

Ruby元编程读书笔记 第二章 方法

2.2动态方法—动态派发

当你调用一个方法时，实际上是给对象发了一个消息。

什么是动态调用方法：

通常调用一个方法的时候是用.，例如：

```
test.rb
1 class MyClass
2   def my_method(arg)
3     arg * 2
4   end
5 end
6 obj = MyClass.new
7 obj.my_method(3) # => 6
```

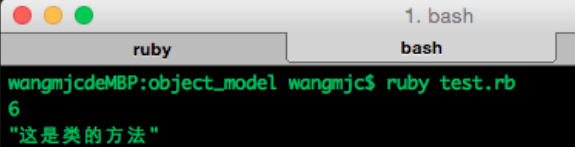
当然你可以使用Object#send方法来取代.的方法调用my_mehtod

```
[103] pry(main)> obj.methods.grep /^send/
=> [:send]
[104] pry(main)> Object.instance_methods.grep /send/
=> [:send, :public_send, :__send__]
```

```
test.rb
1 class MyClass
2   def my_method(arg)
3     arg * 2
4   end
5   def self.class_mehtod
6     p "这是类的方法"
7   end
8 end
9 obj = MyClass.new
10 p obj.send(:my_method, 3)
11 MyClass.send(:class_mehtod)
```

这次是通过send方法实现的，send()的第一个参数是要给对象发送的消息（方法），这里既可以用符号，也可以用字符串，但是一般约定俗称为用符号，其实为什么用符号，有的人会说，符号是不可变的，你可以修改字符串中的字符，但是无法对符号这么做，但是什么时候用符号，什么时候用字符串什么时候用符号，基本是依照惯例进行选择，在绝大多数的情况下，符号用于表示事物的名字，尤其是跟元编程相关的名字，例如方法名。

```
1 class MyClass
2   def my_method(arg)
3     arg * 2
4   end
5   def self.class_mehtod
6     p "这是类的方法"
7   end
8 end
9 obj = MyClass.new
10 p obj.send("my_method", 3)
11 MyClass.send("class_mehtod")
```



```
1. bash
ruby
bash
wangmjcdMBP:object_model wangmjc$ ruby test.rb
6
"这是类的方法"
```

例如：

其实问题是，有的调用方法，为什么要用send的方式呢？

因为通过send方法，你想调用的方法可以成为一个参数，这样可以在代码运行期间，直到最后一刻才决定用哪个方法，这种技术成为**动态派发**

用法举例：

1.例如一个来自 Camping的例子，是一个极简主义的ruby web框架，一个camping应用的配置信息用 键/值对 的方式存储在YAML格式文件中，YAML是一个非常简单且流行的序列化格式。

一个博客应用程序的配置文件可能像下面：

```
blog.yaml
1 admin : Bill
2 title : Rubyland
3 topic : Ruby and more
```

假设将上面的内容存入conf对象中，在理想状态中，配置信息的代码应该像下面这样：

```
test.rb
1 conf.admin = "Bill"
2 conf.title = "Rubyland"
3 conf.topic = "Ruby and more"
```

但是问题是，camping的源码中不可能有这样的代码，这是因为Camping不可能事先知道特定应用中有哪些键值，一次无法知道应该去调用哪个方法（赋值方法），只能在运行时才能根据YAML文件内容，发现给定的键值，所有Camping用了动态派发技术，根据键/值组构建出赋值方法（例如admin=（）方法），并把这个方法发送给conf对象

```
test.rb
1 if conf.rc and File.exist?(conf.rc)
2   YAML.load_file(conf.rc).each do |k, v|
3     conf.send("#{k}=", v)
4   end
5 end
6 #真是超级漂亮

[112] pry(main)> require "yaml"
=> true
[113] pry(main)> YAML.load_file("blog.yaml")
=> {"admin"=>"Bill", "title"=>"Rubyland", "topic"=>"Ruby and more"}
```

2.来自Test::Unit的例子

Test::Unit标准库使用一个命名惯例来判断哪些方法是测试方法，一个TestCase对象会用public_instance_methods查找自己的公开方法，并选择其中以test_开头的方法：例如：

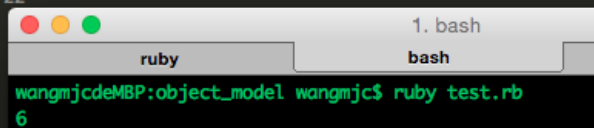
```
test.rb
1 method_names = public_instance_methods
2 tests = method_names.delete_if{|name| name !~ /^test./} #非test.开头的方法都删除
```

这样可以匹配出所有的以test开头的测试方法的数组。现在这个TestCase对象得到了测试方法的数组，然后使用send()方法来调用数组中的每个方法，动态派发这种特殊用法，又是会被称为**模式派发（Pattern Dispatch）**，因它基于方法名的某种模式来过滤方法。

动态定义方法

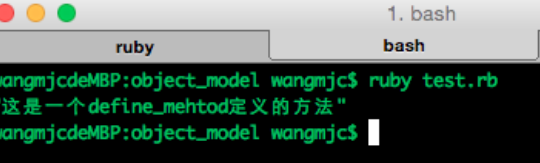
可以利用Module#define_method()方法定义一个方法，只需为其提供一个方法名和一个充当方法主体的块（可以带参数的块，也可以不带参数），例如：

```
test.rb
15 class MyClass
16   define_method :my_method do |arg|
17     arg*3
18   end
19 end
20 obj = MyClass.new
21 p obj.my_method(2)
22
```



如果不跟参数的block，则这个定义的方法是不用传参数的，例如：

```
test.rb
15 class MyClass
16   define_method :my_method do
17     p "这是一个define_mehtod定义的方法"
18   end
19 end
20 obj = MyClass.new
21 obj.my_method
22
```



经过对define_method（Module的私有方法）的查询，一个例子很引人注意原句的解释是：

define_method(symbol, method) → symbol

[click to toggle source](#)

define_method(symbol) { block } → symbol

Defines an instance method in the receiver. The *method* parameter can be a Proc, aMethod or an UnboundMethod object. If a block is specified, it is used as the method body. This block is evaluated using *instance_eval*, a point that is tricky to demonstrate because *define_method* is private. (This is why we resort to the *send* hack in this example.)

为receiver定义一个实例方法，这个method参数可以是一个proc（&block），一个method或者一个UnboundMethod对象（这个后意思的东西，相当于可以临时保存一个方法，是什么时候用什么时候绑定），如果指定block，那么这个block就当做方法的内容，这个block是用instance_eval实现的，这个例子有个小的hack手段，define_method是私有方法，所以依靠send方法来实现

```

1 class A
2   def fred
3     puts "In A's Fred"
4   end
5   def create_method(name, &block) #可接受一个block
6     self.class.send(:define_method, name, &block) #这个block会作为方法的body
7   end
8   define_method(:wilma) { puts "Charge it!" }
9 end
10 Aunmethod = A.instance_method(:fred) #暂存一个A的fred方法,存在全局变量内, 类型Unbound_method
11 class B < A
12   def fred
13     puts "In B's Fred"
14   end
15   define_method(:barney, instance_method(:fred)) #创建一个unbound_method的对象, 并创建一个实例方法
16   define_method(:barney1, Aunmethod) #相当于用暂存的Unbound_method对象, 创建了一个实例方法
17 end
18 a = B.new
19 a.barney
20 a.barney1
21 a.wilma
22 a.create_method(:betty) { p self }
23 a.betty

```

```

1. bash
ruby      bash      bash

wangmjcdeMBP:object_model wangmjc$ ruby test.rb
In B's Fred
In A's Fred
Charge it!
#<B:0x007fbcbb02ac18>
wangmjcdeMBP:object_model wangmjc$

```

关于unbound_method介绍如下：

<http://www.ruby-doc.org/core-2.2.0/UnboundMethod.html>

所以根据 send()和define_method()来重构原来的computer类

第一步：增加动态派发 send()

```

test.rb
1 class Computer
2   def initialize(computer_id, date_source)
3     @id = computer_id
4     @date_source = date_source
5   end
6
7   def mouse
8     component :mouse #相当于componet("mouse")
9   end
10  def cpu
11    component :cpu
12  end
13  def keyboard
14    component :keyboard
15  end
16  def component(name)
17    info = @date_source.send("get_#{name}_info", @id)
18    price = @date_source.send("get_#{name}_price", @id)
19    result = "#{name.to_s.capitalize}: #{info} (${price})"
20    return "* #{result}" if price >= 100
21    result
22  end
23 end

```

第二步：动态创建方法：

使用define_method再次重构函数

```

1 class Computer
2   def initialize(computer_id, date_source)
3     @id = computer_id
4     @date_source = date_source
5   end
6
7   def self.define_component(name)
8     define_method("#{name}") do
9       info = @date_source.send("get_#{name}_info", @id)
10      price = @date_source.send("get_#{name}_price", @id)
11      result = "#{name.to_s.capitalize}: #{info} (${price})"
12      return "* #{result}" if price >= 100
13      result
14    end
15  end
16  define_component("mouse")
17  define_component :cpu
18  define_component :keyboard
19 end

```

注意：define_method是类的方法，执行与Computer的类定义中，此时Computer类是当前隐式self,因为define是私有方法，无法指

明接受对象

第三步：用内省的方式缩减代码

```
class Computer
  def initialize(computer_id, date_source)
    @id = computer_id
    @date_source = date_source
    date_source.methods.grep(/get_(.+)_info/) {define_component $1} #将每个捕获的送给后面的block
    #这样就可以自动创建所有捕获的实例方法
  end

  def self.define_component(name)
    define_method("##{name}") do
      info = @date_source.send("get_#{name}_info", @id)
      price = @date_source.send("get_#{name}_price", @id)
      result = "#{name.to_s.capitalize}: #{info} ($#{price})"
      return "* #{result}" if price >= 100
      result
    end
  end
end
```

字initialize中新添加一行就是奇迹的发生之处，Array#grep每个满足正则表达式的元素都会执行后面的block，通过\$1捕捉的传给define_component方法。

2.3 method_missing()方法

```
irb(main):003:0> class Lawyer; end
=> nil
irb(main):004:0> a = Lawyer.new
=> #<Lawyer:0x000000032b0720>
irb(main):005:0> a.talk_simple
NoMethodError: undefined method `talk_simple' for #<Lawyer:0x000000032b0720>
   from (irb):5
   from C:/Ruby200-x64/bin/irb:12:in `<main>'
irb(main):006:0>
```

当调用一个方法时，ruby会到这个对象的类中查找，没有的话沿着祖先链查找，一直到Object类，并最终来到Kernel模块，由于ruby在哪个都没有找到这个方法，a对象上调用一个名为method_missing()方法，Ruby知道总会存在一个method_missing方法，因为它是Kernel的一个实例方法，而所有对象都继承自Kernel模块

用send测试一下method_miss方法:

```
irb(main):024:0> nick.send(:method_missing, :my_test)
NoMethodError: undefined method `my_test' for #<Lawyer:0x0000000375ee20>
   from (irb):24
   from C:/Ruby200-x64/bin/irb:12:in `<main>'
```

Kernel#method_missing方法会抛出一个NoMethodError进行响应，这就是他的全部功能，所有无法投递的消息最后都会来这里覆写 method_missing方法

```
text.rb
1 class Lawyer
2   def method_missing(method, *args)
3     puts "You called: #{method} (#{args.join(',')})"
4     puts "(You also passed it a block)" if block_given?
5   end
6 end
7 bob = Lawyer.new
8 bob.talk_simple("a", "b") {puts "this is a block"}
```

管理员: Start Command Prompt with Ruby

```
C:\Users\wangmjc\Documents\ruby>ruby text.rb
You called: talk_simple (a,b)
(You also passed it a block)
```

幽灵方法

当需要定义很多类似的方法时，可以通过响应method_missing方法来免去手工定义这些方法，有点像为这个对象顶一个默认方法 “当别人问你一些你不理解的东西，就这样做 ”

一个来自Ruport的例子，Ruport是一个Ruby的报表

```
text.rb
1 require 'ruport'
2
3 table = Ruport::Date::Table.new (:column_names => ["country", "wine"],
4                                   :data => [ ["France", "Bordeaux"], ["Italy", "Chianti"], ["France", "Chablis"] ])
5 puts table.to_texty
```

会输出一个列表如下

Country Wine

France Bordeaux
Italy Chianti
France Chablis

但是如果你仅仅选择法国红酒的数据，并转换为逗号间隔的数据

```
1 require 'ruport'
2
3 table = Ruport::Date::Table.new(:column_names => ["country", "wine"],
4                                 :data => [["France", "Bordeaux"], ["Italy", "Chianti"], ["France", "Chablis"]])
5 puts table.to_texty
6
7 found = table.rows_with_country("France")
8 found.each do |row|
9   puts row.to_csv
10 end
```

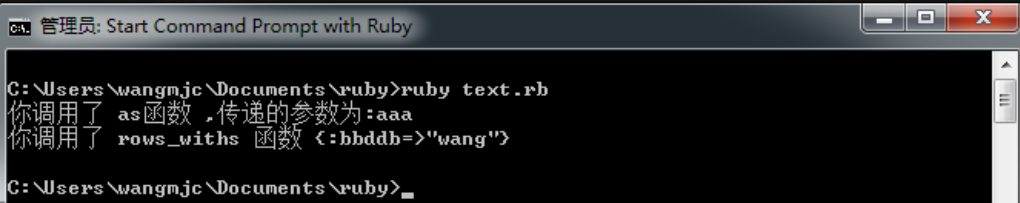
你调用了`rows_with_country`的方法，但是作者怎么知道你要用这个方法和`to_csv`方法呢，其实都有用幽灵方法实现的

```
1 class Table
2   def method_missing(method_name, *args, &block)
3     return as($1.to_sym, *args, &block) if method_name.to_s =~ /to_(.*)/ #将捕获的关键字转成符号，传给as方法，生成相应的方法
4     return rows_with(&1.to_sym => args[0]) if method_name.to_s =~ /^rows_with_(.*)/ #将捕获的关键字转成符号与args[0]形成一个hash传给rows_with函数
5     super #如果调用的函数不是上面2个，则转给Kernel#method_missing方法，这个就是super关键字做的事
6   end
7 end
```

所以根据对上面代码的分析，最根本的理念就是用`method_missing`捕获感兴趣的字符，然后传给定义好的方法，其实没有定义新的方法，但是对外界来看，感觉好像有这个`rows_with_country`方法一样。

所以根据对上面的理解，自己写个代码测试。

```
1 class MyClass
2   def as(name)
3     puts "你调用了 as函数 ,传递的参数为:#{name}"
4   end
5
6   def rows_withs(**options)
7     puts "你调用了 rows_withs 函数 #{options}"
8   end
9
10  def method_missing(method_name, *args, &block)
11    return as($1) if method_name.to_s =~ /^to_(.*)/
12    return rows_withs($1.to_sym => args[0]) if method_name.to_s =~ /^rows_withs_(.*)/
13    super
14  end
15 end
16
17 a = MyClass.new
18 a.to_aaa
19 a.rows_withs_bbbdb("wang")
20
```



在ruby的内置标准库中，`method_missing`的也广泛应用

`Openstruct`是一个ruby内置的标准库，使用起来很神奇，对象的属性用起来就像是ruby的变量，如果想要一个新的属性，就直接给这个属性赋值即可，然后还可以读出来，例如：

```
1 require "ostruct"
2 icecream = Openstruct.new
3 icecream.flavor = "strawberry"
4 icecream.flavor # => "strawberry"
5
```

实现如下：

```

text.rb
1 class OpenStruct
2   def initialize
3     @attributes = Hash.new
4   end
5   def method_missing(method_name, *args, &block)
6     att = method_name.to_s
7     if att =~ /=$/
8       @attributes[att.chop] = args[0]
9     else
10      print @attributes[att]
11    end
12  end
13 end
14
15
16 icecream = OpenStruct.new
17 icecream.flavor = "strawberry"
18 icecream.flavor # => "strawberry"
19

```

动态代理：

代理：

可以使用delegate库来快速得到一个实用的动态代理

```

delegate.rb
1 require "delegate"
2
3 class Assistant
4   def initialize(name)
5     @name = name
6   end
7
8   def read_email
9     p "#{@name} It's mostly spam"
10  end
11
12  def check_schedule
13    p "#{@name} You have a message today"
14  end
15
16 end
17
18 class Manager < DelegateClass(Assistant) #创建一个新的class, Manager继承自这个新的class
19   def initialize(assistant)
20     super(assistant)
21   end
22
23   def attend_meeting
24     p "please hold my calls"
25   end
26 end
27
28 frank = Assistant.new("frank")
29 anne = Manager.new(frank)
30 anne.attend_meeting
31 anne.read_email
32 anne.check_schedule

```

```

1. bash
bash
bash
bogon:object_model wangmjc$ ruby delegate.rb
"please hold my calls"
"frank It's mostly spam"
"frank You have a message today"
bogon:object_model wangmjc$

```

DelegatClass()是一种拟态方法（Mimic Method），这种方法创建并返回一个新的class，这个类会定义一个method_missing方法，并对对它发生的调用转发到被封装的对象(Assistant 类)上，例如本例中的Assistant类，Manger类会继承这个method_missing方法，因此它就成为被封装的对象的一个代理(Assistant 类)，Manager类是Assistant类的代理，结果Manager就会把自己无法识别的方法转发给它封装的Assistant，这样Manger和Asstiant没有继承关系，但是Manager的对象有了Assistant的方法，简单的解释就是把被调用的方法都上传送到本来的对象类那里去


```
delegate.rb
6   end
7
8   def read_email
9     p "#{@name} It's mostly spam"
10  end
11
12  def check_schedule
13    p "#{@name} You have a message today"
14  end
15
16  end
17
18  class Manager < DelegateClass(Assistant) #创建一个新的class, Manager继承自这个新的class
19    def initialize(assistant)
20      super(assistant)
21    end
22
23    def attend_meeting
24      p "please hold my calls"
25    end
26
27    def all
28      read_email
29      check_schedule
30      attend_meeting
31    end
32  end
33
34  end
35  anne = Manager.new(Assistant.new("frank"))
36  anne.attend_meeting
37  anne.read_email
38  anne.check_schedule
39  anne.all

```

```
1. bash
cc1      ruby
bogon:object_model wangmjc$ ruby delegate.rb
"please hold my calls"
"frank It's mostly spam"
"frank You have a message today"
"frank It's mostly spam"
"frank You have a message today"
"please hold my calls"
bogon:object_model wangmjc$

```

```
[15] pry(main)> frank = Assistant.new("frank")
=> #<Assistant:0x007f9482accc68 @name="frank">
[16] pry(main)> anne = Manager.new(frank)
=> #<Assistant:0x007f9482accc68 @name="frank">
[17] pry(main)> anne
=> #<Assistant:0x007f9482accc68 @name="frank">

```

这里anne的对象类型实际是Assistant类型的

重构Computer类

在这里Computer类实际只是一个包装器，它收集方法调用，把他们加工后发给数据源，为了消除重复代码，把computer类转换成一个动态代理

```
text.rb
1  class Computer
2    def initialize(computer_id, data_source)
3      @id = computer_id
4      @ds = data_source
5    end
6
7    def method_missing(method_name, *args)
8      super if !ds.respond_to?("get_#{method_name}_info")#检测一下是否这个对象有这个method
9      info = @ds.send("get_#{method_name}_info", @id)
10     price = @ds.send("get_#{method_name}_price", @id)
11     result = "#{name.to_s.capitalize}: #{info} (${price})"
12     return "" #result if price >= 100
13     result
14   end
15 end
16

```

当调用Computer#mouse方法时，这个调用会被传到method_missing方法，然后检测一下这个对象是否有这个get_mouse_info方法，如果不存在，则会被转发会 Kernel#method_missing，然后抛出一个NoMethodError错误，如果数据源知道这个方法，那么最初的调用会被转化为ds.get_mouse_info和ds.get_mouse_price两个方法调用，然后组成最终的结果

覆写respond_to?方法，需要注意的是 这个幽灵方法不是真正的方法，通过Object#methods获得的方法列表中，也无法找到，所以需要覆写respond_to?()方法来实现让respond_to?能显示这个幽灵方法。

如下：

```

text.rb
1 class Computer
2   def initialize(computer_id, data_source)
3     @id = computer_id
4     @ds = data_source
5   end
6
7   def method_missing(method_name, *args)
8     super if !ds.respond_to?("get_#{method_name}_info") #检测一下是否这个对象有这个method
9     info = @ds.send("get_#{method_name}_info", @id)
10    price = @ds.send("get_#{method_name}_price", @id)
11    result = "#{name.to_s.capitalize}: #{info} (${price})"
12    return "#{result}" if price >= 100
13    result
14  end
15
16  def respond_to?(method)
17    @ds.respond_to?("get_#{method_name}_info") || super
18  end
19 end

```

在respond_to?方法中对super语句的调用是为了保证在查询其他方法时，会调用默认的respond_to?方法，

const_missing方法

当引用一个不存在的一个常量的时候，Ruby将把这个常量作为一个符号传递给const_missing方法

```

irb(main):018:0> puts AAA
NameError: uninitialized constant AAA
from <irb>:18:
from C:/Ruby200-x64/bin/irb:12:in `<main>'

```

可以在一个命名空间（类或模块）中定义const_missing方法，如果在Object类中定义它，那么所有的对象都会继承这个方法

```

text.rb
1 def Object.const_missing(name)
2   name.to_s.downcase.gsub(/_/,"")
3 end
4
5 puts AAA_BBB

```

管理员: Start Command Prompt with Ruby

```

C:\Users\wangmjc\Documents\ruby>ruby text.rb
aaabbb
C:\Users\wangmjc\Documents\ruby>

```

对重构的小结：

第一种方法：动态（创建）方法和动态派发

第二种方法：幽灵方法

小测验：

每个成员都会随机获得一个数字，哪个人的数字最小，谁买早餐，用method_missing方法写

```

text.rb
1 class Member
2   def method_missing(name, *args)
3     person_name = name.to_s.capitalize
4     3.times do
5       number = rand(10) + 1
6       puts "Number is :#{number}"
7     end
8     puts "#{person_name} got a #{number}" #这个number作用域在block的，这里不知道number变量的
9     #这样会把number当成一个在self上商旅了括号的方法调用，类似self.number
10  end
11 end
12
13 number_of = Member.new
14 number_of.wang

```

看似ok，但是会出现死循环，因为number的作用域只是在 block中，所有相当于self调用了number方法，当然没有这个number方法，通常情况下，你会看到一个明显的NoMethodError错误，但是你覆写了method_missing方法，所以又回到了method_missing方法，如此循环，直到堆栈溢出为止。


```
text.rb
1 class Member
2   def method_missing(name, *args)
3     person_name = name.to_s.capitalize
4     number = 0 #让number的作用域在块外
5     3.times do
6       number = rand(10) + 1
7       puts "Number is :#{number}"
8     end
9     puts "#{person_name} got a #{number}" #这个number作用域在block的，这里不知道number变量的
10    #这样会把number当成一个在self上商旅了括号的方法调用，类似self.number
11  end
12 end
13
14 number_of = Member.new
15 number_of.wang
```

```
C:\Users\wangmjc\Documents\ruby>ruby text.rb
Number is :5
Number is :1
Number is :1
Wang got a 1

C:\Users\wangmjc\Documents\ruby>
```

这是一个使用幽灵方法时常出现的问题，：由于调用未定义的方法会导致调用method_missing方法，对象可能会一次接受一个错误的方法调用(例如输入错误)，导致死循环，这在大型项目中很难查找到

所以在使用幽灵方法时，应该尽在必要时才使用幽灵方法，例如仅仅接受制定的范围，其他的交给super (Kernel#method_missing) 处理。

修改如下：

```
text.rb
1 class Member
2   def method_missing(name, *args)
3     person_name = name.to_s.capitalize
4     number = 0 #让number的作用域在块外
5     3.times do
6       number = rand(10) + 1
7       puts "Number is :#{number}"
8     end
9     puts "#{person_name} got a #{number}" #这个number作用域在block的，这里不知道number变量的
10    #这样会把number当成一个在self上商旅了括号的方法调用，类似self.number
11  end
12 end
13
14 number_of = Member.new
15 number_of.wang
```

```
C:\Users\wangmjc\Documents\ruby>ruby text.rb
Number is :5
Number is :1
Number is :1
Wang got a 1

C:\Users\wangmjc\Documents\ruby>
```

2. 5 另一种method_missing方法的陷阱

当一个幽灵方法和 真实方法发生名字冲突时，后者胜出，如果不需要那个继承来的方法（刚才那个真实方法），则可以通过删除它来解决这个问题，为了安全，你也应该在代理类中删除绝大多数继承来的方法，这就是所谓的白板，它所拥有的方法比Object类还要少。

1.可以通过Module#undef_method()方法，它会删除指定的方法，保护父类里面的相同方法都会删除（包含继承的来的），也就是说在整个祖先链都找不到这个方法了

2.或者Module#remove_method()方法，它只会删除接受者自己类的这个方法，如果父类里面有，还是可以调用的

来自Builder的例子：

builder库是一个XML生成器，可以通过调用Builder::XmlMarkup方法创建xml标签

例如：

```
File Edit Selection Find View Goto Tools Project Preferences Help
text.rb
1 require "builder"
2 xml = Builder::XmlMarkup.new(:target => STDOUT, :indent=> 2)
3
4 xml.coder {
5   xml.name "Matsumoto", :nickname => "Matz"
6   xml.language "Ruby"
7 }
8 # => <coder> <name nickname = "Matz">Matsumoto</name> <language>Ruby</language>
```

这里Builder通过Ruby的语法来支持套嵌标签，属性及其他XML的特性，Builder的核心思想很简单：像name() 和 language()的方法会被XmlMarkup#method_missing() 方法实现的，每一种调用会产生一个XML标签：

例如生成一个这样的标签来描述一个大学课程

<semster> <class>Egyptology</class> <class>Ornithology</class>

```
text.rb
1 require "builder"
2 xml = Builder::XmlMarkup.new(:target => STDOUT, :indent=> 2)
3 xml.semster {
4   xml.class "Egyptology"
5   xml.class "Ornithology"
6 }
7 # => <semster> <class>Egyptology</class> <class>Ornithology</class>
```

这里为什么class方法不会和从Object#class()方法冲突呢？

这里是因为Builder中XmlMarkup类继承自一个白班类，其中删除了绝大多数的方法。

例如：

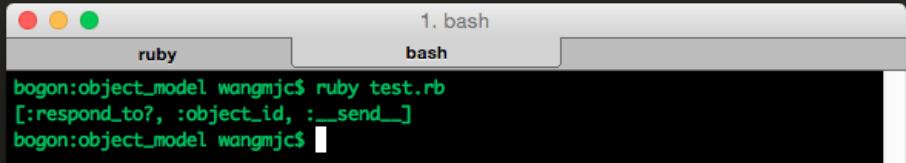
```
blank_state.rb
1 class BlankSlate
2
3
4   def self.hide(name)
5     if instance_methods.include?(name.to_s) and name !~ /^_|instance_eval/
6       #如果是这个类包含的实例方法，并且不是 __打头和 instance_eval方法
7       @hidden_method ||= {}
8       @hidden_method[name.to_sym] = instance_method(name) #把删除掉的方法存起来
9       undef_method name
10    end
11  end
12  instance_methods.each do |method| #删除所有的非__和instance_eval方法
13    hide(method)
14  end
15 end
16
17
```

重新修订computer类，把这computer类变为一个白板类，只保留method_missing和respond_to?类

```
1 class Computer
2   instance_methods.each do |method|
3     undef_method method unless method =~ /method_missing|respond_to?/
4   end
5 end
```

例如如下，删除了大部分的方法

```
1 class Computer
2   instance_methods.each do |method|
3     undef_method method unless method =~ /method_missing|respond_to?|object_id|__send__/
4   end
5 end
6
7
8 p Computer.instance_methods
```



```
bogon:object_model wangmjc$ ruby test.rb
[:respond_to?, :object_id, :__send_]
bogon:object_model wangmjc$
```

BasicObject类

为什么要有BasicObject类，因为语言本身想要提供一个白板类，只包含了几个基本的method，所以相当于白板类了，默认情况下，如果从BasicObject类集成的就会成为白板类

```
[5] pry(main)> BasicObject.instance_methods
=> [:=,
:equal?,
:!,
:!=,
:instance_eval,
:instance_exec,
:__send__,
:__id__,
:__binding__]
```