Rack编程

potian

2010.1.1

Simple is Beautiful

Contents

1 简介 1

1.1 什么是Rack . . . 1

1.2 为什么Rack . . . 2

1.2.1 获得广泛支持的标准接口 . . . 2

1.2.2 中间件 . . . 4

1.2.3 简单 . . . 4

1.3 一试 . . . 4

1.3.1 安装 . . . 4

1.3.2 Handler . . . 5

1.3.3 一个可被call的对象 . . . 5

1.3.4 返回一个数组 . . . 6

1.3.5 其他合法的应用程序 . . . 7

2 Rack初探 9

2.1 环境 . . . 9

2.1.1 Rack相关变量 . . . 11

2.1.2 CGI头 . . . 11

2.2 Request . . . 12

2.3 Response . . . 13

2.3.1 响应体 . . . 14

2.3.2 状态码 . . . 16

2.3.3 响应头 . . . 16

3 中间件 19

3.1 一个简单的中间件 . . . 19

3.2 Rack响应标准 . . . 20

3.3 为什么中间件 . . . 22

3.4 装配中间件 . . . 23
3.4.1 如何装配 . . . 23
3.4.2 实现Builder . . . 25

4 最简单的Web框架 29
4.1 Rack::Builder . . . 29
4.1.1 替换为Rack::Builder . . . 29
4.1.2 路由 . . . 30
4.2 rackup . . . 36
4.2.1 rackup配置文件 . . . 36
4.2.2 rackup 实现 . . . 37
4.2.3 Rack::Server接口 . . . 37
4.3 没有了? . . . 42

5 中间件:第二轮 43
5.1 再议响应体 . . . 43
5.2 Rack自带中间件 . . . 45
5.3 HTTP协议中间件 . . . 46
5.3.1 Rack::Chunked . . . 46
5.3.2 Rack::ConditionalGet . . . 48
5.3.3 Rack::ContentLength . . . 54
5.3.4 Rack::ContentType . . . 56
5.3.5 Rack::Deflater . . . 57
5.3.6 Rack::Etag . . . 63
5.3.7 Rack::Head . . . 64
5.3.8 Rack::MethodOverride . . . 65
5.4 程序开发中间件 . . . 67
5.4.1 Rack::CommonLogger . . . 67
5.4.2 Rack::Lint . . . 67
5.4.3 Rack::Reloader . . . 79
5.4.4 Rack::Runtime . . . 82
5.4.5 Rack::Sendfile . . . 83
5.5 应用配置和组合中间件 . . . 85
5.5.1 Rack::Cascade . . . 85
5.5.2 Rack::Lock . . . 85

5.6 会话管理 86
5.6.1 HTTP Cookies 86
5.6.2 Rack::Session::Cookie 90
5.6.3 ID session 96
5.6.4 Memcache Session 101
5.6.5 Pool Session 105

Chapter 1

简介

1.1 什么是Rack

Rack是Ruby应用服务器和Rack应用程序之间的一个接口。

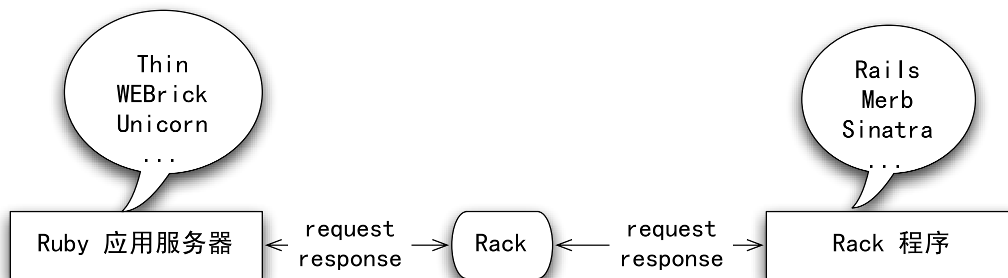

Figure 1.1: Rack接口

图1.1(p. 1)给出了一个简单的示意。用户的请求抵达应用服务器时，应用服务器会调用(*call*)Rack，Rack对请求进行包装，然后调用你的Rack程序。Rack程序可以方便地利用Rack所提供的各种API，分析请求，进行处理，并利用Rack提供的响应设施进行输出，Rack会把用户的响应作为输出返回给Ruby应用服务器。

严格来说，这样讲即不准确也不完整，但是很多概念会随着本书的深入得到澄清。

我们在本书中将不加区别地使用*Web服务器*和*应用服务器*这两个概念。这是因为通常来说用Ruby编写的Web服务器总是位于一个反向代理(例如nginx、lighttpd和Apache等等)后端，用来服务动态内容请求。

1.2 为什么Rack

1.2.1 获得广泛支持的标准接口

首先Rack提供了一种标准的接口，便于应用程序和应用服务器之间的交互。一个Rack应用程序可以被任何和Rack兼容的应用服务器调用。

目前几乎所有的主流Ruby应用服务器都支持Rack接口。图1.1(p. 1)左边只列出了Thin、WEBrick、Unicorn，实际上Rack所支持的应用服务器远比这里列出的多得多。Rack通过一种叫做句柄(*handler*)的机制实现对应用服务器的支持。目前，Rack本身带有的句柄包括:

- Mongrel
- EventedMongrel
- SwiftipliedMongrel
- WEBrick
- FCGI
- CGI
- SCGI
- LiteSpeed
- Thin

下面的应用服务器也在它们自己的代码中包括了Rack handler:

- Ebb
- Fuzed
- Glassfish v3
- Phusion Passenger (which is mod_rack for Apache and for nginx)
- Rainbows!
- Unicorn
- Zbatery

这意味着所有上述的服务器都以Rack接口的形式调用Rack应用程序。

这些句柄都位于`Rack::Handler`名字空间之下，Rack的文档中，我们可以看到下面这些类

```
Rack::Handler::CGI
Rack::Handler::EventedMongrel
Rack::Handler::FastCGI
Rack::Handler::LSWS
Rack::Handler::Mongrel
Rack::Handler::SCGI
Rack::Handler::SwiftipliedMongrel
Rack::Handler::Thin
Rack::Handler::WEBrick
```

另外一个方面，几乎所有的主流Web框架都支持Rack接口，这意味着，用这些框架编写的应用程序都是标准的Rack应用程序。图1.1(p. 1)右边我们只列出了Rails、Merb和Sinatra。事实上，下面这些框架

- Camping
- Coset
- Halcyon
- Mack
- Maveric
- Merb
- Racktools::SimpleApplication
- Ramaze
- Ruby on Rails
- Rum
- Sinatra
- Sin
- Vintage
- Waves
- Wee

完全是和Rack兼容的。这些框架都包含一个Rack适配器(*adapter*)。

因此，任何用上面列出的框架编写的程序都可以不加修改地被上面列出的所有应用服务器调用。

毫无疑问，未来的Ruby web框架和Ruby Web服务器都会支持Rack接口。

1.2.2 中间件

Rack利用中间件实现了最大程度的模块化。这当然可以提高Web应用程序部件的可重用性，从而提高开发的效率。

Rack中间件对Ruby Web框架也有着深远的影响，包括:

- 不同的Web框架之间可以重用中间件，这意味这你可以编写的中间件可以在几乎所有的主流框架中使用
- 可以通过不同的中间件组合组装出同一个Web框架的不同变种，以适合不同的应用场合
- 可以组合多个不同的Web应用框架为同一个更大的系统服务

Web框架的框架

1.2.3 简单

Rack的标准非常简单，整个规格书 http://rack.rubyforge.org/doc/SPEC.html 大约只有2页A4纸的内容。如果你要实现一个Web服务器或者一个Web框架，只需要符合这个简单的标准即可。

1.3 一试

1.3.1 安装

首先请安装rack:

```
[sudo] gem install rack
```

1.3.2 Handler

启动irb，要使用Rack必须先引入rack包。

```
$ irb
irb(main):001:0> require 'rubygems'
=> true
irb(main):002:0> require 'rack'
=> true
```

我们可以查询Rack内嵌的所有Handler:

```
irb> Rack::Handler.constants
=> ["Mongrel", "SCGI", "CGI", "LSWS", "FastCGI",
   "SwiftipliedMongrel", "WEBrick", "Thin", "EventedMongrel"]
```

所有的Rack Handler都有一个`run`方法，你可以用

```
Rack::Handler:: Mongrel.run ...
Rack::Handler:: WEBrick.run ...
Rack::Handler:: Thin.run ...
```

来运行你的Rack程序。

1.3.3 一个可被call的对象

那么一个Rack程序需要符合什么条件呢？Rack规格书中写到:

> A Rack application is an Ruby **object** (not a class) that responds to `call`. It takes exactly one argument, the environment and returns an **Array** of exactly three values: The **status**, the **headers**, and the **body**.

一个Rack应用程序是一个Ruby对象，只要这个对象能够响应call。Ruby中能够响应一个call的对象很多，包括:

- 一个`lambda`或者proc对象
- 一个method对象
- 任何一个对象，它的类包含一个call方法

显然，最简单的能够call的对象是一个空的lambda，因为它可以接受call:

```
irb> lambda {}.call
```

我们先用这个最简单的空`lambda {}`作为run的第一个参数

```
irb>Rack::Handler::WEBrick.run lambda{}, :Port=>3000
 INFO  WEBrick 1.3.1
 INFO  ruby 1.8.7 (2009-06-12) [i686-darwin9.8.0]
 INFO  WEBrick::HTTPServer#start: pid=1513 port=3000
```

第二个参数是一个hash，其中:Port指定WEBrick监听的端口。WEBrick给出的日志信息表示它已经正常启动。

打开你喜爱的浏览器，输入`http://localhost:3000`

```
Internal Server Error

undefined method `each' for nil:NilClass
WEBrick/1.3.1 (Ruby/1.8.7/2009-06-12) at localhost:3000
```

内部错误。

1.3.4 返回一个数组

为什么？Rack的规格书继续写道:

> It takes exactly one argument, the environment and returns an **Array** of exactly three values: The **status**, the **headers**, and the **body**.

这个可被call的对象需要接受一个参数，即环境(*environment*)对象；需要返回一个数组，这个数组有三个成员:

1. 一个状态(*status*)，即http协议定义的状态码
2. 一个头(*headers*)，它可能是一个hash，其中包含所有的http头
3. 一个体(*body*)，它可能是一个字符串数组。(ruby 1.8.x的例子通常会是一个字符串，但这样代码就无法在ruby 1.9.x 中运行。具体的原因参见3.2(p. 20))

根据此要求，我们编写一个最简单的合法的Rack应用程序

```
irb> rack_app = lambda{|env| [200,{},["hello from lambda"]]}
irb> Rack::Handler::WEBrick.run rack_app ,:Port=>3000
```

如果此时你再次在浏览器中输入`http://localhost:3000`，那么将得到

```
hello from lambda
```

成功了！我们写出了第一个符合规格的Rack程序。

如果你安装了Thin服务器，那么你可以:

```
irb> Rack::Handler::Thin.run rack_app ,:Port=>3000
```

你照样可以在浏览器上得到相同的结果。

1.3.5 其他合法的应用程序

除了lambda外，我们的应用程序还可以是method对象:

```
irb> def any_method(env)
irb> [200, {}, ["hello from method"]]
irb> end
=> nil
irb> method(:any_method).call({})
=> [200, {}, "hello from method"]
```

`method(:any_method)`返回一个method对象，它可以被call，所以它也是一个合法的Rack应用程序:

```
irb> rack_app = method(:any_method)
=> #<Method: Object#any_method>
irb> Rack::Handler::Thin.run rack_app ,:Port=>3000
```

在浏览器输入http://localhost:3000，你可以得到

hello from method。

当然一个合法的Rack应用程序也可以是任何对象，只要它的类定义了call方法。

```
irb> class AnyClass
irb>   def call(env)
irb>       [200, {}, ["hello from AnyClass instance with call defined"]]
irb>   end
irb> end
=> nil
irb> rack_app = AnyClass.new
=> #<AnyClass:0x144e8b8>
irb> Rack::Handler::Thin.run rack_app ,:Port=>3000
```

在浏览器输入http://localhost:3000，你可以得到

hello from AnyClass instance with call defined。

Chapter 2

Rack初探

本章我们将探究Rack为Rack应用程序提供的几个基础接口和概念。

2.1 环境

Rack用一个环境参数调用Rack应用程序，它是一个hash的实例。为了取得直观的认识，首先我们来编写一个简单的程序打印这些参数:

```
#!/usr/bin/env ruby
require "rubygems"
require "rack"
def pp(hash)
  hash.map {|key,value|
    "#{key} => #{value}"
  }.sort.join("<br/>")
end

Rack::Handler::WEBrick.run lambda {|env| [200,{},[pp(env)]]} , :Port=>3000
```

Figure 2.1: rack-env.rb

把代码保存到`rack-env.rb`文件，然后用`ruby rack-env.rb`即可运行。

当然，由于图2.1(p. 9)的第一行`#!/usr/bin/env ruby`是shebang行，只需为这个文件加入执行权限:

```
chmod +x rack-env.rb
```

则可以直接在命令行运行`./rack-env.rb`。

接下去是require rubygems和rack，对于ruby 1.9来说，`require rubygems`不是必须的，但是所有rack程序都需要引入rack包。下面的pp方法用来美化打印一个Hash表，在每一个关键字/值对之间插入了一个HTML标签`<br/>`。

整个程序实际做的事情就是最后一行，我们的Rack应用程序是:

```
lambda {|env| [200,{},[pp(env)]]}
```

运行程序。打开浏览器，输入http://localhost: 3000/someuri，我们得到结果如下:

```
GATEWAY_INTERFACE => CGI/1.1
HTTP_ACCEPT => application/xml,application/xhtml+xml,text/html;q=0.9,
text/plain;q=0.8,image/png,*/*;q=0.5
HTTP_ACCEPT_ENCODING => gzip, deflate
HTTP_ACCEPT_LANGUAGE => zh-cn
HTTP_CONNECTION => keep-alive
HTTP_COOKIE => __qca=P0-1624383895-1252173928531;
__utma=111872281.86217066.1252173928.1252173928.1252173928.1;
__utmz=111872281.1252173928.1.1.utmcsr=(direct)|utmccn=(direct)|utmcmd=(none)
HTTP_HOST => localhost:3000
HTTP_USER_AGENT => Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_8; zh-cn)
AppleWebKit/531.21.8 (KHTML, like Gecko) Version/4.0.4 Safari/531.21.10
HTTP_VERSION => HTTP/1.1
PATH_INFO => /someuri
QUERY_STRING =>
REMOTE_ADDR => ::1
REMOTE_HOST => localhost
REQUEST_METHOD => GET
REQUEST_PATH => /
REQUEST_URI => http://localhost:3000/someuri
SCRIPT_NAME =>
SERVER_NAME => localhost
SERVER_PORT => 3000
SERVER_PROTOCOL => HTTP/1.1
SERVER_SOFTWARE => WEBrick/1.3.1 (Ruby/1.8.7/2009-06-12)
rack.errors => #
rack.input => #
rack.multiprocess => false
rack.multithread => true
rack.run_once => false
rack.url_scheme => http
rack.version => 11
```

我们可以看到，env包含key可以分为两类，一类是大写的类CGI的头，还有一类则是rack特定的环境。

2.1.1 Rack相关变量

Rack要求环境中必须包括rack相关的一些变量。这些变量都是rack.xxxx的形式。关于规格书最详细的讨论可以参见5.4.2(p. 67)。

2.1.2 CGI头

当然，我们目前最关心的是CGI头，让我们看看几个非常重要的key:

REQUEST_METHOD 值为GET。这是HTTP请求的方法，可以是GET, POST等等。

PATH_INFO 值为/someuri，这是因为我们输入了http://localhost:3000/someuri。如果你试着输入http://localhost:3000/asdjkasj，那么将得到/asdjkasj。这个是我们程序所要处理的“路径”，利用它我们可以实现不同的“路由”算法。

QUERY_STRING 值为空。

现在输入http://localhost:3000/someuri?name=tony。你可以看到REQUEST_METHOD和PATH_INFO没有发生变化，但是

QUERY_STRING => name=tony

QUERY_STRING是请求的URL中“?”后面的那一部分。所以当我们输入http://localhost:3000/someuri?name=tony的时候，除了协议、主机名、端口外，剩余内容被分为两部分，“?”的前面为PATH_INFO即/someuri，“?’后面的部分是QUERY_STRING，即name=tony。如果没有“?”或者“?”后面为空，则QUERY_STRING为空。

我们可以在程序中从env直接获得这些信息。

```
#!/usr/bin/env ruby
require "rubygems"
require "rack"

Rack::Handler::WEBrick.run lambda {|env| [200,{},
  ["your request:
    http_method => #{env['REQUEST_METHOD']}
    path => #{env['PATH_INFO']}
    params=>#{env['QUERY_STRING']}"]]} , :Port=>3000
```

输入http://localhost:3000,我们得到:

```
your request:
    http_method => GET
    path => /
    params=>
```

输入http://localhost:3000/search,我们得到:

```
your request:
    http_method => GET
    path => /search
    params=>
```

输入http://localhost:3000/search?name=tony,我们得到:

```
your request:
    http_method => GET
    path => /search
    params=>name=tony
```

现在你可以想象编写一个Rack程序，你可以直接判断用户请求的方法、路径名和查询参数，然后直接调用对应的处理程序，非常高效地实现各种丰富多彩的功能。但是直接存取环境虽然直接高效，但却需要手工处理很多麻烦的事情。例如，如何解析查询的参数、如何维护用户会话信息、如果处理某些浏览器不支持PUT的情况、如何在响应时填充合适的HTTP头。 Rack提供了丰富的API帮助你快速方便地编写灵活的应用逻辑，我们首先来看看两个重要的类,Request和Response.

2.2 Request

Rack::Request为存取Rack环境提供了方便的接口。要创建一个Request对象，你只需为它的new方法提供提供一个Rack环境:

```
  request = Rack::Request.new(env)
```

新创建的request对象直接持有传入的env对象并在需要的时候对它进行修改，它自己没有任何状态。

我们可以直接以Hash的形式取得用户请求的参数，例如:

```
  reqest.params[somekey]
```

下面的程序让用户猜测什么是我们最喜欢的浏览器，用户可以输入形如的http://locahost:3000/guess?client=xxx这样的url。

```
#!/usr/bin/env ruby
require "rubygems"
require "rack"
rack_app = lambda { |env|
  request = Rack::Request.new(env)
  if request.path_info == '/guess'
    client = request['client']
    if client && client.downcase == 'safari'
       [200, {}, ["sweet heart"]]
    else
       [200, {}, ["choose another browser"]]
    end
  else
      [200, {}, ["you need guess something"]]
  end
}
Rack::Handler::WEBrick.run rack_app, :Port=>3000
```

如果用户请求的path_info不是/guess，那么我们将回答you need guess something。当用户输入的查询参数不包括client= safari时，我们则要求它们更换另外的浏览器名字。显然，能够直接用Hash存取用户请求的参数在很大程度上方便我们程序的实现。

Request提供了询问当前HTTP请求类型的简便方法：

方法名	含义
request_method()	请求的HTTP方法，包括GET, POST, PUT, DELETE, HEAD
get?()	HTTP请求是否为GET
head?()	HTTP请求是否为HEAD
post?()	HTTP请求是否为POST
put?()	HTTP请求是否为PUT
delete?()	HTTP请求是否为DELETE
xhr?()	HTTP请求是否为XMLHttpRequest请求(即Ajax请求)

关于Request完整的说明可以参考Rack的文档http://rack.rubyforge.org/doc/Rack/Request.html

2.3 Response

在前面的程序中，我们的Rack应用程序返回手工编写的数组。但是在一个复杂的应用程序中，我们可能需要对响应做更多的控制。例如，需要设置各种各样的HTTP响应头，处理cookies等工作。

Reponse提供了对响应的状态、HTTP头和内容进行处理的方便接口。

2.3.1 响应体

Request提供了两种方法来生成响应体：

- 直接设置response.body。此时你必须自己设置响应头中Content-Length的值。
- 用response.write增量写入内容，自动填充Content-Length的值。

要注意的是你不应该混用这两种方法。浏览器需要用Content-Length头信息决定从服务器端读取多少数据，因此这是必须的。

不管用什么方法，最后用response.finish完成。除了一些必要的检查工作外，finish将装配出符合Rack规范的一个数组–这个数组有三个成员：状态码、响应头和响应体–也就是我们原先手工返回的那个数组。

首先我们来看看如何直接设置response.body:

```
#!/usr/bin/env ruby
require "rubygems"
require "rack"
rack_app = lambda { |env|
```

```
  request = Rack::Request.new(env)
  response = Rack::Response.new
  body = "===========header==========<br/>"
  if request.path_info == '/hello'
    body << "you say hello"
    client = request['client']
    body << "from #{client}" if client
  else
    body << "you need provide some client information"
  end
  body << "<br/>===========footer=========="
  response.body = [body]
  response.headers['Content-Length'] = body.bytesize
  response.finish
}
Rack::Handler::WEBrick.run rack_app, :Port=>3000
```

如果运行上述程序，你会在浏览器中看到这样的错误:

```
Internal Server Error
private method `split' called for 88:Fixnum
```

原因在于

```
  response.headers['Content-Length'] = body.bytesize
```

Content-Length的值必须为字符串，所以你应该把这句语句改为:

```
  response.headers['Content-Length'] = body.bytesize.to_s
```

这份代码首先输出一个头，接着根据用户请求输出不同的内容，最后附加一个尾。同样的效果可以用response.write实现,如图2.2(p. 15)所示。

现在看起来这两种方法没有什么区别，我们将在后面进行深入的探讨。关于finish方法，我们也有更多的东西要说。

```
#!/usr/bin/env ruby
require "rubygems"
require "rack"
rack_app = lambda { |env|
  request = Rack::Request.new(env)
  response = Rack::Response.new
  response.write("===========header==========<br/>")
  if request.path_info == '/hello'
    response.write("you say hello")
    client = request['client']
    response.write("from #{client}") if client
  else
    response.write("you need provide some client information")
  end
  response.write("<br/>===========footer==========")
  response.finish
}
Rack::Handler::WEBrick.run rack_app, :Port=>3000
```

Figure 2.2: sayhello.rb

2.3.2 状态码

我们可以直接存取Response的对象来改变状态码。如果没有任何设置，那么状态码为200.

```
response.status = 200
```

Response提供了一个redirect方法直接进行重定向:

```
redirect(target, status=302)
```

下面的程序在你输入`http://localhost:3000/redirect`的时候将把你的浏览器重定向到google，不然打印`here`:

```
#!/usr/bin/env ruby
require "rubygems"
require "rack"
rack_app = lambda { |env|
  request = Rack::Request.new(env)
  response = Rack::Response.new
  if request.path_info == '/redirect'
    response.redirect('http://google.com')
  else
    response.write('here')
  end
  response.finish
}
Rack::Handler::WEBrick.run rack_app, :Port=>3000
```

2.3.3 响应头

你还可以直接写入Response的头信息headers，这是一个Hash。例如:

```
response.headers['Content-Type'] = 'text/plain'
```

修改上面的代码，让它直接返回普通文本而不是html给浏览器:

```
#!/usr/bin/env ruby
require "rubygems"
require "rack"
rack_app = lambda { |env|
  request = Rack::Request.new(env)
  response = Rack::Response.new
  if request.path_info == '/redirect'
    response.redirect('http://google.com')
  else
    response.headers['Content-Type'] = 'text/plain'
    response.write("a simple html document\n<b>bold text</b> ")
  end
```

```
  response.finish
}
Rack::Handler::WEBrick.run rack_app, :Port=>3000
```

如果去掉

```
    response.headers['Content-Type'] = 'text/plain'
```

那么html标签将无法显示，代之以粗体的**bold text**。

Chapter 3

中间件

什么是中间件？简单地说，就是在Ruby应用服务器和Rack应用程序之间执行的代码。

3.1 一个简单的中间件

回忆图2.2(p. 15)的代码，我们在程序输出的前后分别添加了头和尾部信息。我们可以尝试把实际的程序输出和包装的过程分离开来。首先，去掉2.2(p. 15)中前头后尾的输出，让我们的rack_app更加清晰:

```
#!/usr/bin/env ruby
require "rubygems"
require "rack"
require 'decorator'

rack_app = lambda { |env|
  request = Rack::Request.new(env)
  response = Rack::Response.new
  if request.path_info == '/hello'
    response.write("you say hello")
    client = request['client']
    response.write("from #{client}") if client
  else
    response.write("you need provide some client information")
  end
  response.finish
}
Rack::Handler::WEBrick.run Decorator.new(rack_app), :Port=>3000
```

Figure 3.1: hello.rb

注意2.2(p. 15)最后一行为

```
Rack::Handler::WEBrick.run rack_app, :Port=>3000
```

现在变为:

```
Rack::Handler::WEBrick.run Decorator.new(rack_app), :Port=>3000
```

我们需要定义一个新类Decorator，建立Decorator实例时用原始的rack_app作为参数。这个实例也能够被Rack的handler调用–显然这个实例也是合法的Rack应用程序–因此Decorator需要一个call方法。我们不难得出Decorate大致的样子:

```
class Decorator
 def initialize(app)
   .......
 end
 def call(env)
   .......
 end
end
```

建立下面的decorator.rb文件:

```
1   class Decorator
2    def initialize(app)
3      @app = app
4    end
5    def call(env)
6      status, headers, body = @app.call(env)
7      new_body = "===========header==========<br/>"
8      body.each {|str| new_body << str}
9      new_body << "<br/>===========footer=========="
10     headers['Content-Length'] = new_body.bytesize.to_s
11     [status, headers, [new_body]]
12    end
13  end
```

Figure 3.2: decorator.rb 中间件

运行图3.1(p. 20)的程序。我们可以在浏览器上得到和以前一样的结果,例如，输入http://localhost:3000/hello，得到:

```
===========header==========
you say hello
===========footer==========
```

显然，Decorator的实例在Rack和Rack应用程序的中间运行了某些代码，因此它就是一个中间件。我们不难得出结论，任何中间件本身必须是一个合法的Rack应用程序。

3.2 Rack响应标准

Decorator类的`initialize`方法取app作为参数，图3.1(p. 20)最后一行我们可以知道这是我们原始的`rack_app`对象。`initialize`把这个原始的对象保存在`@app`实例变量。当某一个Handler(例如WEBrick Handler)run时，它最终会调用decorator对象的call方法。代码第6行:

```
6       status, headers, body = @app.call(env)
```

首先调用原始`rack_app`的call方法，得到原始对象的响应并分别赋值到局部变量`status`、`headers`和`body`中。

Rack规格书要求Rack应用程序的call方法返回一个数组，包括三个成员，第一个是状态码，第二个是响应头，第三个是响应体。

状态码(status) 这是一个HTTP状态，它不一定必须是整数，但是它必须能够响应`to_i`方法并返回一个整数，这个整数必须大于等于100。显然，整数本身也是符合规格的。

响应头(headers) 这个头必须能够响应`each`方法，并且每次产生一个key和一个value。显然，Hash可以符合这个条件。对于关键字(key)也有明确的规定，所有的关键字必须是字符串（所以不可以是`symbol`类型)。value也必须是字符串，可以包括多行。响应头中，在状态码为1xx, 204和304的时候，必须没有下面两个key。其他任何时候都必须有这两个key。

Content-Type 内容类型

Content-Length 内容的长度

之所以前面的例子中没有设置`Content-Type`和`Content-Length`，是因为如果不存在这两个关键字的时候，Rack会帮助我们建立缺省的值。而如果这两个关键字本身已经存在，那么你需要保证它们的正确性。

响应体(body) 它必须能够响应`each`方法，而且每次必须产生一个字符串。虽然我们第一章的例子都用了字符串作为例子，但是严格来说，这是不合适的。因为虽然在Ruby 1.8.x中，String对象能够响应each方法:

```
"abc".each {|str| puts str}
```

但是Ruby1.9.x的String已经不支持each方法了。所以通常的做法是用一个字符串数组。

显然，我们只知道原先的body可以响应each，每次产生一个String。因此下面的语句在原先的响应体前后分别加入了头和尾信息，结果放在一个新的字符串`new_body`中。

```
7       new_body = "===========header==========<br/>"
8       body.each {|str| new_body << str}
9       new_body << "<br/>===========footer=========="
```

如果你想确切知道body的类型，可以在第8行之前插入一行:

```
puts body.inspect
```

重新运行程序，输入任何一个url。在控制台可以看到:

```
#<Rack::Response:0x14e9520
     @header={"Content-Type"=>"text/html", "Content-Length"=>"40"},
     @writer=#<Proc:0x00472098@/usr/local/ree/lib/ruby/gems/1.8/gems/rack-1.1.0/
lib/rack/response.rb:26>,
     @body=["you need provide some client information"],
     @status=200, @block=nil, @length=40>
```

这是一个Rack::Response的实例，而不是我们期望的字符串。回到图3.1(p. 20),rack_app对象的最后一句语句是:

```
response.finish
```

生成数组中的第三个成员body恰恰是response自己。

Response的实例是合法的响应体，因为它们能够响应each方法。虽然到目前为止，我们手工生成结果的时候都用一个字符串数组作为响应体，但Rack的规格只需要这个响应体能够响应each方法，并且每次产生一个字符串即可。

最后不要忘记设置正确的'Content-Length:

```
10      headers['Content-Length'] = new_body.bytesize.to_s
11      [status, headers, [new_body]]
```

3.3 为什么中间件

首先，中间件可以实现通用的逻辑和业务逻辑进行分离，而这些通用的逻辑可以被应用到各种各样不同的业务逻辑。

前面我们编写的Decorator中间件就可以应用到任何Rack应用程序:

```
#!/usr/bin/env ruby
require 'rubygems'
require 'rack'
require 'decorator'

Rack::Handler::WEBrick.run
    Decorator.new(lambda {|env|[200, {}], ["whatever rack app"]]}),
    :Port=>3000
```

这是一个生造的例子。但假设我们实现了一个用于用户身份认证的中间件，那么这个中间件就可以应用到任何Rack应用程序。由于几乎所有的Web框架编写的应用程序都是Rack程序，这就意味着任何web应用程序都可以不加修改地使用我们的用户身份认证中间件来实现用户身份认证。这样的例子还有很多很多，我们将在后面进行详细的描述。

Web框架的实现者可以用中间件的形式来实现整个Web框架。由于中间件本身也是合法的Rack应用程序，这就意味着中间件外面还可以包装中间件。原先需要单片实现的整个框架可以被分割成多个中间件，每一个中间件只关心自己需要实现的功能。这样做的好处是显而易见的，(a)每一个中间件都可以独立地发展，甚至可以被独立地替换(b)我们可以用不同方式去组合中间件，以最大程度满足不同应用程序的需要 – 即框架可以根据应用动态配置。

3.4 装配中间件

3.4.1 如何装配

我们往往需要在一个应用程序里面使用多个中间件。最直接的方法是new方法–假设我们有一个应用程序rack_app，有两个中间件类Middleware1和Middleware2，那么我们可以这样使用两个中间件:

```
Rack::Handler::XXXX.run Middleware1.new(Middleware2.new(rack_app))
```

当然，也不排除Middleware1和Middleware2创建实例的时候需要另外的参数，如:

```
Rack::Handler::XXXX.run
     Middleware1.new(
            Middleware2.new(rack_app, options2),
            options1)
```

如果我们要使用很多的中间件，这样的代码会变得越来越冗长。而如果要修改中间件的顺序，则更加是一件繁复而容易出错的工作。

在Ruby里面，我们总是可以用DSL优雅地解决这样的问题。我们可以定义一个类和几个方法，这些方法将变成DSL里面的动词。例如，我们可以这样定义一个Builder类:

```
class Builder
    def use
         ......
    end

    def run
         .......
    end
end
```

以后我们就可以使用use和run作为DSL里面的动词–use使用一个中间件，而run则运行原始的rack应用程序–这些DSL使用的范围通常是一个block，例如:

```
Builder.new {
  use Middleware1
  use Middleware2
  run Rack Application
}
```

可以用来生成一个app，这个app可以被某个Rack::Handler运行。

我们还是来看一个实际的例子。图3.2(p. 21)的中间件Decorator必须记得在生成新的响应体new_body以后设置新的Content-Length:

```
10 headers['Content-Length'] = new_body.bytesize.to_s
```

Rack自带了不少中间件。其中一个中间件Rack::ContentLength能够自动设置响应头中的“Content-Length”。请从decorator.rb文件中删除上面这行代码，然后我们希望下面这样的方法就可以使用中间件和运行app:

```
#!/usr/bin/env ruby
require "rubygems"
require 'rack'
require 'decorator'

app =Builder.new {
  use Rack::ContentLength
  use Decorator
  run lambda {|env| [200, {"Content-Type"=>"text/html"}, ["hello world"]]}
}.to_app

Rack::Handler::WEBrick.run app, :Port => 3000
```

3.4.2 实现Builder

稍加思索，我们对Builder几个方法的要求如下:

initialize 它的签名应该是initialize(&block)，为了能够让use、run这些方法成为DSL语言的动词，initialize应该instance_eval当前实例。

use 它的签名应该是use(middlewareclass, options)，它应该记录需要创建的中间件以及它的顺序。

run 它的签名应该是run (rack_app)，它应该纪录原始的rack应用程序

to_app 根据use和run记录的信息创建出最终的应用程序

通常有两类途径来实现这些方法。

传统方法

一类是比较传统的方法，用数组记录所有需要创建的中间件的信息，最后to_app时候把它们创建出来:

```
class Builder
  def initialize(&block)
    @middlewares = []
    self.instance_eval(&block)
  end
  def use(middleware)
    @middlewares << middleware
  end
  def run(app)
    @app = app
  end
  def to_app
    app = @app
    @middlewares.reverse.each do |middleware|
      app = middleware.new(app)
    end
    app
  end
end
```

当然可以直接用inject方法简化to_app:

```
  def to_app
    @middlewares.reverse.inject(@app) { |app, middleware|  middleware.new(app)}
  end
```

完整的test-builder.rb文件:

```
#!/usr/bin/env ruby
require "rubygems"
require 'rack'
require 'decorator'
require 'builder'

app = Builder.new {
  use Rack::ContentLength
  use Decorator
  run lambda {|env| [200, {}, ["hello world"]]}
}.to_app
```

```
Rack::Handler::WEBrick.run app, :Port => 3000
```

以及删除对Content-Length进行设置以后的Decorator中间件文件decorator.rb:

```
class Decorator
 def initialize(app)
   @app = app
 end
 def call(env)
   status, headers, body = @app.call(env)
   new_body = "===========header==========<br/>"
   body.each {|str| new_body << str}
   new_body << "<br/>===========footer=========="
   [status, headers, [new_body]]
  end
 end
```

运行./test-builder.rb，我们可以看到Decorator中间件已经被正确地使用。

如果你仔细观察to_app的实现，可以看到我们首先对加入的middlewares进行了reverse。对所有使用的中间件，我们必须持有它们的顺序信息，第一个被use的中间件包在最外面一层，它包含了第二个被use的中间件，接着包含第三个被use的中间件，等等等等，直至包含了原始的Rack应用程序。如果我们改变了中间件使用的顺序，那么就有可能产生不同的结果。例如修改test-builder.rb中两句use的顺序为:

```
app = Builder.new {
  use Decorator
  use Rack::ContentLength
  run lambda {|env| [200, {"Content-Type"=>"text/html"}, ["hello world"]]}
}.to_app
```

重新运行并在浏览器输入http://localhost:3000，你会发现浏览器的结果只显示了一部分:

```
===========
```

原因是Rack::ContentLength设置了原始Rack应用程序的内容长度，在它外面的Decorator增加了内容，但是却没有再设置内容长度，从而导致浏览器只取到部分的内容。

更Ruby化的方法

上面这种“传统”的方法有自己的局限性。例如，如果我们需要在use中间件的时候带上一些选项，甚至执行某些代码。实际上use描述的是中间件创建的过程，这个

创建过程需要自己的参数，需要执行某些代码–但是这个创建过程并不是现在就要被执行，而是在后面(to_app)时候被执行。

对那些需要在以后执行的代码，Ruby给出更好的答案是lambda。

```
class Builder
  def initialize(&block)
    @middlewares = []
    self.instance_eval(&block)
  end
  def use(middleware_class,*options, &block)
    @middlewares << lambda {|app| middleware_class.new(app,*options, &block)}
  end
  def run(app)
    @app = app
  end
  def to_app
    @middlewares.reverse.inject(@app) { |app, middleware| middleware.call(app)}
  end
end
```

use方法把中间件的创建过程以lambda的方式保存在@middlewares数组中，而中间件的创建过程就是以app为参数创建一个新的app:

`lambda {|app| middleware_class.new(app,*options, &block)}`

修改decorator.rb和test-builder.rb，为我们的中间件加上参数:

```
class Decorator
 def initialize(app, *options, &block)
   @app = app
   @options = (options[0] || {})
 end
 def call(env)
   status, headers, body = @app.call(env)
   new_body =  ""
   new_body << (@options[:header] ||"===========header==========<br/>")
   body.each {|str| new_body << str}
   new_body << (@options[:footer] || "<br/>===========footer==========")
   [status, headers, [new_body]]
  end
 end
```

```
#!/usr/bin/env ruby
require "rubygems"
require 'rack'
```

```
require 'decorator'
require 'builder'

app = Builder.new {
  use Rack::ContentLength
  use Decorator , :header =>  "****************header****************<br/>"
  run lambda {|env| [200, {"Content-Type"=>"text/html"}, ["hello world"]]}
}.to_app

Rack::Handler::WEBrick.run app, :Port => 3000
```

你可以得到不一样的效果。

Chapter 4

最简单的Web框架

Rack为编写Web程序和Web框架提供很多有用的设施。考虑一个最简单的Web框架，它提供:

- 对Request和Response的存取
- 能够根据不同的URL执行不同的程序-即所谓的路由
- 能够处理cookie信息
- 能够存取用户会话-即Session
- 能够生成日志
- ...

你可能会觉得困难。事实上，Rack自带了这样一个框架-rackup。

4.1 Rack::Builder

我们之前的3.4.2(p. 25)构造了一个Builder。Rack自己就有这样的一个Rack::Builder。除了我们先前实现的use, run方法外，Rack::Builder还利用了Rack::URLMap来处理路由。

4.1.1 替换为Rack::Builder

用Rack::Builder重写`test-builder.rb`，只需要去掉`require builder`一行，并把`Builder.new`改为`Rack::Builder.new`:

```
#!/usr/bin/env ruby
require "rubygems"
require 'rack'
```

```
require 'decorator'


app = Rack::Builder.new {
  use Rack::ContentLength
  use Decorator , :header => "==============<br/>"
  run lambda {|env| [200, {"Content-Type"=>"text/html"}, ["hello world"]]}
}.to_app

Rack::Handler::WEBrick.run app, :Port => 3000
```

把这些文件保存为test-rack-builder.rb，运行应该得到和原先一样的结果。

4.1.2 路由

一个Web程序通常用不同的代码处理不同的URL，很多Web应用程序把这种对应关系的处理叫做路由。最简单的路由就是一个路径和一个代码块之间的一一对应关系。

一个简单的路由

利用Rack::Builder的map方法我们可以这样编写一个Rack程序：

```
#!/usr/bin/env ruby
require "rubygems"
require 'rack'

app = Rack::Builder.new {
  map '/hello' do
      run lambda {|env| [200, {"Content-Type" => "text/html"}, ["hello"]] }
  end
  map '/world' do
      run lambda {|env| [200, {"Content-Type" => "text/html"}, ["world"]] }
  end
  map '/' do
      run lambda {|env| [200, {"Content-Type" => "text/html"}, ["all"]] }
  end
}.to_app

Rack::Handler::WEBrick.run app, :Port => 3000
```

当你输入任何以http://localhost:3000/hello开始的URL，浏览器都可以得到hello。当你输入任何以http://localhost:3000/world开始的URL，浏览器都可以得到world。除此之外，你将得到all。

路由实现

use和run方法

Rack::Builder的具体实现大体上和3.4.2(p. 25)描述的一致。

```
def initialize(&block)
  @ins = []
  instance_eval(&block) if block_given?
end

def use(middleware, *args, &block)
  @ins << lambda { |app| middleware.new(app, *args, &block) }
end

def run(app)
  @ins << app #lambda { |nothing| app }
end
```

和我们自己实现的builder不同之处在于我们用一个单独的@app实例变量来保存run的参数–即原始的Rack应用程序，而这里的run直接把app放到数组的最后。所以这个数组的成员依次包含所有的中间件，最后一个成员是将被前面所有这些中间件包装的Rack应用程序。

```
def to_app
  @ins[-1] = Rack::URLMap.new(@ins.last)  if Hash === @ins.last
  inner_app = @ins.last
  @ins[0...-1].reverse.inject(inner_app) { |a, e| e.call(a) }
end
```

to_app首先取得@ins数组的最后一个成员，如果最后一个成员不是一个Hash的话，实现的效果就和我们的Builder完全一样了。

map方法

所以不同之处在于最后一个成员是Hash的情况：如果最后一个成员是Hash，那么就会根据这个Hash生成一个Rack::URLMap的实例，这个实例作为被其他中间件包装的Rack应用程序。这个Hash是map方法产生的。

```
def map(path, &block)
  if @ins.last.kind_of? Hash
    @ins.last[path] = self.class.new(&block).to_app
  else
    @ins << {}
    map(path, &block)
  end
end
```

map方法取一个路径path和一个代码块block为参数。当@ins的最后一个成员不是Hash的时候，就加入一个新的Hash在@ins的末尾。由于to_app方法总是把最后一个成员作为被前面所有中间件包装的Rack应用程序，由此可以看出，如果在Builder.new的代码块出现了一个map的话，那么不可以在相同的范围内出现run,也就是说，下面这样的情况是不合法的:

```
Rack::Builder.new {
  use ....
  use....
  run ....
  map ... do
      .....
  end
}
```

回到前面的map方法。考虑到第一次调用map的情况，程序首先在@ins内部加入一个空的Hash，然后递归调用map方法。由于此时@ins数组最后一个成员已经是一个Hash，所以下面的语句建立了一个对应关系。

```
        @ins.last[path] = self.class.new(&block).to_app
```

这个对应关系的关键字是path参数，但它的值并非代码块本身，而是用这个代码块作为参数继续调用Rack::Builder.new方法，并用to_app方法产生一个Rack应用程序。假设我们有这样一个Rack::Builder的构造过程:

```
#!/usr/bin/env ruby
require "rubygems"
require 'rack'

app = Rack::Builder.new {
  use Rack::ContentLength
  map '/hello' do
      run lambda {|env| [200, {"Content-Type" => "text/html"}, ["hello"]] }
  end
}.to_app

Rack::Handler::WEBrick.run app, :Port => 3000
```

那么现在@ins数组将包括两个成员：一个是创建中间件Rack::ContentLength对应的lambda对象，最后一个是Hash，其中包含了路径/hello对应的一个Rack应用程序–这个应用将调用我们用run运行的lambda对象:

```
 lambda {|env| [200, {Content-Type => text/html}, [hello]] }
```

如果我们继续声明map:

```
#!/usr/bin/env ruby
require "rubygems"
```

```
require 'rack'

app = Rack::Builder.new {
  use Rack::ContentLength
  map '/hello' do
      run lambda {|env| [200, {"Content-Type" => "text/html"}, ["hello"]] }
  end
  map '/world' do
      run lambda {|env| [200, {"Content-Type" => "text/html"}, ["world"]] }
  end
}.to_app

Rack::Handler::WEBrick.run app, :Port => 3000
```

则现在@ins数组还是只有二个成员：第一个中间件不变，最后一个是Hash，有了两个对：

'hello' => lambda {|env| [200, {Content-Type => text/html}, [hello]] }
'world' => lambda {|env| [200, {Content-Type => text/html}, [world]] }

回到to_app方法：

```
    def to_app
      @ins[-1] = Rack::URLMap.new(@ins.last)  if Hash === @ins.last
      inner_app = @ins.last
      @ins[0...-1].reverse.inject(inner_app) { |a, e| e.call(a) }
    end
```

如果最后一个成员是一个Hash，将会用这个成员创建一个新的Rack::URLMap应用程序。Rack::URLMap内部保存了这个URL和Rack程序之间的对应关系，如果用户在url输入了http://localhost:3000/hello开始的URL，那么将调用第一个应用程序。当它同时也作了一些处理，这个匹配的路径'/hello'将变为环境里面的SCRIPT_NAME，而截取的剩余部分则变为PATH_INFO。如果我们修改程序如下：

```
#!/usr/bin/env ruby
require "rubygems"
require 'rack'

app = Rack::Builder.new {
  map '/hello' do
      run lambda {|env| [200, {"Content-Type" => "text/html"},
        ["SCRIPT_NAME=#{env['SCRIPT_NAME']}", "PATH_INFO=#{env['PATH_INFO']}"]] }
  end
  map '/world' do
      run lambda {|env| [200, {"Content-Type" => "text/html"}, ["world"]] }
  end
```

```
}.to_app

Rack::Handler::WEBrick.run app, :Port => 3000
```

则在浏览器输入http://localhost:3000/hello/得到:

SCRIPT_NAME=/helloPATH_INFO=/

而在浏览器输入http://localhost:3000/hello/everyone得到:

SCRIPT_NAME=/helloPATH_INFO=/everyone

这样做的目的是你可以在这个/hello"应用程序"内部实现你自己的分派。

嵌套map

回忆到map方法的实现:

```
def map(path, &block)
       if @ins.last.kind_of? Hash
          @ins.last[path] = self.class.new(&block).to_app
       .......
```

我们递归地用Builder.new创建保存到hash所需要的Rack应用程序，这意味着我们还可以在map内部使用use, run，甚至是嵌套的map。

```
#!/usr/bin/env ruby
require "rubygems"
require 'rack'

app = Rack::Builder.new {
  use Rack::ContentLength
  map '/hello' do
      use Rack::CommonLogger
      map '/ketty' do
        run lambda {|env| [200, {"Content-Type" => "text/html"},
          ["from hello-ketty",
           "SCRIPT_NAME=#{env['SCRIPT_NAME']}",
           "PATH_INFO=#{env['PATH_INFO']}"]] }
      end
      map '/everyone'  do
        run lambda {|env| [200, {"Content-Type" => "text/html"},
          ["from hello-everyone",
          "SCRIPT_NAME=#{env['SCRIPT_NAME']}",
           "PATH_INFO=#{env['PATH_INFO']}"]] }
      end
      map '/' do
```

```
      run lambda {|env| [200, {"Content-Type" => "text/html"},
        ["from hello catch all",
        "SCRIPT_NAME=#{env['SCRIPT_NAME']}",
        "PATH_INFO=#{env['PATH_INFO']}"]] }
    end
  end
  map '/world' do
      run lambda {|env| [200, {"Content-Type" => "text/html"}, ["world"]] }
  end
  map '/' do
      run lambda {|env| [200, {"Content-Type" => "text/html"}, ["here"]] }
  end

}.to_app

Rack::Handler::WEBrick.run app, :Port => 3000
```

4.2 rackup

我们讨论的应用程序最后一行都是用一个handler去运行一个app，带上某些参数。显然作为一个Web框架这样做是不合适的。

4.2.1 rackup配置文件

Rack提供的最简单的rackup命令允许用一个配置文件去运行我们的应用程序。

rackup做的事情很简单，如果你提供一个配置文件`config.ru`(你可以取任何名字，但后缀必须为ru)，然后运行

```
rackup config.ru
```

那么它所做的事情相当于:

```
app = Rack::Builder.new { ... 配置文件 ... }.to_app
```

然后运行这个app。

把前面的程序改成

```
map '/hello' do
  run lambda {|env| [200, {"Content-Type" => "text/html"},
    ["SCRIPT_NAME=#{env['SCRIPT_NAME']}", "PATH_INFO=#{env['PATH_INFO']}"]] }
  end
map '/world' do
    run lambda {|env| [200, {"Content-Type" => "text/html"}, ["world"]] }
end
```

并保存到文件config.ru，运行rackup config.ru即可。

你可以看到我们去掉了对rubygems和rack的引入，不再需要硬编码什么Web服务器，也不须要在程序中指定端口。

rackup提供了一些命令行参数:

```
rackup --help
Usage: rackup [ruby options] [rack options] [rackup config]

Ruby options:
  -e, --eval LINE           evaluate a LINE of code
  -d, --debug               set debugging flags (set $DEBUG to true)
  -w, --warn                turn warnings on for your script
  -I, --include PATH        specify $LOAD_PATH (may be used more than once)
  -r, --require LIBRARY     require the library, before executing your script

Rack options:
  -s, --server SERVER       serve using SERVER (webrick/mongrel)
  -o, --host HOST           listen on HOST (default: 0.0.0.0)
  -p, --port PORT           use PORT (default: 9292)
  -E, --env ENVIRONMENT     use ENVIRONMENT for defaults (default: development)
  -D, --daemonize           run daemonized in the background
  -P, --pid FILE            file to store PID (default: rack.pid)

Common options:
  -h, --help                Show this message
      --version             Show version
```

你可以指定rackup运行的web服务器以及端口。例如:

`rackup -s thin -p 3000`

当然，你必须安装Thin服务器。

4.2.2 rackup 实现

我们要看看rackup是如何实现的，借此了解一个基于Rack的Web框架的实现，将对我们后面实现自己的Web框架大有好处。

rackup本身的实现只有一句语句:

```
#!/usr/bin/env ruby

require "rack"
Rack::Server.start
```

显然Rack::Server才是我们的重点。

4.2.3 Rack::Server接口

Rack::Server的接口非常简单，包括两个类方法、一个构造函数和5个实例方法。

```
module Rack
  class Server

    def self.start
    def self.middleware

    def initialize(options = nil)
    def options
    def app
    def middleware
    def start
    def server

  end
end
```

类方法

类方法start是Rack::Server的入口，它只不过创建一个新的server实例，并调用它的start实例方法。

```
    def self.start
      new.start
    end
```

另一个类方法装配一些缺省的中间件:

```
    def self.middleware
      @middleware ||= begin
        m = Hash.new {|h,k| h[k] = []}
        m["deployment"].concat
          [lambda {|server| server.server =~ /CGI/ ? nil : [Rack::CommonLogger, $stderr] }]
        m["development"].concat
         m["deployment"] + [[Rack::ShowExceptions], [Rack::Lint]]
        m
      end
    end
```

rackup 根据不同的环境（可以用-E开关选择环境)装载不同的中间件:

- 对于缺省的development环境，它会装载ShowExceptions和Lint中间件。

- 对于deployment环境，它会装载ShowExceptions、Lin和CommonLogger中间件。

由于CommonLogger写到`$stderr`,和所以无法和CGI服务器配合，所以对CGI服务器而言，CommonLogger将不会被加载。

@middleware是一个Hash，它的key是环境的名字，它的值是一个数组，其中包含对应环境需要预加载的所有中间件类。要注意的是数组的每一个中间件成员还是一个数组，其中第一个成员是中间件类，而后面的成员则为实例化这个类所需要的参数，例如:

`[Rack::CommonLogger, $stderr]`

意味着将会以`Rack::CommonLogger.new($stderr)`的方式来构造此中间件。

实例方法

`start`是最重要的方法，但是它依赖其他几个方法来实现自己的功能。

options

```
def options
  @options ||= parse_options(ARGV)
end
```

当然最重要的是`parse_options`解析我们在命令行传入的参数。`parse_options`把缺省的参数和命令行传入的参数进行合并，最后返回一个Hash。譬如，如果我们在命令行输入了:

`rackup -s Thin config.ru`

那么`options`将包含`:server=>'Thin',:config=>'config.ru'`这两个关键字、值对。

app

```
1   def app
2    @app ||= begin
3       if !::File.exist? options[:config]
4        abort "configuration #{options[:config]} not found"
5       end
6
7     app, options = Rack::Builder.parse_file(self.options[:config], opt_parser)
8       self.options.merge! options
9       app
10     end
11  end
```

我们知道options[:config]包含了配置文件名。所以3-5是检查这个文件是否存在。最重要的是第7行，它利用Rack::Builder读取配置文件，并创建出一个app。如果你查看Rack::Builder文件，你可以看到:

```
  class Builder
    def self.parse_file(config, opts = Server::Options.new)
        ........
        app = eval "Rack::Builder.new {( " + cfgfile + "\n )}.to_app",
        ..........
    end
```

这个我们已经很熟悉了。至于为何parse_file返回一个options，这是因为Rack::Builder还允许你在配置文件的开头加上选项。如果一个config.ru的第一行是以#\开始的，那么这一行就是选项行。例如你可以这样指定服务器运行的端口，并打开警告。

```
#\ -w -p 8765
run lambda {|env| [200, {"Content-Type" => "text/html"}, ['hello'] ]}
```

sever

```
    def server
      @_server ||= Rack::Handler.get(options[:server]) || Rack::Handler.default
    end
```

它根据我们在命令行上配置的 -s选项获得对应的Rack::Handler，如果没有指定则为Rack::Handler.default，即Rack::Handler::WEBrick。

sever

```
    def middleware
      self.class.middleware
    end
```

无它，调用类方法的middleware而已。

build_app

最后我们还需要了解一个私有的方法build_app。

```
1       def build_app(app)
2         middleware[options[:environment]].reverse_each do |middleware|
3           middleware = middleware.call(self) if middleware.respond_to?(:call)
4           next unless middleware
5          klass = middleware.shift
6           app = klass.new(app, *middleware)
7         end
8         app
9       end
10
11       def wrapped_app
12        @wrapped_app ||= build_app app
13       end
```

传入的参数app是Rack::Server利用Rack::Builder构造出来的应用程序。

`middleware[options[:environment]]`获得对应环境的预加载中间件，之所以需要`reverse_each`的原因和我们之前已经在3.4.2(p. 25)中讨论过。

回忆我们讨论的类方法`middleware`，某一个环境所有需要预加载的中间件是一个数组，数组的每一个成员各自代表一个中间件类-它有两种可能:

- 形如`lambda {|server| server.server =~ /CGI/ ? nil : [Rack::CommonLogger, $stderr] }`，为一个lambda，调用这个lambda可能得到nil或者得到一个数组
- 一个数组，有一个或多个成员，第一个成员是中间件的类，其他成员是实例化中间件需要的参数

build_app方法的第3-4行处理第一种情况，然后用5-6行处理第二种情况。

start

现在理解`start` 已经很容易了。

```
def start
  if options[:debug]
    $DEBUG = true
    require 'pp'
    p options[:server]
    pp wrapped_app
    pp app
  end

  if options[:warn]
    $-w = true
  end

  if includes = options[:include]
    $LOAD_PATH.unshift *includes
  end

  if library = options[:require]
    require library
  end

  daemonize_app if options[:daemonize]
  write_pid if options[:pid]
  server.run wrapped_app, options
end
```

除了一些参数的处理外，最重要的就是最后一句语句:

```
    server.run wrapped_app, options
```

这就好比:

`Rack::Handler:XXX.run app, options`。

4.3 没有了？

我们就要结束关于rackup的描述。

细心的读者会发现rackup没有实现我们最初承诺的会话、日志等等功能，这要依赖于我们下一章要进一步讨论的中间件。

Chapter 5

中间件:第二轮

5.1 再议响应体

我们首先在2.3.1(p. 14)中提到了如何设置响应体，然后在3.2(p. 20)提到了响应体必须能够响应each方法，现在我们将对响应体做更加深入的探讨。

Rack的规格书中对响应体的说明如下:

> **The Body**
> The Body must respond to `each` and must only yield String values. The Body itself should not be an instance of String, as this will break in Ruby 1.9. If the Body responds to `close`, it will be called after iteration. If the Body responds to `to_path`, it must return a String identifying the location of a file whose contents are identical to that produced by calling each; this may be used by the server as an alternative, possibly more efficient way to transport the response. The Body commonly is an Array of Strings, the application instance itself, or a File-like object.

意思可以分几点:

- 对响应体唯一的要求是必须能够响应each方法，each每次只能产生字符串值
- 由于Ruby 1.9已经不支持String.each方法，所以响应体不应该是一个字符串
- 响应体可以响应一个to_path方法，如果确实如此的话，那么这个方法应该返回一个文件的路径名，这可以更加高效地处理文件的情况
- 响应体通常是字符串数组、应用程序自己或者类File对象。
- 如何响应体能够响应`close`方法，在each迭代完成后应该调用`close`方法。我们可以在这里实现一些清除工作，例如关闭文件等。

我们可以写一个响应体是File对象的例子。

```
use Rack::ContentLength
use Rack::ContentType, "text/plain"
run  lambda {|env| [200, {}, File.new(env['PATH_INFO'][1..-1])] }
```

把程序保存为file.ru，并执行rackup file.ru，然后在浏览器输入http://localhost:9292/file.ru，我们可以得到file.ru文件的内容。之所以如此是因为我们的响应体是一个File对象，因此它能够响应each方法，Handler会不断调用each方法，并把每次得到的一个字符串输出。所以each方法有很多好处:

- 响应体可以是任何类的任何对象，只要它必须能够响应each方法，each每次只能产生字符串值，这给我们很多实现的可能性和灵活性
- 可以减少资源的损耗和不必要的处理工作，例如使用File对象的好处可以让我们不需要一次读入整个文件的所有内容，拼装成一个字符串，然后一次性输出。不然的话，如果文件足够大，我们的系统必然会因为内存的消耗殆尽而崩溃。
- 可以让Web服务器有更多的机会和选择进行优化，例如根据不同的内容类型可以选择每次each输出、一次性输出或者缓存到一定大小再输出整个响应的内容。

现在我们再来考虑为什么响应体通常也可能是应用程序自身。

我们把3.2(p. 21)重新取到这里。

```
1 class Decorator
2 def initialize(app)
3    @app = app
4 end
5 def call(env)
6    status, headers, body = @app.call(env)
7    new_body = "===========header==========<br/>"
8    body.each {|str| new_body << str}
9    new_body << "<br/>===========footer=========="
10   [ status, headers, [new_body]]
11 end
12 end
```

我们去掉了3.2(p. 21)设置'Content-Length'相关的代码–这可以交给其他中间件去做。

这个中间件看起来没有什么问题。但是假设我们原始的Rack应用程序的响应体是一个文件，即在前面的config.ru文件中加入对Decorator的使用:

```
use Rack::ContentLength
use Rack::ContentType, "text/plain"
use Decorator
run  lambda {|env| [200, {}, File.new(env['PATH_INFO'][1..-1])] }
```

Decorator的第8行把所有的文件内容都取到一个new_body字符串，当文件非常庞大的时候，这显然是不可行的。要解决这个问题，我们只需要修改Decorator中间件，增加一个each方法，然后把自己作为响应体返回:

```
class Decorator
 def initialize(app, *options, &block)
   @app = app
   @options = (options[0] || {})
 end
 def call(env)
   status, headers, @body = @app.call(env)
   @header =  (@options[:header] ||"===========header==========<br/>")
   @footer =  (@options[:footer] || "<br/>===========footer==========")
   [status, headers, self]
  end

  def each(&block)
    block.call @header
    @body.each(&block)
    block.call @footer
  end
 end
```

each方法首先用@header调用代码块，然后用调用早先得到的@body对象迭代调用代码块，最后输出@footer。

5.2 Rack自带中间件

Rack本身提供了很多中间件，涉及到Web开发的各个方面，包括:

- 处理HTTP协议相关的中间件，这些中间件可以成为Web框架的基石
- 程序开发相关，例如代码重载、日志、规格检查等等
- 处理Web应用程序经常需要处理的问题，包括session、文件等

某些中间件可以缩短我们应用程序的开发时间–它们是对于Web应用开发常见问题的通用解决方案–例如静态文件的处理。而熟悉其他一些中间件可以让我们快速深入地理解很多Web框架–这里的某些中间件已经成为不少Web框架(如Rails)的标准配件。当然，如果你要开发新的Web框架，这里的绝大多数中间件都可以成为你随手可用的组件。

5.3 HTTP协议中间件

5.3.1 Rack::Chunked

HTTP协议有一种分块传输编码的机制(*Chunked Transfer Encoding*)，即一个HTTP消息可以分成多个部分进行传输。它对HTTP请求和HTTP响应都是适用的。我们在这里主要考虑从服务器向客户端传输的响应。

一般来说，HTTP服务一次性地把所有的内容都传输给客户端，这个内容的长度在'Content-Length'头字段中声明。之所以需要这个字段的原因是客户端需要知道响应到什么地方结束了。但在某些时候，服务端可能预先不知道将要传输的内容大小或者因为性能的原因不希望一次性生成并传输所有的响应(压缩是这样一个例子)，那么它就可以利用这种分块传输的机制一块一块地传输数据。一般来说，这些"块"的大小是一样的，但这并不是一个强制性的要求。

要利用分块传输机制，服务器首先要写入一个`Transfer-Encoding`头字段并令它的值为"'chunked'"，每一块的内容包括二个部分(CRLF表示回车加换行):

1. 一个16进制的值表示块的大小，后跟一个`CRLF`
2. 数据本身后跟一个`CRLF`

最后一个块只需一行，它的块大小为0，最后整个HTTP消息以一个`CRLF`结束。下面是整个HTTP消息的一个例子:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked

25
This is the data in the first chunk

1C
and this is the second one

0
```

注意最后一个0以后还有一个空行

现在Rack::Chunked的代码就很容易理解了:

```
  def call(env)
    status, headers, body = @app.call(env)
    headers = HeaderHash.new(headers)
```

```
    if env['HTTP_VERSION'] == 'HTTP/1.0' ||
       STATUS_WITH_NO_ENTITY_BODY.include?(status) ||
       headers['Content-Length'] ||
       headers['Transfer-Encoding']
      [status, headers, body]
    else
      dup.chunk(status, headers, body)
    end
  end
```

HeaderHash是一个Hash的子类，它的key对大小写不敏感,但是内部保存的key保持原来的大小写。

```
require 'rubygems'
 require 'rack'
include
h = Rack::Utils::HeaderHash.new({})
h["abc"] = "234"    #=> "234"
h["ABC"]            #=> "234"
h.keys              #=> ["abc"]
```

它可以用来方便地存取HTTP的头信息。

`call`方法首先判断当前的`HTTP_VERSION`是否1.0，或者状态为`STATUS_WITH_NO_ENTITY_BODY`，或者headers里面是否已经包含`Content-Length`和`Transfer-Encoding`头字段，如果任何一种情况存在，则Rack::Chunked不做任何事情，不然的话就调用

`dup.chunk(status, headers, body)`

chunked方法是典型的返回self作为响应体的例子:

```
  def chunk(status, headers, body)
    @body = body
    headers.delete('Content-Length')
    headers['Transfer-Encoding'] = 'chunked'
    [status, headers, self]
  end
```

这意味着Rack::Chunked必定有一个`each`方法:

```
  def each
    term = "\r\n"
    @body.each do |chunk|
      size = bytesize(chunk)
      next if size == 0
      yield [size.to_s(16), term, chunk, term].join
    end
    yield ["0", term, "", term].join
  end
```

each方法完全遵守我们前面讲到如何进行分块编码输出的HTTP协议标准：每一块首先输出块的大小加一个CRLF，接着是具体的内容加一个CRLF，最后是一个0和一个CRLF。

最后Rack::Chunked还定义了一个close，Rack规格书中规定如果一个body能够响应close方法，那么迭代之后会被调用到:

```
def close
  @body.close if @body.respond_to?(:close)
end
```

很多Web服务器的Rack Handler都会自动加载Rack::Chunked中间件，我们会在后面谈到。

5.3.2 Rack::ConditionalGet

HTTP协议定义了一种缓存机制，当客户端请求某个资源时，如果服务端发现该资源未被修改，那么就可以告诉客户端内容未被修改，可以使用客户端自己缓存的内容。这样做有很多好处，例如服务端可以不必重新生成内容，可以不必重新传输内容，不论对CPU还是网络资源来说，这都是很大的节约。HTTP通过条件获取*Conditional GET*请求来实现这种缓存机制。

当浏览器第一个请求某一个url的时候，由于没有任何缓存数据，所以它直接发送请求:

```
GET /example.html HTTP/1.1
Host: www.mydomain.com
```

缓存控制

最好的缓存效果当然是请求可以不发送到服务器，客户端直接从本地缓存取到内容。如果服务端希望对缓存进行控制，服务端响应会包含一个头信息叫做Cache-Control。例如:

```
Cache-Control: max-age=21600
```

这个例子中，Cache-Control包含一个max-age值，这个值表示过期的秒数。客户端把文档内容保存在自己的缓存中，并记录该文档缓存过期的时间。在内容被缓存开始到max-age指定的秒数内，客户端可以认为这个文档的内容是新鲜的(即服务端此文档没有改变)，因此在此期间对同一文档的后续请求就不需要和服务端进行交互了。客户端可以直接从缓存取得并进行展示。

> 因此，对于静态内容来说，服务端应该发送一个 Cache-Control: max-age=…，并且把这个max-age的值设置得很大。但是我们无法确定到底设置max-age的值多大才是合适的，因此通常的做法是把max-age的值设到最大值。

但是这样做这会造成另外一个困境，即万一我们的静态内容真的发生了改变，浏览器几乎永远得不到最新的内容了。这里有个小诀窍就是使用所谓的“智能URL”，服务端在生成对任何静态文件的引用时，都在实际的URL后面加上一个代表最后修改时间的值，例如:

```
http://www.mydomain.com/javascripts/jquery.js?1263983825
```

客户端得到这个文件时，它的缓存就被设为max-age的最大值–即几乎永不过期。如果这个文件确实从未被修改过，那么任何一次获得这个文件的URL都是上面的URL。而如果这个文件被改变了，那么服务端在生成对对它的引用时，会附加一个不同的时间值。例如:

```
http://www.mydomain.com/javascripts/jquery.js?1373486878
```

这对客户端来说是一个新的URL，所以它就会重新到服务器去获取。

这种方法特别适合于资源文件，包括图片、CSS、JavaScript和静态的页面等等。

`Cache-Control`和`max-age`还有其他更多的选项和语义，我们不再这里一一列出了，请参考http://www.w3.org/Protocols/rfc2616.

如果文档缓存过期了(超过了max-age指定的时间范围)，那么必须到服务器进行重新确认(*revalidation*)。

- 如果验证的结果是“文档未改变”，则可以继续使用缓存的文档(验证会返回一些头信息，包括新的过期时间等等，客户端只需改变缓存中的头信息)。
- 如果验证的结果是“文档改变了”，则得到新的文档内容，用于更新缓存。

除了缓存过期外，还有很多原因可能造成需要重新确认,包括:

- 用户强制刷新客户端的时候，不管缓存的文档是否新鲜，都会重新确认。
- 另外一些Cache-Control指令。我们只讨论了max-age，Cache-Control还有很多指令。例如must-revalidate指令告诉客户端任何时候必须去重新确认文档，所以一般动态内容都会使用此指令。

重新确认的方法是做一个条件获取请求。有两种方法可以实现条件获取请求，分别基于`Last-Modified`和`Etag`。决定了不同的方法后，服务器可能返回不同的响应头信息。

基于Last-Modified

此时服务端第一次响应的HTTP消息，除了消息体、Cache-Control头字段等内容之外，还包括一个`Last-Modified`头字段，它的值是文档最后修改的时间，例如:

```
Last-Modified: 20 Sep 2008 18:23:00 GMT
```

客户端在发送*条件获取请求*时会包括以下的请求头字段:

```
If-Modified-Since: 20 Sep 2008 18:23:00 GMT
```

服务端接受到此请求以后，就会比较被请求文档的最后修改时间和`If-Modified-Since`指定的时间。如果已经发生改变，则最新的文档被正常发送。而如果没有改变，那么服务器会返回一个 304 Not Modified 头。

如果我们能够很方便地计算出一个文档的最后修改时间，那么`Last-Modified`是非常适合的。但是要注意的是，如果页面内容是组合许多片段产生的，那么我们就需要计算所有片段的最后修改时间，并取最新的最后修改时间作为整个文档的最后修改时间。如果有多个Web服务器同时服务，我们也必须保证这些服务器之间的时间是同步的。

基于ETag

有的时候，文档的内容来自很多片段的组合(例如不同的数据库记录)，文档的最后修改时间很难计算，或者说最后修改时间并非决定文档内容是否改变的主要因素。这个时候我们可以用Etag来比较。Etag的值通常依赖于文档的真正内容，我们必须保证不同的内容产生不同的Etag，相同的内容产生相同的Etag。

采用这种方法时，服务器第一次响应时会包含一个Etag值，例如:

```
Etag: 4135cda4de5f
```

客户端在发送条件获取请求时会包括以下的请求头字段:

```
If-None-Match: 4135cda4de5f
```

服务端接受到此请求以后，首先根据被请求文档的实际内容生成一个Etag，如果生成的Etag和请求的Etag不同，则表示内容已经发生改变，最新的文档被正常发送。而如果没有改变，那么服务器返回一个 304 Not Modified 头。

组合Etag和Last-Modified

服务端完全可以组合Etag和Last-Modified。如果客户端从服务端第一次获得的响应头中同时包含这两个头字段，那么在后面验证文档缓存的时候会在请求头中同时包含If-Modified-Since和If-None-Match头字段。

服务器如何判断文档是否改变取决于不同的实现。对Rack::ConditionalGet而言，只需要其中任何一个匹配即可，但对于另外一个中间件、框架或者Web服务器而言，它可能需要这两个条件同时匹配。

RFC2616规定304响应必须没有消息体，在所有的头信息之后跟上一个空行表示响应结束。

Rack::ConditionalGet实现

Rack::ConditionalGet所做的事情就是查看当前的响应是否代表一个未改变的文档，从而设置响应的响应头和响应体。

如果下面两个条件中任何一个满足，就可以认为请求的内容没有改变:

- Etag匹配：请求头中的HTTP_IF_NONE_MATCH值等于响应头中的Etag
- Last-Modified匹配：请求头中的HTTP_IF_MODIFIED_SINCE值等于响应头中的Last-Modified

下面代码中的headers参数是响应头:

```
  def etag_matches?(env, headers)
     etag = headers['Etag'] and etag == env['HTTP_IF_NONE_MATCH']
   end

   def modified_since?(env, headers)
     last_modified = headers['Last-Modified'] and
       last_modified == env['HTTP_IF_MODIFIED_SINCE']
   end
 end
```

确定了这是一个未改变的文档之后，ConditionalGet会做三件事情:

- 把响应状态码设置为304
- 清除响应头中的Content-Type和Content-Length
- 把响应体清空

下面的代码应该很好理解了:

```
def call(env)
  return @app.call(env) unless
            %w[GET HEAD].include?(env['REQUEST_METHOD'])

  status, headers, body = @app.call(env)
  headers = Utils::HeaderHash.new(headers)
  if etag_matches?(env, headers) || modified_since?(env, headers)
    status = 304
    headers.delete('Content-Type')
    headers.delete('Content-Length')
    body = []
  end
  [status, headers, body]
end
```

ConditionalGet要求程序自己合适的Etag或者Last-Modified响应头，它才能在此基础上做适当的处理。

我们可以看看如何使用Etag:

```
use Rack::ConditionalGet

class MyApp
  def call(env)
    [200, {"Content-Type"=>"text/html", "Etag"=>"12345678"}, self]
  end
  def each
    yield "hello world"
  end
end

run MyApp.new
```

请注意编写实际的程序应该用合适的算法(例如Digest提供的方法)来生成`Etag`。
在一个终端用rackup运行程序，并在另一个终端打开telnet:

```
telnet 127.0.0.1 9292
Trying 127.0.0.1...
Connected to sam.usersound.com.
Escape character is '^]'.
GET / HTTP/1.1
Host: localhost
```

我们的请求没有包括任何Etag的信息，所以程序返回的是200 OK。

```
HTTP/1.1 200 OK
Etag: 12345678
Connection: Keep-Alive
Content-Type: text/html
Date: Wed, 20 Jan 2010 16:07:06 GMT
Server: WEBrick/1.3.1 (Ruby/1.8.7/2009-06-12)
Content-Length: 11

hello world
```

响应包含了这个文档的Etag，Content-Length和Content-Type，以及实际的内容。
现在我们知道这个文档的Etag是12345678，可以用一个If-None-Match来指定Etag:

```
GET / HTTP/1.1
Host: localhost
If-None-Match: 12345678
```

这一次程序返回的是304 Not Modified，这表示状态码已经被正确设置。

```
HTTP/1.1 304 Not Modified
Etag: 12345678
Date: Wed, 20 Jan 2010 16:07:41 GMT
Server: WEBrick/1.3.1 (Ruby/1.8.7/2009-06-12)
```

没有Content-Length和Content-Type，以及实际的内容体。

你可以自己编写一个简单的程序测试Last-Modified头字段，注意你可以用httpdate方法把任何一个时间对象转换为合法的http日期格式。

```
require 'time'
Time.now.httpdate
```

这里进一步展示为什么我们不应该在call被调用时直接拼装字符串，而是延迟到each方法被调用的时候才去生成，因为在某些情况下完全有可能不需要生成内容。

5.3.3 Rack::ContentLength

除了某些特殊的情况，HTTP协议需要服务端的响应头中包括正确的ContentLength字段，它必须等于响应体的长度，这样客户端才能知道什么位置本次响应内容已经结束。

一个Rack程序可能使用了多个中间件，每个中间件都可能修改响应体的内容，这样就会要求每个响应体计算并设置正确的ContentLength字段。除了最后一次计算外，所有中间层次的ContentLength值可能都会被覆盖，这时只在必要的时候使用一次Rack::ContentLength就比较合适了(通常在最外面use，除了我们后面会看到的不应该设置ContentLength的某些中间件)。

```
def call(env)
  status, headers, body = @app.call(env)
  headers = HeaderHash.new(headers)

  if !STATUS_WITH_NO_ENTITY_BODY.include?(status) &&
     !headers['Content-Length'] &&
     !headers['Transfer-Encoding'] &&
     (body.respond_to?(:to_ary) || body.respond_to?(:to_str))

    body = [body] if body.respond_to?(:to_str) # rack 0.4 compat
    length = body.to_ary.inject(0) { |len, part| len + bytesize(part) }
    headers['Content-Length'] = length.to_s
  end

  [status, headers, body]
end
```

代码中bytesize是在Rack::Utils模块定义的，它主要为了处理Ruby 1.9和Ruby 1.8的差异。Ruby 1.9开始String.size是返回字符个数而不是字节数(例如对于UTF-8的编码，每个汉字占用一个以上的字节)，只有String.bytesize才返回真正的字节数–这正是HTTP协议要求的。

```
if ''.respond_to?(:bytesize)
  def bytesize(string)
    string.bytesize
  end
else
  def bytesize(string)
    string.size
  end
end
```

要让此中间件设置Content-Length值，必须同时符合四个条件:

- 响应体有内容
- 响应头尚未包含一个Content-Length值。已经包含就不再设置了
- 没有设置Transfer-Encoding头字段。包含Transfer-Encoding头字段的响应，长度是不固定的。
- 能够响应to_ary或者to_str方法
  - to_str主要针对Ruby 1.8编写的代码，可能直接在响应体内使用字符串。
  - to_ary表示可以转换一个数组，这个原因在于很多不固定的长度的消息（例如分块传输）不希望设置Content-Length，或者是无法计算出长度的响应体。

传输长度

为了完整性起见，我们在这里讨论一下消息的传输长度，你不必立刻理解所有的概念，后面会有详细的描述。

服务端和客户端根据消息的传输长度来决定什么时候本次请求/响应的内容已经全部接收完毕。

用协议的标准说法，一条消息的传输长度指的是消息体的长度，也就是在所有的传输编码被应用以后的长度。消息体的长度可以按照下面的规则来确定(按优先级排序):

1. 任何不应该包含消息体的响应消息–这样的消息包括1xx,204,304和任何对HEAD请求的响应，它们永远在所有头字段后面的第一个空行结束。Rack::Utils定义了这些状态码:

   ```
   # Responses with HTTP status codes that should not have an entity body
   STATUS_WITH_NO_ENTITY_BODY = Set.new((100..199).to_a << 204 << 304)
   ```

2. 如果有一个Transfer-Encoding头，而它的值不是`identity`，那么有两种方法可以确定传输长度:
   - 使用“chunked”传输编码来确定(我们已经在5.3.1(p. 46)中讨论)；或者
   - 关闭连接表示消息已经结束。
3. 如果有一个Content-Length头字段，那么它的数字值代表实体常数和传输长度。如果实体的长度和传输的长度不同，一定不能设置Content-Length头字段。(这样的例子包括Transfer-Encoding和Content-Encoding(5.3.5(p. 57))被设置的情况。)如果接收到的消息同时包括了Transfer-Encoding头字段和Content-Length头字段，那么Content-Length应该被忽略。
4. 如果一条消息使用媒体类型“multipart/byteranges”，并且没有用前面的方法指定传输长度，那么用媒体类型本身的分界规则来确定传输长度。
5. 最后一种手段是服务器关闭连接(这个方法只能由服务端使用，客户端关闭连接的话就无法接受响应了)

反过来讲，如果在发送消息的时候，我们可以知道它的消息长度，那么应该设置合适的Content-Length头字段。而如果无法预先知道长度，或者长度的计算非常困难时，我们可以用三种方法告诉对方:

- 采用chunked传输编码，参见5.3.1(p. 46)。HTTP/1.1规定所有HTTP应用程序必须能够处理分块传输。
- 关闭连接(只能由服务端使用)
- 使用multipart/byteranges，但它不是适合所有的消息。

要注意的是，在无法确认长度的情况下，一定**不能设置**Content-Length头字段。如果设置了不正确的Content-Length，就会产生下面这些后果:

- Content-Length的值大于实际的传输长度。这样做会导致接收方(如浏览器)等待接收不可能到达的消息，浏览器就会一直挂起。在持久连接的情况下，接收方会读取下一个响应(或请求)的部分内容，整个通讯过程被破坏。
- Content-Length的值小于实际的传输长度。在持久连接的情况下，本消息的一部分被当作下一个响应(或请求)读取，整个通讯过程被破坏。

5.3.4 Rack::ContentType

你可以在使用ContentType的时候附带一个需要设置的内容类型，例如:

```
use Rack::ContentType, "text/html"
```

如果应用程序和中间件都没有设置ContentType头字段，那么这里指定的参数被设置。

```
def call(env)
  status, headers, body = @app.call(env)
  headers = Utils::HeaderHash.new(headers)
  headers['Content-Type'] ||= @content_type
  [status, headers, body]
end
```

5.3.5 Rack::Deflater

编码

HTTP支持对传输的内容进行压缩，从而减少传输量，提高吞吐能力。如何以及是否进行压缩是客户端和服务端双方协商的结果。对客户端来说，请求头中的Accept-Encoding代表它能够接受的编码，而响应头中的Content-Encoding则表示服务端对响应的内容进行了什么编码。

当然，客户端的请求也可以用Content-Encoding来编码它的内容，这个我们留到后面去讨论。

RFC2616定义的编码包括:

gzip 由文件压缩程序“gzip”(GUN zip)生成的编码格式，具体描述可以参见RFC 1952 [25]. 它的格式是带32位CRC的 Lempel-Ziv 编码 (LZ77)。

compress UNIX文件压缩程序“compress”产生的压缩格式。它的格式是一个自适应的 Lempel-Ziv-Welch编码 (LZW)。

deflate RFC 1950 [31]定义的“zlib”格式和 RFC 1951 [29]描述的“deflate”压缩机制的一个组合。

identity 缺省编码，不做任何压缩。Accept- Encoding可以包含此编码，而Content-Encoding 头不应该使用它。

如果请求不包括Accept-Encoding字段，服务端可能假设客户端会接受任何内容编码，但这种情况下服务端应该使用identity编码。

客户端可以设置Accept-Encoding字段，告诉服务器它所能接受的编码类型。它可以包括多个编码，相互之间用逗号分隔。例如:

```
Accept-Encoding: compress, gzip
Accept-Encoding:
Accept-Encoding: *
Accept-Encoding: compress;q=0.5, gzip;q=1.0
Accept-Encoding: gzip;q=1.0, identity; q=0.5, *;q=0
```

某一个压缩编码的后面还可以包括一个分号加上一个q值:

```
compress;q=0.5
```

q是一个0–1之间的值，表示压缩的质量。我们这里并不关心具体的含义，但要说明的一点是，当$q = 0$的时候，就表示对应的编码是不可接受的。

服务端根据请求的Accept-Encoding字段值判断客户端可以接受的编码，规则如下:

1. 客户端可以接受列在其中的所有编码(q=0除外)。如:

   `Accept-Encoding: compress, gzip`

   表示接受compress和gzip两种编码。

2. “*”表示接受所有的编码，如:

   `Accept-Encoding: *`

3. 如果接受多个编码，那么最高q值的编码优先。如:

   `Accept-Encoding: compress;q=0.5, gzip;q=1.0`

   表示可以接受compress和gzip编码，但gzip编码优先。

4. `identity`编码总是可以接受。除非发生下面两种情况之一:

   - Accept-Encoding值包含一项“`identity;q=0`”明确排除它。
   - Accept-Encoding值包含一项“`*;q=0`”排除所有编码。就算是这种情况，如果另外还包含了一项q值非0的identity编码，那么identity编码还是可以接受的。下面就是这样一个例子:

     `Accept-Encoding: gzip;q=1.0, identity; q=0.5, *;q=0`

5. 如果Accept-Encoding字段的值为空，那么只接受`indetity`编码

服务器判断自己是否能够提供客户端能够接受的编码。一般情况下，服务端总是能够发送符合上面编码条件的响应，但是如果服务器无法产生客户端要求的编码，那么服务端就会发送一个错误的406 (Not Acceptable)响应。

当服务端找到某一种客户端可以接受，而自己又能够生成的编码方式以后，就会对内容进行响应的编码，并在设置响应头字段Content-Encoding的值。例如:

```
Content-Encoding: gzip
```

如果编码的类型是identity，则不应该在响应包含Content-Encoding头字段。

我们提到过，客户端也可以用`Content-Encoding`编码自己发送的请求消息体，如果服务器不能接受这种编码（即无法解码),那么服务器应该返回一个415 (Unsupported Media Type)响应。

另外一个HTTP头字段可能影响是否进行编码，这就是`Cache-Control`。如果在`Cache-Control`值中包含`no-transform`指令，则表示原始内容的产生者（这里是我们的应用程序或其他内部的中间件）不希望任何第三方改变消息体的内容，包括对它进行编码。

内容协商

我们对编码的选择是基于客户端的喜好以及服务端可用的编码格式决定的，这实际上是一种协商过程。

同一个实体(文档)可能会有多种表现形式可用。例如客户请求的某个文档可以有不同的语言、不同的编码。绝大多数的HTTP响应都包含可以被一个人类识别的内容，所以我们希望针对一个请求，服务器可以提供为用户提供“最佳”的内容。但是不同的用户有不同的偏好，不同的浏览器展示不同表现形式的能力也会不同，所以很难定义什么是“最佳”。因此，HTTP提供了几种方法用于在一个响应的多个可用表现形式之中选择一个最佳的表现形式–这个就叫做内容协商。

HTTP内容协商的基本方法有两种：服务器驱动和客户端驱动。但由于它们是正交的，因而可以组合使用。我们在这里关心的是服务端驱动的内容协商方式。

如果由服务器的算法来决定最佳的表现形式，我们就把它称作服务器驱动的协商。服务器的选择基于这个响应可用的表现形式（可以有很多维度，包括语言language、内容编码content-coding等等），请求消息中特定头字段的内容以及属于该请求客户端的信息（例如客户端的网络地址)。

即便采用服务端驱动的协商方式，客户端也不是无能为力的，它往往在请求头字段中描述它喜欢什么样的响应，这些头包括：Accept, Accept-Charset,Accept-Language, Accept-Encoding,User-Agent。除此之外，服务器还可能基于请求的任何方面做出选择。

在客户端和服务器之间，往往存在着很多缓存服务器(不管是正向还是反向的)。尽管是服务端驱动的协商，但是缓存也需要知道服务端选择的标准，因此服务端用一个Vary头字段来描述它是用什么样的参数来进行选择的。缓存需要知道这个选择标准的重要原因之一是协商过程改变了缓存的行为。

尽管我们前面只讨论了客户端的缓存，但客户端和服务器之间的缓存同样可以把一个响应缓存起来用来满足后续的请求。我们只讨论过Cache-Control的参数可以改变缓存的条件，现在Vary头字段也会影响缓存。Vary响应头字段的值包含了服务器用来进行选择的所有参数–即客户端发送到服务段的哪些请求头被用来选择“最佳”的表项形式–这些请求头叫做“选择”请求头。例如：

```
Vary:  Accept-Encoding
```

表示服务端基于请求头中的Accept-Encoding选择最佳的表现形式。

第一次请求时，缓存会保存得到的响应和对应Vary中所有“选择”请求头的值。之后，当缓存决定用原始的请求来满足后续的请求时（针对同一个URI)，它必须判断本次请求中存在的“选择”请求头是否和缓存的“选择”请求头完全匹配，如果不匹配，那么这个缓存就无法使用，它往往需要做出一个条件获取请求，得到304以后才能使用此缓存。

举一个简单的例子，假设第一次请求的头包括：

```
Accept-Encoding gzip,deflate
```

缓存得到的响应中包含：

```
Vary:  Accept-Encoding
```

如果在后续对同一个URI的请求中同样包括：

```
Accept-Encoding gzip,deflate
```

那么它们是匹配的。而如果后续的请求头中包含Accept-Encoding，但是不同的值:

```
Accept-Encoding: zlib
```

那么这个缓存就无法匹配。

之所以缓存能够使用原始请求的响应来满足后续的请求，是因为Vary字段的所有“选择”请求头名字被认为是决定选择算法的唯一因素。缓存可以假设如果所有的“选择”请求头的值不变，那么从服务器得到的“最佳”选择也不会变。(当然需要满足缓存的时间等其他条件)。

如果服务器不能满足上述假设，Vary的值会设成*，它表示服务器做“最佳”选择的参数不仅仅限于请求头(例如：客户端的网络地址或其他因素)。因此，缓存永远不能使用原始的响应，它必须从原始的服务器重新取得内容。

Rack::Deflater实现

现在理解程序就比较容易了，我们分段进行:

```
    def call(env)
      status, headers, body = @app.call(env)
      headers = Utils::HeaderHash.new(headers)

      # Skip compressing empty entity body responses and responses with
      # no-transform set.
      if Utils::STATUS_WITH_NO_ENTITY_BODY.include?(status) ||
          headers['Cache-Control'].to_s =~ /\bno-transform\b/
        return [status, headers, body]
      end
```

如果响应体无内容或者Cache-Control头包含no-transform指令，则原封不动返回。

```
     request = Request.new(env)

      encoding = Utils.select_best_encoding(%w(gzip deflate identity),
                                            request.accept_encoding)
```

首先用env创建一个新的request对象，这样我们可以直接存取request.accept_encoding了。

```
Utils.select_best_encoding(%w(gzip deflate identity),
                                        request.accept_encoding)
```

选择最佳的编码方式，选择的过程我们在前面已经详细讨论过了。第一个参数:

```
%w(gzip deflate identity)
```

表示服务器可用的所有编码。

```
# Set the Vary HTTP header.
vary = headers["Vary"].to_s.split(",").map { |v| v.strip }
unless vary.include?("*") || vary.include?("Accept-Encoding")
    headers["Vary"] = vary.push("Accept-Encoding").join(",")
end
```

这段代码用来处理Vary头，如果其中已经包含了*或者Accept-Encoding，则无需处理。不然，由于后面将对内容进行编码，因此需要在Vary的“选择”请求头中加入Accept-Encoding。

下面的代码对内容进行编码，是一个case语句。Deflater中间件有三种编码可用，分别是gzip,deflate和identity。如果选出的编码不是其中任何一种(select_best_encoding返回nil)，则按照前面讨论过的协议要求必须返回一个406响应。

indentity编码即不做任何编码，亦可原封不动返回。

```
case encoding
when "gzip"
    .........
when "deflate"
     .........
when "identity"
  [status, headers, body]
when nil
   message = "An acceptable encoding for the requested resource
                        #{request.fullpath} could not be found."
  [406, {"Content-Type" => "text/plain",
                      "Content-Length" => message.length.to_s}, [message]]
end
```

gzip编码

上述case语句中gzip分支代码如下:

```
when "gzip"
  headers['Content-Encoding'] = "gzip"
  headers.delete('Content-Length')
  mtime = headers.key?("Last-Modified") ?
    Time.httpdate(headers["Last-Modified"]) : Time.now
  [status, headers, GzipStream.new(body, mtime)]
```

程序设置Content-Encoding头为gzip，删除Content-Length头字段。

获取mtime为文档的最后修改时间。最后在响应中返回

```
[status, headers, GzipStream.new(body, mtime)]
```

响应体是一个新的GzipStream对象。

一个问题是为什么我们要删除Content-Length头信息？

由于我们使用了一定的压缩算法，如果要设置正确的Content-Length，那必须是压缩以后的内容长度。要获得压缩内容的长度，很可能需要把所有的原始内容读到内存，显然对于极大的文件这是无法接受的。

那么消息编码后的传输长度如何确定呢？回忆5.3.3(p. 55)的讨论，在无法确认当前消息的长度的时候,可以通过chunked传输编码或者通过服务端关闭连接来确定传输长度。按照RFC2616，此时应该删除Content-Length头字段。

另外一点，一旦删除了Content-Length头字段，应用程序本身、Web框架、Web服务器就可以使用Rack::Chunked(5.3.1(p. 46))中间件来利用chunked传输机制。如果还存在Content-Length头字段，Rack::Chunked将不起任何作用。

根据Rack对响应体的要求，GzipStream必须能够响应each，每次产生一个字符串：

```
class GzipStream
  def initialize(body, mtime)
    @body = body
    @mtime = mtime
  end

  def each(&block)
    @writer = block
    gzip  =::Zlib::GzipWriter.new(self)
    gzip.mtime = @mtime
    @body.each { |part| gzip.write(part) }
    @body.close if @body.respond_to?(:close)
    gzip.close
    @writer = nil
  end

  def write(data)
    @writer.call(data)
  end
end
```

Zlib::GzipWriter的构造方法取一个对象，这个对象是压缩数据的输出对象，一般来说，这个输出对象是一个IO对象（如File)。但是事实上，输出对象只要能够响应write方法就可以了。当你调用这个GzipWriter实例的write方法时，GzipWriter对象会用经过Gzip压缩算法压缩过的内容调用输出对象的write方法。

上述代码中，GzipStream的each方法首先创建一个Zlib::GzipWriter对象gzip，由于构造函数传入的参数是self，因此当调用gzip.write时，gzip会调用当前GzipStream实例的write方法。

each接着调用body的each方法，每次取得原始的内容，然后把内容写入gzip，gzip把数据压缩后调用当前GzipStream实例的write方法，而write则用这个压缩后的数据调用紧跟each的代码块。

deflate编码

case语句中上述分支代码如下:

```
when "deflate"
  headers['Content-Encoding'] = "deflate"
  headers.delete('Content-Length')
  [status, headers, DeflateStream.new(body)]
```

和gzip编码相关的代码没多大区别。

DeflateStream实现也没有多少复杂的地方，请自行参阅代码。

你可以用下面的代码进行测试，用telnet或者firebug观察返回的响应头:

```
use Rack::Chunked
use Rack::Deflater

run lambda {|env| [200, {'Content-Type'=>"text/html"},["abcde"*1000]]}
```

不同的Web服务器处理方式有所不同，Thin自动会在必要的是使用Rack::Chunked中间件，所以你不必自己use。而WEBrick不恰当地设置了Content-Length。

5.3.6 Rack::Etag

在5.3.2(p. 48)中，我们讨论了如何利用Etag实现HTTP缓存。

Rack::ConditionalGet中间件需要程序自己计算Etag，它所做的事情是比较请求和响应中的Etag，并设置合适的响应头和状态。这样做的好处是你可以在应用程序代码根据某种算法计算Etag，一旦Etag匹配，中间件ConditionalGet可以完全不去调用生成具体内容的each方法，从而节约了服务器的CPU资源，网络的传输也大大减少。

某些情况下，你可能无法在程序内部计算Etag–需要计算ETag的片段非常多，很多片段都有可能发生变化。典型的一个例子是一个Rails程序，某一个页面是否发生改变不但取决于模型的内容，而且取决于外部Layout的内容，而所有这些信息可能随着用户的不同而有所不同。这个时候你可能选择一种方案：每次服务端照样生成内容，但是在输出之前对整个响应体做计算一个Etag–如果内容未发生变化，就不需要把内容再传输到客户端。

```
def call(env)
  status, headers, body = @app.call(env)

  if !headers.has_key?('ETag')
```

```
      parts = []
      body.each { |part| parts << part.to_s }
      headers['ETag'] = %("#{Digest::MD5.hexdigest(parts.join(""))}")
      [status, headers, parts]
    else
      [status, headers, body]
    end
  end
```

Etag中间件只处理头字段中尚未包含“Etag”的情况。它读取整个Body的内容，并转换为一个字符串数组，最后用Digest::MD5.hexdigest计算出整个Etag。

显然Etag需要和Rack::ConditionalGet配合才能起作用:

```
use Rack::ConditionalGet
use Rack::ETag

run lambda{|env| [200, {'Content-Type'=>'text/html'},["any string here"]]}
```

用rackup运行这个文件。第一次响应为200，第二次则为304。

如果响应体的内容不是很大（例如Content-Type为text/html)的情况，把所有内容组合成一个字符串并计算其MD5的开销应该是可以承受的。但如果响应体是一个大文件，那么这种方式显然不可行。所以请谨慎使用，至少判断一下响应体的类型。

5.3.7 Rack::Head

RFC2616要求HEAD请求的响应体必须为空，这就是Rack::HEAD所做的事情:

```
  def call(env)
    status, headers, body = @app.call(env)

    if env["REQUEST_METHOD"] == "HEAD"
      [status, headers, []]
    else
      [status, headers, body]
    end
  end
```

5.3.8 Rack::MethodOverride

浏览器和Web服务器一般不直接支持PUT和DELETE方法，而只支持POST方法。而Rest风格的编程对于PUT、DELETE和POST有比较严格的区分。所以我们需要用POST方法来模拟PUT和DELETE方法。

一般可以有两种方法用POST来模拟:

1. 在POST提交的表单数据中嵌入一个隐含的字段来区分这到底是一个什么方法。

   ```
   <form action="......"  method="post">
       <input name="_method" type="hidden" value="put" />
       ....
   </form>
   ```

   表单中有一个隐含的"_method"字段，它的值是"put"，表示此请求其实是一个PUT而不是POST。

2. 在HTTP请求中加入一个扩展的X_HTTP_METHOD_OVERRIDE请求头，这个头在Rack环境中变为HTTP_ X_HTTP_METHOD_OVERRIDE。

下面是Rack::MethodOverride的具体实现，分别处理这两种情况。

```
HTTP_METHODS = %w(GET HEAD PUT POST DELETE OPTIONS)

METHOD_OVERRIDE_PARAM_KEY = "_method".freeze
HTTP_METHOD_OVERRIDE_HEADER = "HTTP_X_HTTP_METHOD_OVERRIDE".freeze

def call(env)
  if env["REQUEST_METHOD"] == "POST"
    req = Request.new(env)
    method = req.POST[METHOD_OVERRIDE_PARAM_KEY] ||
      env[HTTP_METHOD_OVERRIDE_HEADER]
    method = method.to_s.upcase
    if HTTP_METHODS.include?(method)
      env["rack.methodoverride.original_method"] = env["REQUEST_METHOD"]
      env["REQUEST_METHOD"] = method
    end
  end

  @app.call(env)
end
```

如果该请求是POST请求，而且在请求数据中包含"_method"值或者在环境中包含HTTP_ X_HTTP_METHOD_OVERRIDE值，而且它们的值是合法的HTTP方法，那么此中间件将把原始的POST保存起来，并设置新的REQUEST_METHOD值。

下面的程序测试Rack::MethodOverride是否起到作用:

```
use Rack::MethodOverride

map '/' do
```

```
  form = <<-HERE
  <form action="/user" method="post">
      <input name="_method" type="hidden" value="put" />
      <input name="name" type="text" value="" />
      <input type="submit"  value="Modify!">
  </form>
  HERE
  run lambda {|env| [200, {"Content-Type" => "text/html"}, [form]] }
end
map '/user' do
    run lambda {|env|
      req = Rack::Request.new env
      res = Rack::Response.new
      if(req.put?)
        res.write("you modify user name to #{req.params['name']}")
      else
        res.write("we only support put method to modify user,
                          yours is #{req.request_method}")
      end
      res.finish
    }
end
```

当用户请求`http://localhost:9292`，程序返回一个表单，并在其中加入了一个隐含字段“_method”，把它的值设为“put”。用户提交表单后，程序得到的req为一个PUT请求。

如果去掉对MethodOverride的使用，那么请求的方法依旧为POST，而不是我们期望的PUT。

5.4 程序开发中间件

所有的程序开发都遵循类似的模式，我们需要读写日志、评测性能、检查是否符合一定的规范等等。Rack提供了不少程序开发相关的中间件。

5.4.1 Rack::CommonLogger

任何一个Web程序必须记录日志信息，CommonLogger用Apache common log的格式把每一个请求的信息记录到一个`logger`中去。日志的具体格式可以参考http://httpd.apache.org/docs/1.3/logs.html#common。

这个logger必须是一个合法的错误流*Error Stream*，它必须符合:

- 能够响应 `puts`, `write`和`flush`方法

- 我们应该可以用一个参数调用`puts`，只要这个参数能够响应`to_s`
- 我们应该可以用一个参数调用`write`，只要这个参数是String
- 我们应该可以无参数地调用`flush`，从而保证日志信息确实被写入

标准输出是符合这个条件的:

```
use Rack::CommonLogger, $stderr
```

如果没有给定任何错误流，那么Rack::CommonLogger会从环境变量的`rack.errors`去获得对应的值。

5.4.2 Rack::Lint

Rack::Lint检查请求和响应是否符合Rack规格书，这正是我们完全理解Rack规格的好时机。如果你自己去实现Web服务器、Web框架、Rack中间件，甚至是实现Rack的替代品时，Rack::Lint都是一个很好的工具。

Rack的检查分两种，一种是静态检查，在它的call方法实现；另外一种是动态检查，在它的each方法实现。

如果检查没有通过则抛出Rack::Lint::LintError，为了代码编写方便，定义了一个新的assert方法

```
class LintError < RuntimeError; end
module Assertion
  def assert(message, &block)
    unless block.call
      raise LintError, message
    end
  end
end
include Assertion
```

assert方法取一个消息参数和一个代码块，如果代码块执行的结果为false(或nil)，那么抛出LintError错误。

call检查

```
def call(env=nil)
  dup._call(env)
end

def _call(env)
  ## It takes exactly one argument, the *environment*
  assert("No env given") { env }
  check_env env
```

```
  env['rack.input'] = InputWrapper.new(env['rack.input'])
  env['rack.errors'] = ErrorWrapper.new(env['rack.errors'])

  ## and returns an Array of exactly three values:
  status, headers, @body = @app.call(env)
  ## The *status*,
  check_status status
  ## the *headers*,
  check_headers headers
  ## and the *body*.
  check_content_type status, headers
  check_content_length status, headers, env
  [status, headers, self]
end
```

call主要检查下面的内容:

- 请求的环境(env)是否符合规格
- 响应的状态、头字段、内容类型以及内容长度

检查环境

```
def check_env(env)
  ## The environment must be an instance of Hash that includes
  ## CGI-like headers.  The application is free to modify the
  ## environment.
  assert("env #{env.inspect} is not a Hash, but #{env.class}") {
    env.kind_of? Hash
  }
```

环境必须是一个Hash。

```
  if session = env['rack.session']
    ##                         store(key, value)         (aliased as []=);
    assert("session #{session.inspect} must respond to store and []=") {
      session.respond_to?(:store) && session.respond_to?(:[]=)
    }

    ##                         fetch(key, default = nil) (aliased as []);
    assert("session #{session.inspect} must respond to fetch and []") {
      session.respond_to?(:fetch) && session.respond_to?(:[])
    }
```

```
    ##                          delete(key);
    assert("session #{session.inspect} must respond to delete") {
      session.respond_to?(:delete)
    }

    ##                          clear;
    assert("session #{session.inspect} must respond to clear") {
      session.respond_to?(:clear)
    }
  end
```

如果环境包含rack.session关键字，它应该保存请求的session，它必须是一个类似Hash的对象，也就是说，假设这个对象叫做session，那么它必须响应下列方法:

- store方法和别名[]=方法，即可以用session.store(key,value)或者session[key]=value保存关键字、值对。
- fetch方法和别名[]，即可以用session.fetch(key,default=nil)或者session[key]获取关键字对应的值。
- delete方法，即可以用session.delete(key)删除key对应的关键字、值对。
- clear，即session.clear可以清除所有的条目

显然，session不必一定是 Hash类型，这是Ruby编程常见的Duck Typing。

```
  ## <tt>rack.logger</tt>:: A common object interface for logging messages.
  ##                        The object must implement:
  if logger = env['rack.logger']
    ##                          info(message, &block)
    assert("logger #{logger.inspect} must respond to info") {
      logger.respond_to?(:info)
    }

    ##                          debug(message, &block)
    assert("logger #{logger.inspect} must respond to debug") {
      logger.respond_to?(:debug)
    }

    ##                          warn(message, &block)
    assert("logger #{logger.inspect} must respond to warn") {
      logger.respond_to?(:warn)
    }

    ##                          error(message, &block)
```

```
  assert("logger #{logger.inspect} must respond to error") {
    logger.respond_to?(:error)
  }

  ##                           fatal(message, &block)
  assert("logger #{logger.inspect} must respond to fatal") {
    logger.respond_to?(:fatal)
  }
end
```

如果环境中存在着rack.logger关键字，则Rack程序可以使用它进行日志工作。这个对象必须响应下面的方法:

- info(message, &block)
- debug(message, &block)
- warn(message, &block)
- error(message, &block)
- fatal(message, &block)

这是在很多编程语言和框架日志的一个事实标准。

```
%w[REQUEST_METHOD SERVER_NAME SERVER_PORT
   QUERY_STRING
   rack.version rack.input rack.errors
   rack.multithread rack.multiprocess rack.run_once].each { |header|
  assert("env missing required key #{header}") { env.include? header }
}
```

对于Ruby应用服务器,Rack要求它们提供的env至少包含上述关键字。
env包括的关键字可以分作二类:

- 来自HTTP请求，类CGI的头。全部大写，只有一个部分（中间没有用“.”)。它又包含两类:
  - “HTTP_”开头的关键字。这些值直接来自客户端提供的HTTP请求头字段，Web服务器在调用Rack之前必须在这些头字段之前加上“HTTP_”，例如如果请求头中包含Accept字段，那么必须在环境中包含“HTTP_ACCEPT”。有两个头例外：CONTENT_TYPE和CONTENT_LENGTH，它们之前不应该加上“HTTP_”。

```
## The environment must not contain the keys
## <tt>HTTP_CONTENT_TYPE</tt> or <tt>HTTP_CONTENT_LENGTH</tt>
## (use the versions without <tt>HTTP_</tt>).
```

```
%w[HTTP_CONTENT_TYPE HTTP_CONTENT_LENGTH].each { |header|
 assert("env contains #{header}, must use #{header[5,-1]}") {
    not env.include? header
  }
}
```

- 从用户请求的其他部分得到，它们没有“HTTP_”这样的前缀。REQUEST_METHOD、SERVER_NAME、SERVER_PORT和QUERY_STRING这几个是必须提供的。

• 不是从HTTP请求消息直接得到的，一般来说小写。至少包括两部分，中间用“.”分隔。它也包括两类:

  - Rack保留，前缀是“rack.”。其中rack.version、rack.input、rack.errors、rack.multithread 、rack.multiprocess和 rack.run_once这几个关键字是必须提供的。
  - Web服务器自己使用的，前缀必须不是“rack.”。例如Thin服务器可能使用thin.xxx这样的关键字。

对上述这些必要的关键字对应的值，还有一系列规范:

```
## * <tt>rack.version</tt> must be an array of Integers.
    assert("rack.version must be an Array, was #{env["rack.version"].class}") {
      env["rack.version"].kind_of? Array
    }
    ## * <tt>rack.url_scheme</tt> must either be +http+ or +https+.
    assert("rack.url_scheme unknown: #{env["rack.url_scheme"].inspect}") {
      %w[http https].include? env["rack.url_scheme"]
    }

    ## * There must be a valid input stream in <tt>rack.input</tt>.
    check_input env["rack.input"]
    ## * There must be a valid error stream in <tt>rack.errors</tt>.
    check_error env["rack.errors"]
```

rack.version的值必须是一个数组，如[1, 0], [1, 1]分别表示Rack1.0和Rack1.1；rack.url_scheme的值必须是`http`或者`https`；rack.input和rack.errors对应的值必须是合法的输入和错误流，具体的要求稍后讨论。

```
    ## * The <tt>REQUEST_METHOD</tt> must be a valid token.
    assert("REQUEST_METHOD unknown: #{env["REQUEST_METHOD"]}") {
      env["REQUEST_METHOD"] =~ /\A[0-9A-Za-z!\#$%&'*+.^_`|~-]+\z/
    }

    ## * The <tt>SCRIPT_NAME</tt>, if non-empty, must start with <tt>/</tt>
    assert("SCRIPT_NAME must start with /") {
```

```
    !env.include?("SCRIPT_NAME") ||
    env["SCRIPT_NAME"] == "" ||
    env["SCRIPT_NAME"] =~ /\A\//
  }
  ## * The <tt>PATH_INFO</tt>, if non-empty, must start with <tt>/</tt>
  assert("PATH_INFO must start with /") {
    !env.include?("PATH_INFO") ||
    env["PATH_INFO"] == "" ||
    env["PATH_INFO"] =~ /\A\//
  }
  ## * The <tt>CONTENT_LENGTH</tt>, if given, must consist of digits only.
  assert("Invalid CONTENT_LENGTH: #{env["CONTENT_LENGTH"]}") {
    !env.include?("CONTENT_LENGTH") || env["CONTENT_LENGTH"] =~ /\A\d+\z/
  }

  ## * One of <tt>SCRIPT_NAME</tt> or <tt>PATH_INFO</tt> must be
  ##   set.  <tt>PATH_INFO</tt> should be <tt>/</tt> if
  ##   <tt>SCRIPT_NAME</tt> is empty.
  assert("One of SCRIPT_NAME or PATH_INFO must be set
   (make PATH_INFO '/' if SCRIPT_NAME is empty)") {
    env["SCRIPT_NAME"] || env["PATH_INFO"]
  }
  ##   <tt>SCRIPT_NAME</tt> never should be <tt>/</tt>, but instead be empty.
  assert("SCRIPT_NAME cannot be '/', make it '' and PATH_INFO '/'") {
    env["SCRIPT_NAME"] != "/"
  }
```

正则表达式中，“\A”表示字符串的开始，“\z”表示字符串的结束。上面这些语句分别指出:

- REQUEST_METHOD是一个字符串，值不可以为空，匹配上述正则表达式。
- SCRIPT_NAME和PATH_INFO两者必须至少有一个。
    - 如果存在SCRIPT_NAME的话，它的值可以是空字符串，也可以是一个以斜杠/开始的字符串。
    - 对PATH_INFO的要求和SCRIPT_NAME完全一样。

  但是SCRIPT_NAME代表的是一个应用的名字，所以不可以是单独的一个斜杠，必要的话可以让SCRIPT_NAME为空字符串，让PATH_INFO为一个单独的斜杠。
- CONTENT_LENGTH不是必须的。但如果有的话，必须一个字符串，其中应该完全都是数字。

现在回过头来看看rack.input对应的输入流需要满足什么条件:

```
## === The Input Stream
##
## The input stream is an IO-like object which contains the raw HTTP
## POST data.
def check_input(input)
  ## When applicable, its external encoding must be "ASCII-8BIT" and it
  ## must be opened in binary mode, for Ruby 1.9 compatibility.
assert("rack.input #{input} does not have ASCII-8BIT as its external encoding") {
    input.external_encoding.name == "ASCII-8BIT"
  } if input.respond_to?(:external_encoding)
  assert("rack.input #{input} is not opened in binary mode") {
    input.binmode?
  } if input.respond_to?(:binmode?)

  ## The input stream must respond to +gets+, +each+, +read+ and +rewind+.
  [:gets, :each, :read, :rewind].each { |method|
    assert("rack.input #{input} does not respond to ##{method}") {
      input.respond_to? method
    }
  }
end
```

输入流会包含原始的HTTP POST数据。如果合适的话，它应该使用ASCII-8BIT外部编码(*external_encoding*、用二进制模式打开。外部编码是Ruby 1.9出现的一个概念，它表示保存在文件中的文本编码。(与之对应的内部编码则是用于在Ruby中表示文本的编码)。二进制模式可以保证读到原始的数据。

除了编码的要求之外，输入流必须能够响应:gets, each, read, rewind这4个方法。

rack.errors对应的错误流要求稍低些，能够响应puts、write和flush方法即可。

```
## === The Error Stream
def check_error(error)
  ## The error stream must respond to +puts+, +write+ and +flush+.
  [:puts, :write, :flush].each { |method|
    assert("rack.error #{error} does not respond to ##{method}") {
      error.respond_to? method
    }
  }
end
```

检查状态码

```
## === The Status
def check_status(status)
  ## This is an HTTP status. When parsed as integer (+to_i+), it must be
  ## greater than or equal to 100.
  assert("Status must be >=100 seen as integer") { status.to_i >= 100 }
end
```

响应消息中返回的状态码必须能够用to_i转换为一个整数，这个整数必须大于100

检查响应头

响应头必须能够响应each方法，并且每次产生一个关键字和一个对应的值。Hash符合这个条件，我们前面谈到的Rack::Utils::HeadersHash被很多Rack中间件用于响应头。

```
## === The Headers
def check_headers(header)
  ## The header must respond to +each+, and yield values of key and value.
  assert("headers object should respond to #each,
              but doesn't (got #{header.class} as headers)") {
     header.respond_to? :each
  }
  header.each { |key, value|
    ## The header keys must be Strings.
    assert("header key must be a string, was #{key.class}") {
      key.kind_of? String
    }
    ## The header must not contain a +Status+ key,
    assert("header must not contain Status") { key.downcase != "status" }
    ## contain keys with <tt>:</tt> or newlines in their name,
    assert("header names must not contain : or \\n") { key !~ /[:\n]/ }
    ## contain keys names that end in <tt>-</tt> or <tt>_</tt>,
    assert("header names must not end in - or _") { key !~ /[-_]\z/ }
    ## but only contain keys that consist of
    ## letters, digits, <tt>_</tt> or <tt>-</tt> and start with a letter.
    assert("invalid header name: #{key}")
         { key =~ /\A[a-zA-Z][a-zA-Z0-9_-]*\z/ }

    ## The values of the header must be Strings,
    assert("a header value must be a String, but the value of " +
      "'#{key}' is a #{value.class}") { value.kind_of? String }
    ## consisting of lines (for multiple header values, e.g. multiple
    ## <tt>Set-Cookie</tt> values) seperated by "\n".
    value.split("\n").each { |item|
      ## The lines must not contain characters below 037.
      assert("invalid header value #{key}: #{item.inspect}") {
        item !~ /[\000-\037]/
```

```
        }
      }
    }
  end
```

对于每一个关键字/值对，即每一个响应头字段和它们的值，分别要求如下:

- 对关键字key而言
  - key必须是字符串，而不是Symbol
  - key不能是“Status”这个字符串
  - key不能包括“:”和“\n”这两个字符
  - key必须以字母开头，后跟多个字母、数字、“-”或者“_”，但不能以“-”或者“_”这两个字符结尾。
- 对它们的值(value)来说
  - value必须是字符串
  - value的值不可以包含值为037以下的字符，即控制字符

检查内容类型

```
## === The Content-Type
def check_content_type(status, headers)
  headers.each { |key, value|
    ## There must be a <tt>Content-Type</tt>, except when the
    ## +Status+ is 1xx, 204 or 304, in which case there must be none
    ## given.
    if key.downcase == "content-type"
     assert("Content-Type header found in #{status} response, not allowed") {
        not Rack::Utils::STATUS_WITH_NO_ENTITY_BODY.include? status.to_i
      }
      return
    end
  }
  assert("No Content-Type header found") {
    Rack::Utils::STATUS_WITH_NO_ENTITY_BODY.include? status.to_i
  }
end
```

headers.each检查每一个头，如果发现’Content-Type’，可能发生两种状况:

- 如果assert发现，状态码不是1xx、204或者304这个条件不成立–即它们确实是1xx、204或者304中间的某一个状态，则抛出异常

- 如果状态码不是1xx、204或者304这个条件成立，那么执行assert后面的代码，即return，后续代码不会执行

如果没有发现Content-Type,则程序会做each之后的那个assert，这次要求状态必须是1xx、204或者304。

综合两者就是说，状态码是1xx、204或者304的响应必须不能有Content-Type头，其他所有的状态必须有Content-Type头。

检查内容长度

```
## === The Content-Length
def check_content_length(status, headers, env)
  headers.each { |key, value|
    if key.downcase == 'content-length'
         ## There must not be a <tt>Content-Length</tt> header when the
         ## +Status+ is 1xx, 204 or 304.
         assert("Content-Length header found in #{status} response, not allowed") {
            not Rack::Utils::STATUS_WITH_NO_ENTITY_BODY.include? status.to_i
         }
         ..............
         ..............
         ..............
         return
    end
  }
end
```

代码开头部分和Content-Type 一样：对那么状态为1xx、204或者304的响应而言，必须不能设置Content-Length。

如果不是这些状态，而且确实存在着Content-Length，就要检查Content-Length的值是否和实际的内容长度一致。上面的代码中被省略的代码如下：

```
bytes = 0
string_body = true

if @body.respond_to?(:to_ary)
  @body.each { |part|
    unless part.kind_of?(String)
      string_body = false
      break
    end

    bytes += Rack::Utils.bytesize(part)
  }
```

```
    if env["REQUEST_METHOD"] == "HEAD"
  assert("Response body was given for HEAD request, but should be empty") {
        bytes == 0
      }
    else
      if string_body
    assert("Content-Length header was #{value}, but should be #{bytes}") {
          value == bytes.to_s
        }
      end
    end
  end
```

代码只判断响应体是数组或者能够转换成一个数组的情况。
首先计算是这个body的长度，以及body完全为字符串。

```
    @body.each { |part|
      unless part.kind_of?(String)
        string_body = false
        break
      end

      bytes += Rack::Utils.bytesize(part)
    }
```

接下去检查当请求方法为GET时，响应体的长度必须为0。如果不是GET而且整个响应体确实是字符串时，那么Content-Length响应头的值(value)必须等于响应体实际的长度。

each 检查

call方法是一种静态的检查，each方法则对响应体的进行动态检查。

```
## === The Body
def each
  @closed = false
  ## The Body must respond to +each+
  @body.each { |part|
    ## and must only yield String values.
    assert("Body yielded non-string value #{part.inspect}") {
      part.kind_of? String
    }
    yield part
  }
```

首先检查响应体必须能够响应each方法，并且每次必须产生一个字符串(part)。

```
if @body.respond_to?(:to_path)
  assert("The file identified by body.to_path does not exist") {
    ::File.exist? @body.to_path
  }
end
```

而如果响应体能够响应to_path方法，那么它返回的值应该是一个路径名，这个路径名所代表的文件应该存在。

5.4.3 Rack::Reloader

某些时候，当修改了应用程序以后，我们希望框架能够重新载入修改后的代码。例如，在开发的过程中，我们不希望每次改动代码都要重启整个Web服务器。正因为如此，Rail提供了development和production等不同的环境。

例如我们有一个简单的程序，包括两个文件。一个是test-reloader.ru:

```
require 'simple'
run Simple.new
```

另外一个simple.rb:

```
class Simple
  def call(env)
  [200, {'Content-Type'=>'text/html'},["first"]]
  end
end
```

现在用rackup test-reloader.ru启动程序。在浏览器输入http://localhost:9292，将得到：first。然后我们修改代码，把simple.rb中的first改为second。不管你如何刷新，浏览器得到的永远是first，除非你退出并重启rackup。

使用Rack::Reloader很简单，只需要在test-reloader.rb加入一行：

```
use Rack::Reloader
require 'simple'
run Simple.new
```

重新启动rackup，但这一次如果以按上述过程把first改为second，浏览器输出的结果就会马上发生改变(可能需要刷新几次浏览器，因为重新加载有一定的时间–下面我们可以看到如何配置这个时间)。

由于Rack::Reloader非常高效，你甚至可以生产环境下用它来重新载入源代码。

Rack::Reloader实现

```
def initialize(app, cooldown = 10, backend = Stat)
  @app = app
  @cooldown = cooldown
  @last = (Time.now - cooldown)
  @cache = {}
  @mtimes = {}

  extend backend
end
```

在初始化Rack::Reloader空间件的时候，你可以指定间隔的时间(cooldown)，以及一个模块backend。这个模块提供一个rotation方法，用来计算所有已加载的文件和相关信息。

```
def call(env)
  if @cooldown and Time.now > @last + @cooldown
    if Thread.list.size > 1
      Thread.exclusive{ reload! }
    else
      reload!
    end

    @last = Time.now
  end

  @app.call(env)
end
```

Rack在每一个请求到达的时候进行检查，只有当超过设定的间隔时间，它才可能去做一个重载。它同时判断当前是否有多个线程存在(Thread.list.size > 1)：如果只有一个线程，那么直接调用reload!，不然的话，则在一个临界区内执行reload!。

Thread.exclusive在临界区内执行代码，这个临界区是针对整个Ruby进程的，在临界区内，所有已经存在的线程将不被调度。虽然不完全正确，但你大致可以认为这让Ruby的所有其他Green Thread暂停运行。

```
def reload!(stderr = $stderr)
  rotation do |file, mtime|
    previous_mtime = @mtimes[file] ||= mtime
    safe_load(file, mtime, stderr) if mtime > previous_mtime
  end
end
```

rotation是由Stat模块提供的，它提供所有已加载的文件和对应的最后修改时间。如果此文件上次最后加载的时间是在最后修改时间之前，即加载后又被修改了，那么程序调用safe_load执行真正的加载工作。

```
# A safe Kernel::load, issuing the hooks depending on the results
def safe_load(file, mtime, stderr = $stderr)
  load(file)
  stderr.puts "#{self.class}: reloaded `#{file}'"
  file
rescue LoadError, SyntaxError => ex
  stderr.puts ex
ensure
  @mtimes[file] = mtime
end
```

safe_load确保加载过程出现的加载错误和语法错误不会抛出异常，另外它确保文件的最后加载时间设置为当前的最后修改时间。

现在我们来看看Stat模块的rotation是如何实现的。

```
def rotation
  files = [$0, *$LOADED_FEATURES].uniq
  paths = ['./', *$LOAD_PATH].uniq

  files.map{|file|
    next if file =~ /\.(so|bundle)$/ # cannot reload compiled files

    found, stat = figure_path(file, paths)
    next unless found && stat && mtime = stat.mtime

    @cache[file] = found

    yield(found, mtime)
  }.compact
end
```

rotation首先获得当前的程序名字(`$0`)和所有已经被加载的文件(`$LOADED_FEATURES`)，保存到files变量。接着把paths设置为所有的加载路径(`$LOAD_PATH`)。

对每一个文件，程序首先判断它是否为C库-它们是无法重新加载的(so是linux、而bundle是mac os x下面的库文件后缀)。接着ratation调用figure_path方法去寻找文件，如果找到文件，则用该文件和它的最后修改时间去调用rotation后面的代码块-我们已经在reload!程序中看到了。

figure_path方法分两种情况:

- 如果给定文件名file是绝对路径，那么直接调用safe_stat(file)

- 否则，尝试所有的Ruby加载路径，把它们和文件名File.join起来，再去调用safe_stat(file)，直至真正的文件找到为止

具体的实现请参见Rack::Reloader的源代码。

5.4.4 Rack::Runtime

在日志、性能评测、分析的过程中，我们希望知道一个请求的处理时间，Runtime中间件计算这个时间，并把它放在`X-Runtime`响应头中。代码解释了一切:

```
class Runtime
  def initialize(app, name = nil)
    @app = app
    @header_name = "X-Runtime"
    @header_name << "-#{name}" if name
  end

  def call(env)
    start_time = Time.now
    status, headers, body = @app.call(env)
    request_time = Time.now - start_time

    if !headers.has_key?(@header_name)
      headers[@header_name] = "%0.6f" % request_time
    end

    [status, headers, body]
  end
end
```

有一点需要注意，call并不是处理请求唯一的地方，很多展现逻辑往往是在body的each中实现的。

5.4.5 Rack::Sendfile

请注意：本节内容仅供理解SendFile机制所用，Rack::Sendfile的实现并非如下所述。本节内容也是不完整的，还需要补充lighttpd和apache。本节可能在后续版本中删除。

Web应用程序经常需要处理大文件，包括图片、PDF、Word、视音频文件等供客户端下载和展示。如果文件和应用逻辑没有任何关系，那么可以完全由代理服务器（如ngnix、lighttpd、apache等）处理，无需Ruby应用服务器的任何处理。

但某些时候，你需要根据用户请求的URL搜索具体的文件位置，或者你需要对文件的存取进行控制。典型的例子是某个文件只能被某些用户存取。这个时候请求必须

经过代理服务器到达Ruby应用服务器，经过Web程序的处理才能把实际的文件传输给客户端。

问题是，现在Web应用服务器需要负责读取文件，把文件的内容传输给代理服务器，代理服务器再把文件的内容写到客户端。这样做显然会造成CPU、内存和内部网络资源的大量浪费。

为了解决这个问题，绝大多数的代理服务器支持一种X-Sendfile机制，Ruby程序只需要设置相应的响应头，告诉代理服务器文件的具体位置。当代理服务器在响应头中发现X-Sendfile被设置，那么它就会根据文件的路径直接读取文件内容，并发送给客户端，由于它可以直接对客户端连接读写，就可以大大提高文件传输和性能、降低CPU和内存消耗。

不同的代理服务有不同的X-Sendfile实现，我们来看看几种主流的代理服务器是如何要求的:

nginx Nginx把X-Sendfile叫做X-Accel-Redirect.

首先，你必须设置

```
sendfile on;
```

现在假设实际文件系统的/files/images下有一个abc.jpg文件。

第一个要考虑的问题是这个/files/images目录不能被用户直接存取，不然的话，按照nginx的一般设置，静态文件将由nginx直接处理，我们的ruby应用程序无法控制文件的传输过程。因此我们要把/images目录设置为内部目录，nginx的internal指令确保这一点。下面是对应的配置项:

```
location /images/ {
  internal;
  root   /files;
}
```

现在，当用户请求`http://www.somdomain.com/files/images/abc.jpg`文件的时候，按照我们一般对Ruby程序的配置，它就会把该请求重定向到Ruby应用服务器。

Ruby程序接受到该请求，进行处理后，应该在响应头中包括

```
X-Accel-Redirect: /images/abc.jpg
```

这样nginx就会连接上述设置中的root(即/files)和location(即/images)得到/files/images，然后读取文件系统实际的/files/images/abc.jpg文件，并把它发送给客户端。

更常见的做法是定义一个完整的别名，这样用户URL中的目录和实际存储文件的目录结构可以完全不同。例如，我们希望用户请求`http://www.somdomain.com/images/abc.jpg`就可以直接存取文件。如果文件实际保存在文件系统的/files/images下，那么你可以设置如下:

```
location /images/ {
  internal;
  alias   /files/images/;  #注意结尾的斜杠
}
```

注意，alias最后的斜杠是必不可少的。

同样，Ruby程序只需要设置响应头即可:

```
X-Accel-Redirect: /images/abc.jpg
```

Nginx知道/images对应的路径就是实际文件系统/files/images/，它会到那个目录下去读取合适的文件。

5.5 应用配置和组合中间件

5.5.1 Rack::Cascade

Rack::Cascade中间件可以挂载多个应用程序，请求到来时，它会尝试所有这些应用程序，直到某一个应用程序返回的代码不是404。

```
apps = [lambda {|env| [404, {}, ["File doesn't exists"]]},
        lambda {|env| [200, {}, ["I'm ok"]]}]
use Rack::ContentLength
use Rack::ContentType
run Rack::Cascade.new(apps)
```

这样做的意义何在？考虑我们需要在一个Rails应用程序中嵌入一个sinatra程序，考虑我们想单独处理文件上传、实现Cache处理而不希望加载整个Rails程序，但是缓存不存在的时候依旧能够调用对应的Rails逻辑....

可能性无限，我们将在后面讨论一些具体的例子。

5.5.2 Rack::Lock

某些web框架或程序可以在同一Ruby进程内多线程并发处理多个请求，如果Web服务器也支持多线程，那么env的`rack.multithread`将被设置为true。

某些框架不能处理多线程的情况，例如一般情况下Rails框架只能单线程运行。Rack::Lock中间件会对整个请求过程做一个互斥锁定:

```
module Rack
  class Lock
    FLAG = 'rack.multithread'.freeze

    def initialize(app, lock = Mutex.new)
```

```
    @app, @lock = app, lock
  end

  def call(env)
    old, env[FLAG] = env[FLAG], false
    @lock.synchronize { @app.call(env) }
  ensure
    env[FLAG] = old
  end
 end
end
```

开始处理请求之前，Rack::Lock把rack.multithread值设置为false，然后在锁的同步块内处理整个请求，最后恢复rack.multithread值。这可以保证整个Ruby进程同一时刻只能处理一个请求。

5.6 会话管理

我们这里所说的会话是指服务端能够跟踪用户和它之间的多次交互。要做到这一点，我们首先要能够确定用户的身份。

由于HTTP事务是无状态的，一个请求/响应结束以后，它和下一次请求/响应就没有关系了。为了确定某些请求来自某一个用户，HTTP提供了某些技术来标识用户。

HTTP诞生之初并没有太多考虑如何识别用户的问题，所以人们采用了各种各样的技术来实现，包括:

- 在HTTP头中附带用户身份的信息
- 用IP地址来识别客户
- 用户登录，使用验证来识别用户
- 在URL里面嵌入用户的身份
- Cookie，可以用来有效地维护持久的身份

我们在这里关心的重点是如何用Cookie来维护用户的身份。

5.6.1 HTTP Cookies

当用户第一次访问Web应用时，服务端并不知道这个用户的任何信息。为了确认该用户的后续访问，服务端设置一个唯一的cookie来标识该用户，然后在响应头中设置Set-Cookie字段值为此cookie。浏览器接收到此响应以后，把Cookie保存到

自己的cookie数据库。下一次，当用户访问同一个网站的时候，浏览器选取对应的cookie值，用它设置请求头中的Cookie字段。

Cookie中可以包含多个name=value这样的关键字值对，例如:

```
id = "1234567"; name="jack"; phone="65452334"
```

客户端保存cookies

不同的客户端用不同的方式保存cookies。但是每个cookie一般会有这么几项:

- domain: Cookie的域名。
- path: 和这个cookie相关域名上的开始路径。
- secure: 是否安全，如果被设置，那么只有在https的时候才会把这个cookie发送到服务端。
- expiration: 过期时间
- name: cookie的名字
- value: cookie的值

我们编写一个简单的程序来设置cookie:

```
run lambda {|env|
  [200, {'Content-Type'=>'text/html',
         'Set-Cookie'=>"id = 1234567\nname=jack\nphone=65452334"},
         ['hello world!']]}
```

用rackup运行此程序。

把你的/etc/hosts文件加上一行，暂且让它把example.com指向127.0.0.1:

```
127.0.0.1 www.example.com
```

打开浏览器，清除你的cookie，输入`http://www.example.com:9292`，再观察一下cookie。

firefox下面的显示如图5.6.1(p. 88)。图的上半部分是按照域名(domain)组织的一个cookie列表，而下半部分列出了该cookie对应的名称(name)、内容(value)、主机(host)、路径(path)、发送条件和过期时间(expiration)。

我们的程序中用Set-Cookie设置了三个cookie，它们之间用“\n”分隔。

我们没有为这些cookie设置过期时间，在图5.6.1(p. 88)下半部分，你可以看到过期时间是会话结束。Cookie可以分为两类，会话cookie(*session cookie*)和持久cookie(*persistent cookie*)。这里的会话是指浏览器的一次打开和关闭，也就是说如果你关闭浏览器，重新打开，那么原先的会话cookie将不存在了。持久cookie生存的时间更长，即使关闭、重新打开浏览器也不会被删除。显然，我们需要用持久cookie来维持应用程序的会话。

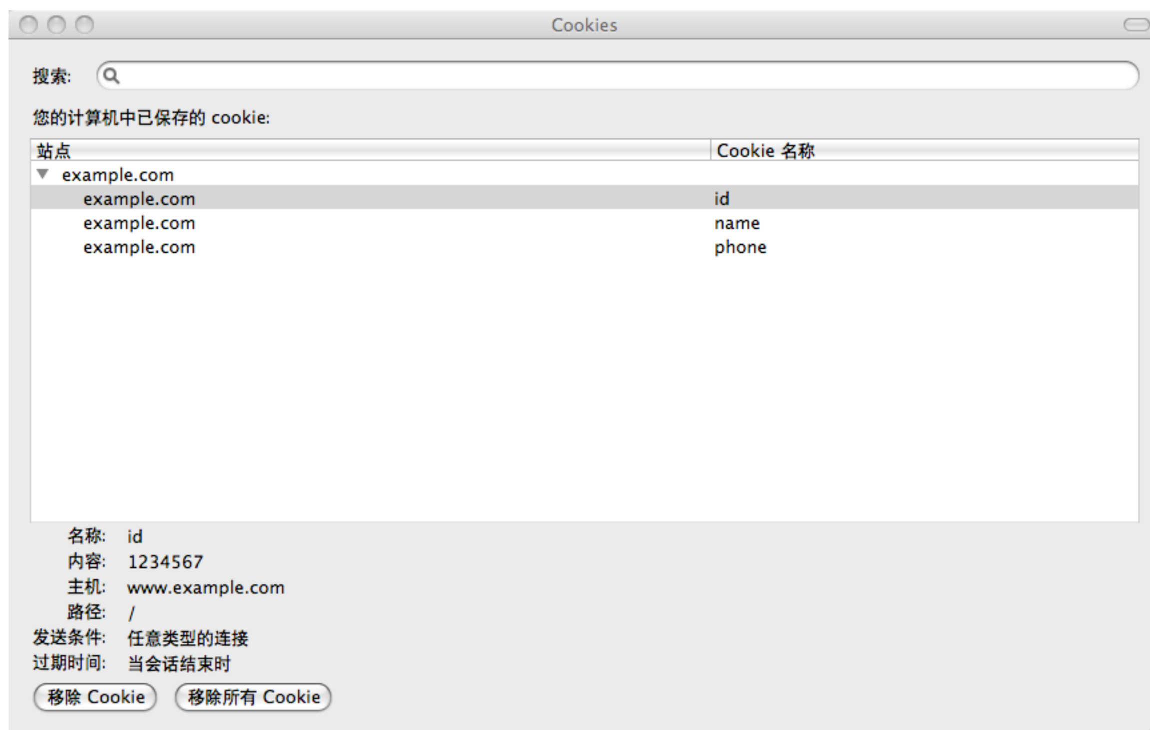

Figure 5.1: Firefox cookie

会话cookie和持久cookie没有本质的区别，唯一的不同是它们什么时候过期。如果一个cookie设置了一个Discard参数，或者即没有设置Expire也没有设置Max-Age参数，那么它就是一个会话session。不然，就是一个持久cookie。

设置Cookie属性

有两个版本的cookie规格书，分别是Version 0和Version1(RFC 2965)。常用的版本是Version 0，它来自Netscape，因为Netscape是第一个引入cookie机制的客户端。我们重点讨论Version 0。

服务端可以用以如下格式在响应头中设置cookie属性:

```
Set-Cookie: name=value [; expires=date] [; path=path] [; domain=domain] [; secure]
```

我们一个一个来看。

expires expires不是必须的。设置与否决定了这是一个会话cookie还是一个持久cookie。而如果有值的话，它决定了持久cookie过期的时间。时区必须为GMT:

```
Weekday, DD-Mon-YY HH::MM::SS GMT
```

年月日之间的分隔符必须为-。下面是一个设置了expires的例子:

```
Set-Cookie: foo=bar; expires=Wednesday, 09-Nov-99 23:12:40 GMT
```

Ruby中，你可以如下来生成合乎规范的时间:

```
Time.now.gmtime.strftime("%a, %d-%b-%Y %H:%M:%S GMT")
```

domain 域名，它也是可选的。浏览器根据它来判断是否要把cookie发送给某一域名(domain)的主机。如果domain是`example.com`，那么它可以匹配`www.example.com`, `blog.example.com`等等主机，但不会匹配`www.abc.com`。

如果没有设置domain，那么它缺省等于生成Set-Cookie响应的那个主机。注意一个主机不可以设置其他域名的domain值。例如，假设生成响应的主机名为`wiki.example.com`，它不能在响应的Set-Cookie中设置domain为`abc.com`。下面是一个设置了domain的例子:

```
Set-Cookie: foo=bar; domain="example.com"
```

为了深入了解domain设置的规则，我们可以做一个实验，在/etc/hosts中加入下面一行:

```
127.0.0.1 www.example.com example.com test.example.com
```

这样，三个主机名都指向本机。

现在编写下面的程序:

```
run lambda {|env|
  [200, {'Content-Type'=>'text/html',
         'Set-Cookie'=>"id = 1234567;domain=example.com"},
         ['hello world!']]}
```

不管我们用www.example.com、 example.com还是test.example.com去访问这个程序，你都可以在客户端看到cookie 的域名为example.com。这表示任何一个同一域名(doamin)内部的主机都可以把cookie的domain设置为自己的域名名字。(请在每一次试验前把cookie都清空。)

现在修改上面的程序，设置domain为test.example.com:

```
run lambda {|env|
  [200, {'Content-Type'=>'text/html',
         'Set-Cookie'=>"id = 1234567;domain=test.example.com"},
         ['hello world!']]}
```

现在用example.com和www.example.com去访问该应用程序，你将无法在客户端看到cookie。只有用test.example.com去访问时，才可以得到domain被设为test.example.com的cookie。

path 路径，它也是可选的。Path属性可以让你的cookie和web网站的一部分关联。如果把path设置为“/”，那么它可以匹配该域名内的所有文档。其他的路径值则表示一个前缀，也就是说，如果你设置了一个path为`/foo`，那么`/foobar`, `/foo/sample.html`等等都能匹配。

如果你没有设置任何Path值，那么缺省的就是生成Set-Cookie值的URL。

我们可以测试这一点，运行前面的应用程序，首先输入URL为`http://www.example.com`，你可以在浏览器中看到一个cookie，它的path为“/”。接着输入`http://www.example.com/foo`，你的客户端cookie对应的Path还是“/”。如果你输入`http://www.example.com/foo/bar`，那么对应的path则为“/foo”。

secure 也是可选的。如果设置了secue，例如:

```
Set-Cookie: order_id=519; secure
```

那么只有你用https访问该网站的时候，此cookie才会被客户端发送到服务端。

客户端发送Cookie

客户端通常会保存成千上万的cookie，它不可能把所有的cookie发送给所有的网站。它根据服务器设置的主机、路径、安全选项和当前访问的URL进行对比，然后把符合条件的cookie发送到对应服务端。

客户端的请求会包括一个头字段Cookie，形如:

```
Cookie: name1=value1 [;name2 =value2]
```

5.6.2 Rack::Session::Cookie

Rack::Session::Cookie提供了一个简单的基于cookie的会话管理。在这里，会话是一个Ruby的Hash对象，其中的数据采用base64编码保存。在使用这个中间件的时候，你可以指定它在env环境的关键字，缺省为`rack.session`。另外，你还可以指定一个保密码即:secrete。

我们先来看一个实际的例子:

```
use Rack::Session::Cookie, :key => 'rack.session',
                            :domain => 'example.com',
                            :path => '/',
                            :expire_after => 2592000,
                            :secret => 'any_secret_key'

run lambda {|env|
  user = env['rack.session'][:user]
  env['rack.session'][:user]  ||= 'test_user'
  [200, {"Content-Type" => "text/html"}, [user || "no current user"]]
}
```

把你的/etc/hosts文件加上一行，暂且让它把example.com指向127.0.0.1:

```
127.0.0.1 www.example.com
```

打开浏览器，清除你的cookie，输入`http://www.example.com:9292`，你会得到`no current user`。

程序已经正常运行，这表明env[’rack.session’]已经存在，并且是一个Hash。

刷新浏览器，你应该得到“test_user”，这表明我们设置了env['rack.session']中对于对应关键字:user的值。

这个时候，如果查看本地浏览器的cookie，你可以看到出现了一项“.example.com”的cookie。

如果你再次清除浏览器的cookie，那么还是会得到`no current user`，因为服务端无法从cookie得到会话信息。

我们来看一下具体的实现:

```
def initialize(app, options={})
  @app = app
  @key = options[:key] || "rack.session"
  @secret = options[:secret]
  @default_options = {:domain => nil,
    :path => "/",
    :expire_after => nil}.merge(options)
end
```

使用Rack::Session::Cookie的时候，允许你设置下面这些选项:

:key key指的是session在cookie中的name(缺省为rack.session)。也就是说，如果key是rack.session，那么最后Set-Cookie头字段如下:

```
Set-Cookie: rack.session=......
    [; expires=date] [; path=path] [; domain=domain] [; secure]
```

:secret 允许你设置一个保密码:secret，用来对你的cookie数据进行加密

:domain 域名

:path 路径

:expire_after cookie的有效期

这些选项的具体含义我们在前面5.6.1(p. 86)中已经详细讨论过了。

当然call是整个中间件的含义所在。

```
def call(env)
  load_session(env)
  status, headers, body = @app.call(env)
  commit_session(env, status, headers, body)
end
```

Rack::Session::Cookie所做的工作可以分为三个阶段:

1. 从请求的cookie中读出session数据，并设置为env[`rack.session`]对应值。
2. 请求处理过程–也就是我们的Rack应用程序可能改变session的内容。

3. 请求处理完以后把session数据(即env[rack.session])写入cookie，并设置响应的Set-Cookie响应头。

我们倒着来，首先看看session是如何被转换为cookie数据的:

```
1      def commit_session(env, status, headers, body)
2        session_data = Marshal.dump(env["rack.session"])
3        session_data = [session_data].pack("m*")
4
5        if @secret
6          session_data = "#{session_data}--#{generate_hmac(session_data)}"
7        end
8
9        if session_data.size > (4096 - @key.size)
10          env["rack.errors"].puts("Warning! Rack::Session::Cookie
                                 data size exceeds 4K. Content dropped.")
11        else
12          options = env["rack.session.options"]
13          cookie = Hash.new
14          cookie[:value] = session_data
15          cookie[:expires] = Time.now + options[:expire_after] unless
                                  options[:expire_after].nil?
16          Utils.set_cookie_header!(headers, @key, cookie.merge(options))
17        end
18
19        [status, headers, body]
20      end
```

当程序进行到这一步的时候，env['rack.session']里面已经包含了会话对象，它是一个Hash。

第2行把整个session dump到一个字符串session_data，第2行pack("m*")则对所有的数据进行Base64编码。

第5-8行，如果你设置了一个保密码，那么generate_hmac方法会根据你的加密码和session数据进行一个ssl的哈希加密，并把加密得到的数据和原先的数据用"–"连接起来。这样做的目的是为了在读取session的时候能够根据原始数据和加密数据进行验证，我们将在load_session方法中看到这个验证过程。

9-11是一种特殊情况，session_data的数据不能超过4k，不然的话没有任何cookie数据被写入。顺便说一句，session中应该保存极少的数据，最好只有原始的数据类型如整数、字符串等等，不然的话会影响你程序的性能。至于第9行为什么要减去@key的长度，因为最后写到cookie里面的数据是"@key = session_data"这样子的。

12-16行把session数据写入到cookie。cookie在这里是一个Hash，它的值就是我们在前面几步计算得到的session_data，如果用户设置了过期时间的话，那么它被解读为从现在开始的秒数。最后commit_session用Utils.set_cookie_header!把这个cookie哈希表写入到响应头中。具体如何写入，我们放到后面去讨论。

16行写入的时候，cookie还合并了来自rack.session.options的值，这个Hash其实就是中间件初始化的那些参数，包括:domain、:path:和expire_after。(在下面load_session的第18行)

现在我们可以比较容易理解加载cookie的工作了，它基本上是commit_session的一个逆向操作:

```
1      def load_session(env)
2        request = Rack::Request.new(env)
3        session_data = request.cookies[@key]
4
5        if @secret && session_data
6         session_data, digest = session_data.split("--")
7          session_data = nil  unless digest == generate_hmac(session_data)
8        end
9
10        begin
11         session_data = session_data.unpack("m*").first
12          session_data = Marshal.load(session_data)
13          env["rack.session"] = session_data
14        rescue
15         env["rack.session"] = Hash.new
16        end
17
18       env["rack.session.options"] = @default_options.dup
19     end
```

1-2行从请求中读取session数据。注意只是取出了关键字为@key的那个cookie。request如何解析cookie的过程涉及到cookie的相关协议，将在后面详细描述。

如果你设置了保密码secret，那么5-8进行解密，在前面commit_session的过程中，我们用--把session的原始数据和加密后的数据连接起来了。所以这里首先分别获取原始数据和加密数据到session_data和digest，并进行比较。只有当:

```
digest == generate_hmac(session_data)
```

条件成立的时候，我们才能认为这个session_data是合法的，不然session_data将被设置为nil–从而导致15行被执行，session将是一个空哈希。

第11行把早先commit_session利用pack编码的数据进行Base64解码，然后在12行重新加载为原始的Hash表，最后13行设置到env的rack.session关键字。

回忆commit_session第16行，cookie写入之前合并来自rack.session.options的选项

```
12          options = env["rack.session.options"]
16          Utils.set_cookie_header!(headers, @key, cookie.merge(options))
```

而rack.session.options的数据正是来自初始化Rack::Session::Cookie中间件的参数。

设置cookie头

真正把cookie写到响应头的方法是Utils的set_cookie_header!方法。
set_cookie_header!方法可以分为两个主要部分。首先是处理value为Hash的情况:

```
def set_cookie_header!(header, key, value)
  case value
  when Hash
    domain  = "; domain="  + value[:domain] if value[:domain]
    path    = "; path="    + value[:path]   if value[:path]
    # According to RFC 2109, we need dashes here.
    # N.B.: cgi.rb uses spaces...
    expires = "; expires=" + value[:expires].clone.gmtime.
      strftime("%a, %d-%b-%Y %H:%M:%S GMT") if value[:expires]
    secure = "; secure"  if value[:secure]
    httponly = "; HttpOnly" if value[:httponly]
    value = value[:value]
  end
```

Rack::Session::Cookie中间件调用set_cookie_header!方法的时候，value就是一个Hash，其中包含了和session的各种cookie选项。根据5.6.1(p. 86)描述的cookie相关规范，代码实现下面的功能:

- 如果value[:domain]选项存在，则cookie的domain属性为"; domain= + value[:domain]"
- 如果value[:path]选项存在，则cookie的path属性为 "; path= + value[:path]"
- 如果value[:expires]选项存在，则cookie的expires属性为"; expires= +value[:expires]转换为GMT时间格式的值"
- 如果value[:secure]选项存在，则cookie的secure属性为"; secure"(回忆这个cookie属性其实是一个boolen值)
- 如果value[: httponly]选项存在，则cookie的secure属性为"; HttpOnly"(HttpOnly是一个安全相关的cookie选项，并非所有浏览器都支持)
- 从value哈希中取得真正的key对应的value，即value[:value]，并将它设置为value变量的值

接下去的任务是真正地设置Set-Cookie响应头字段的值:

```
value = [value] unless Array === value
 cookie = escape(key) + "=" +
   value.map { |v| escape v }.join("&") +
   "#{domain}#{path}#{expires}#{secure}#{httponly}"
```

```
case header["Set-Cookie"]
when Array
  header["Set-Cookie"] << cookie
when String
  header["Set-Cookie"] = [header["Set-Cookie"], cookie]
when nil
  header["Set-Cookie"] = cookie
end
nil
```

如果value中包括多个值，用“&”符号它们连接起来，然后把所有的cookie属性加在后面，我们就得到了一个完整的cookie值，形如:

```
rack.session=......;domain=....;path=.....;expires=...;secure;HttpOnly
```

代码的最后判断header中是否已经存在Set-Cookie的值：如果有的话，header[Set-Cookie]加变成包含多个cookie的数组，不然的话，直接设置为当前的cookie。

记性好的读者可能会注意到5.4.2(p. 67)中我们曾经讲到过header的所有值必须被字符串，包括Set-Cookie的检查:

```
## === The Headers
def check_headers(header)
  .....
  header.each { |key, value|

    ........
    ## The values of the header must be Strings,
    assert("a header value must be a String, but the value of " +
      "'#{key}' is a #{value.class}") { value.kind_of? String }
    ## consisting of lines (for multiple header values, e.g. multiple
    ## <tt>Set-Cookie</tt> values) seperated by "\n".
    value.split("\n").each { |item|
      ## The lines must not contain characters below 037.
      assert("invalid header value #{key}: #{item.inspect}") {
        item !~ /[\000-\037]/
      }
    }
  }
end
```

如果是多个cookie的话，那么Set-Cookie对应的多个cookie也应该用“\n”分开，而不是一个数组。

确实，如果header是一个普通的Hash，那么上面的检查就会出错。然而，某些中间件（我们前面已经看到过）会使用一个HeaderHash，它的each实现如下:

```
def each
```

```
    super do |k, v|
      yield(k, v.respond_to?(:to_ary) ? v.to_ary.join("\n") : v)
    end
  end
```

如果HeaderHash中某一个值是数组，那么这个会首先把这个数组中所有的成员用“\n”连接成一个字符串。这正好符合Rack::Lint的要求。所以我们在编写自己中间件的时候，应该尽量使用HeaderHash而不是Hash处理响应头。

5.6.3 ID session

Rack::Session::Cookie提供了一种在Cookie中直接存放Session的方法。会话的数据在响应的时候写到cookie中传回到客户端保存，客户端请求的时候则把数据重新提交回服务端。

这种做法有一些问题存在。首先如果session中包含的数据太多，那么由于每一次请求/响应都涉及到对象的加载和序列化，就会对系统的性能造成比较大的影响。另外一方面，这些数据也可能在客户端存储和网络的过程中造成安全隐患。所以一般来说，我们不提倡在session中保存很多Ruby对象和数据。最常见的做法是只保存一个用户的ID。

Rack::Session::Abstract::ID类提供了一个简单的框架，可以用它来实现基于id的会话管理。Cookie中的会话数据只包含一个简单的id。你可以覆盖这个框架的某些部分，从而实现你自己的会话管理中间件。

缺省参数

ID抽象中间件的缺省选项包括:

```
    DEFAULT_OPTIONS = {
      :path =>          '/',
      :domain =>        nil,
      :expire_after =>  nil,
      :secure =>        false,
      :httponly =>      true,
      :defer =>         false,
      :renew =>         false,
      :sidbits =>       128
    }
```

除了最后面三个之外，其他所有选项我们应该已经比较熟悉了。余下三个选项的含义分别为:

- defer 如果设置defer为true,那么响应头中将不会设置cookie(暂时还不知道有什么用处)

- renew，如果设置此选项为true，那么在具体的会话管理实现中，不应该把原先客户端通过请求发送的session_id，而是每次生成一个新的session_id，并把原先session_id对应的会话数据和这个新的session_id对应。注意：renew的优先级高于defer，也就是即使defer设置为true，只要设置了renew为true，那么cookie也会被写入到响应头中。

- sidbits: 生成的session_id长度为多少个bit。ID类提供了一个实用的generate_sid方法可以供你的具体实现使用:

```
def generate_sid
  "%0#{@default_options[:sidbits] / 4}x" %
    rand(2**@default_options[:sidbits] - 1)
end
```

Figure 5.2: 生成随机的会话id

它利用rand方法来生成一个随机的16进制字符串。当然，你完全可以自己另外写一个方法来生成session_id。

另外，和Rack::Session::Cookie一样，你可以在initialize中指定session在cookie中的名称:

```
def initialize(app, options={})
  @app = app
  @key = options[:key] || "rack.session"
  @default_options = self.class::DEFAULT_OPTIONS.merge(options)
end
```

主要方法

ID类的方法包括:

- call
- load_session
- commit_session
- get_session
- set_session

call的实现:

```
def call(env)
  context(env)
end

def context(env, app=@app)
  load_session(env)
  status, headers, body = app.call(env)
  commit_session(env, status, headers, body)
end
```

主要的枝干和我们前面分析过的Rack::Session::Cookie没什么大区别，就是加载会话数据、处理请求、提交会话数据三个步骤。

load_session的具体实现也大致和Rack::Session::Cookie的load_session相同:

```
1     def load_session(env)
2       request = Rack::Request.new(env)
3       session_id = request.cookies[@key]
4
5       begin
6         session_id, session = get_session(env, session_id)
7         env['rack.session'] = session
8       rescue
9         env['rack.session'] = Hash.new
10      end
11
12      env['rack.session.options'] = @default_options.
13        merge(:id => session_id)
14    end
```

Figure 5.3: Rack::Session::Abstract::ID 的load_session方法

主要的不同在于:

1. 从request.cookies[@key]取得客户端保存session_id。因为我们只在cookie的session数据中保存了一个id的值。

2. 用这个客户端保存的session_id调用get_session方法获得服务端的session_id和session 而get_session是一个未实现的方法，具体的session实现应该覆盖这个方法，决定如何从客户端cookie中的session_id得到服务端的session_id和session。

   ```
   def get_session(env, sid)
     raise '#get_session not implemented.'
   end
   ```

3. 服务端的session_id连同中间件的缺省参数被设置为env['rack.session.options']的值，以供后面的commit_session获取服务端session_id和其他用途。

commit_session的具体实现也大致和Rack::Session::Cookie的commit_session相同:

```
def commit_session(env, status, headers, body)
  session = env['rack.session']
  options = env['rack.session.options']
  session_id = options[:id]

  if not session_id = set_session(env, session_id, session, options)
    env["rack.errors"].puts(
      "Warning! #{self.class.name} failed to save session. Content dropped.")
  elsif options[:defer] and not options[:renew]
    env["rack.errors"].puts(
              "Defering cookie for #{session_id}") if $VERBOSE
  else
    cookie = Hash.new
    cookie[:value] = session_id
    cookie[:expires] = Time.now + options[:expire_after] unless
                  options[:expire_after].nil?
    Utils.set_cookie_header!(headers, @key, cookie.merge(options))
  end

  [status, headers, body]
end
```

主要的不同在于:

1. env['rack.session.options'][:id]取到服务端的session_id，它的值是在load_session中设置的。

2. 用这个服务端的session_id调用set_session方法获得客户端的session_id，然后这个客户端session_id被写入cookie。一种情况例外，即设置了defer选项而且没有设置renew选项-此时不会把session_id写入cookie。和get_session一样，set_session也是一个未实现的方法，需要具体的中间件去覆盖它:

   ```
   def set_session(env, sid, session, options)
     raise '#set_session not implemented.'
   end
   ```

实现具体的中间件

因此，如果我们要实现一个具体的基于ID的中间件，我们可以继承Rack::Session::Abstact::ID类，并至少实现:

- get_session(env, sid): 其中sid为客户端cookie中得到的会话id，你的实现可以根据这个id得到保存在服务端的对应会话数据，并返回[session_id,session]。分别如下:

- ○ 返回的session_id是服务端的会话id，根据你的需要，服务端的session_id和客户端的session_id可以相同也可以不同，只要能够建立起一一对应关系。
- ○ 服务端把客户端session id对应的具体session数据保存在哪里，这看不同的实现了，可能是在数据库、文件、缓存等等。

 还有一点要注意的是，某些时候客户端的session id可能是nil(显然第一次用户请求就属于这种情况)，那么你的实现应该产生一个新的id。

- set_session(env, sid, session, options)：其中sid是服务端的会话id，options中则包含了使用这个中间件的参数。这个方法应该用session参数去更新保存在服务端的会话数据，并返回对应的客户端session id。

接下去的5.6.4(p. 101)和5.6.5(p. 105)是ID session的两个具体实现。

5.6.4 Memcache Session

Rack::Session::Memcache是一个基于Rack::Session::Abstract::ID(5.6.3(p. 96))的具体实现。因此，它的session_id是通过cookie在客户端和服务端之间传递，而它的会话数据则存放在memcached缓存服务器(http://memcached.org/)。

参数

除了ID类缺省的参数之外，这个中间件还有两个额外的缺省参数，和memcached相关：

```
DEFAULT_OPTIONS = Abstract::ID::DEFAULT_OPTIONS.merge \
  :namespace => 'rack:session',
  :memcache_server => 'localhost:11211'
```

这两个参数和memcached服务器相关：

namespace ：memcached相当于是一个大哈希表。例如：

 foo => bar[1]

 在memcached服务器设置关键字foo对应的值为bar。

 问题是一个应用程序的不同部分或者不同应用程序可能都需要设置关键字foo对应的值。为了防止它们之间相互冲突，你可以指定一个名字空间。例如，你设置了名字空间为rack.session，那么当你用关键字foo去设置的时候，实际上memcached是这样保存的：

 rack.session:foo => bar

[1]这只是一个示意，实际上memcached有自己特定的协议来设置

memcache_server : 指定memcached服务器的主机和端口。你可以指定多个主机和端口组合，如:

`['localhost:11211',"127.0.0.1:11211"]`

中间件的初始化过程检测是否能和memcached服务器连接:

```
def initialize(app, options={})
  super

  @mutex = Mutex.new
  mserv = @default_options[:memcache_server]
  mopts = @default_options.
    reject{|k,v| MemCache::DEFAULT_OPTIONS.include? k }
  @pool = MemCache.new mserv, mopts
  unless @pool.active? and @pool.servers.any?{|c| c.alive? }
    raise 'No memcache servers'
  end
end
```

其中的@mutex变量为后面同步逻辑所用。

该中间件使用了MemCache客户端[2]:

`MemCache.new mserv, mopts`

建立和memcached服务器的链接。两个参数分别为:

mserv 服务器的地址和端口列表，也正是我们在初始化中间件时提供的memcache_server参数的值。

mopts 是对memcached服务器进行操作的一些选项。这意味着你可以直接为中间件提供这些参数，例如:

```
use Rack::Session::Memcache,
                :memcache_server => 'localhost:11211',
                :namespace => 'rack.session',
                :multithread=>true,
                :failover=>true
```

等等。具体的参数请参考MemCache的代码或者文档(http://github.com/mperham/memcache-client/blob/master/lib/memcache.rb)。

要问这里的@default_options参数从何而来，请参考Rack::Session::Abstract::ID (5.6.3(p. 96))的initialize实现。

[2] http://github.com/mperham/memcache-client。它也是Rails使用的缺省memcached客户端。

get_session

第一个要覆盖的方法是get_session，前面已经讲过:

> get_session(env, sid): 其中sid为客户端cookie中得到的会话id，你的实现可以根据这个id得到保存在服务端的对应会话数据，并返回[session_id,session]。

我们先来看看get_session的大体框架，具体的session数据获取过程先被省略了:

```
1     def get_session(env, session_id)
2        @mutex.lock if env['rack.multithread']
         ..........
         ..........
         ..........
11      rescue MemCache::MemCacheError, Errno::ECONNREFUSED
12        # MemCache server cannot be contacted
13        warn "#{self} is unable to find memcached server."
14        warn $!.inspect
15        return [ nil, {} ]
16      ensure
17        @mutex.unlock if @mutex.locked?
18      end
```

如果env['rack.multithread']的值为true，则表示代码可能在多线程下运行，因此在开头和结尾分别用@mutex.lock和@mutex.unlock来保护这一临界区。

rescue子句处理对memcached缓存操作出错的情况，或者无法和memcached进行连接，此时除了做一些日志的工作意外，最后还返回[nil, {}]–即session_id为nil，而session则是一个空哈希–ID抽象类中的后续代码会导致cookie中没有session数据被发送回客户端。

用省略号代替的具体实现过程如下:

```
3        unless session_id and session = @pool.get(session_id)
4          session_id, session = generate_sid, {}
5          unless /^STORED/ =~ @pool.add(session_id, session)
6            raise "Session collision on '#{session_id.inspect}'"
7          end
8        end
9        session.instance_variable_set '@old', @pool.get(session_id, true)
10      return [session_id, session]
```

第3行表示在两种情况下:

- session_id为nil，通常这是用户第一次访问。或者:

- @pool.get(session_id)没有这个session_id对应的值–极有可能memcached服务器中，session_id对应的条目已经被移出缓存。

我们需要重新生成session_id，而session的数据只能是一个空哈希。不然的话，session将包含取自缓存服务器的、这个session_id所对应的session数据。

```
4         session_id, session = generate_sid, {}
```

session_id使用generate_sid重新生成的，生成以后要把这个session_id和session加入到缓存。Memcache的add方法:

```
add(key,value)
```

只会在memcached缓存服务器中不存在这个key的时候加入新的key/value对应条目，并返回“STORED”。不然的话会返回“NOT_STORED”表示无法保存，这个时候第6行就会抛出异常，第15行就会返回[nil, {}]。

在第9行，当前的session设置了一个实例变量@old，它的值为

```
@pool.get(session_id, true)
```

MemCache的get方法可以有两个参数，第一个参数是关键字，第二个布尔参数表示是否取得原始数据(*raw*)。

> Memcached缓存服务器根本不知道什么是Ruby对象。因此在保存任何对象前，Memcache客户端首先会用Marshal.dump把它输出为字符串，并保存到缓存服务器–这就是原始数据。而MemCache客户端从memcached服务器取得的数据就是保存的原始数据。如果raw设为false(缺省情况)，那么Memcache客户端会用Marshal.load把它重新加载为对象。而如果raw设置为true，那么就直接返回从memcached服务器得到的原始数据。

最后在第10行返回了session_id和session。

之所以要把这段代码放在一个临界区内部的原因是:

- 防止不同的用户使用相同的session_id, generate_sid方法的定义如下:

  ```
  def generate_sid
    loop do
      sid = super
      break sid unless @pool.get(sid, true)
    end
  end
  ```

  super调用抽象ID类提供的(5.2(p. 97))随机生成方法。虽然可能性比较小，但是不同的用户还是有机会生成相同的session_id。因此这里的generate_sid根据超类的generate_sid生成

- 多个线程可能同时在判断session是否存在和设置新的session数据之间被调度–从而可能导致为同一个用户多次生成不同的session_id，不同的session数据–这显然是不允许的。

然而，即使是互斥也不能避免session_id的冲突，原因是不同的Ruby进程可能同时为某一个用户设置session数据，因此可能造成两个不同的Ruby进程为同一个

5.6.5 Pool Session