微型web框架的实现原理

Rack + DSL

oliver.chen

一个Web应用需要什么？

1. 能够响应 http 请求
2. 定义路由及响应内容
3. 尽可能少写点代码
4. ...

Ruby世界中，有个Rack

初步使用

安装 `rack`

```
gem install rack
```

app.rb 文件

```
require 'rack'

app = proc do |env|
  [200, {'Content-Type' => 'text/html'}, ['Hello, Rack demo']]
end

Rack::Server.start(app: app)
```

执行 `ruby app.rb`, 访问 http://0.0.0.0:9292/

什么是Rack?

1. 是一份 规范

`应用程序`与`应用服务器`之间的接口规范。

应用程序框架: `Rails` / `Sinatra` / `Roda` / ...

应用服务器: `WEBrick` / `Unicorn` / `Puma` / ...

2. 是一个 gem

封装了对HTTP数据的处理。

比如: `解析path` / `接入middleware` / ...

Rack返回结构

- 一个对象，可以相应 `call` 方法 （可以是 proc 或者 自定义对象）
- 接收一个 环境变量 `env` 作为参数
- 返回一个三个元素的数组
  1. HTTP code
  2. HTTP header
  3. HTTP body – – 一个响应 each 方法的对象，用来处理为body

```
app = proc do |env|
  [200, {'Content-Type' => 'text/html'}, ['Hello, Rack demo']]
end
```

另一种形式

miniweb.rb

```
module Miniweb
 class Base

  def call(env)
    [200, {'Content-Type' => 'text/html'}, ['Hello, MiniAp
  end

 end
end
```

app.rb

```
require 'rack'
require_relative 'miniweb'

app = Miniweb::Base.new

Rack::Server.start(app: app)
```

完成了第一步，响应HTTP请求，那下一步？

- 需要定义路由
- 路由对应的处理

来看看平时使用 Grape 定义路由的方式

```
desc "字段详情页"
params do
  requires :field_id, type: String, desc: "字段 ID"
end
get "/form_fields/show" do
  form_field = RccForm::Metadata::FormField.where(id: params[:field_id]).first
  raise NotFound if form_field.nil?
  render_happy(data: form_field.show_json)
end
```

我们需要的结构

```
get "/show" do
end

post "/create" do
end

put "/update" do
end

delete "/destory" do
end
```

定义路由收集的方法

miniweb.rb

```ruby
module Miniweb
  class Base
    attr_reader :routes

    def initialize
      @routes = {}
    end

    def get(path, &handler)
      route("GET", path, &handler)
    end

    def post(path, &handler)
      route("POST", path, &handler)
    end

    # verb: get/post/put/delete
    def route(verb, path, &handler)
      @routes[verb] ||= {}
      @routes[verb][path] = handler
    end

```

进行路由收集

app.rb

```ruby
require 'rack'
require_relative 'miniweb'

app = Miniweb::Base.new

app.get "/show" do
  p "show me ..."
  [200, {}, ["mini_app show ..."]]
end

app.post "/create" do
  p "create me ..."
  [200, {}, ["mini_app create ..."]]
end

p app.routes
# {
#   "GET"=>{"/show"=>#<Proc:0x00007fe01b162d00@mini_app.rb:31>},
#   "POST"=>{"/create"=>#<Proc:0x00007fe01b162be8@mini_app.rb:35>}
#}
```

call 方法里面获取请求信息

此时，去访问 http://0.0.0.0:9292/show ，还是得到旧的结果。

那是因为我们还没在 `call` 进行路由的判断。

miniweb.rb

```
module Miniweb
 class Base

  # .....

  def call(env)
    [200, {'Content-Type' => 'text/html'}, ['Hello, MiniApp demo']
  end

 end
end
```

查看协议部分 – https://github.com/rack/rack/blob/master/SPEC.rdoc

查看 env 的内容

```
{"rack.version"=>[1, 3],
 "rack.errors"=>#<IO:<STDERR>>,
 "rack.multithread"=>true,
 "rack.multiprocess"=>false,
 "rack.run_once"=>false,
 "SCRIPT_NAME"=>"",
 "QUERY_STRING"=>"",
 "SERVER_PROTOCOL"=>"HTTP/1.1",
 "SERVER_SOFTWARE"=>"puma 3.12.6 Llamas in Pajamas",
 "GATEWAY_INTERFACE"=>"CGI/1.2",
 "REQUEST_METHOD"=>"GET",
 "REQUEST_PATH"=>"/show",
 "REQUEST_URI"=>"/show",
 "HTTP_VERSION"=>"HTTP/1.1",
 "HTTP_HOST"=>"0.0.0.0:9292",
 "HTTP_USER_AGENT"=>
  "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.77 Safari/537
 "HTTP_ACCEPT"=>
  "text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-e
 "HTTP_ACCEPT_ENCODING"=>"gzip, deflate",
 "HTTP_ACCEPT_LANGUAGE"=>"zh-CN,zh;q=0.9,en;q=0.8,no;q=0.7",
 "HTTP_PURPOSE"=>"prefetch",
 ...
```

借助 rack 里的工具

初版

```ruby
# miniweb.rb
module Miniweb
  class Base
    def call(env)
      @request = Rack::Request.new(env)
      verb = @request.request_method
      requested_path = @request.path_info

      handler = @routes[verb][requested_path]

      handler.call
    end
  end
end
```

Rack 中有许多类似 `Rack::Request` 的工具可以使用。

未找到匹配路由

```ruby
# miniweb.rb
module Miniweb
  class Base
    def call(env)
      @request = Rack::Request.new(env)
      verb = @request.request_method
      requested_path = @request.path_info

      handler = @routes[verb][requested_path]

      if handler
        handler.call
      else
        [404, {}, ["url no found ..."]]
      end
    end
  end
end
```

最基本的路由匹配已经有了。 - - 【不过目前只支持等值匹配】

那么如何处理模式匹配呢?

有这么可以库: mustermann: your personal string matching expert

可以指定各种匹配的模式:

Type	Example	Compatible with
rails	/:slug(.:ext)	Ruby on Rails, Journey, HTTP Router, Hanami, Scalatra (if configured), NYNY
shell	/*.{png,jpg}	Unix Shell (bash, zsh)
sinatra	/:slug(.:ext)?	Sinatra (2.x), Padrino (>= 0.13.0), Pendragon, Angelo
...	...	...

Gemfile 文件：

```
gem 'mustermann'
```

使用方式：

```
require 'mustermann'

# 声明一个模式
pattern = Mustermann.new('api/users/:id')

# 尝试匹配
pattern === 'api/users/1' # true
pattern === 'api/posts/1' # false

# 参数解析
params = pattern.params('api/users/1')
# {"id"=>"1"}
```

这样一来，我们就暂时解决了“模式匹配”的问题~

我们在路由收集的时候，可以为每个路由生成一个“模式”，用来后面做匹配。

也就是除了原先保存的一个 `block`，现在还得加多一个 `pattern` .

等值匹配：

```
def route(verb, path, &handler)
  @routes[verb] ||= {}
  @routes[verb][path] = handler
end
```

模式匹配：

```
def route(verb, path, &handler)
  @routes[verb] ||= []

  pattern = Mustermann.new(path)
  @routes[verb] << [pattern, handler]
end
```

已收集路由：

- pattern1: `Mustermann.new('api/users/:id')`
- pattern2: `Mustermann.new('api/posts/:id')`
- pattern3: `Mustermann.new('api/projects/:id')`
- ...

那么此时请求进来 -- `api/projects/1`，如何判断该URL属于上面的哪个模式?

最简单一种就是: 遍历.

等值匹配:

```
def call(env)
  @request = Rack::Request.new(env)
  verb = @request.request_method
  requested_path = @request.path_info

  handler = @routes[verb][requested_path]

  if handler
    handler.call
  else
    [404, {}, ["url no found ..."]]
  end
end
```

模式匹配:

```
def call(env)
  @request = Rack::Request.new(env)
  verb = @request.request_method
  requested_path = @request.path_info

  # 遍历匹配
  routes = @routes[verb] || []
  handler = catch(:halt) do
    routes.each do |pattern, block|
      if pattern === requested_path
        @params = pattern.params(requested_path)
        throw :halt, block
      end
    end

    throw :halt
  end
end
```

这里有个技巧，使用了：

```
respone = catch(:halt) do
  # ...
  # ...

  throw :halt

  # ...
end
```

可以在后续不断加入代码，不管嵌套多深，只需要抛出 `throw :halt`，就可以跳回 `catch(:halt)` 的位置。

`catch ... throw ...` 是成对出现的。

使用遍历的方式，会依赖于路由定义时的顺序，且匹配性能一般。

当然还有更多路由的匹配算法：状态机？基数树？...
有兴趣的小伙伴也可以深挖下，不同语言的不同框架中是如何处理~

比如 `Rails` 的路由，功能就极其强大~

在 Swift 下实现了 Rails 风格的路由库 中提到：

> Rails 是 Web 开发最强框架绝对不是偶然的，
> Journey（tenderlove 为 Rails 设计的路由规则解析库）的原理是为路由设计了一门表达式语言（使用 RACC 定义），
> 然后构造出 AST 进行匹配，没有任何一个语言的同类库把这个问题上升到如此高度去解决。
> 这并不是炫技，通过这个方式可以实现很多其他途径很难做到的功能并且保持在匹配时的高性能，
> 比如：/articles/(/page/:page)(/sort/:sort) 这种双可选的情况，除了 Rails 没有任何一家可以支持。

来看看效果：

```
require 'rack'
require_relative 'miniweb'

get "/show_pattern/:id" do
  [200, {}, ["mini_app pattern match ..., params: #{@params}"]]
end

Rack::Server.start(app: Miniweb::Application)
```

返回 http://localhost:9292/show_pattern/1

如果此时我想在 `block` 使用各种实例变量/实例方法，比如 `@params`，能直接使用吗?

```
# 得到结果 --> params 为空。
mini_app pattern match ..., params:
```

初版：

```
def call(env)
  if handler
    handler.call
  else
    [404, {}, ["url no found ..."]]
  end
end
```

`handler.call` 的 `handler` 是在一个独立的上下文中执行。

可以尝试自行查看 `self` 。

改版：

```
def call(env)
  if handler
    instance_eval(&handler)
  else
    [404, {}, ["url no found ..."]]
  end
end
```

通过 `instance_eval` 让 `handler` 在实例内执行。

这个实例就是 `Miniweb::Application`， 也就是 `Miniweb::Base.new`，因此可以用到改实例内部的实例变量 / 实例方法，包括 `@params` 。

```
# http://localhost:9292/show_pattern/1 访问结果为：
mini_app pattern match ..., params: {"id"=>"1"}
```

让使用者更方便

现在来瞅瞅存在的问题。

app.rb

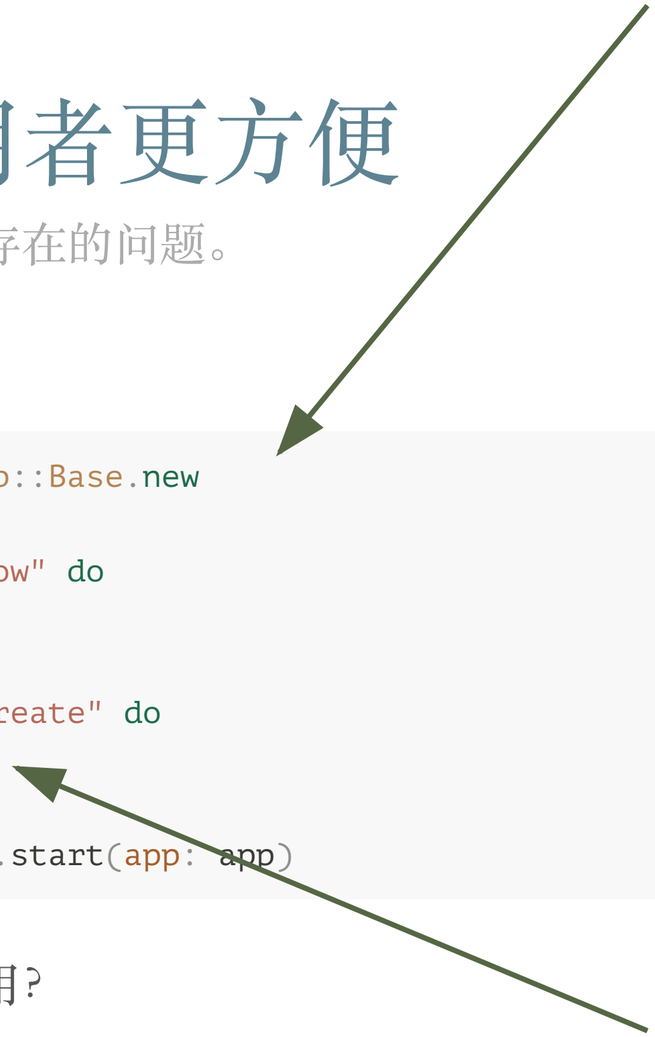
```
app = Miniweb::Base.new

app.get "/show" do
end

app.post "/create" do
end

Rack::Server.start(app: app)
```

更方便地使用?



miniweb.rb:

```
module Miniweb
  class Base
  end

  Application = Base.new    # <-- 新增
end
```

app.rb

```
app = Miniweb::Application

app.get "/show" do
end

app.post "/create" do
end

Rack::Server.start(app: app)
```

如果`get / post`能直接挂到`Miniweb::Application`上，就不用`app`了?

miniweb.rb

```ruby
module Miniweb
  module Delegator
    def self.delegate(*methods, to:)
      Array(methods).each do |method_name|
        define_method(method_name) do |*args, &block|
          to.send(method_name, *args, &block)
        end

        private method_name
      end
    end

    delegate :get, :post, to: Application
  end
end

include Miniweb::Delegator
```

在引入 `miniweb.rb` 时，就会生成对应方法 `get / post`， 会直接调用 `Application.get / Application.post`。

于是，变成了

miniweb.rb

```ruby
require 'rack'
require_relative 'miniweb'

get "/show" do
  p "show me ..."
  [200, {}, ["mini_app show ..."]]
end

post "/create" do
  p "create me ..."
  [200, {}, ["mini_app create ..."]]
end

Rack::Server.start(app: Miniweb::Application)
```

只要引入 `miniweb`，然后就可以直接使用 `get / post` 去定义路由了。

大概就是这样... 这是 miniweb gitlab 代码。

当然，这里主要讲的是思路，没有太多的细节，功能也很简单。

资料

- Rack应用及相关
- 谈谈 Rack 的协议与实现
- lets-build-a-sinatra

路由相关：

- Rails 路由 Journey 与 有限状态自动机
- 在 Swift 下实现了 Rails 风格的路由库