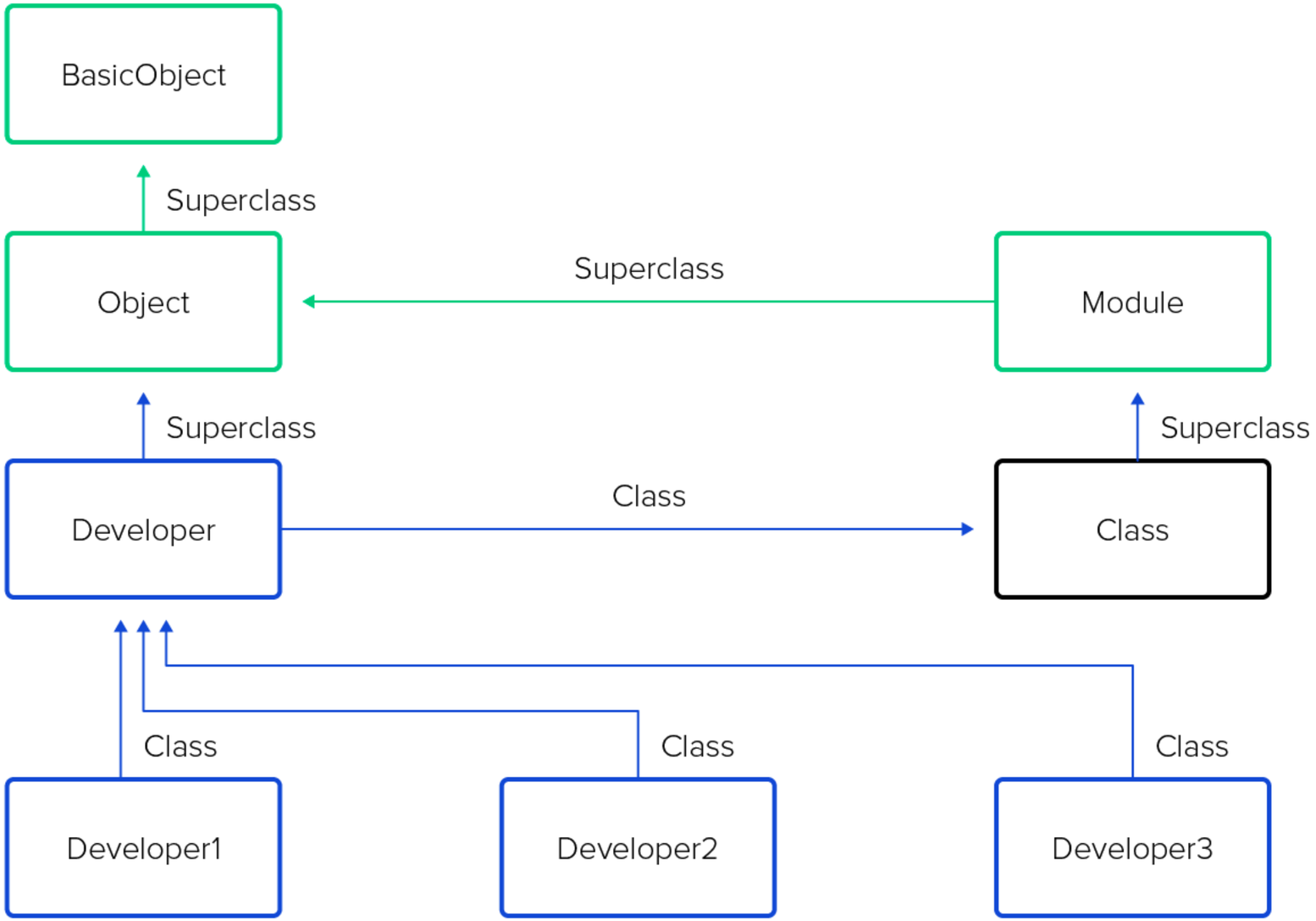
Ruby 元编程

深入Ruby内部理解Ruby

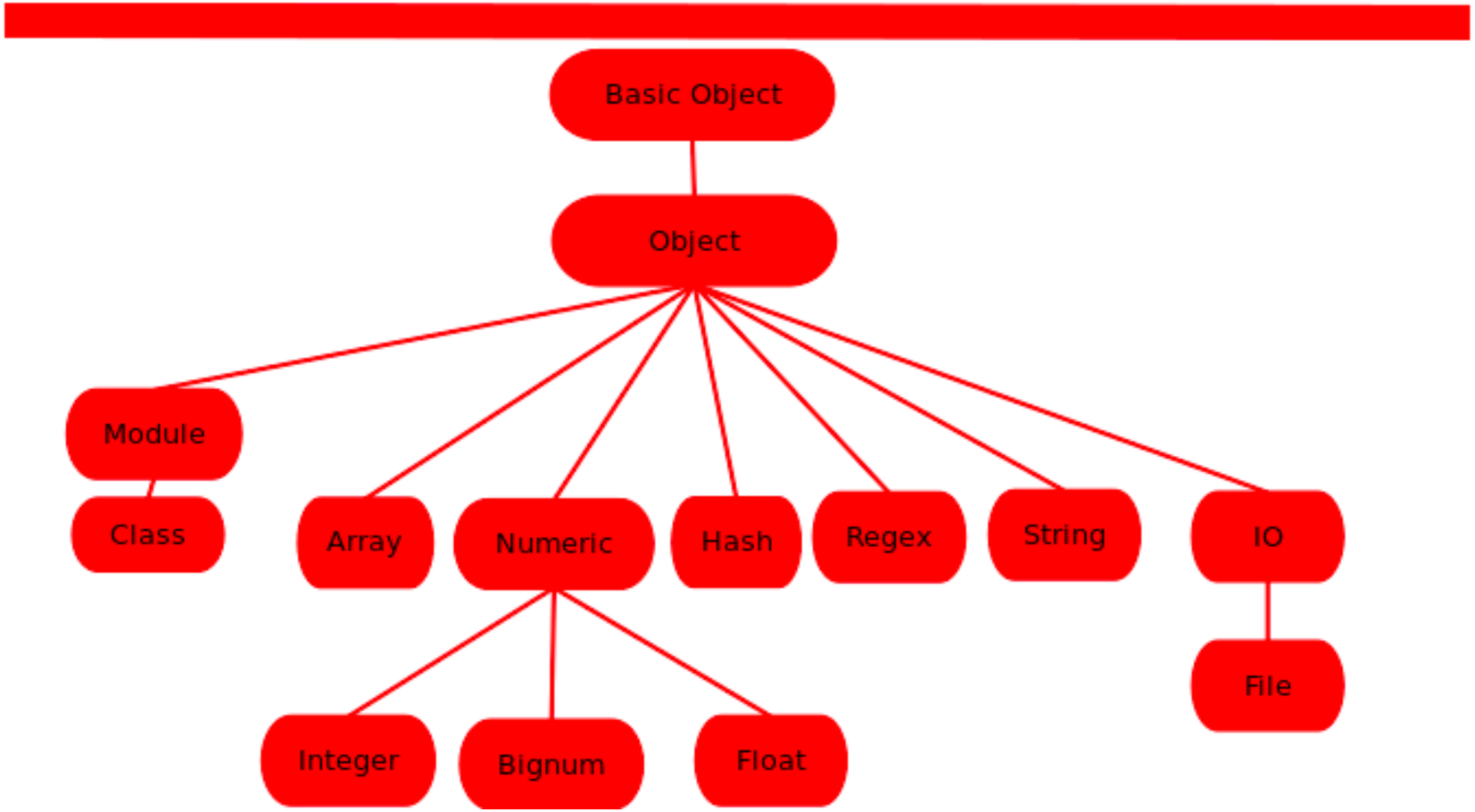
王强生

重新理解Ruby的继承

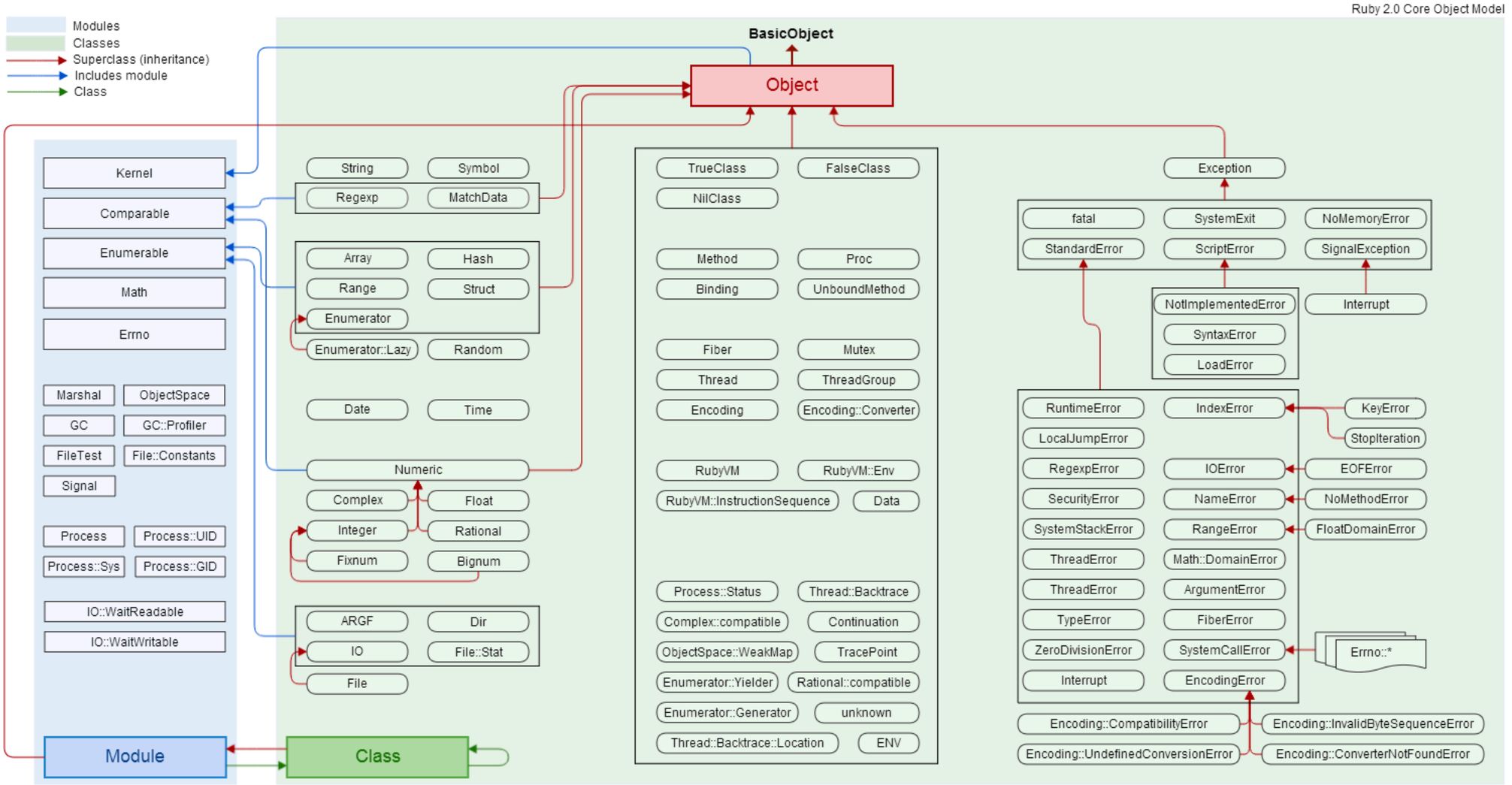


重新理解Ruby的继承

Class Heirarchy



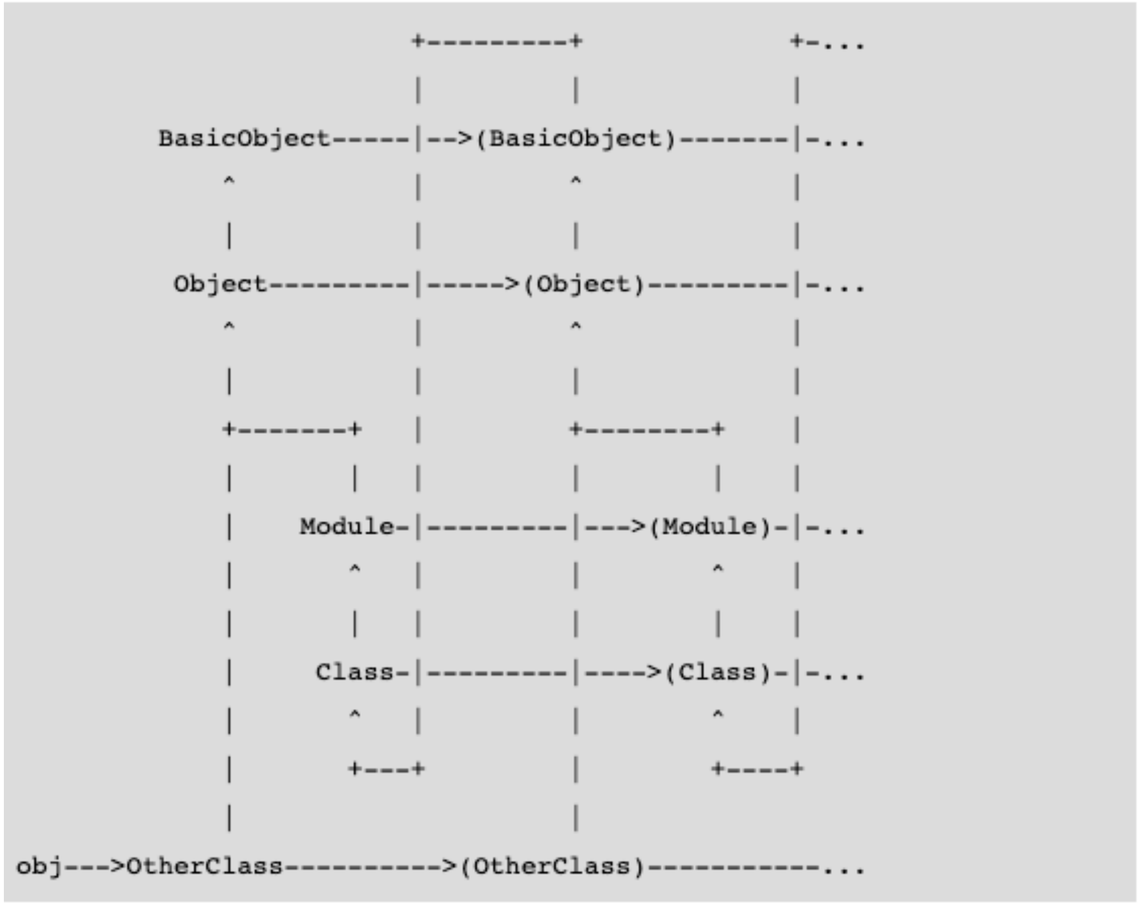
重新理解Ruby的继承



重新理解Ruby的继承

所有的元类都是Class的实例

Classes, modules, and objects are interrelated. In the diagram that follows, the vertical arrows represent inheritance, and the parentheses metaclasses. All metaclasses are instances of the class `Class'.



BasicObject

Methods

::new
#!
#!=
#==
#__id__
#__send__
#equal?
#instance_eval
#instance_exec
#method_missing
#singleton_method_added
#singleton_method_removed
#singleton_method_undefined

```ruby
class MyObjectSystem < BasicObject
end
```

```ruby
obj = "a"
other = obj.dup

obj == other      #=> true
obj.equal? other  #=> false
obj.equal? obj    #=> true
```

```ruby
Object.new.object_id  == Object.new.object_id  # => false
(21 * 2).object_id    == (21 * 2).object_id    # => true
"hello".object_id     == "hello".object_id     # => false
"hi".freeze.object_id == "hi".freeze.object_id # => true
```

重新理解Ruby中一切都是Object

- 变量 + 方法
- 类的实例
- Class、Module 也是Object
- Module 的内部实现是Class

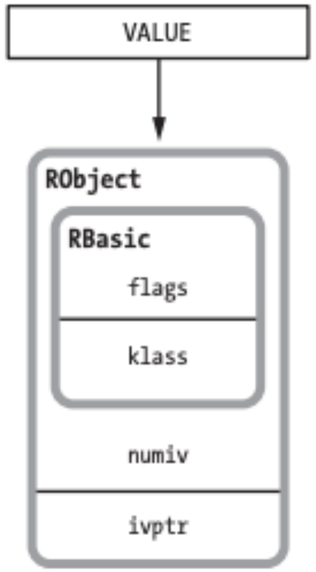


Figure 5-1: The RObject structure

类（Class）：

定义了一件事物的抽象特点。类的定义包含了数据的形式以及对数据的操作

重新理解Ruby中一切都是Object

- 简单立即值：数值、布尔、nil 等

Figure 5-5: Ruby saves integers in the VALUE pointer.

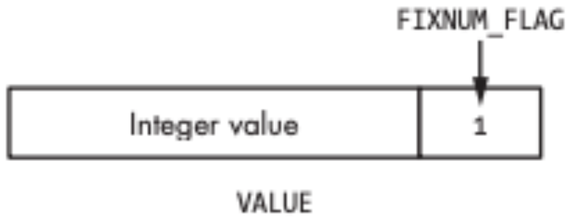


Figure 5-6: FIXNUM_FLAG indicates this is an instance of the Fixnum class.

- 基本类型对象

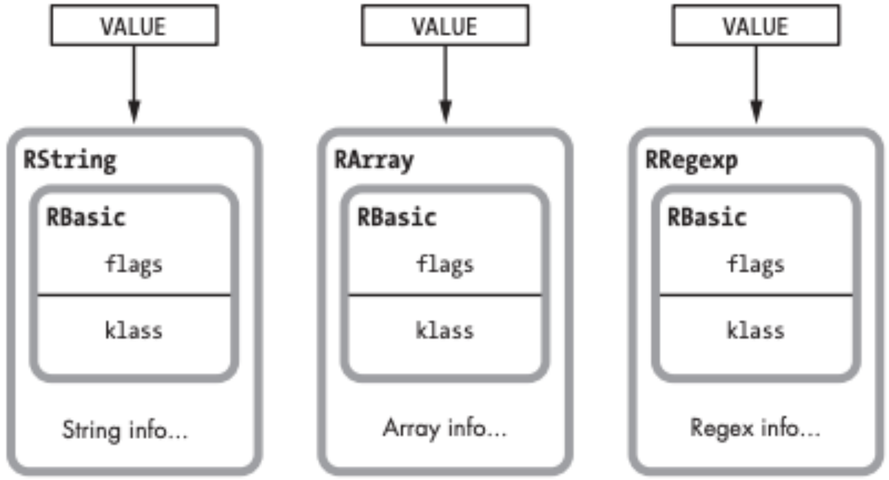


Figure 5-4: Different Ruby object structures all use the RBasic structure.

重新理解Ruby中一切都是Object

```ruby
3.times do
  class C
    p "created class C"
  end
end
```

重新认识类Class

从某种意义上说，Ruby中的class关键字更像是作用域操作符，而非类声明语句

Home Classes Methods

In Files

class.c
object.c

Parent

Module

Methods

::new
#allocate
#inherited
#new
#superclass

Class

Classes in Ruby are first-class objects—each is an instance of class `Class`.

Typically, you create a new class by using:

```
class Name
 # some code describing the class behavior
end
```

When a new class is created, an object of type Class is initialized and assigned to a global constant (`Name` in this case).

When `Name.new` is called to create a new object, the `new` method in `Class` is run by default. This can be demonstrated by overriding `new` in `Class`:

```
class Class
  alias old_new new
```

重新认识类Class

- 实例变量
- 类实例变量：类和子类的值独立
- 类变量：共享该值

```ruby
class Person
    @gental = 'Male'
    def self.gental
       @gental
    end
end

class Dave < Person
    @gental = "Female"
end

p Person.gental
p Dave.gental
```

```
"Male"
"Female"
"Female"
```

```ruby
class Person
    @@gental = 'Male'
    def self.gental
       @@gental
    end
end

class Dave < Person
    @@gental = "Female"
end

p Person.gental  #类变量被修改
p Dave.gental
```

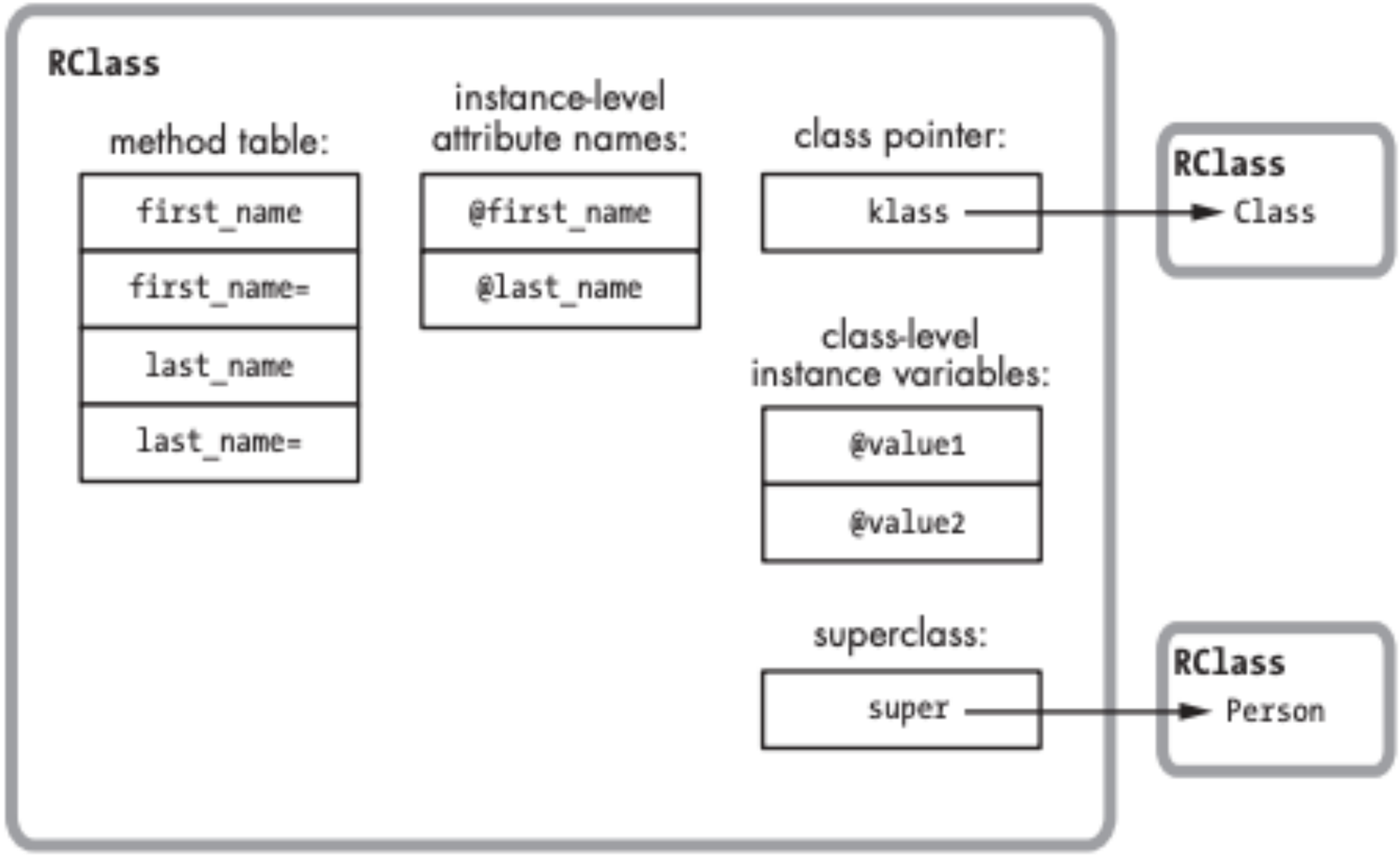
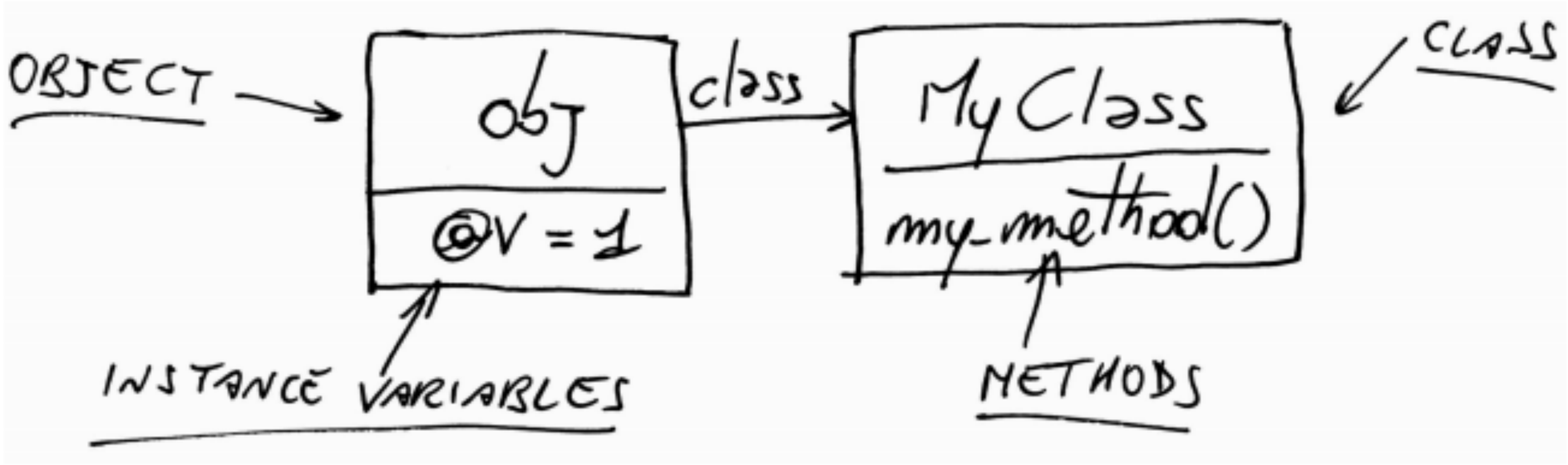


Figure 5-15: Ruby classes also contain a superclass pointer.

重新认识类Class


OBJECT
obj
@v = 1
class
MyClass
my_method()
CLASS
INSTANCE VARIABLES
METHODS

Class Hook

#inherited(subclass)

Methods

::new
#allocate
#inherited
#new
#superclass

```ruby
def inherited(child_class) # :nodoc:
  # initialize cache at class definition for thread safety
  child_class.initialize_find_by_cache
  unless child_class.base_class?
    klass = self
    until klass.base_class?
      klass.initialize_find_by_cache
      klass = klass.superclass
    end
  end
  super
end
```

重新认识Module

- 命名空间
- Mixin
-
```ruby
module Rails
  extend ActiveSupport::Autoload
  extend ActiveSupport::Benchmarkable

  autoload :WelcomeController

  class << self
    @application = @app_class = nil
    ...

    def autoloaders
      Autoloaders
    end
  end
end
```

问题:

- Module 和类的区别? （:new)
- include 后同名方法执行顺序
- extend 和include区别?
- prepend

重新认识Module

- pretend（from 2.0）

通俗地讲：先查找module里面的方法。

科学的说法：

It actually works like include, except that instead of inserting the module between the class and its superclass in the chain, it will insert it at the bottom of the chain, even before the class itself.

```ruby
module ServiceDebugger
  def run(args)
    puts "Service run start: #{args.inspect}"
    result = super
    puts "Service run finished: #{result}"
  end
end

class Service
  prepend ServiceDebugger

  # perform some real work
  def run(args)
    args.each do |arg|
      sleep 1
    end
    {result: "ok"}
  end
end
```

重新认识Module

Tips: include && extend

```ruby
module Logging
  module ClassMethods
    def logging_enabled?
      true
    end
  end

  def self.included(base)
    base.extend(ClassMethods)
  end

  def log(level, message)
    # ...
  end
end
```

Module methods

::constants
::nesting
::new
::used_modules
#<
#<=
#<=>
#==
#===
#>
#>=
#alias_method
#ancestors
#append_features
#attr
#attr_accessor
#attr_reader
#attr_writer
#autoload
#autoload?
#class_eval
#class_exec
#class_variable_defined?
#const_defined?
#const_get
#const_missing
#const_set
#constants
#define_method
#deprecate_constant
#extend_object
#extended
#freeze
#include
#include?
#included
#included_modules
#inspect
#instance_method
#instance_methods
#method_added
#method_defined?
#method_removed
#method_undefined
#module_eval
#module_exec
#module_function
#name
#prepend
#prepend_features
#prepended
#private
#private_class_method
#private_constant
#private_instance_methods
#private_method_defined?
#protected
#protected_instance_methods
#protected_method_defined?
#public
#public_class_method
#public_constant
#public_instance_method
#public_instance_methods
#public_method_defined?
#refine
#remove_class_variable
#remove_const
#remove_method
#singleton_class?
#to_s
#undef_method
#using

Module Hook

included, extended, pretend

```ruby
module Commentable
  def self.included(commentable_entity)
    puts "The #{commentable_entity} entity now accepts comments !"
  end
end

class MediumPost
  include Commentable
end
```

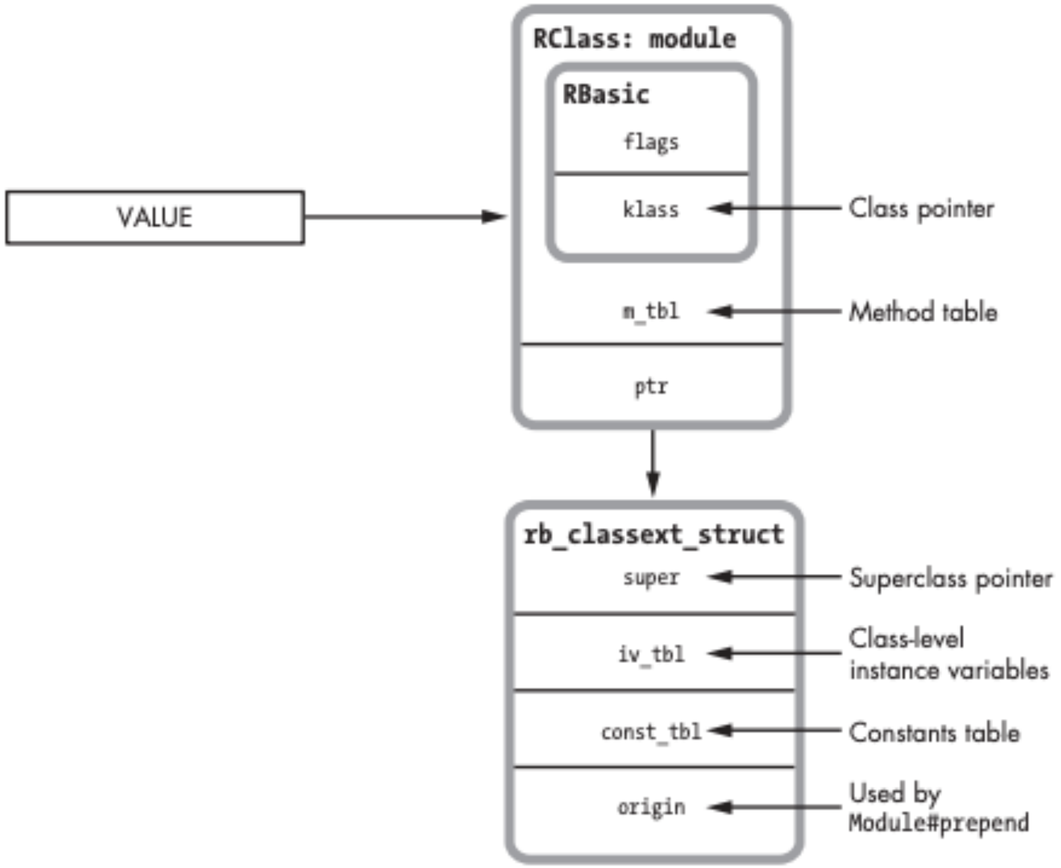
```ruby
module Concern
  class MultipleIncludedBlocks < StandardError # :nodoc:
    def initialize
      super "Cannot define multiple 'included' blocks for a Concern"
    end
  end

  class MultiplePrependBlocks < StandardError # :nodoc:
    def initialize
      super "Cannot define multiple 'prepended' blocks for a Concern"
    end
  end

  def self.extended(base) # :nodoc:
    base.instance_variable_set(:@_dependencies, [])
  end
```

Ruby中如何实现模块

在内部，Ruby创建的是类，而不是模块



3.hours.from_now_on

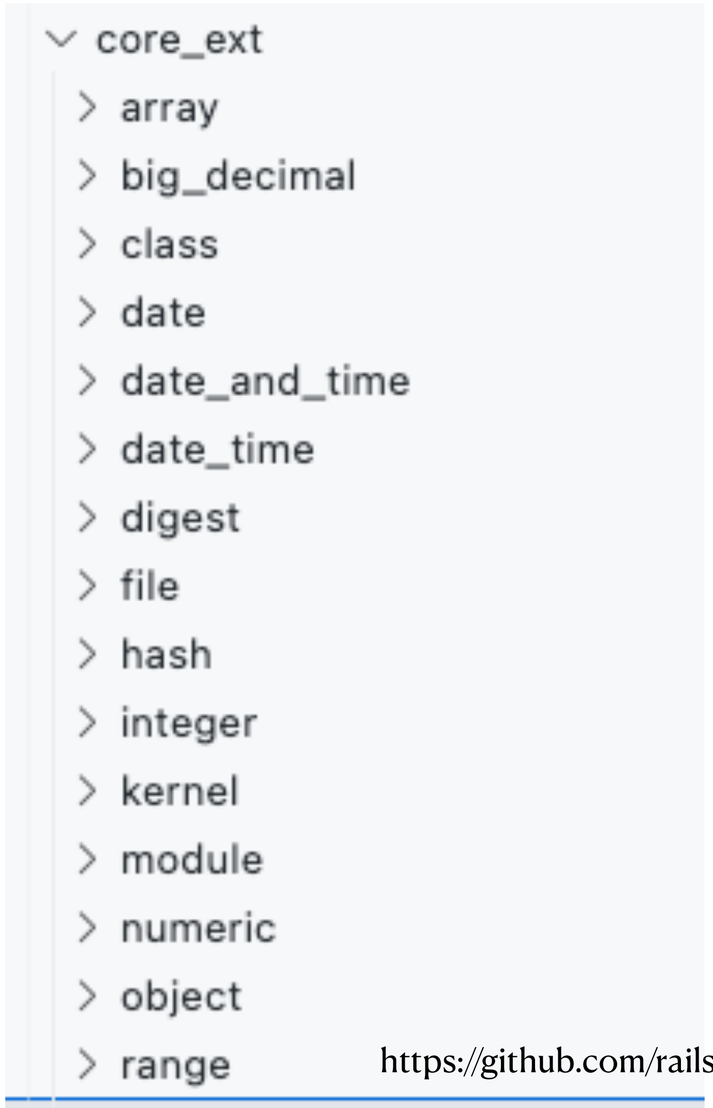
```ruby
class Integer
  def hours
    self * 60 * 60
  end

  def from_now_on
    Time.now + self
  end
end

3.hours.from_now_on # not work in this lab
```

打开类—monkeypatch

Matz认为所有程序员都是合格的程序员




```ruby
class Numeric
  # Returns a Duration instance matching the number of second
  #
  #   2.seconds # => 2 seconds
  def seconds
    ActiveSupport::Duration.seconds(self)
  end
  alias :second :seconds

  # Returns a Duration instance matching the number of minute
  #
  #   2.minutes # => 2 minutes
  def minutes
    ActiveSupport::Duration.minutes(self)
  end
  alias :minute :minutes

  # Returns a Duration instance matching the number of hours
  #
  #   2.hours # => 2 hours
  def hours
    ActiveSupport::Duration.hours(self)
  end
  alias :hour :hours

  # Returns a Duration instance matching the number of days p
  #
  #   2.days # => 2 days
  def days
    ActiveSupport::Duration.days(self)
  end
  alias :day :days
  # Returns a Duration instance matching the number of weeks
```

https://github.com/rails/rails/blob/6b5915895b5e58627d834bbd23bd8e3ceb3296a3/activesupport/lib/active_support/core_ext/numeric/time.rb

类VS模块

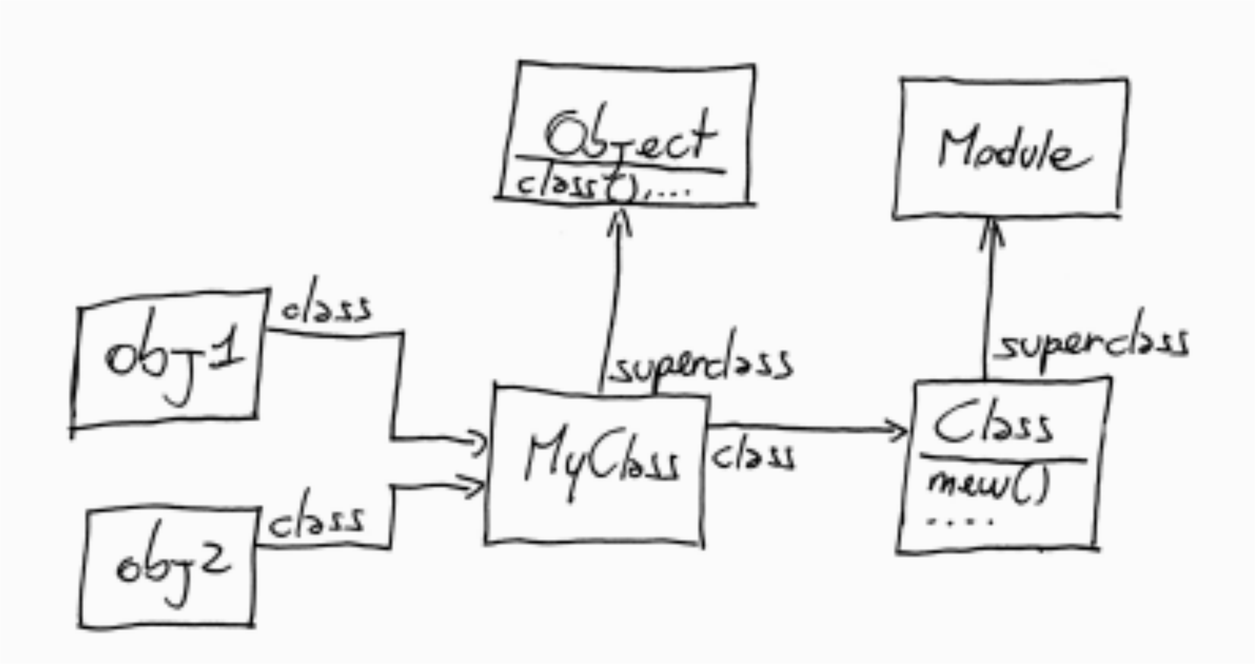
类就是多了三个方法的模块

- 模块： 一组实例方法
- 类：增加了（:new, :superclass, :alloc)的模块
- 模块的优点：清晰性
- 何时用module： 1. 命名空间 2. 需要被include
- 何时用class： 1. 需要被继承
- 注： module中不建议使用实例变量等

quiz

missing lines

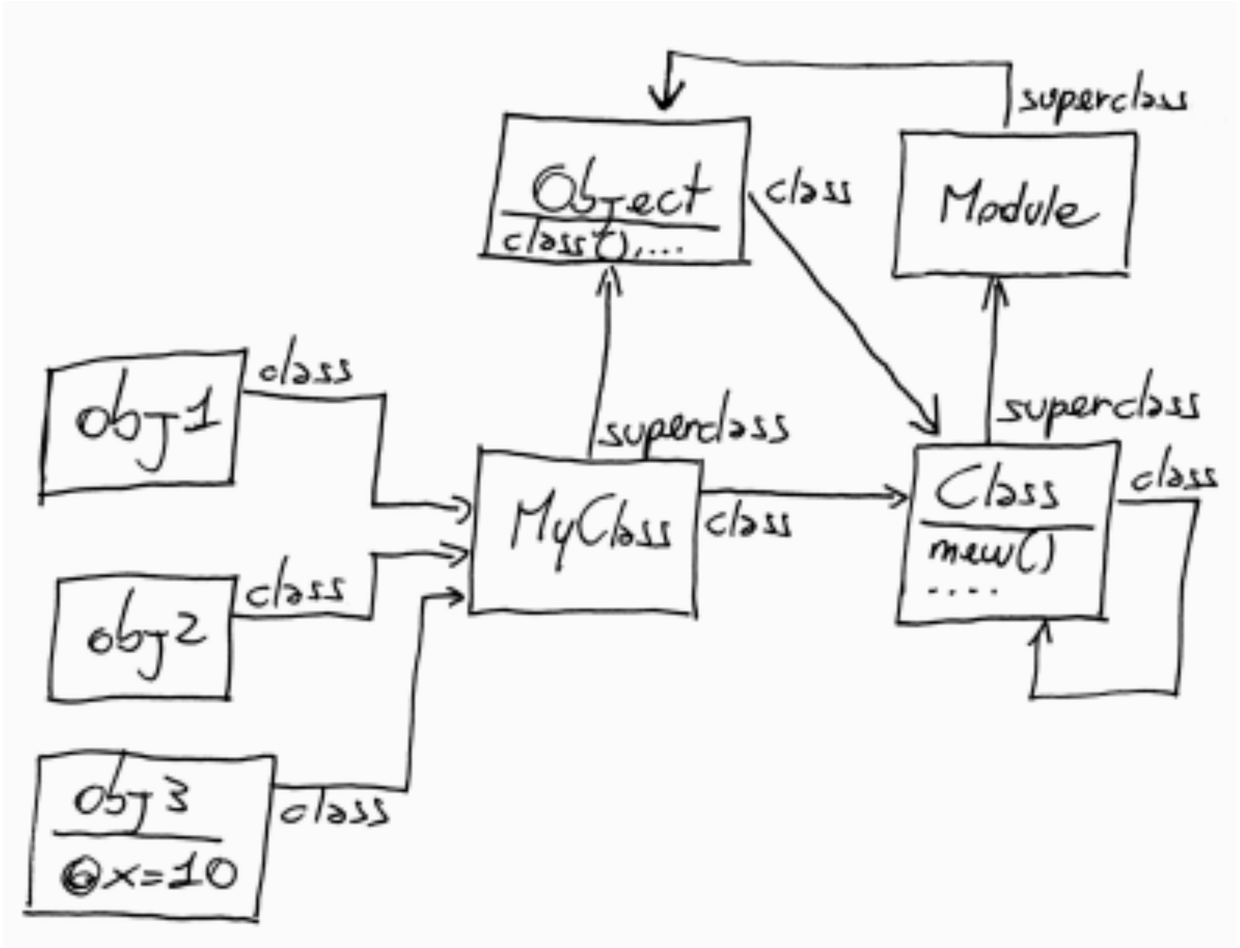
- Object的类是?
- Module的超类是?
- Class的类是?



quiz

missing lines

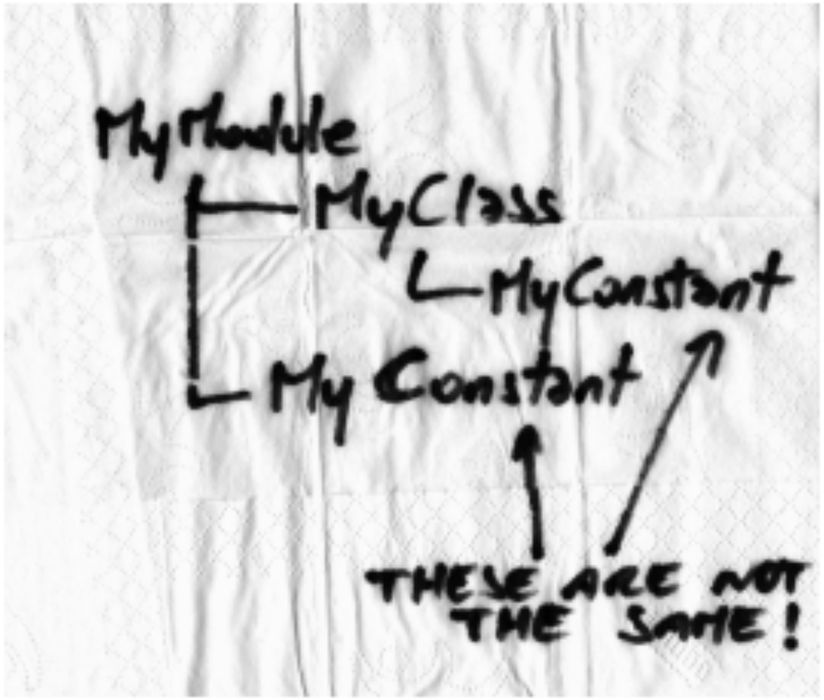
- Object的类是?
- Module的超类是?
- Class的类是?



常量

任何以大写字母开通的引用（包括类名、模块名）

- 常量可以被修改（不建议）
- 作用域和变量不同
- 常量的作用域更像是文件系统一样




```
module MyModule
  CONSTANT = "MODULE CONSTANT"
  class MyClass
    CONSTANT = "MODULE::CLASS CONSTANT"
  end
end
p ::MyModule::CONSTANT => MODULE CONSTANT
p MyModule::MyClass::CONSTANT => MODULE::CLASS CONSTANT
module.constants[0..1]
MyModule.constans    => [:CONSTANT, :MyClass]
```

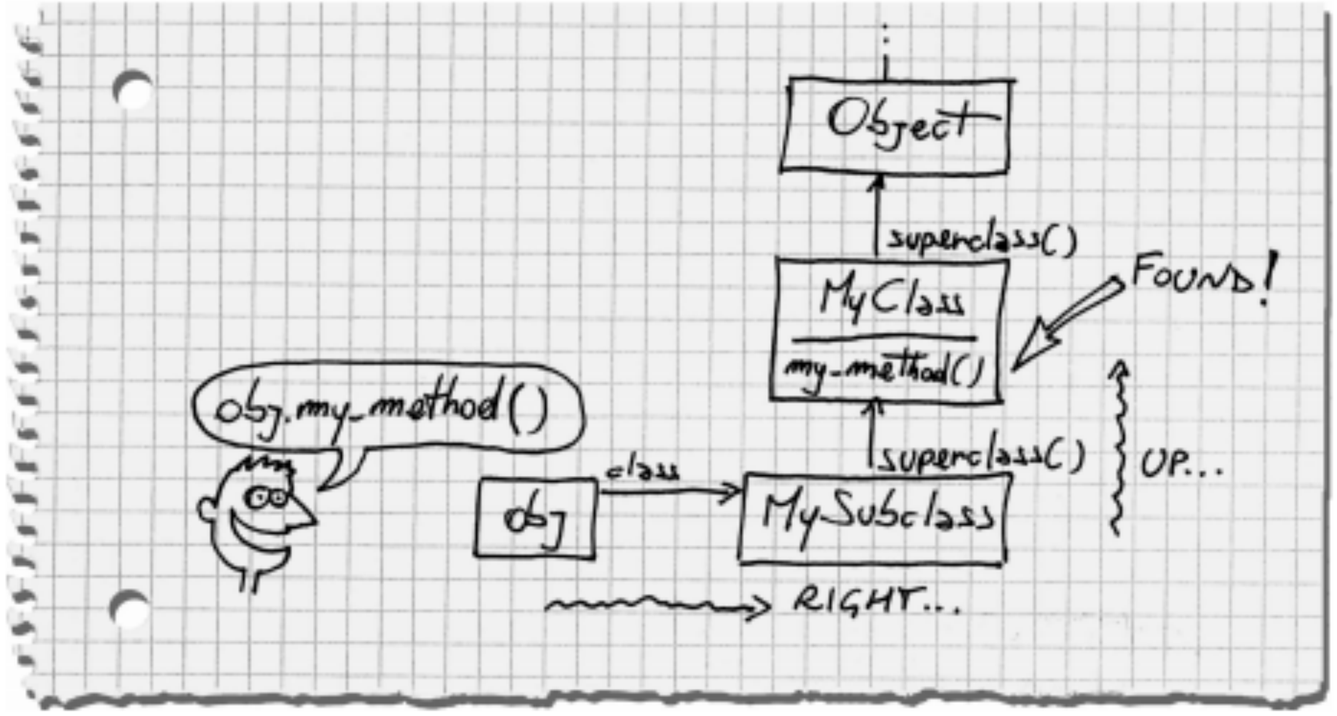
修剪常量树

load(“file_name.rb”, true)

- require: 导入，但不执行
- load： 导入，并执行
- 通过load(“file_name.rb”, true)这种方式调用的，Ruby会创建一个匿名的模块。

方法查找

向右一步，再向上



方法查找

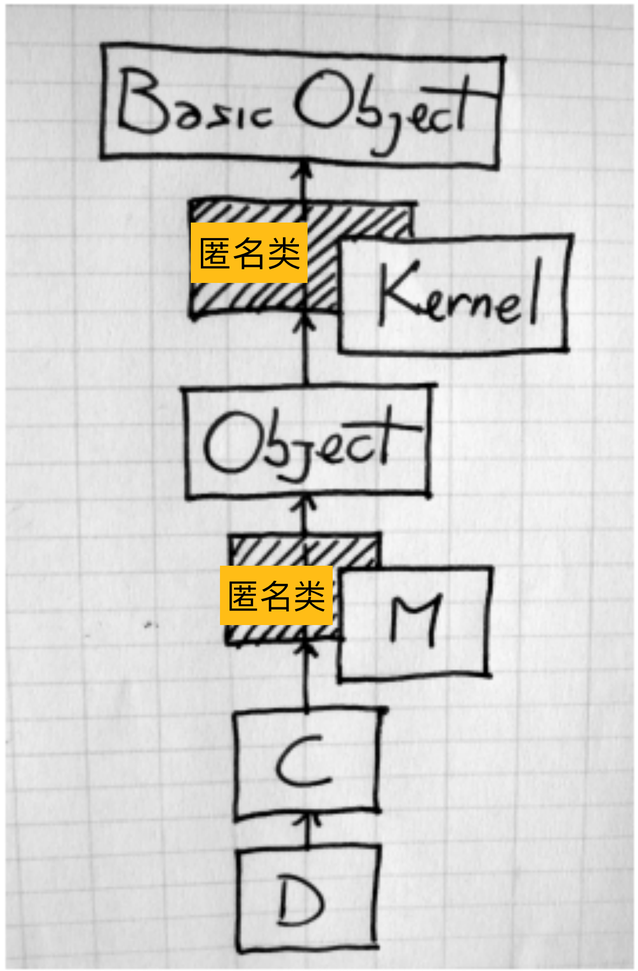
在一个类中include一个模块，Ruby会创建一个匿名类，插入这个类的正上方

```ruby
module M
  def my_module_method
    'My module method'
  end
end

class C
  include M
end

class D < C
end

D.new.my_module_method
```



Kernel

Object包含了Kernel

#Array
#Complex
#Float
#Hash
#Integer
#Rational
#String
#__callee__
#__dir__
#__method__
#`
#abort
#at_exit
#autoload
#autoload?
#binding
#block_given?
#callcc
#caller
#caller_locations
#catch
#chomp
#chop
#eval
#exec
#exit
#exit!
#fail
#fork
#format
#gets
#global_variables
#gsub
#iterator?
#lambda
#load
#local_variables
#loop
#open
#p
#pp
#print
#printf
#proc
#putc
#puts
#raise
#rand
#readline
#readlines
#require
#require_relative
#select
#set_trace_func
#sleep
#spawn
#sprintf
#srand
#sub
#syscall
#system
#test
#throw
#trace_var
#trap
#untrace_var
#warn

方法查找算法

Ruby会把类的祖先的指针用linked list保存起来

- Ruby的内部通过继承来实现模块的include
- 在类中包括多个模块，等价于多重继承！
- Ruby通过强制使用单一的祖先链让一切变得简单

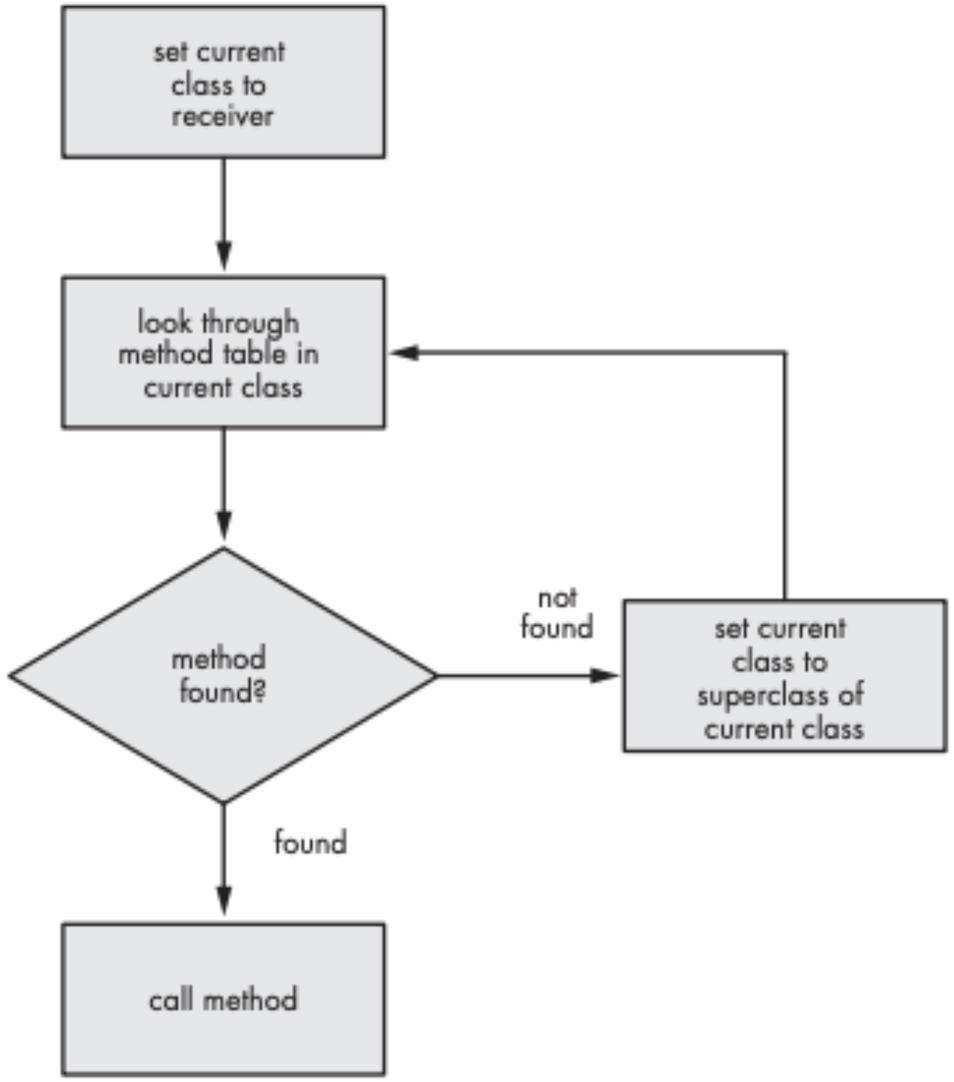


Figure 6-3: Ruby's method lookup algorithm

Ruby中方法查找缓存

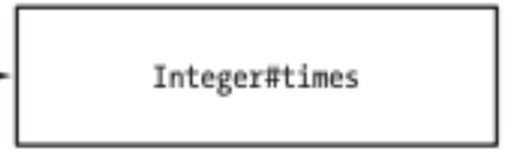
- 全局方法缓存

Table 6-1: An Example of What the Global Method Cache Might Contain

klass	defined_class
Fixnum#times	Integer#times
Object#puts	BasicObject#puts
etc...	etc...

- 内联方法缓存

```
putobject 10
send      <callinfo!mid:times, argc:0,
          block:block in <compiled>>
```



清空Ruby中方法缓存

Ruby中清空方法缓存发生的相当频繁

- 创建或移除方法时
- 在类中include模块时
- 当使用refinement时
- 当采用其他元编程特性时