Sinatra

This page is also available in English, French, German, Hungarian, Korean, Portuguese (Brazilian), Portuguese (European), Russian, Spanish and Japanese.

簡介

注：本文档是英文版的翻译，内容更新有可能不及时。 如有不一致的地方，请以英文版为准。

Sinatra是一个基于Ruby语言的DSL（ 领域专属语言），可以轻松、快速的创建web应用。

```
# myapp.rb
require 'sinatra'

get '/' do
  'Hello world!'
end
```

安装gem，然后运行：

```
gem install sinatra
ruby myapp.rb
```

在该地址查看： http://localhost:4567

这个时候访问地址将绑定到 127.0.0.1 和 localhost ，如果使用 vagrant 进行开发，访问会失败，此时就需要进行 ip 绑定了：

```
ruby myapp.rb -o 0.0.0.0
```

~~~~-o~~~~ 这个参数就是进行 Listening 时候监听的绑定，能从通过 IP、127.0.0.1、localhost + 端口号进行访问。

安装Sintra后，最好再运行`gem install thin`安装Thin。这样，Sinatra会优先选择Thin作为服务器。

~~~~

路由(route)

在Sinatra中，一个路由分为两部分：HTTP方法(GET, POST等)和URL匹配范式。 每个路由都有一个要执行的代码块：

```
get '/' do
  .. 显示内容 ..
end

post '/' do
  .. 创建内容 ..
end

put '/' do
  .. 更新内容 ..
end

delete '/' do
  .. 删除内容 ..
end

options '/' do
  .. 显示命令列表 ..
end

link '/' do
  .. 建立某种联系 ..
end

unlink '/' do
  .. 解除某种联系 ..
end
```

路由按照它们被定义的顺序进行匹配。 第一个与请求匹配的路由会被调用。

路由范式可以包括具名参数，可通过params哈希表获得：

```
get '/hello/:name' do
  # 匹配 "GET /hello/foo" 和 "GET /hello/bar"
  # params['name'] 的值是 'foo' 或者 'bar'
  "Hello #{params['name']}!"
end
```

你同样可以通过代码块参数获得具名参数：

```
get '/hello/:name' do |n|
  "Hello #{n}!"
end
```

路由范式也可以包含通配符参数， 可以通过params['splat']数组获得。

```
get '/say/*/to/*' do
  # 匹配 /say/hello/to/world
```

```
  params['splat'] # => ["hello", "world"]
end

get '/download/*.*' do
  # 匹配 /download/path/to/file.xml
  params['splat'] # => ["path/to/file", "xml"]
end
```

通过正则表达式匹配的路由：

```
get /\A\/hello\/([\w]+)\z/ do
  "Hello, #{params['captures'].first}!"
end
```

或者使用代码块参数：

```
get %r{/hello/([\w]+)} do |c|
  "Hello, #{c}!"
end
```

条件

路由也可以包含多样的匹配条件，比如user agent：

```
get '/foo', :agent => /Songbird (\d\.\d)[\d\/]*?/ do
  "你正在使用Songbird，版本是 #{params['agent'][0]}"
end

get '/foo' do
  # 匹配除Songbird以外的浏览器
end
```

其他可选的条件是 host_name 和 provides：

```
get '/', :host_name => /^admin\./ do
  "管理员区域，无权进入！"
end

get '/', :provides => 'html' do
  haml :index
end

get '/', :provides => ['rss', 'atom', 'xml'] do
  builder :feed
end
```

你也可以自定义条件：

```
set(:probability) { |value| condition { rand <= value } }

get '/win_a_car', :probability => 0.1 do
  "You won!"
end
```

```
get '/win_a_car' do
  "Sorry, you lost."
end
```

返回值

路由代码块的返回值至少决定了返回给HTTP客户端的响应体， 或者至少决定了在Rack堆栈中的下一个中间件。 大多数情况下，将是一个字符串，就像上面的例子中的一样。 但是其他值也是可以接受的。

你可以返回任何对象，或者是一个合理的Rack响应， Rack body对象或者HTTP状态码：

那样，我们可以轻松的实现例如流式传输的例子：

```
class Stream
  def each
    100.times { |i| yield "#{i}\n" }
  end
end

get('/') { Stream.new }
```

自定义路由匹配器

如上显示，Sinatra内置了对于使用字符串和正则表达式作为路由匹配的支持。 但是，它并没有只限于此。 你可以非常容易地定义你自己的匹配器:

```
class AllButPattern
  Match = Struct.new(:captures)

  def initialize(except)
    @except   = except
    @captures = Match.new([])
  end

  def match(str)
    @captures unless @except === str
  end
end

def all_but(pattern)
  AllButPattern.new(pattern)
end

get all_but("/index") do
  # ...
end
```

上面的例子可能太繁琐了， 因为它也可以用更简单的方式表述:

```
get // do
  pass if request.path_info == "/index"
  # ...
end
```

或者，使用消极向前查找:

```
get %r{^(?!/index$)} do
  # ...
end
```

静态文件

静态文件是从 ./public_folder 目录提供服务。你可以通过设置:public 选项设定一个不同的位置：

```
set :public_folder, File.dirname(__FILE__) + '/static'
```

请注意public目录名并没有被包含在URL之中。文件 ./public/css/style.css是通过 http://example.com/css/style.css地址访问的。

视图 / 模板

模板被假定直接位于./views目录。 要使用不同的视图目录：

```
set :views, File.dirname(__FILE__) + '/templates'
```

重要提示：你只可以通过符号引用模板， 即使它们在子目录下 （在这种情况下，使用 :'subdir/template'）。 如果你不用符号、而用字符串的话， 填充方法会只把你传入的字符串当成内容显示出来，而不调用模板。

Haml模板

需要引入 haml gem/library以填充 HAML 模板：

```
# 你需要在你的应用中引入 haml
require 'haml'

get '/' do
  haml :index
end
```

填充 ./views/index.haml。

Haml的选项 可以通过Sinatra的配置全局设定， 参见 选项和配置， 也可以个别的被覆盖。

```
set :haml, {:format => :html5 } # 默认的Haml输出格式是 :xhtml

get '/' do
  haml :index, :haml_options => {:format => :html4 } # 被覆盖，变成:html4
end
```

Erb模板

```
# 你需要在你的应用中引入 erb
require 'erb'
```

```
get '/' do
  erb :index
end
```

这里调用的是 ./views/index.erb

Erubis

需要引入 erubis gem/library以填充 erubis 模板：

```
# 你需要在你的应用中引入 erubis
require 'erubis'

get '/' do
  erubis :index
end
```

这里调用的是 ./views/index.erubis

使用Erubis代替Erb也是可能的:

```
require 'erubis'
Tilt.register :erb, Tilt[:erubis]

get '/' do
  erb :index
end
```

使用Erubis来填充 ./views/index.erb。

Builder 模板

需要引入 builder gem/library 以填充 builder templates：

```
# 需要在你的应用中引入builder
require 'builder'

get '/' do
  builder :index
end
```

这里调用的是 ./views/index.builder。

Nokogiri 模板

需要引入 nokogiri gem/library 以填充 nokogiri 模板：

```
# 需要在你的应用中引入 nokogiri
require 'nokogiri'

get '/' do
  nokogiri :index
end
```

这里调用的是 ./views/index.nokogiri。

Sass 模板

需要引入 haml 或者 sass gem/library 以填充 Sass 模板：

```
# 需要在你的应用中引入 haml 或者 sass
require 'sass'

get '/stylesheet.css' do
  sass :stylesheet
end
```

这里调用的是 ./views/stylesheet.sass。

Sass 的选项 可以通过Sinatra选项全局设定， 参考 选项和配置（英文）, 也可以在个体的基础上覆盖。

```
set :sass, {:style => :compact } # 默认的 Sass 样式是 :nested

get '/stylesheet.css' do
  sass :stylesheet, :style => :expanded # 覆盖
end
```

Scss 模板

需要引入 haml 或者 sass gem/library 来填充 Scss templates：

```
# 需要在你的应用中引入 haml 或者 sass
require 'sass'

get '/stylesheet.css' do
  scss :stylesheet
end
```

这里调用的是 ./views/stylesheet.scss。

Scss的选项 可以通过Sinatra选项全局设定， 参考 选项和配置（英文）, 也可以在个体的基础上覆盖。

```
set :scss, :style => :compact # default Scss style is :nested

get '/stylesheet.css' do
  scss :stylesheet, :style => :expanded # overridden
end
```

Less 模板

需要引入 less gem/library 以填充 Less 模板：

```
# 需要在你的应用中引入 less
require 'less'

get '/stylesheet.css' do
  less :stylesheet
end
```

这里调用的是 ./views/stylesheet.less。

Liquid 模板

需要引入 liquid gem/library 来填充 Liquid 模板：

```
# 需要在你的应用中引入 liquid
require 'liquid'

get '/' do
  liquid :index
end
```

这里调用的是 ./views/index.liquid。

因为你不能在Liquid 模板中调用 Ruby 方法 (除了 yield)， 你几乎总是需要传递locals给它：

```
liquid :index, :locals => { :key => 'value' }
```

Markdown 模板

需要引入 rdiscount gem/library 以填充 Markdown 模板：

```
# 需要在你的应用中引入rdiscount
require "rdiscount"

get '/' do
  markdown :index
end
```

这里调用的是 ./views/index.markdown (md 和 mkd 也是合理的文件扩展名)。

在markdown中是不可以调用方法的，也不可以传递 locals给它。 你因此一般会结合其他的填充引擎来使用它：

```
erb :overview, :locals => { :text => markdown(:introduction) }
```

请注意你也可以从其他模板中调用 markdown 方法：

```
%h1 Hello From Haml!
%p= markdown(:greetings)
```

既然你不能在Markdown中调用Ruby，你不能使用Markdown编写的布局。 不过，使用其他填充引擎作为模版的布局是可能的， 通过传递:layout_engine选项:

```
get '/' do
  markdown :index, :layout_engine => :erb
end
```

这将会调用 ./views/index.md 并使用 ./views/layout.erb 作为布局。

请记住你可以全局设定这个选项:

```
set :markdown, :layout_engine => :haml, :layout => :post

get '/' do
  markdown :index
end
```

这将会调用 ./views/index.markdown (和任何其他的 Markdown 模版) 并使用 ./views/post.haml 作为布局.

也可能使用BlueCloth而不是RDiscount来解析Markdown文件:

```
require 'bluecloth'

Tilt.register 'markdown', BlueClothTemplate
Tilt.register 'mkd',      BlueClothTemplate
Tilt.register 'md',       BlueClothTemplate

get '/' do
  markdown :index
end
```

使用BlueCloth来填充 ./views/index.md 。

Textile 模板

需要引入 RedCloth gem/library 以填充 Textile 模板：

```
# 在你的应用中引入redcloth
require "redcloth"

get '/' do
  textile :index
end
```

这里调用的是 ./views/index.textile。

在textile中是不可以调用方法的，也不可以传递 locals给它。 你因此一般会结合其他的填充引擎来使用它：

```
erb :overview, :locals => { :text => textile(:introduction) }
```

请注意你也可以从其他模板中调用textile方法：

```
%h1 Hello From Haml!
%p= textile(:greetings)
```

既然你不能在Textile中调用Ruby，你不能使用Textile编写的布局。 不过，使用其他填充引擎作为模版的布局是可能的， 通过传递:layout_engine选项:

```
get '/' do
  textile :index, :layout_engine => :erb
end
```

这将会填充 ./views/index.textile 并使用 ./views/layout.erb 作为布局。

请记住你可以全局设定这个选项:

```
set :textile, :layout_engine => :haml, :layout => :post

get '/' do
  textile :index
end
```

这将会调用 ./views/index.textile (和任何其他的 Textile 模版) 并使用 ./views/post.haml 作为布局.

RDoc 模板

需要引入 RDoc gem/library 以填充RDoc模板：

```
# 需要在你的应用中引入rdoc/markup/to_html
require "rdoc"
require "rdoc/markup/to_html"

get '/' do
  rdoc :index
end
```

这里调用的是 ./views/index.rdoc。

在rdoc中是不可以调用方法的，也不可以传递locals给它。 你因此一般会结合其他的填充引擎来使用它：

```
erb :overview, :locals => { :text => rdoc(:introduction) }
```

请注意你也可以从其他模板中调用rdoc方法：

```
%h1 Hello From Haml!
%p= rdoc(:greetings)
```

既然你不能在RDoc中调用Ruby，你不能使用RDoc编写的布局。 不过，使用其他填充引擎作为模版的布局是可能的， 通过传递:layout_engine选项:

```
get '/' do
  rdoc :index, :layout_engine => :erb
end
```

这将会调用 ./views/index.rdoc 并使用 ./views/layout.erb 作为布局。

请记住你可以全局设定这个选项:

```
set :rdoc, :layout_engine => :haml, :layout => :post

get '/' do
  rdoc :index
end
```

这将会调用 ./views/index.rdoc (和任何其他的 RDoc 模版) 并使用 ./views/post.haml 作为布局.

Radius 模板

需要引入 radius gem/library 以填充 Radius 模板：

```
# 需要在你的应用中引入radius
require 'radius'

get '/' do
  radius :index
end
```

这里调用的是 ./views/index.radius。

因为你不能在Radius 模板中调用 Ruby 方法 (除了 yield)， 你几乎总是需要传递locals给它：

```
radius :index, :locals => { :key => 'value' }
```

Markaby 模板

需要引入markaby gem/library以填充Markaby模板：

```
#需要在你的应用中引入 markaby
require 'markaby'

get '/' do
  markaby :index
end
```

这里调用的是 ./views/index.mab。

你也可以使用嵌入的 Markaby:

```
get '/' do
  markaby { h1 "Welcome!" }
end
```

Slim 模板

需要引入 slim gem/library 来填充 Slim 模板：

```
# 需要在你的应用中引入 slim
require 'slim'

get '/' do
  slim :index
end
```

这里调用的是 ./views/index.slim。

Creole 模板

需要引入 creole gem/library 来填充 Creole 模板：

```
# 需要在你的应用中引入 creole
require 'creole'
```

```
get '/' do
  creole :index
end
```

这里调用的是 ./views/index.creole。

CoffeeScript 模板

需要引入 coffee-script gem/library 并至少满足下面条件一项 以执行Javascript:

- node (来自 Node.js) 在你的路径中
- 你正在运行 OSX
- therubyracer gem/library

请察看 github.com/josh/ruby-coffee-script 获取更新的选项。

现在你可以调用 CoffeeScript 模版了:

```
# 需要在你的应用中引入coffee-script
require 'coffee-script'

get '/application.js' do
  coffee :application
end
```

这里调用的是 ./views/application.coffee。

嵌入模板字符串

```
get '/' do
  haml '%div.title Hello World'
end
```

调用嵌入模板字符串。

在模板中访问变量

模板和路由执行器在同样的上下文求值。 在路由执行器中赋值的实例变量可以直接被模板访问。

```
get '/:id' do
  @foo = Foo.find(params['id'])
  haml '%h1= @foo.name'
end
```

或者，显式地指定一个本地变量的哈希：

```
get '/:id' do
  foo = Foo.find(params['id'])
  haml '%h1= foo.name', :locals => { :foo => foo }
end
```

典型的使用情况是在别的模板中按照局部模板的方式来填充。

内联模板

模板可以在源文件的末尾定义：

```
require 'sinatra'

get '/' do
  haml :index
end

__END__

@@ layout
%html
  = yield

@@ index
%div.title Hello world!!!!!
```

注意：引入sinatra的源文件中定义的内联模板才能被自动载入。 如果你在其他源文件中有内联模板， 需要显式执行调用enable :inline_templates。

具名模板

模板可以通过使用顶层 template 方法定义：

```
template :layout do
  "%html\n  =yield\n"
end

template :index do
  '%div.title Hello World!'
end

get '/' do
  haml :index
end
```

如果存在名为“layout”的模板，该模板会在每个模板填充的时候被使用。 你可以单独地通过传送 :layout => false来禁用， 或者通过set :haml, :layout => false来禁用他们。

```
get '/' do
  haml :index, :layout => !request.xhr?
end
```

关联文件扩展名

为了关联一个文件扩展名到一个模版引擎，使用 Tilt.register。比如，如果你喜欢使用 tt 作为Textile模版的扩展名，你可以这样做:

```
Tilt.register :tt, Tilt[:textile]
```

添加你自己的模版引擎

首先，通过Tilt注册你自己的引擎，然后创建一个填充方法:

```
Tilt.register :myat, MyAwesomeTemplateEngine

helpers do
  def myat(*args) render(:myat, *args) end
end

get '/' do
  myat :index
end
```

这里调用的是 ./views/index.myat。察看 github.com/rtomayko/tilt 来更多了解Tilt.

过滤器

前置过滤器在每个请求前，在请求的上下文环境中被执行， 而且可以修改请求和响应。 在过滤器中设定的实例变量可以被路由和模板访问：

```
before do
  @note = 'Hi!'
  request.path_info = '/foo/bar/baz'
end

get '/foo/*' do
  @note #=> 'Hi!'
  params['splat'] #=> 'bar/baz'
end
```

后置过滤器在每个请求之后，在请求的上下文环境中执行， 而且可以修改请求和响应。 在前置过滤器和路由中设定的实例变量可以被后置过滤器访问：

```
after do
  puts response.status
end
```

请注意：除非你显式使用 body 方法，而不是在路由中直接返回字符串， 消息体在后置过滤器是不可用的，因为它在之后才会生成。

过滤器可以可选地带有范式， 只有请求路径满足该范式时才会执行：

```
before '/protected/*' do
  authenticate!
end

after '/create/:slug' do |slug|
  session['last_slug'] = slug
end
```

和路由一样，过滤器也可以带有条件:

```
before :agent => /Songbird/ do
  # ...
```

```
end

after '/blog/*', :host_name => 'example.com' do
  # ...
end
```

辅助方法

使用顶层的 helpers 方法来定义辅助方法， 以便在路由处理器和模板中使用：

```
helpers do
  def bar(name)
    "#{name}bar"
  end
end

get '/:name' do
  bar(params['name'])
end
```

使用 **Sessions**

Session被用来在请求之间保持状态。如果被激活，每一个用户会话 对应有一个session哈希:

```
enable :sessions

get '/' do
  "value = " << session['value'].inspect
end

get '/:value' do
  session['value'] = params['value']
end
```

请注意 enable :sessions 实际上保存所有的数据在一个cookie之中。 这可能不会总是做你想要的（比如，保存大量的数据会增加你的流量）。 你可以使用任何的Rack session中间件，为了这么做， *不要*调用 enable :sessions，而是 按照自己的需要引入你的中间件：

```
use Rack::Session::Pool, :expire_after => 2592000

get '/' do
  "value = " << session['value'].inspect
end

get '/:value' do
  session['value'] = params['value']
end
```

挂起

要想直接地停止请求，在过滤器或者路由中使用：

```
halt
```

你也可以指定挂起时的状态码：

```
halt 410
```

或者消息体：

```
halt 'this will be the body'
```

或者两者;

```
halt 401, 'go away!'
```

也可以带消息头：

```
halt 402, {'Content-Type' => 'text/plain'}, 'revenge'
```

让路

一个路由可以放弃处理，将处理让给下一个匹配的路由，使用 pass：

```
get '/guess/:who' do
  pass unless params['who'] == 'Frank'
  'You got me!'
end

get '/guess/*' do
  'You missed!'
end
```

路由代码块被直接退出，控制流继续前进到下一个匹配的路由。 如果没有匹配的路由，将返回404。

触发另一个路由

有些时候，pass 并不是你想要的，你希望得到的是另一个路由的结果 。简单的使用 call 可以做到这一点:

```
get '/foo' do
  status, headers, body = call env.merge("PATH_INFO" => '/bar')
  [status, headers, body.map(&:upcase)]
end

get '/bar' do
  "bar"
end
```

请注意在以上例子中，你可以更加简化测试并增加性能，只要简单的移动

```
<tt>"bar"</tt>到一个被<tt>/foo</tt>
```

和 /bar同时使用的helper。

如果你希望请求被发送到同一个应用，而不是副本， 使用 call! 而不是 call.

如果想更多了解 call，请察看 Rack specification。

设定 消息体，状态码和消息头

通过路由代码块的返回值来设定状态码和消息体不仅是可能的，而且是推荐的。 但是，在某些场景中你可能想在作业流程中的特定点上设置消息体。 你可以通过 body 辅助方法这么做。 如果你这样做了， 你可以在那以后使用该方法获得消息体:

```
get '/foo' do
  body "bar"
end

after do
  puts body
end
```

也可以传一个代码块给 body，它将会被Rack处理器执行（ 这将可以被用来实现streaming，参见“返回值”）。

和消息体类似，你也可以设定状态码和消息头:

```
get '/foo' do
  status 418
  headers \
    "Allow"   => "BREW, POST, GET, PROPFIND, WHEN",
    "Refresh" => "Refresh: 20; http://www.ietf.org/rfc/rfc2324.txt"
  body "I'm a tea pot!"
end
```

如同 body, 不带参数的 headers 和 status 可以用来访问 他们你的当前值.

媒体(MIME)类型

使用 send_file 或者静态文件的时候，Sinatra可能不能识别你的媒体类型。 使用 mime_type 通过文件扩展名来注册它们：

```
mime_type :foo, 'text/foo'
```

你也可以使用 content_type 辅助方法：

```
get '/' do
  content_type :foo
  "foo foo foo"
end
```

生成 URL

为了生成URL，你需要使用 url 辅助方法， 例如，在Haml中:

```
%a{:href => url('/foo')} foo
```

如果使用反向代理和Rack路由，生成URL的时候会考虑这些因素。

这个方法还有一个别名 to (见下面的例子).

浏览器重定向

你可以通过 redirect 辅助方法触发浏览器重定向:

```
get '/foo' do
  redirect to('/bar')
end
```

其他参数的用法，与 halt相同:

```
redirect to('/bar'), 303
redirect 'http://google.com', 'wrong place, buddy'
```

用 redirect back可以把用户重定向到原始页面:

```
get '/foo' do
  "<a href='/bar'>do something</a>"
end

get '/bar' do
  do_something
  redirect back
end
```

如果想传递参数给redirect，可以用query string:

```
redirect to('/bar?sum=42')
```

或者用session:

```
enable :sessions

get '/foo' do
  session['secret'] = 'foo'
  redirect to('/bar')
end

get '/bar' do
  session['secret']
end
```

缓存控制

要使用HTTP缓存，必须正确地设定消息头。

你可以这样设定 Cache-Control 消息头:

```
get '/' do
  cache_control :public
  "cache it!"
end
```

核心提示: 在前置过滤器中设定缓存.

```
before do
  cache_control :public, :must_revalidate, :max_age => 60
end
```

如果你正在用 expires 辅助方法设定对应的消息头 Cache-Control 会自动设定：

```
before do
  expires 500, :public, :must_revalidate
end
```

为了合适地使用缓存，你应该考虑使用 etag 和 last_modified方法。 推荐在执行繁重任务*之前*使用这些 helpers，这样一来， 如果客户端在缓存中已经有相关内容，就会立即得到显示。

```
get '/article/:id' do
  @article = Article.find params['id']
  last_modified @article.updated_at
  etag @article.sha1
  erb :article
end
```

使用 weak ETag 也是有可能的:

```
etag @article.sha1, :weak
```

这些辅助方法并不会为你做任何缓存，而是将必要的信息传送给你的缓存 如果你在寻找缓存的快速解决方案，试试 rack-cache:

```
require "rack/cache"
require "sinatra"

use Rack::Cache

get '/' do
  cache_control :public, :max_age => 36000
  sleep 5
  "hello"
end
```

发送文件

为了发送文件，你可以使用 send_file 辅助方法:

```
get '/' do
  send_file 'foo.png'
end
```

也可以带一些选项:

```
send_file 'foo.png', :type => :jpg
```

可用的选项有:

filename

响应中的文件名，默认是真实文件的名字。

last_modified

Last-Modified 消息头的值，默认是文件的mtime（修改时间）。

type

使用的内容类型，如果没有会从文件扩展名猜测。

disposition

用于 Content-Disposition，可能的包括： nil (默认), :attachment 和 :inline

length

Content-Length 的值，默认是文件的大小。

如果Rack处理器支持的话，Ruby进程也能使用除streaming以外的方法。 如果你使用这个辅助方法，Sinatra会自动处理range请求。

访问请求对象

传入的请求对象可以在请求层（过滤器，路由，错误处理） 通过 request 方法被访问：

```
# 在 http://example.com/example 上运行的应用
get '/foo' do
  request.body              # 被客户端设定的请求体（见下）
  request.scheme            # "http"
  request.script_name       # "/example"
  request.path_info         # "/foo"
  request.port              # 80
  request.request_method    # "GET"
  request.query_string      # ""
  request.content_length    # request.body的长度
  request.media_type        # request.body的媒体类型
  request.host              # "example.com"
  request.get?              # true (其他动词也具有类似方法)
  request.form_data?        # false
  request["SOME_HEADER"]    # SOME_HEADER header的值
  request.referrer          # 客户端的referrer 或者 '/'
  request.user_agent        # user agent (被 :agent 条件使用)
  request.cookies           # 浏览器 cookies 哈希
  request.xhr?              # 这是否是ajax请求?
  request.url               # "http://example.com/example/foo"
  request.path              # "/example/foo"
  request.ip                # 客户端IP地址
  request.secure?           # false（如果是ssl则为true）
  request.forwarded?        # true （如果是运行在反向代理之后）
  request.env               # Rack中使用的未处理的env哈希
end
```

一些选项，例如 script_name 或者 path_info 也是可写的：

```
before { request.path_info = "/" }

get "/" do
  "all requests end up here"
end
```

request.body 是一个IO或者StringIO对象：

```
post "/api" do
  request.body.rewind  # 如果已经有人读了它
```

```
  data = JSON.parse request.body.read
  "Hello #{data['name']}!"
end
```

附件

你可以使用 attachment 辅助方法来告诉浏览器响应 应当被写入磁盘而不是在浏览器中显示。

```
get '/' do
  attachment
  "store it!"
end
```

你也可以传递一个文件名:

```
get '/' do
  attachment "info.txt"
  "store it!"
end
```

查找模板文件

find_template 辅助方法被用于在填充时查找模板文件:

```
find_template settings.views, 'foo', Tilt[:haml] do |file|
  puts "could be #{file}"
end
```

这并不是很有用。但是在你需要重载这个方法 来实现你自己的查找机制的时候有用。 比如，如果你想支持多于一个视图目录:

```
set :views, ['views', 'templates']

helpers do
  def find_template(views, name, engine, &block)
    Array(views).each { |v| super(v, name, engine, &block) }
  end
end
```

另一个例子是为不同的引擎使用不同的目录:

```
set :views, :sass => 'views/sass', :haml => 'templates', :default => 'views'

helpers do
  def find_template(views, name, engine, &block)
    _, folder = views.detect { |k,v| engine == Tilt[k] }
    folder ||= views[:default]
    super(folder, name, engine, &block)
  end
end
```

你可以很容易地包装成一个扩展然后与他人分享！

请注意 find_template 并不会检查文件真的存在， 而是对任何可能的路径调用给入的代码块。这并不会带来

性能问题， 因为 render 会在找到文件的时候马上使用 break 。 同样的，模板的路径（和内容）会在除 development mode以外的场合 被缓存。你应该时刻提醒自己这一点， 如果你真的想写一个非常疯狂的方法。

配置

运行一次，在启动的时候，在任何环境下：

```
configure do
  # setting one option
  set :option, 'value'

  # setting multiple options
  set :a => 1, :b => 2

  # same as `set :option, true`
  enable :option

  # same as `set :option, false`
  disable :option

  # you can also have dynamic settings with blocks
  set(:css_dir) { File.join(views, 'css') }
end
```

只当环境 (RACK_ENV environment 变量) 被设定为 :production的时候运行：

```
configure :production do
  ...
end
```

当环境被设定为 :production 或者 :test的时候运行：

```
configure :production, :test do
  ...
end
```

你可以使用 settings 获得这些配置:

```
configure do
  set :foo, 'bar'
end

get '/' do
  settings.foo? # => true
  settings.foo  # => 'bar'
  ...
end
```

可选的设置

absolute_redirects

如果被禁用，Sinatra会允许使用相对路径重定向， 但是，Sinatra就不再遵守 RFC 2616标准 (HTTP 1.1), 该标准只允许绝对路径重定向。

如果你的应用运行在一个未恰当设置的反向代理之后， 你需要启用这个选项。注意 url 辅助方法 仍然会生成绝对 URL，除非你传入 false 作为第二参数。

默认禁用。

add_charset

设定 content_type 辅助方法会 自动加上字符集信息的多媒体类型。

你应该添加而不是覆盖这个选项: settings.add_charset << "application/foobar"

app_file
主应用文件，用来检测项目的根路径， views和public文件夹和内联模板。
bind
绑定的IP 地址 (默认: 0.0.0.0)。 仅对于内置的服务器有用。
default_encoding
默认编码 (默认为 "utf-8")。
dump_errors
在log中显示错误。
environment
当前环境，默认是 ENV['RACK_ENV']， 或者 "development" 如果不可用。
logging
使用logger
lock

对每一个请求放置一个锁， 只使用进程并发处理请求。

如果你的应用不是线程安全则需启动。 默认禁用。

method_override
使用 _method 魔法以允许在旧的浏览器中在 表单中使用 put/delete 方法
port
监听的端口号。只对内置服务器有用。
prefixed_redirects
是否添加 request.script_name 到 重定向请求，如果没有设定绝对路径。那样的话 redirect '/foo' 会和 redirect to('/foo')起相同作用。默认禁用。
public_folder
public文件夹的位置。
reload_templates
是否每个请求都重新载入模板。 在development mode和 Ruby 1.8.6 中被企业（用来 消除一个Ruby内存泄漏的bug）。
root
项目的根目录。
raise_errors
抛出异常（应用会停下）。
run
如果启用，Sinatra会开启web服务器。 如果使用rackup或其他方式则不要启用。
running
内置的服务器在运行吗？ 不要修改这个设置！
server
服务器，或用于内置服务器的列表。 默认是 ['thin', 'mongrel', 'webrick'], 顺序表明了 优先级。
sessions
开启基于cookie的sesson。
show_exceptions
在浏览器中显示一个stack trace。

static
: Sinatra是否处理静态文件。 当服务器能够处理则禁用。 禁用会增强性能。 默认开启。

views
: views 文件夹。

错误处理

错误处理在与路由和前置过滤器相同的上下文中运行， 这意味着你可以使用许多好东西，比如 haml, erb, halt，等等。

未找到

当一个 Sinatra::NotFound 错误被抛出的时候， 或者响应状态码是404，not_found 处理器会被调用：

```
not_found do
  'This is nowhere to be found'
end
```

错误

error 处理器，在任何路由代码块或者过滤器抛出异常的时候会被调用。 异常对象可以通过sinatra.error Rack 变量获得：

```
error do
  'Sorry there was a nasty error - ' + env['sinatra.error'].message
end
```

自定义错误：

```
error MyCustomError do
  'So what happened was...' + env['sinatra.error'].message
end
```

那么，当这个发生的时候：

```
get '/' do
  raise MyCustomError, 'something bad'
end
```

你会得到：

```
So what happened was... something bad
```

另一种替代方法是，为一个状态码安装错误处理器：

```
error 403 do
  'Access forbidden'
end

get '/secret' do
  403
end
```

或者一个范围：

```
error 400..510 do
  'Boom'
end
```

在运行在development环境下时，Sinatra会安装特殊的 not_found 和 error 处理器。

Rack 中间件

Sinatra 依靠 Rack, 一个面向Ruby web框架的最小标准接口。 Rack的一个最有趣的面向应用开发者的能力是支持“中间件”——坐落在服务器和你的应用之间， 监视 并/或 操作HTTP请求/响应以 提供多样类型的常用功能。

Sinatra 让建立Rack中间件管道异常简单， 通过顶层的 use 方法：

```
require 'sinatra'
require 'my_custom_middleware'

use Rack::Lint
use MyCustomMiddleware

get '/hello' do
  'Hello World'
end
```

use 的语义和在 Rack::Builder DSL(在rack文件中最频繁使用)中定义的完全一样。例如，use 方法接受 多个/可变 参数，包括代码块：

```
use Rack::Auth::Basic do |username, password|
  username == 'admin' && password == 'secret'
end
```

Rack中分布有多样的标准中间件，针对日志， 调试，URL路由，认证和session处理。 Sinatra会自动使用这里面的大部分组件， 所以你一般不需要显示地 use 他们。

测试

Sinatra的测试可以使用任何基于Rack的测试程序库或者框架来编写。 Rack::Test 是推荐候选：

```
require 'my_sinatra_app'
require 'minitest/autorun'
require 'rack/test'

class MyAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end
```

```
  def test_my_default
    get '/'
    assert_equal 'Hello World!', last_response.body
  end

  def test_with_params
    get '/meet', :name => 'Frank'
    assert_equal 'Hello Frank!', last_response.body
  end

  def test_with_rack_env
    get '/', {}, 'HTTP_USER_AGENT' => 'Songbird'
    assert_equal "You're using Songbird!", last_response.body
  end
end
```

请注意: 内置的 Sinatra::Test 模块和 Sinatra::TestHarness 类 在 0.9.2 版本已废弃。

Sinatra::Base - 中间件，程序库和模块化应用

把你的应用定义在顶层，对于微型应用这会工作得很好， 但是在构建可复用的组件时候会带来客观的不利， 比如构建Rack中间件，Rails metal，带有服务器组件的简单程序库， 或者甚至是Sinatra扩展。顶层的DSL污染了Object命名空间， 并假定了一个微型应用风格的配置 (例如, 单一的应用文件， ./public 和 ./views 目录，日志，异常细节页面，等等）。 这时应该让 Sinatra::Base 走到台前了：

```
require 'sinatra/base'

class MyApp < Sinatra::Base
  set :sessions, true
  set :foo, 'bar'

  get '/' do
    'Hello world!'
  end
end
```

Sinatra::Base子类可用的方法实际上就是通过顶层 DSL 可用的方法。 大部分顶层应用可以通过两个改变转换成Sinatra::Base组件：

- 你的文件应当引入 sinatra/base 而不是 sinatra; 否则，所有的Sinatra的 DSL 方法将会被引进到 主命名空间。
- 把你的应用的路由，错误处理，过滤器和选项放在 一个Sinatra::Base的子类中。

+Sinatra::Base+ 是一张白纸。大部分的选项默认是禁用的， 包含内置的服务器。参见 选项和配置 查看可用选项的具体细节和他们的行为。

模块化 vs. 传统的方式

与通常的认识相反，传统的方式没有任何错误。 如果它适合你的应用，你不需要转换到模块化的应用。

和模块化方式相比只有两个缺点:

- 你对每个Ruby进程只能定义一个Sinatra应用，如果你需要更多， 切换到模块化方式。

- 传统方式使用代理方法污染了 Object 。如果你打算 把你的应用封装进一个 library/gem，转换到模块化方式。

没有任何原因阻止你混合模块化和传统方式。

如果从一种转换到另一种，你需要注意settings中的 一些微小的不同:

Setting	Classic	Modular
app_file	file loading sinatra	nil
run	$0 == app_file	false
logging	true	false
method_override	true	false
inline_templates	true	false

运行一个模块化应用

有两种方式运行一个模块化应用，使用 run!来运行:

```
# my_app.rb
require 'sinatra/base'

class MyApp < Sinatra::Base
  # ... app code here ...

  # start the server if ruby file executed directly
  run! if app_file == $0
end
```

运行:

```
ruby my_app.rb
```

或者使用一个 config.ru，允许你使用任何Rack处理器:

```
# config.ru
require './my_app'
run MyApp
```

运行:

```
rackup -p 4567
```

使用config.ru运行传统方式的应用

编写你的应用:

```
# app.rb
require 'sinatra'

get '/' do
  'Hello world!'
end
```

加入相应的 config.ru:

```
require './app'
run Sinatra::Application
```

什么时候用 config.ru?

以下情况你可能需要使用 config.ru:

- 你要使用不同的 Rack 处理器部署 (Passenger, Unicorn, Heroku, ...).
- 你想使用一个或者多个 Sinatra::Base的子类.
- 你只想把Sinatra当作中间件使用，而不是端点。

你并不需要切换到config.ru仅仅因为你切换到模块化方式， 你同样不需要切换到模块化方式， 仅仅因为要运行 config.ru.

把Sinatra当成中间件来使用

不仅Sinatra有能力使用其他的Rack中间件，任何Sinatra 应用程序都可以反过来自身被当作中间件，被加在任何Rack端点前面。 这个端点可以是任何Sinatra应用，或者任何基于Rack的应用程序 (Rails/Ramaze/Camping/...)。

```
require 'sinatra/base'

class LoginScreen < Sinatra::Base
  enable :sessions

  get('/login') { haml :login }

  post('/login') do
    if params['name'] = 'admin' and params['password'] = 'admin'
      session['user_name'] = params['name']
    else
      redirect '/login'
    end
  end
end

class MyApp < Sinatra::Base
  # 在前置过滤器前运行中间件
  use LoginScreen

  before do
    unless session['user_name']
      halt "Access denied, please <a href='/login'>login</a>."
    end
  end

  get('/') { "Hello #{session['user_name']}." }
end
```

变量域和绑定

当前所在的变量域决定了哪些方法和变量是可用的。

应用/类 变量域

每个Sinatra应用相当与Sinatra::Base的一个子类。 如果你在使用顶层DSL(`require 'sinatra'`)，那么这个类就是 Sinatra::Application，或者这个类就是你显式创建的子类。 在类层面，你具有的方法类似于 `get` 或者 `before`，但是你不能访问 `request` 对象或者 `session`, 因为对于所有的请求， 只有单一的应用类。

通过 `set` 创建的选项是类层面的方法：

```
class MyApp < Sinatra::Base
  # 嘿，我在应用变量域!
  set :foo, 42
  foo # => 42

  get '/foo' do
    # 嘿，我不再处于应用变量域了!
  end
end
```

在下列情况下你将拥有应用变量域的绑定：

- 在应用类中
- 在扩展中定义的方法
- 传递给 `helpers` 的代码块
- 用作`set`值的过程/代码块

你可以访问变量域对象（就是应用类）就像这样：

- 通过传递给代码块的对象 (`configure { |c| ... }`)
- 在请求变量域中使用`settings`

请求/实例 变量域

对于每个进入的请求，一个新的应用类的实例会被创建 所有的处理器代码块在该变量域被运行。在这个变量域中， 你可以访问 `request` 和 `session` 对象，或者调用填充方法比如 `erb` 或者 `haml`。你可以在请求变量域当中通过`settings`辅助方法 访问应用变量域：

```
class MyApp < Sinatra::Base
  # 嘿，我在应用变量域!
  get '/define_route/:name' do
    # 针对 '/define_route/:name' 的请求变量域
    @value = 42

    settings.get("/#{params['name']}") do
      # 针对 "/#{params['name']}" 的请求变量域
      @value # => nil (并不是相同的请求)
    end

    "Route defined!"
  end
end
```

在以下情况将获得请求变量域：

- get/head/post/put/delete 代码块
- 前置/后置 过滤器
- 辅助方法
- 模板/视图

代理变量域

代理变量域只是把方法转送到类变量域。可是， 他并非表现得100%类似于类变量域, 因为你并不能获得类的绑定: 只有显式地标记为供代理使用的方法才是可用的， 而且你不能和类变量域共享变量/状态。(解释：你有了一个不同的 `self`)。 你可以显式地增加方法代理，通过调用 `Sinatra::Delegator.delegate :method_name`。

在以下情况将获得代理变量域：

- 顶层的绑定，如果你做过 `require "sinatra"`
- 在扩展了 `Sinatra::Delegator` mixin的对象

自己在这里看一下代码: Sinatra::Delegator mixin 已经 被包含进了主命名空间。

命令行

Sinatra 应用可以被直接运行：

```
ruby myapp.rb [-h] [-x] [-e ENVIRONMENT] [-p PORT] [-o HOST] [-s HANDLER]
```

选项是：

```
-h # help
-p # 设定端口 (默认是 4567)
-o # 设定主机名 (默认是 0.0.0.0)
-e # 设定环境 (默认是 development)
-s # 限定 rack 服务器/处理器 (默认是 thin)
-x # 打开互斥锁 (默认是 off)
```

必要条件

推荐在 Ruby 1.8.7, 1.9.2, JRuby 或者 Rubinius 上安装Sinatra。

下面的Ruby版本是官方支持的:

Ruby 1.8.6
: 不推荐在1.8.6上安装Sinatra， 但是直到Sinatra 1.3.0发布才会放弃对它的支持。 RDoc 和 CoffeScript模板不被这个Ruby版本支持。 1.8.6在它的Hash实现中包含一个内存泄漏问题， 该问题会被1.1.1版本之前的Sinatra引发。 当前版本使用性能下降的代价排除了这个问题。你需要把Rack降级到1.1.x， 因为Rack \>= 1.2不再支持1.8.6。

Ruby 1.8.7
: 1.8.7 被完全支持，但是，如果没有特别原因， 我们推荐你升级到 1.9.2 或者切换到 JRuby 或者 Rubinius.

Ruby 1.9.2
: 1.9.2 被支持而且推荐。注意 Radius 和 Markaby 模板并不和1.9兼容。不要使用 1.9.2p0, 它被已知会产生 segmentation faults.

Rubinius
: Rubinius 被官方支持 (Rubinius \>= 1.2.2)， 除了Textile模板。

JRuby
: JRuby 被官方支持 (JRuby \>= 1.5.6)。 目前未知和第三方模板库有关的问题， 但是，如果你选择了JRuby，请查看一下JRuby rack 处理器， 因为 Thin web 服务器还没有在JRuby上获得支持。

我们也会时刻关注新的Ruby版本。

下面的 Ruby 实现没有被官方支持， 但是已知可以运行 Sinatra:

- JRuby 和 Rubinius 老版本
- MacRuby
- Maglev
- IronRuby
- Ruby 1.9.0 and 1.9.1

不被官方支持的意思是，如果在不被支持的平台上有运行错误， 我们假定不是我们的问题，而是平台的问题。

Sinatra应该会运行在任何支持上述Ruby实现的操作系统。

紧追前沿

如果你喜欢使用 Sinatra 的最新鲜的代码，请放心的使用 master 分支来运行你的程序，它会非常的稳定。

```
cd myapp
git clone git://github.com/sinatra/sinatra.git
ruby -Isinatra/lib myapp.rb
```

我们也会不定期的发布预发布gems，所以你也可以运行

```
gem install sinatra --pre
```

来获得最新的特性。

通过Bundler

如果你想使用最新的Sinatra运行你的应用，通过 Bundler 是推荐的方式。

首先，安装bundler，如果你还没有安装:

```
gem install bundler
```

然后，在你的项目目录下，创建一个 Gemfile:

```
source :rubygems
gem 'sinatra', :git => "git://github.com/sinatra/sinatra.git"
```

```
# 其他的依赖关系
gem 'haml'                    # 举例，如果你想用haml
gem 'activerecord', '~> 3.0'  # 也许你还需要 ActiveRecord 3.x
```

请注意在这里你需要列出你的应用的所有依赖关系。 Sinatra的直接依赖关系 (Rack and Tilt) 将会， 自动被Bundler获取和添加。

现在你可以像这样运行你的应用:

```
bundle exec ruby myapp.rb
```

使用自己的

创建一个本地克隆并通过 `sinatra/lib` 目录运行你的应用， 通过 `$LOAD_PATH`:

```
cd myapp
git clone git://github.com/sinatra/sinatra.git
ruby -Isinatra/lib myapp.rb
```

为了在未来更新 Sinatra 源代码:

```
cd myapp/sinatra
git pull
```

全局安装

你可以自行编译 gem :

```
git clone git://github.com/sinatra/sinatra.git
cd sinatra
rake sinatra.gemspec
rake install
```

如果你以root身份安装 gems，最后一步应该是

```
sudo rake install
```

更多

- 项目主页（英文） - 更多的文档， 新闻，和其他资源的链接。
- 贡献 - 找到了一个bug？ 需要帮助？有了一个 patch?
- 问题追踪
- Twitter
- 邮件列表
- IRC: #sinatra on freenode.net
- Sinatra宝典 Cookbook教程

- Sinatra使用技巧 网友贡献的实用技巧
- 最新版本API文档；http://rubydoc.info的当前HEAD
- CI服务器