Metaprogramování v Ruby

Pavel Pokorný

1. května 2012

Ruby je dynamický jazyk se zaměřením na jednoduchost a produktivitu. Elegantní syntaxe se přirozeně čte a jednoduše píše. Toto tvrzení jsem převzal z oficiální webové stránky jazyka Ruby a mohu jen souhlasit. I velice složité věci je možné v napsat v tomto jazyce s nebývalou elegancí. Co se zdá na první pohled jednoduché, nemusí být ale jednoduché pod povrchem, což dokazují i slova tvůrce tohoto jazyka. Pro vytvoření tak použitelného a přístupného jazyka jako je Ruby musel Yukihiro Matsumoto vynaložit obrovské úsilí.

> „Ruby je jazyk jednoduchý navenek, ale uvnitř je velice komplexní, stejně jako lidské tělo."
> *Yukihiro 'Matz' Matsumoto, tvůrce Ruby*

V tomto článku jsou popsány základy objektového modelu, spustitelných objektů a možnosti metaprogramování v jazyce Ruby.

Metaprogramování je psaní kódu, který nějakým způsobem za běhu programu manipuluje s konstrukcemi jazyka. Ruby přímo vyzývá svými vlastnostmi k metaprogramování, vše může být vytvořeno nebo změněno za běhu programu. Programátor má tedy velikou volnost a takřka neomezené možnosti, ale s tím přichází i větší zodpovědnost. Není těžké 'rozbít' základní funkčnost vestavěných tříd a objektů a taktéž při použití jakékoliv knihovny je možné redefinovat kteroukoliv z poskytovaných funkčností. Od programátora se tedy předpokládá jakási ukázněnost a je na něj převedena veškerá zodpovědnost. Na oplátku získává volnost a možnost si vše přizpůsobit či rozšířit.

Nejspíše i kvůli těmto vlastnostem jazyka lpí komunita Ruby vývojářů tak moc na otestovaném kódu. Mnohdy kód bez testů jako by ani nebyl. Když už jsem se zmínil o komunitě, je tu ještě jedna věc - a sice podpora open source software. Sdílet zdrojový kód a nechat kohokoliv na světě ho volně užívat a popřípadě vylepšovat je zkrátka trend a Ruby vývojáři můžou jít ostatním jen příkladem. Většina projektů využívá server GitHub, který

nabízí pro open source projekty zdarma hosting verzovacího systému Git, správu issues, wiki a vestavěné sociální prvky, což vystihuje podtitul '*social coding*'. GitHub se zkrátka stává Facebookem pro programátory.

V článku předpokládám, že má čtenář základní znalosti jazyka Ruby, pokud tomu tak není, doporučuju si projít nespočet online tutoriálů nebo skvělou knihu *Eloquent Ruby* [3], která obsahuje také velkou sekci o metaprogramování. Mnoho užitečných informací jsem se dozvěděl také v knize *Metaprogramming Ruby* [4], která pokrývá pokročilejší vlastnosti tohoto jazyka a obsahuje mimo jiné spoustu zažitých vzorů řešení různých problémů (v knize jsou nazývány tyto vzory jako kouzla - *spells*).

1 Základní struktury jazyka

Vše je objekt, je tedy rozumné začít u něj. Objekt v Ruby je na první pohled jednoduchá struktura, obsahuje své metody, proměnné, referenci na svou třídu a unikátní identifikátor (*object_id*).

Podobně jako v jiných objektově orientovaných jazycích obsahuje **instanční proměnné** a narozdíl od statických proměnných nejsou instační proměnné závislé na třídě objektu. Proměnná objektu je vytvořena, když je jí přiřazena hodnota. Můžou tedy existovat objekty stejné třídy obsahující úplně jiné instační proměnné. Seznam instančních proměnných objektu je možné získat od každého objektu zavoláním metody *instance_variables*.

Jak se dá předpokládat, objekty mají **metody**, které je možné zavolat na daném objektu. Objekty dědí metody ze svého tzv. *ancestors chain*. Metody však nejsou uloženy přímo v objektu samém, ale v třídě daného objektu (což je objekt třídy *Class*). Seznam metod, které má objekt k dispozici, je možné získat metodou *methods* nebo variantami *instance_methods*, *singleton_methods*.

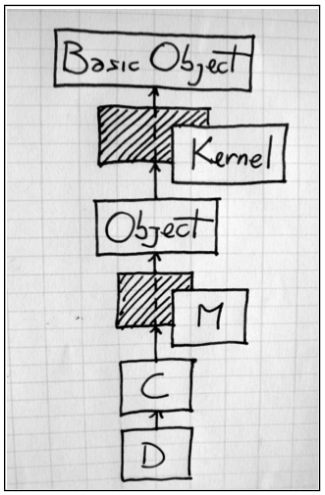
Třídy jsou také objekty a vztahují se k ním stejná pravidla jako k objektům. Třídy mají také svou třídu, která se nazývá *Class*, a mají také metody, které budu dále označovat jako třídní metody. Třída *Class* není v hierarchii na vrcholu, dědí od třídy *Module*. Třída je tedy modul, který obsahuje některé důležité metody navíc (*new*, *allocate*, *superclass*), díky kterým lze vytvořit její instanci. Každá instance třídy *Class* (tedy každá třída) je potomkem třídy *Object*, která je potomkem třídy *BasicObject*. *BasicObject* je opravdovým vrcholem v objektovém modelu jazyka Ruby. Není těžké se v tomto světě propletených tříd a objektů ztratit, k pochopení snad přispěje ilustrace 3, ke které se dostaneme ke konci článku.

Moduly a třídy jsou úzce spjaty a v mnoha případech jsou zaměnitelné. Moduly jsou používány, když je kód určen k vložení do jiných metod

a mluvíme pak o tzv. *code mixins*. To zvyšuje znovupoužitelnost takového kódu. Druhou funkcí modulů je organizace kódu do tzv. *namespaces*, což jednak brání kolizím v názvech tříd a dovoluje seskupovat logicky související metody, třídy a konstanty. Třídy jsou používány, pokud je třeba vytvářet instance nebo využívat dědičnost.

Reference na objekty jsou udržovány v proměnných a protože třídy a moduly jsou také objekty, je nutné reference na ně také někde udržovat. Reference na třídy a moduly jsou **konstanty**, což jsou v Ruby všechny názvy začínající velkým písmenem. Všechny konstanty jsou uspořádány do stromu, kde uzly jsou konstanty ukazující na třídy a moduly a listy ukazují na ostatní konstanty.

Když je zavolána **metoda**, vyhledá ji Ruby interpret v *ancestors chain*, tedy „šňůře" tříd začínající třídou objektu a končící třídou *BasicObject*. O volání metod se můžeme vyjadřovat jako o zasílání zpráv, příjemce budiž tedy objekt, jehož metoda byla zavolána. Tento přístup je v ruby přirozený, každý objekt má metodu *send*, kterou je možné mu zaslat zprávu i s případnými parametry. Když interpret nalezne metodu, nastaví příjemce jako aktuální objekt (který je vždy možné získat klíčovým slovem *self*) a spustí kód metody. Pokud není explicitně uveden příjemce, interpret předpokládá, že se jedná o metodu objektu *self* a taktéž instanční proměnné hledá u tohoto objektu. Každý kousek Ruby kódu je vykonáván v prostředí nějakého objektu a tento aktuální objekt lze získat klíčovým slovem *self*. Při definici nové třídy zastává roli *self* právě definovaná třída. V tělech metod je *self* objektem, na kterém byla metoda volaná. V kontextu nejvyšší úrovně zastává roli *self* objekt nazvaný *main*, který je automaticky vytvořen Ruby interpretem. Object *main* je speciální také tím, že metody vytvořené v jeho kontextu jsou uloženy jako privátní metody třídy *Object* a jsou tak jednoduše přístupné všem objektům.



Obrázek 1: Moduly *Kernel* a *M* vložené do *ancestors chain* objektů třídy D [4]

V Ruby existuje spousta metod, které jsou dostupné odkudkoliv, např. *puts*, *print*, *rand*. Tyto metody jsou definovány v tzv. **Kernel** modulu jako privátní instační proměnné. *Kernel* modul je vložen do třídy *Object* a je tak součástí *ancestors chain* každého objektu. Díky způsobu, jakým Ruby vyhledává metody, jsou tyto metody dostupné z každého objektu.

Na obrázku 1 je vidět modul *Kernel* vložen do třídy *Object* a modul *M* vložen do třídy *C*.

Z pohledu metaprogramování je zajímavá *Kernel* metoda **eval**, která přijímá textový řetězec obsahující Ruby kód, provede jej a vrátí výsledek. Tímto způsobem může být za běhu spuštěn libovolný kód. Řetězce s kódem jsou velice podobné blokům, ale mají nevýhody. Jsou velice těžce čitelné a udržovatelné. Pokud string obsahuje chybu v syntaxi, interpret to nenahlásí, protože není možné určit, které stringy obsahují kód a nějakým způsobem ho kontrolovat. Z bezpečnostního pohledu přináší tato metoda problém v možnosti *code injection*, pokud jsou součástí nějaká data získaná od uživatele. Z těchto důvodů je vhodnější používat jiné možnosti.

2 Spustitelné objekty

V Ruby jsou metody součástí rodiny spustitelných objektů, do níž patří také bloky, procedury a lambda procedury. S bloky se v Ruby setkáte velice často. Díky blokům je možné metodám předat kus vlastního kódu využívajícího hodnot parametrů, které mu metoda dodá. Je to taky dobrý nástroj pro kontrolu oblasti viditelnosti proměnných a metod.

2.1 Bloky

Bloky mohou být vytvořeny pomocí složených závorek nebo klíčových slov *do* a *end*. Mohou být definovány pouze při volání metody, blok je předán metodě a ta může spustit jeho kód pomocí klíčového slova *yield*. Blok může mít argumenty a má návratovou hodnotu. Uvnitř metody je možné se dotázat, zda byl metodě předán blok, pomocí *block_given?*.

```
class SomeData
  def initialize(a,b); @a = a; @b = b; end
  def get
    return yield(@a, @b) if block_given?
    [@a, @b]
  end
end

data = SomeData.new(3,4)
data.get { |x,y| x+y }          # 7
data.get                        # [3, 4]
```

```
MyClass = Class.new {}          # class MyClass;end
MyModule = Module.new {}        # module MyModule;end
define_method :my_method {}     # def my_method;end
```

Obrázek 2: Definice nové třídy, modulu a metody pomocí volání metody

Pro spuštění kódu je zapotřebí nejen kód samotný, ale i prostředí, ve kterém má běžet. Prostředím mám na mysli soubor názvů, které se vážou k objektům (proměnné, metody, self). Toto prostředí je reprezentováno objektem třídy *Binding* a je možné ho získat klíčovým slovem *binding*. V momentě definice bloku je zachycen a uložen aktuální stav prostředí (aktuální *bindings*). Blok si tento stav 'nese' s sebou a proto jsou bloky nazývány *closures*.

V určitých okamžicích je vytvořeno nové prostředí, stává se to při definici nové metody pomocí klíčového slova *class*, modulu pomocí *module* a metody pomocí *def*. Tyto konstrukce se nazývají *scope gates* a po vytvoření nového *scope* např. ztrácíme přístup k dosud definovaným lokálním proměnným. Toto chování je však možné obejít nahrazením zmíněných klíčových slov voláním metody, které má stejný efekt, ale je jí předán blok s kódem, který si s sebou nese i prostředí. Na obrázku 2 jsou použity tyto alternativní definice. Bloky mohou být vykonány v prostředí kteréhokoliv objektu pomocí metody *instance_eval*.

O Ruby se často dočtete, že vše v tomto jazyce je objekt. Bloky ale nejsou objekty a to z výkonnostních důvodů. Ovšem k uložení kódu a jeho pozdějšímu spuštění objekt potřebujeme a k tomu důvodu existuje *Proc*. *Proc* je objekt vytvořený z bloku, uložený kód je možné později spustit zavoláním metody *call*.

Blok může být převeden do objektu třídy *Proc* hned čtyřmi způsoby:

- Objekt *Proc* je vytvořen při předání bloku metodě *Proc.new*.
- V *Kernel* modulu existují dvě pomocné metody *proc* a *lambda*, které fungují obdobně.
- Pomocí & operátoru při zachycení bloku v seznamů parametrů metody.

Block je anonymním argumentem volané metody a obvykle je zavolán již zmíněným klíčovým slovem *yield*. Pokud však předaný blok chceme uložit k pozdějšímu použití nebo předat dál jiné metodě, musíme blok převést na *Proc* objekt. Toho dosáhneme tak, že bloku dáme název, který uvedeme na konec seznamu parametrů metody. Tento název musí začínat **operátorem &**, tím dáváme najevo, že se jedná o Proc objekt, který chceme používat jako

blok. Ruby automaticky vytvoří z dodaného bloku *Proc* objekt. Pokud dále v metodě chceme použít *Proc* objekt, použijeme název bez znaku &. Pokud chceme použít blok (a předat ho dále jiné metodě), použijeme celý název i se znakem &.

```
def my_method(&my_proc)
  my_proc.class          # Proc
  yield
end
my_method { "block" }  # block

other_proc = proc { 'proc turned to a block' }
my_method(&my_proc)     # proc turned to a block
```

2.2 Proc a Lambda

Ať už použijeme metodu *proc* nebo *lambda*, dostaneme objekt třídy *Proc*. Je mezi nimi však drobný rozdíl. Klíčové slovo *return* se chová odlišně, v lambda *Proc* pouze ukončí vykonávání kódu lambdy a určí její návratovou hodnotu. V klasickém *Proc* objektu se však navrátí z místa, kde byla procedura definována, což může být zdrojem nemilého překvapení pro neznalého programátora. Druhý rozdíl je ve zpracování argumentů. Lambda kontroluje počet parametrů a pokud je jí předáno méně parametrů, než kolik očekává, vytvoří ArgumentError výjimku. Pokud je však zavolán blok nebo Proc se špatným počtem parametrů, nastaví chybějící parametry na nil hodnotu a žádnou chybu neohlásí.

V Ruby verze 1.8 byla metoda *proc* pouze aliasem pro metodu *lambda*, v Ruby 1.9 a je synonymem pro *Proc.new*.

2.3 Metody

Metody jsou velice podobné lambda *Proc*, ale jsou spouštěny v prostředí objektu, ke kterému jsou vázany. Lze získat objekt třídy *Method*, který lze spustit metodou *call*.

2.3.1 Dynamické volání metody

Ve většině případech voláme metodu přes tzv. tečkovou notaci (*object.method_name*). U tohoto přistupu nastává problém, pokud název metody bude znám až za běhu programu (*Dynamic Dispatch*). V Ruby můžeme využít metody

send, která je definovaná v *Kernel* modulu, takže je přístupná všem objektům. První parametr této metody je název metody, kterou chceme zavolat a všechny další parametry jsou předány cílové metodě. Použitím této metody lze také překonat nastavení viditelnosti volaných metod. Metoda se zavolá i pokud je definována jako *protected* nebo *private*.

2.3.2 Dynamické vytvoření metody

Metody mohou být dynamicky i vytvářeny. Slouží k tomu metoda *define_method*, které je třeba předat jméno nové metody a blok, který se stane tělem metody. Tento způsob navíc zachovává aktuální *scope* a v metodě lze pracovat s informacemi, které by při použití klasického *def* nebyly dostupné.

```
names = %w[red green blue]
NewClass = Class.new do
  names.each do |name|
    define_method name do
      puts "I am #{name}"
    end
  end
end
```

2.3.3 Neexistující metody

Na rozdíl od kompilovaných jazyků, nemáme v Ruby překladač, který by kontroloval, zda používané metody existují. Můžeme tedy volat i neexistující metody. Pokud Ruby nenajde volanou metodu, zavolá metodu nazvanou *method_missing* s parametrem názvu hledané metody, všemi předanými parametry a případně blokem. Tato metoda je definovaná v *Kernel* modulu a vyhodí výjimku *NoMethodError*. Pokud ji však přetížíme v naši třídě, můžeme zachytávat volání neexistujících metod a podle názvu se rozhodnout, jaká akce se má provést. Objekty tak mohou odpovídat na dynamicky vytvořený soubor metod. Toho je využito například v ORM *ActiveRecord* webového frameworku *Ruby on Rails*, kde model dokáže odpovědět na metody, jejichž názvy jsou založeny na sloupcích tabulky, se kterou je model spjat. Tyto metody ve skutečnosti neexistují, ale v modelu je uložen seznam názvů sloupců tabulky a v metodě *method_missing* provádí kontrolu, zda hledaná metoda není *getter* nebo *setter* některého ze sloupců tabulky. Pokud ano, provede příslušnou akci a navíc definuje i reálnou metodu, aby se při příštím volání této metody nemusel celý proces opakovat.

2.3.4 Notifikace událostí

Důležité události v objektovém modelu jsou doprovázeny voláním tzv. *hook* metod. Ve výchozím stavu nedělají tyto metody nic, ale je možné je přetížit a spustit náš kód. Mezi tyto události patří zdědění třídy, vložení modelu do třídy nebo rozšíření třídy modelem (těžko se mi hledají česká slova, v angličtině se řekne: *module is included (or extended) into the class*, odtud také označení modulu *mixin*), definice metody nebo její odebrání.

Když je třída děděná jinou, je zavolána metoda *inherited*. Když je modul vkládán do třídy, tzn. metody jsou přidávány do samotné třídy a jsou dostupné jako instační metody, je zavolána metoda *included*. Pokud modul rozšiřuje třídu, tzn. metody jsou vkládány do její eigenclass a jsou tedy třídními metodami, je zavolána metoda *extended*. Při definici nové metody je zavolána metoda *method_added*, obdobně *method_removed* nebo *singleton_method_added*.

3 Třídy

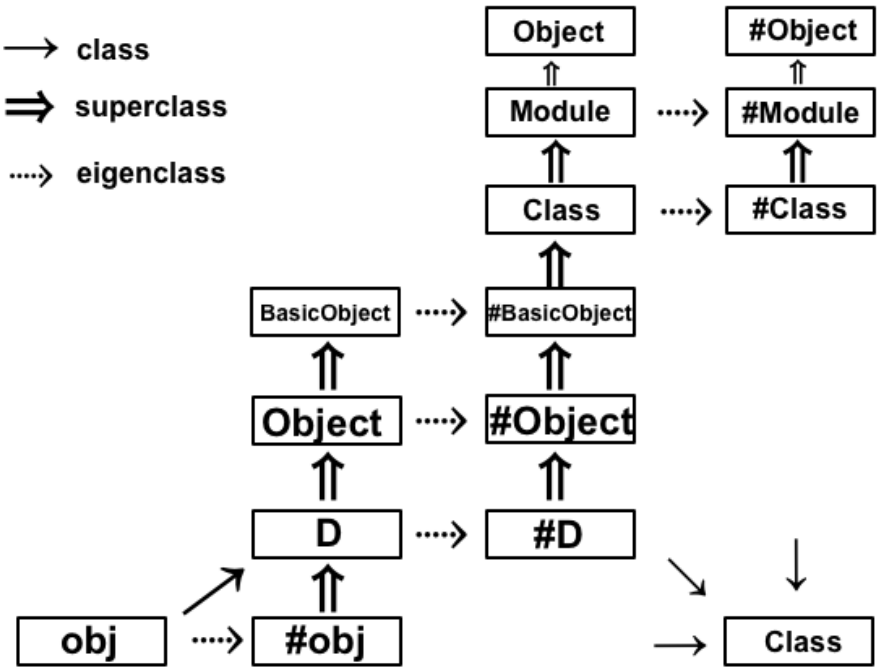
Definice nové třídy se v Ruby liší od většiny ostatních jazyků. Klíčové slovo *class* přenese dění do kontextu nově vytvářené třídy, kde mohou být vytvářeny metody, proměnné a spouštěn libovolný kód. Roli aktuálního objektu (*self*) přebírá definovaná třída nebo modul, což je možné, protože třída je také objekt.

Můžeme tak jednoduše „otevřít" kteroukoliv existující třídu a modifikovat ji. Tímto způsobem lze vytvořit spoustu těžko dohledatelných chyb a rozbít funkčnost. Často je tato technika nazývána *monkey patching*. Tato volnost má své zastánce i odpůrce a je samozřejmě tématem mnoha diskusí. Aby programátor předcházel rozbití existujícího kódu, je třeba pečlivě zkontrolovat, zda neredefinuje nějakou důležitou součást a hlavně mít kód řádně otestovaný automatickými testy. Stále ale platí, pokud existuje jiná cesta, která nezahrnuje změnu cizího kódu za běhu, určitě je to cesta vhodnější. Pokud si totiž upravíme tímto způsobem např. část cizí knihovny, může se stát a pravděpodobně se v budoucnu stane, že autor knihovnu aktualizuje a naše úprava už nebude funkční. Je tedy lepší využít zdokumentovaného rozhranní knihovny, které má určitě delší životnost než interní implementace. Ovšem né vždy je toto možné a to je pak případ pro využití monkey patche nebo při nutnosti velkých změn i úpravy zdrojových kódů knihovny. Pak jsou zde samozřejmě případy, kdy si programátor rozšíří funkčnost objektů a tříd standardní knihovny Ruby a je na zvážení každého, jak moc velké úpravy si dovolí a zda se mu vyplatí.

V Ruby je možné definovat metody pro jednotlivé objekty zvlášť, takzvané *singleton* metody. Třídní metody (odpovídající funkčností statickým metodám z jiných jazyků) jsou také *singleton* metody, jsou definovány pouze pro jeden objekt, který je zároveň třída. Typ objektu je dán jeho třídou, ale ne takovou mírou jako je tomu u statických jazyků. Typ objektu je v Ruby dán souborem metod, na které dokáže objekt odpovědět. Dva různé objekty stejné třídy totiž mohou obsahovat jiné metody. Tento výklad definice typu objektu je nazýván *duck typing* („Vidím ptáka. Pokud chodí jako kachna a kváká jako kachna, pak to musí být kachna.").

3.1 Skryté vlastní třídy *Eigenclasses*

Singleton metody nemohou být uloženy v objektu samém, protože objekt není třída. Nemohou být uloženy v třídě objektu, protože pak by byly k dispozici všem objektům této třídy a byly by to tedy běžné metody. Tak kdepak jsou? Tyto metody jsou uloženy ve speciální skryté třídě, která je nazývaná různě - *Eigenclass*, *Singleton Class*, *Metaclass*. Eigenclass lze přeložit jako „vlastní



Obrázek 3: Kompletní objektový model jazyka Ruby

třída objektu" a odpovídá to tomu, kde tato třída skutečně je. Každý objekt (tedy i třída) má jen pro sebe kopii své třídy, ve které jsou definovány singleton metody. Teprve s touto znalostí můžeme zobrazit kompletní objektový model. Na obrázku 3 je zobrazena situace, kdy jsme si definovali třídu *D* a vytvořili jednu její instanci, objekt *obj*. Značení na obrázku by mělo být jasné. Když zavoláme metodu na objektu *obj*, Ruby začne s vyhledáváním v eigenclass daného objektu a pokračuje vzhůru po předcích až k *BasicObject*. Pokud zavoláme třídní metodu třídy *D*, začne se vyhledávat u *#D* a pokračuje se opět až k *BasicObject*.

Literatura

[1] *Oficiální web jazyka Ruby*, www.ruby-lang.org

[2] *Oficiální dokumentace*, www.ruby-doc.org

[3] Russ Olsen, *Eloquent Ruby*. Addison Wesley Proffesional Ruby Series, Massachusetts, 2011.

[4] Paolo Perrotta, *Metaprogramming Ruby: Program Like the Ruby Pros*. Pragmatic Bookshelf, 2010.