The Pragmatic Programmers

Ruby元编程

Metaprogramming Ruby

Program Like the Ruby Pros

松本行弘 作序推荐

[意] Paolo Perrotta 著

廖志刚 译

陈睿杰 审校

華中科技大學出版社
http://www.hustp.com

The Facets of Ruby Serie

Ruby 元编程

[意] Paolo Perrotta 著

廖志刚 译

陈睿杰 审校

华中科技大学出版社

中国·武汉

内 容 简 介

本书以案例形式循序渐进地介绍了 Ruby 特有的实用编程技巧（元编程）。通过分析案例、讲解例题、回顾 Ruby 代码库的实现细节，作者不仅向读者展示了 Ruby 编程的优势和 Ruby 特有的解决问题的方式，更详细列出了发挥其优势的技巧和常用的 Ruby 设计模式。Ruby 创始人松本行弘作序推荐。

978-1-93435-647-0：Metaprogramming Ruby.

湖北省版权局著作权合同登记　图字：17-2011-179 号

图书在版编目（CIP）数据

Ruby 元编程 /［意］Paolo Perrotta 著；廖志刚　译；陈睿杰. 审校. —武汉：华中科技大学出版社，2012.2

ISBN 978-7-5609-7458-3

Ⅰ. R… Ⅱ. ①P… ②廖… ③陈… Ⅲ. 计算机网络—程序设计 Ⅳ. TP393.09

中国版本图书馆 CIP 数据核字(2011)第 229926 号

［意］Paolo Perrotta　著

Ruby 元编程

廖志刚　译　　陈睿杰　审校

策划编辑：徐定翔
责任编辑：陈元玉
封面设计：潘　群
责任校对：李　琴
责任监印：张正林
出版发行：华中科技大学出版社（中国·武汉）
武昌喻家山　邮编：430074　电话：（027）87557473
录　　排：武汉美乐广告设计工作室
印　　刷：湖北新华印务有限公司
开　　本：787mm×980mm　1/16
印　　张：18
字　　数：362 千字
版　　次：2012 年 2 月第 1 版第 1 次印刷
定　　价：56.00 元

读者对《Ruby 元编程》的评价

阅读此书仿佛是一头扎入一个崭新的思维世界。在上一个项目里，我尝试把 Java 和 JRuby 元编程混合使用。现在如果只使用 Java，就感觉像拿着一根香蕉参加一场决斗，而我的对手挥舞着一把半人长的日本刀。

Sebastian Hennebrüder
Java 咨询师和培训师，*laliluna.de*

本书填补了 Ruby 语言参考手册和编程案例之间的空白。它不仅解释了各种元编程的技术，还展示了编写更精练、更优良代码的方法。不过要事先警告你，熟悉了新方法后，你会难以忍受目前主流的编程方法。

Jurek Husakowski
软件设计师，Philips 应用技术

在读本书之前，我从没有见过一本书能把 Ruby 的各种概念组织和解释得如此清楚，这些概念包括 Ruby 对象模型、闭包、领域专属语言以及 eigenclass，其中很多例子都来自日常使用的类库。本书绝对值得一读。

Carlo Pecchia
软件工程师

过去我一直找不到好的方法来学习元编程，本书用一种前所未有的方式做到了。作者用轻松、愉快的方式解释了 Ruby 中最复杂的秘密，并将它们灵活应用于实际程序。

Chris Bunch
软件工程师

前言

Foreword

Ruby 的很多特性都继承自很多其他语言，这些语言包括 Lisp、Smalltalk、C、Perl 等。它的元编程的特点来自于 Lisp（以及 Smalltalk）。元编程看起来有点像魔术，让人震惊。这个世界上有两种类型的魔术：白魔术做好事，黑魔术则做肮脏的事。与之相似，元编程也有两面性。如果约束自己，则可以做出好事，比如可以发挥语言的威力，并且不需要使用宏修改语言的语法；或者也可以创建内部领域专属语言。但是，你也可能陷入元编程的黑暗之中，元编程是一种容易让人迷惑的技术。

Ruby 相信你。Ruby 把你看做是一个成熟的程序员。它赋予你诸如元编程这样强大的能力。但是你必须牢记：能力越大，责任越大。

享受 Ruby 编程吧。

matz

2009. 10

致谢

Acknowledgments

在开始之前，我需要感谢一些人。他们是：Joe Armstrong、Satoshi Asakawa、Paul Barry、Emmanuel Bernard、Roberto Bettazzoni、Ola Bini、Piergiuliano Bossi、Simone Busoli、Andrea Cisternino、Davide D'Alto、Mauro Di Nuzzo、Marco Di Timoteo、Mauricio Fernandez、Jay Fields、Michele Finelli、Neal Ford、Florian Frank、Sanne Grinovero、Federico Gobbo、Florian Groß、Sebastian Hennebrüder、Doug Hudson、Jurek Husakowski、Lyle Johnson、Luca Marchetti、MenTaLguY、Carlo Pecchia、Andrea Provaglio、Mike Roberts、Martin Rodgers、Jeremy Sydik、Andrea Tomasini、Marco Trincardi、IvanVaghi、Giancarlo Valente、DavideVarvello、Jim Weirich，以及很多在本书 beta 版时向我指出问题和错误的读者。不管你是提供了意见、引用、修订，还是精神支持，都让本书中至少一行文字变得更好。我说了“一行文字”么？对于你们中的有些人来说，它应该变成“好几章”。尤其是 Ola、Satoshi 和 Jurek，你们值得在本页中占一个特殊的位置，我对你们充满了敬意。

感谢 Pragmatic Bookshelf 的工作人员：Janet Furlow、Seth Maislin、Steve Peter、Susannah Davidson Pfalzer 和 Kim Wimpsett。Dave 和 Andy，谢谢你们在困难的时刻也依然相信我。Jill，感谢你把我那些蹩脚的文字变得轻松流畅。虽然我们在威尼斯度过了艰难的一周，但是这一切都是值得的。感谢你，Lucio，我亲爱的老朋友。

爸爸妈妈，感谢你们的支持、你们的爱以及你们从不追问本书为什么费时如此之久。

绝大多数作者会在最后感谢他们的搭档，现在我明白其原因了。当你要完成一本书时，回顾你开始写作的日子，你会感觉如此遥远。我还记得那些没有午休、夜晚以及周末的时光；那些连续几日甚至数周呆在自己的书房、异乡的某个旅馆或某个海滨别墅度过的日子，那些在最适合当隐士的地方做着孤独的努力——然而，我却从不孤单。谢谢你，Mirella。

引言

Introduction

元编程（*Metaprogramming*）……听起来很酷！听起来像是一种用于高级企业架构的设计技术，或是一个在报纸、杂志上流行的时尚词。

事实上，元编程远非一个抽象概念或一个宣传名词，它其实是一种脚踏实地的、务实的编程技术。它不只是听起来酷，它确实酷。在 Ruby 中，可以用它完成如下一些工作。

- 编写一个 Ruby 程序来连接外部系统——也许是一个 Web 服务或一个 Java 程序。使用元编程，可以编写一个包装器用来接受任何方法调用，然后把这些调用转发给这个外部系统。如果某个人后来为这个外部系统添加了方法，则无须修改这个 Ruby 的包装器，它会立刻自动支持这些新加入的方法。这很神奇吧！
- 当遇到一个最好是使用某种特定的语言来解决的问题，但如果自己去定义这种语言、自己去定制解析器及其他种种工作会遇到不小的麻烦时，你可以仅仅使用 Ruby 语言，把它改造成像专门处理这个问题的专属语言。你甚至还可以写出一个微型解释器从文件中读取这个基于 Ruby 语言的代码。
- 可以把Ruby程序的重复性降到一个Java程序员都不敢想的程度。比如有一个包含 20 个方法的类，这些方法看起来都差不多。如果这些方法只用几行代码一块定义会怎么样？或者你想调用一长串名字相似的方法，如果能用一小行代码就能调用这些名字具有一定模式的方法（比如所有的方法名都以 test 打头），那么你喜不喜欢呢？

- 可以改造 Ruby 使之满足你的需要，而不是去适应语言本身。例如，可以用你喜欢的方式去增强任何一个类（即使是像 Array 这样的核心类）；可以把日志功能封装到你所要监控方法的起止处；当客户继承你关心的类时，可以执行一段定制代码……这个列表还可以继续列下去，远远超出你的想象力。

元编程赋予了你这些能力。下面让我们一起来看看它是个什么样的。

“元”这个字眼
The “M” Word

你很可能期望从定义开始理解元编程。下面是元编程的定义。

元编程是写出编写代码的代码。

下面会给出一个更准确的定义，但是目前先使用这个。“编写代码的代码”究竟是什么意思？它到底在日常编程中有什么用处？在回答这些问题之前，先看看编程语言本身。

鬼城与自由市场
Ghost Towns and Marketplaces

如果把代码看成是一个世界，则它充斥着各种生机勃勃的市民：变量、类、方法等。如果说得更技术性一些，则可以把这些市民称为**语言构件**（*language construct*）。

在很多编程语言中，语言构件的行为更像是幽灵，而不是有血有肉的人：你可以在源代码中看到它们，然而在程序运行前它们就消失了。以 C++为例，一旦编译器完成了它的工作，像变量和方法这样的东西就看不见摸不着了：它们只是内存的位置而已。你不能向类询问它的实例方法，因为在问这个问题时，这个类已经淡出视野了。像 C++这样的语言，运行时是一个可怕的寂静之所——鬼城。

在其他的语言中（比如 Ruby），运行时看起来更像是一个繁忙的自由市场，绝大多数的语言构件依然存在，还在到处忙碌着。你甚至可以走到一个构件面前并且询问关于它自己的问题。这称为**内省**（*introspection*）。下面通过一个实例来看看什么是内省。

代码生成器与编译器

在元编程中，你可以写出编写代码的代码。这不是代码生成器与编译器所做的工作么？比如，你可以写出带注解的 Java 代码，然后使用代码生成器输出 XML 配置文件。从广义上说，这个 XML 生成过程也是元编程的一个例子。事实上，很多人听到“元”这个字眼时，首先会想到代码生成。

这种方式的元编程意味着你会先使用一个程序来创建或处理另外一个独立的程序，然后运行后面这个程序。在运行完代码生成器之后，最终运行之前，你可以阅读这些生成的代码（如果想检验自己的忍耐力），甚至手工修改这些代码。这也是 C++模板背后的原理：C++编译器会在编译之前把模板转换为一个普通的 C++代码，然后再运行那个编译好的程序。

在本书中，笔者会专注于元编程的另外一种含义，那种在运行时操作自身的代码。只有几种语言可以有效地做到这点，Ruby 是其中之一。你可以把这种方式称为**动态元编程**(*dynamic metaprogramming*)，以区别那种代码生成器和编译器方式的**静态元编程**（*static metaprogramming*）。

内省

看看下面的代码：

introduction/introspection.rb
```
class Greeting
  def initialize(text)
    @text = text
  end

  def welcome
    @text
  end
end

my_object = Greeting.new("Hello")
```

这里定义了一个 `Greeting` 类并创建了一个 `Greeting` 对象。现在可以来到语言构件前向它们提问。

```
my_object.class                                  # => Greeting
my_object.class.instance_methods(false)          # => [:welcome]
my_object.instance_variables                     # => [:@text]
```

可以向 my_object 对象询问它所属的类，它会准确无误地回答："我是一个 Greeting。"接着可以向这个类要它的实例方法。（这里的 false 参数表明，"我只是要你自己的方法，而不要那些继承来的方法。"）这个类回答了一个数组，其中只有一个方法名：welcome()。还可以查看这个对象本身，询问它的实例变量。这个对象再一次响亮而清晰地回答了问题。由于类和对象都是 Ruby 世界的一等公民，所以可以询问出很多信息。

然而，这只是事情的一半。你可以在运行时读取语言构件。能写出它们么？在程序运行的时候能在 welcome()方法之外再添加一个实例方法么？你也许在想，究竟是什么人会有这样的需求呢？请允许笔者用一个故事来解释这个问题。

元编程程序员 Bob 的故事
The Story of Bob, Metaprogrammer

Bob 是一个 Java 程序员，他刚刚开始学习 Ruby，他有一个宏伟的计划：他要为电影迷开发一个世界上最大的因特网社交网络系统。要实现这个计划，他需要一个包含电影和影评的数据库。Bob 希望用它来练习编写可重用的代码，因此他决定创建一个简单的库来把对象持久化到数据库中。

Bob 的第一次尝试

Bob 的库把数据库中的每个表映射到一个类中，而每条记录则映射到一个对象中。当 Bob 创建一个对象或访问它的属性时，这个对象会产生一条 SQL 语句并发送给数据库。所有这些都包装在一个基类中，代码如下：

introduction/orm.rb

```
class Entity
  attr_reader :table, :ident

  def initialize(table, ident)
    @table = table
    @ident = ident
    Database.sql "INSERT INTO #{@table} (id) VALUES (#{@ident})"
  end

  def set(col, val)
    Database.sql "UPDATE #{@table} SET #{col}='#{val}' WHERE id=#{@ident}"
  end
```

```
  def get(col)
    Database.sql("SELECT #{col} FROM #{@table} WHERE id=#{@ident}")[0][0]
  end
end
```

在 Bob 的数据库里，每个表都有一个 id 字段。每个 Entity 会保存这个字段的内容及它引用的表名。当 Bob 创建一个 Entity 对象时，这个对象会把它自身保存在数据库中。Entity#set() 方法会创建 SQL 语句来更新一个字段的值，而 Entity#get() 方法会创建 SQL 语句读取一个字段的值。（万一你感兴趣，Bob 的 Database 类用数组的数组作为返回的数据集。）

Bob 可以继承 Entity 类来映射一个特定的类。例如，Movie 类可以用来映射一个名为 movies 的表：

```
class Movie < Entity
  def initialize(ident)
    super("movies", ident)
  end

  def title
    get("title")
  end

  def title=(value)
    set("title", value)
  end

  def director
    get("director")
  end

  def director=(value)
    set("director", value)
  end
end
```

Movie 类对每个字段有两个方法：一个像 Movie#title() 这样的读方法和一个像 Movie#title=() 这样的写方法。现在，Bob 可以通过 Ruby 命令行解释器输入如下命令，把一部新电影加载到数据库中：

```
movie = Movie.new(1)
movie.title = "Doctor Strangelove"
movie.director = "Stanley Kubrick"
```

上面的代码在 movies 表中创建了一个新记录，它的 id、title 和 director 字段的值分别是 1、Doctor Strangelove 和 Stanley Kubrick。[1]

Bob 为他的工作感到自豪，并把代码交给更有经验的老程序员 Bill 看。Bill 很快就让 Bob 的自豪感烟消云散。“重复代码太多了，”Bob 说道。“数据库中有一个带有 title 字段的 movies 表，代码中有一个带有 @title 成员的 Movie 类，并且还有一个 title() 方法，一个 title=() 方法，以及两个 "title" 字符串常量。其实，可以使用一点元编程的魔法，用更少的代码解决这个问题。”

进入元编程的世界

在专家的建议下，Bob 寻求一个元编程的解决方案。他发现正好有一个名为 ActiveRecord 的类库，这是一个流行的 Ruby 类库，用于将对象映射到数据表中。[2]看完简短的教程后，Bob 写出了一个 ActiveRecord 版本的 Movie 类：

```
class Movie <ActiveRecord::Base
end
```

是的，就是这么简单。Bob 只是从 ActiveRecord::Base 类继承了一个子类，无需指定哪个表用来映射 Movie 对象。更妙的是，不用写诸如 title() 和 director() 这些看起来都差不多的方法。一切运转都正常：

```
movie = Movie.create
movie.title = "Doctor Strangelove"
movie.title                              # => "Doctor Strangelove"
```

上面的代码首先创建了一个 Movie 对象，并包装了 movies 表中的一条记录。可通过 Movie#title() 和 Movie#title=() 方法访问 title 字段，但是这些方法在源代码中无迹可查。如果它们根本没有定义过，title() 和 title=() 怎么能存在呢？可以从 ActiveRecord 的工作方式中找到答案。

表名部分很直接：ActiveRecord 通过内省机制查看类的名字，然后根据某

[1] 你很可能已经知道这些，不过再听听也无妨：在 Ruby 中，movie.title = "Doctor Strangelove" 实际上是 title=() 方法的变相调用方式，它等同于 movie.title=("Doctor Strangelove")。

[2] ActiveRecord 是 Rails 的一部分，Rails 是最卓越的 Ruby 框架。你会在第 7 章 “ActiveRecord 的设计”（第 171 页）中读到更多关于 Rails 和 ActiveRecord 的内容。

种简单的习惯即可得到表名。因为类名是 Movie，所以 ActiveRecord 会把它映射到名为 movies 的表中。（这个库知道怎么处理英语单词的复数。）

那些像 `title()` 和 `title=()` 这样访问属性的方法（简称为访问器）是怎样处理的呢？这就是元编程发挥作用的地方：Bob 无须编写那些方法。从表的模式中得到字段名后，ActiveRecord 会自动定义这些方法。`ActiveRecord::Base` 会在运行时读取数据库模式，如果发现 `movies` 表有两个名为 `title` 和 `director` 的字段，就通过定义访问器创建两个同名属性。这意味着 ActiveRecord 在程序运行时无中生有地创建了诸如 `Movie#title()` 和 `Movie#director=()` 这样的方法！[3]

相对于内省的“阴”，这就是它的“阳”：不仅可以读出语言构件，还可以写入它们。如果认为这个特性非常有用，那么你是对的。

再谈“元”

现在，你可以得到一个更加正式的元编程定义：

元编程是编写在运行时操纵语言构件的代码。

ActiveRecord 的作者是怎样应用这个概念的？他不是为每个类的属性编写访问器方法，而是编写代码为每个继承自 `ActiveRecord::Base` 的类在运行时定义方法。这就是“编写代码的代码”时所指的东西。

你也许会认为这只是一个孤例，但是，如果看看 Ruby（正如马上会做的），你会发现它们无处不在。

元编程和 Ruby
Metaprogamming and Ruby

还记得早先讨论的鬼城和自由市场么？如果你希望“操纵语言构件”，那么那些构件必须在运行时存在。下面快速看看几种语言在运行时能给你多少控制权。

[3] ActiveRecord 对于访问器的实际实现比这里描述的更微妙一些，你会在第 8 章“深入 ActiveRecord”（第 187 页）中看到更详细的说明。

一个用C语言写的程序会跨越两个不同的世界：编译时和运行时。在编译时，可以拥有像变量和函数这样的语言构件；而在运行时，只有一大堆机器码。由于绝大多数编译时的信息在运行时都丢失了，所以C语言不支持元编程或内省。在C++语言中，一些语言构件可以在编译后生存下来，这也是为什么你可以向C++对象询问它的类的原因。在Java语言中，编译时和运行时的界线甚至更加模糊，你有足够的内省能力来列出一个类的方法，或者一直向上查询其超类链。

Ruby 无疑是现今流行语言中对元编程最友好的一种语言。没有编译时，Ruby程序中所有的语言构件在运行时可用。在运行一个程序时，无须翻过一道横亘在所写程序与所运行程序中间的墙。这里只有一个世界。

在这个世界中，元编程无处不在。事实上，元编程如此深入 Ruby 语言，甚至无法和“普通”编程明确区分开来。你无法看着一段 Ruby 代码说，“这部分是元编程，而其他部分不是。”从某种程度上说，元编程不过是每个 Ruby 程序员的例行工作。

为明确起见，元编程并不是 Ruby 高手的抽象艺术，它也不只是创建像ActiveRecord 这种复杂东西的利器。如果你要通向 Ruby 的高级编程之路，就会在每一步上看到元编程的身影。即使满足于目前你所学的 Ruby 知识，也很可能会在编程之旅中被元编程绊倒：可能是在某个流行框架的源代码中，也可能是在某个喜欢的类库中，甚至是从某个博客上看到的小例子。除非掌握了元编程，否则你不会拥有 Ruby 语言全部的战斗力。

学习元编程，还有一种也许不那么明显的原因。尽管 Ruby 第一眼看上去很简单，但很快会被它的精妙细微所打击。迟早你会问自己一些问题，比如“一个对象可以调用同属一个类的其他对象的私有方法么？”或者“可以通过导入一个模块来创建类方法么？”最终，这些看起来复杂的行为实际上都是基

于非常简单的原则。通过元编程，你会对这门语言更熟悉；通过学习这些规则，你会找到这些问题的答案。

如果你已经知道元编程是干什么的了，就表明已经做好阅读本书正文的准备了。

关于本书
About This Book

第 1 部分“Ruby 元编程”是本书的核心。它讲述了你和 Bill（一个资深 Ruby 程序员）在办公室一周中发生的故事。

- Ruby 的对象模型是元编程的土壤。第 1 章“星期一：对象模型”（第 3 页），给你提供了这块土地的地图。本章主要介绍了最基本的元编程技巧，揭示了 Ruby 类和方法查找背后的秘密，方法查找就是 Ruby 查找并执行方法的过程。
- 一旦理解了方法查找，就可以用方法做一些很炫的事情：可以在运行时创建方法、插入方法调用、把调用转发给其他对象，甚至调用一个不存在的方法。所有这些技术都在第 2 章“星期二：方法”（第 37 页）中得以说明。
- 方法是大家族中的一员，这个家族中还有像块和 lambda 这样的成员。第 3 章“星期三：代码块”（第 69 页）中会介绍这些成员的方方面面。该章还使用了一个例子来演示领域专属语言，这是在开发团体中日渐流行的一种概念。另外，该章还给出了一些专门技巧，用于说明怎样打包一段代码并在将来执行它，以及怎样携带变量穿越作用域。
- 说到作用域，Ruby 有一种值得深入探究的特殊作用域：类定义的作用域。第 4 章“星期四：类定义”（第 101 页）中讨论了作用域，并且介绍了元编程兵工厂中一些最具杀伤力的武器。该章还介绍了 eigenclass（也称为单件类），这是在 Ruby 中最后一个让人困惑且需要弄明白的概念。
- 第 5 章“星期五：编写代码的代码”（第 139 页）把前面各章中的各种技术融入到一个扩展的例子中。本章还为元编程知识库提供了两条新知识：一个有些争议性的 `eval()` 方法和一个可以拦截对象模型事件中的回调方法。

本书的第 2 部分“Rails 中的元编程”是一个元编程的实例。Rails 是 Ruby 标志性的框架。它包含 3 个较短的章节，每个章节关注 Rails 不同的领域。通过查看 Rails 的源代码，你会看到 Ruby 大师是怎样使用元编程编出伟大的软件的。

在你着手阅读本书之前，需要了解本书的 3 个附录。附录 A（第 223 页）描述了一些常见的技巧，尽管严格来说它们不是元编程的技巧，但它们很有用。附录 B（第 235 页）会带你看看领域专属语言。附录 C（第 239 页）是本书中所有法术的一个快速索引，并附上了示例代码。

“等等，”听到了你在说什么“**法术**（*spell*）？”哦，是的，对不起。下面做些解释。

法术
Spells

本书包含了不少元编程的技术，可以把它们用于你自己的编程中。一些人会把它们叫做**模式**（*pattern*）或**惯用法**（*idiom*）。这两种术语在 Ruby 主义者中都很不流行，因此把它们叫做了法术。尽管它们实际上没什么神奇的，但是它们确实在 Ruby 初学者看来有神奇的法术。

本书中到处都有法术的引用。笔者会通过像**白板**（第 61 页）或**代码字符串**（第 142 页）这样的方式来表示它们。括号中的数字表示那个法术第一次出现的页码。如果需要一个法术的快速参考，则可在附录 C（第 239 页）中找到完整的法术书。

小测验
Quizzes

本书不时会抛出一些小测验。你可以跳过这些测验而直接阅读解决方案，但是你也可能希望解决它们，因为它们挺有趣。

有些测验就是传统的编码练习，有些则要求你离开键盘进行一些思考。所有的小测验都包含一个解决方案，但是绝大多数的测验有不止一个方案。尽情地发挥吧！