

KAPITOLA 8

Pole, haš a ostatní výčty

Všechny části by do sebe měly zapadat bez použití hrubé síly. Mějte na paměti, že díly, které právě skládáte, jste předtím rozebrali. Pokud je nemůžete znovu složit dohromady, musí to mít nějaký důvod. V žádném případě nepoužívejte kladivo.

– IBM, návod k obsluze, 1925

Jednoduché proměnné rozhodně nepostačují pro skutečné programování. Každý moderní jazyk podporuje nejenom složitější formy strukturovaných dat, ale také mechanismy pro vytváření nových abstraktních datových typů. Historicky jsou nejstarší a nejrozšířenější datovou strukturou pole. Ve Fortranu se jim říkalo indexové proměnné, a ačkoliv později došlo k určitým změnám, jejich základní myšlenka zůstala stejná ve všech programovacích jazycích.

V současné době se extrémně populárním programovacím nástrojem stává haš. Podobně jako v případě pole je i haš indexovanou kolekcí datových položek, ovšem na rozdíl od něj může být indexován libovolným objektem. (V Ruby – jako ve většině ostatních programovacích jazyků – jsou prvky pole přístupné prostřednictvím číselného indexu.)

Později v této kapitole se obecněji podíváme na modul `Enumerable` a na to, jakým způsobem pracuje. Jak pole, tak i haš tento modul zahrnují jako `mix-in`. Stejně tak mohou modul využívat i všechny třídy, pro které má taková funkcionalita smysl. Ale nepředbíhejme. Začneme s poli.

8.1 – Práce s poli

Pole v Ruby jsou indexována celými čísly od nuly, stejně jako v případě polí v jazyce C. Nicméně zde veškerá podobnost končí. Pole v Ruby jsou dynamická. Při jejich vytváření je možné (ale nikoliv nezbytné) specifikovat jejich velikost. Po svém vytvoření mohou růst podle potřeby, bez jakéhokoliv zásahu programátora.

Pole v Ruby jsou heterogenní v tom smyslu, že mohou uchovávat různé datové typy, nikoliv pouze jeden. Ve skutečnosti je to tak, že ukládají reference na objekty (nikoliv objekty samotné), s výjimkou případů bezprostřední hodnoty jako u `Fixnum`.

Pole si uchovává svou velikost, takže se nemusíme zdržovat jejím výpočtem nebo ji ukládat v externí proměnné, kterou bychom museli udržovat synchronizovanou s daným polem. V praxi jsou iterátory často definovány tak, abychom zřídka kdy potřebovali znát velikost pole.

A konečně – třída `Array` v Ruby poskytuje polím mnoho užitečných funkcí pro přístup, hledání, zřetězení a další manipulace s poli. Ve zbývajících částech této sekce tuto třídu podrobně prozkoumáme a rozšíříme její vestavěnou funkčnost.

8.1.1 – Vytvoření a inicializace pole

Speciální metoda třídy `[]` je používána pro vytvoření pole. Datové položky, které jsou uvedeny v hranatých závorkách, jsou použity po naplnění pole. Následující řádky kódu nám ukazují tři způsoby volání této metody. (Pole `a`, `b` a `c` budou naplněna stejnými hodnotami).

```
a = Array.[](1,2,3,4)
b = Array[1,2,3,4]
c = [1,2,3,4]
```

V Ruby existuje metoda třídy nazvaná `new`, která může akceptovat žádný, jeden nebo dva parametry. První parametr je počáteční velikost pole (počet prvků). Druhý parametr je počáteční hodnota pro každý prvek:

```
d = Array.new           # Vytvoří prázdné pole
e = Array.new(3)        # [nil, nil, nil]
f = Array.new(3, "blah") # ["blah", "blah", "blah"]
```

Pečlivě se podívejte na poslední řádek předcházejícího fragmentu kódu. Obvyklou začátečnickou chybou je myslet si, že objekty v poli jsou odlišné. Ve skutečnosti se jedná o tři reference (odkazy) na stejný objekt. Pokud tedy z nějakého důvodu změníte objekt (místo jeho nahrazení za jiný objekt), změníte všechny prvky pole. Abyste se mohli vyhnout tomuto chování, použijte blok. Potom bude tento blok vyhodnocen pro každý prvek, takže každý prvek bude jiný objekt:

```
f[0].capitalize!      # f je nyní: ["Blah", "Blah", "Blah"]
g = Array.new(3) { "blah" } # ["blah", "blah", "blah"]
g[0].capitalize!      # g je nyní: ["Blah", "blah", "blah"]
```

8.1.2 – Zpřístupnění a přiřazení prvků pole

Reference na prvky a přiřazování se provádí prostřednictvím metod třídy `[]` a `[]=` (v tomto pořadí). Každá metoda třídy akceptuje celočíselný parametr, dvojici celých čísel (začátek a délku) nebo rozsah. Záporný index počítá od konce pole (začíná číslem `-1`).

Speciální metoda instance `at` slouží jako jednoduchá reference na prvek. Protože může přijímat pouze jediný celočíselný parametr, je o něco málo rychlejší.

```
a = [1, 2, 3, 4, 5, 6]
b = a[0]                # 1
c = a.at(0)             # 1
d = a[-2]               # 5
e = a.at(-2)            # 5
f = a[9]                # nil
g = a.at(9)             # nil
h = a[3,3]              # [4, 5, 6]
i = a[2..4]             # [3, 4, 5]
j = a[2...4]            # [3, 4]

a[1] = 8                # [1, 8, 3, 4, 5, 6]
a[1,3] = [10, 20, 30]   # [1, 10, 20, 30, 5, 6]
a[0..3] = [2, 4, 6, 8]  # [2, 4, 6, 8, 5, 6]
a[-1] = 12              # [2, 4, 6, 8, 5, 12]
```

V následujícím příkladě si povšimněte, jak reference směřující za konec pole způsobí změnu jeho velikosti. Také si povšimněte, že podpole (subarray) může být nahrazeno větším množstvím prvků, než v něm původně bylo, což také způsobí změnu jeho velikosti.

```
k = [2, 4, 6, 8, 10]
k[1..2] = [3, 3, 3]     # [2, 3, 3, 3, 8, 10]
k[7] = 99               # [2, 3, 3, 3, 8, 10, nil, 99]
```

A nakonec bychom se měli zmínit o tom, že pole, které je přiřazeno jedinému prvku, se ve skutečnosti vloží jako vnořené pole (na rozdíl od přiřazení do rozsahu):

```
m = [1, 3, 5, 7, 9]
m[2] = [20, 30]         # [1, 3, [20, 30], 7, 9]

# na druhou stranu...
m = [1, 3, 5, 7, 9]
m[2..2] = [20, 30]      # [1, 3, 20, 30, 7, 9]
```

Metoda `slice` slouží jako alias pro metodu `[]`:

```
x = [0, 2, 4, 6, 8, 10, 12]
a = x.slice(2)          # 4
b = x.slice(2,4)        # [4, 6, 8, 10]
c = x.slice(2..4)       # [4, 6, 8]
```

Speciální metody `first` a `last` vrací první a poslední prvek pole. Pokud je pole prázdné, bude vráceno `nil`, viz následující fragment kódu:

```
x = %w[alpha beta gamma delta epsilon]
a = x.first           # "alpha"
b = x.last            # "epsilon"
```

Předvedli jsme vám, že některé techniky pro odkazování na prvky skutečně vrací celé podpole. Existuje pár dalších způsobů, jak hromadně přistupovat k prvkům, takže se na ně teď podíváme.

Metoda `values_at` akceptuje seznam indexů a vrací pole skládající se pouze z těchto prvků. Toto může být použito tam, kde nelze použít rozsah (tzn. v situaci, ve které spolu nesousedí všechny prvky). V předchozí verzi Ruby byla metoda `values_at` nazvána jako `indices`, přičemž aliasem byl `indexes`. Tyto názvy v současné verzi Ruby nefungují.

```
x = [10, 20, 30, 40, 50, 60]
y = x.values_at(0, 1, 4)    # [10, 20, 50]
z = x.values_at(0..2,5)     # [10, 20, 30, 60]
```

8.1.3 – Nalezení velikosti pole

Metoda `length` (nebo její alias `size`) vrací počet prvků v poli. (Jako vždy je tato hodnota o jedničku větší než index posledního prvku).

```
x = ["a", "b", "c", "d"]
a = x.length             # 4
b = x.size               # 4
```

Metoda `nilitems` je úplně stejná, až na to, že nepočítá prvky `nil`:

```
y = [1, 2, nil, nil, 3, 4]
c = y.size               # 6
d = y.length             # 6
e = y.nilitems          # 4
```

8.1.4 – Porovnávání polí

Porovnávání polí je docela choulostivé, takže pokud to potřebujete udělat, dělejte to opatrně. Metoda instance `<=>` se používá pro porovnání polí. Pracuje stejně jako v jiných kontextech – vrací buď `-1` (znamenající "menší než"), `0` (znamenající "rovno"), nebo `1` (znamenající "větší než"). Na této metodě jsou závislé metody `==` a `!=`.

Pole jsou porovnávána pěkně prvek po prvku. První dva prvky, které se nerovnají, určí nerovnost celého procesu porovnání. (To znamená, že přednost je dána prvku, který je nejvíce vlevo, stejně jako je tomu v případě, když porovnáváme dvě velká celá čísla "od oka", postupně po jedné číslici).

```
a = [1, 2, 3, 9, 9]
b = [1, 2, 4, 1, 1]
c = a <=> b                # -1 (znamená a < b)
```

Pokud se všechny prvky rovnají, pak se rovnají i pole. Pokud je jedno pole delší než druhé, přičemž do délky kratšího pole jsou si rovny, pak je delší pole považováno za větší.

```
d = [1, 2, 3]
e = [1, 2, 3, 4]
f = [1, 2, 3]
if d < e                    # false
  puts "d is less than e"
end
if d == f
  puts "d equals f"        # Vytiskne "d equals f"
end
```

Protože třída `Array` není šířena modulem `Comparable`, nejsou pro třídu definovány běžné operátory `<`, `>`, `<=` a `>=`. Ale pokud chcete, můžete je jednoduše nadefinovat sami:

```
class Array

  def <(other)
    (self <=> other) == -1
  end

  def <=(other)
    (self < other) or (self == other)
  end

  def >(other)
    (self <=> other) == 1
  end

  def >=(other)
    (self > other) or (self == other)
  end

end
```

Pokud ovšem sami rozšíříte `Array` o modul `Comparable`, bude všechno jednodušší:

```
class Array
  include Comparable
end
```

Jakmile máme definované tyto nové operátory, můžeme je použít tak, jak byste očekávali:

```
if a < b
    print "a < b"      # Vytiskne "a < b"
else
    print "a >= b"
end
if d < e
    puts "d < e"      # Vytiskne "d < e"
end
```

Je možné, že výsledkem porovnání polí bude porovnání dvou prvků, pro které operátor `<=>` není definován nebo nemá význam. Následující kód způsobí chybu za běhu (`TypeError`), protože porovnání `3 <=> "x"` je problematické:

```
g = [1, 2, 3]
h = [1, 2, "x"]
if g < h          # Chyba!
    puts "g < h"  # Žádný výstup
end
```

Nicméně – v případě, že stále nejste zmateni, rovnost a nerovnost budou v tomto případě stále pracovat. To proto, že dva objekty různého typu jsou přirozeně považovány za nerovné, i když nemůžeme říct, který z nich je větší, nebo menší než druhý.

```
if g != h          # Bez problémů.
    puts "g != h"  # Vytiskne "g != h"
end
```

A konečně – je možné, že dvě pole, která obsahují neodpovídající si datové typy, budou přeci jen porovnána pomocí operátorů `<` a `>`. V takovém případě dostaneme výsledek ještě předtím, než narazíme na neporovnatelné prvky:

```
i = [1, 2, 3]
j = [1, 2, 3, "x"]
if i < j          # Bez problémů.
    puts "i < j"  # Vytiskne "i < j"
end
```

8.1.5 – Řazení polí

Nejjednodušším způsobem, jak seřadit pole, je použít zabudovanou metodu `sort`:

```
words = %w(the quick brown fox)
list = words.sort    # ["brown", "fox", "quick", "the"]
```

Nebo takto:

```
words.sort! # ["brown", "fox", "quick", "the"]
```

Tato metoda předpokládá, že všechny prvky v poli jsou porovnatelné s ostatními. Pomíchané pole, jako třeba [1, 2, "three", 4], normálně vrátí chybu typu. V případě, jako je tento, můžete použít blokovou formu volání stejné metody. Následující příklad předpokládá, že existuje alespoň metoda `to_s` pro každý prvek (pro převedení na `string`):

```
a = [1, 2, "three", "four", 5, 6]
b = a.sort {|x,y| x.to_s <=> y.to_s}
# b is now [1, 2, 5, 6, "four", "three"]
```

Je samozřejmé, že takové řazení (v tomto případě závisující na ASCII) nemusí být smysluplné. Pokud máte takové různorodé pole, zeptejte se prvně sami sebe, proč jej vlastně řadíte nebo proč vůbec ukládáte objekty různých typů.

Tato technika funguje, protože blok vrací celé číslo (-1, 0 nebo 1) při každém volání. Když je vráceno číslo -1, znamená to, že `x` je menší než `y`; dva prvky jsou zaměněny. Takže pro řazení v sestupném pořadí můžeme jednoduše prohodit pořadí porovnání:

```
x = [1, 4, 3, 5, 2]
y = x.sort {|a,b| b <=> a} # [5, 4, 3, 2, 1]
```

Blok může být také použit pro komplexnější řazení. Předpokládejme, že chceme seřadit seznam knih a filmových titulů tímto způsobem – ignorovat velikost písmen, zcela ignorovat mezery a ignorovat jistý druh vložené interpunkce. Zde prezentujeme jednoduchý příklad. (Věříme, že jak učitelé angličtiny, tak i programátoři budou zmateni z tohoto druhu řazení podle abecedy).

```
titles = ["Starship Troopers",
          "A Star is Born",
          "Star Wars",
          "Star 69",
          "The Starr Report"]
sorted = titles.sort do |x,y|
  # Smaže členy (a, an, the)
  a = x.sub(/^(a |an |the )/i, "")
  b = y.sub(/^(a |an |the )/i, "")
  # Smaže mezery a interpunkci
  a.delete!(".", "-?!")
  b.delete!(".", "-?!")
  # Převede na velká písmena
  a.upcase!
  b.upcase!
  # Porovná a a b
  a <=> b
```

```
end
# Výsledek je nyní:
# [ "Star 69", "A Star is Born", "The Starr Report"
#   "Starship Troopers", "Star Wars"]
```

Tento příklad není příliš použitelný a určitě by mohl být napsán kompaktněji. Pointa je v tom, že při porovnání dvou operandů může být na nich vykonán libovolně složitý soubor operací. (Nicméně si povšimněte, že originální operandy jsme nechali nedotčeny; pracovali jsme s jejich kopiemi.) Tato metoda může být užitečná v mnoha situacích – například při řazení podle několika klíčů nebo podle klíčů, které jsou vypočítány až při běhu programu.

V novějších verzích Ruby obsahuje modul Enumerable metodu `sort_by`, která je samozřejmě součástí Array. Je velmi důležité ji pochopit. Metoda `sort_by` používá to, co lidé od Perlu nazývají Schwartzovou transformací (podle Randala Schwartze). Místo abychom řadili podle prvků samotných, aplikujeme na ně nějakou funkci nebo mapování a řadíme podle výsledku.

Představte si, že máte seznam souborů, který chcete seřadit podle velikosti. Přímočarý způsob by vypadal nějak takto:

```
files = files.sort {|x,y| File.size(x) <=> File.size(y) }
```

Nicméně jsou zde dva problémy. Zaprvé – vypadá to trochu ukecaně. Měli bychom být schopni tento fragment kódu trochu zestručnit. Zadruhé – dochází k vícenásobnému přístupu na disk, což je docela drahá operace (ve srovnání s jednoduchými operacemi v paměti). Čím více takových operací, tím hůře. Použitím metody `sort_by` ovšem vyřešíme oba tyto problémy najednou. Zde je správný způsob, jak to udělat:

```
files = files.sort_by {|x| File.size(x) }
```

V předchozím fragmentu kódu je každý klíč počítán pouze jednou. Výsledek je následně interně uložen jako dvojice klíč/data. Ačkoliv v případě menších polí může mít tento způsob efektivitu naopak nižší, může zvýšení čitelnosti kódu i tak stát za to.

Metoda `sort_by`! neexistuje. Nicméně si můžete vždy napsat svou vlastní.

A co řazení na základě více klíčů? Představte si, že máte pole objektů, která potřebujete seřadit na základě těchto tří atributů – jméno, věk a výška. Skutečnost, že pole jsou vzájemně porovnatelná, znamená, že následující technika bude funkční:

```
list = list.sort_by {|x| [x.name, x.age, x.height] }
```

Je samozřejmé, že nejste omezení pouze na jednoduché prvky pole, které byly použity ve výše uvedených příkladech. Prvkem pole může být libovolný výraz.

8.1.6 – Výběr z pole na základě kritéria

Někdy chceme lokalizovat prvek (nebo prvky) v poli podobným způsobem, jako se dotazujeme na tabulky v databázi. Existuje několik způsobů, jak to udělat. Všechny níže nastíněné způsoby pochází z modulu Enumerable.

Metoda `detect` nalezne nanejvýš jediný prvek. Akceptuje blok (ve kterém jsou prvky procházeny sekvenčně), přičemž vrátí první prvek, pro nějž je výraz v bloku pravdivý.

```
x = [5, 8, 12, 9, 4, 30]
# Najde první násobek 6
x.detect {|e| e % 6 == 0 }      # 12
# Najde první násobek 7
x.detect {|e| e % 7 == 0 }      # nil
```

Objekty v poli mohou být samozřejmě libovolně složité, stejně jako test v bloku.

Metoda `find` je synonymem pro metodu `detect`. Metoda `find_all` je varianta, která vrátí i více prvků. Metoda `select` je pak synonymem pro metodu `find_all`:

```
# Pokračování předchozího příkladu...
x.find {|e| e % 2 == 0 }        # 8
x.find_all {|e| e % 2 == 0 }    # [8, 12, 4, 30]
x.select {|e| e % 2 == 0 }      # [8, 12, 4, 30]
```

Metoda `grep` volá operátor rovnosti `case` pro porovnání každého prvku s daným vzorem. V nejjednodušší formě vrátí pole obsahující shodující se prvky. Protože je použit operátor rovnosti `case` (`==`), vzor nemusí být regulárním výrazem. (Název `grep` samozřejmě pochází ze světa Unixu a historicky souvisí s příkazem `g/re/p`).

```
a = %w[January February March April May]
a.grep(/ary/)                  # ["January", "February"]
b = [1, 20, 5, 7, 13, 33, 15, 28]
b.grep(12..24)                 # [20, 13, 15]
```

Existuje bloková forma, která transformuje každý výsledek ještě předtím, než ho uloží do pole. Výsledné pole pak obsahuje návratové hodnoty bloku, nikoliv hodnoty, které byly do bloku předány:

```
# Pokračování předchozího příkladu...
# Pojďme uložit délky řetězců
a.grep(/ary/) {|m| m.length}   # [7, 8]
# Pojďme umocnit každou hodnotu
b.grep(12..24) {|n| n*n}       # {400, 169, 225}
```

Metoda `reject` je doplňková (komplementární) k metodě `select`. Vylučuje každý prvek, který blok vyhodnotí jako `true`. Je také definována metoda `reject!`: