**从其他程式语言到 Ruby**

当你第一眼看见 Ruby 程式码，你一定会想起你熟悉的程式语言。这正是本文件的目的。Ruby 有许多语法和 Perl、Python 和 Java (以及其他程式语言) 类似，所以如果你已经熟悉这些程式语言，那么学习 Ruby 易如反掌。

这份文件包括两大部份。这一部分的用意是整理从 *X* 语言到 Ruby 的重点。第二部分则从 Ruby 的重要功能及特色着手，与其他程式语言来做比较。

**重点整理: 从 *程式语言 X* 到 Ruby**

* [从 C 和 C++ 到 Ruby](http://www.ruby-lang.org/zh_TW/documentation/ruby-from-other-languages/to-ruby-from-c-and-c-/)
* [从 Java 到 Ruby](http://www.ruby-lang.org/zh_TW/documentation/ruby-from-other-languages/to-ruby-from-java/)
* [从 Perl 到 Ruby](http://www.ruby-lang.org/zh_TW/documentation/ruby-from-other-languages/to-ruby-from-perl/)
* [从 PHP 到 Ruby](http://www.ruby-lang.org/zh_TW/documentation/ruby-from-other-languages/to-ruby-from-php/)
* [从 Python 到 Ruby](http://www.ruby-lang.org/zh_TW/documentation/ruby-from-other-languages/to-ruby-from-python/)

**重要的语言特色及一些诀窍**

这里是你学习 Ruby 的一些重点及提示。

**迭代 (Iteration)**

Ruby 有两个常用的特色你可能没见过，那就是 “程式区块(blocks)” 和迭代子(iterators)”。不像使用索引的回圈(例如 C, C++ 和 pre-1.5 Java)，或是回圈控制结构(例如 Perl 的 for (@a) {...}，或是 Python 的 for i in aList: ...)。在 Ruby 里你会常常看到：

some\_list.each do |this\_item|  
  # 我们在程式区块中  
  # 处理 this\_item  
end

关于更多 each 的资讯 (以及 collect, find, inject, sort 等等)，请参考 ri Enumerable (和 ri Enumerable#*func\_name*).

**一切东西都有值**

表达式(expression)和叙述(statement)没有差别，都会有回传值，即使那个值是 **nil**。例如下述用法：

x = 10  
y = 11  
z = if x < y  
  true  
else  
  false  
end  
z # => true

**Symbols 不是轻量化的字串**

许多 Ruby 新手会搞不清楚什么是 Symbols(符号) 可以做什么用。

Symbols 就如同一个识别符号。一个 symbol 就代表它是”谁”了，而不是代表它是”什么”。打开 irb 来看一看它们的区别：

irb(main):001:0> :george.object\_id == :george.object\_id  
=> true  
irb(main):002:0> "george".object\_id == "george".object\_id  
=> false  
irb(main):003:0>

object\_id 方法会回传物件的识别编号。如果有两个物件有相同的 object\_id 表示它们其实是同一个(指向同一个记忆体位置)。

如你所见，使用过 Symbols 之后，任何相同名字的 Symbol 都是指记忆体里的同一个物件。对任何相同名字的 Symbols，它们的 object\_id 都一样。

让我们来看看字串 “george”，它们的 object\_id 并不相同。这表示它们在记忆体里面是不同的物件。每次你建立一个新的字串，Ruby 就会分配新的记忆体空间给它。

如果你不清楚何时使用 Symbol 何时用字串(String)，想想看用途究竟是物件的识别(例如一个杂凑 Hash 的 key)，还是物件内容(比如这个例子的 “george”)。

**所有东西都是物件**

“所有东西都是物件” 并不是夸大，甚至是类别跟整数也是物件，你可以与其他物件一样操作它们：

# 这是等价的程式：  
# class MyClass  
# attr\_accessor :instance\_var  
# end  
MyClass = Class.new do  
  attr\_accessor :instance\_var  
end

译注：在 Ruby 中任何类别都是由 Class 类别所实例(new)出来的物件。

**可变的常数**

常数(Constant)并不真的无法改变。如果你修改了一个已经有值的常数，你会得到一个警告讯息，但程式不会终止。当然这不表示你”应该”修改常数的值。

**命名惯例**

Ruby 规定了一些命名惯例。变数的识别名称，大写字母开头的是常数、钱号($)开头的是全域变数、@ 开头是实例变数(instance variable)、@@ 开头则是类别变数。

方法名称可以允许大写字母开头，虽然可能造成一些混淆，例如：

Constant = 10  
def Constant  
  11  
end

这里的 Constant 是 10，但是 Constant() 却是 11。

**虚拟关键字参数**

Ruby 不像 Python 有关键字参数(keyword parameters)功能，但是可以用 symbols 和杂凑(hash) 来替代。Ruby on Rails 和非常多的函式库都使用了这个方法，例如：

def some\_keyword\_params( params )  
  params  
end  
some\_keyword\_params( :param\_one => 10, :param\_two => 42 )  
# => {:param\_one=>10, :param\_two=>42}

**一切为 true**

在 Ruby 里，除了 **nil** 和 **false** 之外的所有东西，都可以当做 true 值。在 C, Python 和其他语言中，0 和一些其他值，例如空列表，会被当做 false。例如我们看看以下的 Python 程式(其他语言亦同)：

# in Python  
if 0:  
  print "0 is true"  
else:  
  print "0 is false"

这会输出 “0 is false”。而在 Ruby 里:

# in Ruby  
if 0  
  puts "0 is true"  
else  
  puts "0 is false"  
end

这会输出 “0 is true”。

**存取修饰词会作用到底**

在下面的 Ruby 程式中，

class MyClass  
  private  
  def a\_method; true; end  
  def another\_method; false; end  
end

你可能会认为 another\_method 是 public 的，但不是这样。这个 ‘private’ 存取修饰到作用域(scope)结束，或是直到另一个存取修饰词开始作用。方法预设都是 public 的：

class MyClass  
  # 这个 a\_method 是 public 的  
  def a\_method; true; end  
  
  private  
  
  # 这个 another\_method 是 private 的  
  def another\_method; false; end  
end

public, private 和 protected 其实也是一种方法，所以可以接受参数。如果你传入一个 Symbol，那个该 Symbol 代表的方法就会改变存取权限。

**方法存取权限**

在 Java 里，public 表示方法可以被任何人呼叫。protected 表示只有这个类别的实例、衍生类别的实例，以及相同 package 类别的实例可以呼叫，而 private 表示除了这个类别的实例之外，其他都不行呼叫。

在 Ruby 中，public 还是一样是公开的意思，其他则有一点差异。private 表示只有不指定接受者(receiver)时才可以呼叫，也就是只有 **self** 可以当成 private 方法的接受者。

protected 也有点不同。一个 protected 方法除了可以被一个类别或衍生类别的实例呼叫，也可以让另一个相同类别的实例来当做接受者。

来看看 [Ruby FAQ](http://faq.rubygarden.org/entry/show/57?controller_prefix=faq%2F) 的例子：

$ irb  
irb(main):001:0> class Test  
irb(main):002:1> # 预设是 public 的  
irb(main):003:1\* def func  
irb(main):004:2> 99  
irb(main):005:2> end  
irb(main):006:1>   
irb(main):007:1\* def ==(other)  
irb(main):008:2> func == other.func  
irb(main):009:2> end  
irb(main):010:1> end  
=> nil  
irb(main):011:0>   
irb(main):012:0\* t1 = Test.new  
=> #<Test:0x34ab50>  
irb(main):013:0> t2 = Test.new  
=> #<Test:0x342784>  
irb(main):014:0> t1 == t2  
=> true  
irb(main):015:0> # 来让 `func` 变成 protected，一样没问题  
irb(main):016:0\* # 因为 protected 允许其他相同类别的实例呼叫  
irb(main):017:0\* class Test  
irb(main):018:1> protected :func  
irb(main):019:1> end  
=> Test  
irb(main):020:0> t1 == t2  
=> true  
irb(main):021:0> # 来让 `func` 变成 private  
irb(main):022:0\* class Test  
irb(main):023:1> private :func  
irb(main):024:1> end  
=> Test  
irb(main):025:0> t1 == t2  
NoMethodError: private method `func' called for #<Test:0x342784>  
        from (irb):8:in `=='  
        from (irb):25  
        from :0  
irb(main):026:0>

**类别是开放的**

Ruby 的类别是开放的，你可以随时打开它新增一点程式或是修改。即使是核心类别如 Fixnum 或是 Object(这是所有类别的父类别) 都一样。 Ruby on Rails 甚至定义了一堆时间方法到 Fixnum 去，例如：

class Fixnum  
  def hours  
    self \* 3600 # 一小时有多少秒  
  end  
  alias hour hours  
end  
    
# 从一月一号 00:00 往后数 14 个小时  
# (你终于醒了吧 ;)  
Time.mktime(2006, 01, 01) + 14.hours # => Sun Jan 01 14:00:00

**有趣的方法名称**

在 Ruby 里，方法名称允许用问号或惊叹号结尾。惯例上，用来回答是非题的方法会用问号结尾 (例如 Array#empty? 会回传 **true** 如果方法接收者是空的)。有潜在 “危险” (表示有某种副作用，会修改 **self** 或参数值。例如 exit! 等) 的方法会用惊叹号结尾。

但是这不表示所有会修改参数的方法一定有惊叹号结尾，例如 Array#replace 就会替换内容成别的阵列，毕竟 replace 的意思就是要修改替换自己。

**单件方法**

单件方法(Singleton methods)是个别物件才有的方法。它们只存在于你要定义的物件之中。

class Car  
  def inspect  
    "Cheap car"  
  end  
end  
  
porsche = Car.new  
porsche.inspect # => Cheap car  
def porsche.inspect  
  "Expensive car"  
end  
  
porsche.inspect # => Expensive car  
  
# 其他物件就不受影响  
other\_car = Car.new  
other\_car.inspect # => Cheap car

**Missing 方法**

当你呼叫一个不存在的方法，Ruby 仍然有办法处理。它会改呼叫 method\_missing 这个方法，并把这个不存在的方法名称传进去当做参数。method\_missing 预设会丢出一个 NameError 例外，但是你可以根据你的需求重新定义过，也有许多函式库这么做。这是一个例子：

# id 是被呼叫方法的名字，而 \* 符号会收集  
# 所有传进来的参数变成一个叫做 'arguments' 的阵列  
def method\_missing( id, \*arguments )  
  puts "Method #{id} was called, but not found. It has " +   
       "these arguments: #{arguments.join(", ")}"  
end  
  
\_\_ :a, :b, 10  
# => Method \_\_ was called, but not found. It has these  
# arguments: a, b, 10

以上程式会输出呼叫的细节，但你可以随意定义这个讯息。

**传递讯息，不是呼叫函数**

一个方法呼叫(method call)其实就是送一个 **讯息(message)** 给一个物件：

# 这个  
1 + 2  
# 等同于 ...  
1.+(2)  
# 也等同于:  
1.send "+", 2

**Blocks 也算是物件**

程式区块 Blocks (或叫做 closures) 被广泛应用在标准函式库。要执行一个程式区块，可以用 yield ，或是透过一个特别的参数让它变成 Proc，例如：

def block( &the\_block )  
  # 在这里面，the\_block 是被传进来的程式区块  
  the\_block # return the block  
end  
adder = block { |a, b| a + b }  
# adder 是一个 Proc 物件  
adder.class # => Proc

你也可以透过 Proc.new 或 lambda 在方法外建立程式区块。

同样的，方法也可以当做物件：

method(:puts).call "puts is an object!"  
# => puts is an object!

**操作符只是语法包装**

大部分的 Ruby 操作符(operators)只是一种方法呼叫的语法包装(syntactic sugar)，加上一些优先权规则。你要的话，举例来说，我们可以覆写掉 Fixnum 的 + 方法：

class Fixnum  
  # 可以这么做，但请不要这么改  
  def +( other )  
    self - other  
  end  
end

你不需要 C++ 的 operator+ 等等。

甚至有如陣列存取的 [] 和 []= 可以定義。要定義一元的 + and – (想想看 +1 跟 -2)，你必須分別定義 +@ 和 -@ 方法。

以下的操作符則**不是**語法包裝。它們不是方法，不能被覆寫定義：

=, .., ..., !, not, &&, and, ||, or, !=, !~, ::

此外 +=, \*= 等只是 var = var + other\_var 跟 var = var \* other\_var 等的縮寫，因此也不能被覆寫定義。

**更多資料**

如果你需要更多 Ruby 知識，請參考 [文件](http://www.ruby-lang.org/zh_TW/documentation/) 。