Broadview
www.broadview.com.cn
PEARSON

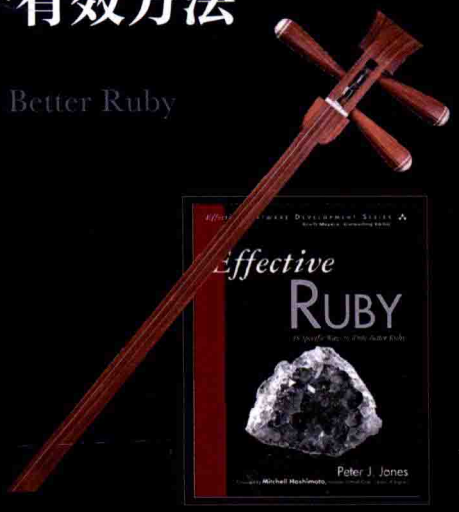
·原味精品书系·

Effective Ruby（英文版）

编写高质量Ruby代码的48个有效方法

Effective Ruby: 48 Specific Ways to Write Better Ruby

[美] Peter J. Jones 著



中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

· 原味精品书系 ·

Effective Ruby（英文版）

编写高质量Ruby代码的48个有效方法

Effective Ruby: 48 Specific Ways to Write Better Ruby

[美] Peter J. Jones 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书是作者 Peter J. Jones 近十年 Ruby 开发经验的结晶。书中为 Ruby 开发的每个主要领域提供了切实可行的建议，从模块到内存，再到元编程。作者利用 48 个 Ruby 实战案例，揭示了 Ruby 鲜有人知的风格、特色、缺陷，以及对代码行为和执行极具影响的复杂性。每种实践方法都包含了具体的、实用的、组织清晰的指导方针，细致的建议，详细的专业理由，以及详尽的示例代码讲解。

本书旨在通过全面地介绍 Ruby 编程技术，帮助 Ruby 程序员及爱好者写出更健壮、更高效、更易维护和运行的代码。

版权贸易合同登记号　图字：01-2015-6092

图书在版编目（CIP）数据

Effective Ruby：编写高质量 Ruby 代码的 48 个有效方法 =Effective Ruby: 48 Specific Ways to Write Better Ruby，1E：英文 /（美）琼斯（Jones,P.J.）著. — 北京：电子工业出版社，2016.4
（原味精品书系）
ISBN 978-7-121-27306-3
I. ① E…II. ① 琼…III. ① 计算机网络－程序设计－英文 IV. ① TP393.09
中国版本图书馆 CIP 数据核字（2015）第 231525 号

策划编辑：张春雨　刘　芸
责任编辑：徐津平
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980 1/16　　印张：14.25　　字数：275 千字
版　　次：2016 年 4 月第 1 版
印　　次：2016 年 4 月第 1 次印刷
定　　价：65.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。
质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。
服务热线：（010）88258888。

序言

当受邀对一本关于 Ruby 的书进行技术审核并为其作序时，我有点不知所措。市面上已经有不少 Ruby 相关的书籍，从入门到深入 Ruby VM 内部实现机制已经面面俱到。我当时想，“这本 Ruby 书能与众不同吗？”但是我同意先通读一遍。令我吃惊的是，这的确是一本出色并且新颖的书籍。它和其他 Ruby 书籍很不一样，我觉得不管是谁，初学者亦或专家，只要将其读完，都会成为更加出色的 Ruby 编程人员。

从我开始接触 Ruby 到现在已经十几年了。这些年里，Ruby 已经成为一门成熟的语言。早期，它也经历过过度宣传的阶段，当时 Ruby 被吹捧为结束一切、造就一切的语言。随后涌现了很多版本的 Ruby 库，当时感觉库几乎每天都会被废弃然后重建，没有一个库可以保证是最新的。后来，其他“新一代”语言开始涌现，Ruby 被定位为过时的语言。但是现在，最终，Ruby 通过解决很多实际问题，证明了自己是一门实用且有效的语言，虽然它并不能解决所有问题。（你不会想用 Ruby 编写另外一个大型操作系统。）

本书除了介绍基本语法和高级使用方法外，还重点介绍了上佳实践，讲述如何编写不会崩溃、可维护、高效快速的 Ruby 应用程序。Ruby 初学者通过学习上佳实践能够更好地理解语言，而有经验的开发人员可以重新审视他们的实践经验，并能够学习一些新的技巧。

我很喜欢本书的行文方式：大量示例，并且示例不仅解释“是什么”和“如何做”，而且解释了“为什么”。虽然这些都是 Ruby 社区多年发展积累下来的上佳实践，但是保持怀疑、提出问题仍然很重要，旧实践上的改进很可能产出新的上佳实践。

希望你能从本书中收获良多，也真心希望在阅读了本书几百页的内容之后，你能成长为一名 Ruby 程序员。

Mitchell Hashimoto

HashiCorp 的创始人和 CEO，Vagrant 的作者

前言

学习一门新的编程语言通常需要经历两个阶段。第一阶段是花费时间学习语义和语言的结构。如果之前有学习新编程语言的经验，那么这一阶段通常很短。以 Ruby 为例，它的语法和其他面向对象语言非常相似。语言的结构——如何基于语法构建出程序——对于有经验的程序员而言也十分类似。

另一方面，第二阶段可能需要花费更多的时间。这一阶段需要深入语言，学习其常用模式。大部分语言在解决通用问题上都有独特的方式，Ruby 也是这样。比如，Ruby 使用 block 和 iterator 模式来替代显式循环。学习如何使用“Ruby 的方式”来解决问题，并同时避免严重错误，是这一阶段的重点。

这也是本书要解决的问题。但是本书不是一本介绍性图书。书中假定读者已经完成了 Ruby 学习的第一阶段——已经学会了其语法和结构。本书的目标是让读者学习到 Ruby 语言的精髓，以及如何编写更为可靠且易于维护的高效代码。同时，本书也会介绍 Ruby 解释器内部的工作原理，了解这些知识有助于编写出更加高效的程序。

Ruby 的实现和版本

众所周知，Ruby 社区有很多积极的贡献者。他们在各种各样的项目里工作，包括 Ruby 解释器的不同实现。除了官方的 Ruby 解释器（即 MRI），还有另外一些解释器可以选择。如果你需要将 Ruby 应用程序部署到已经配置好用来运行 Java 应用程序的生产服务器上，别担心，这正是 JRuby 所解决的问题。其他领域情况如何？ Ruby 应用程序能否放到智能手机和平板设备里？同样也有相应的 Ruby 实现。

Ruby 实现有多种选择，这是 Ruby 活跃并且健康的标志。当然，这些实现都有独特的内在方式。但是从编写 Ruby 代码的程序员的角度来看，这些不同的解释器的行为和 MRI 都非常类似，无须担心它们之间的差异。

本书的大部分内容适用于所有这些各异的 Ruby 实现。唯一需要注意的是 Ruby 内部细节介绍，比如垃圾回收工作机制。在这些领域，本书介绍基于官方 Ruby 实现——MRI。当书中提到某个特定 Ruby 版本的时候，你就会知道我们在讨论特定于 MRI 相关的事情。

关于特定版本，本书的所有代码都支持 Ruby 1.9.3 及更高版本。撰写本书时，Ruby 2.1 是最新版本，Ruby 2.2 正在开发中。如果本书内容没有特别提及某个版本，那么样例代码在所有支持的版本上都能工作。

风格介绍

Ruby 程序员大部分时候使用相同的方式格式化 Ruby 代码。甚至有一些 Ruby-Gem 可以帮助检查代码，在格式不满足预设定的风格规则时给出提示。特别提到这一点是因为本书示例代码所选的风格和大家通常选择的风格有些不同。

当调用某个方法并传递参数时，书中使用圆括号括住参数，在左圆括号和方法之间没有空格。实际使用中，经常能看到调用方法时并没有使用圆括号，这是因为 Ruby 并不强制要求使用圆括号。但是正如本书第 1 章所述，在一些情况下忽略圆括号会导致代码意思模糊，反而会要求 Ruby 来猜测你的真实意图。因为这些可能造成歧义，所以我认为忽略圆括号的习惯很不好，需要在社区里呼吁大家注意这个问题。

使用圆括号的另一个原因是能够在标识符是方法调用(而不是关键字)时清楚地表明之。有时你可能会把是方法调用的标识符误认为是关键字（比如 require），使用圆括号能够帮助辨明这样的情况。

既然这一节在讨论风格问题，必须说明的一点是，在本书中提到方法时，会使用 RI notation。如果你还不熟悉 RI notation，可以自行了解，它易学且非常有用。其最大的作用是区分类和实例方法。当涉及类的方法时，本书会用两个冒号（::）分隔类名和方法。比如，File::open 表示 open 类方法来自于 File 类。类似地，实例方法用井号（#）分隔类名和实例方法名（比如 Array#each）。同样的风格也应用于模块方法（比如 GC::stat）和模块实例方法（比如 Enumerable#grep）。第 40 项详细介绍了 RI notation，以及如何使用它来查找方法文档。了解了本节上述的基础介绍之后就完全可以开始阅读本书了。

如何获得源代码

本书会介绍很多样例源代码。为了更容易理解吸收，代码通常会被分割成小段，每次分析其中一段。也有些时候会跳过一些不重要的代码。有时也需要查看所有代码来实现整体理解，因此，所有本书展示的代码都能在 http://effectiveruby.com 里找到。

致谢

能够写出大家愿意花时间阅读且愿意花钱购买的书籍不是我一个人的功劳。实际上，除了为本书直接做出贡献的人，还有很多人以这样那样的方式为本书默默奉献。比如，我的朋友 Michael Garriss 不会意识到，正是他鼓励我学习 Ruby，才有了本书的问世。当年他肯定不会想到我会将他从一个公司拽到另一个公司，去介绍 Ruby 的每一个细节。然而，这的确发生了。

可能这样做有些奇怪，但是我还是想借此机会感谢曾经为开源社区贡献时间和创造力的所有人。撰写本书时所使用的每一个工具，包括那些我专门为此创建的工具，都是开源项目。如果我无法查看 Ruby 解释器和本书所讨论的 gem 的源码，就不可能完成本书。我花费了很多时间研究代码，仔细分析，做实验，过程中也曾哭泣。最终，事实证明这所有的付出都是值得的。

当然，如果没有那些慷慨的自愿和我一起工作的人们，这本书不会这么有价值。一些人放弃了他们的空闲时间来帮助我审查本书的初稿，并且给了我很多非常有用的反馈。Isaac Foraker、Timothy Clayton，以及我的妻子 Shanna Jones，花了大量时间阅读本书并且验证本书代码的正确性。非常感谢你们的帮助。

Bruce Williams 和 Bobby Wilson 担任了本书的技术审稿人，可能一开始他们没有意识到将要付出多少精力。他们帮助我改进了本书的样例及其解释。当我由于别人干涉我的工作而过分焦虑时，也是他们鼓励了我。

Pearson 的所有工作人员都为本书竭尽全力。Trina MacDonald、Olivia Basegio 和 Songlin Qiu 都给予了我非常耐心的帮助，最终将本书塑造成了现在的模样。在这个项目里我收获良多，很大一部分成果应该归功于他们。

Scott Meyers 是我的偶像，和他一起工作是我梦寐以求的事。在 20 世纪 90 年代末，我阅读了 Scott 的 *Effective C++*，这本书改变了我的编程方式。它也启发了我如何将知识有效地教给别人。将我的成果交给 Scott 审核时我非常忐忑，但是 Scott 给了我无尽的鼓励和帮助。谢谢你，Scott。

我的妻子，Shanna Jones，一直给予我无私的鼓励和理解。明知写作此书会占用很多我陪伴她的时间，她仍然督促我写完这本书。Shanna，你教会了我很多东西。谢谢你一直以来的支持。

关于作者

Peter J. Jones 从 2005 年就开始使用 Ruby。他在偶然发现了一台 Commodore 64 之后就开始了编程之旅，那时他还不会正确使用键盘，而是使用一些代码列表和卡式磁带。Peter 现在是一名自由职业软件工程师，也是 Devalot.com 编程相关的 workshop 资深讲师。

目录

1 Accustoming Yourself to Ruby

With each programming language you learn, it's important to dig in and discover its idiosyncrasies. Ruby is no different. While it borrows heavily from the languages that preceded it, Ruby certainly has its own way of doing things. And sometimes those ways will surprise you.

We begin our journey through Ruby's many features by examining its unique take on common programming ideas. That is, those that impact every part of your program. With these items mastered, you'll be prepared to tackle the chapters that follow.

Item 1: Understand What Ruby Considers to Be True

Every programming language seems to have its own way of dealing with Boolean values. Some languages only have a single representation of true or false. Others have a confusing blend of types that are sometimes true and sometimes false. Failure to understand which values are true and which are false can lead to bugs in conditional expressions. For example, how many languages do you know where the number zero is false? What about those where zero is true?

Ruby has its own way of doing things, Boolean values included. Thankfully, the rule for figuring out if a value is true or false is pretty simple. It's different than other languages, which is the whole reason this item exists, so make sure you understand what follows. In Ruby, every value is true *except* `false` and `nil`.

It's worth taking a moment and thinking about what this means. While it's a simple rule, it has some strange consequences when compared with other mainstream languages. In a lot of programming languages the number zero is false, with all other numbers being true. Using the rule just given for Ruby, zero is *true*. That's probably one of the biggest gotchas for programmers coming to Ruby from other languages.

Another trick that Ruby plays on you if you're coming from another language is the assumption that `true` and `false` are keywords. They're not. In fact, they're best described as global variables that don't follow the naming and assignment rules. What I mean by this is that they don't begin with a "$" character, like most global variables, and they can't be used as the left-hand side of an assignment. But in all other regards they're global variables. See for yourself:

```
irb> true.class
---> TrueClass

irb> false.class
---> FalseClass
```

As you can see, `true` and `false` act like global objects, and like any object, you can call methods on them. (Ruby also defines `TRUE` and `FALSE` constants that reference these `true` and `false` objects.) They also come from two different classes: `TrueClass` and `FalseClass`. Neither of these classes allows you to create new objects from them; `true` and `false` are all we get. Knowing the rule Ruby uses for conditional expressions, you can see that the `true` object only exists for convenience. Since `false` and `nil` are the only false values, the `true` object is superfluous for representing a true value. Any non-`false`, non-`nil` object can do that for you.

Having two values to represent false and all others to represent true can sometimes get in your way. One common example is when you need to differentiate between `false` and `nil`. This comes up all the time in objects that represent configuration information. In those objects, a `false` value means that something should be disabled, while a `nil` value means an option wasn't explicitly specified and the default value should be used instead. The easiest way to tell them apart is by using the `nil?` method, which is described further in Item 2. Another way is by using the "==" operator with `false` used as the left operand:

```
if false == x
  ...
end
```

With some languages there's a stylistic rule that says you should always use immutable constants as the left-hand side of an equality operator. That's not why I'm recommending `false` as the left operand to the "==" operator. In this case, it's important for a functional reason. Placing `false` on the left-hand side means that Ruby parses the expression as a call to the `FalseClass#==` method (which comes from the `Object` class). We can rest safely knowing this method only returns `true` if the right operand is also the `false` object. On the other

hand, using false as the *right* operand may not work as expected since other classes can override the Object#== method and loosen the comparison:

```
irb> class Bad
       def == (other)
         true
       end
     end

irb> false == Bad.new
---> false
irb> Bad.new == false
---> true
```

Of course, something like this would be pretty silly. But in my experience, that means it's more likely to happen. (By the way, we'll cover the "==" operator more in Item 12.)

Things to Remember

- Every value is true *except* false and nil.
- Unlike in a lot of languages, the number zero is true in Ruby.
- If you need to differentiate between false and nil, either use the nil? method or use the "==" operator with false as the left operand.

Item 2: Treat All Objects as If They Could Be nil

Every object in a running Ruby program comes from a class that, in one way or another, inherits from the BasicObject class. Imagining how all these objects relate to one another should conjure up the familiar tree diagram with BasicObject at the root. What this means in practice is that an object of one class can be substituted for an object of another (thanks to polymorphism). That's why we can pass an object that *behaves* like an array—but is not actually an array—to a method that expects an Array object. Ruby programmers like to call this "duck typing." Instead of requiring that an object be an instance of a specific class, duck typing shifts the focus to what the object can do; in other words, interface over type. In Ruby terms, duck typing means you should prefer using the respond_to? method over the is_a? method.

But in reality, it's rare to see a method inspect its arguments using respond_to? to make sure it supports the correct interface. Instead, we tend to just invoke methods on an object and if the object doesn't respond to a particular method, we leave it up to Ruby to raise a

NoMethodError exception at run time. On the surface, it seems like this could be a real problem for Ruby programmers. Well, just between you and me, it is. It's one of the core reasons testing is so very important. There's nothing stopping you from accidentally passing a Time object to a method expecting a Date object. These are the kinds of mistakes we have to tease out with good tests. And thanks to testing, these types of problems can be avoided. But one of these polymorphic substitutions plagues even well-tested applications:

```
undefined method 'fubar' for nil:NilClass (NoMethodError)
```

This is what happens when you call a method on an object and it turns out to be that pesky nil object...the one and only object from the NilClass class. Errors like this tend to slip through testing only to show up in production when a user does something out of the ordinary. Another situation where this can occur is when a method returns nil and then that return value gets passed directly into another method as an argument. There's a surprisingly large number of ways nil can unexpectedly get introduced into your running program. The best defense is to assume that any object might actually be the nil object. This includes arguments passed to methods and return values from them.

One of the easiest ways to avoid invoking methods on the nil object is by using the nil? method. It returns true if the receiver is nil and false otherwise. Of course, nil objects are always false in a Boolean context, so the if and unless expressions work as expected. All of the following lines are equivalent to one another:

```
person.save if person
person.save if !person.nil?
person.save unless person.nil?
```

It's often easier to explicitly convert a variable into the expected type rather than worry about nil all the time. This is especially true when a method should produce a result even if some of its inputs are nil. The Object class defines several conversion methods that can come in handy in this case. For example, the to_s method converts the receiver into a string:

```
irb> 13.to_s
---> "13"

irb> nil.to_s
---> ""
```

As you can see, NilClass#to_s returns an empty string. What makes to_s really nice is that String#to_s simply returns self without performing any conversion or copying. If a variable is already a string then using to_s will have minimal overhead. But

if nil somehow winds up where a string is expected, to_s can save the day. As an example, suppose a method expects one of its arguments to be a string. Using to_s, you can hedge against that argument being nil:

```
def fix_title (title)
  title.to_s.capitalize
end
```

The fun doesn't stop there. As you'd expect, there's a matching conversion method for almost all of the built-in classes. Here are some of the more useful ones as they apply to nil:

```
irb> nil.to_a
---> []

irb> nil.to_i
---> 0

irb> nil.to_f
---> 0.0
```

When multiple values are being considered at the same time, you can make use of a neat trick from the Array class. The Array#compact method returns a copy of the receiver with all nil elements removed. It's common to use it for constructing a string out of a set of variables that might be nil. For example, if a person's name is made up of first, middle, and last components—any of which might be nil—you can construct a complete full name with the following code:

```
name = [first, middle, last].compact.join(" ")
```

The nil object has a tendency to sneak into your running programs when you least expect it. Whether it's from user input, an unconstrained database, or methods that return nil to signal failure, always assume that every variable could be nil.

Things to Remember

- Due to the way Ruby's type system works, any object can be nil.
- The nil? method returns true if its receiver is nil and false otherwise.
- When appropriate, use conversion methods such as to_s and to_i to coerce nil objects into the expected type.
- The Array#compact method returns a copy of the receiver with all nil elements removed.

Item 3: Avoid Ruby's Cryptic Perlisms

If you've ever used the Perl programming language then you undoubtedly recognize its influence on Ruby. The majority of Ruby's perlisms have been adopted in such a way that they blend perfectly with the rest of the ecosystem. But others either stick out like an unnecessary semicolon or are so obscure that they leave you scratching your head trying to figure out how a particular piece of code works.

Over the years, as Ruby matured, alternatives to some of the more cryptic perlisms were added. As more time went on, some of these holdovers from Perl were deprecated or even completely removed from Ruby. Yet, a few still remain, and you're likely to come across them in the wild. This item can be used as a guide to deciphering those perlisms while acting as a warning to avoid introducing them into your own code.

The corner of Ruby where you're most likely to encounter features borrowed from Perl is a set of cryptic global variables. In fact, Ruby has some pretty liberal naming rules when it comes to global variables. Unlike with local variables, instance variables, or even constants, you're allowed to use all sorts of characters as variable names. Recalling that global variables begin with a "$" character, consider this:

```
def extract_error (message)
  if message =~ /^ERROR:\s+(.+)$/
    $1
  else
    "no error"
  end
end
```

There are two perlisms packed into this code example. The first is the use of the "=~" operator from the `String` class. It returns the position within the string where the right operand (usually a regular expression) matches, or `nil` if no match can be found. When the regular expression matches, several global variables will be set so you can extract information from the string. In this example, I'm extracting the contents of the first capture group using the $1 global variable. And this is where things get a bit weird. That variable might look and smell like a global variable, but it surely doesn't act like one.

The variables created by the "=~" operator are called *special* global variables. That's because they're scoped *locally* to the current thread and method. Essentially, they're local values with global names. Outside of the `extract_error` method from the previous example, the $1 "global" variable is `nil`, even after using the "=~" operator. In the example,