

Ruby元编程读书笔记 第一章

作者 bluetea

网站<https://github.com/bluetea>

Ruby中包含了大量的语言构件：包括对象，类，module以及实例变量(instance variable)，而元编程操纵的就是这些语言构件。

第一个概念，包含所有上面语言构件的系统成为 对象模型(object model)，在对象模型中，你可以找到，“这个方法来自哪个类” 或当我包含这个模块时会发生什么此类问题的答案。

1.2 打开类

定义一个方法，去掉字符串中的标点符号等特殊字符

原来的代码定义：

```
1 def to_alphanumeric(str)
2   str.gsub(/[\^w\s]/, '')
3 end
4
5 #单元测试
6 require 'test/unit'
7
8 class ToAlphanumericTest < Test::Unit::TestCase
9   def test_strips_non_alphanumeric_characters
10    assert_equal '3 the Magic Number ', to_alphanumeric('#3, the *Magic, Number* ?')
11  end
12 end
```

to_alphanumeric只是一个string对象的方法，没必要重新定义一个ToAlphanumericT类，可以用打开String类，并在里面加入to_alphanumeric方法，当然单元测试也需要修改的：

```
1 class String
2   def to_alphanumeric
3     gsub(/[\^w\s]/, '')
4   end
5 end
6 #单元测试
7 require 'test/unit'
8
9 class StringExtensionTest < Test::Unit::TestCase #StringExtensionTest 是自己起的名字， ::表示的是类的方法
10   def test_strips_non_alphanumeric_characters #也是自己起的名字，为了好记
11     assert_equal '3 the Magic Number ', '#3, the *Magic, Number* ?'.to_alphanumeric
12     #assert_equal 后跟想要的结果，后面跟的是用实际定义的方法来调用测试字符
13   end
14 end
```

```
bogon:object_model wangmjc$ ruby alphanumeric.rb
Loaded suite alphanumeric
Started
.
Finished in 0.001278 seconds.

1 tests, 1 assertions, 0 failures, 0 errors, 0 pendings, 0 omissions, 0 notifications
100% passed

782.47 tests/s, 782.47 assertions/s
bogon:object_model wangmjc$
```

类定义的揭秘

在ruby中定义类和其它语句没有任何区别，你也可以在定义类中放置任何代码，例如：

```
[13] pry(main)> 3.times do
[13] pry(main)*   class C
[13] pry(main)*     puts "hello"
[13] pry(main)*   end
[13] pry(main)* end
hello
hello
hello
=> 3
```

上面的代码不是定义了3个同名的类，答案是no，如下，每个建立的class C的object_id都是一样的

```
[17] pry(main)> 3.times do
[17] pry(main)*   class C
[17] pry(main)*     puts "hello"
[17] pry(main)*   end
[17] pry(main)*   p self.object_id
[17] pry(main)* end
hello
70121796603460
hello
70121796603460
hello
70121796603460
=> 3
```

第二次class C的时候已经是打开类了，例如：

```
[18] pry(main)> class D
[18] pry(main)*   def x; "x"; end
[18] pry(main)* end
=> :x
[19] pry(main)> class D
[19] pry(main)*   def y; 'y'; end
[19] pry(main)* end
=> :y
[20] pry(main)> obj = D.new
=> #<D:0x007f8d01bf2760>
[21] pry(main)> obj.x
=> "x"
[22] pry(main)> obj.y
=> "y"
```

上面的例子，第一次定义class D的时候，Ruby开始着手定义这个类，并添加了第一个x方法，第二次class D的时候，Ruby只是重新打开了它，并又添加了一个y方法。

所以说class可以创建一个不存在的类，也可以打开一个存在类，关键是class关键字，的核心任务是把你带到类的上线文中，让你可以定义其中的方法。

打开类所带来的问题

下一个打开类的机会，替换array元素值的改写，原有的如下

```
1 def replace(array, form, to)
2   array.each_with_index do |e, i|
3     array[i] = to if e == form #如果元素=from, 就替换成to
4   end
5 end
6 #单元测试
7 def test_replace
8   book_topic = ['html', 'java', 'css'] #定义一个数组
9   replace(book_topic, 'java', 'ruby') #调用一个replace方法
10  expected = ['html', 'ruby', 'css']
11  assert_equal expected, book_topic #自认为 这个是错误的，应该是等于 replace(book_topic, 'java', 'ruby')
12 end
```

改写之后

```
1 # def replace(array, form, to)
2 #   array.each_with_index do |e, i|
3 #     array[i] = to if e == form #如果元素=from, 就替换成to
4 #   end
5 # end
6 #这是改写之后的，把方法加入到了Array的类中
7 class Array
8   def replace(from, to)
9     each_with_index do |e, i|
10      self[i] = to if e == from
11    end
12  end
13 end
14
15 #单元测试
16 def test_replace
17   book_topic = ['html', 'java', 'css'] #定义一个数组
18   book_topic.replace('java', 'ruby') #调用一个replace方法
19   expected = ['html', 'ruby', 'css']
20   assert_equal expected, book_topic #book_topic的值是替换之后的
21 end
```

看似没有问题，但是运行的时候失败了如下

```
replace.rb
1 # def replace(array, form, to)
2 #   array.each_with_index do |e, i|
3 #     array[i] = to if e == from #如果元素=from, 就替换成to
4 #   end
5 # end
6 #这是改写之后的, 把方法加入到了Array的类中
7 class Array
8   def replace(from, to)
9     each_with_index do |e, i|
10       self[i] = to if e == from
11     end
12   end
13 end
14
15 #单元测试
16 require "test/unit"
17 class ArrayExtentionTest < Test::Unit::TestCase
18   def test_replace
19     book_topic = ['html', 'java', 'css'] #定义一个数组
20     book_topic.repalc('java', 'ruby') #调用一个replace方法
21     expected = ['html', 'ruby', 'css']
22     assert_equal expected, book_topic #book_topic的值是替换之后的
23   end
24 end
```

```
ruby
bagon:object_model wangmjc$ ruby replace.rb
Loaded suite replace
Started
E

Error: test_replace(ArrayExtentionTest)
: NoMethodError: undefined method `repalc' for ["html", "java", "css"]:Array
replace.rb:20:in `test_replace'
17: class ArrayExtentionTest < Test::Unit::TestCase
18:   def test_replace
19:     book_topic = ['html', 'java', 'css'] #定义一个数组
=> 20:     book_topic.repalc('java', 'ruby') #调用一个replace方法
21:     expected = ['html', 'ruby', 'css']
22:     assert_equal expected, book_topic #book_topic的值是替换之后的
23:   end

Finished in 0.002384 seconds.

-----
1 tests, 0 assertions, 0 failures, 1 errors, 0 pendings, 0 omissions, 0 notifications
0% passed
-----
419.46 tests/s, 0.00 assertions/s
bagon:object_model wangmjc$
```

错误如下:

: NoMethodError: undefined method `repalc' for ["html", "java", "css"]:Array

明明已经定义了replace函数, 经过检验如下

```
ast login: Thu Jan 15 23:11:30 on ttys002
bagon:~ wangmjc$ pry
1] pry(main)> [].methods.grep /^re/
> [:reverse_each,
:reverse,
:reverse!,
:reject,
:reject!,
:replace,
:repeated_permutation,
:repeated_combination,
:reduce,
:remove_instance_variable,
:respond_to?]

```

默认的Array已经有replace函数了, 重名了, 这就是打开类的隐患, 无意中覆盖了类中的replace定义, 这样就会遇到bug, 一些人对于这种修订类的鲁莽方式深感不爽, 管这种叫做猴子补丁(MonkeyPath).

为了修改这个问, 需要把replace方法改名为substitute()

```
replace.rb
1 # def replace(array, form, to)
2 #   array.each_with_index do |e, i|
3 #     array[i] = to if e == from #如果元素=from, 就替换成to
4 #   end
5 # end
6 #这是改写之后的, 把方法加入到了Array的类中
7 class Array
8   def substitute(from, to)
9     each_with_index do |e, i|
10       self[i] = to if e == from
11     end
12   end
13 end
14
15 #单元测试
16 require "test/unit"
17 class ArrayExtentionTest < Test::Unit::TestCase
18   def test_substitute
19     book_topic = ['html', 'java', 'css'] #定义一个数组
20     book_topic.substitute('java', 'ruby') #replace方法改为 substitute方法, 因为Array类已经有了repalc方法了
21     expected = ['html', 'ruby', 'css']
22     assert_equal expected, book_topic #book_topic的值是替换之后的
23   end
24 end
```

```
2. bash
bagon:object_model wangmjc$ ruby replace.rb
Loaded suite replace
Started
.

Finished in 0.000832 seconds.

-----
1 tests, 1 assertions, 0 failures, 0 errors, 0 pendings, 0 omissions, 0 notifications
100% passed
-----
1201.92 tests/s, 1201.92 assertions/s
bagon:object_model wangmjc$
```

1.3类的真相

```
[1] pry(main)> class MyClass
[1] pry(main)*   def my_method
[1] pry(main)*     @v = 1
[1] pry(main)*   end
[1] pry(main)* end
=> :my_method
[2] pry(main)>
[3] pry(main)> obj = MyClass.new
=> #<MyClass:0x007fbc42189040>
[4] pry(main)> obj.my_method
=> 1
```

实例变量-----对象包含了实例变量，可以调用Object#instance_variables来查看实例变量。但是看实例 obj1，当没有调用my_methods时，是不会初始化实例变量的。

```
[6] pry(main)> obj.instance_variables
=> [:@v]
[8] pry(main)> obj1 = MyClass.new
=> #<MyClass:0x007fbc423ec580>
[9] pry(main)> obj1.instance_variables
=> []
```

方法----对象除了包含实例变量，还包含方法，通过调用Object#methods方法，可以获取一个对象的列表，绝对多数的对象，都从（包含上面的obj对象）Object类中继承了一组方法，也包含后来定义的my_method方法

```
[8] pry(main)> obj.methods.grep(/^my_/)
=> [:my_method]
```

其实如果撬开Ruby解释器查看obj，会看到对象其实并没有真正存放一组方法，在对象内部，一个对象仅仅包含它的实例变量和自身类的引用（还包含一个唯一的标示符，可以通过Object#object_id获得，与 tainted或 frozen状态），可以看出这个对象是属于哪个类的

```
[11] pry(main)> obj
=> #<MyClass:0x007f959118bb08 @v=1>
[12] pry(main)> obj.object_id
=> 70140180585860
```

注意区分类的方法和类的实例方法，根本就

```
[14] pry(main)> String.instance_methods == "abc".methods
=> true
[15] pry(main)> String.methods == "abc".methods
=> false
```

一个非常重要的挂念一类的自身也是对象

一个普通的类是Class类的对象，所以既然是对象，那么适用于对象的规则也适用于类，所以类和其它对象一样，也有自己的类，叫Class

```
[16] pry(main)> "hello".class
=> String
[17] pry(main)> String.class
=> Class
```

所以，一个实例的方法也是其类的实例方法，所以这意味着一个类的方法，也是Class类的实例方法

```
[19] pry(main)> inherited = false
=> false
[20] pry(main)> Class.instance_methods(inherited)
=> [:allocate, :new, :superclass]
```

new()方法就是来创造对象的方法，allocate（）方法是new方法的支撑方法，superclass（）方法就是返回这个类的父类（也叫超类）

```
[32] pry(main)> String.allocate
=> ""
[33] pry(main)> String.superclass
=> Object
[34] pry(main)> Object.superclass
=> BasicObject
[35] pry(main)> BasicObject.superclass
=> nil
```

所以可以看出，所有的类最终都继承与Object，Object本身又继承与BasicObject，BasicObject是Ruby对象体系中的根节点，那么Class类的父类是什么呢？

```
[40] pry(main)> Class.superclass
=> Module
[41] pry(main)> Module.superclass
=> Object
```

因此一个类只是一个增强的Module，增加了3个方法——new(), allocate(), superclass()而已，这几个方法可以让人创建对象，并将对象纳入到类体系架构中，类和模块基本上是一样的，所以绝大多数适用于类的内容也适用于模块，反之亦然

回归上面的代码，会发现obj和MyClass都是引用——唯一的区别是，obj是一个变量（小写字母），MyClass是一个常量（任何大写字母开头的引用（包括类名和模块名）都是常量），换句话说，就像类名也是对象一样，只不过类这个对象是常量。

```
[44] pry(main)> class MyClass
[44] pry(main)*   def my_method
[44] pry(main)*     @v = 1
[44] pry(main)*   end
[44] pry(main)* end
=> :my_method
[45] pry(main)>
[46] pry(main)> obj = MyClass.new
=> #<MyClass:0x007f9591160f48>
[47] pry(main)> obj.my_method
=> 1
```

模块的优点：

既然类和模块那么相似（类只是比模块多了 new() allocate() superclass()方法），那么为什么要在Ruby中存在两个不同的概念呢？

同时拥有类和模块的主要原因在于清晰性，分类大概如下：

- 1.使用模块的时候，希望它在别处被包含（include）或当成命名空间时。
- 2.使用类的时候，希望它被实例化或者继承时，应该选择使用类。

常量：

任何大写字母开头的引用（包括类名和模块名）都是常量，常量的作用域不同于变量，他有自己特殊的规则。

```
test.rb
1 module MyModule
2   MyConstant = "outer constant"
3
4   class MyClass
5     MyConstant = "inter constant"
6   end
7 end
```

所有常量像文件系统一样组织成树形结构，其中模块和类像目录，常量则像文件，所以跟文件系统一样，只要不在同一个目录下，不同的文件但是名可以相同（不同目录下的常量是不一样的），可以想文件系统一样通过路径方式来引用一个变量，例如：

```
test.rb
1 module MyModule
2   MyConstant = "outer constant"
3
4   class MyClass
5     MyConstant = "inter constant"
6   end
7 end
8
9 p MyModule::MyConstant
10 p MyModule::MyClass::MyConstant
```

```
1. bash
ruby
bash
wangmjcdMBP:object_model wangmjc$ ruby test.rb
"outer constant"
"inter constant"
wangmjcdMBP:object_model wangmjc$
```

常量的路径：

```
test.rb
1 module M
2   class C
3     X = "A constant"
4   end
5   C::X # => "A constant", 只是在module内访问 class C里面的常量
6 end
7
8 p M::C::X
```

```
1. bash
ruby
bash
bash
wangmjcdMBP:object_model wangmjc$ ruby test.rb
"A constant"
wangmjcdMBP:object_model wangmjc$
```

当然也可以在class 层级下访问Module层级下的常量，需要在常量前加上一组双冒号表示跟路径，就得到了一个绝对的路径

```
test.rb
1 module M
2   class C
3     X = "A constant"
4   end
5   p C::X # => "A constant", 只是在module内访问 class C里面的常量
6 end
7
8 module M
9   Y = "another constant"
10  class C
11    p ::M::Y # => "another constant" 通过绝对路径的方式访问module内的常量
12  end
13 end
14
```

```
1. bash
ruby
wangmjcdMBP:object_model wangmjc$ ruby test.rb
"A constant"
"another constant"
wangmjcdMBP:object_model wangmjc$
```

Module 类提供了两个叫constant()的方法其中一个实例方法，另一个则是类方法，Module#constant方法，返回当前系统内的常量（所以说String之类的是常量）

```
[54] pry(main)> Module.methods.grep(/^consta/)
=> [:constants]
[62] pry(main)> Class.constants
=> []
[55] pry(main)> Module.constants
=> [:Object,
:Module,
:Class,
:BasicObject,
:Kernel,
:NilClass,
:NIL,
:Data,
:TrueClass,
:TRUE,
:FalseClass,
:FALSE,
:Encoding,
:Comparable,
:Enumerable,
:String,
:Symbol,
:Exception,
:SystemExit,
```

所有Module对象也有constant方法，例如

```
test.rb
1 module M
2   class C
3     X = "A constant"
4   end
5   p C::X # => "A constant", 只是在module内访问 class C里面的常量
6 end
7
8 module M
9   Y = "another constant"
10  class C
11    p ::M::Y # => "another constant" 通过绝对路径的方式访问module内的常量
12  end
13 end
14 p M.constants
15 p Module.constants[0..1]
```

```
1. bash
ruby
wangmjcdMBP:object_model wangmjc$ ruby test.rb
"A constant"
"another constant"
[:C, :Y]
[:Object, :Module]
wangmjcdMBP:object_model wangmjc$
```

如果想获得当前常量的路径，可以使用Module.nesting方法返回一个array，[0]是全路径，[1]是上一级，[2]是最上面一级，例如：


```
test.rb
1 module M
2   class C
3     module M2
4       p Module.nesting
5     end
6   end
7 end
8 end

1. bash
wangmjcdeMBP:object_model wangmjc$ ruby test.rb
[M::C::M2, M::C, M]
wangmjcdeMBP:object_model wangmjc$
```

修剪常量树

例如，如果从网上找到一个motd.rb文件用在控制台显示“当天的消息”，且想把这段代码集成到程序中，如何加载呢

load('motd.rb')

load方法有一个副作用，motd.rb文件很可能定义了变量和类，尽管变量在加载完成后会落在当前作用域之外，但是常量不会，这样motd.rb会通过他的常量（尤其是类名）污染当前程序的命名空间。

此时可以通过load的第二个可选参数来强化其常量仅在自身范围内有效

load("motd.rb", true)，通过这种方式ruby会创建一个匿名模块，使用它作为命名空间，来容纳motd.rb中定义的所有常量等内容，加载完成后，该模块会被销毁

require方法与load方法类似，但是它们目的不同，通过load方法可以执行代码，但是require则是用来导入类库，这就是require 没有第二个参数的原因，这些类库的类名，通常是你导入这些类库中希望得到的，因此没有理由在加载后销毁它们。

对象和类的小结：

1.对象无非就是一组实例变量外加一个指向其类的引用，对象的方法并不存在于对象的本身，而是存在于对象的类中，在类中，这些方法成为实例方法

2.类无非就是一个对象（Class类的一个对象，外加一组实例方法和一个对其超类的引用，Class类是Module类的子类，因此一个类也是一个模块（只不过多了new和superclass方法）

跟任何对象一样，类也有自己的方法（new()），这些方法都是Class的实例方法，跟其他对象一样，类必须通过引用进行访问，类的名字就是Module类的常量。

Object的类是什么？

Module的超类是什么？

Class的类是什么？

```
[3] pry(main)> Object.class
=> Class
[4] pry(main)> Module.superclass
=> Object
[5] pry(main)> Class.class
=> Class
[9] pry(main)> class MyClass
[9] pry(main)* end
=> nil
[10] pry(main)>
[11] pry(main)> a = MyClass.new
=> #<MyClass:0x007ff34c061408>
[12] pry(main)> a.instance_variable_set("@x", 10)
=> 10
[13] pry(main)> a
=> #<MyClass:0x007ff34c061408 @x=10>
[15] pry(main)> a.instance_variable_set("@x", 20)
=> 20
```

通过 object#instance_variable_set可以修改一个实例里面的实例变量

当你调用一个方法是发生了什么？

当调用一个方法时，ruby会做两件事

1.找到这个方法，——方法查找

先在对象所属的类中查找，然后在祖先链中一层一层查找（祖先链中可以包含模块），一直到BasicObject类，在Class#ancestors方法来查看祖先链，注意祖先链是类的方法

```
[23] pry(main)> Object.instance_methods.grep /^ance/
=> []
[24] pry(main)> Class.instance_methods.grep /^ance/
=> [:ancestors]
```

```
test.rb
1 class MyClass
2 end
3 class SubClass < MyClass
4 end
5 c = SubClass.new
6 p SubClass.ancestors
```

```
1. bash
wangmjcdMBP:object_model wangmjc$ ruby test.rb
[SubClass, MyClass, Object, Kernel, BasicObject]
wangmjcdMBP:object_model wangmjc$
```

当一个类（甚至是另外一个模块）中包含一个模块时，Ruby创建了一个封装该模块的匿名类，并把这个匿名类插入到祖先链中，其在链中的位置正好在它的类上方，这些封装类（wrapper）叫做包含类（include class），有时候也叫做代理类（proxy class），包含类是Ruby的一个密码，superclass方法会假装它不存在，所以在superclass中是看不到的。例如

```
test.rb
1 module M
2   def my_method
3     p "Module 方法"
4   end
5 end
6
7 class C
8   include M
9 end
10
11 class D < C
12 end
13
14 t = D.new
15 t.my_method
16 p D.ancestors
17 p D.superclass
18 p C.superclass
```

```
1. bash
wangmjcdMBP:object_model wangmjc$ ruby test.rb
"Module 方法"
[D, C, M, Object, Kernel, BasicObject]
C
Object
```

2. 执行这个方法，Ruby需要一个self的东西

探索 self：每一行代码都会在一个对象中被执行，这个对象就是所谓的当前对象，当前对象也可以用self表示，在给定时刻，只有一个对象可以充当当前对象，没有那个对象可以长期充当这个角色，特别注意：当调用一个方法时，接收者就成为self，从这一刻起，所有的实例变量都是self（接收者）的实例变量，所有没有明确执行的方法都在self调用的，当你的代码调用其它对象的方法时，这个对象就成为self，例如：

```
test.rb
1 class MyClass
2   def testing_self #一定要注意这个方法的返回值为调用的add_method的返回值
3     @var = 10
4     add_method
5     #所以最后一定要加self, 指定自己想要的返回值
6   end
7   def add_method
8     @var = @var + 1
9     "哈哈"
10  end
11 end
12 obj = MyClass.new
13 p obj.testing_self
```

```
1. bash
wangmjcdMBP:object_model wangmjc$ ruby test.rb
"哈哈"
wangmjcdMBP:object_model wangmjc$
```

应该改为如下，才能返回自己想要的


```
test.rb
1 class MyClass
2   def testing_self #一定要注意这个方法的返回值为调用的add_method的返回值
3     @var = 10
4     add_method
5     self          #所以最后一定要加self, 指定自己想要的返回值
6   end
7   def add_method
8     @var = @var + 1
9     "哈哈"
10  end
11 end
12 obj = MyClass.new
13 p obj.testing_self
```

```
1. bash
ruby
wangmjcdeMBP:object_model wangmjc$ ruby test.rb
哈哈
wangmjcdeMBP:object_model wangmjc$ ruby test.rb
#<MyClass:0x007ff590852428 @var=11>
wangmjcdeMBP:object_model wangmjc$
```

Kernel模块

Ruby中像print()方法, 你可以在任何地方调用它, 看起来好像每个对象都有print方法一样, 实际上都是Kernel模块的私有方法(注意不是instance_method, 而是private_instance_method), 秘密在于Object类密码的包含了Kernel模块, 因此Kernel就进入了每个对象的祖先链, 这样某个对象可以随意调用Kernel的方法但是为的是Kernel模块的私有实例方法, 为什么一个模块的私有方法可以被继承者访问呢??

答:

Ruby中私有方法的限制很简单, 即不能为私有方法制定一个显示的接受者(例如abc.private_method, 其中abc就是显示接受者), 就是

```
[81] pry(main)> self
=> main
[82] pry(main)> self.class
=> Object
```

说, 私有方法只能被自己以隐式的接受者(即self)的方式调用, 所以当你执行时你会发现, 当前的执行环境就在一个Object内部, 所以可以直接访问私有方法。

注意不是模块方法, 而是instance_method

```
[36] pry(main)> Kernel.private_instance_methods.grep /Apri/
=> [:printf, :print]
```

注意是public_methods而不是public_instance_methods

```
[37] pry(main)> Kernel.public_methods.grep /Apri/
=> [:printf, :print, :private_instance_methods, :private_constant, :private_method_defined?, :private_class_method, :private_methods]
```

所有如果给Kernel增加一个方法, 那么这个内核方法就对所有对象可用。

在Rubygems这个ruby的包管理器中, 有个例子

rubygems.rb里面有一行

```
test.rb
1 module Kernel
2   def gem(gem_name, Version_requirements)
3
```

这样这个gem方法可以在全局使用了。

顶级上下文

在任何时刻, 只要调用摸个对象的方法, 那么这个对象就会成为self, 那么没有调用任何方法的时候, self是谁呢, 如下

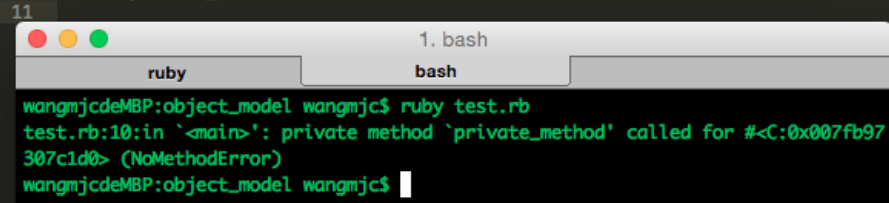
```
[2] pry(main)> self
=> main
[3] pry(main)> self.class
=> Object
[4] pry(main)>
```

当运行ruby是, ruby解释器会创建一个名为main的对象作为当前对象, 这个对象称为顶级上下文(top level context), 这个名字的来由是因为这时出在调用堆栈的顶层: 这时妖魔还没有调用任何方法, 妖魔调用的所有方法已经返回。

private方法究竟意味着什么？

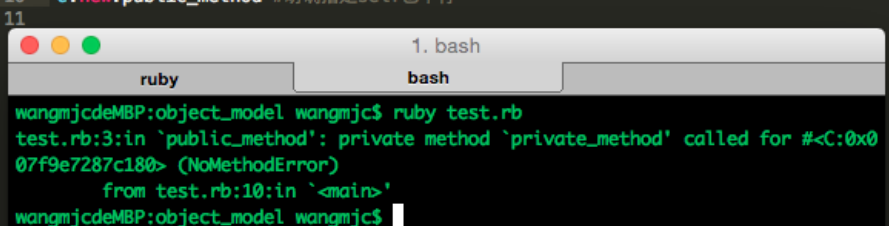
私有方法有一个简单的规则，不能明确指定一个接受者来调用一个私有方法，换言之，每次看到一个private方法只能被隐含的接受者self调用。例如：

```
1 class C
2   def public_method
3     self.private_method
4   end
5   private
6   def private_method
7     print "你调用了私有方法\n"
8   end
9 end
10 C.new.private_method #自己调用也不行
11
```



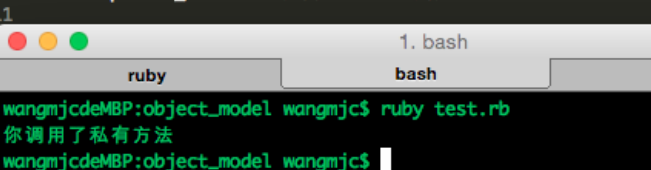
```
wangmjcdMBP:object_model wangmjc$ ruby test.rb
test.rb:10:in `<main>': private method `private_method' called for #<C:0x007fb97307c1d0> (NoMethodError)
wangmjcdMBP:object_model wangmjc$
```

```
1 class C
2   def public_method
3     self.private_method #明确指定self也不行
4   end
5   private
6   def private_method
7     print "你调用了私有方法\n"
8   end
9 end
10 C.new.public_method #明确指定self也不行
11
```



```
wangmjcdMBP:object_model wangmjc$ ruby test.rb
test.rb:3:in `public_method': private method `private_method' called for #<C:0x007f9e7287c180> (NoMethodError)
    from test.rb:10:in `<main>'
wangmjcdMBP:object_model wangmjc$
```

```
1 class C
2   def public_method
3     private_method #明确指定self也不行，删除self就行了
4   end
5   private
6   def private_method
7     print "你调用了私有方法\n"
8   end
9 end
10 C.new.public_method #明确指定self也不行
11
```



```
wangmjcdMBP:object_model wangmjc$ ruby test.rb
你调用了私有方法
wangmjcdMBP:object_model wangmjc$
```

你能调用超类的私有方法吗，当然可以，因为你无需明确指明接收者调用继承而来的方法，所以print方法就是继承自Kernel模块的私有实例方法。

对象模型的小结：

- 1.对象由一组变量和一个类的引用构成
- 2.对象的方法存在于对象所属的类中，从类的角度看，他们叫做实例方法。
- 3.类本身是Class类的对象，名字只是Module的常量而已
- 4.Class类是Module的子类，一个模块基本又一组方法构成，类除了具有模块的特性之外，可以被实例化(new()方法)，以及被组织为层次结构（通过superclass()方法实现的）
- 5.常量像文件系统一样，是按照树形结构组织的，其中模块名和类的名字扮演目录角色，常量扮演文件角色
- 6.每个类都有一个祖先链，这个链从自己所属的类开始，向上知道BasicObject类结束
- 7.每当一个类包含一个模块时，该块会被插入到祖先链中，位置在该类的正上方。
- 8.每当调用一个方法时，接受者会扮演self角色
- 9.当定义一个模块（或类）时，该模块扮演self的角色。
- 10.实例变量永远被认为self的实例变量
- 11.没有任何明确指定接受者的方法调用，都是self调用方法

