5 天搞定 Ruby on Rails

企业内训纲要

第一天：Ruby 编程基础

版本： V1

口号：快速迭代，不断完善

ruby 技术交流--南方群 95824005

ruby 技术交流--北方群 101388340

ruby 技术交流--东部群 236263776

ruby 技术交流--西部群 230015785

ruby 技术交流--中部群 104131248

文档，实例地址：

https://github.com/nienwoo/ruby-learn

1. 第一天目标

- 了解 Ruby 的概貌 ；
- 掌握如何配置 Ruby 开发环境，如何下载安装和使用 Ruby 软件包;
- 下载、安装、和使用 Ruby IDE 集成开发软件。
- 如何编写与运行 Ruby 程序 ；掌握运行 Ruby 程序的步骤。
- 了解 Ruby 的数据类型。掌握各种变量的声明方式。理解运算符的优先级。
- 掌握 Ruby 基本数据类型、运算符与表达式、数组的使用方法。
- 理解 Ruby 程序语法结构 ，掌握顺序结构、选择结构和循环结构语法。
- 掌握 Ruby 语言的编程规则。

2. Ruby 简介

- **Ruby 是一种纯粹的面向对象编程语言。**

它由日本的松本行弘创建于 1993 年。

- **Ruby 是"程序员的最佳朋友"。**

Ruby 的特性与 Smalltalk、Perl 和 Python 类似。Perl、Python 和 Smalltalk 是脚本语言。Smalltalk 是一个真正的面向对象语言。Ruby，与 Smalltalk 一样，是一个完美的面向对象语言。使用 Ruby 的语法比使用 Smalltalk 的语法要容易得多。

2.1. Ruby 的特性

Ruby 是开源的，在 Web 上免费提供，但需要一个许可证。

Ruby 是一种通用的、解释的编程语言。

Ruby 是一种真正的面向对象编程语言。

Ruby 语法简单，这使得新的开发人员能够快速轻松地学习 Ruby。

Ruby 与 C++ 和 Perl 等许多编程语言有着类似的语法。

Ruby 可扩展性强，用 Ruby 编写的大程序易于维护。

Ruby 可用于开发的 Internet 和 Intranet 应用程序。

2.2. Rails 前景广阔

（1）在国外，一大批的创业公司采用 Rails 作为核心技术栈.

广为人知的:

- **Twitter (推特: https://twitter.com)**

Twitter 从 06 年起便开始将其核心构建在 Rails 之上，之后发展超过 5 年，在 11 年才逐步将其搜索业务与消息队列从 Ruby 分别切换到 Java 与 Scala 上.随后 12 年之后，技术栈开始慢慢地多元化，并开源了前端框架 Bootstrap(没错，就是大名鼎鼎的它)

- **Github (找程序员哪里去, Github 等着你: https://github.com)**

Github 是程序员事实的社交圈子，它从 09 年开始起家，一直构建于 Rails 技术栈之上. 经历过防火长城的恶意攻击，却仍然坚挺. 只要有良好的横向扩展能力, Rails 的伸缩性也十分给力.

- **Kickstarter(https://kickstarter.com)**

Kickstarter 是全球最著名的众筹平台, 09 年创建，目前每天的访问量十分巨大，其核心构建于 Rails.

只是找一些知名的案例，大家肯定不满意，别的平台与框架下也有不少案例，对吧. 但接下来就有意思了，越来越多的创业团队开始在使用 Rails.

- **比如最近的 producthunt**

(一个发现新产品的平台，国内有很多赝品了: http://www.producthunt.com/)

基于 Rails 构建，据说，只花了几天就把原型做好了.

- **hired (程序员拍卖网，国内已有类似产品: https://hired.com/)**

基于 Rails 构建.

- **dribbble (设计师社交平台: https://dribbble.com)**

已经是全球设计师社交圈子事实的标准，基于 Rails 构建.

(2) 国内的一些作品:

- **Tower (一个项目协作工具: https://tower.im)**

完全基于 Rails 构建.

- **36kr (创业媒体: http://36kr.com)**

基于 Rails 构建.

- **Knewone (非常有势头的科技与设计产品社区: https://knewone.com)**

完全基于 Rails 构建.

- **100offer (另一个程序员拍卖网，用户体验很出色: http://100offer.com)**

完全基于 Rails 构建.

3. Ruby 开发环境

3.1. 开发 Ruby 的操作系统环境

（1）如何选择操作系统

目前的主流操作系统分为：

- Windows
- Linux/Unix

开发 Ruby 程序，建议最好是在 Linux/Unix 环境。在 Windows 搭建开发环境会冒出很多令人疼痛不已的问题，问题很多是小问题， 但是甚至难以逾越。

✧ 小提示：

如何选择 Linux/Unix 操作系统？ 请百度。 这里的建议是 Ubuntu

如何安装 Ubuntu 操作系统？ 请百度。

- 问题：

Windows 带给我们的方便，是 Linux 类系统没有办法比拟的。

可以折中起来，在 Windows 系统下安装虚拟环境使用 Linux。具体如何做？ 有很多的方式。具体可以百度。 就目前来说，下面的方法最为简单的。

3. 1. 1. vagrant + virtualbox 搭建 Ruby 开发环境

（1）安装 virtualbox 和 vagrant 和 linux

我使用的是苹果笔记本，跑 Mac 系统，但是我的 rails 程序要跑在 ubuntu 系统上。那最简单的解决方案就是安装 virtualbox 虚拟机，再装上 vagrant 。 这样就很容易的把 ubuntu 装上了。当然如果你已经习惯了用其他的方式安装 ubuntu 也是可以的。

到 vagrant 的下载页面 ，选择 Mac 版本下载，双击来安装。到终端中，执行

```
vagrant -v
```

如果看到输出，表示已经装好了。

（2）到 vagrantcloud 上找一个 box

更新： vagrantcloud.com 现在已经合并到了 https://atlas.hashicorp.com .

个干净的 ubuntu14.04 系统就行。使用 https://vagrantcloud.com/ubuntu/boxes/trusty64 。

这个就是我要的 64 位 ubuntu14.04 系统。到终端里执行

```
mkdir rails10-va
cd rails10-va
vagrant init ubuntu/trusty64
```

❖ 接下来执行

```
vagrant up
```

安装过程就开始了，一般首次运行需要十几分钟时间。

❖ 基本操作

vagrant up 之后，系统就装好并启动起来了，可以运行

```
vagrant ssh
```

登陆到虚拟机里面，默认的用户叫做 vagrant，可以用 whoami 查看一下。再查看一下内存，用 free -m 命令，发现默认内存大小还不到 500M，所以敲 Ctrl-D 退出来。添加下面几行到 Vagrantfile 文件。

```
config.vm.provider "virtualbox" do |v|
    v.memory = 2048
end
```

❖ 然后执行

```
vagrant reload
```

来加载设置就可以了。

（3）安装 cygwin

建议通过搜索引擎找方法， cygwin 的安装时间比较长， 需要有写耐心。

3.2. Linux 下载安装 Ruby

下面列出了在 Linux 机器上安装 Ruby 的步骤。

注意：在安装之前，请确保您有 root 权限。

（1）源码安装

下载最新版的 Ruby 压缩文件。请点击这里下载。

下载 Ruby 之后，解压到新创建的目录下：

```
$ tar -xvzf ruby-2.2.3.tgz
$ cd ruby-2.2.3
```

现在，配置并编译源代码，如下所示：

```
$ ./configure
$ make
$ sudo make install
```

安装后，通过在命令行中输入以下命令来确保一切工作正常：

```
$ruby -v
ruby 2.2.3.....
```

如果一切工作正常，将会输出所安装的 Ruby 解释器的版本，如上所示。如果您安装了其他版本，则会显示其他不同的版本。

自动安装 Ruby

如果您的计算机已经连接到 Internet，那么最简单安装 Ruby 的方式是使用 yum 或 apt-get。在命令提示符中输入以下的命令，即可在您的计算机上安装 Ruby。

```
sudo apt-get install ruby-full
```

如果你是苹果系统，可以使用 brew 命令安装：

```
$ brew install ruby
```

3. 3. Ruby IDE 集成开发工具

为了编写 Ruby 程序，您需要一个编辑器：

如果您是在 Windows 上进行编写，那么您可以使用任何简单的文本编辑器，比如 Notepad 或 Edit plus。

VIM（Vi IMproved）是一个简单的文本编辑器，几乎在所有的 Unix 上都是可用的，现在也能在 Windows 上使用。另外，您还可以使用您喜欢的 vi 编辑器来编写 Ruby 程序。

RubyWin 是一个针对 Windows 的 Ruby 集成开发环境（IDE）。

Ruby Development Environment （RDE） 对于 Windows 用户来说，也是一个很好的集

成开发环境（IDE）。

3. 4. 交互式 Ruby（IRb）

交互式 Ruby（IRb）为体验提供了一个 shell。在 IRb shell 内，您可以逐行立即查看解释结果。

这个工具会随着 Ruby 的安装自动带有，所以您不需要做其他额外的事情，IRb 即可正常工作。

- **如何启动？**

只需要在命令提示符中键入 irb，一个交互式 Ruby Session 将会开始，如下所示：

```
$irb
irb 0.6.1(99/09/16)
```

4. 第一个 ruby 程序

4.1. 从交互式命令开始

只需要在命令提示符中键入 irb，一个交互式 Ruby Session 将会开始，如下所示：

```
$irb
irb 0.6.1(99/09/16)
```

4.1.1. 最简单的程序

一句话输出 "Hello World"。

- **命令如下：**

```
vagrant@precise32:/var/www/lessons/chapter1$ irb
2.1.4 :001 > puts "hello wold"
```

- **输出结果：**

```
hello wold

 => nil
```

4.1.2. 复杂一点，搞一个函数

- **定义一个函数，irb 命令如下**

```
irb(main):001:0> def hello
irb(main):002:1> out = "Hello World"
irb(main):003:1> puts out
irb(main):004:1> end
nil
```

- **执行函数，irb 环境下执行结果如下**

```
irb(main):005:0> hello
Hello World
nil
irb(main):006:0>
irb(main):006:0>quit
```

4.2. 离开 irb，使用 rb 文件完成“hello word”

- 如何使用文件？

前一小节的都是交互式的 ruby 命令。如何将程序都放在文件中执行呢？

让我们编写一个简单的 Ruby 程序。所有的 Ruby 文件扩展名都是 .rb。所以，把下面的源代码放在 helloworld.rb

```
```

```
def    hello
out = "Hello World"
puts out
end

hello
```

- **使用 ruby 命令执行，执行结果如下**

```
vagrant@precise32:/var/www/lessons/chapter1$ ruby helloworld.rb

Hello World
```

5. Ruby 语法基础

5.1. Ruby 标识符

❖ 什么是标识符？

标识符是变量、常量和方法的名称。

❖ 是不是大小写敏感？

Ruby 标识符是大小写敏感的。

✧ 这意味着 Ram 和 RAM 在 Ruby 中是两个不同的标识符。

❖ 标识符有哪些内容？

Ruby 标识符的名称可以包含字母、数字和下划线字符（ _ ）。

5.2. 标识符之间的空白

❖ 小心空白！

实例：

```
a + b 被解释为 a+b （这是一个局部变量）
a  +b 被解释为 a(+b) （这是一个方法调用）
```

在 Ruby 代码中的空白字符，如空格和制表符一般会被忽略，一旦它们出现在标识符

字符串中时，就不一样了才不会被忽略。然而，有时候它们用于解释模棱两可的语句。

当启用 -w 选项时，这种解释会产生警告。

5.3. Ruby 程序中的行尾

Ruby 把分号和换行符解释为语句的结尾。但是，如果 Ruby 在行尾遇到运算符，比如 +、- 或反斜杠，它们表示一个语句的延续。

5.4. 保留字

- **保留字不能作为常量或变量的名称。**
- **保留字不能不可以作为方法名。**

下表列出了 Ruby 中的保留字。

BEGIN	do	next	then	
END	else	nil	true	
alias	elsif	not	undef	
and	end	or	unless	
begin	ensure	redo	until	
break	false	rescue	when	
case	for	retry	while	
class	if	return	while	
def	in	self	__FILE__	
defined?	module	super	__LINE__	

5.5. 多行字符串

- **基本规则：**

在 << 之后，您可以指定一个字符串或标识符来终止字符串，且当前行之后直到终止符为止的所有行是字符串的值。

❖ **例子**

```
word_a= <<EOF
    这是第一种方式创建 here document 。
    多行字符串。
EOF
```

✧ 请注意

<< 和终止符之间必须没有空格。

❖ **完整的例子代码文件：multline_word.rb**

```
word_a= <<EOF
    这是第一种方式创建 here document 。
    多行字符串。
EOF

word_b= <<"EOF";                    # 与上面相同
    这是第二种方式创建 here document 。
    多行字符串。
EOF

word_c= <<'EOC'                     # 执行命令
echo hi there
echo lo there
```

```
EOC

word_d= <<"foo", <<"bar"          # 您可以把它们进行堆叠
    I said foo.
foo
    I said bar.
bar

puts    word_a
puts   "word_b=", word_b
puts   "word_c=", word_c
puts   "word_d=", word_d
```

❖ **执行结果**

```
vagrant@precise32:/var/www/lessons/chapter1$ ruby multline_word.rb

    这是第一种方式创建 here document 。

    多行字符串。

word_b=

    这是第二种方式创建 here document 。

    多行字符串。

word_c=

echo hi there

echo lo there
```

```
word_d=
        I said foo.
        I said bar.
```

5.6. Ruby 注释

5.6.1. 行注释

✧ 什么是行注释？

注释会对 Ruby 解释器隐藏一行，或者一行的一部分，或者若干行。您可以在行首使用字符（ # ）：

❖ **例子 1：**

```
# 我是注释，请忽略我。
```

或者，注释可以跟着语句或表达式的同一行的后面。

❖ **例子 2：**

```
name = "Madisetti" # 这也是注释
```

5.6.2. 块注释

✧ 什么是块注释？

块注释是一次注释一块，多行。

- **格式为**

```
=begin
.....
=end
```

- **例子：**

```
=begin
这是注释。
这也是注释。
这也是注释。
这还是注释。
=end
```

6. Ruby 数据类型

本章节我们将为大家介绍 Ruby 的基本数据类型。

Ruby 支持的数据类型包括基本的 Number、String、Ranges、Symbols，以及 true、false 和 nil 这几个特殊值，同时还有两种重要的数据结构——Array 和 Hash。

6.1. 数值类型(Number)

6.1.1. 整型(Integer)

整型分两种，如果在 31 位以内（四字节），那为 Fixnum 实例。如果超过，即为 Bignum 实例。

整数范围从 -230 到 230-1 或 -262 到 262-1。在这个范围内的整数是类 Fixnum 的对象，在这个范围外的整数存储在类 Bignum 的对象中。

您可以在整数前使用一个可选的前导符号，一个可选的基础指标（0 对应 octal，0x 对应 hex，0b 对应 binary），后跟一串数字。下划线字符在数字字符串中被忽略。

您可以获取一个 ASCII 字符或一个用问号标记的转义序列的整数值。

6.1.2. 浮点型

Ruby 支持浮点数。它们是带有小数的数字。浮点数是类 Float 的对象，且可以是下列中任意一个。

6.2. 字符串类型

（1）字符串=String 对象

- **人类语言的字符串=Ruby 中的 String 对象。**

✧ 字符串作用：

持有和操纵的任意序列的一个或多个字节

简单的字符串文本括在单引号（单引号字符）。

引号内的文本的字符串值，代码如下：

- **字符串简单例子**

```
'This is a simple Ruby string literal'
```

（2）字符串内如何含有单引号字符？

- **方法一：**

如果需要内放置一个单引号，单引号的字符串文字，在它前面加上一个反斜杠：

```
'This is a \'simple Ruby string literal'
```

- **方法二**

```
"This is a simple Ruby string literal"
```

双引号标记的字符串可以包含单引号，不需要转义。

✧ 双引号和单引号的区别在哪里？

双引号标记的字符串允许替换，单引号标记的字符串不允许替换。

什么是替换？

（3）替换

您可以使用序列 #{ expr } 替换任意 Ruby 表达式的值为一个字符串。在这里，expr 可以是任意的 Ruby 表达式。

（4）反斜线转义字符

❖ **下表列出了 Ruby 支持的反斜线符号：**

\n 换行符 (0x0a)

\r 回车符 (0x0d)

\f 换页符 (0x0c)

\b 退格键 (0x08)

\a 报警符 Bell (0x07)

\e 转义符 (0x1b)

\s 空格符 (0x20)

\nnn 八进制表示法 (n 是 0-7)

\xnn 十六进制表示法 (n 是 0-9、a-f 或 A-F)

6.3. 范围类型

一个范围表示一个区间。

范围是通过设置一个开始值和一个结束值来表示。

✧ 范围的构造

- **可使用 s..e 和 s...e 来构造**
- **或者通过 Range.new 来构造**

✧ s..e 和 s...e 的不同

- **使用 .. 构造的范围从开始值运行到结束值（包含结束值）。**
- **使用 ... 构造的范围从开始值运行到结束值（不包含结束值）。**

当作为一个迭代器使用时，范围会返回序列中的每个值。

范围 (1..5) 意味着它包含值 1, 2, 3, 4, 5，范围 (1...5) 意味着它包含值 1, 2, 3, 4 。

6.4. 数组

数组字面量通过[]中以逗号分隔定义，且支持 range 定义。

（1）数组通过[]索引访问

（2）通过赋值操作插入、删除、替换元素

（3）通过+，-号进行合并和删除元素，且集合做为新集合出现

（4）通过<<号向原数据追加元素

（5）通过*号重复数组元素

（6）通过｜和&符号做并集和交集操作（注意顺序）

6.5. 哈希类型

Ruby 哈希是在大括号内放置一系列键/值对，键和值之间使用逗号和序列 => 分隔。尾部的逗号会被忽略。

6.6. 示例代码：

data_type.rb

```
#使用#{ expr } 取得表达式的值
puts "Multiplication Value : #{24*60*60}"

name="Ruby"
puts name
puts "#{name+",ok"}"
```

```
#范围类型示例
(100..150).each do |n|
   print n, ' '
end

#数组类型示例
ary = [ "fred", 10, 3.14, "This is a string", "last element", ]
ary.each do |i|
   puts i
end


#hash 类型示例
hsh = colors = { "red" => 0xf00, "green" => 0x0f0, "blue" => 0x00f }
hsh.each do |key, value|
   print key, " is ", value, "\n"
End
```

7. 控制结构

7.1. 条件

Ruby 提供了几种很常见的条件结构。在这里，我们将解释所有的条件语句和 Ruby 中可用的修饰符。

（1）if...else 语句

- **语法**

```
if conditional [then]
        code...
[elsif conditional [then]
        code...]...
[else
        code...]
end
```

if 表达式用于条件执行。如果 conditional 为真，则执行 code。如果 conditional 不为真，则执行 else 子句中指定的 code。

值 false 和 nil 为假，其他值都为真。请注意，Ruby 使用 elsif，不是使用 else if 和 elif。

通常我们省略保留字 then 。若想在一行内写出完整的 if 式，则必须以 then 隔开条件式和程式区块。如下所示:

```
if a == 4 then a = 7 end
```

❖ **实例**

```
x=1
if x > 2
    puts "x 大于 2"
elsif x <= 2 and x!=0
    puts "x 是 1"
else
    puts "无法得知 x 的值"
end
```

尝试一下 »

以上实例输出结果：

```
x 是 1
```

（2）if 修饰符

语法

```
code if condition
```

if 修饰词组表示当 if 右边之条件成立时才执行 if 左边的式子。即如果 conditional 为真，则执行 code。

❖ **实例**

```
#!/usr/bin/ruby
```

```
$debug=1
print "debug\n" if $debug
```

❖ **以上实例输出结果：**

```
debug
```

（3）unless 语句

❖ **语法**

```
unless conditional [then]
    code
[else
    code ]
end
```

unless 式和 if 式作用相反，即如果 conditional 为假，则执行 code。如果 conditional 为真，则执行 else 子句中指定的 code。

❖ **实例**

```
#!/usr/bin/ruby
# -*- coding: UTF-8 -*-

x=1
unless x>2
     puts "x 小于 2"
 else
    puts "x 大于 2"
end
```

❖ **以上实例输出结果为：**

```
x 小于 2
```

(4) unless 修饰符

❖ 语法

```
code unless conditional
```

如果 conditional 为假，则执行 code。

❖ 实例

```
#!/usr/bin/ruby
# -*- coding: UTF-8 -*-

$var =  1
print "1 -- 这一行输出\n" if $var
print "2 -- 这一行不输出\n" unless $var

$var = false
print "3 -- 这一行输出\n" unless $var
```

❖ 以上实例输出结果：

```
1 -- 这一行输出

3 -- 这一行输出
```

(5) case 语句

❖ 语法

```
case expression
[when expression [, expression ...] [then]
    code ]...
[else
    code ]
end
```

case 先对一个 expression 进行匹配判断，然后根据匹配结果进行分支选择。

它使用 ===运算符比较 when 指定的 expression，若一致的话就执行 when 部分的内容。

通常我们省略保留字 then 。若想在一行内写出完整的 when 式，则必须以 then 隔开条件式和程式区块。如下所示:

```
when a == 4 then a = 7 end
```

- **实例 1：**

```
case expr0
when expr1, expr2
    stmt1
when expr3, expr4
    stmt2
else
    stmt3
end
```

- **基本上类似于：**

```
_tmp = expr0
if expr1 === _tmp || expr2 === _tmp
    stmt1
elsif expr3 === _tmp || expr4 === _tmp
```

```
    stmt2
else
    stmt3
end
```

❖ **实例 2**

```
$age =  5
case $age
when 0 .. 2
    puts "baby"
when 3 .. 6
    puts "little child"
when 7 .. 12
    puts "child"
when 13 .. 18
    puts "youth"
else
    puts "adult"
End
```

❖ **以上实例输出结果为：**

```
little child
```

✧ case 后面的表达式还可以省去！

当 case 的"表达式"部分被省略时，将计算第一个 when 条件部分为真的表达式。

```
foo = false
bar = true
quu = false
```

```
case
when foo then puts 'foo is true'
when bar then puts 'bar is true'
when quu then puts 'quu is true'
end
```

❖ **结果**

```
# 显示 "bar is true"
```

7.2. 循环结构

Ruby 中的循环用于执行相同的代码块指定的次数。本章将详细介绍 Ruby 支持的循环语句。

(1) while

Ruby while 语句: 语法:

```
while conditional [do]
    code
end
```

执行代码当条件为 true 时。while 循环的条件是代码中的保留字，换行，反斜杠（）或一个分号隔开。

❖ **实例:**

```
#!/usr/bin/ruby

$i = 0
$num = 5

while $i < $num  do
   puts("Inside the loop i = #$i" )
   $i +=1
end
```

这将产生以下结果：

```
Inside the loop i = 0

Inside the loop i = 1

Inside the loop i = 2

Inside the loop i = 3

Inside the loop i = 4
```

(2) while 修辞符

Ruby while 修辞符: 语法:

```
code while condition
```

OR

```
begin
```

```
    code
end while conditional
```

执行代码，当条件为 true。

如果 while 修饰符紧跟一个 begin 语句但是没有 rescue 或 ensure 子句，代码被执行前一次条件求值。

- **实例:**

```
#!/usr/bin/ruby

$i = 0
$num = 5
begin
    puts("Inside the loop i = #$i" )
    $i +=1
end while $i < $num
```

这将产生以下结果：

```
Inside the loop i = 0

Inside the loop i = 1

Inside the loop i = 2

Inside the loop i = 3

Inside the loop i = 4
```

（3）until 语句:

```
until conditional [do]
   code
end
```

执行代码当条件为 false。until 条件语句从代码分离的保留字，换行符或分号。

- **语句:**

```
#!/usr/bin/ruby

$i = 0
$num = 5

until $i > $num    do
   puts("Inside the loop i = #$i" )
   $i +=1;
end
```

- **这将产生以下结果：**

```
Inside the loop i = 0

Inside the loop i = 1

Inside the loop i = 2

Inside the loop i = 3

Inside the loop i = 4
```

```
Inside the loop i = 5
```

（4）until 修辞符:

❖ 语法:

```
code until conditional

OR

begin
    code
end until conditional
```

执行代码当条件为 false。

如果 until 修辞符跟着 begin 语句但没有 rescue 或 ensure 子句，代码一旦被执行在条件求值之前。

❖ 例子:

```
#!/usr/bin/ruby

$i = 0
$num = 5
begin
    puts("Inside the loop i = #$i" )
    $i +=1;
end until $i > $num
```

这将产生以下结果：

```
Inside the loop i = 0

Inside the loop i = 1

Inside the loop i = 2

Inside the loop i = 3

Inside the loop i = 4

Inside the loop i = 5
```

(5) for 语句

- **语法:**

```
for variable [, variable ...] in expression [do]
    code
end
```

一次执行代码的每个元素在 in 表达式。

实例:

```
#!/usr/bin/ruby

for i in 0..5
    puts "Value of local variable is #{i}"
End
```

这里我们定义的范围 0 .. 5 。因为在语句 for i in 0..5 将允许取值的范围从 0 到 5(含 5),

这将产生以下结果：

```
Value of local variable is 0

Value of local variable is 1

Value of local variable is 2

Value of local variable is 3

Value of local variable is 4

Value of local variable is 5
```

for...in 循环几乎是完全等同于：

```
(expression).each do |variable[, variable...]| code end
```

除了一个 for 循环不创建一个新的局部变量的范围。一个循环的表情从代码分离，保留字，一个换行符，或分号。

例子:

```
#!/usr/bin/ruby

(0..5).each do |i|
    puts "Value of local variable is #{i}"
end
```

这将产生以下结果：

```
Value of local variable is 0

Value of local variable is 1
```

```
Value of local variable is 2

Value of local variable is 3

Value of local variable is 4

Value of local variable is 5
```

（6）break 语句:

- **语法:**

```
Break
```

终止大多数内部的循环。如果调用的方法与相关块,终止块内的方法返回 nil。

实例:

```
#!/usr/bin/ruby

for i in 0..5
   if i > 2 then
      break
   end
   puts "Value of local variable is #{i}"
end
```

这将产生以下结果：

```
Value of local variable is 0

Value of local variable is 1

Value of local variable is 2
```

(7) next 语句:

❖ 语法:

```
Next
```

跳转到最内部循环的下一次迭代。如果调用块一个块内终止执行(带 yield 或调用返回 nil) 。

❖ 例子:

```
#!/usr/bin/ruby

for i in 0..5
   if i < 2 then
      next
   end
   puts "Value of local variable is #{i}"
end
```

这将产生以下结果：

```
Value of local variable is 2

Value of local variable is 3

Value of local variable is 4

Value of local variable is 5
```

(8) Redo

❖ 语法:

```
Redo
```

会重新启动启动这个最内部的循环迭代，而不检查循环条件。

✧ 会重新启动 yield or call ，如果一个块内调用。

❖ 例子:

```
#!/usr/bin/ruby

for i in 0..5
    if i < 2 then
        puts "Value of local variable is #{i}"
        redo
    end
end
```

这将产生以下结果，将执行无限循环：

```
Value of local variable is 0

Value of local variable is 0

.....................
```

（9）retry

- **语法:**

```
retry
```

如果 retry 表达出现在 rescue 子句，则从开始重新开始。

```
begin
    do_something # exception raised
rescue
    # handles error
    retry    # restart from beginning
end
```

如果出现重试迭代，块，或体内的表达，重新启动迭代调用。迭代器的参数条件将重新计算。

```
for i in 1..5
    retry if some_condition # restart from i == 1
end
```

实例:

```
for i in 1..5
    retry if    i > 2
    puts "Value of local variable is #{i}"
end
```

这将产生以下结果，将进入无限循环：

```
Value of local variable is 1

Value of local variable is 2

Value of local variable is 1

Value of local variable is 2

Value of local variable is 1

Value of local variable is 2

.....
```

8. 迭代器

迭代器是不过是所支持的集合的方法 。存储一组数据成员的对象被称为集合。在 Ruby 中，可以被称为数组和哈希集合。

迭代器返回的所有元素的集合.。

8.1. each 迭代

每个迭代器返回一个数组或哈希的所有元素。

语法：

```
collection.each do |variable|
   code
end
```

集合中的每个元素执行代码。这里收集可能是一个数组或一个 Ruby 的哈希值。

例如：

```
#!/usr/bin/ruby

ary = [1,2,3,4,5]
ary.each do |i|
   puts i
end
```

这将产生以下结果：

```
1
2
3
4
5
```

始终相关联每个迭代器块。它返回的数组的每个值块，一个接一个。该值存储在变量 i 中，然后在屏幕上显示。

8. 2. collect 迭代：

收集迭代器返回集合中的所有元素。

- **语法：**

```
collection = collection.collect
```

收集方法并不总是需要相关联块。收集方法返回整个集合，无论它是一个数组或哈希。

例如：

```
#!/usr/bin/ruby

a = [1,2,3,4,5]
```

```
b = Array.new
b = a.collect
puts b
```

这将产生以下结果：

```
1
2
3
4
5
```

注意:　收集方法是不正确的方式做阵列之间的复制。还有另一种方法称为克隆应该使用哪个复制到另一个数组的一个数组。

通常使用的收集方法，当想做些什么，每个值可用来获得新的数组。例如，该代码产生一个数组 b 中的每个值的 10 倍。

- **代码**

```
#!/usr/bin/ruby

a = [1,2,3,4,5]
b = a.collect{|x| 10*x}
puts b
```

这将产生以下结果：

```
10

20

30

40

50
```

9. 方法

9.1. 方法以及命名规则

Ruby 方法跟其他编程语言中的函数非常相似， Ruby 方法用于捆绑到一个单元中的一个或多个重复的语句。

方法名称应以小写字母开始。如果一个方法的名称以大写字母开始，Ruby 可能会认为这是一个常数，因此可以正确解析调用。

方法应该定义 Ruby 的之前调用他们，否则会引发一个异常未定义的方法调用。

9.2. 语法

（1）方法定义

```
def method_name [( [arg [= default]]...[, * arg [, &expr ]])]
    expr..
end
```

所以，可以定义一个简单的方法如下：

```
def method_name
    expr..
End
```

（2）方法的参数

可以表示方法，接受这样的参数：

```
def method_name (var1, var2)
    expr..
End
```

可以设置默认值，如果不传递所需的参数调用方法的参数将用于：

```
def method_name (var1=value1, var2=value2)
    expr..
End
```

无论何时调用方法很简单，只需写方法的名称如下：

```
method_name
```

然而，当调用一个方法带有参数，编写方法的名称以及参数，如：

```
method_name 25, 30
```

使用带参数的方法的最重要缺陷是，每当调用这些方法需要记住的参数个数。例如，如果一个方法接受三个参数传递只有两个，那么 Ruby 的将显示一条错误。

实例:

```
#!/usr/bin/ruby

def test(a1="Ruby", a2="Perl")
    puts "The programming language is #{a1}"
    puts "The programming language is #{a2}"
end
test "C", "C++"
```

Test

这将产生以下结果：

```
The programming language is C
The programming language is C++
The programming language is Ruby
The programming language is Perl
```

9.3. 从方法中返回值：

（1）默认返回值

在 Ruby 中的每一个方法返回默认值。这个返回值将是最后一个语句的值。例如：

```
def test
    i = 100
    j = 10
    k = 0
```

```
end
```

此方法被调用时，将返回的最后声明的变量 k 的值。

（2）return 语句:

Ruby 的 return 语句用于从一个 Ruby 方法返回一个或多个值。

语法:

```
return [expr[`,' expr...]]
```

如果有两个以上的表达式给出，数组包含这些值将返回值。如果没有表达式，将会是 nil 值返回。

实例:

```
return

OR

return 12

OR

return 1,2,3
```

看看这个例子：

#!/usr/bin/ruby

```
def test
    i = 100
    j = 200
    k = 300
return i, j, k
end
var = test
puts var
```

这将产生以下结果：

```
100
200
300
```

9. 4. 可变参数：

假设声明一个方法需要两个参数。每当你调用这个方法，需要随着它传递两个参数。

但是 Ruby 允许声明与可变数目的参数的方法。让我们来看看这一个示例：

#!/usr/bin/ruby

```
def sample (*test)
    puts "The number of parameters is #{test.length}"
    for i in 0...test.length
        puts "The parameters are #{test[i]}"
```

```
    end
end
sample "Zara", "6", "F"
sample "Mac", "36", "M", "MCA"
```

在这段代码中，已经声明接受一个参数测试方法示例。但是，这个参数是一个可变参数。这意味着，这个参数可以在任意数量的变量。所以上面的代码将产生以下结果：

```
The number of parameters is 3

The parameters are Zara

The parameters are 6

The parameters are F

The number of parameters is 4

The parameters are Mac

The parameters are 36

The parameters are M

The parameters are MCA
```

类方法：

类定义之外定义一个方法时，该方法被默认标记为私有。另一方面，在类定义中定义的方为默认标示公有。可以改变默认可视性和私有标记的方法，由公共或私有的模块。

每当想访问一个类的方法，首先需要实例化的类。然后，使用对象可以访问任何类的成员。

Ruby 提供了一种方法来访问的方法，没有实例化一个类。让我们来看看如何声明一个类的方法和访问：

```
class Accounts
    def reading_charge
    end
    def Accounts.return_date
    end
end
```

看看方法 return_date 声明。声明随后的一个时期，这是其次的方法名与类名。可以直接访问这个类的方法如下：

```
Accounts.return_date
```

要使用这种方法，不需要创建对象之类的帐户。

9.5. alias 语句:

方法或全局变量的别名。别名不能被定义在方法体。方法 aliase 保持当前定义的方法，即使方法是覆盖。

为全局变量（$1，$2，...）的取别名是禁止。覆盖内置的全局变量，可能导致严重的问题。

语法:

```
alias method-name method-name
alias global-variable-name global-variable-name
```

例如：

```
alias foo bar
alias $MATCH $&
```

这里我们定义了 foo 的别名 bar 和$MATCH 函数的别名$&

9.6. undef 语句:

这取消的方法定义。一个是 undef 不能出现在方法体中。

通过使用 undef 和 alias, 可以从超类独立修改类的接口，但注意到这可能被打破程序由内部自行的方法调用。

10. 练习一:控制结构+方法练习

10.1. 练习 1

❖ 练习题

写一个方法，打印出 1 到 100 间的整数。

❖ 练习要求：

一、使用 while 语句完成练习

二、使用 while 修饰符完成

三、使用 while 语句完成练习

四、使用 while 修饰符完成

五、使用 for 语句完成练习

六、使用 each 迭代器完成

10.2. 练习 2

❖ 练习题

编写程序，从 100 个随机数中找出最大值。

❖ 提示

✧ 如何生成随机数？

ruby 使用函数 rand（n）来生成随机数。

例子：a = rand(11)

结果：生成 0-10 之间的随机数 a

说明：由于 rand（int）生成的是小于 int，大于等于 0 的随机数，所以要想生成 0-10（包括 0,10）之间的随机数，int 必须是 11.

10. 3. Case 练习

随机生成 100 个成绩，根据考试成绩的大小，打印出成绩的等级。分数与等级的关系如下：

A 为 90 分以上；

B 为 80 分以上；

C 为 70 分以上；

D 为 60 分 以上；

E 为 59 分以下。

11. 程序块

11. 1. Ruby 区块的概念

在前一节我们已经看到了 Ruby 如何定义方法，在这里我们可以把一些语句，然后调用该方法。这就是类似的 Ruby 区块的概念。

- 块由大块的代码组成。
- 将名称分配给一个块。
- 块中的代码总是大括号包围 ({}).
- 一个程序块段总是调用功能块使用相同名称。这意味着，如果有一个块的名称 test ，那么使用函数 test 来调用这个块。
- 使用 yield 语句调用块。

11. 1. 1. 语法：

```
block_name{
    statement1
    statement2
    .....
}
```

在这里，将学习如何通过使用一个简单的 yield 语句调用块。还将学习使用 yield 语句

具有参数调用块。将检查的示例代码，这两种类型的 yield 语句。

11.1.2. yield 语句:

让我们来看看在 yield 语句的一个例子：

```
#!/usr/bin/ruby

def test
   puts "You are in the method"
   yield
   puts "You are again back to the method"
   yield
end
test {puts "You are in the block"}
```

这将产生以下结果：

```
You are in the method
You are in the block
You are again back to the method
You are in the block
```

（1）参数与屈服声明

也可以通过参数与屈服声明。下面是一个例子：

```
#!/usr/bin/ruby

def test
   yield 5
   puts "You are in the method test"
   yield 100
end
test {|i| puts "You are in the block #{i}"}
```

这将产生以下结果：

```
You are in the block 5

You are in the method test

You are in the block 100
```

这里的 yield 语句写到后面跟着参数。甚至可以传递多个参数。在该块中放置在两条垂直线之间的变量（| |）接收的参数。因此，在上面的代码中，yield5 语句将试块作为一个参数值 5。

现在看看下面的语句：

```
test {|i| puts "You are in the block #{i}"}
```

在这里，在变量 i 中的值为 5。现在遵守以下 puts 语句：

```
puts "You are in the block #{i}"
```

puts 语句的输出是：

```
You are in the block 5
```

如果想超过一个参数，然后 yield 语句就变成了：

```
yield a, b
```

那么块是：

```
test {|a, b| statement}
```

这些参数将用逗号隔开。

11.1.3. 块和方法:

我们已经看到了如何将一个块和方法关联。通常调用块从块具有相同名称的方法，通过使用 yield 语句。因此，编写：

#!/usr/bin/ruby

```
def test
```

```
    yield
end
test{ puts "Hello world"}
```

这个例子是最简单的方式来实现一个块。调用块 test 使用 yield 语句。

但是，如果最后一个参数的方法前面加上＆，那么可以通过一个块这种方法，此块将被分配到最后一个参数。

*和＆在参数列表中＆还在后面。

```
#!/usr/bin/ruby

def test(&block)
    block.call
end
test { puts "Hello World!"}
This will produce following result:

Hello World!
```

11.1.4. BEGIN 和 END 块

每一个 Ruby 源文件都可以声明的代码块作为文件被加载运行（BEGIN 块）后，该程序已执行完毕（END 块）。

```
#!/usr/bin/ruby

BEGIN {
   # BEGIN block code
   puts "BEGIN code block"
}

END {
   # END block code
   puts "END code block"
}
   # MAIN block code
puts "MAIN code block"
```

一个程序可能包括多个 BEGIN 和 END 块。 BEGIN 块以遇到它们的顺序执行。 END 块以相反的顺序执行。上述程序执行时，会产生以下结果：

```
BEGIN code block
MAIN code block
END code block
```

11.2. 模块和组合

11.2.1. 模块

模块是组合在一起的方法，类和常量。模块两个主要好处：

- 模块提供了一个命名空间，并避免名称冲突。

- 模块实现混合工厂。

模块定义了一个命名空间，一个沙箱中方法和常量可以自由使用，而不必担心踩到其他的方法和常数。

语法:

```
module Identifier
   statement1
   statement2
   ...........
end
```

✧ 模块的命名规则

就像被命名为类常量模块中的常量，首字母大写。

定义的方法看起来很相似，模块定义方法就像类的方法。

调用一个模块方法和类方法一样，通过模块的名称它名字前，引用一个常数使用该模块的名称和两个冒号。

```
例子:
#!/usr/bin/ruby

# Module defined in trig.rb file

module Trig
```

```
    PI = 3.141592654
    def Trig.sin(x)
    # ..
    end
    def Trig.cos(x)
    # ..
    end
end
```

我们可以定义一个函数名相同，但在不同的功能模块：

```
#!/usr/bin/ruby

# Module defined in moral.rb file

module Moral
    VERY_BAD = 0
    BAD = 1
    def Moral.sin(badness)
    # ...
    end
end
```

和类的方法一样，当在一个模块中定义的方法，指定模块名称后面跟着一个点，那么该方法的名称。

11.2.2. require 语句:

require 语句声明的是类似于 C/C++ 的 include 语句 和 Java 的 import 语句。如果有

第三个程序要使用任何定义的模块，它可以简单地使用 Ruby require 语句加载的模块文件：

语法:

```
require filename
```

在这里，它不是必需的 .rb 文件名扩展。

例如：

```
require 'trig.rb'
require 'moral'

y = Trig.sin(Trig::PI/4)
wrongdoing = Moral.sin(Moral::VERY_BAD)
```

重要: 在这里，这两个文件都包含相同的函数名。因此，这将导致在代码中的歧义，同时包括在调用程序，但的模块避免这个代码模糊，我们能够调用适当的功能模块的名称。

Ruby include 语句:

可以嵌入在一个类模块。要在一个类中嵌入模块，可以使用类中 include 语句：

语法:

```
include modulename
```

如果一个模块被定义在单独的文件 ,那么它需要包含该文件需要隐藏于公开的模块在一个类的 require 语句之前。

例子:

考虑以下模块写在 support.rb 文件。

```
module Week
    FIRST_DAY = "Sunday"
    def Week.weeks_in_month
        puts "You have four weeks in a month"
    end
    def Week.weeks_in_year
        puts "You have 52 weeks in a year"
    end
end
```

现在，可以在如下一类包括这个模块：

```
#!/usr/bin/ruby
require "support"

class Decade
include Week
    no_of_yrs=10
    def no_of_months
        puts Week::FIRST_DAY
        number=10*12
        puts number
    end
```

```
end
d1=Decade.new
puts Week::FIRST_DAY
Week.weeks_in_month
Week.weeks_in_year
d1.no_of_months
```

这将产生以下结果：

```
Sunday
You have four weeks in a month
You have 52 weeks in a year
Sunday
120
```

11.2.3. 混合类型：

通过本节之前，假设有面向对象的概念和知识。

当一个类可以从多个父类继承的特点，类应该显示多重继承。

Ruby 没有直接中 suppoprt 多继承，但 Ruby 的模块有另一个精彩使用。他们几乎消除多重继承的需要，提供了一个工厂称为混入。

混合类型给一个精彩的控制方式增加功能类。在代码中混合类，使用它的代码能进行进行交互。

让我们来看看下面的示例代码来获得混合类型了解：

```
module A
    def a1
    end
    def a2
    end
End

module B
    def b1
    end
    def b2
    end
end

class Sample
include A
include B
    def s1
    end
end

samp=Sample.new
samp.a1
samp.a2
```

```
samp.b1
samp.b2
samp.s1
```

模块 A 包括一种方法，a1 和 a2。模块 B 包括一种方法，b1 和 b2。类示例包括两个模块 A 和 B 类的样品可以访问所有四种方法，即 a1, a2, b1 或 b2。因此，可以看到这个类继承自两个模块样品。因此，可以说类的示例显示了多重继承或混入。

12. Ruby 异常

异常和执行总是被联系在一起。如果您打开一个不存在的文件，且没有恰当地处理这种情况，那么您的程序则被认为是低质量的。

如果异常发生，则程序停止。异常用于处理各种类型的错误，这些错误可能在程序执行期间发生，所以要采取适当的行动，而不至于让程序完全停止。

Ruby 提供了一个完美的处理异常的机制。我们可以在 begin/end 块中附上可能抛出异常的代码，并使用 rescue 子句告诉 Ruby 完美要处理的异常类型。

12. 1. 语法

```
begin #开始

  raise.. #抛出异常

#捕获指定类型的异常 缺省值是 StandardException
rescue [ExceptionType = StandardException]
  $! #表示异常信息
  $@ #表示异常出现的代码位置

else #其余异常
  ..
ensure #不管有没有异常，进入该代码块

end #结束
```

❖ 代码分段之：保护段

从 begin 到 rescue 中的一切是受保护的。保护段代码中可能抛出异常。

代码中 raise 子句抛出异常。

✧ 小提示：

除了代码中显示抛出的异常，还有系统抛出的运行时隐式异常哦！

一旦异常抛出，控制会传到 rescue 和 end 之间的块。

❖ 代码分段之：捕获段

rescue 子句捕获异常，捕获的方式是： 把抛出的异常与每个参数进行轮流比较。如果某个 rescue 子句异常与当前抛出的异常类型相同，则匹配成功。

✧ 小提示

如果 rescue 子句异常是当前异常类的父类，也能匹配成功。

如果异常不匹配所有指定的错误类型，我们可以在所有的 rescue 子句后使用一个 else 子句。

实例

```
#!/usr/bin/ruby
```

```
begin
    file = open("/unexistant_file")
    if file
        puts "File opened successfully"
    end
rescue
        file = STDIN
end
print file, "==", STDIN, "\n"
```

以上实例运行输出结果为。您可以看到，STDIN 取代了 file ，因为打开失败。

```
#<IO:0xb7d16f84>==#<IO:0xb7d16f84>
```

12. 2. 使用 retry 语句

您可以使用 rescue 块捕获异常，然后使用 retry 语句从开头开始执行 begin 块。

- **语法**

```
begin
    # 这段代码抛出的异常将被下面的 rescue 子句捕获
rescue
    # 这个块将捕获所有类型的异常
    retry   # 这将把控制移到 begin 的开头
End
```

❖ **实例**

```
#!/usr/bin/ruby

begin
    file = open("/unexistant_file")
    if file
        puts "File opened successfully"
    end
rescue
    fname = "existant_file"
    retry
end
```

❖ **以下是处理流程：**

打开时发生异常。

跳到 rescue。fname 被重新赋值。

通过 retry 跳到 begin 的开头。

这次文件成功打开。

继续基本的过程。

注意：如果被重新命名的文件不存在，本势力代码会无限尝试。所以异常处理时，谨慎使用 retry。

12. 3. 使用 raise 语句

您可以使用 raise 语句抛出异常。下面的方法在调用时抛出异常。它的第二个消息将被输出。

语法

```
raise
```

或

```
raise "Error Message"
```

或

```
raise ExceptionType, "Error Message"
```

或

```
raise ExceptionType, "Error Message" condition
```

第一种形式简单地重新抛出当前异常（如果没有当前异常则抛出一个 RuntimeError）。这用在传入异常之前需要解释异常的异常处理程序中。

第二种形式创建一个新的 RuntimeError 异常，设置它的消息为给定的字符串。该异常

之后抛出到调用堆栈。

第三种形式使用第一个参数创建一个异常，然后设置相关的消息为第二个参数。

第四种形式与第三种形式类似，您可以添加任何额外的条件语句（比如 unless）来抛出异常。

实例

#!/usr/bin/ruby

```
begin
    puts 'I am before the raise.'
    raise 'An error has occurred.'
    puts 'I am after the raise.'
rescue
    puts 'I am rescued.'
end
puts 'I am after the begin block.'
```

以上实例运行输出结果为：

```
I am before the raise.

I am rescued.

I am after the begin block.
```

另一个演示 raise 用法的实例：

```
#!/usr/bin/ruby

begin
```

```
    raise 'A test exception.'
rescue Exception => e
    puts e.message
    puts e.backtrace.inspect
end
```

以上实例运行输出结果为：

```
A test exception.

["main.rb:4"]
```

12.4. 使用 ensure 语句

有时候，无论是否抛出异常，您需要保证一些处理在代码块结束时完成。例如，您可能在进入时打开了一个文件，当您退出块时，您需要确保关闭文件。

ensure 子句做的就是这个。ensure 放在最后一个 rescue 子句后，并包含一个块终止时总是执行的代码块。它与块是否正常退出、是否抛出并处理异常、是否因一个未捕获的异常而终止，这些都没关系，ensure 块始终都会运行。

语法

```
begin
    #.. 过程
    #.. 抛出异常
rescue
```

```
    #.. 处理错误
ensure
    #.. 最后确保执行
    #.. 这总是会执行
end
```

实例

```
begin
   raise 'A test exception.'
rescue Exception => e
   puts e.message
   puts e.backtrace.inspect
ensure
   puts "Ensuring execution"
end
```

以上实例运行输出结果为：

```
A test exception.

["main.rb:4"]

Ensuring execution
```

12.5. 使用 else 语句

如果提供了 else 子句，它一般是放置在 rescue 子句之后，任意 ensure 之前。

else 子句的主体只有在代码主体没有抛出异常时执行。

语法

```
begin
    #.. 过程
    #.. 抛出异常
rescue
    #.. 处理错误
else
    #.. 如果没有异常则执行
ensure
    #.. 最后确保执行
    #.. 这总是会执行
end
```

实例

```
begin
  # 抛出 'A test exception.'
  puts "I'm not raising exception"
rescue Exception => e
    puts e.message
    puts e.backtrace.inspect
else
     puts "Congratulations-- no errors!"
ensure
    puts "Ensuring execution"
end
```

以上实例运行输出结果为：

```
I'm not raising exception

Congratulations-- no errors!
```

Ensuring execution

12.6. Catch 和 Throw

raise 和 rescue 的异常机制能在发生错误时放弃执行，有时候需要在正常处理时跳出一些深层嵌套的结构。此时 catch 和 throw 就派上用场了。

catch 定义了一个使用给定的名称（可以是 Symbol 或 String）作为标签的块。块会正常执行知道遇到一个 throw。

- **语法**

```
throw :lablename
#.. 这不会被执行
catch :lablename do
#.. 在遇到一个 throw 后匹配将被执行的 catch
end
```

- **或**

```
throw :lablename condition
#.. 这不会被执行
catch :lablename do
#.. 在遇到一个 throw 后匹配将被执行的 catch
end
```

- **实例**

下面的实例中，如果用户键入 '!' 回应任何提示，使用一个 throw 终止与用户的交互。

```
def promptAndGet(prompt)
    print prompt
    res = readline.chomp
    throw :quitRequested if res == "!"
    return res
end

catch :quitRequested do
    name = promptAndGet("Name: ")
    age = promptAndGet("Age: ")
    sex = promptAndGet("Sex: ")
    # ..
    # 处理信息
end
promptAndGet("Name:")
```

上面的程序需要人工交互，您可以在您的计算机上进行尝试。以上实例运行输出结果为：

```
Name: Ruby on Rails

Age: 3

Sex: !

Name:Just Ruby
```

12. 7. 类 Exception

Ruby 的标准类和模块抛出异常。所有的异常类组成一个层次，包括顶部的 Exception 类在内。下一层是七种不同的类型：

Interrupt
NoMemoryError
SignalException
ScriptError
StandardError

SystemExit

Fatal 是该层中另一种异常，但是 Ruby 解释器只在内部使用它。

ScriptError 和 StandardError 都有一些子类，但是在这里我们不需要了解这些细节。最重要的事情是创建我们自己的异常类，它们必须是类 Exception 或其子代的子类。

让我们看一个实例：

```
class FileSaveError < StandardError
    attr_reader :reason
    def initialize(reason)
        @reason = reason
    end
end
```

现在，看下面的实例，将用到上面的异常：

```
File.open(path, "w") do |file|
begin
    # 写出数据 ...
rescue
    # 发生错误
    raise FileSaveError.new($!)
end
end
```

在这里，最重要的一行是 raise FileSaveError.new($!)。我们调用 raise 来示意异常已经发生，把它传给 FileSaveError 的一个新的实例，由于特定的异常引起数据写入失败。

✧ 小提示

使用 $! 变量可以捕获抛出的错误消息。

13. 练习二:程序块+异常练习

13.1. 练习 1

❖ 练习题

程序功能:

在 myEgerb 源程序中，首先声明使用的模块名称 MyEge，然后创建 getage 函数，该方法接受一个带有年月日的字符串，取得该日期的年份。

在 myCompute.rb 源程序中，给定某人姓名与出生日期，计算该人年龄，并输出该人姓名，年龄，出生日期。

13.2. 练习 2

❖ 练习题: 异常类型匹配的处理方式

首先说明"这是一个异常处理的例子";

然后在程序中主动产生一个 StandardError 类型被 0 除的异常 ,并用 rescue 语句捕获这个异常。

最后通过 StandardError 类的对象 e 的方法 message 给出异常的具体类型并显示出来。