EFFECTIVE 系列丛书

HZ BOOKS 华章IT

PEARSON

"Effective Software Development Series" 系列经典著作，Amazon全五星评价

从模块、内存到元编程，全面总结和探讨Ruby编程中48个鲜为人知和容易被忽视的特性与陷阱

包含大量实用范例代码，为编写易于理解、便于维护、易于扩展和高效的Ruby应用提供了解决方案

Effective Ruby

48 Specific Ways to Write Better Ruby

Effective Ruby

改善Ruby程序的48条建议

[美] 彼得 J. 琼斯(Peter J. Jones) 著

刘璐 杨政权 秦五一 孟繁超 译

机械工业出版社 China Machine Press

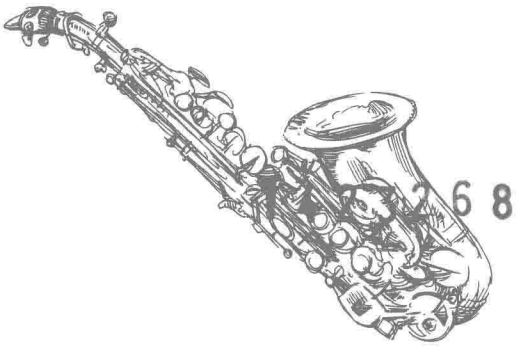
EFFECTIVE
系列丛书

Effective Ruby
48 Specific Ways to Write Better Ruby

Effective Ruby

改善Ruby程序的48条建议

[美] 彼得 J. 琼斯(Peter J. Jones) 著
刘璐 杨政权 秦五一 孟繁超 译



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Effective Ruby：改善 Ruby 程序的 48 条建议 /（美）琼斯（Jones，P. J.）著；刘璐等译 .
—北京：机械工业出版社，2015.11
（Effective 系列丛书）
书名原文：Effective Ruby：48 Specific Ways to Write Better Ruby

ISBN 978-7-111-52124-2

I. E… II. ①琼… ②刘… 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（2015）第 270562 号

本书版权登记号：图字：01-2015-1917

Effective Ruby：改善 Ruby 程序的 48 条建议

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：陈佳媛　　责任校对：殷　虹
印　　刷：中国电影出版社印刷厂　　版　　次：2016 年 1 月第 1 版第 1 次印刷
开　　本：186mm×240mm　1/16　　印　　张：12
书　　号：ISBN 978-7-111-52124-2　　定　　价：49.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

Preface 序

当我受邀为一本关于 Ruby 的书做技术审校并作序时，我曾有所怀疑。为什么呢？现有关于 Ruby 的书籍已覆盖了整个知识领域：从入门的初学者指南，到高阶的 Ruby 虚拟机内部实现。我想，“这本 Ruby 书籍又能写出什么不同呢？”不过随后我还是同意阅读一下。让我惊异的是，呈现在我眼前的是一本优秀且新颖的 Ruby 书籍。这本书完全不同于其他 Ruby 书籍，在数百页里涵盖了很多精彩内容，这些内容有助于任何读者（无论是初学者还是专家）进阶成为更好的 Ruby 程序员。

当我开始使用 Ruby 时，作为一门语言，它已经在过去十年里成熟了很多。早期，在宣传阶段，Ruby 被吹捧为所有语言的终结者和替代者。接着，大量的库出现了，它们似乎每天都在被抛弃和重写，没有哪个可以被信赖为最新版本。随后，其他新的流行语言开始出现，Ruby 经历了一个被视为过时语言的阶段。然而，如今，Ruby 终于被看作是一种可以解决多种问题的实用、高效的语言，当然，它并不能解决所有的问题，这也是可以理解的。（你当然不会试图用 Ruby 开发一个大型操作系统。）

相比于成为一本覆盖基本语法或进阶实践的书，这本书巧妙地涵盖了如何写出避免崩溃、易维护且运行快速的 Ruby 应用程序的最佳实践。它是一本任何 Ruby 程序员都该读的书。初学者应该学习书中的最佳实践从而更好地认识这门语言，有经验的开发者应该再次审读自己的程序，也许能学习一些新的方法。

这本书采用我最喜欢的方式撰写：辅以大量的例子。这些例子不仅在阐释“是什么”（what）和“怎样做”（how），还阐释了“为什么”（why）。尽管这些最佳实践源于多年来

日益成熟的 Ruby 社区中的精华内容，不过总是保持质疑的态度、提出疑问仍然是很重要的，也许以后能找到新的最佳实践来对原来的加以改进。

祝你阅读愉快，我完全预料到读完这数百页之后你会成长为一个更好的 Ruby 程序员。

——Mitchell Hashimoto

HashiCorp 创始人、CEO，Vagrant 创建者

Preface 前　言

学习一门新的编程语言通常需要经过两个阶段。第一阶段是学习这门编程语言的语法和结构。如果我们具有其他编程语言的经验，这个阶段通常只需要很短的时间。以Ruby为例，接触过其他面向对象语言的程序员对Ruby的语法也会比较熟悉。有经验的程序员对于语言的结构（如何根据语法构建应用程序）是很熟悉的。

在第二阶段则需要更多的努力。这个阶段也是你深入语言、学习语言风格的时候，许多编程语言在解决常见的问题时都使用了独一无二的方法，Ruby也不例外。比如循环，Ruby没有使用显式循环体来实现，而是使用了代码块和迭代器模式。学习如何突破思维定势，用Ruby的方式解决问题是这个阶段的精髓。

本书也将致力于编程语言学习的两个阶段。但本书并不是一本编程语言的入门级书籍。如果你已经完成了第一阶段，即语法和结构的学习，通过本书你将会对Ruby的理解更加深入和全面，编写出更具可读性、可维护性的代码。在这个过程中，我也会介绍Ruby解释器的内部工作原理，并分享编写更高效程序的相关知识。

Ruby的不同实现和不同版本

如你所知，Ruby有一批非常活跃的社区贡献者。他们负责各种各样的项目，其中也包括Ruby解释器的不同实现。除了大家熟知的Ruby官方实现（MRI），我们也有很多其他的选择。想把Ruby应用程序部署到已配置Java环境的机器？没问题，这就是JRuby的用武之地。那Ruby应用程序是否支持智能手机和平板电脑呢？当然，同样也有一款相应的实现。

可以选择的 Ruby 实现有多种，这也证明了 Ruby 的活跃性。虽然每种的内部实现都不一样，但对 Ruby 程序员来说无需太过担心，因为这些解释器的行为和官方的 MRI 很接近。

本书介绍的很多方法在不同 Ruby 实现上都是通用的，但也有一些只针对 MRI 版本，诸如书中提及的垃圾回收机制。本书中指定 Ruby 特定版本的时候，都是针对 MRI 版本的。

书中所有代码适用于 Ruby 1.9.3 及其后的版本。在本书编写时，Ruby 最新版本是 2.1，Ruby 2.2 即将面世。若书中未提及特定的 Ruby 版本，则示例代码可在所有支持的版本上运行。

关于代码风格的注释

在多数情况下，Ruby 程序员喜欢使用单一的风格来格式化 Ruby 代码。当代码没有按照预定义的风格规则编写时，一些 RubyGem 甚至能够检测出这些不一致的代码风格并斥责你。之所以提到这件事，是因为本书中示例代码所采用的风格可能与通常情况下使用的略有不同（也可能与你之前的风格不同）。

当调用一个方法（method）并向它传参时，我使用圆括号把参数括起来，而紧跟在方法名之后，中间没有空格。在本书之外，我们很容易看到没有圆括号的方法调用，这可能是由于 Ruby 传参时并不需要圆括号。但是，在第 1 章中我们会看到，不使用圆括号传参在某些情况下，会导致代码多义性，这迫使 Ruby 猜测你的意图。由于存在这种多义性，我认为省略掉圆括号是一种不好的编码习惯，我们需要摒弃这种习惯。

我使用圆括号的另一个原因，是为了清晰地表明一个标识符是方法调用还是关键字。你可能会惊讶地发现，你认为的可能是关键字的东西实际上是个方法调用（例如 require）。而使用圆括号有助于说明这一点。

在这里讨论代码风格时，我应该指出，当本书中提及方法时，我会使用 RI 标记法。如果你不熟悉 RI 标记法也没有关系，因为它很容易学习并很有帮助。它的主要目的是区分类方法（class method）与实例方法（instance method）。当写类方法时，我会用双冒号（“::”）放在类名与方法名之间。例如，File::open 是 File 类中名为 open 的类

方法。同样，当写实例方法时，我会用井号（“#”）放在类名与实例方法名之间（例如Array#each）。上述写法同样适用于模块方法（module method）名（例如GC::stat）与模块实例方法（module instance method）名（例如Enumerable#grep）。本书的第40条建议会涉及RI标记法与使用它查询方法文档的更多细节。不过，上述这点入门足以让你开始本书的旅途了。

下载源代码

本书会深入探讨很多示例代码。为了更容易理解和消化，代码通常将会分割成小的代码片段，我们会一次一小段地进行讲解。有些不重要的细节代码，我们甚至会忽略掉。可以说，有时候一次看完所有的代码对形成宏观理解是很不错的。本书中所有的代码，可以通过本书的网站下载，网址为http://effectiveruby.com。

目　录 *Contents*

第 1 章 Chapter 1

让自己熟悉 Ruby

像你所学的每门程序设计语言一样，深入发掘它的特性是很重要的，从这点上来说 Ruby 也没有什么不同。Ruby 是由很多语言发展而来的，它在借鉴了这些语言的很多特性的同时，也有自己处理问题的方式。有时候这些方式会让你惊讶万分。

我们的 Ruby 特性学习之旅从检验其独特展现常见的编程思想这个方面开始。也就是说，这些特性会影响编程的每一部分。掌握了这些特性，你将为学习之后的章节做好准备。

第 1 条：理解 Ruby 中的 True

似乎每门语言处理布尔值都有其自己的方式。有些语言仅有一种真假值的表示方法。其他语言使用令人困惑的多种类型来表示，它们时真时假。当对条件表达式的真假值判断错误时会导致程序错误。比如，你知道有多少语言用零值表示假吗？零值为真的语言又有哪些呢？

Ruby 有自己的做事方式，包括布尔值。幸好，区别真假值的规则非常简单。因为它不同于其他语言（这也是写这一条的原因），所以请确认你理解了以下内容。在 Ruby 中，除了 false 和 nil，其他值都是真值。

我们有必要花点时间来想一想这意味着什么。这条简单的规则相比其他主流语言显得有些奇怪。在很多编程语言中，数字 0 表示 false，而其他数字表示 true。而在 Ruby 的规则中，数字 0 表示 true。这也许是从其他编程语言转为 Ruby 程序员时会遇到的最大的陷阱。

如果你过去熟悉的编程语言假设 true 和 false 是关键字，这将是 Ruby 对你玩弄的另一个把戏。它们不是。事实上，true 和 false 被描述为不遵循命名和赋值规范的全局变量。也就是说，它们并不像大多数全局变量一样以字符“$”开头，并且不可以被作为赋值语句的左半边。不过在其他方面都可以将它们视为全局变量。你看：

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

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

正如你所见的，true 和 false 的行为都和全局对象一样，与任何对象一样，你能够调用它们之上的方法。（Ruby 也定义了 TRUE 和 FALSE 这种常量，它们是对这些 true 和 false 对象的引用。）同样，它们来源于两个类：TrueClass 和 FalseClass。两个类中任何一种都允许你创建新的对象；你创建的对象就是 true 或 false。如果了解 Ruby 条件表达式的用法，你就知道 true 对象的存在只是为了方便而已。因为 false 和 nil 是唯二的假值，因此用 true 对象表示真值是冗余的，任何非 false、非 nil 的对象都可以表示真值。

用两个值表示假而用其他所有值表示真有时候可能造成困扰。一个常见的例子是如何区别 false 和 nil。这在表示配置信息的对象中会贯穿始终。这些对象中，false 表示应该被禁用，而 nil 表示选项没有显式定义，因而应使用默认值。最简单的区分方法是使用 nil? 方法，我会在第 2 条中进一步描述 nil? 方法。另一种方式是使用“==”操作符并将 false 作为左操作对象：

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

在某些语言中，形式化规则要求必须把不变量放在等号操作符的左边。这并不是我建议把 false 放在“==”操作符左边的原因。在该情况下是有功能性而非形式化原因的。将 false 放在左边意味着 Ruby 会将表达式解析为对 FalseClass#== 方法的调用（该方法继承自 Object 类）。这样我们可以很放心地知道：如果右边的操作对象也是 false 对象，

那么返回值为 true。换句话说，把 false 置为右操作对象是有风险的，可能不同于我们的期望，因为其他类可能覆盖 Object#== 方法从而改变这个比较：

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

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

当然，这样的写法太愚蠢了。不过在我的经验中，这种方式发生的可能性很大。（顺便提一下，我们将在第 12 条中多讲一点“==”操作符。）

要点回顾

★ 除了 false 和 nil 外所有值都表示真值。

★ 和很多语言不同，Ruby 中的 0 值是真值。

★ 如果你需要区分 false 和 nil，可以使用 nil? 方法或“==”操作符并将 false 作为左操作对象。

第 2 条：所有对象的值都可能为 nil

运行的 Ruby 程序中的每个对象都源自同一个类，即以某种方式继承自类 BasicObject。试想这一个个相互关联的对象是怎样以 BasicObject 为根节点构成一个熟悉的属性图的。在实践中，这意味着一个类的对象能够被另一个类的对象替换（感谢多态）。这就是我们能将一个行为类似数组却又不真的是数组的对象传递给一个以 Array 对象为预期参数的方法的原因。Ruby 程序员喜欢称之为“鸭子类型”（duck typing）。与其要求对象是某个给定类的实例，不如将注意力放在该对象能做什么上；换句话说，接口高于类型。用 Ruby 的术语来说，鸭子类型意味着，相比 is_a? 方法你更喜欢使用 respond_to? 方法。

不过实际中很少见到有方法通过使用 respond_to? 进行参数检查来保证其使用正确的接口。作为替代，我们更倾向于直接调用对象的方法，在对象没有该方法时，让

Ruby 自己在运行时触发 NoMethodError 异常。表面上看，这似乎给 Ruby 程序员带来了真正的问题。好吧，是这样的，只你我二人知道。这也是测试如此重要的原因。没有什么会阻止你意外地将 Time 类型对象传递给接收 Date 对象的方法。我们需要通过优秀的测试挑出各种各样的错误。感谢测试，这些类型的问题是可以通过测试避免的。但即使这样，这些多态替换也可能使经过测试的应用程序出现问题：

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

当你调用一个对象的方法而其返回值刚好是讨厌的 nil 对象时，这种情况就会发生……nil 是类 NilClass 的唯一对象。这样的错误会悄然逃过测试而仅在生产环境下出现：如果一个用户做了些超乎寻常的事情。另一种导致该结果的情况是，当一个方法返回 nil 并将其作为参数直接传给另一个方法时。事实上存在数量惊人的方式可以将 nil 意外地引入你运行中的程序。最好的防范方式是：假设任何对象都可以为 nil，包括方法参数和调用方法的返回值。

避免在 nil 对象上调用方法的最简单的方式是使用 nil? 方法。如果方法接收者（receiver）是 nil，该方法将返回真值，否则返回假值。当然，nil 对象在 Boolean 上下文中总是假值，因此 if 和 unless 表达式可以如你所期望的那样工作。以下几行代码是等价的：

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

将变量显式转换为期望的类型常常比时刻担心其为 nil 要容易得多。尤其是在一个方法即使是部分输入为 nil 时也应该产生结果的时候。Object 类定义了几种转换方法，它们能在这种情况下派上用场。比如，to_s 方法会将方法接收者转化为 string：

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

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

如你所见，NilClass#to_s 返回一个空字符串。使 to_s 如此之棒的原因是 String#to_s 方法只是简单返回 self 而不做任何转换和复制。如果一个变量是 string，那么调用 to_s 的开销最小。但如果变量期待 string 而恰好得到 nil，to_s 能帮你扭转局面。作为例子，假设一个方法期待其参数之一为 string，使用 to_s，你可以避免参数为 nil 产生的问题：

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

有趣的事情还在发生。如你所愿，对几乎所有的内置类（built-in classes）来说都存在一个匹配转换方法。这里有一些适用于 nil 的最有用的例子。

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

irb> nil.to_i
---> 0

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

当需要同时考虑多个值时，你可以使用类 Array 提供的优雅的讨巧方式。Array#compact 方法返回去掉所有 nil 元素的方法接收者的副本。这在将一组可能为 nil 的变量组装成 string 时很常用。比如，如果一个人的名字由 first、middle 和 last 组成（其中任何一个都可能为 nil），那么你可以用下面的代码组成这个名字：

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

nil 对象的嗜好是在你不经意间偷偷溜进正在运行的程序中。无论它来自用户输入、无约束数据库，还是用 nil 来表示失败的方法，意味着每个变量都可能为 nil。

要点回顾

★ 根据 Ruby 的类型系统的运作方式，任何对象都可以为 nil。

★ 如果方法接收者是 nil，nil? 方法返回真值，反之为假。

★ 在适合的时候使用转换方法，如 to_s 和 to_i，可以将 nil 对象强制转换为你期待的类型。

★ Array#compact 方法返回去除所有 nil 元素的接收者的副本。

第 3 条：避免使用 Ruby 中古怪的 Perl 风格语法

如果你曾用过 Perl 语言，那么无疑，你会意识到它对 Ruby 的影响。Ruby 的大部分 Perl 风格语法都已和 Ruby 生态系统的其余部分混合得非常完美。不过其他一些则不然，比如坚持使用不必要的分号，或是代码如此杂乱以致为了明白它如何工作，你抓耳挠腮却不得其解。

多年以来，随着 Ruby 的成熟，有些古怪杂乱的 Perl 风格语法已经被新的语法替代。再后来，有些 Perl 的残留语法被弃用或是从 Ruby 中直接移除。不过，仍然有一些还在，因此你可能会意外地遇到它们。本条可以作为解密那些 Perl 风格语法的指南，并指引你避免在自己的代码中引入它们。

你在 Ruby 中最容易遇到的借自 Perl 的特性是一组特殊的全局变量。事实上，Ruby 对于全局变量拥有漂亮的自由命名规范。与局部变量、实例变量，甚至常量不同，Ruby 允许你使用所有类型的字符作为变量名。回顾一下，全局变量是以“$”字符开头的，思考下面的代码段：

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

这段代码中有两例 Perl 语法。第一例是类 String 的“=~”方法。如果匹配到，它会返回字符串被右操作对象（往往是正则表达式）匹配到的位置，否则返回 nil。当正则表达式匹配时，会将匹配结果设置到几个全局变量中以供导出。本例中，我使用全局变量 $1 导出第一个匹配组的内容。这就是有些古怪的地方了。因为这个变量看起来像是全局变量，但实际上并不是。

由“=~”操作符创建出的变量被称为特殊的全局变量。其原因是，它们的作用域仅限于当前进程的当前方法。从本质上说，它们是形如全局变量的局部变量。前例中，在方法 extract_error 之外，“全局”变量 $1 的值是 nil，即使在使用“=~”操作符之后也是一样。把返回值赋值给 $1 就好比赋值给一个局部变量。整个情况都很令人费解。不过好消息是这一切都不是必要的。来看以下的替代方法：

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

使用方法 String#match 则更符合语言习惯，并且该方法不使用任何由操作符“=~”所产生的特殊的全局变量。这是因为方法 match 的返回值是一个 MatchData 对象（在