

Ruby 代码风格指南

这是 Airbnb 的 Ruby 代码风格指南

指南灵感来自于 [Github](#) 的指南 和 [Bozhidar Batsov](#) 的指南

Airbnb 也在维护 [JavaScript](#) 风格指南

内容表 (Table of Contents)

1. 空格 (Whitespace)
 1. 缩进 (Indentation)
 2. 行内 (Inline)
 3. 换行 (Newlines)
2. 行宽 (Line Length)
3. 注释 (Commenting)
 1. 文件级/类级 注释 (File/class-level comments)
 2. 函数注释 (Function comments)
 3. 块级和行内注释 (Block and inline comments)
 4. 标点符号, 拼写和语法 (Punctuation, spelling, and grammar)
 5. 待办注释 (TODO comments)
 6. 注释掉的代码 (Commented-out code)
4. 方法 (Methods)
 1. 方法定义 (Method definitions)
 2. 方法调用 (Method calls)
5. 条件表达式 (Conditional Expressions)
 1. 关键字 (Conditional keywords)
 2. 三元操作符 (Ternary operator)
6. 语法 (Syntax)
7. 命名 (Naming)
8. 类 (Classes)
9. 异常 (Exceptions)
10. 集合 (Collections)
11. 字符串 (Strings)
12. 正则表达式 (Regular Expressions)
13. 百分比字面量 (Percent Literals)

14. [Rails](#)

1. [范围 \(Scopes\)](#)

15. [保持一致 \(Be Consistent\)](#)

空格 (Whitespace)

缩进 (Indentation)

- 始终用 2 个空格做缩进。 [\[link\]](#)
- when 的缩进和 case 一致。 [\[link\]](#)

Ruby

```
1  case
2  when song.name == 'Misty'
3    puts 'Not again!'
4  when song.duration > 120
5    puts 'Too long!'
6  when Time.now.hour > 21
7    puts "It's too late"
8  else
9    song.play
10 end
11
12 kind = case year
13         when 1850..1889 then 'Blues'
14         when 1890..1909 then 'Ragtime'
15         when 1910..1929 then 'New Orleans Jazz'
16         when 1930..1939 then 'Swing'
17         when 1940..1950 then 'Bebop'
18         else 'Jazz'
19         end
```

- 函数的参数要么全部在同一行，如果参数要分成多行，则每行一个参数， 相同缩进。 [\[link\]](#)

```
1 # 错误
2 def self.create_translation(phrase_id, phrase_key, target_locale,
3                             value, user_id, do_xss_check, allow_ve
4     ...
5 end
6
7 # 正确
8 def self.create_translation(phrase_id,
9                             phrase_key,
10                            target_locale,
11                            value,
12                            user_id,
13                            do_xss_check,
14                            allow_verification)
15     ...
16 end
17
18 # 正确
19 def self.create_translation(
20     phrase_id,
21     phrase_key,
22     target_locale,
23     value,
24     user_id,
25     do_xss_check,
26     allow_verification
27 )
28     ...
29 end
```

- 多行的布尔表达式，下一行缩进一下。[link]

```

1 | # 错误
2 | def is_eligible?(user)
3 |   Trebuchet.current.launch?(ProgramEligibilityHelper::PROGRAM_TREB
4 |   is_in_program?(user) &&
5 |   program_not_expired
6 | end
7 |
8 | # 正确
9 | def is_eligible?(user)
10 |   Trebuchet.current.launch?(ProgramEligibilityHelper::PROGRAM_TREB
11 |   is_in_program?(user) &&
12 |   program_not_expired
13 | end

```

行内 (Inline)

- 行末不要留空格。 [\[link\]](#)
- 写行内注释的时候，在代码和注释之间放 1 个空格。 [\[link\]](#)

```

1 | # 错误
2 | result = func(a, b)# we might want to change b to c
3 |
4 | # 正确
5 | result = func(a, b) # we might want to change b to c

```

- 操作符两边放一个空格；逗号，冒号，分号后面都放一个空格；左大括号 { 两边放空格，右大括号 } 左边放空格。 [\[link\]](#)

```

1 | sum = 1 + 2
2 | a, b = 1, 2
3 | 1 > 2 ? true : false; puts 'Hi'
4 | [1, 2, 3].each { |e| puts e }

```

- 逗号前面永远不要放空格 [\[link\]](#)

```
1 | result = func(a, b)
```

- block 语法里，|| 内部的两边不应该带多余的空格，参数之间应该有1个空格，|| 后面应该有一个空格 [\[link\]](#)

```
1 | # 错误
2 | {}.each { | x, y | puts x }
3 |
4 | # 正确
5 | {}.each { |x, y| puts x }
```

- 感叹号和参数间不要留空格，下面是个正确的例子。 [\[link\]](#)

```
1 | !something
```

- (, [后面不要有空格
],) 前面不要有空格 [\[link\]](#)

```
1 | some(arg).other
2 | [1, 2, 3].length
```

- 字符串插值时候忽略空格。 [\[link\]](#)

```
1 | # 错误
2 | var = "This #{ foobar } is interpolated."
3 |
4 | # 正确
5 | var = "This #{foobar} is interpolated."
```

- 当表达范围时，不要写额外的空格。 [\[link\]](#)

```
1 | # 错误
2 | (0 ... coll).each do |item|
3 |
4 | # 正确
5 | (0...coll).each do |item|
```

换行 (Newlines)

- if 条件保持相同缩进，方便识别哪些是条件，哪些是内容。 [\[link\]](#)

```
1 | if @reservation_alteration.checkin == @reservation.start_date &&
2 |     @reservation_alteration.checkout == (@reservation.start_date +
3 |
4 |     redirect_to_alteration @reservation_alteration
5 | end
```



- 条件语句，块，case 语句，等等东西后面换一行， 例子如下。 [\[link\]](#)

```
1 | if robot.is_awesome?
2 |     send_robot_present
3 | end
4 |
5 | robot.add_trait(:human_like_intelligence)
```

- 不同缩进的代码之间无需空行 (比如 class 或 module 和内容之间)。 [\[link\]](#)

```
1 | # 错误
2 | class Foo
3 |
4 |     def bar
5 |         # body omitted
6 |     end
7 |
8 | end
9 |
10 | # 正确
11 | class Foo
12 |     def bar
13 |         # body omitted
14 |     end
15 | end
```

- 方法之间 1 个空行就好。 [\[link\]](#)

```
1 | def a
2 | end
3 |
4 | def b
5 | end
```

- 1 个空行隔开类似的逻辑。 [\[link\]](#)

```
1 | def transformorize_car
2 |   car = manufacture(options)
3 |   t = transformer(robot, disguise)
4 |
5 |   car.after_market_mod!
6 |   t.transform(car)
7 |   car.assign_cool_name!
8 |
9 |   fleet.add(car)
10 |   car
11 | end
```

- 文件末尾只放一个空行。 [\[link\]](#)

行宽 (Line Length)

- 把每一行控制在可读宽度内，除非有特别理由，每一行应小于 100 个字符 ([rationale](#)) [\[link\]](#)

注释 (Commenting)

虽然写注释很痛苦,但是对于保持代码可读非常重要

下面的规则描述了你应该怎么写注释,以及写在哪里。

但是记住: 注释虽然非常重要,但是最好的注释是代码本身。

直观的变量名本身就是注释,比起起一个奇怪的名字,然后用注释解释要好得多

当写注释时,为你的观众去写: 下一个贡献者需要理解你的代码。

请慷慨一些 - 下一个可能就是你自己!

—Google C++ 风格指南

这一部分的指南从 Google C++ 和 Python 风格指南那边借鉴了很多。

文件/类 级别的注释 (File/class-level comments)

每个类的定义都应该带些注释，说明它是干什么的，以及怎么用它。

如果文件里没有任何 class，或者多于一个 class，顶部应该有注释说明里面是什么。

```
Ruby
1  # Automatic conversion of one locale to another where it is possible,
2  # American to British English.
3  module Translation
4    # Class for converting between text between similar locales.
5    # Right now only conversion between American English -> British, Can
6    # Australian, New Zealand variations is provided.
7    class PrimAndProper
8      def initialize
9        @converters = { :en => { :en-AU => AmericanToAustralian.new,
10                                :en-CA => AmericanToCanadian.new,
11                                :en-GB => AmericanToBritish.new,
12                                :en-NZ => AmericanToKiwi.new,
13                                } }
14      end
15
16      ...
17
18      # Applies transforms to American English that are common to
19      # variants of all other English colonies.
20      class AmericanToColonial
21        ...
22      end
23
24      # Converts American to British English.
25      # In addition to general Colonial English variations, changes "apart
26      # to "flat".
27      class AmericanToBritish < AmericanToColonial
28        ...
29      end
```

所有文件，包括数据和配置文件，都应该有文件级别(file-level)的注释。

```
1 | # List of American-to-British spelling variants.
2 | #
3 | # This list is made with
4 | # lib/tasks/list_american_to_british_spelling_variants.rake.
5 | #
6 | # It contains words with general spelling variation patterns:
7 | #   [trave]led/lled, [real]ize/ise, [flav]or/our, [cent]er/re, plus
8 | # and these extras:
9 | #   learned/learnt, practices/practises, airplane/aeroplane, ...
10 |
11 | sectarianizes: sectarianises
12 | neutralization: neutralisation
13 | ...
```

函数注释 (Function comments)

函数声明的前面都应该有注释，描述函数是做什么的，以及怎么用。

这些注释应该是 描述性的(descriptive) 比如 (“Opens the file”)

而不是 命令式的(imperative) 比如 (“Open the file”);

注释描述这个函数，而不是告诉这个函数应该做什么。

总的来说，函数前面的注释并不负责解释函数是怎么做到它提供的功能的。

解释功能怎么实现的注释，应该零散的分布在函数内部的注释里。


每个函数都应该提到输入和输出是什么，除非碰到以下情况：

- 不是外部可见的函数 (not externally visible)
- 非常短 (very short)
- 很明显 (obvious)

你喜欢什么格式都行。在 Ruby 里两种常见的格式是 [TomDoc](#) 和 [YARD](#)。

你也可以直接简洁明了的写出来，比如这样：

```
1 | # Returns the fallback locales for the_locale.  
2 | # If opts[:exclude_default] is set, the default locale, which is other  
3 | # always the last one in the returned list, will be excluded.  
4 | #  
5 | # For example:  
6 | #   fallbacks_for("pt-BR")  
7 | #     => [:"pt-BR", :pt, :en]  
8 | #   fallbacks_for("pt-BR", :exclude_default => true)  
9 | #     => [:"pt-BR", :pt]  
10 | def fallbacks_for(the_locale, opts = {})  
11 |   ...  
12 | end
```



块级和行内注释 (Block and inline comments)

这个部分有点小麻烦. 如果你下次 code review 需要解释这些代码的话, 你现在就应该写注释。
复杂的操作应该把注释写在前面。

单行的不太容易理解的代码, 注释应该写在代码后面。

```

1 | def fallbacks_for(the_locale, opts = {})
2 |   # dup() to produce an array that we can mutate.
3 |   ret = @fallbacks[the_locale].dup
4 |
5 |   # We make two assumptions here:
6 |   # 1) There is only one default locale (that is, it has no less-speci-
7 |   #     children).
8 |   # 2) The default locale is just a language. (Like :en, and not :en-l
9 |   if opts[:exclude_default] &&
10 |       ret.last == default_locale &&
11 |       ret.last != language_from_locale(the_locale)
12 |     ret.pop
13 |   end
14 |
15 |   ret
16 | end

```

在另一方面，永远不要在注释里描述代码。你要假设读代码的人对这门语言的了解比你知道的多。

相关的一个事儿：不要用块级注释，他们前面没有空格，不方便一眼认出来是注释。 [\[link\]](#)

```

1 | # 错误
2 | =begin
3 | comment line
4 | another comment line
5 | =end
6 |
7 | # 正确
8 | # comment line
9 | # another comment line

```

标点符号, 拼写和语法 (Punctuation, spelling, and grammar)

要注意注释里的标点符号, 拼写和语法;

注释写得好读起来也容易。

注释应该和叙述文 (narrative text) 一样易读 (readable), 有着正确的大小写和标点符号。在很多情况下。完整的句子比句子碎片容易读得多。短的注释, 比如跟在代码后面的, 可以不那么正式, 但风格应该一致。

虽然在提交代码时, 别人指出你在应该用分号的地方用了逗号, 这种小事蛮折磨人的。但重要的是源代码应该保持高度的清晰和可读。正确的标点符号, 拼写, 和语法非常重要。

待办注释 (TODO comments)

当代码是临时解决方案, 够用但是并不完美时, 用 TODO 注释。

TODO 应该全大写, 然后是写这个注释的人名, 用圆括号括起来, 冒号可写可不写。然后后面是解释需要做什么, 统一 TODO 注释样式的目的是方便搜索。括号里的人名不代表这个人负责解决这个问题, 只是说这个人知道这里要解决什么问题。每次你写 TODO 注释的时候加上你的名字。

Ruby

```
1 | # 错误
2 | # TODO(RS): Use proper namespacing for this constant.
3 |
4 | # 错误
5 | # TODO(drumm3rz4lyfe): Use proper namespacing for this constant.
6 |
7 | # 正确
8 | # TODO(Ringo Starr): Use proper namespacing for this constant.
```

注释掉的代码 (Commented-out code)

- 注释掉的代码就不要留在代码库里了。 [\[link\]](#)

方法 (Methods)

方法定义 (Method definitions)

- 函数要接参数的时候就用括号, 不用参数时就不用写括号了。下面是正确的例子。 [\[link\]](#)

```
1 | def some_method
2 |   # body omitted
3 | end
4 |
5 | def some_method_with_parameters(arg1, arg2)
6 |   # body omitted
7 | end
```

- 不要用默认参数，用一个选项 hash 来做这个事。 [\[link\]](#)

```
1 | # 错误
2 | def obliterate(things, gently = true, except = [], at = Time.now)
3 |   ...
4 | end
5 |
6 | # 正确
7 | def obliterate(things, options = {})
8 |   default_options = {
9 |     :gently => true, # obliterate with soft-delete
10 |    :except => [], # skip obliterating these things
11 |    :at => Time.now, # don't obliterate them until later
12 |  }
13 |   options.reverse_merge!(default_options)
14 |
15 |   ...
16 | end
```

- 避免定义单行函数，虽然有些人喜欢这样写，但是这样容易引起一些奇怪的问题 [\[link\]](#)

```
1 | # 错误
2 | def too_much; something; something_else; end
3 |
4 | # 正确
5 | def some_method
6 |   # body
7 | end
```

方法调用 (Method calls)

应该用 **圆括号** 的情况：

- 如果方法会返回值。 [\[link\]](#)

```
1 | # 错误
2 | @current_user = User.find_by_id 1964192
3 |
4 | # 正确
5 | @current_user = User.find_by_id(1964192)
```

- 如果第一个参数需要圆括号。 [\[link\]](#)

```
1 | # 错误
2 | put! (x + y) % len, value
3 |
4 | # 正确
5 | put!((x + y) % len, value)
```

- 方法名和左括号之间永远不要放空格。 [\[link\]](#)

```

1 | # 错误
2 | f (3 + 2) + 1
3 |
4 | # 正确
5 | f(3 + 2) + 1

```

- 对于不用接收参数的方法，**忽略圆括号**。[\[link\]](#)

```

1 | # 错误
2 | nil?()
3 |
4 | # 正确
5 | nil?

```

- 如果方法不返回值 (或者我们不关心返回值)，那么带不带括号都行。
(如果参数会导致代码多于一行，建议加个括号比较有可读性) [\[link\]](#)

```

1 | # okay
2 | render(:partial => 'foo')
3 |
4 | # okay
5 | render :partial => 'foo'

```

不论什么情况:

- 如果一个方法的最后一个参数接收 hash，那么不需要 { }。[\[link\]](#)

```

1 | # 错误
2 | get '/v1/reservations', { :id => 54875 }
3 |
4 | # 正确
5 | get '/v1/reservations', :id => 54875

```

条件表达式 (Conditional Expressions)

关键字 (Conditional keywords)

- 永远不要把 `then` 和多行的 `if/unless` 搭配使用。 [\[link\]](#)

Ruby

```
1 | # 错误
2 | if some_condition then
3 |     ...
4 | end
5 |
6 | # 正确
7 | if some_condition
8 |     ...
9 | end
```

- `do` 不要和多行的 `while` 或 `until` 搭配使用。 [\[link\]](#)

Ruby

```
1 | # 错误
2 | while x > 5 do
3 |     ...
4 | end
5 |
6 | until x > 5 do
7 |     ...
8 | end
9 |
10 | # 正确
11 | while x > 5
12 |     ...
13 | end
14 |
15 | until x > 5
16 |     ...
17 | end
```

- `and`, `or`, 和 `not` 关键词禁用。 因为不值得。 总是用 `&&`, `||`, 和 `!` 来代替。 [\[link\]](#)

- 适合用 `if/unless` 的情况：内容简单，条件简单，整个东西能塞进一行。
不然的话，不要用 `if/unless`。 [\[link\]](#)

Ruby

```
1 | # 错误 - 一行塞不下
2 | add_trebuchet_experiments_on_page(request_opts[:trebuchet_experime
3 |
4 | # 还行
5 | if request_opts[:trebuchet_experiments_on_page] &&
6 |     !request_opts[:trebuchet_experiments_on_page].empty?
7 |
8 |     add_trebuchet_experiments_on_page(request_opts[:trebuchet_experi
9 | end
10 |
11 | # 错误 - 这个很复杂,需要写成多行,而且需要注释
12 | parts[i] = part.to_i(INTEGER_BASE) if !part.nil? && [0, 2, 3].incl
13 |
14 | # 还行
15 | return if reconciled?
```



- 不要把 `unless` 和 `else` 搭配使用。 [\[link\]](#)

Ruby

```
1 | # 错误
2 | unless success?
3 |     puts 'failure'
4 | else
5 |     puts 'success'
6 | end
7 |
8 | # 正确
9 | if success?
10 |     puts 'success'
11 | else
12 |     puts 'failure'
13 | end
```

- 避免用多个条件的 `unless`。 [\[link\]](#)

```

1 | # 错误
2 | unless foo? && bar?
3 |   ...
4 | end
5 |
6 | # 还行
7 | if !(foo? && bar?)
8 |   ...
9 | end

```

- 条件语句 if/unless/while 不需要圆括号。 [\[link\]](#)

```

1 | # 错误
2 | if (x > 10)
3 |   ...
4 | end
5 |
6 | # 正确
7 | if x > 10
8 |   ...
9 | end

```

三元操作符 (Ternary operator)

- 避免使用三元操作符 (?:), 如果不用三元操作符会变得很啰嗦才用。对于单行的条件, 用三元操作符(?:) 而不是 if/then/else/end. [\[link\]](#)


```

1 | # 错误
2 | result = if some_condition then something else something_else end
3 |
4 | # 正确
5 | result = some_condition ? something : something_else

```

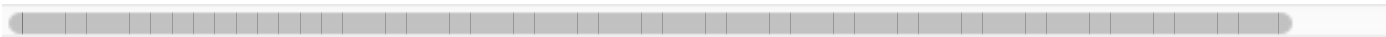
- 不要嵌套使用三元操作符, 换成 if/else. [\[link\]](#)

```
1 | # 错误
2 | some_condition ? (nested_condition ? nested_something : nested_som
3 |
4 | # 正确
5 | if some_condition
6 |     nested_condition ? nested_something : nested_something_else
7 | else
8 |     something_else
9 | end
```



- 避免多条件三元操作符。最好判断一个条件。 [\[link\]](#)
- 避免拆成多行的?: (三元操作符), 用 if/then/else/end 就好了。 [\[link\]](#)

```
1 | # 错误
2 | some_really_long_condition_that_might_make_you_want_to_split_lines
3 |     something : something_else
4 |
5 | # 正确
6 | if some_really_long_condition_that_might_make_you_want_to_split_li
7 |     something
8 | else
9 |     something_else
10 | end
```



语法 (Syntax)

- 永远不要用 for, 除非有非常特殊的理由。
绝大部分情况都应该用 each。for 是用 each 实现的(所以你间接加了一层), 但区别是 - for 不会有新 scope (不像 each) 里面定义的变量外面可见。 [\[link\]](#)

```
1 | arr = [1, 2, 3]
2 |
3 | # 错误
4 | for elem in arr do
5 |   puts elem
6 | end
7 |
8 | # 正确
9 | arr.each { |elem| puts elem }
```

- 单行的情况下, 尽量用 `{...}` 而不是 `do...end`。
多行的情况下避免用 `{...}`。对于“control flow”和“方法定义”(举例: 在 Rakefiles 和某些 DSLs 里) 总是用 `do...end`。
方法连用(chaining)时 避免使用 `do...end` 。 [\[link\]](#)

```
1 | names = ["Bozhidar", "Steve", "Sarah"]
2 |
3 | # 正确
4 | names.each { |name| puts name }
5 |
6 | # 错误
7 | names.each do |name| puts name end
8 |
9 | # 正确
10 | names.each do |name|
11 |     puts name
12 |     puts 'yay!'
13 | end
14 |
15 | # 错误
16 | names.each { |name|
17 |     puts name
18 |     puts 'yay!'
19 | }
20 |
21 | # 正确
22 | names.select { |name| name.start_with?("S") }.map { |name| name.up
23 |
24 | # 错误
25 | names.select do |name|
26 |     name.start_with?("S")
27 | end.map { |name| name.upcase }
```

有些人会说多行连用(chaining) 用 `{...}` 符号 其实也没那么难看, 但他们应该问问自己这代码真的可读吗, 而且 block 里的内容是否可以抽出来弄好看些.

- 尽可能用短的自赋值操作符 (self assignment operators)。 [\[link\]](#)

```

1 | # 错误
2 | x = x + y
3 | x = x * y
4 | x = x**y
5 | x = x / y
6 | x = x || y
7 | x = x && y
8 |
9 | # 正确
10 | x += y
11 | x *= y
12 | x **= y
13 | x /= y
14 | x ||= y
15 | x &&= y

```

- 避免用分号。除非是单行 class 定义的情况下。而且当使用分号时，分号前面不应该有空格。[\[link\]](#)

```

1 | # 错误
2 | puts 'foobar'; # 多余的分号
3 | puts 'foo'; puts 'bar' # 两个表达式放到一行
4 |
5 | # 正确
6 | puts 'foobar'
7 |
8 | puts 'foo'
9 | puts 'bar'
10 |
11 | puts 'foo', 'bar' # this applies to puts in particular

```

- :: 的使用场景是引用常量，(比如 classes 和 modules 里的常量) 以及调用构造函数 (比如 Array() 或者 Nokogiri::HTML())。普通方法调用就不要使用 :: 了。[\[link\]](#)

```
1 | # 错误
2 | SomeClass::some_method
3 | some_object::some_method
4 |
5 | # 正确
6 | SomeClass.some_method
7 | some_object.some_method
8 | SomeModule::SomeClass::SOME_CONST
9 | SomeModule::SomeClass()
```

- 尽量避免用 `return`。 [\[link\]](#)

```
1 | # 错误
2 | def some_method(some_arr)
3 |   return some_arr.size
4 | end
5 |
6 | # 正确
7 | def some_method(some_arr)
8 |   some_arr.size
9 | end
```

- 条件语句里不要用返回值 [\[link\]](#)


```

1 | # 错误 - shows intended use of assignment
2 | if (v = array.grep(/foo/))
3 |   ...
4 | end
5 |
6 | # 错误
7 | if v = array.grep(/foo/)
8 |   ...
9 | end
10 |
11 | # 正确
12 | v = array.grep(/foo/)
13 | if v
14 |   ...
15 | end

```

- 请随意用 `||=` 来初始化变量。 [\[link\]](#)

```

1 | # 把 name 赋值为 Bozhidar, 仅当 name 是 nil 或者 false 时
2 | name ||= 'Bozhidar'

```

- 不要用 `||=` 来初始化布尔变量。(想象下如果值刚好是 `false` 会咋样。) [\[link\]](#)

```

1 | # 错误 - would set enabled to true even if it was false
2 | enabled ||= true
3 |
4 | # 正确
5 | enabled = true if enabled.nil?

```

- 当调用 lambdas 时, 明确使用 `.call`。 [\[link\]](#)

```

1 | # 错误
2 | lambda.(x, y)
3 |
4 | # 正确
5 | lambda.call(x, y)

```

- 避免使用 Perl 风格的特殊变量名 (比如 \$0-9, \$, 等等.). 因为看起来蛮神秘的. 建议只在单行里使用. 建议用长名字, 比如 \$PROGRAM_NAME. [\[link\]](#)
- 当一个方法块只需要 1 个参数, 而且方法体也只是读一个属性, 或者无参数的调用一样方法, 这种情况下用 &:. [\[link\]](#)

```

1 | # 错误
2 | bluths.map { |bluth| bluth.occupation }
3 | bluths.select { |bluth| bluth.blue_self? }
4 |
5 | # 正确
6 | bluths.map(&:occupation)
7 | bluths.select(&:blue_self?)

```

- 当调用当前实例的某个方法时, 尽量用 some_method 而不是 self.some_method. [\[link\]](#)

```

1 | # 错误
2 | def end_date
3 |   self.start_date + self.nights
4 | end
5 |
6 | # 正确
7 | def end_date
8 |   start_date + nights
9 | end

```

在下面 3 种常见情况里, 需要用, 而且应该用self.:

1. 当定义一个类方法时: def self.some_method.
2. 当调用一个赋值方法 (assignment method) 时, 左边应该用 self, 比如当 self 是 ActiveRecord 模型然后你需要赋值一个属性: self.guest = user.

3. 指向 (Referencing) 当前实例的类: `self.class`.

命名 (Naming)

- 用 蛇命名法 (snake_case) 来命名 methods 和 variables。 [\[link\]](#)
- 用 驼峰命名法 (CamelCase) 命名 class 和 module。 (缩写词如 HTTP, RFC, XML 全部大写) [\[link\]](#)
- 用 尖叫蛇命名法 (SCREAMING_SNAKE_CASE) 来命名常量。 [\[link\]](#)
- 断定方法的名字 (predicate methods) (意思是那些返回布尔值的方法) 应该以问号结尾。 (比如 `Array#empty?`)。 [\[link\]](#)
- 有一定“潜在危险”的方法 (意思就是那些. 会修改 `self` 的方法, 或者原地修改参数的方法, 或者带有 `exit!` 的方法, 等等) 应该以感叹号结尾. 这种危险方法应该仅当同名的不危险方法存在之后, 才存在. ([More on this.](#)) [\[link\]](#)
- 把不用的变量名命名为 `_`。 [\[link\]](#)

```
1 | payment, _ = Payment.complete_paypal_payment!(params[:token],  
2 |                                                    native_currency,  
3 |                                                    created_at)
```

Ruby

类 (Classes)

- 避免使用类变量 (@@), 因为在继承的时候它们会有“淘气”的行为。 [\[link\]](#)

```

1 | class Parent
2 |   @@class_var = 'parent'
3 |
4 |   def self.print_class_var
5 |     puts @@class_var
6 |   end
7 | end
8 |
9 | class Child < Parent
10 |   @@class_var = 'child'
11 | end
12 |
13 | Parent.print_class_var # => 会输出"child"

```

你可以看到在这个类的继承层级了，所有的类都共享一个类变量。尽量使用实例变量而不是类变量。

- 用 `def self.method` 来定义单例方法(singleton methods). 这样在需要改类名的时候更方便. [\[link\]](#)

```

1 | class TestClass
2 |   # 错误
3 |   def TestClass.some_method
4 |     ...
5 |   end
6 |
7 |   # 正确
8 |   def self.some_other_method
9 |     ...
10 |   end

```

- 除非必要，避免写 `class << self`，必要的情况比如 single accessors 和 aliased attributes. [\[link\]](#)

```
1 class TestClass
2   # 错误
3   class << self
4     def first_method
5       ...
6     end
7
8     def second_method_etc
9       ...
10    end
11  end
12
13  # 正确
14  class << self
15    attr_accessor :per_page
16    alias_method :nwo, :find_by_name_with_owner
17  end
18
19  def self.first_method
20    ...
21  end
22
23  def self.second_method_etc
24    ...
25  end
26 end
```

- `public`, `protected`, `private` 它们和方法定义保持相同缩进。并且上下各留一个空行。 [\[link\]](#)

```
1 class SomeClass
2   def public_method
3     # ...
4   end
5
6   private
7
8   def private_method
9     # ...
10  end
11 end
```

异常 (Exceptions)

- 不要把异常用于控制流里 (flow of control) [\[link\]](#)

```
1 # 错误
2 begin
3   n / d
4 rescue ZeroDivisionError
5   puts "Cannot divide by 0!"
6 end
7
8 # 正确
9 if d.zero?
10   puts "Cannot divide by 0!"
11 else
12   n / d
13 end
```

- 避免捕捉 Exception 这个大类的异常 [\[link\]](#)

```
1 | # 错误
2 | begin
3 |   # an exception occurs here
4 | rescue Exception
5 |   # exception handling
6 | end
7 |
8 | # 正确
9 | begin
10 |   # an exception occurs here
11 | rescue StandardError
12 |   # exception handling
13 | end
14 |
15 | # 可以接受
16 | begin
17 |   # an exception occurs here
18 | rescue
19 |   # exception handling
20 | end
```

- 传 2 个参数调 raise 异常时不要明确指明 RuntimeError。尽量用 error 子类这样比较清晰和明确。[\[link\]](#)

```
1 | # 错误
2 | raise RuntimeError, 'message'
3 |
4 | # 正确一点 - RuntimeError 是默认的
5 | raise 'message'
6 |
7 | # 最好
8 | class MyExplicitError < RuntimeError; end
9 | raise MyExplicitError
```

- 尽量将异常的类和讯息两个分开作为 raise 的参数，而不是提供异常的实例。[\[link\]](#)

```

1 | # 错误
2 | raise SomeException.new('message')
3 | # 注意, 提供异常的实例没办法做到 `raise SomeException.new('message'), b
4 |
5 | # 正确
6 | raise SomeException, 'message'
7 | # 可以达到 `raise SomeException, 'message', backtrace`.

```

- 避免使用 rescue 的变异形式。 [\[link\]](#)

```

1 | # 错误
2 | read_file rescue handle_error($!)
3 |
4 | # 正确
5 | begin
6 |   read_file
7 | rescue Errno::ENOENT => ex
8 |   handle_error(ex)
9 | end

```

集合 (Collections)

- 尽量用 map 而不是 collect。 [\[link\]](#)
- 尽量用 detect 而不是 find。find 容易和 ActiveRecord 的 find 搞混 - detect 则是明确的说明了 是要操作 Ruby 的集合, 而不是 ActiveRecord 对象。 [\[link\]](#)
- 尽量用 reduce 而不是 inject。 [\[link\]](#)
- 尽量用 size, 而不是 length 或者 count, 出于性能理由。 [\[link\]](#)
- 尽量用数组和 hash 字面量来创建, 而不是用 new。除非你需要传参数。 [\[link\]](#)


```

1 | # 错误
2 | arr = Array.new
3 | hash = Hash.new
4 |
5 | # 正确
6 | arr = []
7 | hash = {}
8 |
9 | # 正确，因为构造函数需要参数
10 | x = Hash.new { |h, k| h[k] = {} }

```

- 为了可读性倾向于用 `Array#join` 而不是 `Array#*`。 [\[link\]](#)

```

1 | # 错误
2 | %w(one two three) * ', '
3 | # => 'one, two, three'
4 |
5 | # 正确
6 | %w(one two three).join(', ')
7 | # => 'one, two, three'

```

- 用 符号(symbols) 而不是 字符串(strings) 作为 hash keys。 [\[link\]](#)

```

1 | # 错误
2 | hash = { 'one' => 1, 'two' => 2, 'three' => 3 }
3 |
4 | # 正确
5 | hash = { :one => 1, :two => 2, :three => 3 }

```

- 如果可以的话, 用普通的 symbol 而不是字符串 symbol。 [\[link\]](#)

```
1 | # 错误
2 | : "symbol"
3 |
4 | # 正确
5 | :symbol
```

- 用 Hash#key? 而不是 Hash#has_key? 用 Hash#value? 而不是 Hash#has_value?. 根据 Matz 的说法, 长一点的那种写法在考虑要废弃掉。 [\[link\]](#)

```
1 | # 错误
2 | hash.has_key?(:test)
3 | hash.has_value?(value)
4 |
5 | # 正确
6 | hash.key?(:test)
7 | hash.value?(value)
```

- 用多行 hashes 使得代码更可读, 逗号放末尾。 [\[link\]](#)

```
1 | hash = {
2 |   :protocol => 'https',
3 |   :only_path => false,
4 |   :controller => :users,
5 |   :action => :set_password,
6 |   :redirect => @redirect_url,
7 |   :secret => @secret,
8 | }
```

- 如果是多行数组, 用逗号结尾。 [\[link\]](#)

```
1 | # 正确
2 | array = [1, 2, 3]
3 |
4 | # 正确
5 | array = [
6 |     "car",
7 |     "bear",
8 |     "plane",
9 |     "zoo",
10| ]
```

字符串 (Strings)

- 尽量使用字符串插值，而不是字符串拼接[\[link\]](#)

```
1 | # 错误
2 | email_with_name = user.name + ' <' + user.email + '>'
3 |
4 | # 正确
5 | email_with_name = "#{user.name} <#{user.email}>"
```

另外，记住 Ruby 1.9 风格的字符串插值。比如说你要构造出缓存的 key 名：

```
1 | CACHE_KEY = '_store'
2 |
3 | cache.write(@user.id + CACHE_KEY)
```

那么建议用字符串插值而不是字符串拼接：

```
1 | CACHE_KEY = '%d_store'
2 |
3 | cache.write(CACHE_KEY % @user.id)
```

- 在需要构建大数据块时，避免使用 `String#+`。
而是用 `String#<<`。它可以原位拼接字符串而且它总是快于 `String#+`，这种用加号的语法会创建一堆新的字符串对象。[\[link\]](#)

Ruby

```
1 | # 正确而且快
2 | html = ''
3 | html << '<h1>Page title</h1>'
4 |
5 | paragraphs.each do |paragraph|
6 |   html << "<p>#{paragraph}</p>"
7 | end
```

- 对于多行字符串，在行末使用 `\`，而不是 `+` 或者 `<<` [\[link\]](#)

Ruby

```
1 | # 错误
2 | "Some string is really long and " +
3 |   "spans multiple lines."
4 |
5 | "Some string is really long and " <<
6 |   "spans multiple lines."
7 |
8 | # 正确
9 | "Some string is really long and " \
10 |   "spans multiple lines."
```

正则表达式 (Regular Expressions)

- 避免使用 `$1-9` 因为可能难以辨认出是哪一个，取个名。 [\[link\]](#)

```

1 | # 错误
2 | /(regexp)/ =~ string
3 | ...
4 | process $1
5 |
6 | # 正确
7 | /(?!<meaningful_var>regexp)/ =~ string
8 | ...
9 | process meaningful_var

```

- 小心使用 `^` 和 `$` 因为它们匹配的是 行头/行末，而不是某个字符串的结尾。如果你想匹配整个字符串，用: `\A` 和 `\z`。[\[link\]](#)

```

1 | string = "some injection\nusername"
2 | string[/^username$/] # matches
3 | string[/\Ausername\z/] # don't match

```

- 使用 `x` 修饰符在复杂的正则表达式上。这使得它更可读, 并且你可以加有用的注释。只是要注意空格会被忽略。[\[link\]](#)

```

1 | regexp = %r{
2 |     start          # some text
3 |     \s             # white space char
4 |     (group)        # first group
5 |     (?:alt1|alt2)  # some alternation
6 |     end
7 | }x

```

百分比字面量 (Percent Literals)

- 统一用圆括号，不要用其他括号， 因为它的行为更接近于一个函数调用。[\[link\]](#)

```

1 | # 错误
2 | %w[date locale]
3 | %w{date locale}
4 | %w|date locale|
5 |
6 | # 正确
7 | %w(date locale)

```

- 随意用 `%w` [\[link\]](#)

```

1 | STATES = %w(draft open closed)

```

- 在一个单行字符串里需要 插值(interpolation) 和内嵌双引号时使用 `%()`。对于多行字符串，建议用 heredocs 语法。 [\[link\]](#)

```

1 | # 错误 - 不需要字符串插值
2 | %(<div class="text">Some text</div>)
3 | # 直接 '<div class="text">Some text</div>' 就行了
4 |
5 | # 错误 - 无双引号
6 | %(This is #{quality} style)
7 | # 直接 "This is #{quality} style" 就行了
8 |
9 | # 错误 - 多行了
10 | %(<div>\n<span class="big">#{exclamation}</span>\n</div>)
11 | # 应该用 heredoc.
12 |
13 | # 正确 - 需要字符串插值，有双引号，单行.
14 | %(<tr><td class="name">#{name}</td>)

```

- 仅在需要匹配 多于一个 `'/'` 符号的时候使用 `%r`。 [\[link\]](#)

```
1 | # 错误
2 | %r(\s+)
3 |
4 | # 依然不好
5 | %r(^/(.*)$)
6 | # should be /\^\/(.*)$/
7 |
8 | # 正确
9 | %r(^/blog/2011/(.*)$)
```

- 避免使用 %x ，除非你要调用一个带引号的命令(非常少的情况)。 [\[link\]](#)

```
1 | # 错误
2 | date = %x(date)
3 |
4 | # 正确
5 | date = `date`
6 | echo = %x(echo `date`)
```

Rails

- 当调用 render 或 redirect_to 后需要马上“返回”时，把 return 放到下一行，不要放到同一行。 [\[link\]](#)

```
1 | # 错误
2 | render :text => 'Howdy' and return
3 |
4 | # 正确
5 | render :text => 'Howdy'
6 | return
7 |
8 | # still bad
9 | render :text => 'Howdy' and return if foo.present?
10 |
11 | # 正确
12 | if foo.present?
13 |   render :text => 'Howdy'
14 |   return
15 | end
```

范围 (Scopes)

- 当定义 ActiveRecord 的模型 scopes 时，把内容用大括号包起来。如果不包的话，在载入这个 class 时就会被强迫连接数据库。 [\[link\]](#)

```
1 | # 错误
2 | scope :foo, where(:bar => 1)
3 |
4 | # 正确
5 | scope :foo, -> { where(:bar => 1) }
```

保持一致 (Be Consistent)

在你编辑某块代码时，看看周围的代码是什么风格。
如果它们在数学操作符两边都放了空格，那么你也应该这样做。
如果它们的代码注释用 # 井号包成了一个盒子，那么你也应该这样做。

风格指南存在的意义是 让看代码时能关注“代码说的是什么”。
而不是“代码是怎么说这件事的”。这份整体风格指南就是帮助你做这件事的。
注意局部的风格同样重要。如果一个部分的代码和周围的代码很不一样。
别人读的时候思路可能会被打断。尽量避免这一点。
