regulární výraz chybnou syntaxi, objeví se výjimka při běhu aplikace, ne při její kompilaci.

Jestli plánujete, že regulární výraz použijete pouze jednou či několikrát, měli byste namísto statických členů třídy `String` použít raději třídu `Pattern`. I když k tomu bude zapotřebí několika dalších řádků kódu, výsledek bude efektivnější. Statické volání regulární výraz překompiluje pokaždé. Java vlastně statická volání nabízí jen u několika velice základních úloh regulárních výrazů.

Objekt `Pattern` ukládá pouze zkompilovaný regulární výraz a vlastně nevykonává žádnou práci. Vlastní vyhledávání shody regulárního výrazu se provádí třídou `Matcher`. Budete-li ji chtít vytvořit, volejte na zkompilovaném regulárním výrazu metodu `matcher()`. Zadejte jí jediný argument, řetězec zdrojového textu.

Metodu `matcher()` lze volat kdykoliv, když na více řetězcích budete chtít použít stejný regulární výraz. Najednou můžete pracovat s více těmito metodami, ale musíte vše uchovat v jednom podprocesu. Ke třídám `Pattern` a `Matcher` není přístup z více vláken bezpečný. Budete-li chtít stejný regulární výraz použít ve více vláknech (podprocesech), volejte v každém vlákně metodu `Pattern.compile()`.

Jestli jste s aplikací jednoho řetězce hotovi a chcete stejný regulární výraz použít na jiný řetězec, můžete objekt `Matcher` voláním metody `reset()` použít znovu. Další řetězec se zdrojovým textem předejte jako jediný argument. Dosáhnete tak lepších výsledků než vytvořením nového objektu `Matcher`. Metoda `reset()` vrací stejný objekt `Matcher`, na kterém jste ji volali a umožní vám snadno znovu nastavit a vyhledávat shodu v řádku kódu, např. metodou `regexMatcher.reset(nextString).find()`.

Javascript

V notaci literálních regulárních výrazů zobrazených v receptu 3.2 již vytváříme nový objekt regulárního výrazu. Když budete chtít stejný objekt použít opakovaně, jednoduše ho přiřaďte proměnné.

Uchováváte-li regulární výraz v proměnné řetězce (např. protože jste po uživateli chtěli, aby zadal svůj regulární výraz), zkompilujte regulární výraz pomocí konstruktoru `RegExp()`. Všimněte si, že regulární výraz obsažený v řetězci není oddělen lomítky. Tato lomítka představují v jazyce JavaScript část notace literálních objektů `RegExp`, nejsou jen součástí samotného regulárního výrazu.

Protože je přiřazení literálního regulárního výrazu proměnné triviální, ve většině řešení v jazyce JavaScript v této kapitole tento řádek kódu vynecháváme a literální regulární výraz používáme přímo. Když budete stejný regulární výraz používat více než jednou, měli byste v kódu výraz přiřadit proměnné a pracovat pak s ní, ne do kódu překopírovávat stejný literální výraz znovu a znovu. Zlepší se tím výkon a kód se vám bude lépe udržovat.

PHP

Jazyk PHP ukládání zkompilovaného regulárního výrazu v proměnné neumožňuje. Kdykoliv budete s regulárním výrazem něco podnikat, musíte ho některé z funkcí `preg` předat ve formě řetězce.

Funkce `preg` uchovává vyrovnávací paměť s až 4 096 zkompilovanými regulárními výrazy. I když není vyhledávání ve vyrovnávací paměti založené na haši tak rychlé, jako odkazování na proměnnou, nebude výkon postižen tak dramaticky, abyste museli tentýž regulární výraz opětovně překompilovávat. Jakmile se paměť zaplní, odstraní se regulární výraz, který byl kompilován před nejdelší dobou.

Perl

Za účelem kompilace regulárního výrazu a jeho přiřazení proměnné lze použít operátor `qr` (citace regulárního výrazu). Ten pracuje se stejnou syntaxí jako operátor pro vyhledávání shody, který jsme si popsali v receptu 3.1, ovšem namísto písmenem `m` začíná písmeny `qr`.

Perl je obecně při opětovném užívání dříve zkompilovaných regulárních výrazů docela účinný. Proto ve vzorcích kódu v této kapitole operátor `qr//` nepoužíváme. Užití tohoto výrazu předvádíme pouze v receptu 3.5.

Operátor `qr//` se hodí při interpolaci proměnných v regulárním výrazu, nebo při získávání celého regulárního výrazu jako řetězce (např. z uživatelského vstupu). Pomocí operátoru `qr/$regexstring/` lze hlídat, kdy je regulární výraz znovu kompilován, a reflektovat tak nový obsah řetězce `$regexstring`. Operátor `m/$regexstring/` by regulární výraz překompiloval vždy, zatímco operátor `m/$regexstring/o` nikdy. Část `/o` je vysvětlena v receptu 3.4.

Python

Funkce `compile()` pojímá v modulu `re` řetězec s regulárním výrazem a vrací objekt se zkompilovaným regulárním výrazem.

Funkci `compile()` byste měli, pakliže plánujete používat stejný regulární výraz opakovaně, volat explicitně. Všechny funkce v modulu `re` nejprve volají funkci `compile()` a poté funkci, kterou jste si na zkompilovaném objektu regulárního výrazu přáli provádět.

Funkce `compile()` uchovává odkazy na posledních sto zkompilovaných regulárních výrazů. Opětovná kompilace u kteréhokoliv ze 100 naposledy použitých regulárních výrazů se tak omezí na vyhledání ve slovníku. Jakmile se mezipaměť zaplní, dojde k jejímu kompletnímu vyprázdnění.

Jestli se vám nejedná o výkon, mezipaměť funguje dostatečně dobře, abyste mohli funkce v modulu `re` používat přímo. Působil-li by však oslabený výkon potíže, je lepší volat funkci `compile()`.

Ruby

Notace literálního regulárního výrazu, kterou jste viděli v receptu 3.2, již nový objekt regulárního výrazu vytvořila. Když budete stejný objekt používat opakovaně, přiřaďte ho jednoduše proměnné.

Máte-li v proměnné řetězce uložený nějaký literální regulární výraz (např. protože jste po uživateli požadovali, aby regulární výraz zadal), použijte ke kompilaci regulárního výrazu faktorii `Regexp.new()` nebo synonymní funkci `Regexp.compile()`. Pamatujte si, že regulární výraz v řetězci není oddělen lomítky. Ta jsou v notaci jazyka Ruby součástí literálních objektů `Regexp`, nikoliv však samotného regulárního výrazu.

Protože je přiřazení literálního regulárního výrazu proměnné triviální, ve většině řešení v jazyce JavaScript v této kapitole tento řádek kódu vynecháváme a literální regulární výraz používáme přímo. Když budete stejný regulární výraz používat více než jednou, měli byste v kódu výraz přiřadit proměnné a pracovat pak s ní, ne překopírovávat stejný literální výraz do kódu znovu a znovu. Zlepší se tím výkon a kód se vám bude lépe udržovat.

Kompilace regulárního výrazu do jazyka CIL

C#

```
Regex regexObj = new Regex("regex pattern", RegexOptions.Compiled);
```

VB.NET

```
Dim RegexObj As New Regex("regex pattern", RegexOptions.Compiled)
```

Popis řešení

Když v rozhraní .NET vytvoříte objekt `Regex`, aniž byste předali nějaké volby, zkompiluje se regulární výraz tak, jak je to popsáno na straně 109 v oddíle „Popis řešení“. Když jako druhý parametr předáte konstruktoru `Regex()` volbu `RegexOptions.Compiled`, třída `Regex` provede něco docela odlišného: zkompiluje regulární výraz do jazyka CIL, rovněž známého jako MSIL. Zkratka CIL znamená Common Intermediate Language, jedná se o nízkoúrovňový programovací jazyk, který je blíže sestavení než jazykům C# či Visual Basic. Všechny kompilátory v rozhraní .NET mají za výstup kód v jazyce CIL. Při prvním spuštění aplikace zkompiluje rozhraní .NET kód v jazyce CIL dále do strojového kódu, který bude příhodný uživatelově počítači.

Výhodou kompilování regulárního výrazu s volbou `RegexOptions.Compiled` je to, že výraz se může provádět až desetkrát rychleji než výraz zkompilovaný bez této volby. Nevýhodou je to, že taková kompilace může trvat až o dva řády déle než jednoduché trasování řetězce regulárního výrazu do stromu. Kód CIL se navíc až do ukončení stane pevnou součástí aplikace. CIL kód není automaticky uvolňován z paměti.

Volbu `RegexOption.Compiled` použijte jen tehdy, je-li regulární výraz příliš složitý, anebo musí zpracovat tolik textu, že uživatel při práci s ním pocítí značná zpozdění operací. U regulárních výrazů, které pracují zlomek sekundy, nemá práce s kompilací a sestavením význam.

Další informace

Další informace najdete v receptech 3.1, 3.2 a 3.4.

3.4 Nastavení voleb regulárních výrazů

Úloha

Zkompilujete regulární výraz tak, aby měl dostupné všechny režimy vyhledávání: přehlížení bílých znaků, nerozlišování velkých a malých znaků, tečka odpovídající zalomení řádku a znaky stříšky a dolaru značí pozici zalomení řádku.

Řešení

C#

```
Regex regexObj = new Regex("regex pattern",
    RegexOptions.IgnorePatternWhitespace | RegexOptions.IgnoreCase |
    RegexOptions.Singleline | RegexOptions.Multiline);
```

VB.NET

```
Dim RegexObj As New Regex("regex pattern",
    RegexOptions.IgnorePatternWhitespace Or RegexOptions.IgnoreCase Or
    RegexOptions.Singleline Or RegexOptions.Multiline)
```

Java

```
Pattern regex = Pattern.compile("regex pattern",
    Pattern.COMMENTS | Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE |
    Pattern.DOTALL | Pattern.MULTILINE);
```

JavaScript

Literální regulární výraz ve vašem kódu:

```
var myregexp = /regex pattern/im;
```

Regulární výraz získaný jako řetězec z uživatelského vstupu:

```
var myregexp = new RegExp(userinput, "im");
```

PHP

```
regexstring = '/regex pattern/simx';
```

Perl

```
m/regex pattern/simx;
```

Python

```
reobj = re.compile("regex pattern",
    re.VERBOSE | re.IGNORECASE |
    re.DOTALL | re.MULTILINE)
```

Ruby

Literální regulární výraz ve vašem kódu:

```
myregexp = /regex pattern/mix;
```

Regulární výraz získaný jako řetězec z uživatelského vstupu:

```
myregexp = Regexp.new(userinput,
    Regexp::EXTENDED or Regexp::IGNORECASE or
    Regexp::MULTILINE);
```

Popis řešení

Mnohé z regulárních výrazů v této knize, a těch, které najdete i jinde, jsou napsány pro užití v určitém režimu vyhledávání shody. Existují čtyři základní režimy, jež podporují téměř všechny moderní druhy regulárních výrazů. Některé druhy však při implementaci těchto režimů nejsou důsledné a jejich názvy si pletou. Když použijete špatný režim, obyčejně tím regulární výraz poškodíte.

Všechna řešení v tomto receptu užívají příznaky či volby poskytované programovacím jazykem či třídou regulárního výrazu určenou k nastavování režimu. Nastavit režim lze i tak, že v regulárním výrazu použijete modifikátor režimu. Modifikátory režimu v regulárním výrazu vždy přepíší volby a příznaky nastavené mimo regulární výraz.

.NET

Konstruktor `Regex()` přijímá volitelný druhý parametr, ve kterém se regulárnímu výrazu zadávají volby. Dostupné volby naleznete ve výčtu třídy `RegexOptions`.

Přehlížení bílých znaků: `RegexOptions.IgnorePatternWhitespace`

Žádné rozlišování mezi malými a velkými znaky: `RegexOptions.IgnoreCase`

Tečka značící zalomení řádku: `RegexOptions.Singleline`

Znak stříšky a dolaru označující zalomení řádku: `RegexOptions.Multiline`

Java

Faktorie třídy `Pattern.compile()` přijímá volitelný druhý parametr s volbami pro regulární výraz. Třída `Pattern` definuje několik konstant, kterými se nastavují různé volby. Zkombinováním s operátorem `|` neboli s bitovým operátorem OR lze nastavit více voleb.

Přehlížení bílých znaků: `Pattern.COMMENTS`

Žádné rozlišování mezi malými a velkými znaky:
`Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE`

Tečka značící zalomení řádku: `Pattern.DOTALL`

Znak stříšky a dolaru označující zalomení řádku: `Pattern.MULTILINE`

Skutečně existují dvě volby, kterými se nastavuje, aby nebyly rozlišovány velké a malé znaky, a aby k tomuto rozlišování nedocházelo, budete je muset nastavit obě dvě. Když nastavíte pouze volbu `Pattern.CASE_INSENSITIVE`, budou se bez rozlišování velikosti hledat pouze anglické znaky A až Z. Nastavíte-li obě volby, budou se bez rozlišování velikosti vyhledávat všechny znaky ve všech písmech. Jediný důvod pro to, abyste nepoužili volbu `Pattern.UNICODE_CASE` je výkon v situaci, kdy víte, že budete pracovat pouze s textem v kódování ASCII. Při práci s modifikátory režimů v regulárním výrazu zadejte `<(?i)>`, čímž docílíte nerozlišování velikosti znaků pouze v tabulce ASCII, či `<(?iu)>`, což zajistí, že se velikost nebude rozlišovat u žádných znaků.

JavaScript

V jazyce JavaScript můžete specifikovat volby tak, že literálu `RegExp` za lomítko, které regulární výraz ukončuje, připojíte jednopísmenné příznaky. V dokumentaci bývají tyto příznaky obyčej-

ně zapsány jako `/i` a `/m`, ačkoliv příznak tvoří pouze jedno písmeno. Při zadávání příznaků určujících režim regulárního výrazu se nezadávají žádná další lomítka.

Při kompilování řetězce do regulárního výrazu prostřednictvím konstruktoru `RegExp()` můžete díky příznakům konstruktoru předat druhý volitelný parametr. Tím by měl být řetězec s písmeny voleb, které chcete nastavit. Nezadávejte do řetězce žádná lomítka.

Přehlížení bílých znaků: V jazyce JavaScript nepodporováno

Žádné rozlišování mezi malými a velkými znaky: `/i`

Tečka značící zalomení řádku: V jazyce JavaScript nepodporováno

Znak stříšky a dolaru označující zalomení řádku: `/m`

PHP

V receptu 3.1 jsme si řekli, že funkce `preg` v jazyce PHP vyžaduje, aby byly literální regulární výrazy odděleny dvěma interpunkčními znaky, obyčejně lomítky, a aby byl celek formátován jako literální řetězec. Volby regulárního výrazu lze zadat tak, že na konec řetězce připojíte alespoň jeden jednopísmenný modifikátor. Takže modifikační písmena se dostanou na řadu za zavíracím oddělovačem regulárního výrazu, ovšem stále se budou nacházet mezi jednoduchými či dvojitými uvozovkami, které řetězec ohraničují. V dokumentaci jsou tyto modifikátory obyčejně značeny jako `/x`, ačkoliv je příznak reprezentován pouze jedním písmenem, a přestože oddělovač mezi regulárním výrazem a modifikátory nemusí být lomítko.

Přehlížení bílých znaků: `/x`

Žádné rozlišování mezi malými a velkými znaky: `/i`

Tečka značící zalomení řádku: `/s`

Znak stříšky a dolaru označující zalomení řádku: `/m`

Perl

Volby regulárního výrazu je možné zadat tak, že na konec operátoru porovnání či substitučního operátoru umístíte jednopísmenný modifikátor. V dokumentaci jsou tyto modifikátory obyčejně značeny jako `/x`, ačkoliv je příznak reprezentován pouze jedním písmenem, a přestože oddělovač mezi regulárním výrazem a modifikátory nemusí být lomítko.

Přehlížení bílých znaků: `/x`

Žádné rozlišování mezi malými a velkými znaky: `/i`

Tečka značící zalomení řádku: `/s`

Znak stříšky a dolaru označující zalomení řádku: `/m`

Python

Funkce `compile()` (vysvětlená v předchozím receptu) přijímá ve volbách regulárního výrazu volitelný druhý parametr. Tento parametr lze sestavit pomocí operátoru `|`, který zkombinuje konstanty definované v modulu `re`. Mnohé z ostatních funkcí modulu `re`, jež přijímají literální regulární výraz jako parametr, rovněž přijmou volby regulárního výrazu jako poslední a volitelný parametr.

Konstanty voleb regulárních výrazů se vyskytují v párech. Každou volbu je možné zadávat buď jako konstantu celým názvem, nebo jako jediné písmeno. Jejich funkce bude ekvivalentní. Jediným rozdílem je to, že kompletní název umožní vývojářům, kteří písmenkovou polévku voleb regulárních výrazů neznají, kód číst snáze. Základní jednopísmenné volby uvedené v tomto oddíle se shodují s volbami v jazyce Perl.

Přehlížení bílých znaků: `re.VERBOSE` nebo `re.X`

Žádné rozlišování mezi malými a velkými znaky: `re.IGNORECASE` nebo `re.I`

Tečka značící zalomení řádku: `re.DOTALL` nebo `re.S`

Znak stříšky a dolaru označující zalomení řádku: `re.MULTILINE` nebo `re.M`

Ruby

V jazyce Ruby lze volby specifikovat tak, že k literálu `Regexp` připojíte za lomítko, které regulární výraz ukončuje, alespoň jeden jednopísmenný příznak. V dokumentaci jsou tyto příznaky obyčejně uváděny jako `/i` a `/m`, i když je příznak tvořen pouze jedním písmenem. Chcete-li regulárnímu výrazu specifikovat více příznaků, žádná další lomítka zadávat nebudete.

Až budete pomocí faktorie `Regexp.new()` kompilovat řetězec do regulárního výrazu, můžete konstruktoru předávat volitelný druhý parametr prostřednictvím příznaků. Druhý parametr by měl buď mít hodnotu `nil`, aby deaktivoval všechny volby, anebo by měl být kombinací konstant ze třídy `Regexp` a operátoru `or`.

Přehlížení bílých znaků: `/r` nebo `Regexp::EXTENDED`

Žádné rozlišování mezi malými a velkými znaky: `/i` nebo `Regexp::IGNORECASE`

Tečka značící zalomení řádku: `/m` nebo `Regexp::MULTILINE`. Ruby v tomto režimu skutečně používá „m“ a „multi line“, zatímco všechny ostatní druhy pracují s písmenem „s“ či „single line“.

Znak stříšky a dolaru označující zalomení řádku: Znak stříšky a dolaru v Ruby vždy označují zalomení řádku. Toto nastavení nelze deaktivovat. Začátek a konec zdrojového řetězce označíte tokeny `<\A>` a `<\Z>`.

Další volby konkrétních jazyků

.NET

Volba `RegexOptions.ExplicitCapture` učiní ze všech skupin, včetně pojmenovaných skupin, skupiny bez zachycení. S takovým nastavením budou výrazy `<(group)>` a `<(?:group)>` shodné. Pakliže skupiny zachycení pojmenováváte vždy, aktivujte tuto volbu. Váš regulární výraz bude výkonnější a přitom ani nebude vyžadovat syntaxi `<(?:group)>`. Tuto volbu lze namísto zadání `RegexOptions.ExplicitCapture` aktivovat i tak, že na začátek regulárního výrazu vložíte `<(?n)>`. O seskupování se dozvíte více v receptu 2.9. Pojmenované skupiny jsme si vysvětlili v receptu 2.11.

Jestli v kódu .NET a JavaScript používáte stejný regulární výraz a chcete zajistit, aby se choval v obou prostředích stejně, využijte volbu `RegexOptions.ECMAScript`. Zvláště se vám bude hodit při vývoji klientské části webové aplikace v jazyce JavaScript a serverové části v prostředí ASP.NET. Nejdůležitějším důsledkem nasazení této volby je to, že `\w` a `\d` se budou týkat pouze znaků z tabulky ASCII, stejně jako v jazyce JavaScript.

Java

Java nabízí jedinečnou funkci, `Pattern.CANON_EQ`, která umožňuje „kanonickou ekvivalenci“. Jak jsme si vysvětlili na straně 52 v oddíle “Unicode grafém“, standard Unicode nabízí jiný způsob reprezentace znaků prostřednictvím diakritiky. Když tuto volbu aktivujete, bude regulární výraz odpovídat znaku, i když je ve zdrojovém textu kódován jinak. Například regulární výraz `<\u00E0>` bude odpovídat výrazům „`\u00E0`“ i „`\u0061\u0300`“, protože ty jsou kanonicky ekvivalentní. Oba se na obrazovce objeví jako „à“ a uživatel mezi nimi nebude schopen rozlišit.

Kdyby nebyly výrazy kanonicky ekvivalentní, výraz „\u0061\u0300“ by řetězci neodpovídal. Takto se chovají všechny ostatní druhy regulárních výrazů, které v knize popisujeme.

Konečně funkce `Pattern.UNIX_LINES` Javě říká, aby při zalamování řádku u znaků tečky, stříšky a dolaru pracovala pouze s tokenem `<\n>`. Implicitně se se všemi zalomeními řádku ve standardu Unicode zachází jako se znaky značícími zalomení řádku.

JavaScript

Chcete-li na stejný řetězec použít regulární výraz opakovaně – např. při iteraci na všech shodách, anebo při hledání a nahrazování všech shod, nejen první shody – zadejte příznak `/g`, neboli „globální“ příznak.

PHP

Modifikátor `/u` říká knihovně PCRE, že si má regulární výraz i zdrojový řetězec vykládat jako řetězce v kódování UTF-8. Současně umožňuje i tokeny regulárních výrazů Unicode, jakými jsou například `<\p{FFFF}>` a `<\p{L}>`. Ty jsou vysvětleny v receptu 2.7. Bez těchto modifikátorů zachází PCRE s každým bajtem jako se samostatným znakem a tokeny regulárních výrazů Unicode vyvolají chybu.

Modifikátor `/U` přepne „chamtivé“ a „líné“ chování, při kterém se do kvantifikátoru vkládá otazník navíc. Kvantifikátor `<.*>` je normálně chamtivý a kvantifikátor `<.*?>` je líný. Když použijete modifikátor `/U`, kvantifikátor `<.*>` bude líný a kvantifikátor `<.*?>` chamtivý. Před tímto modifikátorem vás důrazně varuji, protože jenom splete programátory, kteří budou později číst váš kód a modifikátor `/U` přehlédnou, protože se vyskytuje pouze v jazyce PHP. Stejně tak si dejte pozor, abyste si `/U` nepletli s `/u`, když na ně narazíte v kódu někoho jiného. Modifikátory regulárních výrazů jsou citlivé na velikost znaků.

Perl

Chcete-li na stejný řetězec použít regulární výraz opakovaně (např. při iteraci na všech shodách, anebo při hledání a nahrazování všech shod, nejen první shody), zadejte příznak `/g`, neboli „globální“ příznak.

Při interpolaci proměnné v regulárním výrazu – například `m/I am $name/` – překompiluje Perl regulární výraz pokaždé, když je třeba ho použít, protože obsah proměnné `$name` se mohl změnit. Toto chování lze potlačit modifikátorem `/o`. Výraz `m/I am $name/` se zkompiluje tehdy, když ho Perl potřebuje použít prvně, později se už používá tak, jak je. Změní-li se obsah proměnné `$name`, regulární výraz nebude změnu reflektovat. Chcete-li mít kontrolu nad tím, kdy je regulární výraz kompilován, nahlédněte do receptu 3.3.

Python

Python nabízí dvě další volby, které mění význam ohraničení slov (viz recept 2.6), třídy zkratkových znaků `<\w>`, `<\d>` a `<\s>`, a současně i jejich negované protějšky (viz recept 2.3). Implicitně se tyto tokeny zabývají pouze písmeny ze sady ASCII, číslicemi a bílými znaky.

Volba `re.LOCALE`, či `re.L`, činí tyto tokeny závislé na aktuálním národním prostředí. Toto prostředí pak určí, se kterými znaky mají tyto tokeny zacházet jako s písmeny, číslicemi a bílými znaky. Tuto volbu byste měli aktivovat tehdy, když zdrojový řetězec není řetězcem standardu Unicode a vy chcete, aby se s písmeny s diakritickými znaménky zacházelo žádoucím způsobem.

Volba `re.UNICODE` či `re.U` učiní tyto tokeny závislými na standardu Unicode. Všechny znaky definované ve standardu jako písmena, číslice a bílé znaky pak budou příslušným způsobem chápány i těmito tokeny regulárního výrazu. Tuto volbu byste měli aktivovat tehdy, když zdro-

jový řetězec není řetězcem standardu Unicode a vy chcete, aby se s písmeny s diakritikou zacházelo vhodným způsobem.

Ruby

Faktorie `Regexp.new()` přijímá volitelný třetí parametr, pomocí kterého vybírá kódování řetězce, které podporuje váš regulární výraz. Když u svého regulárního výrazu kódování nespecifikujete, použije se stejné kódování jako ve zdrojovém souboru. Většinou je vhodné používat kódování zdrojového souboru.

Když budete chtít kódování zadat explicitně, předejte parametru jediný znak. Parametr je citlivý na velikost znaků. Možné hodnoty jsou:

- `n`: Tento znak zastupuje „žádné“ (none) kódování. S každým bajtem se v řetězci bude zacházet jako s jedním znakem. Tuto volbu užívejte v ASCII textech.
- `e`: Povoluje kódování „EUC“ pro jazyky dálného východu.
- `s`: Aktivuje japonské kódování „Shift-JIS“.
- `u`: Umožňuje kódování UTF-8, které používá jeden až čtyři bajty na jeden znak, a které podporuje všechny jazyky ve standardu Unicode (kam spadají všechny živé, alespoň trochu významné jazyky).

U literálních regulárních výrazů můžete nastavit kódování pomocí modifikátorů `/n`, `/e`, `/s` a `/u`. Na jednom regulárním výrazu je možné použít pouze jeden z těchto modifikátorů. Lze je používat v kombinaci s kterýmkoliv z modifikátorů `/x`, `/i` a `/m`.

Nezapleťte si modifikátor `/s` v jazyce Ruby s tímtéž modifikátorem v jazycích Perl, Java či .NET. V jazyce Ruby vynucuje modifikátor `/s` kódování Shift-JIS. V jazyce Perl a ve většině ostatních druhů regulárních výrazů aktivuje režim shody tečky se zalomením řádku. V jazyce Ruby lze téhož docílit modifikátorem `/m`.

Další informace

Účinky režimů shody jsou podrobně vysvětleny ve druhé kapitole. V oněch oddílech vysvětlujeme také použití modifikátorů režimů v rámci regulárních výrazů.

Přehlížení bílých znaků: Recept 2.18

Žádné rozlišování mezi malými a velkými znaky: Oddíl “Vyhledávání bez ohledu na velikost znaků“ na straně 37 v receptu 2.1

Tečka značící zalomení řádku: Recept 2.4

Znak stříšky a dolaru označující zalomení řádku: Recept 2.5

Recepty 3.1 a 3.3 vysvětlují použití literálních regulárních výrazů ve zdrojovém kódu a vytváření objektů regulárních výrazů. Volby regulárních výrazů se zadávají v průběhu vytváření regulárního výrazu.

3.5 Test na výskyt shody ve zdrojovém řetězci

Úloha

Budete zjišťovat, zda lze v konkrétním řetězci vyhledat shodu konkrétního regulárního výrazu. Postačí i částečná shoda. Například regulární výraz `<regex•pattern>` bude částečně odpovídat