Ruby学习回忆

> *参考教材：《Ruby基础教程》第5版 —— [日]高桥征义 后藤裕藏著*

第一部分

第一章：Ruby初探

1. Ruby官网：http://www.ruby-lang.org/zh_cn/
2. Ruby命令的执行方法：
    a. 命令行工具中：`ruby <filename>.rb`
    b. irb：交互式Ruby运行环境（使用时需要注意中文的问题）
3. 字符串
    a. `''` 和 `""` 的区别：在表示一般字符串时，两者没有区别，不过当字符串内容含有`\n \s`等转义字符时，`''` 中的字符不会经过转义。
    b. `puts` 函数和 `print` 函数还有 `p` 函数的区别：
        - `puts` 函数会原封不动并将转义字符进行转义后输出，而且末尾在每次输出完一个参数后自动加个换行
        - `print` 函数： `puts` 函数的不加换行版
        - `p` 函数： 对内容不进行转义直接输出，该函数一般只给编程者使用
4. 方法的调用
    a. `function_name(arg1, arg2...)`：带括号的参数调用，适合参数较多的情况
    b. `function_name arg1, arg2...`：不带括号的调用，调用格式简洁明了，没有固定的要求，合适的地方用合适的方法才是最好的
5. 注释
    a. 单行注释法： `# 这里是注释`
    b. 多行注释法：

```
=begin
这里是注释
=end
```

6. 控制语句：基本四类顺序，条件，循环和异常，具体下面再写。

第二章：便利的对象

1. 数组：`[]`
2. 散列：在Java和C++中称为Map，在Python和JavaScript中称为字典，大差不差，key-value形式的键值对。

3. 正则表达式：基本格式为 `/模式/ =~ 希望匹配的字符串序列` （ `=~` 符号的使用）

第三章：创建命令

1. `ARGV` 系统变量的使用，例如： `ruby print_argv.rb 1st, 2nd` 那么 `ARGV[0] == "1st"; ARGV[1] == "2nd"` 以此类推
2. 文件的读取：开-读（写）-关三个基本处理流程，例如：

```
filename = ARGV[0]
text = File.open(filename).read
puts text
text.close
```

   哈哈，就是这么简单，不像有的语言那么复杂。说你呢C和Java。

3. 引用其他文件： `require 希望使用的库名/require_relative 希望使用的库名`，前者用来引用已存在的库，后者用来引用相对路径下的库。

第二部分

第四章：对象，变量和常量

1. 对象：数值对象，字符串对象，数组散列对象，正则表达式对象，事件对象，文件对象，符号对象，范围对象和异常对象等等等等等...
2. 类：对象的抽象层面（一般的面向对象思想哈哈哈）
3. 变量（在Ruby中变量有点特殊，小心作用域。此外，变量的命名方法和上下文共同决定了变量种类）
    a. 局部变量（local variable）：英文小写字母或_开头 `name = "哈哈哈"`
    b. 全局变量（global variable）：使用美元符 `$` 开头 `$name = "哈哈哈"`
    c. 实例变量（instance variable）：以 `@` 开头 `@name = "哈哈哈"`
    d. 类变量（class variable）：以 `@@` 开头 `@@name = "哈哈哈"`
    e. 实际上还有预定义变量和伪变量
4. 常量：使用大写英文字母开头，例如 `NAME = "哈哈哈大写版"`
5. 保留字（一个大大的表）

__LINE__	__ENCODING__	__FILE__	BEGIN	END
alias	and	begin	break	case
class	def	defined?	do	else
elsif	end	ensure	false	for
if	in	module	next	nil
not	or	redo	rescue	retry

return	self	super	then	true
__LINE__	__ENCODING__	__FILE__	BEGIN	END
undef	unless	until	when	while
yield	``	``	``	``

6. 多重赋值：就是 `a, b = 1, 2`，可以有灵活的编程方法，例如 `a, b = b, a`
7. 命名方法：驼峰表示法或者下划线表示法，都可以。不过约定俗成的是：

> ***Ruby** 中的变量名和方法名要用下划线表示法，类名和模块名要用后者。*

第五章：条件判断

1. 主要有三种语句： `if, unless, case`
2. 真假值：

条目	判断依据
真	`false` 和 `nil` 以外的所有对象
假	`false` 和 `nil`

3. 特别的 `unless`： 正好和 `if` 相对
4. `===` 和 `case` 语句： `case` 语句在判断的时候，实际上用的是 `===` 运算符。但是 `===` 可以匹配更广泛的等于，例如正则表达式是否匹配，右边的对象是否属于左边的类。当然，对于一般的数值或字符串比较和 `==` 一样效果。
5. 充当修饰符： `puts "a比b大" if a > b`，这个特性不过是为了写出更符合人类逻辑的语句。
6. **检查对象的一致性**需要使用 `.equal?` 方法来进行检查和判断。而对于对象值的相等性用 `==` 和 `.eql?` 就可以了。（注意基本数据类型的陷阱！）

第六章：循环

1. Ruby中实现循环的几（两）种方式：
    a. 循环语句
    b. 方法循环： `times\while\each\for\until\loop`
        i. `times` 方法：

```ruby
循环次数.times do |i|
  puts "#{i}"
end
```

        ii. `for` 方法：

```
for i in 1..5
end
# 或者
for item in [1, 2, 3, 4]
end
```

iii. `each` 方法：

```
[1, 2, 3].each { |item|
}
```

iv. `loop` 方法：

```
loop do
  print "Hello for ∞"
end
```

v. `while` 和 `until` 省略（和Python中的一猫猫一样）

2. 循环控制： `break` ， `next` ， `redo` 。三者含义分别是：跳出循环，跳过本次循环（等同于C中的 `continue` ）， 重新来一次这个循环。
3. 题外话 `do ~ end` 写法和 `{~}` 的区别：本质上效果一样，不过跨行写程序用前者，否则用后者。

第七章：方法

1. 方法类别：实例方法，类方法，函数式方法
2. 方法的标记法：
   a. 类名#方法名标记类的**实例方法**： `Array#each`
   b. 类名.方法名/类名::方法名标记**类方法**：
   `File.open(filename)`
3. 方法定义： `def 方法名(参数列表[=默认值]) ... end` 默认值可选是否添加。
4. 方法的返回：可省略 `return` 语句，省略后使用式子最后一个语句进行返回。
5. 带块的方法： `yield` 关键字的使用。具体见下第十一章。
6. 接收可变个数参数： `def foo(a, b, *c)` ，这样一来不可逐个指定的参数会以数组形式传给 `c` 。
7. 关键字参数： `def foo(a, b: 1, **args)` ，同样的，多余的参数会以散列的结构存在 `args` 参数列表中。反过来亦可以使用散列进行方法、
   参数指定，例如：

```
def foo(a: 1, b: 2, **args)
end
args = {a:100, b:200, c:300}
foo(args)
```

8. 在Ruby社区中有一个特殊的约定：**使用两个空白进行缩进**

第八章：类

1. 类是面向对象的基础，其重要性不言而喻。**类名必须大写！**
2. 创建类：

```
def 类名
  类的定义
end
```

3. 类的初始化：使用签名为 `initialize` 的方法。在使用 `类名.new(args)` 方式获取对象时， `new` 中的参数会原封不动传给 `initialize`
4. 存取器：在大部分面向对象的语言中称为 `getter` 和 `setter` 方法，在Ruby中由于从对象外部不便直接访问实例变量，所以需要定义相关的
   访问方法：

```
class Hello
  def initialize(name="Bob")
    @name = name
  end
  def name
    @name
  end
  def name=(value)
    @name = value
  end
end
```

   其实关于存取器不用这么麻烦，使用更为便捷的定义方法即可完成实现，下面这个写法和上面实现的功能一样。

```
class Hello
  # attr_reader :name # 只定义name实例变量的 name 方法
  # attr_writer :name # 只定义name实例变量的 name= 方法
  attr_accessor :name # 上面两个合起来
  def initialize(name="Bob")
    @name = name
  end
end
```

5. 特殊变量 `self` ，一个指向调用者本身的特殊指针， `self` 本身和局部变量形式相同。

6. 类方法：在其他语言中像Java、C++中称为静态方法（static methods），提供的是一种和类的实例无关而和类本身有关的方法集合。在Ruby中
使用以下方法定义：

```
# 第一种class << 类名 ~ end，这种比较特殊，称为单例类定义
class << Hello
  def hello(name)
    puts "#{name}"
  end
end

# 第二种def 类名.方法名 ~ end
def Hello.hello(name)
  puts "#{name}"
end

# 第三种
class Hello
  def self.hello(name)
    puts "#{name}"
  end
end
```

上面几种定义之后的方法都可以使用 `Hello.hello("Bob")` 来进行访问调用，具体使用哪一种好，还是仁者见仁智者见智，没有哪一种就一定好
哪一种就一定差的说法。

7. 常量：

```
class HelloWorld
 version = 1.0
end
p HelloCount::version
```

8. 类变量：在其它语言中称为静态变量，和对象无关，和类本身有关的变量。定义时候使用 `@@name` 定义，访问时使用 `.` 访问。
9. 方法的访问控制：在Ruby中有三种访问级别，分别是 `public` ，`protected` 和 `private` 三个。声明方法为作用域法和符号声明法两种。例如：

```ruby
def Point
  attr_accessor :x, :y
  protected :x=, :y=  # 符号声明，表示x=和y=两个方法均被设置为protected访问级别。
  # public # 不指定参数的时候，默认把下面出现的所有方法都设置为public级别。
  def initialize(x, y)
    @x, @y = x, y
  end
end
```

在Ruby中的方法默认使用 `public` 访问级别，但是 `initialize` 方法默认使用 `private` 访问级别。

10. 扩展类：两种方法，使用继承或者在原有类上添加方法。继承使用 `<` 符号实现，例如： `class MyPoint < Point` 。不过，Ruby只支持单继承。
11. `alias` 和 `undef` ：使用 `alias 别名 原名` 设置方法别名，也可以直接使用符号来实现 `alias :别名 :原名` 来实现。 `undef` 用来删除已经存在的方法，同样有直接使用方法名和符号两种方式实现。
12. 单例类：使用单例类定义可以给对象添加方法。例如：

```ruby
str1 = "Bob"
class << str1
  def hello
    "Hello, #{self}"
  end
end
p str1.hello # 这个时候对于String类的对象str1有了自己独有的hello方法
```

13. 模块：模块帮Ruby实现了多继承，模块不能拥有实例，模块不能被继承。用法如下：

```ruby
module MyModel1
  # 公用的方法
end
class MyClass
  include MyModel1 # 那么MyModel1中的方法可以在以下作用域下使用。
  include MyModel2 # 如果MyModel2中有和MyModel1中一样的方法，那么会使用MyModel2中的方法。
end
```

除此之外，模块还可以提供命名空间，例如： `Math::PI` 或者 `Math.sqrt` 方法，这种实现方式需要使用到 `module_function` 方法，例如：
`module_function :hello`

14. `extend` 方法：使用该方法可以批量定义单例方法， `obj.extend(模块名)` ，那么模块中的所有方法都可以被定义成单例方法。此外，在类中
使用 `include` 包含进来的方法定义为**实例方法**。
15. 在Ruby中的面向对象和Python一样有个特殊约定：鸭子类型（长得像鸭子，走路像鸭子，那就是鸭子）。

第九章：运算符

1. 赋值运算符，逻辑运算符和其他程序一样，不赘述。
2. 特殊的 `&` 运算符：

```
item = array && array.first # 如果array有效，那么取出array的第一个元素赋给item
item = array&.first # 安全运算符或nil检查方法调用。实现的效果和上面的一样。
```

不过上面的用法只有在Ruby2.3.0之后才可以使用！类似的方法还有： `var = var || 1` ，如果 `var` 为空，那么给一个默认值1。

3. 定义运算符：除去不能自定义的几个其他都可以，不能定义的有： `:: && || .. ... ?: not = and or` 。

> *高度的自由和灵活会带来不可预估的问题——沃兹基索德*

第十章：错误处理与异常

1. 三个语法： `begin ~ rescue[=>引用异常对象的类和变量] ~ end` 还有 `begin ~ rescue[=>引用异常对象的类和变量] ~ ensure ~ end`
还有 `raise [信息][, 异常类]` 三个，哈哈哈，没了。
2. 发生异常后Ruby自动赋值的两个系统变量： `$!` 和 `$@` 前者表示最后发生的异常对象，后者表示异常发生的位置。

第十一章：块

> *块的存在让Ruby有了很强大的功能，包括算法替换，循环遍历，隐藏常规处理等。*

1. 语法： 使用 `yield` 关键字来实现。

```
def myloop
  while true
    yield "Bob" # yield部分将会替换成块中的程序段，当然，传给yield的参数将会原封不动传给块程序段。
  end
end

myloop do |name|
  puts "Hello, world, #{name}"
end
# 上面的程序将会不停输出Hello, World,  Bob
```

2. 块传递参数：首先可以使用 `block_given?` 来判断是否给出了块程序。
3. 控制块的执行：使用 `break` 方法或者 `next` 方法以及 `redo` 方法，它们可以指定参数，如果不指定会默认返回 `nil` 。两者区别：`break` 会立刻返回调用块处，忽略掉块中的计算结果，返回指定参数或者 `nil` 。而 `next` 则会中断当前的处理，执行下面的处理，会返回 `yield` 处。 `redo` 方法会返回块的开头并重新执行，以相同的块变量进行重复处理。
4. 当然，还有将块封装成对象的操作，没看懂哈哈哈哈哈哈。
5. 关于局部变量和块变量的作用域：块内部可以使用局部变量，块内部的变量不会影响到外部，即使同名也不行。例如：

```ruby
x = 1
y = 1
array = [1, 2, 3]
array.each do |x|
  y = x
end
p [x, y] # [1, 3]
```

第一、二部分总结

最好的资料呢，来自第一手资料，所以最好的参考文件也是Ruby的官方帮助文件。帮助文件地址：https://www.ruby-lang.org/zh_cn/documentation/

几个看帮助文档的技巧:

1. 查阅类和方法时，从Ruby的核心库和标准库开始查找。
2. 使用“搜索”功能查找方法。
3. 向上追溯查找父类方法。
4. 对于API文档，没必要楞记，记住一些元信息，仅在需要的时候阅读需要的部分即可（因为很可能刚背下来的东西一觉起来就解放前了）。
