

KAPITOLA 3

Práce s regulárními výrazy

Zvolil bych vést ho v bludišti po vyšlapaných cestách...

– Any Lowell, *Patterns*

Síla regulárních výrazů jako výpočetního nástroje byla často podceňována. Od jejich prvotních teoretických začátků ve čtyřicátých letech si v letech šedesátých našly cestu k počítačovým systémům a odtud k různým nástrojům v operačním systému Unix. V devadesátých letech došlo ke zdomácnění regulárních výrazů (hlavně díky popularitě Perlu), takže už to nebyly těžko srozumitelné pomůcky, se kterými pracovali pouze ti největší zasvěcenci.

Krása regulárních výrazů spočívá v tom, že téměř veškerou naši zkušenost můžeme chápat jako vzory. Jakmile máme vzory, které můžeme popsat, dokážeme je nacházet; dokážeme nacházet střípky reality, které odpovídají těmto vzorům, a dokážeme je ovlivnit podle své volby.

V době psaní tohoto textu dochází ve vývoji Ruby k mnoha změnám. Stávající engine regulárních výrazů bude nahrazen novým, který se nazývá Oniguruma. Tomuto enginu je věnována pozdější sekce "3.13 – Ruby a Oniguruma" této kapitoly. V kapitole 4 jsou pak uvedena specifika regulárních výrazů v souvislosti s mezinárodním použitím.

3.1 – Syntaxe regulárních výrazů

Typický regulární výraz je ohraničen dvojicí lomítek. Regulární výrazy rovněž mohou být zapsány ve formě %r . Tabulka 3.1 ukazuje některé jednoduché příklady:

Tabulka 3.1. Základní regulární výrazy.

| Regex | Vysvětlení |
|--------|-----------------------------------|
| /Ruby/ | Odpovídá samostatnému slovu Ruby. |

| Regex | Vysvětlení |
|--------------|---|
| / [Rr] uby / | Odpovídá Ruby nebo ruby. |
| / ^abc / | Odpovídá abc na začátku řádku. |
| %r (xyz\$) | Odpovídá xyz na konci řádku. |
| %r [0-9] * | Odpovídá sekvenci (nula nebo více) čísel. |

Dále je možné použít modifikátory, které se skládají z jednoho písmene, jež je umístěno ihned za samotným regulárním výrazem. Tabulka 3.2 ukazuje nejběžnější modifikátory:

Tabulka 3.2. Modifikátory v regulárních výrazech.

| Modifikátor | Význam |
|-------------|---|
| i | Ignorovat velikost písmen v regulárním výrazu. |
| o | Vykonat nahrazení výrazu pouze jednou. |
| m | Víceřádkový režim (tečka odpovídá novému řádku). |
| x | Rozšířený regulární výraz (povoluje prázdné znaky a komentáře). |

Další modifikátory jsou popsány v kapitole 4. Na konec tohoto úvodu o regulárních výrazech si ještě v tabulce 3.3 vyjmenujme nejběžnější symboly a notace.

Tabulka 3.3. Běžné notace používané v regulárních výrazech.

| Zápis | Význam |
|-------|--|
| ^ | Začátek řádku nebo řetězce. |
| \$ | Konec řádku nebo řetězce. |
| . | Libovolný znak kromě nového řádku (s výjimkou víceřádkového režimu). |
| \w | Alfanumerické znaky. |
| \W | Jiné než alfanumerické znaky. |
| \s | Prázdný znak (mezera, tabulátor, nový řádek atd.). |
| \S | Neprázdné znaky. |
| \d | Číslice (totéž jako [0-9]). |
| \D | Jiné znaky než číslice. |
| \A | Začátek řetězce. |
| \Z | Konec řetězce nebo před začátkem nového řádku. |

| Zápis | Význam |
|--------------------------------|--|
| <code>\z</code> | Konec řetězce. |
| <code>\b</code> | Hranice slova (pouze vně <code>[]</code>). |
| <code>\B</code> | Jiné znaky než hranice slova. |
| <code>\b</code> | Znak <code>backspace</code> (pouze uvnitř <code>[]</code>). |
| <code>[]</code> | Kterýkoliv znak množiny. |
| <code>*</code> | Žádný nebo libovolný počet výskytů předchozího znaku. |
| <code>*?</code> | Žádný nebo libovolný počet výskytů předchozího znaku (<i>non-greedy</i>). |
| <code>+</code> | Jeden nebo libovolný počet výskytů předchozího znaku. |
| <code>+</code> | Jeden nebo libovolný počet výskytů předchozího znaku (<i>non-greedy</i>). |
| <code>{m,n}</code> | od <code>m</code> do <code>n</code> výskytů předcházejícího podvýrazu. |
| <code>{m,n}?</code> | od <code>m</code> do <code>n</code> výskytů předcházejícího podvýrazu (<i>non-greedy</i>). |
| <code>?</code> | Žádný nebo jeden výskyt předchozího znaku. |
| <code> </code> | Alternativy (<code>X Y</code> znamená <code>X</code> nebo <code>Y</code>). |
| <code>(?=)</code> | Pozitivní vyhlížení. |
| <code>(?!)</code> | Negativní vyhlížení. |
| <code>()</code> | Skupina výrazů. |
| <code>(?>)</code> | Vnořený výraz. |
| <code>(?:)</code> | Skupina nezačleněná do výsledku. |
| <code>(?imx - imx)</code> | Nastavení volby na <i>on/off</i> , platné od tohoto okamžiku. |
| <code>(?imx - imx:expr)</code> | Nastavení volby na <i>on/off</i> , platné pro tento výraz. |
| <code>(?#)</code> | Komentář. |

Znalost regulárních výrazů přináší modernímu programátorovi velké množství výhod. Protože vyčerpávající popis tohoto tématu je mimo rámec této knihy, doporučujeme vám se podívat do knihy *Mastering Regular Expressions*, kterou napsal Jeffrey Friedl.

Pro další informace o tématu tohoto oddílu se podívejte na sekce "3.13 – Ruby a Oniguruma".

3.2 – Kompilování regulárních výrazů

Regulární výrazy mohou být zkompileovány pomocí metody `Regexp.compile` (která je ve skutečnosti synonymem pro `Regexp.new`). První parametr je povinný a může to být řetězec nebo `regex`.

(Povšimněte si, že pokud je parametrem regulární výraz s nějakým modifikátorem, nebude tento modifikátor platný pro právě zkompilovaný regulární výraz.)

```
pat1 = Regexp.compile("^foo.*")           # /^foo.*/
pat2 = Regexp.compile(/bar$/i)           # /bar/ (i není propagováno)
```

Druhý parametr, pokud je uveden, je obvykle jednou z následujících konstant nebo bitovým OR více z nich – `Regexp::EXTENDED`, `Regexp::IGNORECASE` a `Regexp::MULTILINE`. Navíc libovolná hodnota parametru, která není `nil`, bude mít za následek vytvoření regulárního výrazu, jenž nebude citlivý na velikost písmen (case-insensitive). Toto vám však nedoporučujeme praktikovat.

```
options = Regexp::MULTILINE || Regexp::IGNORECASE
pat3 = Regexp.compile("^foo", options)
pat4 = Regexp.compile(/bar/, Regexp::IGNORECASE)
```

Třetí parametr, pokud je specifikován, je jazykový parametr, který umožňuje podporu vícebajtových znaků. Akceptuje jakoukoliv ze čtyř následujících řetězcových hodnot:

```
"N" or "n" means None
"E" or "e" means EUC
"S" or "s" means Shift-JIS
"U" or "u" means UTF-8
```

Literály regulárních výrazů mohou být samozřejmě specifikovány bez volání `new` nebo `compile`, stačí je uzavřít mezi lomítka.

```
pat1 = /^foo.*/
pat2 = /bar$/i
```

Pro více informací nahlédněte do kapitoly 4.

3.3 – Ošetření speciálních znaků

Metoda třídy `Regexp.escape` zajišťuje ošetření všech speciálních znaků, které jsou použity v regulárních výrazech. Jsou to znaky jako hvězdička, otazník a hranaté závorky.

```
str1 = "[*?]"
str2 = Regexp.escape(str1)           # "\[*\*\?\]"
```

Metoda `Regexp.quote` je pouze alias.

3.4 – Používání kotev (anchors)

Kotva je speciální výraz, který odpovídá pozici v řetězci (nikoliv znaku nebo sekvenci znaků). Jak uvidíme později, jedná se o jednoduchý případ předpokladu s nulovou délkou (zero-width assertion). Jinak řečeno – je to vzor, který v případě shody nespotřebává žádné znaky z řetězce.

Nejběžnější kotvy byly již uvedeny na začátku této kapitoly. Nejjednodušší jsou `^` a `$`, které odpovídají začátku a konci řetězce.

```
string = "abcXdefXghi"
/def/ =~ string      # 4
/abc/ =~ string      # 0
/ghi/ =~ string      # 8
/^def/ =~ string     # nil
/def$/ =~ string     # nil
/^abc/ =~ string     # 0
/ghi$/ =~ string     # 8
```

Nicméně – právě jsem vám řekl malou lež. Tyto kotvy vlastně neodpovídají začátku a konci řetězce, ale řádku. Pouvažujte nad stejnými vzory, které jsou aplikovány na podobný řetězec, jenž ovšem obsahuje nové řádky:

```
string = "abc\ndef\nghi"
/def/ =~ string      # 4
/abc/ =~ string      # 0
/ghi/ =~ string      # 8
/^def/ =~ string     # 4
/def$/ =~ string     # 4
/^abc/ =~ string     # 0
/ghi$/ =~ string     # 8
```

Máme ovšem k dispozici i speciální kotvy `\A` a `\Z`, které skutečně odpovídají začátku a konci řetězce.

```
string = "abc\ndef\nghi"
/\Adef/ =~ string    # nil
/def\Z/ =~ string    # nil
/\Aabc/ =~ string    # 0
/ghi\Z/ =~ string    # 8
```

`\z` je totéž jako `\Z`, ovšem s tím rozdílem, že `\Z` ignoruje případný ukončující znak nového řádku, kdežto v případě `\z` musí být shoda explicitní.

```
string = "abc\ndef\nghi"
str2 << "\n"
```

```
/ghi\Z/ =~ string      # 8
/\Aabc/ =~ str2        # 8
/ghi\z/ =~ string      # 8
/ghi\z/ =~ str2        # nil
```

Dále je možné pomocí `\b` porovnávat hranici slova, nebo pomocí `\B` místo, které není hranicí slova. Příklady s `gsub` vám pomohou pochopit, jak to pracuje:

```
str = "this is a test"
str.gsub(/\b/, "|")      # "|this| |is| |a| |test|"
str.gsub(/\B/, "-")      # "t-h-i-s i-s a t-e-s-t"
```

Neexistuje žádný způsob, jak rozlišit počáteční a koncovou hranici slova.

3.5 – Používání kvantifikátorů

Velkou částí regulárních výrazů je práce s nepovinnými položkami a opakováním. Položka následující za otazníkem je nepovinná – může být uvedena, nebo může chybět; porovnání je závislé na zbytku regulárního výrazu. (Nemá smysl tohle aplikovat na kotvy, ale pouze na část vzoru (subpattern) s nenulovou délkou.)

```
pattern = /ax?b/
pat2 = /a[xy]?b/
pattern =~ "ab"          # 0
pattern =~ "acb"         # nil
pattern =~ "axb"         # 0
pat2 =~ "ayb"            # 0
pat2 =~ "acb"            # nil
```

Pro entity je obvyklé, aby se nekonečně opakovaly (což můžeme specifikovat kvantifikátorem `+`). Například tento vzor odpovídá jakémukoliv celému kladnému číslu:

```
pattern = /[0-9]+/
pattern =~ "1"           # 0
pattern =~ "2345678"     # 0
```

Další běžný případ je vzor, který nenastane ani jednou, nebo nastane vícekrát. To samozřejmě můžete udělat pomocí `+` a `?`. V následujícím fragmentu kódu porovnáváme řetězec `Huzzah` následovaný žádným, nebo více vykřičníky:

```
pattern = /Huzzah(!+)?/  # Závorky jsou zde nutné
pattern =~ "Huzzah"      # 0
pattern =~ "Huzzah!!!!"  # 0
```

Nicméně existuje i lepší způsob. Toto chování popisuje kvantifikátor `*`.

```
pattern = /Huzzah!*/          # * se aplikuje pouze na !
pattern =~ "Huzzah"           # 0
pattern =~ "Huzzah!!!"       # 0
```

Co když chceme porovnávat číslo sociálního pojištění? K tomu poslouží tento vzor:

```
ssn = "987-65-4320"
pattern = /\d\d\d-\d\d-\d\d\d\d/
pattern =~ ssn                # 0
```

Ale tento způsob není příliš čistý. Pojdme jednoznačně říci, kolik číslic je v každé skupině. Číslo v závorkách je kvantifikátor:

```
pattern = /\d{3}-\d{2}-\d{4}/
```

Tohle sice není ten nejkratší možný vzor, který lze napsat, nicméně je jednoznačný a docela čitelný. Jako oddělovač může být použita i čárka. Představte si telefonní číslo obyvatele Elbonie, které se skládá z části se třemi až pěti čísly a z části se třemi až sedmi čísly. Tady je odpovídající vzor:

```
elbonian_phone = /\d{3,5}-\d{3,7}/
```

První a poslední číslice jsou nepovinné (ačkoliv musíme mít jednu, nebo druhou):

```
/x{5}/          # pro
/x{5,7}/        # pro 5-7
/x{,8}/         # pro až 8
/x{3,}/         # pro alespoň 3
```

Tímto způsobem mohou být samozřejmě přepsány kvantifikátory `?`, `+` a `*`:

```
/x?/           # totéž jako /x{0,1}/
/x*/           # totéž jako /x{0,}/
/x+/           # totéž jako /x{1,}/
```

Terminologie regulárních výrazů je plná barvitých personifikujících termínů jako *greedy* (chamtivý), *reluctant* (zdráhavý), *lazy* (líný) a *possessive* (majetnický). Rozdíl mezi *greedy*/*non-greedy* je jeden z nejdůležitějších. Zamysleme se nad touto částí kódu. Můžete očekávat, že tento *regex* bude odpovídat "Where the", ovšem místo toho odpovídá nejdelší části řetězce "Where the sea meets the":

```
str = "Where the sea meets the moon-blanch'd land,"
match = /\.the/.match(str)
p match[0]          # Zobrazí celou shodu:
                    # "Where the sea meets the"
```

Důvodem je, že operátor `*` je *greedy* (chamtivý) – při porovnání zkonzumuje tolik řetězce, kolik je možné pro nejdelší možnou shodu. Pomocí otazníku z něj můžeme udělat *non-greedy*:

```
str = "Where the sea meets the moon-blanch'd land,"
match = /. *?the/.match(str)
p match[0]          # Zobrazí celou shodu:
                    # "Where the"
```

Tyto ukázky nám předvádí, že operátor `*` je standartně chamtivý, greedy (bez připojeného `?`). Totéž platí pro kvantifikátory `+` a `{m, n}`, a také pro kvantifikátor `?`. Nebyl jsem ovšem schopen vymyslet dobré příklady pro situaci s `{m, n}?` a `??`. Pokud nějaké znáte, podělte se.

Pro více informací o kvantifikátorech nahlédněte do sekce "3.13 – Ruby a Oniguruma".

3.6 – Pozitivní a negativní vyhlížení

Regulární výraz je porovnáván vůči řetězci přímočarým způsobem (se zpětnou kontrolou, backtracking, v případě potřeby). Z tohoto důvodu existuje v řetězci koncept "aktuálního umístění" – v podstatě se jedná o souborový ukazatel nebo kurzor.

Termín vyhlížení (lookahead) se odkazuje na konstrukci, která odpovídá části řetězce za aktuální pozici. Jedná se o vzor s nulovou délkou, protože dokonce i tehdy, když je srovnání úspěšné, nedojde ke zkonzumování žádné části řetězce (to znamená, že aktuální pozice se nemění).

V následujícím příkladě bude řetězec "New World" odpovídat pouze v případě, pokud bude následován slovem "Symphony" nebo "Dictionary" (třetí slovo není součástí porovnání):

```
s1 = "New World Dictionary"
s2 = "New World Symphony"
s3 = "New World Order"
reg = /New World(?: Dictionary| Symphony)/
m1 = reg.match(s1)
m.to_a[0]          # "New World"
m2 = reg.match(s2)
m.to_a[0]          # "New World"
m3 = reg.match(s3)  # nil
```

A zde je ukázka negativního vyhlížení:

```
reg2 = /New World(?:! Symphony)/
m1 = reg2.match(s1)
m.to_a[0]          # "New World"
m2 = reg2.match(s2)
m.to_a[0]          # nil
m3 = reg2.match(s3)  # "New World"
```

V tomto příkladu bude řetězec "New World" odpovídat pouze tehdy, pokud nebude následován slovem "Symphony".

3.7 – Přístup ke zpětným referencím

Každá část regulárního výrazu, která je umístěna do závorek, je samostatně přístupnou částí výsledku porovnání. Tyto části jsou očíslovány, takže prostřednictvím těchto čísel se na ně můžete odkazovat. Existuje několik způsobů, jak to provést. Pojďme nejprve prozkoumat tradičnější (tzn. ošklivější) způsoby. Speciální globální proměnné jako \$1, \$2 atd. mohou být použity jako reference na shody.

```
str = "a123b45c678"
if /(a\d+)(b\d+)(c\d+)/ =~ str
  puts "Matches are: '#$1', '#$2', '#$3'"
  # Vytiskne: Matches are: 'a123', 'b45', 'c768'
end
```

Uvnitř substitute (jako například sub nebo gsub) nemohou být tyto proměnné použity.

```
str = "a123b45c678"
str.sub(/(a\d+)(b\d+)(c\d+)/, "1st=#$1, 2nd=#$2, 3rd=#$3")
# "1st=, 2nd=, 3rd="
```

Možná se ptáte, proč nemůže fungovat? Odpověď je jednoduchá – argumenty pro sub jsou vyhodnoceny ještě předtím, než dojde k zavolání sub. Následující kód je ekvivalentní:

```
str = "a123b45c678"
s2 = "1st=#$1, 2nd=#$2, 3rd=#$3"
reg = /(a\d+)(b\d+)(c\d+)/
str.sub(reg, s2)
# "1st=, 2nd=, 3rd="
```

Tento kód názorně demonstruje, že hodnoty \$1 až \$3 nesouvisí s porovnáním, které bylo provedeno uvnitř volání sub. V tomto případě mohou být použity speciální kódy \1, \2 atd.

```
str = "a123b45c678"
str.sub(/(a\d+)(b\d+)(c\d+)/, '1st=\1, 2nd=\2, 3rd=\3')
# "1st=a123, 2nd=b45, 3rd=c768"
```

Povšimněte si, že v předchozím fragmentu kódu jsme použili apostrofy. Pokud použijete klasické uvozovky, budou obrácená lomítka interpretována jako osmičkové (octal) únikové sekvence:

```
str = "a123b45c678"
str.sub(/(a\d+)(b\d+)(c\d+)/, "1st=\1, 2nd=\2, 3rd=\3")
# "1st=\001, 2nd=\002, 3rd=\003"
```

Způsob, jak tohle obejít, spočívá v použití dvojitéch únikových sekvencí:

```
str = "a123b45c678"
```

```
str.sub(/(a\d+)(b\d+)(c\d+)/, "1st=\\1, 2nd=\\2, 3rd=\\3")
# "1st=a123, 2nd=b45, 3rd=c678"
```

Dále je možné použít blokovou formu nahrazování, ve které mohou být použity globální proměnné, viz následující fragment kódu:

```
str = "a123b45c678"
str.sub(/(a\d+)(b\d+)(c\d+)/) { "1st=#$1, 2nd=#$2, 3rd=#$3" }
# "1st=a123, 2nd=b45, 3rd=c678"
```

Při použití bloku tímto způsobem není možné použít speciální čísla s obrácenými lomítky uvnitř řetězce obklopeného klasickými uvozovkami (nebo dokonce apostrofy).

Zde je vhodný okamžik, abych se zmínil o možnosti nezachytávajících (noncapturing) skupin. Někdy můžete chtít na znaky pohlížet jako na skupinu, například z důvodu vytvoření nějakého rafinovaného regulárního výrazu, ale nepotřebujete se odkazovat na odpovídající hodnoty při pozdějším použití. V těchto případech můžete použít nezachytávající skupinu (noncapturing group), která je označena prostřednictvím syntaxe (?:...):

```
str = "a123b45c678"
str.sub(/(a\d+)(?:b\d+)(c\d+)/, "1st=\\1, 2nd=\\2, 3rd=\\3")
# "1st=a123, 2nd=c678, 3rd="
```

V tomto fragmentu kódu byla druhá skupina "zapomenuta", takže to, co bylo třetí částí výsledku porovnání, se stalo druhou.

Osobně nemám rád notaci \\1 stejně jako notaci \$1. Ano – je pravda, že někdy jsou tyto notace pohodlné, nicméně vůbec není nutné je používat, protože to můžeme dělat "hezčím", více objektově orientovaným, způsobem.

Metoda třídy Regexp.last_match vrací objekt třídy MatchData (stejně jako to dělá metoda match instance). Tento objekt poskytuje metody instance, které umožňují programátorovi zpřístupňovat zpětné reference. S objektem MatchData se manipuluje prostřednictvím notace s hranatými závorkami (jako by se jednalo o pole shod). Speciální prvek 0 obsahuje kompletní text odpovídajícího řetězce. Následně se prvek n odkazuje na n-tou shodu:

```
pat = /(.[aiu])(.[aiu])(.[aiu])(.[aiu])/i
# Čtyři identické skupiny v tomto vzoru
refs = pat.match("Fujiyama")
# refs je nyní: ["Fujiyama", "Fu", "ji", "ya", "ma"]
x = refs[1]
y = refs[2..3]
refs.to_a.each {|x| print "#{x}\n"}
```

Povšimněte si skutečnosti, že objekt ref není opravdovým polem. Takže – když s ním chceme zacházet jako s polem za použití iterátoru each, musíme ho prostřednictvím to_a (jak je v případě výše ukázáno) převést na pole.

K tomu, abyste lokalizovali odpovídající podřetězec uvnitř původního řetězce, můžete použít i další techniky. Metody `begin` a `end` vrací počáteční a koncový offset shod. (Je důležité si uvědomit, že koncový offset je ve skutečnosti indexem znaku následujícího po posledním znaku shody při porovnání.)

```
str = "alpha beta gamma delta epsilon"
#      0.....5.....0.....5.....0.....5....
#      (pro naše konvence)
```

```
pat = /(b[ ^ ]+ )(g[ ^ ]+ )(d[ ^ ]+ )/
# Tři slova, pro každé jedna shoda
refs = pat.match(str)
```

```
# "beta "
p1 = refs.begin(1)      # 6
p2 = refs.end(1)        # 11
# "gamma "
p3 = refs.begin(2)      # 11
p4 = refs.end(2)        # 17
# "delta "
p5 = refs.begin(3)      # 17
p6 = refs.end(3)        # 23
# "beta gamma delta"
p7 = refs.begin(0)      # 6
p8 = refs.end(0)        # 23
```

Metoda `offset` podobným způsobem vrací pole se dvěma čísly, což je počáteční a koncový offset tohoto srovnání. Pokračování předchozího příkladu:

```
range0 = refs.offset(0) # [6,23]
range1 = refs.offset(1) # [6,11]
range2 = refs.offset(2) # [11,17]
range3 = refs.offset(3) # [17,23]
```

Části řetězce před a po srovnání podřetězce mohou být získány prostřednictvím metod `pre_match` a `post_match`. Pokračování předchozího příkladu:

```
before = refs.pre_match # "alpha "
after = refs.post_match # "epsilon"
```