



Rack 是怎么选到 Puma 作为默认服务器?

RACK + PUMA

oliver.chen

1、项目背景

`Gemfile` 文件:

```
source 'https://gems.ruby-china.com/'

git_source(:github) { |repo| "https://github.com/#{repo}.g

ruby '2.6.3'

gem 'rack', '~> 2.2.3'
```

`config.ru` 文件:

```
require 'rack'

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

# run(base_app, :Port => 8090, :Host => '0.0.0.0')
run base_app
```

目录如下:

```
.
├── Gemfile
├── Gemfile.lock
└── config.ru
```

2、在终端执行：

```
→ rackup config.ru
```

```
Puma starting in single mode...
```

```
* Version 3.12.6 (ruby 2.6.3-p62), codename: Llamas in Pajamas
```

```
* Min threads: 0, max threads: 16
```

```
* Environment: development
```

```
* Listening on tcp://localhost:9292
```

```
Use Ctrl-C to stop
```

如果已经安装了 `Puma`，会默认使用。

然后就可可在浏览器访问 `http://localhost:9292/`。

3、那是怎么优先选择到 `Puma` ？

把 `rack` 的源码拉下来 ...

目录大概如下：

```
.
├── bin
│   └── rackup
├── lib
│   ├── rack
│   │   ├── handler
│   │   │   ├── cgi.rb
│   │   │   └── webrick.rb
│   │   ├── handler.rb
│   │   └── server.rb
│   └── rack.rb
```

然后从 `rack/bin/rackup` 入手。

4、为什么要从 `rack/bin/rackup` 入手呢?

这就涉及到 `rackup config.ru` 命令是怎么执行的。

1. `rackup` 为什么可以被终端识别?
2. 最终又以 `rack gem` 的哪个地方作为入口?

5、`rackup` 为什么可以被终端识别？

`unix` 可执行文件 —— `executable`

```
# unix 的可执行文件
echo "echo hello world" > my_executable
chmod +x my_executable
./my_executable

# 配合 $PATH
# echo $PATH。
# 打开一个终端，会有一个session，其中有个环境变量$PATH，会在$PATH查找可执行的文件。
mkdir $HOME/.bin
export PATH="$HOME/.bin:$PATH"
mv my_executable $HOME/.bin
my_executable
=> hello world

mkdir foo
cd foo
my_executable
=> hello world
```

6、查看 rackup 究竟是何物

执行 `which rackup`

```
/Users/Don/.rvm/gems/ruby-2.6.3/bin/rackup
```

执行 `cat /Users/Don/.rvm/gems/ruby-2.6.3/bin/rackup` :

```
#!/usr/bin/env ruby
#
# This file was generated by RubyGems.
#
# The application 'rack' is installed as part of a gem, and
# this file is here to facilitate running it.
#

require 'rubygems'

version = ">= 0.a"

# ...
load Gem.activate_bin_path('rack', 'rackup', version)
```

进入 ``irb``:

```
require 'rubygems'

version = ">= 0.a"

Gem.activate_bin_path('rack', 'rackup', version)
# => "/Users/Don/.rvm/gems/ruby-2.6.3/gems/rack-2.2.3/bin/rackup"
```

所以，最终执行的是 ``rack/bin/rackup`` 这个文件~

```
#!/usr/bin/env ruby
# frozen_string_literal: true

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

小结:

1. ``rackup`` 为什么可以被终端识别? -- ``gem`` 安装时，会生成的一个可执行文件存于 ``$PATH`` 中。
2. 最终又以 ``rack gem`` 的哪个地方作为入口? -- 最终入口是 ``rack/bin/rackup`` 这个文件。

7、深入

那现在来看看 `lib/rack/server.rb`

```
module Rack
  class Server
    def self.start
      new(options).start
    end

    def start
      # ...
      server.run(wrapped_app, **options, &block)
    end

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

      unless @_server
        @_server = Rack::Handler.default

        ...
      end
    end
  end
end
```

```

module Rack
  module Handler
    # puma 位于第一位
    SERVER_NAMES = %w(puma falcon webrick).freeze
    private_constant :SERVER_NAMES

    def self.default
      if rack_handler = ENV[RACK_HANDLER]
        self.get(rack_handler)
      else
        pick SERVER_NAMES
      end
    end
  end
end

```

```

# Select first available Rack handler given an `Array` of
# Raises `LoadError` if no handler was found.
#
#   > pick ['puma', 'webrick']
#   => Rack::Handler::WEBrick
def self.pick(server_names)
  server_names = Array(server_names)
  server_names.each do |server_name|
    begin
      return get(server_name.to_s)
    rescue LoadError, NameError
    end
  end

  raise LoadError, "Couldn't find handler for: #{server_na
end

```

也就是说：

- `Rack::Handler.default`，会去调用 `pick`。
- `pick` 根据 `puma falcon webrick` 的顺序去遍历，检查是否能够正确加载相关模块。

所以，最终确定使用服务器是 `puma` 就在 `Rack::Handler.get` 里面。

```
module Rack
  module Handler
    # 传进来的server --> puma
    def self.get(server)
      return unless server
      server = server.to_s

      # 尝试把 server (也就是puma) require 进来
      load_error = try_require('rack/handler', server)

      const_get(server, false)

      rescue NameError => name_error
        raise load_error || name_error
      end
    end
  end
end
```


来看了一眼 ``rack/handler`` 文件夹，好家伙！！！！

里面长这样：

```
├─ handler
│   ├── cgi.rb
│   └── webrick.rb
```

(⊙o⊙)...？按理说，不是应该有个 ``puma.rb`` 在里面的？

难不成 ``try_require`` 做了啥操作？来看看。

```
#   Foo # => 'foo'
#   FooBar # => 'foo_bar.rb'
#   FooBAR # => 'foobar.rb'
#   FOObar # => 'foobar.rb'
#   FOOBAR # => 'foobar.rb'
#   FooBarBaz # => 'foo_bar_baz.rb'
def self.try_require(prefix, const_name)
  file = const_name.gsub(/^[A-Z]+/) { |pre| pre.downcase }
    gsub(/[A-Z]+[A-Z]/, '_\&').downcase

  # ::File.join(prefix, file) ==> 'rack/handler/puma'
  require(::File.join(prefix, file))
  nil
rescue LoadError => error
  error
end
```

感觉也没啥特别之处。

虽然 `handler` 目录下没有 `puma.rb` 文件，

但确实是可以 `require('rack/handler/puma')`。

而 `rack` 中，确实也没有引入 `gem puma`，但也确实启用了 `puma` 作为服务器的。

所以呢，之前对 `require`，可能有点认知不足了？

添加 ``pry-bydebug`` :

```
gem "pry-bydebug", "~> 3.8.0"
```

加个断点来看看:

```
75: def self.try_require(prefix, const_name)
76:   file = const_name.gsub(/^[A-Z]+/) { |pre| pre.do
77:     gsub(/([A-Z]+[^\A-Z])/, '_\&').downcase
78:
79:     binding.pry

=> 80:   require(::File.join(prefix, file))
81:   nil
82: rescue LoadError => error
83:   error
84: end
```

来查看一些信息:

```
[22] pry(Rack::Handler)> Puma
=> Rack::Handler::Puma
```

```
[23] pry(Rack::Handler)> $ Puma
From: /Users/Don/.rvm/gems/ruby-2.6.3/gems/puma-3.12.6/lib
Module name: Rack::Handler::Puma
Number of lines: 107
```

所以, 这里 ``rack/handler/puma.rb`` 是来自于 ``puma`` 这个 ``gem`` 的。

可以看出, ``rack`` 会根据本地环境是否已经安装了 ``puma gem``, 来决定是否使用 ``puma`` 作为服务器。

在 Ruby 中，有个全局变量，`\$LOAD_PATH`，`require` 文件时，会从该变量里的路径进行匹配。

```
# irb
# => $LOAD_PATH

[/Users/Don/.rvm/rubies/ruby-2.6.3/lib/ruby/gems/2.6.0/gems/did_you_mean-1.3.0/lib",
/Users/Don/.rvm/gems/ruby-2.6.3/gems/rack-2.2.3/lib",
/Users/Don/.rvm/gems/ruby-2.6.3/gems/method_source-1.0.0/lib",
/Users/Don/.rvm/gems/ruby-2.6.3/gems/pry-0.13.1/lib",
/Users/Don/.rvm/gems/ruby-2.6.3/gems/pry-byebug-3.9.0/lib",
/Users/Don/.rvm/gems/ruby-2.6.3/gems/coderay-1.1.3/lib",
/Users/Don/.rvm/gems/ruby-2.6.3/gems/byebug-11.1.3/lib",
/Users/Don/.rvm/gems/ruby-2.6.3/extensions/x86_64-darwin-18/2.6.0/byebug-11.1.3",
/Users/Don/.rvm/rubies/ruby-2.6.3/lib/ruby/site_ruby/2.6.0",
/Users/Don/.rvm/rubies/ruby-2.6.3/lib/ruby/site_ruby/2.6.0/x86_64-darwin18",
/Users/Don/.rvm/rubies/ruby-2.6.3/lib/ruby/site_ruby",
/Users/Don/.rvm/rubies/ruby-2.6.3/lib/ruby/vendor_ruby/2.6.0",
/Users/Don/.rvm/rubies/ruby-2.6.3/lib/ruby/vendor_ruby/2.6.0/x86_64-darwin18",
/Users/Don/.rvm/rubies/ruby-2.6.3/lib/ruby/vendor_ruby",
/Users/Don/.rvm/rubies/ruby-2.6.3/lib/ruby/2.6.0",
/Users/Don/.rvm/rubies/ruby-2.6.3/lib/ruby/2.6.0/x86_64-darwin18"]
```

输入 ``$ require`` 找到 ``require`` 的来源。

```
[1] pry(Rack::Handler)> $ require

From: /Users/Don/.rvm/rubies/ruby-2.6.3/lib/ruby/2.6.0/rubygems
Owner: Kernel
Visibility: private
Signature: require(path)
Number of lines: 100

def require(path)
  # ...
rescue LoadError => load_error
  # ...
  Gem.try_activate(path))
  # ...
end
```

输入 ``$ Gem.try_activate`` 找到 ``try_activate`` 的定义。

```
def self.try_activate(path)
  # 这一步能够找到 puma gem 里面包含了 "rack/handler/puma"
  spec = Gem::Specification.find_by_path path

  # 将这个包添加进 $LOAD_PATH
  spec.activate
end
```

再来看一下 ``$LOAD_PATH`` :

```
[
  "/Users/Don/.rvm/gems/ruby-2.6.3/gems/puma-3.12.6/lib",          <----- 多了这个
  "/Users/Don/.rvm/gems/ruby-2.6.3/extensions/x86_64-darwin-18/2.6.0/puma-3.12.6", <----- 多了这个
  ...
]
```

整体下来，就是：

1. ``rackup config.ru`` 启动，终端找到 ``rackup`` 可执行文件
2. ``rackup`` 可执行文件，会加载 ``gem`` 包里 ``bin/rackup`` 对应的文件
3. ``Rack::Server.start`` 执行到 ``Rack::Handler.get``
4. ``Rack::Handler.get`` 尝试 ``require('rack/handler/puma')``
5. ``require`` 不是 ``Ruby`` 原生，是来自 ``rubygems`` 的重写
6. ``rubygems require`` 会尝试去 ``Gem`` 匹配搜索
7. 一旦匹配到，会将相关目录添加到 ``$LOAD_PATH``
8. 然后会在 ``$LOAD_PATH`` 执行匹配

8、资料

- 像 `rails/rake/sidekiq` 这些命令，为什么可以直接在终端执行？
- Rack应用及相关