

Ruby元编程读书笔记 第四章 类

作者 bluetea

网站<https://github.com/bluetea>

在Java和C#等语言中，类的定义时是没有运行任何代码的，知道你创建了这个类的对象，然后调用对象的方法才会有实际的工作。

但是在Ruby中不一样，类的定义是不同的，当使用class关键字时，并非是在指定对象未来的行为方式，而是真真正正的执行代码。例如：

```
[1] pry(main)> class A
[1] pry(main)* puts "I'm inside a class"
[1] pry(main)* end
I'm inside a class
=> nil
```

注意：类只是一个增强版的模块（多了new等类方法），所以基本类的任何知识都适用module，所以“类定义”的内容，基本等于“模块定义”

4.1 深入类的定义

跟方法和块一样，类定义也会返回最后一条语句的值，例如

```
=> nil
[2] pry(main)> result = class MyClass
[2] pry(main)* self
[2] pry(main)* end
=> MyClass
[3] pry(main)> result
=> MyClass
[4] pry(main)>
```

在上面的例子中，类本身充当了对象self的角色，因为类和模块也是对象（Class类的对象），所以类也能是self，称为当前类

当前类

不管在ruby程序中的哪个位置，总会存在一个当前对象，self，所以也总会有一个当前类（或当前模块）存在。当定义一个方法时，该方法为当前类的一个实例方法。每当用class关键字打开一个类的时候，这个类就是当前类

```
class_example.rb
1 class MyClass
2   #现在当前类是 MyClass
3   def my_method
4     #这里的当前类也是MyClass,my_method是当前类的一个实例方法
5   end
6
7 end
```

class关键字必须知道一个类的名字才能打开类，例如想要给一个类添加一个实例方法

```
class_example.rb
1 def add_method_to(a_class)
2   #TODO: 在a_class内添加
3   a_class.class_eval do
4     def m; puts "hello"; end
5   end
6 end
7 add_method_to(String)
8 "abc".m

1. bash
ruby
bash
bagon:ruby wangmjc$ ruby class_example.rb
hello
bagon:ruby wangmjc$
```

Module#class_eval()方法或者他的别名 module_eval()会在已存在的class的上下文执行一个块，类似于用class_eval打开了原有类的上下文环境。

注意：当前类及其特殊情况

Ruby解释器总是会追踪当前类，例如：

```
class_example.rb x
1 class MyClass
2   def one
3     def two
4       puts "in two method"
5       #当前类是MyClass
6     end
7     puts "in one method"
8     #当前类也是MyClass
9   end
10 end
11 obj = MyClass.new
12 obj.one
13 obj.two
```

```
1. bash
ruby
bash
bogon:ruby wangmjc$ ruby class_example.rb
in one method
in two method
bogon:ruby wangmjc$
```

根据上面的情况在定义two的时候，当前类肯定不是self，因为self根本不是类，是一个实例对象，所以这个当前类的角色由self的类（MyClass）来充当，这样就可以理解了。

同理，在顶级作用域的时候，当前类为main对象的类Object

```
[5] pry(main)> self
=> main
[6] pry(main)> self.class
=> Object
[7] pry(main)> def my_method
[7] pry(main)* puts "当前类是main这个对象的类 Object"
[7] pry(main)* end
=> :my_method
[8] pry(main)> Object.instance_methods.grep/my_method/
[8] pry(main)* Object.instance_methods.grep /my_method/
SyntaxError: unexpected tREGEXP_BEG, expecting end-of-input
Object.instance_methods.grep /my_method/
      ^
[8] pry(main)> Object.instance_methods.grep /my_method/
=> [:my_method]
```

instance_eval和class_eval的不同

1.instance_eval仅仅会修改self

2.class_eval不仅会修改self，还会修改当前类，通过修改当前类，class_eval相当于重新打开了该类

```
class_example.rb x
1 class MyClass
2
3 end
4 MyClass.instance_eval do
5   #这种方式没有修改当前类
6   def three
7     puts "in three method"
8   end
9 end
10
11 MyClass.class_eval do
12   def four
13     puts "in four method"
14   end
15 end
16
17 obj = MyClass.new
18 p MyClass.instance_methods.grep /four/
19 p MyClass.instance_methods.grep /three/
20 obj.four
21 obj.three
22
```

```
1. bash
ruby
bash
bogon:ruby wangmjc$ ruby class_example.rb
[:four]
[]
in four method
class_example.rb:21:in `<main>': undefined method `three' for #<MyClass:0x007ffa21277d60> (NoMethodError)
bogon:ruby wangmjc$
```

所以这么看 Module#class_eval 实际比class关键字灵活，可以打开任何代表类的变量，但是class后面只能跟常量，class会打开一个新的作用域，并丧失了对当前绑定（变量等）的可见性。

如何选择instance_eval和class_eval 的选择

如果打开的是类，并用def关键字定义方法，选择class_eval方法，改变self和当前类
如果打开的是类，不想用def关键字定义方法，选择class_eval方法或instance_eval都可以
如果打开的不是类，使用instance_eval方法

所以总结一下就是，如果只修改self，那么两种都可以，但是最好的就是根据意图走，打开对象用instance_eval,打开类，用class_eval

总结类定义：

- 1.在类的定义中，当前对象self就是正在定义的类
- 2.Ruby解释器总是追踪当前类的引用，所有def定义的方法都成为当前类的实例方法
- 3.在类定义中，当前类就是self
- 4.如果有一个类的引用，可以用class_eval打开。

类实例变量

Ruby解释器假定所有的实例变量都属于当前对象self，所以在定义类的时候，当前的实例变量也是属于类的，例如

```
class_example.rb
1 class MyClass
2   @var = 1#self为当前类，所以为类实例变量
3
4   def self.read
5     @var
6   end
7
8   def write
9     @var = 2 #这个@var是类的对象的实例变量
10  end
11
12  def read
13    @var
14  end
15
16 end
17 obj = MyClass.new
18 p obj.write
19 p obj.read
20 p MyClass.read
21
```

```
1. bash
ruby
bash
bogon:ruby wangmjc$ ruby class_example.rb
2
2
1
```

上面的2个@var属于不同的作用域，并且属于不同的对象，一个属于类（类也是对象）实例变量，一个属于obj对象的实例变量

注意，关于类变量的说明

以@@开头的为类变量，类变量与类实例变量不同，他们可以被子类或类的实例所使用，而类的实例变量只可以被类本身使用，例如

```
class_example.rb
1 class C
2   @@v = 1
3 end
4
5 class D < C
6   def one
7     puts "@@v = #{@v}"
8   end
9 end
10 D.new.one
```

```
1. bash
ruby
bash
bogon:ruby wangmjc$ ruby class_example.rb
@@v = 1
bogon:ruby wangmjc$
```

但是类的实例变量有一个非常大的缺陷，例如

```
class_example.rb x
1 @@v = 1 #在顶级作用域, 实际是Object类的类变量
2 class MyClass
3   @@v = 2 #M
4 end
5 puts MyClass.superclass
6 puts @@v
```

```
1. bash
ruby bash bash
bogon:ruby wangmjc$ ruby class_example.rb
class_example.rb:1: warning: class variable access from toplevel
Object
class_example.rb:6: warning: class variable access from toplevel
2
bogon:ruby wangmjc$
```

这是因为@@类变量并不真正属于这个类, 而是属于类体系, 就如上文@@v属于main的类Object的, 所以继承自Object的所有类都可以访问。所以为了尽量避免以外, 不要用类变量

修改书中程序, 添加一个特殊类型的测试单元

```
class_example.rb x
1 class Loan
2   def initialize(book)
3     @book = book
4     @time = Time.now #每次生成都不一样
5   end
6   def to_s
7     "#{@book.upcase} loaded on #{@time}"
8   end
9 end
```

为了完成这个测试, 需要添加一个类方法self.time_class

```
class_example.rb x
1 class Loan
2   def initialize(book)
3     @book = book
4     @time = Loan.time_class.now
5   end
6   def self.time_class
7     @time_class || Time #这种方式也叫做空指针保护
8     #@time_class是类实例变量, 如果是nil就返回Time
9   end
10
11   def to_s
12     "#{@book.upcase} loaded on #{@time}"
13   end
14 end
```

其实在实际产品中@time_class总会返回nil, 所以这个time_class类方法总会返回Time, 但是在单元测试中, 我们可以我们为@time_class赋值, 这样就永远会返回同一个值, 为了给@time_class赋值, 我们需要用Instance_eval或者class_eval给这个类实例变量赋值, 例如

```
class_example.rb x
1 class Loan
2   def initialize(book)
3     @book = book
4     @time = Loan.time_class.now
5   end
6   def self.time_class
7     @time_class || Time #这种方式也叫做空指针保护
8     #@time_class是类实例变量, 如果是nil就返回Time
9   end
10   def to_s
11     "#{@book.upcase} loaded on #{@time}"
12   end
13 end
14
15 class FakeTime #因为调用的时候带now, 所以这个now必须是类方法
16   def self.now
17     '2015-01-27 21:58:51 +0800'
18   end
19 end
20 require "test/unit"
21 class TestLoan < Test::Unit::TestCase #创建一个测试类
22   def test_conversion_to_string #创建一个测试方法
23     Loan.instance_eval do #打开类, 并改变类实例变量
24       @time_class = FakeTime
25     end
26     loan = Loan.new("War And Peace") #创建一个实例
27     assert_equal "WAR AND PEACE loaded on 2015-01-27 21:58:51 +0800", loan.to_s
28     #期待相等
29   end
30 end
```

```
1. bash
ruby bash bash
bogon:ruby wangmjc$ ruby class_example.rb
Loaded suite class_example
Started
.
Finished in 0.000944 seconds.
-----
1 tests, 1 assertions, 0 failures, 0 errors, 0 pendings, 0 omissions, 0 notifications
100% passed
-----
1059.32 tests/s, 1059.32 assertions/s
bogon:ruby wangmjc$
```

小挑战, 很简单

```
class_example.rb x
1 c = Class.new(Array) do
2   def my_method
3     puts "hello"
4   end
5 end
6 MyClass = c
7 MyClass.new.my_method

1. basl
ruby bash
bogon:ruby wangmjc$ ruby class_example.rb
hello
=> String
[16] pry(main)> c = Class.new(Array) do
[16] pry(main)*   def my_method
[16] pry(main)*     puts "hello"
[16] pry(main)*   end
[16] pry(main)* end
=> #<Class:0x007fd88b379a40>
[17] pry(main)> c
=> #<Class:0x007fd88b379a40>
[18] pry(main)> a = c.new
=> []
[19] pry(main)> a = c.new.my_method
hello
=> nil
```

你生成了一个引用类的变量c，这个类是一个匿名类(因为这个类没有一个常量的名字)，但是类需要是一个常量，所以要把c赋值给MyClass

4.3单件方法

Ruby允许给单个对象增加一个方法

像上面那样，只为str添加了一个title?方法，其它对象不会得到这个方法的

关于类方法的真相

类只是对象，而类名只是常量，所以你会发现，在类上调用方法和对象调用方法是一样的

obj.my_method

MyClass.class_method

模式都是 对象（常量或变量）.方法

类方法的实质就是，他们是类的单件方法。

所以类方法的定义可以下面2种模式,还有一种后面说。

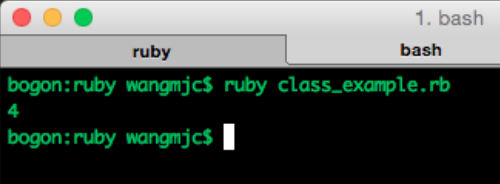
```
class_example.rb x
1 class MyClass
2   def self.class_method
3   end
4 end
5
6 class MyClass
7   def MyClass.class_method
8   end
9 end
10
```

类宏

attr_accessor的例子

Ruby对象没有属性的（外部读，写），如果想要有些像属性的东西，得定义2个拟态方法
一个读方法，一个写方法

```
1 class MyClass
2   def my_attribute=(value)
3     @my_attribute = value
4   end
5   def my_attribute
6     p @my_attribute
7   end
8 end
9 obj = MyClass.new
10 obj.my_attribute = 4
11 obj.my_attribute
12
```



```
1. bash
ruby
bogon:ruby wangmjc$ ruby class_example.rb
4
bogon:ruby wangmjc$
```

通过Module#attr_*方法可以实现定义访问器，分别为 “

Module#attr_reader 定义读方法

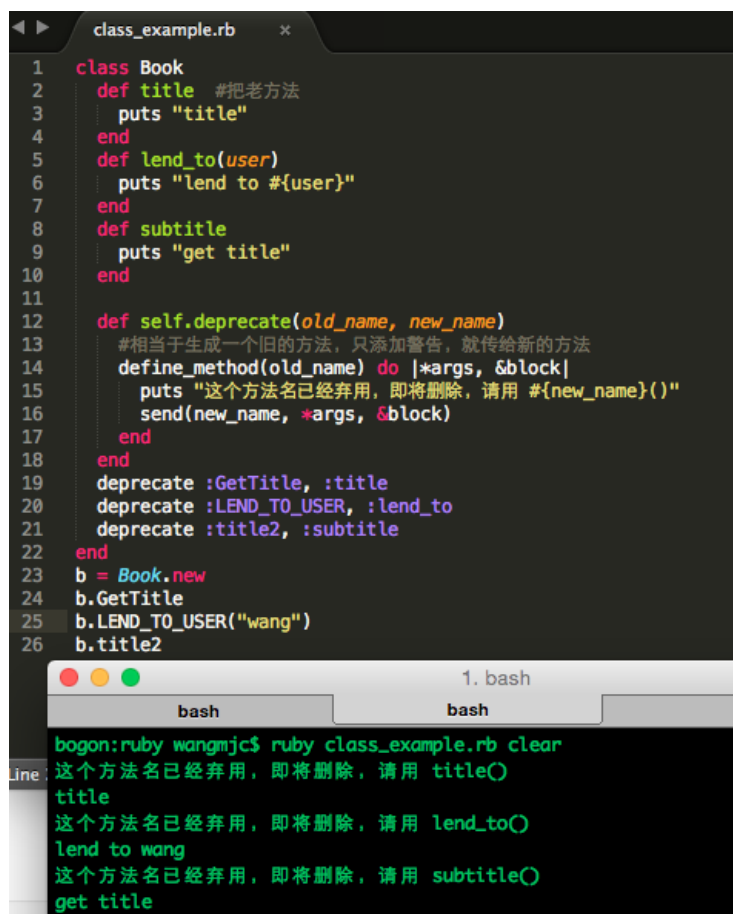
Module#attr_writer 定义写方法

Module#attr_accessor 定义读写方法

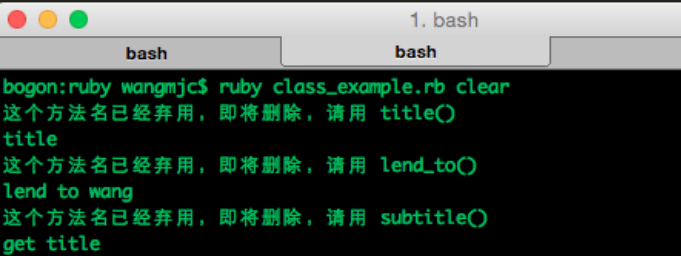
像这样的attr_*方法叫类宏(class Macro)

如何应用类宏

例如，原来的书虫的Book类，里面有不规范的命名如果 GetTitle(), title2()和LEND_TO_USER方法，如何让别人继续访问，但是访问中提示别人这个方法已经更新为新的方法了，如下：



```
class_example.rb x
1 class Book
2   def title #把老方法
3     puts "title"
4   end
5   def lend_to(user)
6     puts "lend to #{user}"
7   end
8   def subtitle
9     puts "get title"
10  end
11
12  def self.deprecate(old_name, new_name)
13    #相当于生成一个旧的方法，只添加警告，就传给新的方法
14    define_method(old_name) do |*args, &block|
15      puts "这个方法名已经弃用，即将删除，请用 #{new_name}()"
16      send(new_name, *args, &block)
17    end
18  end
19  deprecate :GetTitle, :title
20  deprecate :LEND_TO_USER, :lend_to
21  deprecate :title2, :subtitle
22 end
23 b = Book.new
24 b.GetTitle
25 b.LEND_TO_USER("wang")
26 b.title2
```



```
1. bash
bash
bogon:ruby wangmjc$ ruby class_example.rb clear
Line: 这个方法名已经弃用，即将删除，请用 title()
title
这个方法名已经弃用，即将删除，请用 lend_to()
lend to wang
这个方法名已经弃用，即将删除，请用 subtitle()
get title
```

确实很酷，一个类方法，就将旧的方法名都改了

4.4 Eigenclass

如果一个obj有一个单件方法，首先单件方法肯定不能处在obj中，因为obj是对象，不是类，无法容纳方法，

```
eigenclass.rb x
class MyClass
  def my_method
    puts self
  end
end

obj = MyClass.new
def obj.singleton_method
  puts "这是obj的 singleton method"
end

obj.singleton_method

Run eigenclass
C:\Ruby200-x64\bin\ruby.exe -e $stdout.sync=true;$stderr.sync=true;load C
这是obj的 singleton method
Process finished with exit code 0
```

但是也不能是MyClass类，因为如果是这个，其他任何对象都可以调用这个类，当然也不可能在Myclass的父类(superclass)内，所以 singleton method真正的藏身之地在如下：

当你向一个对象查询它的类的时候，ruby并没有返回一个隐藏的类，这个类称为这个对象的eigenclass (singleton 别名)。

我们可以通过ruby的特殊的基于class关键字的语法，如下

```
eigenclass.rb x
class MyClass
  def my_method
  end
end

obj = MyClass.new
puts obj #返回对象

class << obj #打开单件类
  puts self #这个是返回对象的单件类
  puts self.class #返回对象单件类的 父类
end

Run eigenclass
C:\Ruby200-x64\bin\ruby.exe -e $stdout.sync=true;$stderr.sync=true;load C
#<MyClass:0x00000002c7a590>
#<Class:#<MyClass:0x00000002c7a590>>
Class
```

根据上面得知，eigenclass是个类，一个很特殊的类，同时每个eigenclass只有一个实例（obj），但是这个单例类可以有多个单例方法，也只会生成一个实例对象，注意上图(我标清的)：

#<MyClass: 0x0000002c7a590> 表示的是 obj对象MyClass类的一个实例

#<Class: #<MyClass:0x0000002c7a590 >>,表示 这个#<MyClass: 0x0000002c7a590> （单例类）是Class类的一个实例


```
eigenclass.rb x
class MyClass
  def my_method
  end
end

obj = MyClass.new
abc = class << obj #打开单件类
  puts self
  def my_singleton_test#定义一个单例方法，这个单例方法属于eigenclass类
  end
  self
end
abc.class
def obj.my_singleton_method; end#定义另一个单例方法，这个单例方法属于eigenclass类
puts abc.instance_methods.grep(/my_/) #显示单例类的

Run eigenclass
C:\Ruby200-x64\bin\ruby.exe -e $stdout.sync=true;$stderr.sync=true;load($0=
#<Class:#<MyClass:0x00000002c89f68>>
my_singleton_test
my_singleton_method
my_method
Process finished with exit code 0
```

eigen (是本征，固有的意思)

注意：Eigenclass和instance_eval方法比较

在class_eval()方法会修改为self，和当前类

实际，在instance_eval方法会修改为self，其实也会修改当前类，只不过把当前类修改为eigenclass（实例和类的单件类），例如

```
eigenclass.rb x
s1, s2 = "abc", "def"
s1.instance_eval do
  def swooh!; puts reverse; end
end
s1.swooh!
s2.swooh! #会失败，以为你swooh!只是s1的单例方法

Run eigenclass
C:\Ruby200-x64\bin\ruby.exe -e $stdout.sync=true;$stderr.sync=true;load($0=ARG
cba
C:/Users/wangmic/Documents/ruby/eigenclass.rb:6:in `<top (required)>': undefin
from -e:1:in `load'
from -e:1:in `<main>'
Process finished with exit code 1
```

注意：类定义的语法和方式

定义类方法的方式：

```
domain.rb singleton_class.rb testevents.rb
1 #1第一种方法
2 class MyClass
3   def self.one
4   end
5 end
6 #2第二种方法
7 def MyClass.two
8
9
10 #3第三种方法
11 class MyClass
12   class << self #打开类的单例类，然后在里面定义方法
13     def three
14     end
15   end
16 end
```

eigenclass和方法的查找

在下面的例子中，#表示一个eigenclass，#obj表示obj的一个eigenclass，#C表示类C的eigenclass。

```
eigenclass.rb x
1 class C
2   def a_method
3     "C#a_method"
4   end
5 end
6
7 class D < C; end
8
9 obj = D.new
10
11 class << obj
12   def a_singleton_method
13     "obj#a_singleton_method"
14   end
15
16   puts self.superclass #obj对象的eigenclass的父类其实是D
17 end
```

Run eigenclass

C:\Ruby200-x64\bin\ruby.exe -e \$stdout.sync=true;\$stderr.sync=true;load(\$0=ARGV.shift) D

Process finished with exit code 0

从上面的图可以看出，一个D类实例对象obj，查找方法不会第一个从自己所在的类查找，而是先找自己的eigenclass类，如果找不到，然后再找自己的类D（因为D是eigenclass的父类），然后找C类，然后找Object类

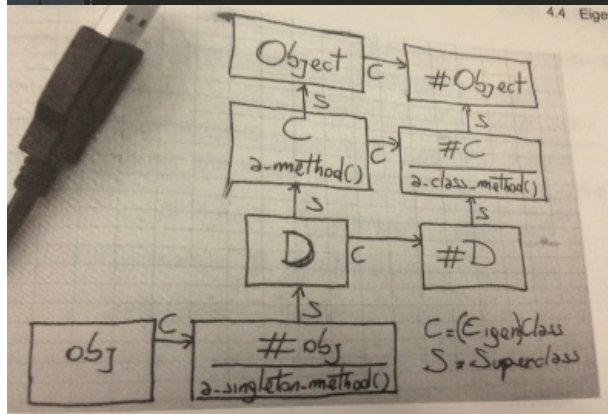
eigenclass和继承

```
eigenclass.rb x
1 class C
2   class << C
3     puts "C的实例类是 #{self}"
4     puts "C的实例类的父类是 #{self.superclass}" #类C单件类的父类是#Object (Object类的Eigenclass)
5     def d_class_method #在C类的eigenclass类中定义类的方法
6       "C.a_class_method"
7     end
8   end
9 end
10
11 class D < C
12   class << D
13     puts "D的实例类是 #{self}"
14     puts "D的实例类的父类是 #{self.superclass}" #类D单件类的父类是#C, 所以单件类也有继承关系
15     def d_class_method #在D类的eigenclass类中定义类的方法
16       "D.d_class_method"
17     end
18   end
19 end
```

Run eigenclass

C:\Ruby200-x64\bin\ruby.exe -e \$stdout.sync=true;\$stderr.sync=true;load(\$0=ARGV.shift) C:/Users/wangmjc/Documents/ruby/eigenclass

C的实例类是 #<Class:C>
C的实例类的父类是 #<Class:Object>
D的实例类是 #<Class:D>
D的实例类的父类是 #<Class:C>

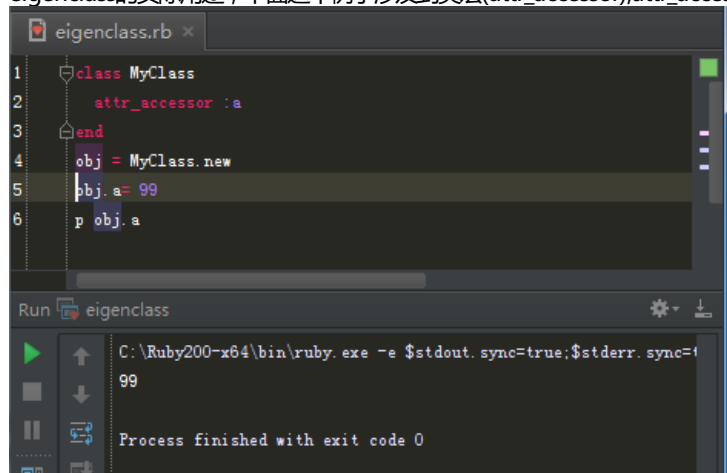


上图解释，S表示超类，C表示(Eigen)Class,注意这里的C箭头并不是class()方法返回的类，因为class()根本不知道有Eigenclass的存在，

一个句话：eigenclass的超类，就是超类的eigenclass
这样组织之后，子类就可以调用父类的类方法了（父类的类方法时定义在eigenclass内的）

类的属性

eigenclass的实际用途，下面这个例子涉及到类宏(attr_accessor),attr_accessor可以为任何对象（变量）创建属性，例如



```
1 class MyClass
2   attr_accessor :a
3 end
4 obj = MyClass.new
5 obj.a = 99
6 p obj.a
```

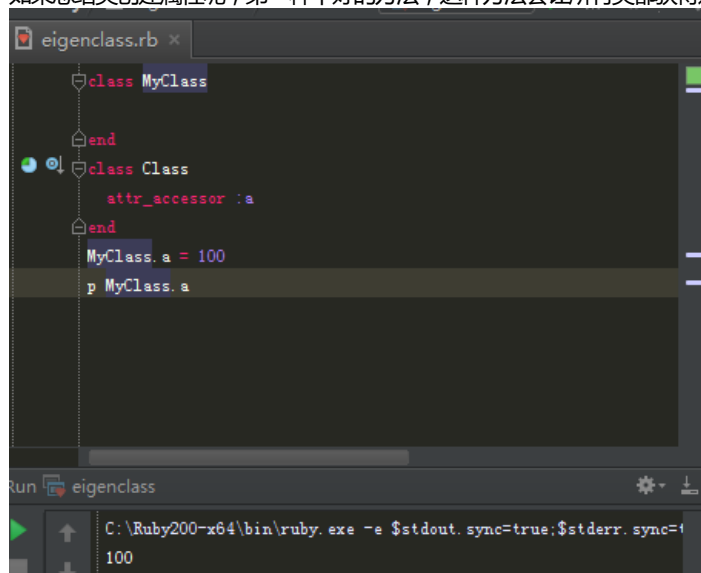
Run eigenclass

C:\Ruby200-x64\bin\ruby.exe -e \$stdout.sync=true;\$stderr.sync=true;print(\$?.exit_code)

99

Process finished with exit code 0

如果想给类创建属性呢，第一种不好的方法，这种方法会让所有类都获得这个变量



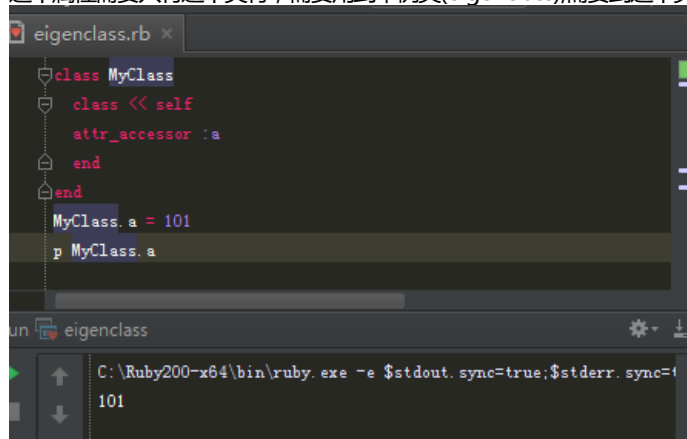
```
1 class MyClass
2 end
3 class Class
4   attr_accessor :a
5 end
6 MyClass.a = 100
7 p MyClass.a
```

Run eigenclass

C:\Ruby200-x64\bin\ruby.exe -e \$stdout.sync=true;\$stderr.sync=true;print(\$?.exit_code)

100

这个属性需要只有这个类有，需要用到单例类(eigenclass),需要到这个类的独立空间内去建立这个类的属性（类变量的方法）：



```
1 class MyClass
2   class << self
3     attr_accessor :a
4   end
5 end
6 MyClass.a = 101
7 p MyClass.a
```

Run eigenclass

C:\Ruby200-x64\bin\ruby.exe -e \$stdout.sync=true;\$stderr.sync=true;print(\$?.exit_code)

101

其实一个属性实际上是一个方法，如果在类的eigenclass添加这个方法，那么就相当于添加类方法，相当于如下代码创建类的实例变量，并给类的实例变量创建方法

```
1 class MyClass
2   class << self
3     attr_accessor :a
4   end
5 end
6 MyClass.a = 200
7 puts MyClass.a
8
9 class MyClass1
10 end
11 def MyClass1.a= (value)
12   @a = value
13 end
14 def MyClass1.a
15   @a
16 end
17 MyClass1.a= 100
18 p MyClass1.a
```

Run shuxing

```
/Users/wangmjc/.rvm/rubies/ruby-1.9.3-p5
200
100
```

另外注意的是#BasicObject (BasicObject的eigenclass) 的超类是Class类，所以说BasicObject类调用自己的类方法也是先查找#BasicObject,然后在查找Class的类。

4.5小测验，模块的麻烦

很多程序员想通过包含一个模块，来定义类方法，但是是不成功的，例如：

```
shuxing.rb x
1 def MyModule
2   def self.my_method
3     puts "this is in MyMoudle"
4   end
5 end
6
7 class MyClass
8   include MyModule #通过包含快只能获得模块的实例方法,类方法存在于模块的eigenclass内
9 end
10 MyClass.my_method
```

其实仔细想想，能够继承模块的eigenclass的为类的eigenclass，所以需要在类的eigenclass内include就可以得到 模块的类方法了,这样的话，my_method成了MyClass eigenclass的实例方法（也就是类的方法），这种技术成为类的扩展(class extension)。例如：

```
1 module MyModule
2   def my_method #就直接在模块中定义实例方法
3     puts "this is in MyMoudle\nself is #{self}"
4   end
5 end
6
7 class MyClass
8   class << self
9     include MyModule #在累的eigenclass内include 模块，模块的实例方法就会变为类方法
10   end
11 end
12 MyClass.my_method
```

Run shuxing

```
/Users/wangmjc/.rvm/rubies/ruby-1.9.3-p551/bin/ruby -e $stdout.sync=true;$s
this is in MyMoudle
self is MyClass
```

类方法和include ()

类的扩展，就是通过把模块的实例方法混合到类的eigenclass类中，就成了类方法，类方法其实就是单件方法的特例，这种方法扩展到对象上，成为对象扩展 (object extension)。

例如：

```
1 module MyModule
2   def my_method
3     puts "this is in MyMoudle\nself is #{self}"
4   end
5 end
6
7 obj = Object.new
8 class << obj
9   include MyModule
10 end
11 obj.my_method
12 puts obj.singleton_methods
```

Run shuxing

```
/Users/wangmjc/.rvm/rubies/ruby-1.9.3-p551/bin/ruby
this is in MyMoudle
self is #<Object:0x007fe87c08bf80>
my_method
```

如果认为代开eigenclass的方式太麻烦,有一种代替方式 Object#extend()方法,就是把模块的方法加载到接受者的eigenclass中,来实现对象的单例方法,和类方法.Object#extend()方法是在接受者的eigenclass中包含模块的快捷方式

```
1 module MyModule
2   def my_method
3     puts "this is in MyMoudle\nself is #{self}"
4   end
5 end
6
7 obj = Object.new
8 obj.extend MyModule
9 obj.my_method
10 puts obj.singleton_methods
```

Run shuxing

```
/Users/wangmjc/.rvm/rubies/ruby-1.9.3-p551/bin/ruby
this is in MyMoudle
self is #<Object:0x007ff4748136f0>
my_method
Process finished with exit code 0
```

当然类也一样适合

```
shuxing.rb x
1 module MyModule
2   def my_method
3     puts "this is in MyMoudle\nself is #{self}"
4   end
5 end
6
7 class MyClass
8   extend MyModule
9 end
10 MyClass.my_method
11 p MyClass.singleton_methods
```

Run shuxing

```
/Users/wangmjc/.rvm/rubies/ruby-1.9.3-p551/bin/ruby
this is in MyMoudle
self is MyClass
[:my_method]
```

4.6别名 method alias

方法别名 method alias, 一个不修改方法名, 为这个方法包装额外特性, 这样客户可以自动获得额外特性, 通过alias关键字实现, 例如

```
shuxing.rb
1 class MyClass
2   def my_method; puts "my_method----" ;end
3   alias :m :my_method
4   def m(*args, &block)
5     puts "m method ----"
6     send :my_method, *args, &block
7   end
8 end
9 obj = MyClass.new
10 obj.m

Run shuxing
/Users/wangmjc/.rvm/rubies/ruby-1.9.3-p551/bin/ruby shuxing.rb
m method ----
my_method----
my_method----
```

alias语法中，第一个出现的是新名字，第二个出现的是原始名字，最好用符号模式

注意alias是关键字，不是方法，所以 新名字和旧名字之间没有逗号，另外 Module#alias_method()方法与alias功能相同，例如：

```
1 class MyClass
2   def my_method; puts "my_method----" ;end
3   alias_method :m, :my_method
4   def m
5     puts "m method ----"
6   end
7 end
8 obj = MyClass.new
9 obj.m

Run shuxing
/Users/wangmjc/.rvm/rubies/ruby-1.9.3-p551/bin/ruby shuxing.rb
m method ----
```

在Ruby中别名非常常见，例如String#length就是String#size的别名

环绕别名

先给方法命名一个别名，然后重新定义老的方法，例如：

```
1 class String
2   alias :real_size :size
3   def size
4     real_size > 5 ? "long" : "short"
5   end
6 end
7 puts "wangxiaoming".size
8 puts "wangxiaoming".real_size

Run shuxing
/Users/wangmjc/.rvm/rubies/ruby-1.9.3-p551/bin/ruby shuxing.rb
long
12
```

这里虽然重新定义了size方法，但是别名（real_size）还是引用的原始方法，所以当重新定义一个方法时，并不是真正修改这个方法，相反只是将当前存在的方法名(size)和新的方法名(real_size)绑定起来，所以访问绑定的名字，就会访问老的方法。

环绕别名步骤：

- 1.给方法定义一个别名
- 2.重定义这个方法
- 3.在新方法中调用老方法

一个来自Rubygem的例子

```
shuxing.rb
1 module Kernel
2   alias gem_original_require require #给老的require起个新名字
3   def require(path)
4     gem_original_require path #用老的require加载
5     rescue LoadError => load_error
6     if load_error.message =~ /#{Regexp.escape(path)}\z/ and spec = Gem.searcher.find(path)
7       #生成一个\*等符号的正则表达式
8       #如果加载失败，并满足if条件，在已安装的gem中找到
9       then Gem.activate(spec.name, "= #{spec.version}")#那么也加载它
10      gem_original_require path #还是用老的require加载
11    else
12      raise load_error
13    end
14  end
15 end
```

另外一个来自JCode的例子

Jcode是在Ruby版本1.9之前，没有支持Unicode之前，处理字符的一个gem

如果想要在ruby1.8支持unicode，需要一个库，这样就可以用自己字符串类（UnicodeString）来包装ruby的字符串，UnicodeString会覆盖那些需要处理Unicode的方法（例如succ()），然后把不处理Unicode的其它方法再转发给原来的字符串处理方法，这里程序就使用了环绕比别名的方法。

```
eigenclass.rb x
1 class String
2   alias original_succ! succ!
3   private :original_succ!
4   def succ!
5     if #如果自定义了一种编码方式
6       #查找该编码下的 后继字符 (succ)
7     else
8       #否则调用原来的succ!
9       original_succ!
10    end
11  end
12 end
```

注意，同一个方法，不同别名可以有不同的可见性，别名是方法名，而非方法本身，别名可以是public，也可以是private，例如下面，可以改变新方法名字的属性，完全可以是老的方法时private，新方法时public

```
eigenclass.rb x
1 class MyClass
2   def one
3     puts "this is one"
4   end
5   private :one #源方法是私有方法
6   alias :two :one #为原始方法起个别名
7   public :two #改变新的方法的可见性
8 end
9 a = MyClass.new
10 a.two
11 a.one
```

Run eigenclass

```
C:\Ruby200-x64\bin\ruby.exe -e $stdout.
C:/Users/wangmic/Documents/ruby/eigenclass.rb
from -e:1:in `load'
from -e:1:in `'
this is one
```

注意，警告

1. 环绕别名是猴子补丁，可能会破坏现有代码，需要谨慎命名
2. 像上面的例子，Jcode改变了String#succ,但是没有重新定义String#length的方法，这样老的length方法计算unicode字节数的时候，会计算错误，在这里Jcode的解决方式是新定义了一个String#jlength的方法来计算unicode的字符个数。使用的技术越强，需要更多的测试

这里真正开始解决Amazon的问题

为Amazon#reviews_of()方法加上日志和异常处理的功能，这种环绕别名的方式可以给库里面没有异常处理的方法，添加异常处理，而不改源码。例如：

```
1 class Amazon
2   alias :old_reviews_of :reviews_of
3   def reviews_of(book) #环绕方法
4     start = Time.now
5     result = old_reviews_of(book)
6     time_taken = Time.now - start
7     puts "reviews_of() took more than #{time_taken}" #报告一下花的时间
8     result
9   rescue #增加错误处理
10     puts "reviews_of() failed"
11   end
12 end
```

4.7 小测验：打破数学规律

绝大多数的 Ruby的操作符是拟态方法 (+, -, *, %)，整数的+操作符是Fixnum#+()方法，当写1+1的时候，解析器会转换为 1.+(1)

```
irb(main):001:0> 1.+(1)
=> 2
irb(main):002:0> _
```

我们可以给Fixnum#+()重新定义的，是1+1 =3，呵呵
如下：

```
ruby > eigenclass.rb >
eigenclass.rb x
1 require "test/unit"
2 class Fixnum
3   alias :old_plus :+
4   def +(value)
5     self.old_plus(value).old_plus(1)
6   end
7 end
8 class TestPlusMath < Test::Unit::TestCase
9   def test_math_is_broken
10     assert_equal 3, 1+1
11     assert_equal 1, -1 + 1
12     assert_equal 111, 100 +10
13   end
end
Run eigenclass
C:\Ruby200-x64\bin\ruby.exe -e $stdout.sync=true;$stderr.sync=true;load
Run options:
# Running tests:
.
Finished tests in 0.004001s, 749.8125 tests/s, 2249.4376 assertions/s.
3 tests, 9 assertions, 0 failures, 0 errors, 0 skips
ruby -v: ruby 2.0.0p598 (2014-11-13) [x64-mingw32]
```

注意：

类的几乎所有概念都可以推广到模块身上，因此，有当前类也会有当前模块，当然类有eigenclass，那么module也会有eigenmodule等等。

```
eigenclass.rb x
1 module Mod
2   def one
3     "this is one"
4   end
5   module_function :one #被包含之后，会变为class的私有方法
6 end
7 class MyClass
8   include Mod
9   def cls
10     one
11   end
12 end
13 a = MyClass.new
14 p Mod.one
15 p a.cls
16 p a.one
Run eigenclass
C:\Ruby200-x64\bin\ruby.exe -e $stdout.sync=true;$stderr.sync=true;load($0=ARGV.shift) C:/Users/wangmjc/Documents/ruby/eigenclass.rb
"this is one"
"this is one"
C:/Users/wangmjc/Documents/ruby/eigenclass.rb:16:in `<top (required)>': private method `one' called for #<MyClass:0x00000002db9f78>
from -e:1:in `load'
from -e:1:in `<main>'
```