5 天搞定 Ruby on Rails

企业内训纲要

第 3 天：Rails 控制器

版本： V1

口号：快速迭代，不断完善

ruby 技术交流--南方群 95824005

ruby 技术交流--北方群 101388340

ruby 技术交流--东部群 236263776

ruby 技术交流--西部群 230015785

ruby 技术交流--中部群 104131248

文档，实例地址：

https://github.com/nienwoo/ruby-learn

1. 第 3 天目标

- 掌握 controller。

2. 控制器基础

2.1. 控制器的作用

Action Controller 是 MVC 中的 C（控制器）。路由决定使用哪个控制器处理请求后，控制器负责解析请求，生成对应的请求。Action Controller 会代为处理大多数底层工作，使用易懂的约定，让整个过程清晰明了。

在大多数按照 REST 规范开发的程序中，控制器会接收请求（开发者不可见），从模型中获取数据，或把数据写入模型，再通过视图生成 HTML。如果控制器需要做其他操作，也没问题，以上只不过是控制器的主要作用。

因此，控制器可以视作模型和视图的中间人，让模型中的数据可以在视图中使用，把数据显示给用户，再把用户提交的数据保存或更新到模型中。

路由的处理细节请查阅 Rails Routing From the Outside In。

2.2. 控制器命名约定

Rails 控制器的命名习惯是，最后一个单词使用复数形式，但也是有例外，比如 ApplicationController。例如：用 ClientsController，而不是 ClientController；用 SiteAdminsController，而不是 SiteAdminController 或 SitesAdminsController。

遵守这一约定便可享用默认的路由生成器（例如 resources 等），

无需再指定 :path 或 :controller，URL 和路径的帮助方法也能保持一致性。详情参阅 Layouts & Rendering Guide。

控制器的命名习惯和模型不同，模型的名字习惯使用单数形式。

2.3. 方法和动作

控制器是一个类，继承自 ApplicationController，和其他类一样，定义了很多方法。程序接到请求时，路由决定运行哪个控制器和哪个动作，然后创建该控制器的实例，运行和动作同名的方法。

```
class ClientsController < ApplicationController
  def new
  end
end
```

例如，用户访问 /clients/new 新建客户，Rails 会创建一个 ClientsController 实例，运行 new 方法。注意，在上面这段代码中，即使 new 方法是空的也没关系，因为默认会渲染 new.html.erb 视图，除非指定执行其他操作。在 new 方法中，声明可在视图中使用的 @client 实例变量，创建一个新的 Client 实例：

```
def new
  @client = Client.new
end
```

详情参阅 Layouts & Rendering Guide。

ApplicationController 继承自 ActionController::Base 。ActionController::Base 定义了很多实用方法。本文会介绍部分方法，如果想知道定义了哪些方法，可查阅 API 文档或源码。

只有公开方法才被视为动作。所以最好减少对外可见的方法数量，例如辅助方法和过滤器方法。

3. 请求参数

在控制器的动作中，往往需要获取用户发送的数据，或其他参数。在网页程序中参数分为两类。

第一类随 URL 发送，叫做“请求参数”，即 URL 中 ? 符号后面的部分。

第二类经常成为“POST 数据”，一般来自用户填写的表单。之所以叫做“POST 数据”是因为，只能随 HTTP POST 请求发送。Rails 不区分这两种参数，在控制器中都可通过 params Hash 获取：

```
class ClientsController < ApplicationController
  # This action uses query string parameters because it gets run
  # by an HTTP GET request, but this does not make any difference
  # to the way in which the parameters are accessed. The URL for
  # this action would look like this in order to list activated
  # clients: /clients?status=activated
  def index
    if params[:status] == "activated"
```

```
    @clients = Client.activated
  else
    @clients = Client.inactivated
  end
end

# This action uses POST parameters. They are most likely coming
# from an HTML form which the user has submitted. The URL for
# this RESTful request will be "/clients", and the data will be
# sent as part of the request body.
def create
  @client = Client.new(params[:client])
  if @client.save
    redirect_to @client
  else
    # This line overrides the default rendering behavior, which
```

```
        # would have been to render the "create" view.
        render "new"
      end
    end
  end
```

3.1. Hash 和数组参数

params Hash 不局限于只能使用一维键值对，其中可以包含数组和嵌套的 Hash。要发送数组，需要在键名后加上一对空方括号（[]）：

```
GET /clients?ids[]=1&ids[]=2&ids[]=3
```

“[”和“]”这两个符号不允许出现在 URL 中，

所以上面的地址会被编码成

```
/clients?ids%5b%5d=1&ids%5b%5d=2&ids%5b%5d=3。
```

大多数情况下，无需你费心，浏览器会为你代劳编码，接收到这样的请求后，Rails 也会自动解码。如果你要手动向服务器发送这样的请求，就要留点心了。

此时，params[:ids] 的值是 ["1", "2", "3"]。注意，参数的值始终是字符串，

Rails 不会尝试转换类型。

默认情况下，基于安全考虑，参数中的 []、[nil] 和 [nil, nil, ...] 会替换成 nil。详情参阅安全指南。

要发送嵌套的 Hash 参数，需要在方括号内指定键名：

```
<form accept-charset="UTF-8" action="/clients" method="post">
  <input type="text" name="client[name]" value="Acme" />
  <input type="text" name="client[phone]" value="12345" />
  <input type="text" name="client[address][postcode]" value="12345" />
  <input type="text" name="client[address][city]" value="Carrot City" />
</form>
```

提交这个表单后，params[:client] 的值是 { "name" => "Acme", "phone" => "12345", "address" => { "postcode" => "12345", "city" => "Carrot City" } }。

注意 params[:client][:address] 是个嵌套 Hash。

注意，params Hash 其实是 ActiveSupport::HashWithIndifferentAccess 的实例，虽和普通的 Hash 一样，但键名使用 Symbol 和字符串的效果一样。

3.2. JSON 参数

开发网页服务程序时，你会发现，接收 JSON 格式的参数更容易处理。

如果请求的 Content-Type 报头是 application/json，Rails 会自动将其转换成 params Hash，按照常规的方法使用：

例如，如果发送如下的 JSON 格式内容：

{ "company": { "name": "acme", "address": "123 Carrot Street" } }

得到的是 params[:company] 就是 { "name" => "acme", "address" => "123 Carrot Street" }。

如果在初始化脚本中开启了 config.wrap_parameters 选项，或者在控制器中调用了 wrap_parameters 方法，可以放心的省去 JSON 格式参数中的根键。Rails 会以控制器名新建一个键，复制参数，将其存入这个键名下。因此，上面的参数可以写成：

{ "name": "acme", "address": "123 Carrot Street" }

假设数据传送给 CompaniesController，那么参数会存入 :company 键名下：

{ name: "acme", address: "123 Carrot Street", company: { name: "acme", address: "123 Carrot Street" } }

如果想修改默认使用的键名，或者把其他参数存入其中，请参阅 API 文档。

解析 XML 格式参数的功能现已抽出，制成了 gem，名为 actionpack-xml_parser。

3.3. 路由参数

params Hash 总有 :controller 和 :action 两个键，但获取这两个值应该使用 controller_name 和 action_name 方法。

路由中定义的参数，例如 :id，也可通过 params Hash 获取。例如，假设有个客户列表，可以列出激活和禁用的客户。

我们可以定义一个路由，捕获下面这个 URL 中的 :status 参数：

```
get '/clients/:status' => 'clients#index', foo: 'bar'
```

在这个例子中，用户访问 /clients/active 时，params[:status] 的值是 "active"。

同时，params[:foo] 的值也会被设为 "bar"，就像通过请求参数传入的一样。params[:action] 也是一样，其值为 "index"。

3.4. default_url_options

在控制器中定义名为 default_url_options 的方法，可以设置所生成 URL 中都包含的参数。

这个方法必须返回一个 Hash，其值为所需的参数值，而且键必须使用 Symbol：

```
class ApplicationController < ActionController::Base
  def default_url_options
    { locale: I18n.locale }
  end
end
```

这个方法定义的只是预设参数，可以被 url_for 方法的参数覆盖。

如果像上面的代码一样，在 ApplicationController 中定义 default_url_options，则会用于所有生成的 URL。default_url_options 也可以在具体的控制器中定义，只影响和该控制器有关的 URL。

3.5. 健壮参数

加入健壮参数功能后，Action Controller 的参数禁止在 Avtive Model 中批量赋值，除非参数在白名单中。也就是说，你要明确选择那些属性可以批量更新，避免意外把不该暴露的属性暴露了。

而且，还可以标记哪些参数是必须传入的，如果没有收到，会交由 raise/rescue 处理，返回“400 Bad Request”。

```
class PeopleController < ActionController::Base
  # This will raise an ActiveModel::ForbiddenAttributes exception
  # because it's using mass assignment without an explicit permit
  # step.
  def create
    Person.create(params[:person])
  end

  # This will pass with flying colors as long as there's a person key
  # in the parameters, otherwise it'll raise a
  # ActionController::ParameterMissing exception, which will get
  # caught by ActionController::Base and turned into that 400 Bad
  # Request reply.
  def update
    person = current_account.people.find(params[:id])
    person.update!(person_params)
    redirect_to person
  end

  private
    # Using a private method to encapsulate the permissible parameters
    # is just a good pattern since you'll be able to reuse the same
    # permit list between create and update. Also, you can specialize
    # this method with per-user checking of permissible attributes.
```

```
    def person_params
      params.require(:person).permit(:name, :age)
    end
end
```

4. 会话

4.1. CookieStore

程序中的每个用户都有一个会话（session），可以存储少量数据，在多次请求中永久存储。会话只能在控制器和视图中使用，可以通过以下几种存储机制实现：

ActionDispatch::Session::CookieStore：所有数据都存储在客户端

ActionDispatch::Session::CacheStore：数据存储在 Rails 缓存里

ActionDispatch::Session::ActiveRecordStore：使用 Active Record 把数据存储在数据库中（需要使用 activerecord-session_store gem）

ActionDispatch::Session::MemCacheStore ： 数 据 存 储 在 Memcached 集群中（这是以前的实现方式，现在请改用 CacheStore）

所有存储机制都会用到一个 cookie，存储每个会话的 ID（必须使用 cookie，因为 Rails 不允许在 URL 中传递会话 ID，这么做不安全）。

大多数存储机制都会使用这个 ID 在服务商查询会话数据，例如在数据库中查询。不过有个例外，即默认也是推荐使用的存储方式

CookieStore。CookieStore 把所有会话数据都存储在 cookie 中（如果需要，还是可以使用 ID）。CookieStore 的优点是轻量，而且在新程序中使用会话也不用额外的设置。cookie 中存储的数据会使用密令签名，以防篡改。cookie 会被加密，任何有权访问的人都无法读取其内容。（如果修改了 cookie，Rails 会拒绝使用。）

CookieStore 可以存储大约 4KB 数据，比其他几种存储机制都少很多，但一般也足够用了。不过使用哪种存储机制，都不建议在会话中存储大量数据。应该特别避免在会话中存储复杂的对象（Ruby 基本对象之外的一切对象，最常见的是模型实例），服务器可能无法在多次请求中重组数据，最终导致错误。

如果会话中没有存储重要的数据，或者不需要持久存储（例如使用 Falsh 存储消息），可以考虑使用 ActionDispatch::Session::CacheStore。这种存储机制使用程序所配置的缓存方式。CacheStore 的优点是，可以直接使用现有的缓存方式存储会话，不用额外的设置。不过缺点也很明显，会话存在时间很多，随时可能消失。

关于会话存储的更多内容请参阅安全指南

如果想使用其他的会话存储机制，可以在 config/initializers/session_store.rb 文件中设置：

Use the database for sessions instead of the cookie-based default,

which shouldn't be used to store highly confidential information

(create the session table with "rails g active_record:session_migration")

YourApp::Application.config.session_store :active_record_store

签署会话数据时，Rails 会用到会话的键（cookie 的名字），这个值可以在 config/initializers/session_store.rb 中修改：

Be sure to restart your server when you modify this file.

YourApp::Application.config.session_store :cookie_store, key: '_your_app_session'

还可以传入 :domain 键，指定可使用此 cookie 的域名：

Be sure to restart your server when you modify this file.

YourApp::Application.config.session_store :cookie_store, key:

'_your_app_session', domain: ".example.com"

Rails 为 CookieStore 提供了一个密令，用来签署会话数据。这个密令可以在 config/secrets.yml 文件中修改：

Be sure to restart your server when you modify this file.

Your secret key is used for verifying the integrity of signed cookies.

If you change this key, all old signed cookies will become invalid!

Make sure the secret is at least 30 characters and all random,

no regular words or you'll be exposed to dictionary attacks.

You can use `rake secret` to generate a secure secret key.

Make sure the secrets in this file are kept private

if you're sharing your code publicly.

development:

```
  secret_key_base: a75d...

test:
  secret_key_base: 492f...

# Do not keep production secrets in the repository,
# instead read values from the environment.
production:
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>
```

使用 CookieStore 时，如果修改了密令，之前所有的会话都会失效。

4.2. 获取会话

在控制器中，可以使用实例方法 session 获取会话。

会话是惰性加载的，如果不在动作中获取，不会自动加载。因此无需禁用会话，不获取即可。

会话中的数据以键值对的形式存储，类似 Hash：

```
class ApplicationController < ActionController::Base

  private

  # Finds the User with the ID stored in the session with the key
  # :current_user_id This is a common way to handle user login in
  # a Rails application; logging in sets the session value and
  # logging out removes it.
  def current_user
    @_current_user ||= session[:current_user_id] &&
      User.find_by(id: session[:current_user_id])
  end
end
```

要想把数据存入会话，像 Hash 一样，给键赋值即可：

```
class LoginsController < ApplicationController

  # "Create" a login, aka "log the user in"

  def create

    if user = User.authenticate(params[:username], params[:password])

      # Save the user ID in the session so it can be used in

      # subsequent requests

      session[:current_user_id] = user.id

      redirect_to root_url

    end

  end

end
```

要从会话中删除数据，把键的值设为 nil 即可：

```
class LoginsController < ApplicationController

  # "Delete" a login, aka "log the user out"
```

```
    def destroy
      # Remove the user id from the session
      @_current_user = session[:current_user_id] = nil
      redirect_to root_url
    end
  end
```

要重设整个会话，请使用 reset_session 方法。

4.3. Flash 消息

Flash 是会话的一个特殊部分，每次请求都会清空。也就是说，其中存储的数据只能在下次请求时使用，可用来传递错误消息等。

Flash 消息的获取方式和会话差不多，类似 Hash。Flash 消息是 FlashHash 实例。

下面以退出登录为例。控制器可以发送一个消息，在下一次请求时显示：

```
class LoginsController < ApplicationController
  def destroy
    session[:current_user_id] = nil
    flash[:notice] = "You have successfully logged out."
    redirect_to root_url
  end
end
```

注意，Flash 消息还可以直接在转向中设置。可以指定 :notice、:alert 或者常规的 :flash：

```
redirect_to root_url, notice: "You have successfully logged out."
redirect_to root_url, alert: "You're stuck here!"
redirect_to root_url, flash: { referral_code: 1234 }
```

上例中，destroy 动作转向程序的 root_url，然后显示 Flash 消息。注意，只有下一个动作才能处理前一个动作中设置的 Flash 消息。一般都会在程序的布局中加入显示警告或提醒 Flash 消息的代码：

{:lang="erb"} ``` <!--

<% flash.each do |name, msg| -%> <%= content_tag :div, msg, class: name %> <% end -%>

<!-- more content -->

```

如此一來，如果动作中设置了警告或提醒消息，就会出现在布局中。

Flash 不局限于警告和提醒，可以设置任何可在会话中存储的内容：

{:lang="erb"}

<% if flash[:just_signed_up] %>

  <p class="welcome">Welcome to our site!</p>

<% end %>
```

如果希望 Flash 消息保留到其他请求，可以使用 keep 方法：

```
class MainController < ApplicationController

  # Let's say this action corresponds to root_url, but you want

  # all requests here to be redirected to UsersController#index.

  # If an action sets the flash and redirects here, the values

  # would normally be lost when another redirect happens, but you

  # can use 'keep' to make it persist for another request.

  def index

    # Will persist all flash values.

    flash.keep



    # You can also use a key to keep only some kind of value.

    # flash.keep(:notice)

    redirect_to users_url
```

```
    end
end
```

4.3.1. now

默认情况下，Flash 中的内容只在下一次请求中可用，但有时希望在同一个请求中使用。例如，create 动作没有成功保存资源时，会直接渲染 new 模板，这并不是一个新请求，但却希望希望显示一个 Flash 消息。针对这种情况，可以使用 flash.now，用法和 flash 一样：

```
class ClientsController < ApplicationController
  def create
    @client = Client.new(params[:client])
    if @client.save
      # ...
    else
      flash.now[:error] = "Could not save client"
      render action: "new"
    end
```

```
    end

end
```

5. Cookies

程序可以在客户端存储少量数据（称为 cookie），在多次请求中使用，甚至可以用作会话。在 Rails 中可以使用 cookies 方法轻松获取 cookies，用法和 session 差不多，就像一个 Hash：

```
class CommentsController < ApplicationController
  def new
    # Auto-fill the commenter's name if it has been stored in a cookie
    @comment = Comment.new(author: cookies[:commenter_name])
  end

  def create
    @comment = Comment.new(params[:comment])
    if @comment.save
      flash[:notice] = "Thanks for your comment!"
      if params[:remember_name]
```

```
        # Remember the commenter's name.
        cookies[:commenter_name] = @comment.author
      else
        # Delete cookie for the commenter's name cookie, if any.
        cookies.delete(:commenter_name)
      end
      redirect_to @comment.article
    else
      render action: "new"
    end
  end
end
```

注意，删除会话中的数据是把键的值设为 nil，但要删除 cookie 中的值，要使用 cookies.delete(:key) 方法。

Rails 还提供了签名 cookie 和加密 cookie，用来存储敏感数据。

签名 cookie 会在 cookie 的值后面加上一个签名，确保值没被修改。加密 cookie 除了会签名之外，还会加密，让终端用户无法读取。详细信息请参阅 API 文档。

这两种特殊的 cookie 会序列化签名后的值，生成字符串，读取时再反序列化成 Ruby 对象。

序列化所用的方式可以指定：

```
Rails.application.config.action_dispatch.cookies_serializer = :json
```

新程序默认使用的序列化方法是 :json。为了兼容以前程序中的 cookie，如果没设定 cookies_serializer，就会使用 :marshal。

这个选项还可以设为 :hybrid，读取时，Rails 会自动返序列化使用 Marshal 序列化的 cookie，写入时使用 JSON 格式。把现有程序迁移到使用 :json 序列化方式时，这么设定非常方便。

序列化方式还可以使用其他方式，只要定义了 load 和 dump 方法即可：

```
Rails.application.config.action_dispatch.cookies_serializer =
MyCustomSerializer
```

6. 过滤器

6.1. 后置过滤器和环绕过滤器

除了前置过滤器之外，还可以在动作运行之后，或者在动作运行前后执行过滤器。

后置过滤器类似于前置过滤器，不过因为动作已经运行了，所以可以获取即将发送给客户端的响应数据。显然，后置过滤器无法阻止运行动作。

环绕过滤器会把动作拉入（yield）过滤器中，工作方式类似 Rack 中间件。

例如，网站的改动需要经过管理员预览，然后批准。可以把这些操作定义在一个事务中：

```
class ChangesController < ApplicationController

  around_action :wrap_in_transaction, only: :show
```

```
    private

    def wrap_in_transaction
      ActiveRecord::Base.transaction do
        begin
          yield
        ensure
          raise ActiveRecord::Rollback
        end
      end
    end
  end
```

注意，环绕过滤器还包含了渲染操作。在上面的例子中，视图本身是从数据库中读取出来的（例如，通过作用域（scope）），读取视图的操作在事务中完成，然后提供预览数据。

也可以不拉入动作，自己生成响应，不过这种情况不会运行动作。

6.2. 前置

过滤器（filter）是一些方法，在控制器动作运行之前、之后，或者前后运行。

过滤器会继承，如果在 ApplicationController 中定义了过滤器，那么程序的每个控制器都可使用。

前置过滤器有可能会终止请求循环。前置过滤器经常用来确保动作运行之前用户已经登录。这种过滤器的定义如下：

```
class ApplicationController < ActionController::Base
  before_action :require_login

  private

  def require_login
    unless logged_in?
      flash[:error] = "You must be logged in to access this section"
```

```
      redirect_to new_login_url # halts request cycle
    end
  end
end
```

如果用户没有登录，这个方法会在 Flash 中存储一个错误消息，然后转向登录表单页面。如果前置过滤器渲染了页面或者做了转向，动作就不会运行。如果动作上还有后置过滤器，也不会运行。

在上面的例子中，过滤器在 ApplicationController 中定义，所以程序中的所有控制器都会继承。程序中的所有页面都要求用户登录后才能访问。很显然（这样用户根本无法登录），并不是所有控制器或动作都要做这种限制。如果想跳过某个动作，可以使用 skip_before_action：

```
class LoginsController < ApplicationController
  skip_before_action :require_login, only: [:new, :create]
end
```

此时，LoginsController 的 new 动作和 create 动作就不需要用

户先登录。:only 选项的意思是只跳过这些动作。还有个 :except 选项，用法类似。定义过滤器时也可使用这些选项，指定只在选中的动作上运行。

6.3. 过滤器的其他用法

6.4. 防止请求伪造

跨站请求伪造（CSRF）是一种工具方式，A 网站的用户伪装成 B 网站的用户发送请求，在 B 站中添加、修改或删除数据，而 B 站的用户绝然不知。

防止这种攻击的第一步是，确保所有析构动作（create，update 和 destroy）只能通过 GET 之外的请求方法访问。如果遵从 REST 架构，已经完成了这一步。不过，恶意网站还是可以很轻易地发起非 GET 请求，这时就要用到其他防止跨站攻击的方法了。

我们添加一个只有自己的服务器才知道的难以猜测的令牌。如果请求中没有该令牌，就会禁止访问。

如果使用下面的代码生成一个表单：

{:lang="erb"}

```
<%= form_for @user do |f| %>
  <%= f.text_field :username %>
  <%= f.text_field :password %>
<% end %>
```

会看到 Rails 自动添加了一个隐藏字段：

```
<form accept-charset="UTF-8" action="/users/1" method="post">
<input type="hidden"
       value="67250ab105eb5ad10851c00a5621854a23af5489"
       name="authenticity_token"/>
<!-- username & password fields -->
</form>
```

所有使用表单帮助方法生成的表单，都有会添加这个令牌。如果想自己编写表单，或者基于其他原因添加令牌，可以使用 form_authenticity_token 方法。

form_authenticity_token 会生成一个有效的令牌。在 Rails 没有自动添加令牌的地方（例如 Ajax）可以使用这个方法。

安全指南一文更深入的介绍了请求伪造防范措施，还有一些开发网页程序需要知道的安全隐患。