**二十分钟 Ruby 体验**

**简介**

这是一个简短的 Ruby 入门体验，应该可以在二十分钟内完成。您必须先安装好 Ruby (如果还没有，请先[下载](http://www.ruby-lang.org/zh_TW/downloads/)安装)

**互动式 Ruby**

Ruby 附带了一支程式可以让你即时看到执行 Ruby 叙述的结果。使用这种互动式环境来学习 Ruby 可说是非常地方便。

打开 IRB (表示 Interactive Ruby)。

* 如果你使用 **Mac OS X** 请打开 Terminal 然后输入 irb 和 enter。
* 如果你使用 **Linux**，请打开一个 shell 然后输入 irb 和 enter。
* 如果你使用 **Windows**，请从开始选单中打开 Ruby 的 fxri 。

irb(main):001:0>

Ok，打开了，现在怎么办？

请输入："Hello World"

irb(main):001:0> "Hello World"

=> "Hello World"

**让 Ruby 听话!**

发生什么事了? 你是不是刚刚写下了全世界最短的 “Hello World” 程式？也不尽然。第二行只是 IRB 告诉我们最后的叙述执行结果。如果我们需要输出 “Hello World” 还需要多一行：

irb(main):002:0> puts "Hello World"

Hello World

=> nil

puts 是 Ruby 的基本输出指令。但是什么是 => nil 呢? 那是那一行叙述的执行结果。 puts 总是回传 nil，nil 在 Ruby 中表示一个绝对的空值。

**你的第一个免费计算机**

其实，我们已经可以用 IRB 来当做一个简单的计算机了。

irb(main):003:0> 3+2

=> 5

三加二，够简单了。那怎么三乘二呢? 你可以试试，一点都不难，你也能够随意输入数字试试。试着按按看 **上** 它会显示上一行的 3+2，然后你可以往左移动到 + 更改成 \* 乘号。

irb(main):004:0> 3\*2

=> 6

接下来，让我们试试看三的平方：

irb(main):005:0> 3\*\*2

=> 9

在 Ruby 中 \*\* 是 “次方” 的意思。但是如果你想开根号呢?

irb(main):006:0> Math.sqrt(9)

=> 3.0

等一下，上面那是什么? 如果你想说：”它是 9 的平方根” 那就对了。让我们来仔细看一下。首先，什么是 Math?

**给程式分类的模组(Modules)**

Math 是一个内建的数学模组。Ruby 的模组有两种作用。这里介绍的是第一种：把功能相似的方法放在一起。Math 模组还包括了像是 sin() 和 tan()。

接下来介绍点(dot)。这个点是做什么用的? 这个点是用来告诉接收者(即 Math 模组)一个讯息。什么讯息呢? 在这个例子，讯息是 sqrt(9)，也就是呼叫 sqrt 方法，并传递 9 作为参数。sqrt 正是 “square root” 平方根的意思。

这个方法的回传值是 3.0。你可能会发现到这不只是 3 而已，还包括小数点。这是因为大多数的情况开根号不会刚好是整数，所以这个函数总是回传浮点数。

如果我们想记住运算的结果呢? 指定到一个变数即可。

irb(main):007:0> a = 3 \*\* 2

=> 9

irb(main):008:0> b = 4 \*\* 2

=> 16

irb(main):009:0> Math.sqrt(a+b) => 5.0

尽管这是一个不错的计算机，但我们将逐渐脱离基本的 Hello World，[让我们继续吧](http://www.ruby-lang.org/zh_TW/documentation/quickstart/2/)。

如果您想说很多次 “Hello”，却不想敲太多按键。是时候定义一个方法了!

irb(main):010:0> def h

irb(main):011:1> puts "Hello World!"

irb(main):012:1> end

=> nil

从 def h 开始定义一个方法。它告诉 Ruby 方法名字是 h。下一行是方法的内容，也就是我们之前看过的 puts "Hello World"。接着最后一行 end 表示方法定义结束。 Ruby 回应 => nil 告诉我们它知悉了这个方法定义。

## 这个简短可以反覆执行的方法

现在来试试看执行这个方法数次：

irb(main):013:0> h

Hello World!

=> nil

irb(main):014:0> h()

Hello World!

=> nil

简单吧。在 Ruby 中呼叫方法只要输入方法名字就可以了。如果方法不需要参数，那么括号可以省略。你也可以加上空括号，但不是必要的。

如果你只想对某个人打招呼呢? 只要重新定义 h 方法接受一个参数即可。

irb(main):015:0> def h(name)

irb(main):016:1> puts "Hello #{name}!"

irb(main):017:1> end

=> nil

irb(main):018:0> h("Matz")

Hello Matz!

=> nil

正如预期... 让我们仔细看看到底发生什么事情。

## 字串内嵌

什么是 #{name} 啊? 这是 Ruby 用来在字串中插入资料的方式。大括号里面的程式会被执行后变成一个字串，然后将整个大括号符号替换掉。例如，我们来把人名变成大写：

irb(main):019:0> def h(name = "World")

irb(main):020:1> puts "Hello #{name.capitalize}!"

irb(main):021:1> end

=> nil

irb(main):022:0> h "chris"

Hello Chris!

=> nil

irb(main):023:0> h

Hello World!

=> nil

这里还有几个小诀窍。第一是我们再次省略了方法的括号。如果方法呼叫很简单，那么括号建议你可以省略掉。另一个诀窍是参数有预设值 World。意思是说 “如果没有给 name 参数，那么就使用预设值 "World"“。

## 发展成 Greeter (接待员)

如果我们需要一个接待员，可以记住你的名字，并且礼貌地欢迎你。你会开始需要使用物件(object)了。让我们来建立 “Greeter” 类别(class)。

irb(main):024:0> class Greeter

irb(main):025:1> def initialize(name = "World")

irb(main):026:2> @name = name

irb(main):027:2> end

irb(main):028:1> def say\_hi

irb(main):029:2> puts "Hi #{@name}!"

irb(main):030:2> end

irb(main):031:1> def say\_bye

irb(main):032:2> puts "Bye #{@name}, come back soon."

irb(main):033:2> end

irb(main):034:1> end

=> nil

新的关键字是 class。这定义了一个新的类别叫做 Greeter，以及属于这个类别的方法。注意到 @name 这是一个实例变数，可以在类别里面的方法中存取到。你可以看到在方法 say\_hi 和 say\_bye 里使用了它。

如何让 Greeter 类别动起来呢? 请接着看 [建立物件](http://www.ruby-lang.org/zh_TW/documentation/quickstart/3/)。

让我们建立一个 greeter 物件来使用：

irb(main):035:0> g = Greeter.new("Pat")

=> #<Greeter:0x16cac @name="Pat">

irb(main):036:0> g.say\_hi

Hi Pat!

=> nil

irb(main):037:0> g.say\_bye

Bye Pat, come back soon.

=> nil

一旦建立了 g 物件，它就会记得它的名字是 Pat。嗯，但是我们如何拿到这个名字的值呢?

irb(main):038:0> g.@name

SyntaxError: compile error

(irb):52: syntax error

        from (irb):52

啊，这样不行。

## 揭开物件的面纱

物件中的实例变数(Instance variables)是隐藏的。虽然可以透过检查(inspect)物件看到这个变数，不过 Ruby 采用了物件导向的思维(译注: 即封装)，内部属性资料基本上是隐藏起来的。

到底 Greeter 物件有哪些方法呢?

irb(main):039:0> Greeter.instance\_methods

=> ["method", "send", "object\_id", "singleton\_methods",

    "\_\_send\_\_", "equal?", "taint", "frozen?",

    "instance\_variable\_get", "kind\_of?", "to\_a",

    "instance\_eval", "type", "protected\_methods", "extend",

    "eql?", "display", "instance\_variable\_set", "hash",

    "is\_a?", "to\_s", "class", "tainted?", "private\_methods",

    "untaint", "say\_hi", "id", "inspect", "==", "===",

    "clone", "public\_methods", "respond\_to?", "freeze",

    "say\_bye", "\_\_id\_\_", "=~", "methods", "nil?", "dup",

    "instance\_variables", "instance\_of?"]

哇。有这么多。我们不是只定义了两个方法，怎么回事呢? 这里列出的是**所有** Greeter 物件的方法，因此也包括了它所继承的类别的方法。如果我们只需要 Greeter 自己的方法，可以传入一个 **false** 参数，表示我们不希望包括父类别的方法。

irb(main):040:0> Greeter.instance\_methods(false)

=> ["say\_bye", "say\_hi"]

看起来好多了。让我们看看 greeter 物件对哪些方法有反应?

irb(main):041:0> g.respond\_to?("name")

=> false

irb(main):042:0> g.respond\_to?("say\_hi")

=> true

irb(main):043:0> g.respond\_to?("to\_s")

=> true

它知道 say\_hi 和 to\_s (意思是转换成字串，这是每个物件都有的方法)，但是不知道 name 这个方法。

## 变更类别，永不嫌晚

那么要怎么能够读取或修改名字呢? Ruby 提供了一个简单的方式来让你存取物件的变数：

irb(main):044:0> class Greeter

irb(main):045:1> attr\_accessor :name

irb(main):046:1> end

=> nil

在 Ruby 里你可以再度打开一个类别然后修改它。这个改变会对之后产生的物件，甚至是已经产生的物件产生即时效果。所以，我们来建立一个新的物件试试看 @name 属性。

irb(main):047:0> g = Greeter.new("Andy")

=> #<Greeter:0x3c9b0 @name="Andy">

irb(main):048:0> g.respond\_to?("name")

=> true

irb(main):049:0> g.respond\_to?("name=")

=> true

irb(main):050:0> g.say\_hi

Hi Andy!

=> nil

irb(main):051:0> g.name="Betty"

=> "Betty"

irb(main):052:0> g

=> #<Greeter:0x3c9b0 @name="Betty">

irb(main):053:0> g.name

=> "Betty"

irb(main):054:0> g.say\_hi

Hi Betty!

=> nil

attr\_accessor 会定义两个新的方法，name 用来取值，而 name= 用来给值。

## 可以欢迎任何东西的 MegaGreeter!

这个 greeter 玩腻了，它一次只能处理一个人。要如何能够有 MegaGreeter 可以不只欢迎这个世界，还可以欢迎不同人，甚至是一群人呢?

接着让我们开一个新档案来撰写 Ruby 程式吧，互动式的 IRB 不适合撰写较长的程式。

要离开 IRB 请输入 “quit” 或 “exit” 或按下 Control-D。

#!/usr/bin/env ruby

class MegaGreeter

  attr\_accessor :names

  # 初始化这个物件

  def initialize(names = "World")

    @names = names

  end

  # 向每个人说 hi

  def say\_hi

    if @names.nil?

      puts "..."

    elsif @names.respond\_to?("each")

      # @names 是可以迭代的阵列容器

      @names.each do |name|

        puts "Hello #{name}!"

      end

    else

      puts "Hello #{@names}!"

    end

  end

  # 向每个人说 bye

  def say\_bye

    if @names.nil?

      puts "..."

    elsif @names.respond\_to?("join")

      # 用逗号将阵列中的元素串接成一个字串

      puts "Goodbye #{@names.join(", ")}. Come back soon!"

    else

      puts "Goodbye #{@names}. Come back soon!"

    end

  end

end

if \_\_FILE\_\_ == $0

  mg = MegaGreeter.new

  mg.say\_hi

  mg.say\_bye

  # 变更成 "Zeke"

  mg.names = "Zeke"

  mg.say\_hi

  mg.say\_bye

  # 变更成一个名字的阵列

  mg.names = ["Albert", "Brenda", "Charles",

    "Dave", "Englebert"]

  mg.say\_hi

  mg.say\_bye

  # 变更成 nil

  mg.names = nil

  mg.say\_hi

  mg.say\_bye

end

把这个档案存成 “ri20min.rb”，然后输入 “ruby ri20min.rb” 来执行它。您应该可以看到输出是：

Hello World!

Goodbye World. Come back soon!

Hello Zeke!

Goodbye Zeke. Come back soon!

Hello Albert!

Hello Brenda!

Hello Charles!

Hello Dave!

Hello Englebert!

Goodbye Albert, Brenda, Charles, Dave, Englebert. Come

back soon!

...

...

这个最后的范例有许多新东西，让我们来[仔细瞧瞧](http://www.ruby-lang.org/zh_TW/documentation/quickstart/4/)。

我们来深入看看这个新程式。注意到由井号(#)开头的第一行，在 Ruby 里，任何在井号之后的内容都是注解，会被直译器忽略。除了档案的第一行是个例外，在 Unix-like 作业系统下这告诉 Shell 如何执行这个档案。其余的注解则用来说明程式。

我们的 say\_hi 方法也改变了：

# 对每个人说 hi

def say\_hi

  if @names.nil?

    puts "..."

  elsif @names.respond\_to?("each")

    # @names 是可以迭代的阵列容器

    @names.each do |name|

      puts "Hello #{name}!"

    end

  else

    puts "Hello #{@names}!"

  end

end

它现在会根据 @names 参数的不同而有不同的行为。如果是 nil，它会输出三个点。没必要对空气打招呼，对吧?

## 循环、回圈，迭代 (Iteration)

如果 @names 物件可以回应 each 方法，那表示你可以迭代它，好让我们轮流对其中的元素打招呼。如果 @names 什么都不是，我们透过字串内嵌把它转化成字串，用预设的打招呼方式。

下面来看一看这是怎么迭代的：

@names.each do |name|

  puts "Hello #{name}!"

end

each 是一个可以接受程式区块(a block of code)的方法，它会对 @names 里的每个元素执行这个程式区块，也就是从 do 到 end 的程式码。一个程式区块就像是一个匿名方法，也像是 lambda。而在直线 | 符号之间的是这个程式区块的参数。

具体来说就是，程式区块里的 name 参数会被轮流指定为容器里的每个元素，然后执行 puts "Hello #{name}!" 。

大多数的程式语言会用 for 回圈来做这件事情，例如在 C 里面：

for (i=0; i<number\_of\_elements; i++)

{

  do\_something\_with(element[i]);

}

这样也行，只是没这么漂亮。你需要一个用过即丢的 i 变数、需要计算容器的长度、检查离开回圈的条件。而 Ruby 的方式漂亮多了，所有的工作都被包装在 each 方法里。在 each 内部会去轮流呼叫 yield "Albert"、yield "Brenda"、yield "Charles" 等等。(译注：yield 的意思是”转交”，也就是跳去执行程式区块)

## 让 Ruby 发光发亮的程式区块(Blocks)

程式区块真正厉害的地方是，它可以处理比迭代更复杂的任务。你可以用来设定(setup)、卸载(teardown)和错误处理，这些都可以隐藏起来让方法使用者无需担心。

# 向每个人说 bye

def say\_bye

  if @names.nil?

    puts "..."

  elsif @names.respond\_to?("join")

    # 用逗号将阵列中的元素串接成一个字串

    puts "Goodbye #{@names.join(", ")}. Come back soon!"

  else

    puts "Goodbye #{@names}. Come back soon!"

  end

end

say\_bye 方法没有用到 each，而是检查 @names@ 是不是可以回应 join 方法。如果可以，就呼叫它。不然它就把它当做字串输出。这个方法并不在乎变数真正的 型别(type)，而只关心变数是不是可以回应支援某个方法，这种风格叫做 “鸭子型别(Duck Typing)”，意义就是 “如果它走起来像鸭子，叫起来也像鸭子，那就当它是鸭子”。这种思维的好处是它不会被变数的型别所限制。如果有人写了新的容器类别，并且也实作了 join 这个方法，那么我们就可以在这里使用它。

## 让脚本跑起来

这就是 MegaGreeter 类别了。剩下的部份是就是使用这个类别而已。唯一要注意的技巧是以下这行：

if \_\_FILE\_\_ == $0

\_\_FILE\_\_ 是一个预先定义好的变数，内容是目前这个档案的名称。而 $0 是执行这只程式的执行档名称。这个检查是说 “如果这个档案就是执行程式的档案....”。这可以允许将这个档案当做方法库使用。也就是说，这个档案如果当做方法库使用时，不会执行这段程式。如果当做执行档执行，就会执行这段程式。

## 就这样啦

这就是二十分钟的 Ruby 入门体验了。还有许多值得探索的地方，例如不同的控制结构、如何使用程式区块及 yield、模组(modules)的第二个用途 mixin 等等。希望这份体验教学让你有兴趣继续学习。

如果你希望进一步了解，欢迎前往我们的 [文件](http://www.ruby-lang.org/zh_TW/documentation/)，那里提供了一些免费的线上文件和导览。

或是你希望找本书，请参考看看 [书籍清单](http://www.ruby-doc.org/bookstore) (外部连结)，或是你本地的书店。