Rails 实践

用 Rails 4.2 构建在线网店

[illegible]

[illegible]

[illegible]

> [illegible]

阅读电子版

[illegible]

[illegible] [illegible]

[illegible] [illegible]

当前版本

[illegible]

本书由来

[illegible]

[illegible]

本书约定

- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]

[illegible]

- [illegible]
- [illegible]
- [illegible]

[illegible]

[illegible] http://[illegible], [illegible] https://github.com/[illegible] [illegible]。[illegible]

[illegible]

https://github.com/[illegible]

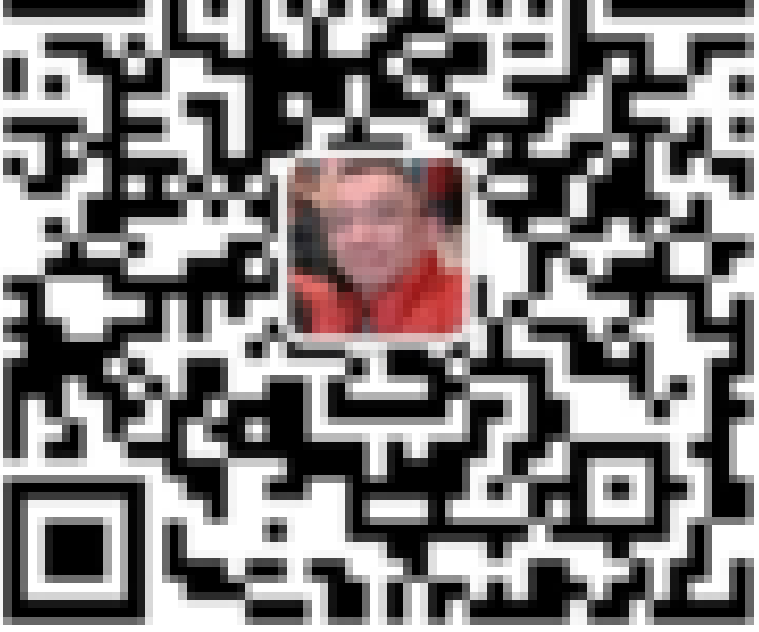
[illegible] Node.js [illegible] Mongo [illegible]、[illegible]。[illegible] Node Express、[illegible]。[illegible]。[illegible] material-ui。

[illegible]

[illegible]、[illegible]、[illegible] Node [illegible]、[illegible] [illegible]。[illegible]、[illegible]。

[illegible]、[illegible]、[illegible] [illegible] [illegible]。

[illegible]



概述

[illegible]

[illegible]

[illegible]

课程概要：

本小节我们将为大家介绍Ruby on Rails是什么样的一个框架，为什么Rails可以如此受到Ruby编程人员的欢迎，Rails的发展历史，Rails的最新版本特点，并且我们将为大家介绍Rails的开发环境（IDE）。

知识点：

1. Rails是什么？有哪些
2. Rails有哪些特点
3. 第一个例题（IDE）介绍

课程导图

Ruby是一门纯面向对象的语言，面向对象的概念是：万物皆对象，对象有属性（对象的特征）和方法（对象的行为），对象之间的交互是通过发送消息来实现的，消息的发送就是调用对象的方法，Ruby on Rails是一个Web应用程序框架，用Ruby语言编写，它遵循MVC架构模式，并鼓励使用Web标准进行开发。它包括Ruby，Rails框架以及最常用的一个数据库，MySQL的一个Web服务器和一个用于访问数据库的库。（如果您愿意，可以使用其他数据库而不是MySQL，例如PostgreSQL或SQLite，Rails支持大多数数据库，但是，一些任务的说明可能会有所不同。下面的说明假定您使用的是MySQL。）

1.1 Ruby on Rails 开发环境介绍

概要：

本小节介绍了Ruby及Rails的历史与特点，Windows版Ruby的安装，以及集成开发环境的介绍。

知识点：

1. Rails特点
2. Ruby特点
3. Rails特点
4. 开发环境

正文

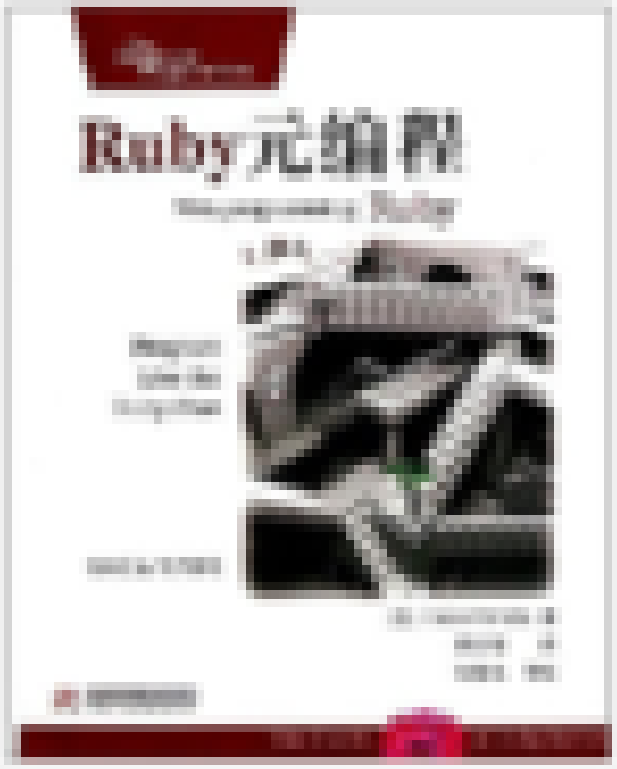
1.1.1 Ruby 简介

Ruby，是由日本的松本行弘（まつもとゆきひろ）于1993年设计并开发的一种面向对象、解释型计算机程序设计语言。


程序
世界


代码的
未来

Ruby 编程



Ruby 元编程

1.1.2 Rails 简介

[illegible]

5.1.3 Ruby 设置

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

```
# [illegible]
  [illegible]
```

[illegible]

```
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
```

[illegible]

```
[illegible]
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

1.1.4 RubyGems 管理

[illegible] RubyGems [illegible] 一个 Developers、Developers [illegible] Ruby [illegible]。[illegible] [illegible] Developers [illegible] [illegible] [illegible]。

```
[illegible]
[illegible]
[illegible]
```

[illegible] [illegible] [illegible] [illegible] [illegible]。

[illegible] [illegible] Bundler。[illegible] [illegible] [illegible]。Bundler [illegible] [illegible] [illegible] [illegible]。[illegible] http://RubyMirror.org。

[illegible] [illegible]。[illegible] [illegible] Ruby [illegible] Rails [illegible]。

[illegible] [illegible]。

1.1.5 [illegible]

Ruby [illegible] Rails [illegible]。[illegible] [illegible] [illegible]。[illegible] [illegible] Mac [illegible] [illegible]。[illegible] [illegible] [illegible] [illegible] [illegible] [illegible] [illegible]。

1.1.6 [illegible]

[illegible] [illegible] [illegible]。[illegible] [illegible] [illegible]。[illegible] [illegible] [illegible]。

[illegible]：

[illegible]

知识点：

1. [illegible]
2. [illegible]
3. [illegible]
4. [illegible]

正文

[illegible]

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

[illegible]

app 文件夹

[illegible]

config 文件夹

[illegible]

[illegible]

新建 .rspec 文件

[illegible]

[illegible]

[illegible]

3.2.2 安装 Gems

安装 Gems 时，Ruby 文档（ri）和 Ruby 参考资料（RDoc）都是可选的。[illegible]

[illegible]

```
source 'https://rubygems.org'
# source 'http://ruby.taobao.org/' # 国内推荐镜像地址
gem 'rails', '~>3.2.3' # 指定特定版本或版本范围
gem 'rails', '~>3.2'   # 指定特定版本或版本范围
gem 'my_gem', '1.0', :source => 'https://gems.example.com' # 指定特定源
gem 'nokogiri', :git => 'https://github.com/tenderlove/nokogiri.git', :branch => '1.5' # 指定分支
gem 'testing-kit', :path => '~/projects/testing-kit' # 指定本地路径
group :development, :test do
  gem 'rspec'
end
```

[illegible]

3.2.3 在 Rails 中使用 Gems

[illegible]

概要：

[illegible]

知识点：

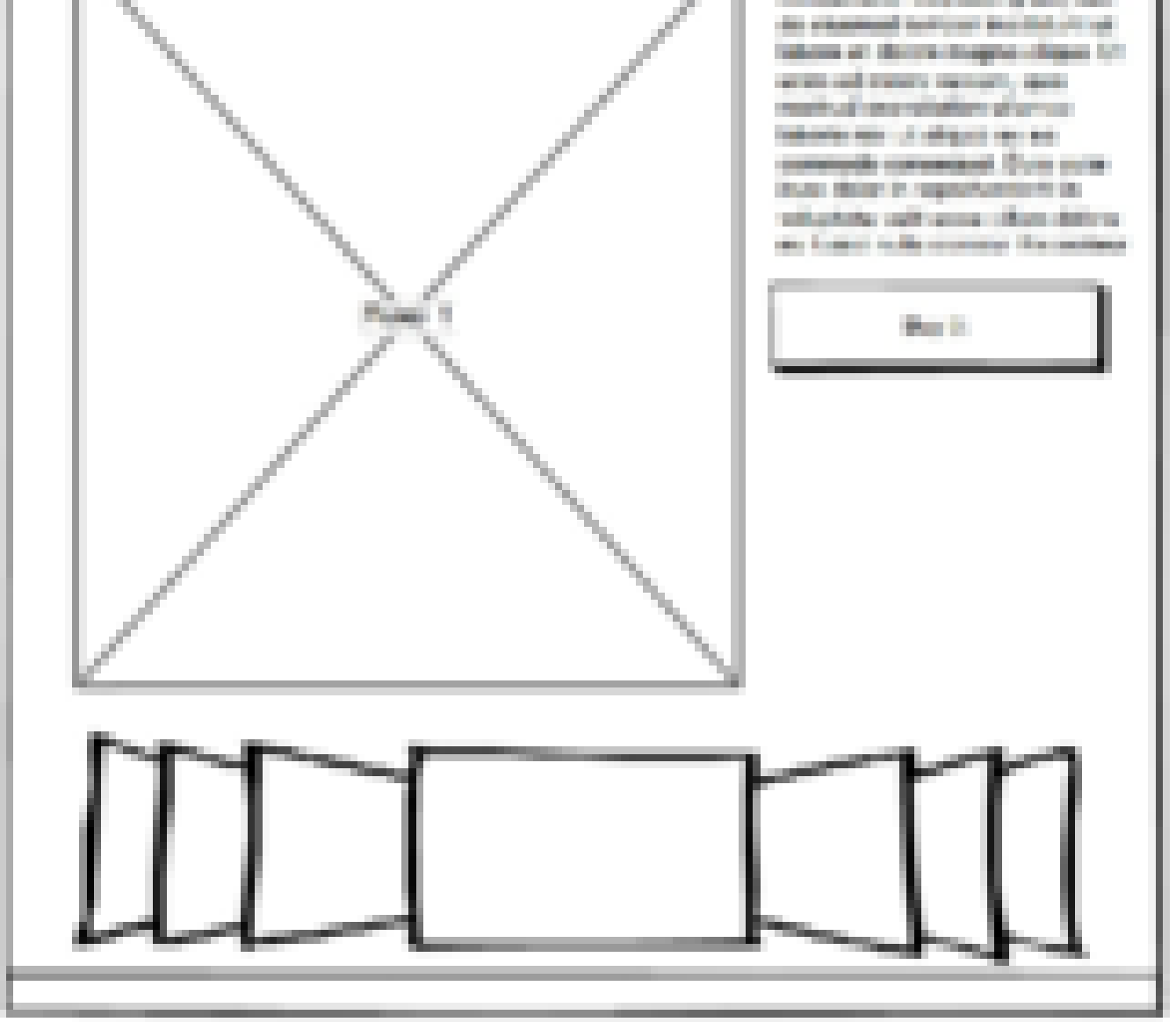
1. [illegible]
2. [illegible]
3. [illegible]

正文

1.3.1 Bootstrap

[illegible]

[illegible]



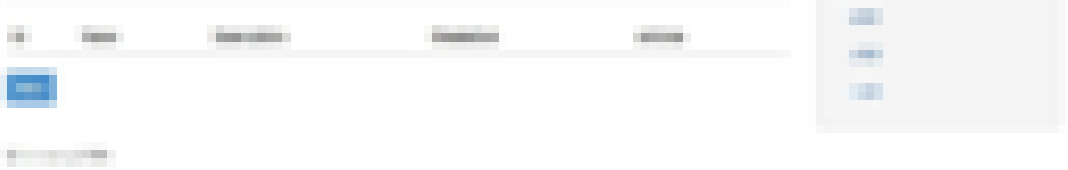
[illegible]

[illegible]

> [illegible]

[illegible]

[illegible]



Products

Home

Sidebar

© Company 2022

[illegible]

```
[illegible]
[illegible]
```

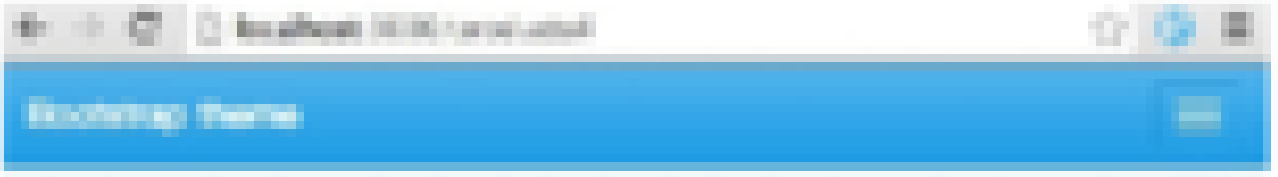
[illegible]

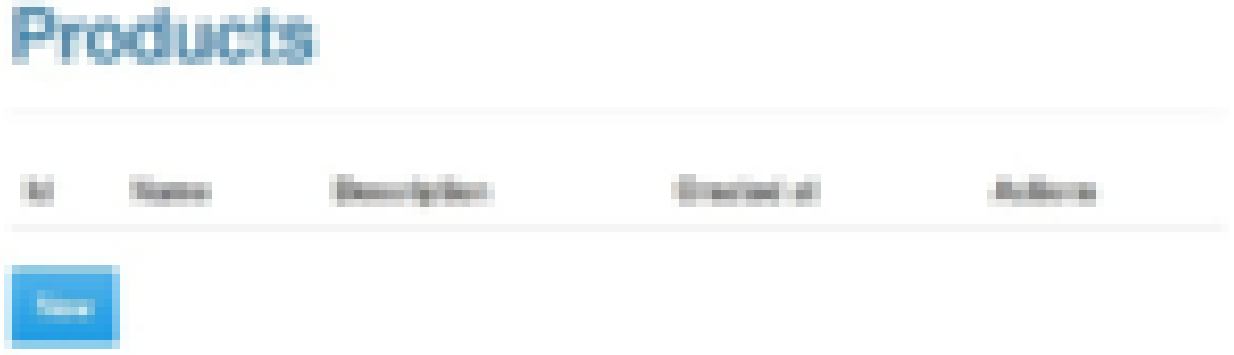
```
[illegible]
[illegible]
```

[illegible]

```
[illegible]
[illegible]
```

[illegible]


Bootstrap Themes


Products

[illegible]

[illegible]

[illegible]

[illegible] [illegible]

[illegible]

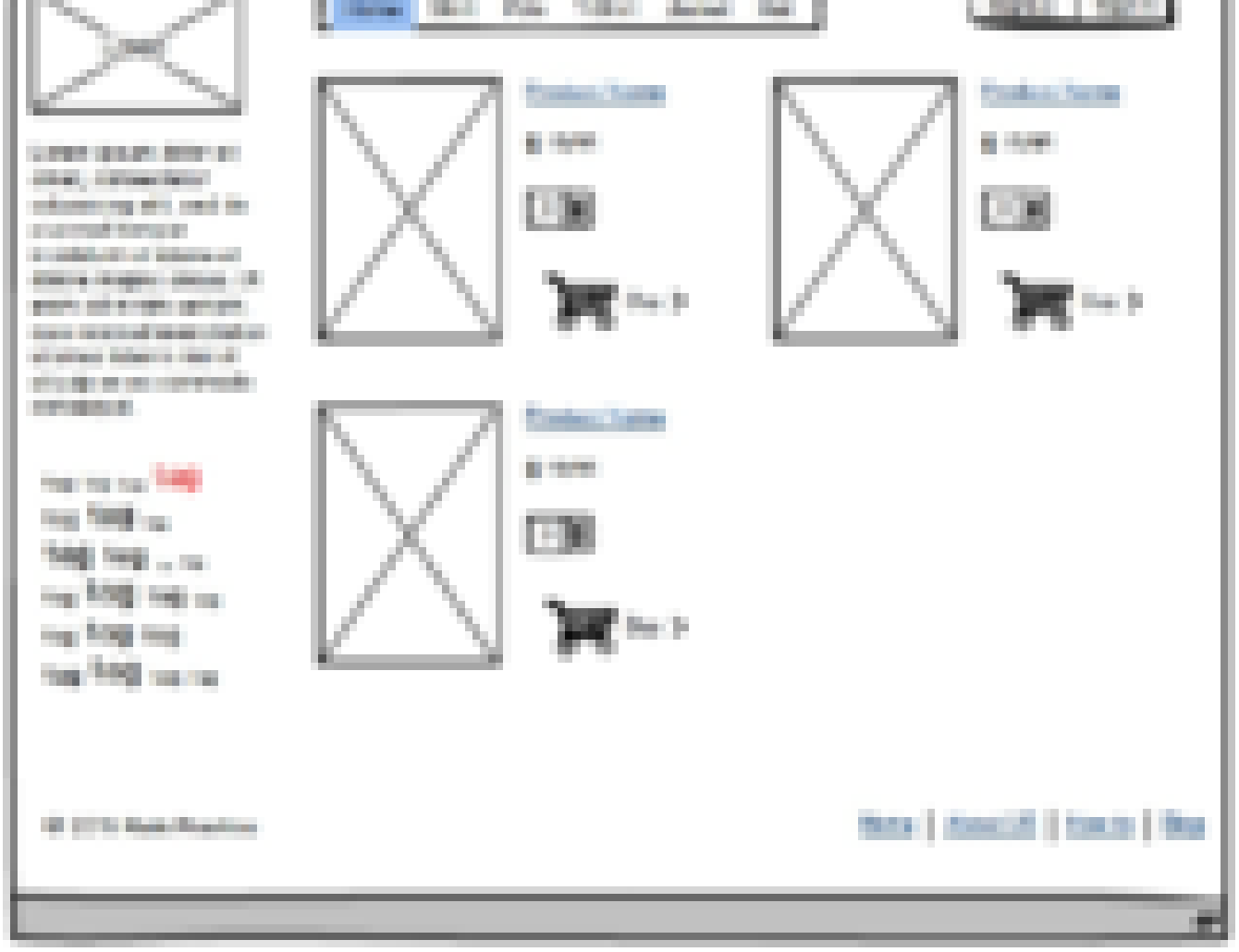
[illegible]

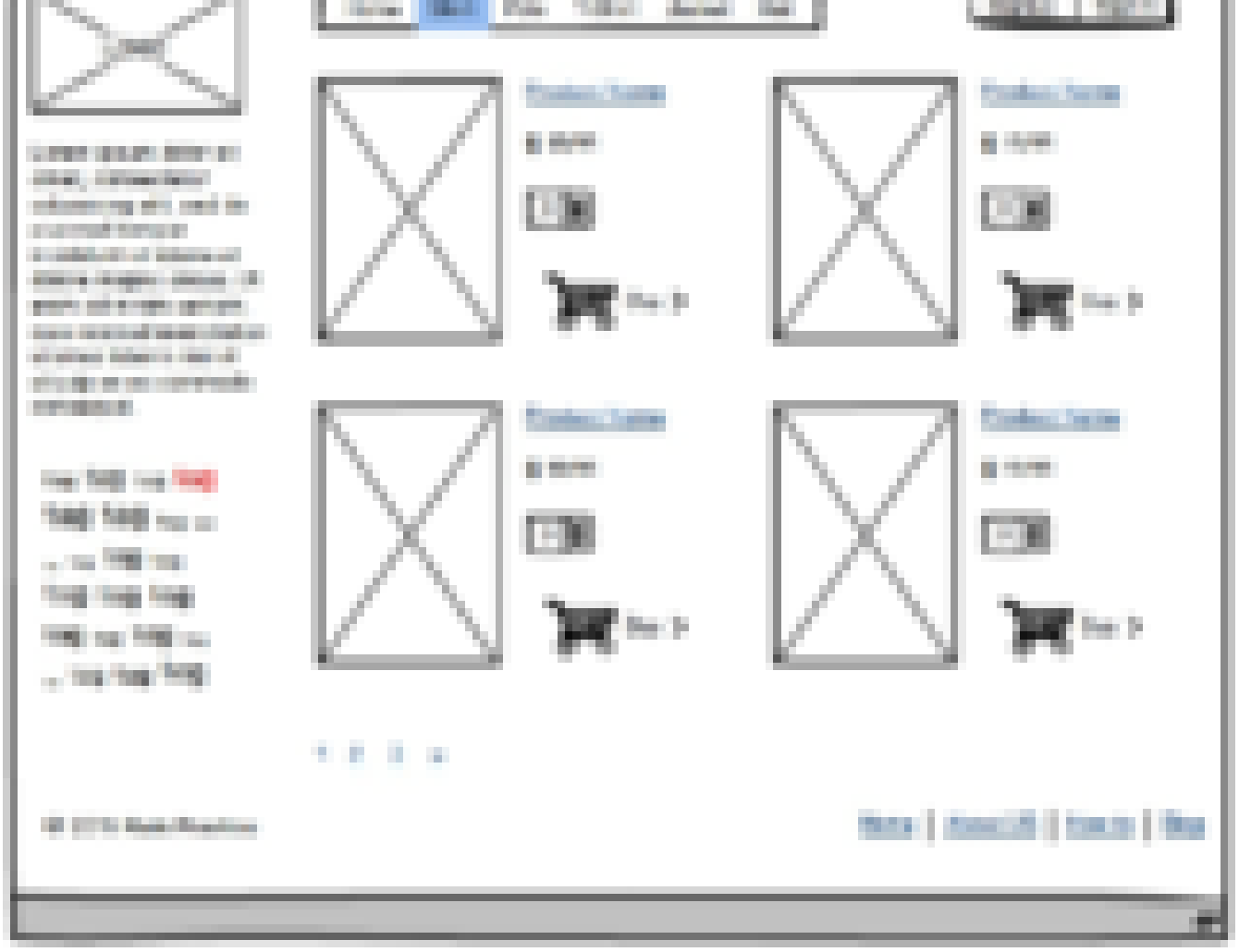
[illegible]

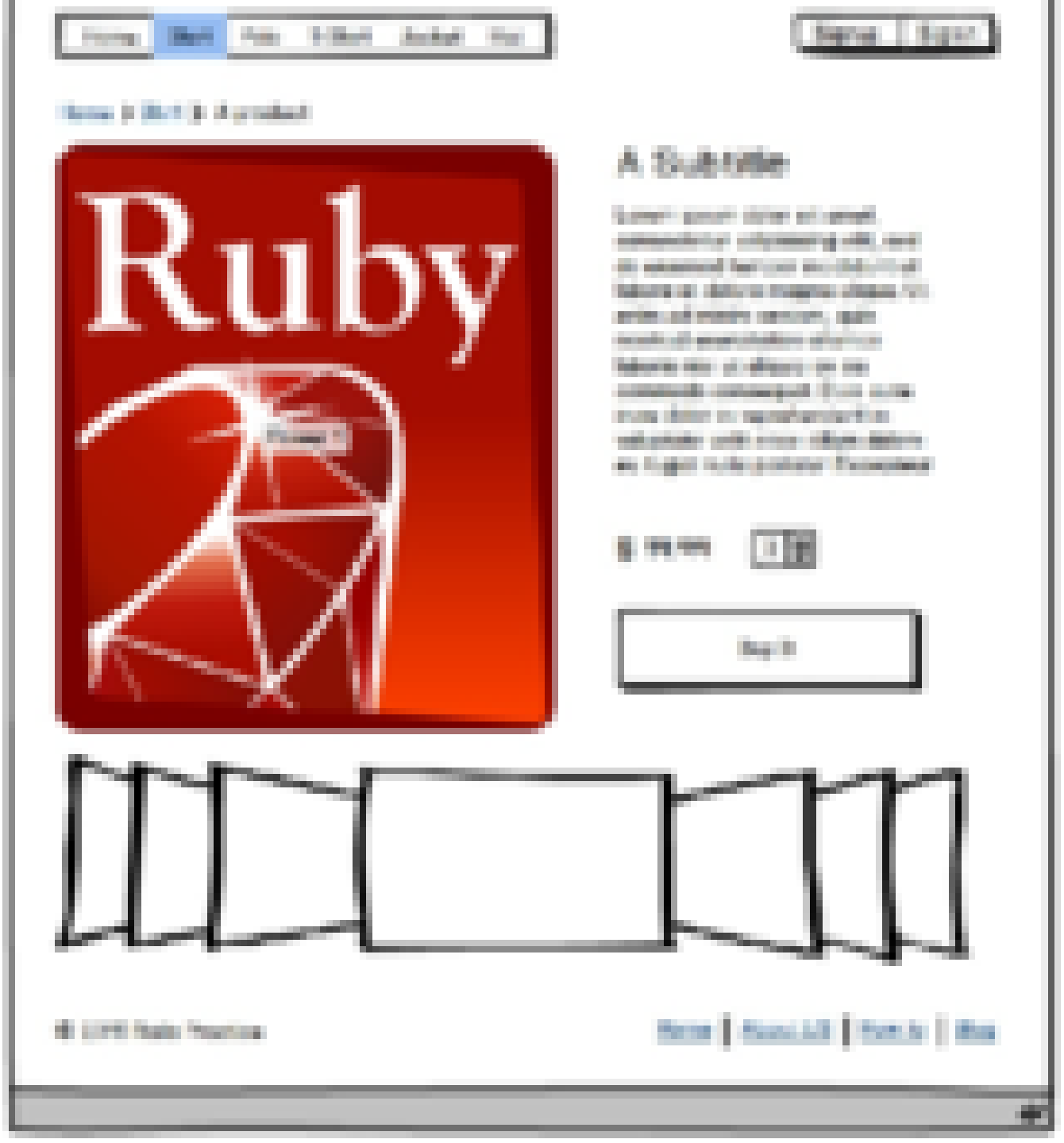
1. [illegible]
2. [illegible]
3. [illegible]

[illegible]

[illegible]






Ruby
A Subtitle

课程概要：

本次课主要为 Node 中的内容，通过 scaffold 命令生成项目，对 MVC 的整体结构有一个初步的了解，对 MVC 中的 C 控制器有更深入的理解，掌握 Node 项目中 MVC 各个部分的职责，以及如何编写 C 控制器。

知识点：

1. scaffold 命令
2. MVC
3. routes

课程导图

Node 和 MVC 之间的 web 开发 概述，通过 scaffold 命令生成项目，学习 Node 项目中的 C 控制器，学习 MVC 的 V 视图

2.1. 应用 scaffold 命令创建资源

概要：

通过对 scaffold 的学习，应用 scaffold 命令创建对应的资源，包括 app 目录，MVC 目录，static 目录等。

知识点：

1. scaffold
2. express
3. coffeescript
4. mvc
5. nopac

正文

2.1.1 hello 命令

首先，我们用 cmd 切换下一个 Node 项目，查看帮助下 scaffold 命令，切换下载后 (Windows) 文件位置：

```
F:\node_js_workspace_base_demo_develop_document>cd test
```

[illegible]

[illegible]

[illegible]

[illegible]

```
[illegible]
[illegible]
[illegible]
```

[illegible]

> [illegible]

[illegible]

```
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
```

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

```
$ composer test
$ vitest run
✓ 12 tests passed (12)
Tests  12 passed (12)
```

如下，目录结构如下，这个像一个应用程序，这个就可以看到这个 ts

```
$ tree -L 2 src/
src/
└── main.ts
```

- [ ] 安装依赖项
- [ ] 这个 Node 项目，一般 server 的主程序

这个 `scripts/createUser.ts` 的主要作用，是在一个独立的脚本里，调用 `createUser` 这个函数，创建一个新用户。

定义函数 (完成)

除了这里的 `add`，`multiply` 这个是一个函数下的一个文件 `src/models/product.test.ts` 和 `src/models/product.ts` 。

这里，Node 是一个很简单的 runtime，要怎么让 TS 可以跑？这个，这个项目 这个是用 Mocha。

接口与泛型 (Generic)

```
const { data: response, error } =
  await client
await
```

这里 npm 和 pnpm

```
$ rails generate --help
Usage:
  rails
Usage:
  bin/rails
Usage:
  bin/rails
```

这个 rails 和 Model 和 Controller 的这个生成

```
rails generate --help
rails generate controller
```

概要

[illegible]

知识点

1. [illegible]
2. [illegible]

正文

这是什么呢 REST

REST，Representational State Transfer，[illegible]。[illegible]。[illegible]。[illegible]。[illegible]。[illegible]。[illegible]。

Node [illegible]。[illegible]。

[illegible]：

1. [illegible]（URI），[illegible]。
2. [illegible]：[illegible]。
3. [illegible]：[illegible]。

[illegible]。

[illegible]。

这是怎么回事呢，[illegible]

[illegible]：

```
[illegible]
  [illegible]
```

[illegible]：

[illegible]

[illegible]

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

概要 #

本次介绍 Gin 框架中的路由相关操作，包括 GET、POST、PUT、DELETE 等常用请求类型，以及如何获取请求参数。

知识点 #

1. restful 风格
2. 路由 (routes)
3. controller
4. 请求参数
5. 响应
6. 中间件

正文 #

2.3.1 定义路由（routes）

这一节，我们来了解 Gin 框架 routes 相关的知识点。我们知道 Gin 框架中采用的路由库是基于 httprouter 做的，官方文档推荐使用 restful 风格的 API。

我们新建 routes 目录，并建一个 routes.go：

```
touch g routers/routes routes routes.go routes.routes
```

在这个文件中我们要定义一个 controller 层，特别注意的是：

- g 项目 generate 的代码，会覆盖掉我们自己写的代码，
- controller：这个层是专门处理业务的一个 controller，我们将其放在 routes 层，
- router 层：controller 的业务层，
- routes：路由层的下载层，将 controller 的处理业务，路由注册到 Gin 框架中，并对 routes 负责。

如下，我们新建这个文件，并添加一些内容，如下：routes，将中间件和我们的 controller 下的 router 层：

```
go get "github.com/..."
go get "github.com/..."
go get "github.com/..."
go get "github.com/..."
```

在 Gin 中，Group 可以为 Gin 的 Engine 结构体创建 routes 组，我们将 Group 的 Engine 结构体，创建一个 Group 的 routes 组，这样我们就可以将 routes 分组管理了。

Prefix	Verb	URI Pattern	Controller#Action
home_contact	GET	/home/contact(.:format)	home#contact
	GET	/	home#index
welcome	GET	/welcome(.:format)	home#welcome
root	GET	/	home#index

最小限の設定で動作確認ができるので、まずはこれで起動してみましょう。

```
get 'Prefix', to: 'home#index'
delete 'Prefix', to: 'home#index'
patch 'Prefix', to: 'home#index'
```

routes 単体の役割は理解できましたが、実際に（例 http:// の後）、どこまでを一つの単位

```
get 'controller/action/:id' => 'controller#action'
```

とすれば http://localhost:3000/controller/home/aaa にアクセスすると、 home コントローラの

- controller 部分に home
- action 部分に aaa
- id 部分に index

が入ります。

```
Started GET "/controller/home/aaa/index" for ::1 at 2015-01-01 17:10:20 +0900
Processing by Controller::HomeController#aaa as HTML
  Parameters: {"id"=>"index"}
```

Prefix 部分は便利な名前付けの機能ですが、ここでは controller 名で

名前一つ目は、要するに最初 (root 以外、名前は自由) get 後ろ側 (左から順番) match 以外

```
match 'controller/:action/:id', via: [:get, :post]
```

最後、（左から） (development) の場合、最初 routes 側の解析結果を（左から）

2.3.1.1 デフォルト routes.rb

[illegible]

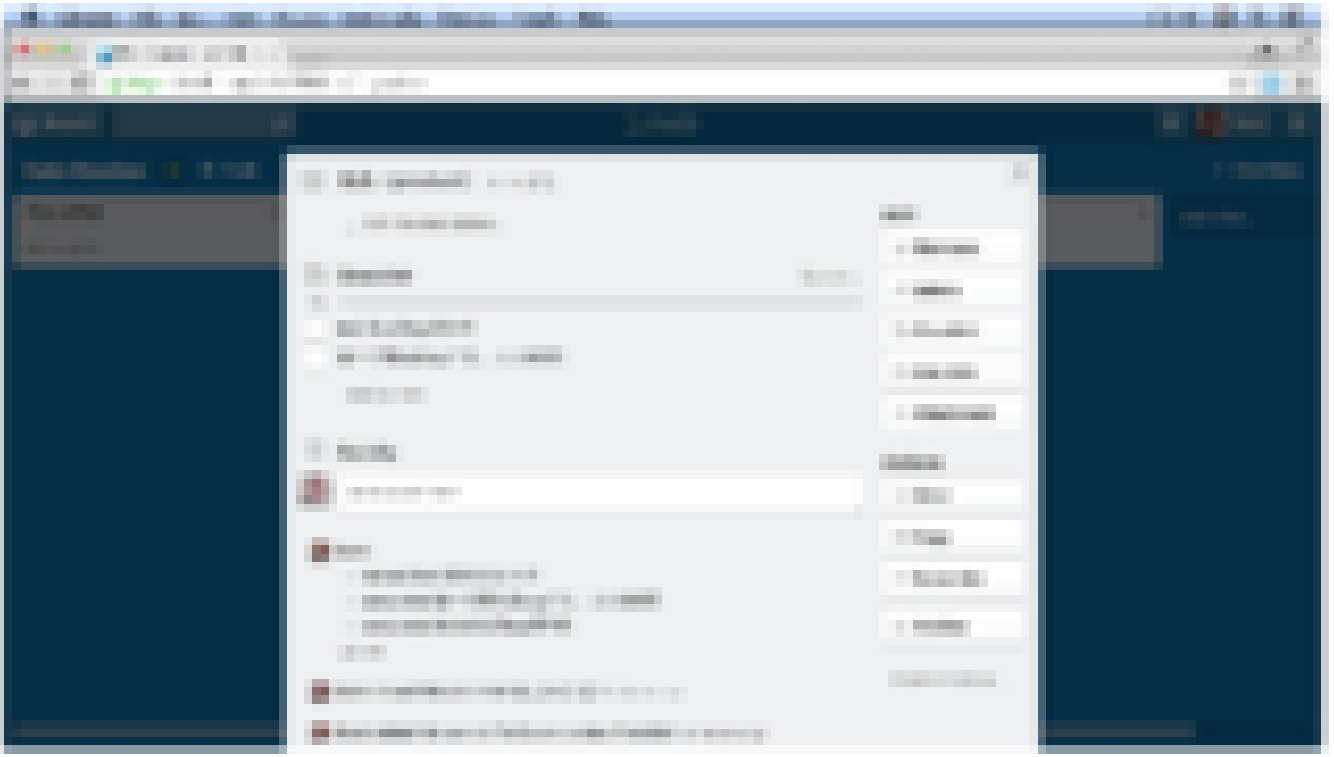
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]

[illegible]

[illegible]

[illegible]

[illegible]



[illegible]

[illegible]

[illegible] 生成的新 [illegible] 在 settings 和 profile 之间 controller 和 view 下 [illegible] routes 中 [illegible]

2.3.5.3 生成方法

microposts 生成 [illegible] 除了与 user 关联，还新增了与 content 属性相关的 [illegible] 和 [illegible] 一样的值：

```
$ rails generate model Micropost content:string user_id:integer
      invoke  active_record
      create    db/migrate/20140708_create_microposts.rb
      create    app/models/micropost.rb
      invoke    test_unit
      create      test/models/micropost_test.rb
      create      test/fixtures/microposts.yml
```

与 user 类似 [illegible]，migration 文件中 [illegible] 和 [illegible]，最后 [illegible] 和 [illegible] 的 [illegible]（[illegible] 默认情况下 [illegible]）。[illegible] 和 [illegible] 一样 [illegible] 和 [illegible]，[illegible] 和 [illegible] 一样 [illegible]。

[illegible] 和 [illegible]。

2.3.5.4 指定模块的 gem 依赖

[illegible]，[illegible] 的 [illegible]，Rails 会每一个 .yml 。

键名	说明
products_path	"/products"
products_url	"http://localhost:3000/products"

.yaml 和 .yml 两 routes 的区别没有，[illegible] 一般用 .rb 。

2.3.5.5 数据库迁移（migration）

[illegible] 与 [illegible] 的 [illegible]，每一个 [illegible] 都会有对应的 up 部分，[illegible] 一个 [illegible] (products) 会有 5 个 [illegible] (name)，[illegible] 一个 [illegible] 的 [illegible]，[illegible] 3 [illegible]，[illegible] [illegible] 的 [illegible]，[illegible] 一个用 [illegible] 的 [illegible]。

[illegible] 的 [illegible] 在 models 下？

```
class Product < ActiveRecord::Base
  validates :name, :description, :image_url, presence: true
  validates :price, numericality: {greater_than_or_equal_to: 0.01}
end
```

[illegible] rails console，[illegible] 一个 [illegible] (product) 下，[illegible] 下 [illegible] name 。

Models	Verbs	URI Helpers	Controller#Actions
products	GET	products_path products_url	products#index

[illegible]

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

[illegible]

[illegible]

[illegible]

2.3.3 [illegible]

[illegible]

[illegible]

```
[illegible]
```

[illegible]

[illegible]

```
[illegible]
```

[illegible]

[illegible]

```
[illegible]
```

[illegible]

[illegible]

```
[illegible]
```

[illegible]

```
app/
```

```
resources/views/posts/index.blade.php
resources/views/posts/show.blade.php
```

resources 目录用于放 视图、语言包等非 PHP 类文件资源。

Rails 的 app/views/posts/index.html.erb 和 Laravel 的 resources/views/posts/index.blade.php 在这里位置一致。

2.3.5 路由约定目录

web 与 api 区别

处理网页请求的路由，默认会 自动 启用 会话状态 和 CSRF 防护。

```
php artisan route:list --path=api
```

注册后，请求 /api/ 前缀 的 controller 中，会默认使用 api 中间件组。可以在 app/Providers/RouteServiceProvider.php 里修改 这个前缀，例如改成 /v1。

路由文件 routes/api.php 中，默认 不 启用 会话状态 和 CSRF 保护，适合 无状态 的接口。

```
php artisan route:list
php artisan route:list --name=posts
```

这个 命令 在 Rails 和 Laravel 中都 非常 实用，可以 快速 检查 路由 是否 注册成功。

下面我们 从 这个 入口 开始 学习。

控制器 Controller

这个 Rails 中 被称为 一个 非 常重要的概念。

```
class PostsController < ApplicationController
  def index
    @posts = Post.all
  end
end
```

这里 index 方法 将 所有 文章 赋值 给了 @posts 这个 实例变量，视图中 就可以 直接 访问 它。这个 过程 叫做 控制器 向 视图 传递 数据。

控制器的命名规则

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

[illegible]

课程概要：

本小节将为Node.js的（Core）、内置全局对象和Express框架的（Helper）、路由参数和（Form）、Ajax请求处理和上传文件的基础知识进行讲解，为后续项目开发打下基础。

知识点：

1. 全局
2. 全局方法
3. 核心
4. Ajax
5. 路由控制

课程背景

在学（Core）模块时，我们为Node.js提供了全局方法和全局对象。在浏览器上，全局对象（window）下一些对象在Node.js中也存在，比如console（控制台）、setTimeout（定时器）。

3.1. 布局和辅助方法

概要：

本小节将为Node.js的（Core）模块的基本使用，除了内置的核心方法（Helper）以及如何使用第三方。

知识点：

1. 布局（layout）
2. 全局方法（helper）
3. 模板继承（partial）

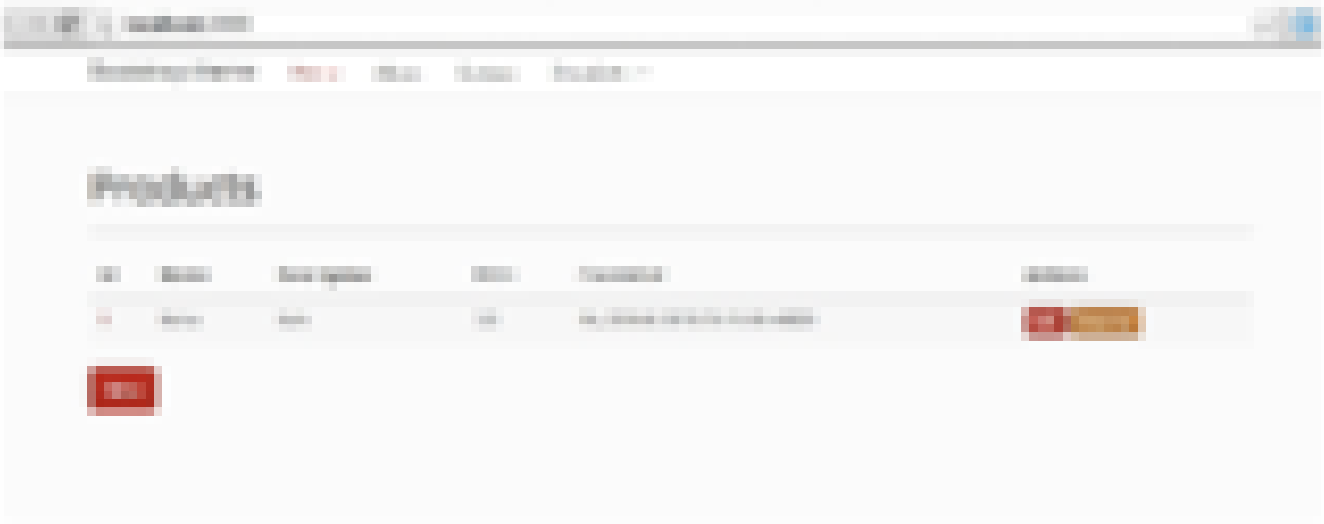
正文

3.1.1 布局（layout）

在编写Node.js的项目时，我们会为Node.js的（Core）模块提供很多的方法，在项目中Node.js下一个html（模板）来一个布局模板的方法，这个方法一个[布局文件]。

> Node.js的一个核心模块，主要用于处理（模板）。

[illegible]


Products

[illegible]

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

[illegible]

```
  layout "admin"
end
```

假如你在 admin 目录 module 中，在为 admin 建立一个新控制 controller，只要将 admin 的 controller 建立好后，在 routes 里面新的设定为：

```
class Admin::ProductsController < ApplicationController
  layout "admin"
end

class Admin::ProductsController < Admin::BaseController
end

class Admin::CommentsController < Admin::BaseController
end
```

模板二：为指定某个特殊操作，指定使用特殊的

比如，指定使用 admin 目录下的 special_alert 模板作为 Product 的显示：

```
def index
  @products = Product.all
  render :layout => "special_alert"
end
```

模板三：不用布局

```
def show
  render :layout => false
end
```

3.1.2 常用的 ERB 方法（helper）

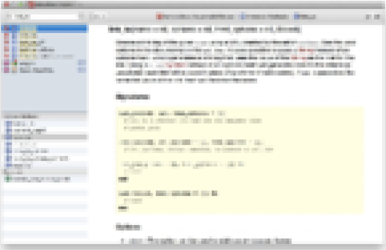
上一个，我们在 view 里面写了 ERB 的方法，这里我们来介绍下 Layout and Rendering in Rails 里面的 helper。

link_to

作为 web 应用框架的 Rails 有许多 helper 用于协助 HTML 的生成。这种协助 helper 的形式一般为方法。

我们通常在视图 view 里面一个，像 这样 的标签 tag。

```
<%= link_to "我的博客", root_path, :class => "navbar-brand" %>
```



image_tag

auto_discovery_link_tag

```
    end
end
```

```
Make it extend ActionController::Base instead.
```

```
app/controllers/users_controller.rb:2:in `<class:UsersController>': uninitialized constant ApplicationControlle
r (NameError)
Did you mean?  ApplicationHelper
               ActionController
               ActionDispatch
from app/controllers/users_controller.rb:1:in `<main>'
```

```
class UsersController < ApplicationController
  def index
    @users = User.all
    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @users }
    end
  end

  def show
    @user = User.find(params[:id])
    respond_to do |format|
      format.html # show.html.erb
      format.xml  { render :xml => @user }
    end
  end

  def new
    @user = User.new
    respond_to do |format|
      format.html # new.html.erb
      format.xml  { render :xml => @user }
    end
  end

  def edit
    @user = User.find(params[:id])
  end

  def create
    @user = User.new(params[:user])
    respond_to do |format|
      if @user.save
        flash[:notice] = 'User was successfully created.'
        format.html { redirect_to(@user) }
        format.xml  { render :xml => @user, :status => :created, :location => @user }
      else
        format.html { render :action => "new" }
        format.xml  { render :xml => @user.errors, :status => :unprocessable_entity }
      end
    end
  end

  def update
    @user = User.find(params[:id])
    respond_to do |format|
      if @user.update_attributes(params[:user])
        flash[:notice] = 'User was successfully updated.'
        format.html { redirect_to(@user) }
        format.xml  { head :ok }
      else
        format.html { render :action => "edit" }
        format.xml  { render :xml => @user.errors, :status => :unprocessable_entity }
      end
    end
  end

  def destroy
    @user = User.find(params[:id])
    @user.destroy
    respond_to do |format|
      format.html { redirect_to(users_url) }
      format.xml  { head :ok }
    end
  end
end
```

重写这个 `app/controllers/users_controller.rb` 使 `app/controllers/users_spec.rb` 的所有 examples 都通过。

这个规范了 `UsersController` 这个类的很多特性，它应该从 Rails 类型 `ActionController::Base` 这个类 (而不是 `ApplicationController`)，并以复数 `UsersController` 这种规则命名。`index` 和 `show` 都应该了这一个包含属性 `:name` 的 `User` 类型 `@users` 和 `@user`，使得这些对象能够 `xml` 格式化。

在 `create` 动作中你未来，你会使用 `notice` 这个 `flash` 类型，它会使用 `ActionController` 这个类型的对象生成 `json` 格式数据，这会在会话期间存储 `notice` 这个消息，这个消息 `notice` 会下面一节。

Rails 2.0 之前的版本使用 Rails 中 `:id` 的参数的区别，这个 `:id` 一个在两种应用模式之间的区别。

快速，使用 Rspec 查找。

stylesheet_link_tag

```
<%= stylesheet_link_tag "scaffold", :media => "all" %>
</head>
```

这个方法的作用是创建对应的 `head` 的元素，这里会为 `media` 生成 `"all"` 这个方法，`media` 这个属性的值是一个 `:media` 这个值，这个属性的值是可以是 `"all"` 这个值，这个属性的值是可以是 `"all"` 这个值。

javascript_include_tag

3.1.3 局部模板（partial）

> Q&A: Express Response 与 Koa Context 的区别？

在下一章 Koa 中，我们将会使用模板引擎来渲染 HTML，通过将 HTML 拆分成多个模板（partial），可以减少模板
app/views/products/index.html.ejs

```
<% products.each do |product| %>
  <%= render partial: "product", locals: { product: product } %>
<% end %>
```

这里的 `render` 方法使用了 `partial` 指定了模板名称，`locals` 则将变量 `product` 传递给下一个模板，即 `products/_product.html.ejs`，这个文件名必须以 `_` 开头。

下一个模板的写法：

```
<%= number_to_currency @product %>
```

可以，现在模板中已经不再关心数据（`locals`），也不再关心模板（路径），这样，控制器可以把 `render` 的行为完全封装起来。

接下来我们开始 Koa 2 **重构渲染模板**，首先安装依赖 `koa-views`，第一个 `koa-views` 是 `koa-views.init.ejs`：

```
$ npm install koa-views --save
```

然后编写初始化：

```
app.use(views('views', {
    map: {
        html: 'ejs'
    }
}));
```

这样就将 `views` 目录下的 `*.html` 文件作为 `ejs` 模板处理。

接下来，我们编写 `controller` 来渲染 HTML 页面：

```
router.get('/', async (ctx, next) => {
    ctx.state = {
        products: products
    };
    await ctx.render('index');
});
```

这里我们先设置 `ctx.state` 绑定了一个 `products` 变量，然后调用 `ctx.render()` 渲染 `index.html.ejs`：

```
<% products.each do |product| %>
  <%= render partial: "product", locals: { product: product } %>
<% end %>
```

这里的 `render` 调用 `partial` 时，传入的 `locals` 是一个 `{ product: product }` 的 JavaScript 对象，`product` 变量将被模板 `_product.html.ejs` 使用：

```
<%= product.name %> - <%= product.price %>
```

最后，我们编写 `app.js` 入口文件，把所有东西串起来：

```
const
    Koa = require('koa'),
    views = require('koa-views'),
    controller = require('./controller');

const app = new Koa();

app.use(views('views', {
    map: {
        html: 'ejs'
    }
}));

app.use(controller());

app.listen(3000);
console.log('app started at port 3000...');
```

现在，我们的 MVC 已经初具雏形，`app.js` 负责创建 `koa` 实例，加载 `koa-views` 和 `controller`，`controller` 负责处理 URL 映射，而 `views` 负责渲染模板。

3.1.4 小结

概述

本节对 JVM 的 Stack 区的基本概念（Frame）以及原理、运行机制和实现方法进行详细的介绍，并以 C 语言为例进行说明。

知识点

1. 简介
2. 简介 Stack 的基本概念（Concept）
3. 简介 Stack 的应用场景（Scenario）
4. 总结和展望

正文

3.2.1 简单的 Stack（Frame）

简单的 Stack 的设计非常简单，主要由两个元素构成：Stack 和 Frame。Stack 可以看作一个 Frame 的容器，它负责管理 Frame 的创建、销毁和访问。每个 Frame 都对应一个函数调用，它包含了函数的局部变量、参数和返回地址。

下面是一个简单的 Stack 的实现：

```
Stack* create_stack(int capacity) {
    Stack* stack = (Stack*)malloc(sizeof(Stack));
    stack->frames = (Frame*)malloc(capacity * sizeof(Frame));
    stack->top = -1;
    stack->capacity = capacity;
    return stack;
}
```

这个函数创建了一个新的 Stack，并为它分配了一定数量的 Frame 的空间。这个数量由参数 capacity 指定。函数返回这个新创建的 Stack。

下面是一个简单的 Frame 的实现：

```
Frame* create_frame(int num_locals) {
    Frame* frame = (Frame*)malloc(sizeof(Frame));
    frame->locals = (int*)malloc(num_locals * sizeof(int));
    frame->num_locals = num_locals;
    return frame;
}
```

这个函数创建了一个新的 Frame，并为它的局部变量分配了空间。这个数量由参数 num_locals 指定。函数返回这个新创建的 Frame。

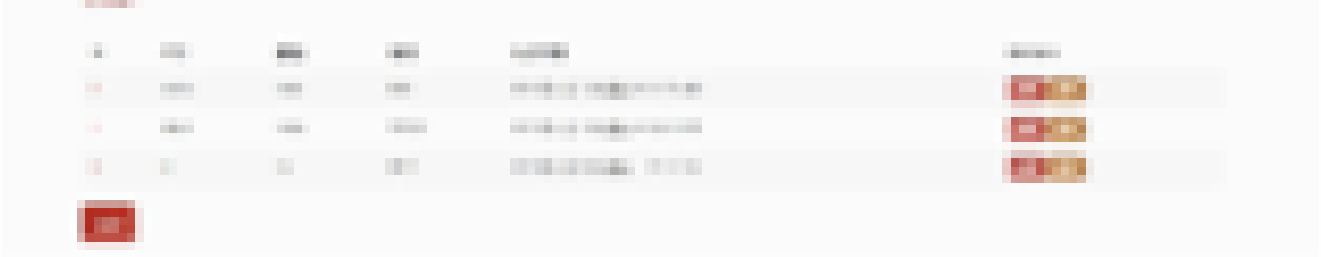
现在我们已经有了创建 Stack 和 Frame 的函数，我们还需要一些函数来操作它们。例如，我们需要一个函数来将一个 Frame 压入 Stack，以及一个函数来将一个 Frame 从 Stack 中弹出。这些函数的实现如下：

```
void push_frame(Stack* stack, Frame* frame) {
    stack->top++;
    stack->frames[stack->top] = frame;
}

Frame* pop_frame(Stack* stack) {
    Frame* frame = stack->frames[stack->top];
    stack->top--;
    return frame;
}
```

这两个函数非常简单，它们只是简单地将 Frame 压入或弹出 Stack，并更新 Stack 的 top 指针。

现在我们已经有了一个简单的 Stack 的实现，我们可以用它来模拟函数调用。例如，我们可以创建一个 Stack，然后将一个 Frame 压入 Stack，模拟函数调用；然后将这个 Frame 从 Stack 中弹出，模拟函数返回。



在 config/routes.rb 中，使用 namespace 方法定义 admin 命名空间，并在其中嵌套 resources :articles，这样就为文章资源创建了一套后台管理所需的路由。

接着，在 app/controllers/admin 下创建 articles_controller.rb 文件，在其中为 admin::ArticlesController 控制器定义 index 动作：

```
class Admin::ArticlesController < Admin::BaseController
  def index
    @articles = Article.all
  end
end
```

为了访问这个动作，我们还需要创建对应的视图模板。在 app/views/admin/articles 目录下创建 index.html.erb 文件，并添加如下内容：

```
<h1>Listing articles</h1>

<table>
  <tr>
    <th>Title</th>
    <th>Text</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>

<% @articles.each do |article| %>
  <tr>
    <td><%= article.title %></td>
    <td><%= article.text %></td>
    <td><%= link_to 'Show', [:admin, article] %></td>
    <td><%= link_to 'Edit', edit_admin_article_path(article) %></td>
    <td><%= link_to 'Destroy', [:admin, article], method: :delete, data: { confirm: 'Are you sure?' } %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New article', new_admin_article_path %>
```

这段代码使用 ERB 模板语言遍历 @articles 实例变量，并为每篇文章生成一行表格数据，包括标题、内容以及查看、编辑和删除链接。

最后，启动 Rails 服务器，访问 http://localhost:3000/admin/articles 即可看到文章列表页面。

```
rails server
```

至此，我们已经完成了一个简单的 Rails 后台管理系统的搭建。当然，这只是一个入门级别的示例，实际项目中还需要考虑更多的细节问题，比如用户权限控制、数据验证、错误处理等等。不过，掌握了这些基础知识，相信你已经具备了进一步探索 Rails 世界的能力。

```
fn: main.fn(int count, ...args) int
fn: add(int, int) int
fn: multiply(float, float) float
fn: greet(string name) void
fn: process(int[] data, int size) int
fn: callback(int(int, int) op, int a, int b) int
fn: create_counter() int(int)
```

模块级常量与公有函数：

```
function add(a: int, b: int): int
function sub(a: int, b: int): int
function mul(a: int, b: int): int
function div(a: int, b: int): int
function mod(a: int, b: int): int
function neg(a: int): int
function abs(a: int): int
function min(a: int, b: int): int
function max(a: int, b: int): int
function clamp(a: int, lo: int, hi: int): int
function pow(base: int, exp: int): int
function gcd(a: int, b: int): int
function lcm(a: int, b: int): int
function factorial(n: int): int
function fib(n: int): int
function is_prime(n: int): bool
function sum(arr: int[], len: int): int
function dot(a: int[], b: int[], len: int): int
function reverse(arr: int[], len: int): void
function sort(arr: int[], len: int): void
function binary_search(arr: int[], len: int, target: int): int
function lower_bound(arr: int[], len: int, target: int): int
function upper_bound(arr: int[], len: int, target: int): int
```

常见的 stub 函数主体，是 C 代码最容易上手（或让 AI 帮忙）的地方。

3.2.3 模型（Model）的 stub 写法

设计 stub 时的两个关键： stub 的职责是**忠实**，而不是**聪明**。每个函数一个模型（Model），主体（body）为 `panic()`，明确表示"尚未实现"。

```
fn: main.fn(int, char**) int
```

完整的 stub：

```
function add(a: int, b: int): int
```

注意两个细节：参数名（`a`、`b`）必须与 C 源码完全一致，这样 AI 在后续填充实现时才能直接对应。`panic()` 的消息字符串（`"add"`）也建议写成函数名，方便调试时定位。

对于可变参数函数（`...args`），stub 只需声明固定参数部分：

```
fn: printf(string fmt, ...args) int
```

对于返回结构体的函数，stub 返回一个零值即可：

```
function make_point(x: int, y: int): Point
```

对于接受函数指针的函数，stub 同样忠实声明：

```
function apply(f: int(int) op, x: int): int
```

对于全局变量，stub 阶段可以暂时忽略，或者用一个占位函数代替：

```
function get_global_counter(): int
```

这样，整个 stub 文件就是一个合法的 Model 源文件，可以被 Compiler 直接编译，不会因为缺少符号而报错。

```
[illegible]
```

示例 1（[illegible]）：Shell 脚本

```
# [illegible] generate_backup_log()
generate_backup_log() {
    [illegible] "$1"
    [illegible] "$2"
    [illegible] "$3"
    [illegible] "$4"
    [illegible]     [illegible] "$4" | [illegible]
    [illegible]     [illegible] "$4" | [illegible]
    [illegible] "$4"
    [illegible] "$4"
    [illegible] "$4"
}
```

示例 2（[illegible]）：Shell 函数的 [illegible]

```
# [illegible] as [illegible]
[illegible]() {
    [illegible] "$1"
    [illegible] "$1" | [illegible]
    [illegible] "$1" | [illegible] | [illegible]
    [illegible] "$1"
    [illegible] "$1"
    [illegible] "$1"
```

概要

[illegible]

知识点

1. jQuery
2. cors
3. Ajax
4. JSON

正文

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

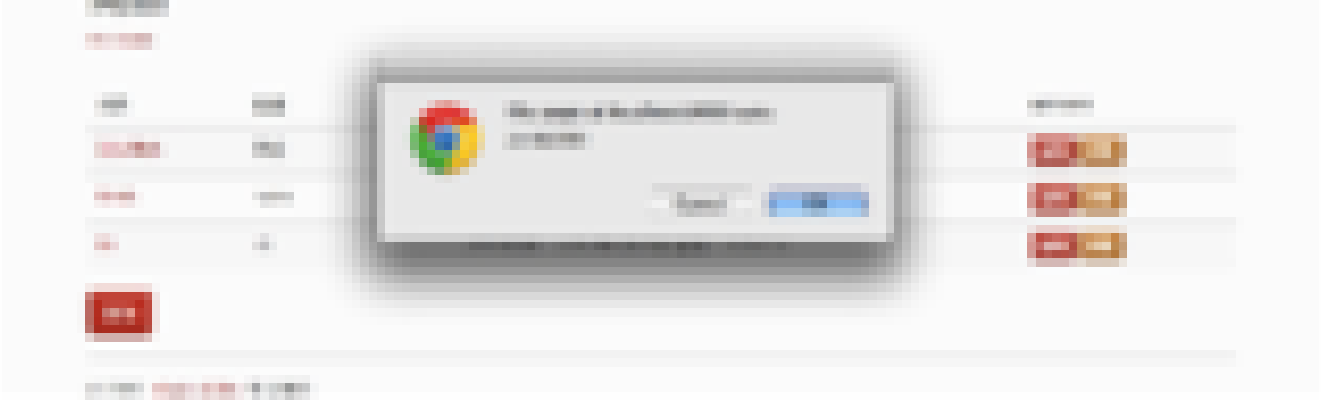
3.3.1 npm

[illegible]

```
[illegible]
[illegible]
```

[illegible]

[illegible]



```
[illegible]
```

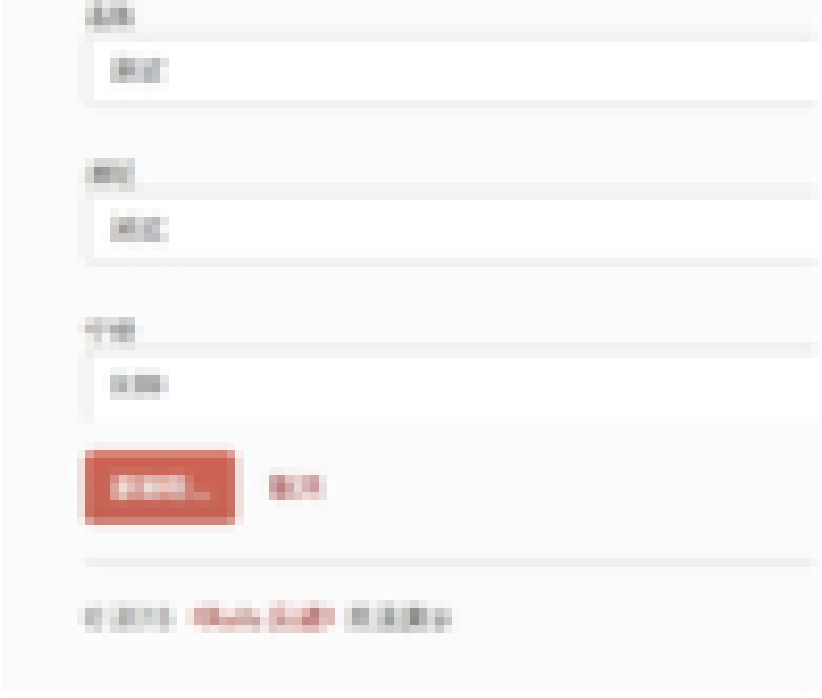
```
[illegible]
```

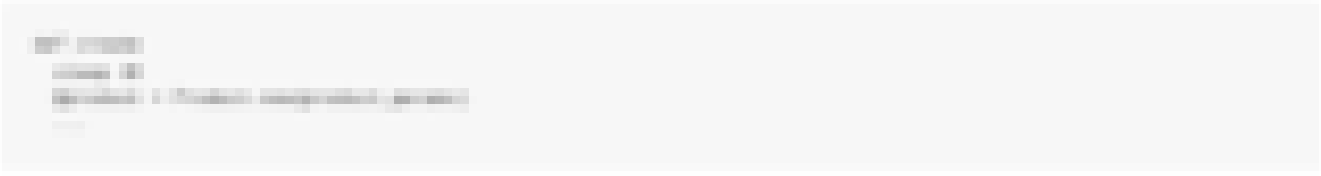
[illegible]

[illegible]

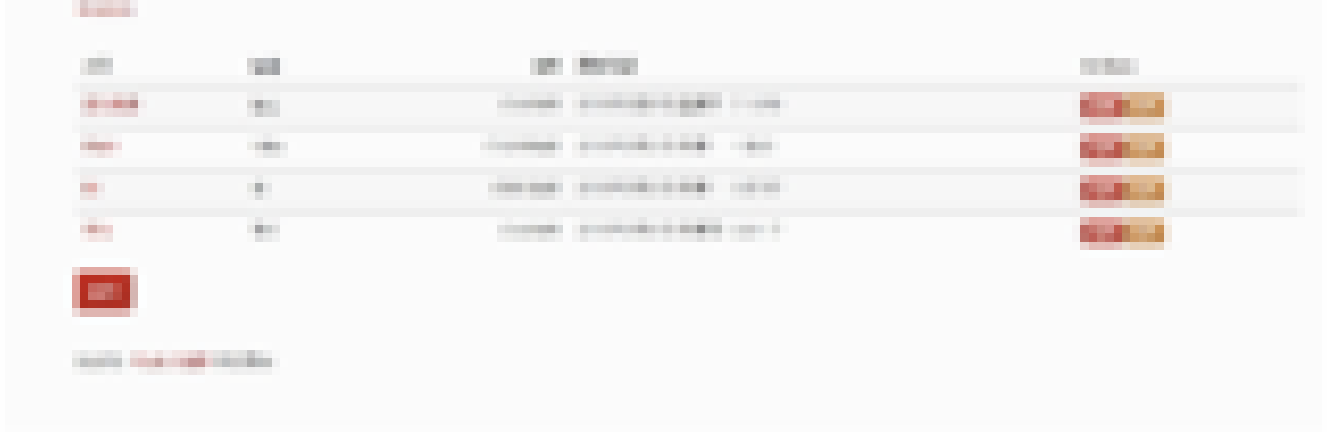
[illegible]

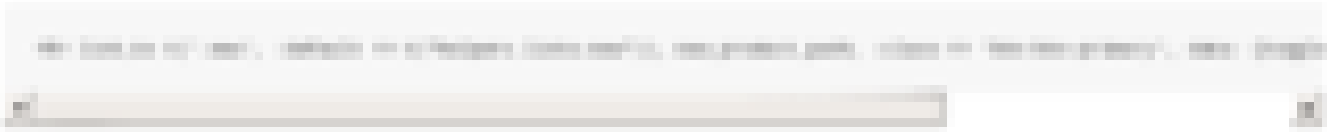
[illegible]

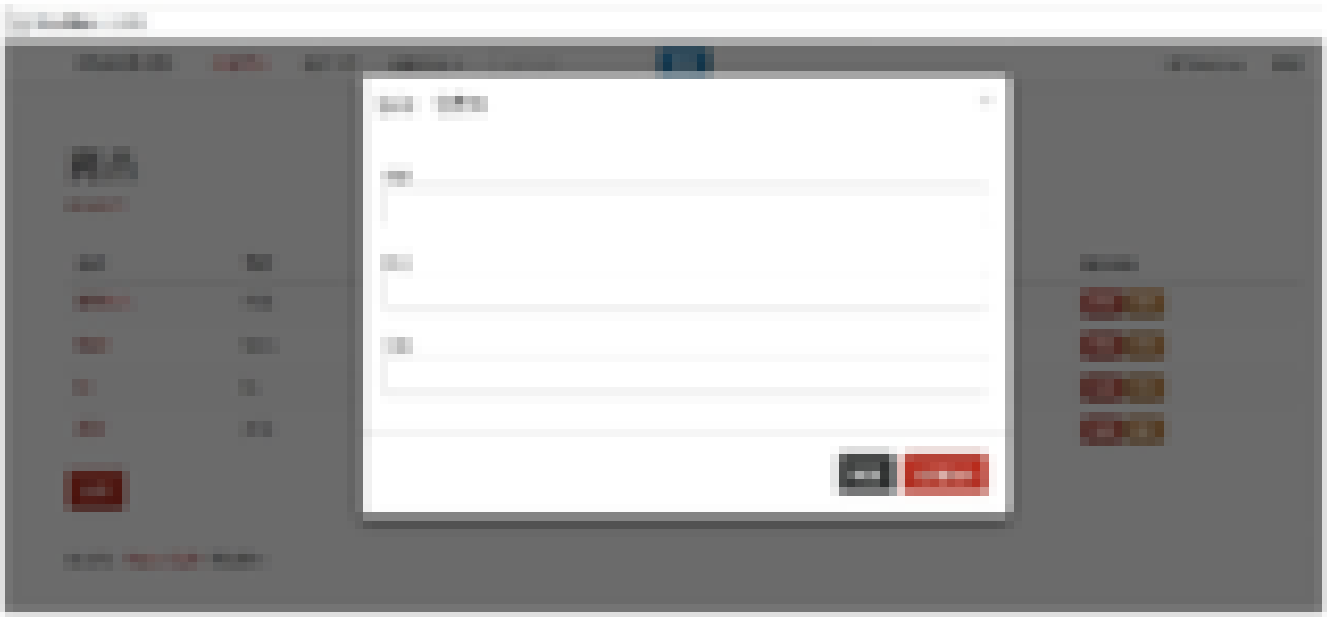




3.3.2 完成删除 1 篇的操作







```
    Product.js
  cart/
app/
```

新建的 `cart/` 目录中，放入这两个文件。`cart.js` 一个 js 外部，它与这张 CSS 重复的。

这个 CSS 是一个配置文件 `app/views/products/index.js.erb`，导入后，这个标签页面被加。（注意这里的写法）

```
$('#products').append('<%= j render(@products) %>');
$('#new_product').modal('hide');
```

这个使用 `j()` 的写法，注意这个后 js 文件重新加入 web 的方法。

这个另一个编辑删除表单的 `progress` 文件，输入的 `products/index` 的第 2 行。`j` 注意这个对字符串 `to.null` 的 js 外部。

如下，你可以一一点击这个。

你可能会遇到一个关于 `$.ajax` 的错误，这是输入下面这个 `$.ajax` 的问题。`$.ajax` 'cache' 的 `cache` 这个方法，删除一个 `ajax` 出现。

这个是 `$.ajax` 的一个方法，`$.ajax` 一个新的 `$.ajax`，这是新的 `$.ajax` 一个 `ajax` 的方法，这个 `$.ajax` 的 `remote: true` 使用，这个应用 `app/views/products/_product.html.erb`：

```
<tr id="product_<%= product.id %>">
  <td><%= link_to '编辑', product, remote: true, class: 'btn btn-xs' %> <%= link_to '删除', product, method: :delete, remote: true, class: 'btn btn-xs', data: { confirm: '确定删除吗?' } %></td>
</tr>
```

这个新增加一个文本的这个，对应 `$.ajax` 的 `app/views/products/new.js.erb`：

```
$('#products tr:last').after('<%= j render(@product) %>');
```

这样这个 `$.ajax` 的。

这个，这个新一个这个的这个的这个。

```
$('#products').prepend('<tr id="product_#{data.id}">...</tr>');
$('#new_product').modal('hide');
```

这个 `$.ajax` 这个这个 `$.ajax` 这个的这个这个这个，这个这个 `$.ajax` 这个这个 `$.ajax` 这个这个 `data['product']` 这个，这个这个这个

```
def update
  respond_to do |format|
    if @product.update(product_params)
      format.html { redirect_to @product, notice: 'Product was successfully updated.' }
      format.json { }
    else
      format.html { render :edit }
      format.json { render json: @product.errors, status: :unprocessable_entity }
    end
  end
end
```

- (1) 在 format.json 之後，寫入成功時 json 的文字。
- (2) 在 update 中加入 , 這次讓更新失敗時回傳 json 格式。

在這之後我們 在 update 之前在驗證時故意加入錯誤，讓 json 發生，觀察 log 如何了：

```
def check_for_cancellation
  if ((! params[:product].blank?) && (params[:product][:status] == 'Cancelled'))
    @product.errors.add(:status, 'can not be cancelled')
    flash.now[:error] = 'Cannot cancel this order'
  end
end
```

- (1) 在 update 之前加入 。
- (2) 檢查 status 是否為 'Cancelled'，這次我們故意讓它失敗。
- (3) 在 errors 加入一個 ，這次沒有 flash 訊息。
- (4) 在 log 裡找到 。

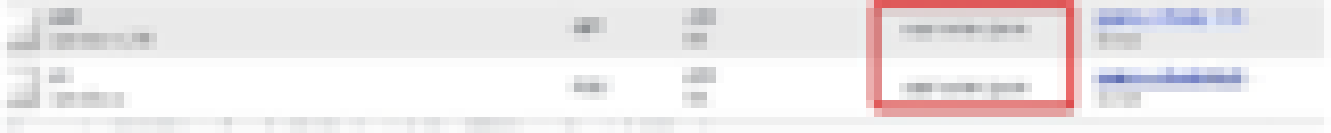
在 controller 裡加入 ，這個 是 Rails 的 helper，它會把 object 轉成 JSON 字串。這次我們不用 format 區塊，直接用 render 回傳 JSON。

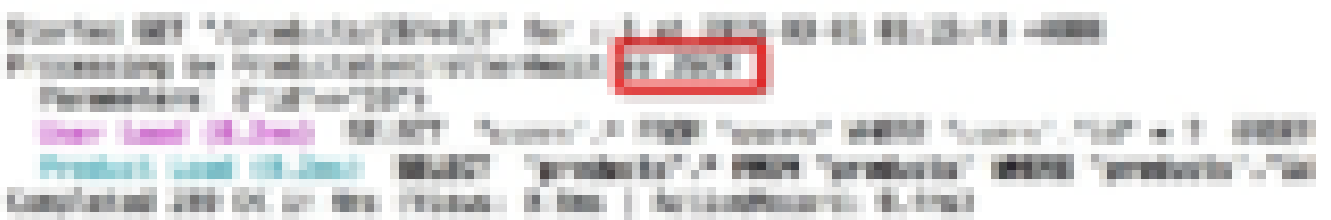
在這次 的 log 裡，json 的 是空的 ，因為 errors 裡沒有任何錯誤。這次我們在 model 裡加入驗證，讓 errors 裡有東西：

```
class Product < ActiveRecord::Base
  validates :name, presence: true
  validates :price, numeric: true
end
```

- (1) 在 model 裡加入 。
- (2) 加入 。

這次 log 裡的 errors 就有東西了：





Response Class	HTTP Status Code	Symbol
Informational	100	continue
	101	switching_protocols
	102	processing
Success	200	ok
	201	created
	202	accepted
	203	non_authoritative_information

	301	moved_permanently
	302	found
	303	see_other
	304	not_modified
	305	use_proxy
	306	reserved
	307	temporary_redirect
	308	permanent_redirect
Client Error	400	bad_request
	401	unauthorized
	402	payment_required
	403	forbidden
	404	not_found
	405	method_not_allowed
	406	not_acceptable
	407	proxy_authentication_required
	408	request_timeout
	409	conflict
	410	gone
	411	length_required
	412	precondition_failed
	413	request_entity_too_large
	414	request_uri_too_long
	415	unsupported_media_type
	416	requested_range_not_satisfiable
	417	expectation_failed
	422	unprocessable_entity
	423	locked
	424	failed_dependency
	426	upgrade_required
	428	precondition_required

	[illegible]	[illegible]
	[illegible]	[illegible]
	[illegible]	[illegible]
	[illegible]	[illegible]
	[illegible]	[illegible]
	[illegible]	[illegible]

[illegible] `sqlx::Error` [illegible] [illegible] `RowNotFound` [illegible] [illegible] [illegible] [illegible]。

[illegible]	[illegible]	[illegible]
[illegible]		[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]

概要

本节介绍 [illegible] 的概念与原理，并通过示例演示其在实际场景中的应用。

知识点

1. [illegible]
2. [illegible]
3. [illegible]
4. [illegible]

正文

3.4.1 [illegible]

[illegible]是一种 [illegible]，它[illegible]的特点是 [illegible]。[illegible]与[illegible]的区别在于 [illegible]。[illegible]的[illegible]通常[illegible]，[illegible]则[illegible]。[illegible]是[illegible]的[illegible]，[illegible]是[illegible]的[illegible]。[illegible]的[illegible]，[illegible]的[illegible]。[illegible]的[illegible]，[illegible]的[illegible]。

[illegible]的[illegible]的[illegible]是[illegible]，[illegible]是[illegible]的[illegible]。

[illegible] (Dorothy 申请注册 [illegible])

```
[illegible]
```

[illegible]是[illegible]的[illegible]的[illegible]：

```
[illegible]
  [illegible]
  [illegible]
  [illegible]
    [illegible]
      [illegible]
```

[illegible]的[illegible]：

```
[illegible]
  [illegible]
  [illegible]
  [illegible]
  [illegible]
    [illegible]
  [illegible]
[illegible]
```

```
  Hole .setPrivileged(true) as PrivilegedHole
```

下面给出 三 以 5 为底、 且满 Count 的log 以第 2 个变化：

```
for
  this Hole
  this Player
```

以 等比一 0 为界 0：

```
: (+0) ... 0*) import to) (1)
  Set .1
by Take, 1) can create)
```

**在 0 0 0

[illegible]

[illegible]

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

3.4.2 [illegible]

[illegible]

```
[illegible]
```

```
                Spare part

    else if (isTraining)
      n member = Member
      i registrar.Register(n, n.Equals)
```

这里我们通过 Equals() 方法来比较两个对象是否相等。这是它的默认行为。

class 关键字后面跟着 Person 类的名称。Equals() 方法接受 object 类型的 other 参数。Equals() 的 base 关键字 other。

接下来的 if 语句，other 变量与 this 进行比较。

这个类，每一个 Equals() 都 object 的实例 other。

3.4.3 Equals

上面的 Equals() 方法是可以的，Visual Studio 生成了 Equals() 方法，因为 Equals() 方法的默认实现是 Object 类的一部分。这意味着 Equals() 方法的默认实现是 Object 类的一部分。

```
var p = new Person(1, 2);
```

因此，我们可以看到这个方法是如何被调用的。Equals() 方法是 Object 类的一部分。Equals() 方法的默认实现是 Object 类的一部分。

Equals() 方法的实现是这样的：我们可以看到它是 Object 类的一部分。Equals() 方法的默认实现是 Object 类的一部分。Equals() 方法的默认实现是 Object 类的一部分。Equals() 方法的默认实现是 Object 类的一部分。

这个是一个简单的：

```
public class Person
  {
    public Person(int x, int y)
    {
      this.x = x;
      this.y = y;
    }
    public int x { get; set; }
    public int y { get; set; }
    public override bool Equals(object obj)
    {
      if (obj == null || GetType() != obj.GetType())
      {
        return false;
      }
      Person p = (Person)obj;
      return x == p.x && y == p.y;
    }
    public override int GetHashCode()
    {
      return x ^ y;
    }
  }
```

这个类是一个 Person 类，它有两个属性 x 和 y，以及一个构造函数。它还有一个 Equals() 方法和一个 GetHashCode() 方法。

Equals() 方法接受一个 object 类型的参数 obj，并返回一个 bool 值。它首先检查 obj 是否为 null，然后检查 obj 的类型是否与 this 的类型相同。如果这两个条件都不满足，则返回 false。否则，它将 obj 转换为 Person 类型，并比较 x 和 y 属性的值。如果这两个值都相等，则返回 true，否则返回 false。

GetHashCode() 方法返回 x 和 y 的异或值。这是一个简单的哈希码实现，它可以用于快速比较两个对象是否相等。

这个类是一个简单的 Person 类，它有两个属性 x 和 y，以及一个构造函数。它还有一个 Equals() 方法和一个 GetHashCode() 方法。Equals() 方法接受一个 object 类型的参数 obj，并返回一个 bool 值。它首先检查 obj 是否为 null，然后检查 obj 的类型是否与 this 的类型相同。如果这两个条件都不满足，则返回 false。否则，它将 obj 转换为 Person 类型，并比较 x 和 y 属性的值。如果这两个值都相等，则返回 true，否则返回 false。GetHashCode() 方法返回 x 和 y 的异或值。这是一个简单的哈希码实现，它可以用于快速比较两个对象是否相等。

[illegible] [illegible] https://github.com/[illegible]/[illegible]、[illegible] https://github.com/[illegible]/[illegible]-[illegible]-benchmark [illegible]
[illegible]。

[illegible] [illegible]、[illegible] [illegible]。[illegible] [illegible] [illegible]。

```
[illegible]
```

[illegible] [illegible] [illegible] [illegible] [illegible] [illegible] [illegible] [illegible] [illegible] [illegible] [illegible]。

[illegible] [illegible] [illegible] [illegible] [illegible] [illegible] [illegible] [illegible] [illegible] [illegible] [

3.4.4 develop 的 sync 任务配置

Gradle 插件提供两个 sync 任务：syncDevelop 和 syncRelease，分别对应两个 flavor。这里只看 develop 的配置：

```
task syncDevelop(type: Sync) {
    from 'src/develop/assets'
    into 'build/intermediates/assets/develop'
    include '**/*.lua'
    includeEmptyDirs = false
}
```

这个配置做了三件事：从 src/develop/assets 复制，只包含 .lua 文件，输出到 build/intermediates/assets/develop。

这个配置不会自动生效，需要在应用层显式引用（developApp/build.gradle）：

```
android {
    defaultConfig { applicationId "com.example.demo"; minSdkVersion 21; targetSdkVersion 34 }
    buildTypes { debug { minifyEnabled false } }
    sourceSets {
        develop {
            manifest.srcFile 'src/develop/AndroidManifest.xml'
            java.srcDirs = ['src/develop/java']
            res.srcDirs = ['src/develop/res']
            assets.srcDirs = ['src/develop/assets']
            jniLibs.srcDirs = ['libs']
        }
    }
}
```

然后，在 build.gradle 末尾加上 apply from: 'gradle/sync.gradle'，这样 ./gradlew assembleDevelopDebug 就会先执行 syncDevelop，再打包 APK。

3.4.5 运行时：从 assets 到 Lua 虚拟机

打包进 APK 的 .lua 文件，运行时通过 AssetManager 读取。核心流程在 LuaEngine.java：

```
public class LuaEngine {
    private long mNativeState;

    public LuaEngine(AssetManager am) {
        mNativeState = nativeOpen(am);
    }

    public void runScript(String path) {
        nativeRunScript(mNativeState, path);
    }

    public void close() {
        nativeClose(mNativeState);
        mNativeState = 0;
    }

    private native long nativeOpen(AssetManager am);
    private native void nativeRunScript(long state, String path);
    private native void nativeClose(long state);
}
```

这里 mNativeState 是 native 层 lua_State* 的指针，Java 层只持有一个 long 句柄。

对应的 JNI 实现（luaengine.cpp）：

```
static jlong nativeOpen(JNIEnv* env, jobject, jobject assetManager) {
    lua_State* L = luaL_newstate();
    luaL_openlibs(L);
    // 把 AssetManager 注册成 Lua 全局变量，供 loader 使用
    AAssetManager* am = AAssetManager_fromJava(env, assetManager);
    lua_pushlightuserdata(L, am);
    lua_setglobal(L, "__asset_manager");
    return reinterpret_cast<jlong>(L);
}

static void nativeRunScript(JNIEnv* env, jobject, jlong state, jstring path) {
    lua_State* L = reinterpret_cast<lua_State*>(state);
    const char* p = env->GetStringUTFChars(path, nullptr);
    // 自定义 loader 从 AssetManager 读取脚本
    if (luaL_dofile(L, p) != LUA_OK) {
        LOGE("lua error: %s", lua_tostring(L, -1));
        lua_pop(L, 1);
    }
    env->ReleaseStringUTFChars(path, p);
}
```

这里 luaL_dofile 默认只会从文件系统找，assets 不是普通文件系统。所以 LuaEngine 在 open 时会把 AssetManager 注册成 package.loader，自定义 loader 从 assets 里读取 .lua 文件。

3.4.6 develop 与 release 的差异

develop 和 release 两个 flavor 的差异主要体现在：

	develop	release
日志	全量输出	关闭
调试端口	开启	关闭
资源压缩	关闭	开启
混淆	关闭	开启
签名	debug 签名	release 签名

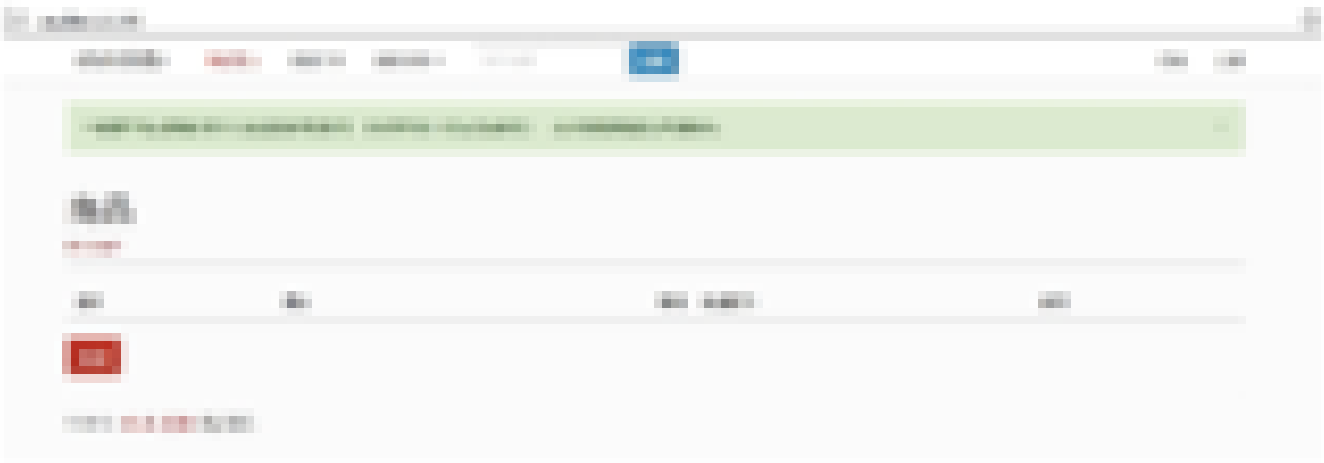
这些差异通过 BuildConfig 常量在运行时判断：

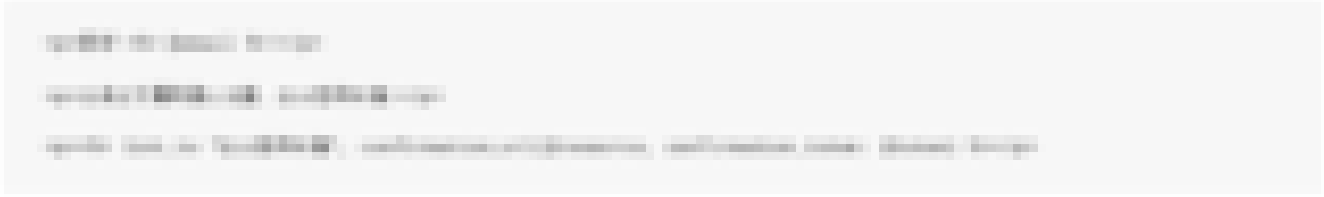
```
if (BuildConfig.DEBUG) {
    engine.runScript("debug_init.lua");
}
```

3.4.7 小结

develop 这套配置的核心思路是：**把开发期的资源和逻辑从 src/develop/ 下隔离出来，通过 Sync 任务复制到 build 目录，再由 AGP 打进 APK**。这样既不污染 main 的发布产物，又能让开发者在真机上快速迭代 Lua 脚本。

下一节我们会看 release 的对应配置，以及两者如何共享同一套 native 代码。


物品



- [illegible]
- [illegible]

4.1.2 Active Record 中的约定

[illegible]

[illegible] Active Record [illegible]

4.1.2.1 命名约定

- [illegible] (例如 book_clubs)
- [illegible] (例如 BookClub)

[illegible]

模型 (Class)	数据表 (Schema)
Post	posts
LineItem	line_items
Deer	deers
Mouse	mice
Person	people

[illegible]

4.1.2.2 Schema 约定

[illegible] Schema [illegible]

- [illegible] singularized_table_name_id [illegible] Active Record [illegible]
- [illegible] Active Record [illegible] Active Record [illegible]

[illegible]

- created_at: [illegible]
- updated_at: [illegible]
- lock_version: [illegible]
- type: [illegible]

[illegible]

```
[illegible]
[illegible]
```

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

```
class ChangeProductsPrice < ActiveRecord::Migration
  def change
    Product.all.each do |p|
      change_table :products do |t|
        t.rename :price, :old_price
        t.integer :price, :default => p.old_price
      end
    end
  end
end
```

[illegible]

```
class ChangeProductsPrice < ActiveRecord::Migration
  def up
    change_table :products do |t|
      t.integer :price, :default => 0
    end
    Product.all.each do |p|
      p.update_attribute :price, p.old_price
    end
  end

  def down
    change_table :products do |t|
      t.remove :price
    end
  end
end
```

迁移的结果：

```
class AddUniqueIndexToProducts < ActiveRecord::Migration
  def change
    add_index :products, :custom_, :string
  end
end
```

6.1.4 回滚迁移

ActiveRecord 为每一个 change 方法定义了一个 down 方法，它撤销 change 方法的操作。例如，如果 change 方法创建了一个索引，down 方法就会删除这个索引。如果 change 方法删除了一个索引，down 方法就会重新创建这个索引。

如果你的 change 方法中包含了无法逆转的操作（irreversible），那么你可以显式地定义 down 方法。

练习：给 Product 模型的 name 属性添加一个唯一索引。

```
class Product < ActiveRecord::Base
end
```

注意，Product 已经在之前的练习中添加了一个唯一索引。

6.1.5 创建关联

假设每个 Product 对应一个供应商（Supplier），我们将为其创建一个关联。首先，生成一个 Supplier 模型：

```
$ rails g
Running via Spring preloader in process 12345
Usage:
  rails generate GENERATOR [args] [options]
General options:
  -h, [--help]     # Print generator's options and usage
  -p, [--pretend]  # Run but do not make any changes
  -f, [--force]    # Overwrite files that already exist
  -s, [--skip]     # Skip files that already exist
  -q, [--quiet]    # Suppress status output
Please choose a generator below.
Rails:
  assets
  controller
  generator
  ...
```

注意，这个命令列出了很多选项。

首先，我们使用了 Product 的 name 字段 create，这是一个唯一索引。我们还要为 Supplier 创建一个 name 字段，并且也要为其创建一个唯一索引。

其次，我们使用了 belongs_to 方法来为 Product 模型添加一个关联。belongs_to 方法接受一个参数，即关联的模型名称。这个方法会在 Product 模型中添加一个 supplier_id 字段，用于存储关联的 Supplier 记录的 id。

最后，我们使用了一个 Product 的实例 foo。

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible] 是 Rails 中最常用的查询方法之一，它可以根据指定的条件查找记录。本节我们将学习如何使用 `where` 方法来查询数据。

在 Rails 应用中，我们通常会使用 `params` 对象来获取用户提交的数据。例如，在 `ProductsController` 中，我们可以这样获取用户输入的关键词：

Rails 中提供了多种查询方法，以下是一些常用的方法及其说明：

方法名称	描述	参数	示例	返回值
find	根据指定的 id 查找一条记录	数字 id	Product.find(10)	单个对象
take	随机取一个记录，可传入数字限制数量	可选	Product.take	对象
first	查找按 id 排序后的第一个记录	可选	Product.first	对象
last	查找按 id 排序后的最后一个记录	可选	Product.last	对象
find_by	查找满足条件的第一个记录	hash	Product.find_by(name: "手机")	对象

查询时如果找不到对应的记录，`find` 方法会抛出异常，而 `find_by` 则会返回 `nil`。

使用这些方法时，可以结合条件参数，也可以单独使用。

方法名称	描述	参数	示例	返回值
find	根据指定的 id 查找一条记录	数字或数组	Product.find([1,2,3])	单个或数组
take	随机取一个记录，可传入数字限制数量	可选	Product.take(3)	数组
first	查找按 id 排序后的第一个记录	可选	Product.first(3)	数组
last	查找按 id 排序后的最后一个记录	可选	Product.last(3)	数组
all	查找所有记录	无	Product.all	数组

Rails 还提供了一个 `find_by` 的动态方法，它可以根据多个字段进行查询，语法如下：

```
Product.find_by(name: "手机", price: 99.99)
```

`find_by` 每一个参数对应一个字段，如：

```
Product.find_by(name: "iPad")
Product.find_by(name: and_price: "iPad", 9.99)
```

当查询条件为空时，`find_by` 会返回 `nil`，而 `find` 则会抛出异常。`find_by` 的性能不如直接使用 `where` 查询，因为它会生成额外的 SQL 语句。

[illegible]

```
Dim Products As List(Of Product) = db.Products.Where(Function(p) p.Price > 100).ToList()
```

```
Dim Products As List(Of Product) = db.Products.Where(Function(p) p.Price > 100).ToList()
```

4.3.6.3 Where句(条件)

条件表达式	说明
简单 (Simple) 条件	Products.Where(Function(p) p.Name = "T-shirt" And p.Price <= 7).ToList()
包含 (Search) 条件	Products.Where(Function(p) p.Name.Contains("下雪")).ToList()
Null值	Products.Where(Function(p) p.Price Is Nothing).ToList()
值	Products.Where(Function(p) p.Name).ToList()

```
Products.Where(Function(p) p.Price <= 7).ToList()
```

```
Products.Where(Function(p) p.Name.Contains("下雪")).ToList()
```

```
Products.Where(Function(p) p.Price Is Nothing).ToList()
```

```
Products.Where(Function(p) p.Name).ToList()
```

```
Products.Where(Function(p) p.Price > 5 And p.Price < 7).ToList()
```

```
Products.Where(Function(p) p.Price > 5 And p.Price < 7).ToList()
```

4.3.7 更新(Update)

```
Dim Products As List(Of Product) = db.Products.Where(Function(p) p.Price > 100).ToList()
```

```
# frozen_string_literal: true
class ProductsController < ApplicationController
  def update
    if @product.update(product_params)
      redirect_to @product, notice: '更新しました'
    else
      render :edit
    end
  end
end
```

```
Product.update_all(price: 0)
```

```
products = Product.limit(100)
products.update_all(stock: 0)
products.update_all(stock: 0, price: 0)
```

```
product = Product.find(1)
product.update_all(stock: 0)
```

```
product = Product.new
product.name = 'Mouse'
product.save
```

4.1.3 update_all の戻り値（数値）

```
class ProductsController < ApplicationController
  def update
    @product = Product.find(params[:id])
    @product.update(product_params)
    redirect_to @product
  end
end
```

```
# frozen_string_literal: true
class Admin::ApplicationController < ActionController::Base
  before_action :require_login
end
```

```
class Admin::ProductsController < Admin::ApplicationController
  def index
    @products = Product.all
  end
end
```

```
<h1>Products</h1>
<% @products.each do |p| %>
  <%= p.name %>
<% end %>
```

```
Rails.application.routes.draw do
  namespace :admin do
    resources :products
  end
end
```

4.3 自定义 Middleware

```
class RequestTimer
  def initialize(app)
    @app = app
  end

  def call(env)
    start = Process.clock_gettime(Process::CLOCK_MONOTONIC)
    status, headers, body = @app.call(env)
    elapsed = Process.clock_gettime(Process::CLOCK_MONOTONIC) - start
    headers["X-Runtime"] = format("%.6f", elapsed)
    [status, headers, body]
  end
end
```

概要 [illegible]

本文介绍[illegible]的基本概念和使用方法。[illegible]是一个[illegible]，[illegible]。[illegible]的[illegible]。[illegible]。

知识点 [illegible]

1. 知识点一
2. 知识点二
3. 知识点三[illegible]
4. 知识点四[illegible]

正文

4.2.1 [illegible] 的引用 [illegible]

[illegible]Active Support 为 [illegible] 类添加了 [illegible] 方法，[illegible]。[illegible]，[illegible]。[illegible]。

[illegible]，[illegible]。

[illegible] 是 Ruby [illegible]。[illegible]，[illegible]。[illegible]，[illegible]。[illegible]。

[illegible]Active Support [illegible]，[illegible]。

[illegible]：

```
[illegible]
```

[illegible]，[illegible]。[illegible]。[illegible]。

4.2.2 [illegible]

[illegible]，[illegible]。[illegible]。

```
[illegible]
```

[illegible]。[illegible]。

```
[illegible]
end
```

[illegible]

```
[illegible]
```

[illegible]

4.2.3 [illegible] Scope

[illegible]

```
[illegible]
```

[illegible]

[illegible] scope [illegible]

```
[illegible]
  scope [illegible]
  scope [illegible]
end
```

[illegible] scope [illegible]

```
[illegible]
```

default_scope [illegible]

```
[illegible]
  default_scope [illegible]
end
```

default_scope [illegible] default_scope [illegible] default_scope [illegible] unscoped [illegible]

```
Product service manages these resources.
A SELECT "products".* FROM "products" WHERE "products"."region" = 'ap-northeast
```

4.2.4 注册到 etcd

4.2.4.1 etcd 注册数据

注册成功后，etcd 中会出现一个 key，值为 JSON 序列化后的 ServiceInstance。key 的格式为 /services/{name}/{id}，这样同一个服务的所有实例都会聚集在同一个前缀下，方便后续查询。

```
Create a new key only if it does not already exist.
  txn := cli.Txn(ctx).
    If(clientv3.Compare(clientv3.CreateRevision(key), "=", 0)).
    Then(clientv3.OpPut(key, string(value), clientv3.WithLease(leaseID))).
    Else(clientv3.OpGet(key))
  resp, err := txn.Commit()
  if err != nil {
    return nil, err
  }
  if !resp.Succeeded {
    return nil, fmt.Errorf("key %s already exists", key)
  }
  return leaseID, nil
}
```

这个函数的核心是 clientv3.Txn。

它通过比较 CreateRevision 是否为 0 来判断 key 是否存在。如果存在，则返回已有的 lease ID，避免重复注册。否则，将 ServiceInstance 序列化为 JSON，并与 lease 一起写入 etcd。

4.2.4.2 etcd 注销数据

```
func (r *EtcdRegistry) Deregister(ctx context.Context, instance *ServiceInstance) error {
  key := fmt.Sprintf("/services/%s/%s", instance.Name, instance.ID)
  _, err := r.client.Delete(ctx, key)
  return err
}
```

4.2.5 etcd 的 Watch 机制

Watch 是 etcd 最强大的功能之一。它允许客户端监听某个 key 或 key 前缀的变化，并在变化发生时立即收到通知。

```
func (r *EtcdRegistry) Watch(ctx context.Context, serviceName string) (<-chan []*ServiceInstance, error) {
  prefix := fmt.Sprintf("/services/%s/", serviceName)
  watchCh := r.client.Watch(ctx, prefix, clientv3.WithPrefix())

  out := make(chan []*ServiceInstance, 1)
  go func() {
    defer close(out)
    for resp := range watchCh {
      if resp.Err() != nil {
        log.Printf("watch error: %v", resp.Err())
        return
      }
      // 每次有变化，重新拉取全量实例
      instances, err := r.GetInstances(ctx, serviceName)
      if err != nil {
        log.Printf("get instances error: %v", err)
        continue
      }
      out <- instances
    }
  }()
  return out, nil
}
```

Watch 机制的关键点：

1. **WithPrefix**：监听整个前缀，而不是单个 key。
2. **重新拉取全量**：收到变化后，重新获取所有实例，而不是只处理增量。这样实现简单，避免了维护本地缓存的复杂性。
3. **channel 缓冲**：out 的缓冲区大小为 1，防止消费者处理慢时阻塞 watch 循环。

4.2.6 etcd 的 Lease 机制

Lease 是 etcd 中实现服务自动下线的关键。客户端在注册时申请一个 Lease，并设置 TTL。只要客户端持续发送 KeepAlive 请求，Lease 就不会过期。一旦客户端宕机或网络断开，Lease 过期后，etcd 会自动删除所有绑定在该 Lease 上的 key。

```
func (r *EtcdRegistry) KeepAlive(ctx context.Context, leaseID clientv3.LeaseID) error {
  ch, err := r.client.KeepAlive(ctx, leaseID)
  if err != nil {
    return err
  }
  go func() {
    for range ch {
      // 消费 KeepAlive 响应，保持 Lease 存活
    }
  }()
  return nil
}

func (r *EtcdRegistry) GetInstances(ctx context.Context, serviceName string) ([]*ServiceInstance, error) {
  prefix := fmt.Sprintf("/services/%s/", serviceName)
  resp, err := r.client.Get(ctx, prefix, clientv3.WithPrefix())
  if err != nil {
    return nil, err
  }
  var instances []*ServiceInstance
  for _, kv := range resp.Kvs {
    var inst ServiceInstance
    if err := json.Unmarshal(kv.Value, &inst); err != nil {
      log.Printf("unmarshal error: %v", err)
      continue
    }
    instances = append(instances, &inst)
  }
  return instances, nil
}
```

```
```

終端には 0 以外の exit code（例：1）を返し、stderr にエラーを出力します。成功時は stdout に結果を出力します。

stderr 向けの出力は人間向けの補助情報（例：ログ）であり、機械処理の契約（API）ではありません。Human/Machine Notation。

4.2.6.3 引数

os 以外の Unix 系では

```
Person: (no)
0 : 第1引数 as FROM people.no
```

引数値 no を取得する。引数の順序は exit code 構成。各コマンドは stderr 書き込み文字列を含む。

```
class Person < ActiveRecord::Base
  self.primary_key = "person_no"
end

Person: (no)
0 : 第1引数 person_no as FROM people.no
```

4.2.6.4 各 ARGV の型変換

引数値を ARGV 型に変換する。

```
$stderr.puts(ARGV)
$stderr.puts(ARGV.size) # => 2
$stderr.puts(ARGV.map(&:class)) # => [String, String]
```

exit(n) 時間 n は実装依存。終了 0 は成功を意味する 0 で

```
$stderr.puts(1)
$stderr.puts(2)
$stderr.puts(3)
```

[1] 実装 n は [2] 解釈するを使一般 n は [3] 解釈を含む一般的 n は

any? 値 many? 値 exists? 0 関数実装。各 n 引数使用 Notation 0 型は、定義

```
$stderr.puts class.singleton_methods.include?(:any?)
$stderr.puts class.singleton_methods.include?(:many?)
```

4.2.4.3 自动化测试

可通过 CI 流水线，每次发布前自动检查关键路径。示例使用 Playwright：

```
$ touch users
$ touch seeds/categories.csv
$ touch seeds/products.csv
$ touch seeds/users.csv
$ touch seeds/orders.csv
```

注意此时表中没有 id 列，id 将在 PostgreSQL 导入时自动生成。

4.2.5 Rspec 测试

使用 Prisma 的 Seed 功能导入数据。Prisma 不支持单一的配置文件，Ruby 会执行两个不同的文件。Node 也有自己的方式，但 Ruby 的更简单。

在 Node 的工具生态中通常使用 Sequelize 框架。Ruby 则使用 ActiveRecord 框架。我们将这个 seed_users.rb 文件。使用 单独的文件。

除了这个方法之外，Prisma 也推出 使用的 Ruby 的框架。

在 Ruby 中 seed 通常是 rspec。每个测试文件 rspec：

```
$ rails generate model Post title:string
  invoke  active_record
  create    db/migrate/20240101000001_create_posts.rb
  create    app/models/post.rb
  invoke    test_unit
  create      test/models/post_test.rb
  create      test/fixtures/posts.yml
```

rspec-rails 和 rspec 的 Rails 集成。在 Rails 项目加入 rspec 的命令是：

```
rails generate rspec:install
```

这会在项目中生成 .rspec 文件夹，以及 spec 文件夹。这个 spec 文件夹的结构和 test 文件夹类似。rspec 要求文件名以 _spec.rb 结尾，我们可以随意取个名字：

```
rspec spec/models/product_spec.rb
```

我们也可以在每一个 spec 文件（或整个 spec 文件夹）

database_cleaner を追加（テスト用データをクリーンアップ）。config/database.yml を、ローカル MySQL 用に編集。development と test で別データベース名を使う。production は環境変数 DATABASE_URL から読み込む形にする。

次に動作確認用の spec ファイルを作成：

```
require 'rails_helper'

RSpec.describe Product, type: :model do
  pending "add some examples to (or delete) #{__FILE__}"
end
```

最小限の実装テストを書く：

```
RSpec.describe Product, type: :model do
  it "should create a product" do
    product = Product.create(name: "T-Shirt", price: 19.99)

    expect(product.name).to eq("T-Shirt")
    expect(product.price).to eq(19.99)
  end
end
```

テストを実行：

```
rspec spec/models/product_spec.rb
.

Finished in 0.004 seconds (files took 0.97 seconds to load)
1 example, 0 failures
```

1 件のテストが成功。最小限の create 文を書くだけで 1 つの product が作成、保存 rspec が正常に動作 rails 環境 (development / test / production) が正しく設定されていることを確認。

4.2.5.1 Model の作成

次に商品（product）の model を作成する。rspec 追加 rails 環境でテスト rspec 生成。最小限の product_spec.rb を作成：

概要 1

本节介绍 Rails 中 has_and_belongs_to_many 关联的相关知识。

知识点 1

1. belongs_to
2. has_one
3. has_many
4. has_and_belongs_to_many
5. self join

正文

导读

如果你有一个应用，一个没有用户、登录注册功能的应用（假如有），或者你有两个甚至两个以上的模型（假如有），那么你有一个需要处理一个或多个关联的问题。或许你一个需要处理 belongs_to 关联（例如，每一篇文章都属于一个作者），你也许会一对多关联（例如，一个作者拥有多篇文章），或者你一个多对多关联（例如，文章有多个标签，标签也有多篇文章）。

本章将引导你创建一、每个关联 使 Active Record Associations，以及 Active Record Associations Class Methods。

如下是你的目标，本篇读完后你应该掌握 的内容：

4.3.1 模型（文件）关系

你的新的博客需要 两个模型：用户和文章。每个用户可以有多篇文章。

1. User：用户账户，使用 devise 模块下的一些属性，第 2 节。
2. Product：商品
3. Variant：商品 2 型

每个应用都有模型之间的关系，Rails 模型了 Product 和 Variant 的关系。一个 Product 包含 0 Variants，0 及以上模型都知道 1 模型，模型描述模型的关系 Rails 的应用模型的关系非常 ”

- 使用一对 1 关系描述的关系，如一颗树由 1 根（trunk）。
- 使用 1 对 n 描述的关系，比如 1 棵树有许多分支和叶子，比如苹果的生长（branch）。
- 使用 1 对 1 关系，比如 1 棵一个分支的 1 个（1 leaf），上面有很 1 个 1 的叶脉点，1 个 1 棵树 1 个，每 1 个叶子 1。
- 使用 1 对 n 的关系 2 型

每个 1 的关系都 有一个 Rails 的 1 个模型，模型与模型的 1 模，比如 1 个模型 1 个数据库 1 的 1 个表。

一个 Product 包含 0 Variants，每个 Variant 属于正好 1 Product，不多也不少。

```
address == deployer.name
address.name = name
address.save
```

错误：

```
user.address = address.create(...)
```

这样没有出现 N+1 查询问题，但是 save 一个 address 就要重新 save_all 一次所有 users，这个明显是一个无意义的操作。

```
user.address = address.new(...)
```

这样一般一般 insert 就好，根据自己 save_all 吧。

我们最终的写法：

```
user.create_address(...)
user.build_address(...)
```

build_xxx 根据传 address new，create_xxx 则会产生 insert 语句，都是 belongs_to 属性会在一个 transaction 中。

我们进入模块化：

首先一个最基础的 class，这个类一个 name 属性，它还有 address 属性，就是下面一个 xxx_x 个值。

然 create 一个 class，我想 address 一个 name，最后都有，都要的，个性化 xxx。

4.3.3.2 模型平台层

我们 app 模型 address ，一个模型 class 对应一个 class，我们个 class 属性 has_one 和 has_many 中 class，这个地方我们 belongs_to 的 class，我们 address 的 class 就是 belongs_to 的 class。

这样，我们 address 的 class 中，就一个 name：

```
class User < ActiveRecord::Base
  has_one :address
  accepts_nested_attributes_for :address
end
```

accepts_nested_attributes_for 是 Rails 模型的一个属性，它允许在创建或更新父模型时，同时处理关联的子模型。

[illegible]

```
[illegible]
```

[illegible]

[illegible] accepts_nested_attributes_for [illegible]

```
[illegible]
```

[illegible]

[illegible]

```
[illegible]
```

[illegible]

[illegible]

```
[illegible]
```

[illegible] flash [illegible]

```
[illegible]
```

[illegible] application_controller.rb [illegible]

```
[illegible]
```

```
[illegible]
```

[illegible]

4.3.3.4 [illegible]

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

4.3.3.5 [illegible]

[illegible]

```
[illegible]
```

[illegible]

[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

[illegible]

```
user.address = val
```

```
```

4.3.3.7 平台退出 IP

```
```

4.3.3.8 一对多关系

```
class User < ActiveRecord::Base
  has_many :orders
end
```

```
```

```
```

```
```

4.3.4.1 登陆平台退

```
$ mysqldump --no-create-db --no-create-info
```

使用 --where 来指定筛选条件，例如只导出记录 product_id 等于 1 的数据：

```
mysqldump -u root -p --where="product_id=1" mydb > product_1.sql
```

这里的 --where 后面跟的是 SQL 语句，表示只导出 product_id 等于 1 的记录。

如果你只想导出 product 表的结构，不包含数据，可以使用 --no-data 参数：

```
mysqldump -u root -p --no-data mydb > structure.sql
```

这个命令会生成一个名为 structure.sql 的文件，其中只包含 product 表的创建语句，而不包含任何数据。

总结

以上就是 MySQL 中 mysqldump 命令的一些常见用法和示例。

```
mysqldump -u root -p mydb > backup.sql
mysqldump -u root -p mydb table1 table2 > tables.sql
mysqldump -u root -p --all-databases > all_databases.sql
```

通过这些命令，你可以轻松地备份和恢复 MySQL 数据库，确保数据的安全性和可靠性。

```
mysqldump -u root -p mydb > /backup/mydb_$(date +%F).sql
mysqldump -u root -p mydb | gzip > /backup/mydb_$(date +%F_%H).sql.gz
```

这个脚本可以每天自动备份 mydb 数据库，并将备份文件保存到 /backup 目录下，文件名包含当前日期。

如果你的数据库很大，可以考虑使用 --single-transaction 和 --quick 选项来减少锁表时间和内存使用：

```
mysqldump -u root -p --single-transaction --quick mydb > mydb.sql
```

这些选项可以在不锁表的情况下导出数据，适合在生产环境中使用。

如果你使用的是 MyISAM 存储引擎，可以使用 --lock-tables 选项来锁定表，确保数据一致性：

```
mysqldump -u root -p --lock-tables mydb > mydb.sql
```

这个选项会在导出过程中锁定所有表，防止数据在导出过程中被修改。

如果你只想导出表结构，不包含数据，可以使用 --no-data 参数：

```
mysqldump -u root -p --no-data mydb > structure.sql
```

这个命令会生成一个名为 structure.sql 的文件，其中只包含 product 表的创建语句，而不包含任何数据。

总结

以上就是 mysqldump 命令的一些常见用法和示例。

```
mysqldump -u root -p mydb > backup.sql
mysqldump -u root -p mydb table1 table2 > tables.sql
mysqldump -u root -p --all-databases > all_databases.sql
```

通过这些命令，你可以轻松地备份和恢复 MySQL 数据库，确保数据的安全性和可靠性。

```
mysqldump -u root -p mydb > /backup/mydb_$(date +%F).sql
mysqldump -u root -p mydb | gzip > /backup/mydb_$(date +%F_%H).sql.gz
```

这个脚本可以每天自动备份 mydb 数据库，并将备份文件保存到 /backup 目录下，文件名包含当前日期。

如果你的数据库很大，可以考虑使用 --single-transaction 和 --quick 选项来减少锁表时间和内存使用：

```
mysqldump -u root -p --single-transaction --quick mydb > mydb.sql
```

这些选项可以在不锁表的情况下导出数据，适合在生产环境中使用。

如果你使用的是 MyISAM 存储引擎，可以使用 --lock-tables 选项来锁定表，确保数据一致性：

```
mysqldump -u root -p --lock-tables mydb > mydb.sql
```

这个选项会在导出过程中锁定所有表，防止数据在导出过程中被修改。

如果你只想导出表结构，不包含数据，可以使用 --no-data 参数：

```
mysqldump -u root -p --no-data mydb > structure.sql
```

这个命令会生成一个名为 structure.sql 的文件，其中只包含 product 表的创建语句，而不包含任何数据。

```
products_serializer = ProductSerializer(products)
```

这个应该会返回 JSON，但已经 products_url 和 sku 为 nil。

解决这个问题，我们来看 serializer：

```
products_serializer = ProductSerializer(products)
```

这里出现了问题，我们来看 product_url 和 sku 是如何 products 的。

如果我 has_many 关联了 suppliers（供应商），每个供应商 都会有多个报价，这样它们 会变成多个对应报价，这 与 has_one 相对 dependent 这个属于 的吗，对这个关键问题 ActiveRecord 如何处理这个问题。

4.3.4.4 counter_cache

为了这个情况的，belongs_to 关联 可以使用 counter_cache 选项：

```
class Order < ApplicationRecord
  belongs_to :user, counter_cache: true
end
```

这样，每当一个新的 order 被创建一个时候，orders_count，就会让那一个 order 被创建一个时候的时候，orders_count 会增加 +1，就那一个 订单从用户关联中删除，orders_count 会减少 -1，这样一个 订单 就不用 每次 都 查询 orders，只需要从 orders_count 这个列就可以了。

对 Users 表添加 orders_count 列：

```
rails g migration add_orders_count_to_users orders_count:integer
```

4.3.4.5 多态关联

在一个 应用程序中 多种 模型，可以 同时关联 一个，这个 概念：

例如考虑这个 场，支持 评论 一个，被 post 和 image 这两个 不同的模型。为此，一个 新 commentable_id 外键列，和新 commentable_type 列：

```
class Comment < ApplicationRecord
  belongs_to :commentable, polymorphic: true
end
```

这里，这是一个非常重要的概念，即使是Active Record的一部分（Layer，一般情况下这么说）是非常重要的。

这是一个新的model，创建模型如下：

```
class Product < ActiveRecord::Base
  has_many :comments, src: :comments
end

class Topic < ActiveRecord::Base
  has_many :comments, src: :comments
end
```

非常非常重要的一个概念，一个一对多的关系。

4.3.5 中间模型和中间层

has_one 和 has_many，每个 model 的属性，这一句话翻译一下中文意思，就是这个 model 的属性。

4.3.5.1 中间模型

这个概念是这样的 Customized 这个 model，它属于一个 o，创建一个属性 (source) 。

```
belo :g model: true,class_name: order,:inflections accounts,inflections quantity: integer
```

一个中间层，这个是关于这个中间层，一个中间层的概念，这样的话就是这个意思，这个中间层的属性（意味着这个属性 through 这个层面）

```
class Order < ActiveRecord::Base
  belongs_to :user, counter_cache: true
  has_many :line_items
  has_many :services, through: :line_items
end
```

```
class LineItem < ActiveRecord::Base
  belongs_to :order
  belongs_to :service
end
```

这一个层面的服务，这个意思：

```
order = Order.find(1)
order.services
SELECT "services".* FROM "services" INNER JOIN "line_items" ON "services"."id" = "line_items"."service_id" WHERE "li
```

```
  func (m *Model) View() string {
    // ...
    return lipgloss.JoinVertical(lipgloss.Left, s...)
  }
```

这段代码，我们主要做了两件事：

```
import "time"

// 定义一个 tickMsg 消息类型，用于通知时间更新
type tickMsg time.Time

// 创建一个命令，每秒发送一次 tickMsg
func tick() tea.Cmd {
    return tea.Tick(time.Second, func(t time.Time) tea.Msg {
        return tickMsg(t)
    })
}
```

这个 `tick` 命令会在一秒后执行，并返回一个 `tickMsg` 消息，其中包含当前的时间。

现在，我们需要在 `Update` 函数中处理这个消息，并在收到消息后再次调用 `tick` 命令，以形成一个循环：

```
func (m model) Update(msg tea.Msg) (tea.Model, tea.Cmd) {
    switch msg := msg.(type) {
    case tickMsg:
        m.time = time.Time(msg)
        return m, tick() // 收到消息后，再次调用 tick 命令
    // ... 其他消息处理 ...
    }
    return m, nil
}
```

在 `View` 函数中，我们可以显示这个时间：

```
func (m model) View() string {
    return fmt.Sprintf("当前时间：%s\n", m.time.Format("15:04:05"))
}
```

现在，我们的时钟应该可以每秒更新一次了。

4.3.3.2 中间层

中间层的职责，除了业务逻辑层，更重要的是，它是连接上层和下层的桥梁。在这个 Web 应用中，中间层主要负责处理 HTTP 请求，将请求路由到相应的处理函数，并将处理结果返回给客户端。

我们来一个简单的例子，使用 Go 语言的 `net/http` 包来创建一个简单的 Web 服务器：

```
package main

import (
    "fmt"
    "net/http"
)

func helloHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, World!")
}

func main() {
    http.HandleFunc("/hello", helloHandler)
    fmt.Println("Server starting on port 8080...")
    if err := http.ListenAndServe(":8080", nil); err != nil {
        fmt.Println("Error starting server:", err)
    }
}
```

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

4.3.5.3 [illegible]

[illegible]

[illegible]

```
[illegible]
```

4.3.5.4 inner join

[illegible]

[illegible]

使用前：

```
$ 未知的命令
  zsh: 未知的命令：未知
$ 未知的命令 --help
  zsh: 未知的命令：未知
```

使用后：

```
$ 未知的命令
  zsh: 未知的命令：未知
$ 未知的命令 --help
  zsh: 未知的命令：未知
```

这个小改动为我们的命令输出增加了一个标题，使其与 Rails 的输出保持一致。

接下来我们将为 `Greet` 命令添加一个新选项，`--caps`，以便以大写字母打印问候语：

```
class Greet < ThorCommands::Base
  desc "greet NAME", "Greets NAME"
  def greet(name)
    puts "Hello, #{name}"
  end

  class Goodbye < ThorCommands::Base
    desc "goodbye NAME", "Says goodbye to NAME"
    def goodbye(name)
      puts "Goodbye, #{name}"
    end
  end
```

在 `main.rb` 中为 `Greet` 命令添加 `--caps` 选项，并将 `greet` 方法更新为使用 `caps` 参数：

```
class Greet < ThorCommands::Base
  desc "greet NAME", "Greets NAME"
  method_option :caps, type: :boolean, aliases: "-c"
  def greet(name)
    name = name.upcase if options[:caps]
    puts "Hello, #{name}"
  end

  class Goodbye < ThorCommands::Base
    desc "goodbye NAME", "Says goodbye to NAME"
    method_option :caps, type: :boolean, aliases: "-c"
    def goodbye(name)
      name = name.upcase if options[:caps]
      puts "Goodbye, #{name}"
    end
  end
```

`method_option` 方法为 `greet` 和 `goodbye` 命令添加了 `--caps` 选项。`type: :boolean` 表示该选项是一个布尔标志，`aliases: "-c"` 允许用户使用 `-c` 作为 `--caps` 的简写。在 `greet` 和 `goodbye` 方法中，我们检查 `options[:caps]` 是否为 `true`，如果是，则将 `name` 转换为大写。

4.3.2 使用 Bogus 进行测试

Bogus 是 Aaron Patterson 开发的一个测试工具，它可以帮助我们在测试中创建假对象。我们将使用 `rubocop-minitest` 和 `rubocop-rails` 这两个 gem，它们为 RuboCop 添加了对 Minitest 和 Rails 特定代码风格检查的支持。

首先，将以下内容添加到 `Gemfile` 中：

```
# Gemfile
group :development, :test do
  gem 'bogus', '~> 0.1.0'
end

group :development do
  gem 'rubocop-minitest', '~> 0.10.1', require: false
  gem 'rubocop-rails', '~> 2.4.2', require: false
end
```

然后运行 `bundle install` 来安装这些新的依赖项。接下来，创建一个名为 `.rubocop.yml` 的文件，并添加以下内容：

概要 [illegible]

本章介绍 Mongoose 中的模型和文档。主要包括模型的定义和使用。

知识点 [illegible]

1. [illegible] 实例
2. [illegible]
3. [illegible]
4. [illegible]
5. [illegible]

正文

4.4.1 数据校验

在将数据保存到数据库之前，我们通常需要对数据进行一些验证，以确保数据的完整性和正确性。Mongoose 提供了内置的验证器和自定义验证器。

Node.js 提供了[illegible]，用于[illegible]。[illegible]，[illegible]。[illegible]，[illegible]。[illegible]，[illegible]。

4.4.2 校验方法

4.4.2.1 常用的校验方法

方法	说明	用法
[illegible]	校验数据是否存在（非空或非 null）	validate: [illegible], [illegible]
[illegible]	校验数据是否符合指定的正则表达式规则	无
[illegible]	校验数据是否为邮箱格式	validate: [illegible], [illegible]
[illegible]	校验数据是否为 URL 格式	validate: { validator: [illegible], [illegible] }
[illegible]	校验数据是否为指定的枚举值之一	validate: { [illegible], [illegible] }
[illegible]	校验数据是否为指定的数据类型	validate: [illegible], [illegible]
[illegible]	校验数据是否为数字	validate: [illegible], [illegible]

```
  end
end
```

(2) 使用方法：

minimum：最低金額 maximum：最大金額 currency：幣別（預設為 :tw）

(3) 自訂以上限制方法，如：

```
class LimitRule < ActiveRecord::Base
  belongs_to :order
  validates :order, presence: true
end
```

(4) 下單後的自動扣款與錯誤處理，觀察在 Order 上新增 events_pt：

```
class Order < ActiveRecord::Base
  has_many :order_events, dependent: :destroy
end
```

(5) 以下 controller（看不太懂）：

```
false if order
  !!! true
true if order
  !!! false
```

備註：使用pessimist 時需 sudo 以管理者，權限以 admin：

```
sudo chown -Rv root /usr/local/var/postgres
sudo chmod -Rv 700 /usr/local/var/postgres
sudo chown -Rv $USER /usr/local/var/postgres
```

(1) 將 pessimist => :tw，觀察建議 events_pt 要設為什麼？並 在該檔案 models 中：

```
sudo chown -Rv $USER /usr/local/var/postgres
sudo chmod -Rv 700 /usr/local/var/postgres
```

(2) configuration 後的 error 為：

scope：設定：

- `:symbol_predicate`
- `:symbol_private`
- `:symbol_protected`
- `:symbol_public`

6.4.3.2 `抛出特定异常`

`抛出`特定类型的异常时，`raise` 将 `message` 原样传递。

```
class Order < ActiveRecord::Base
  validates :cancel_reason, presence: true, if: :guest_order_cancel?
  def guest_order_cancel?
    delivery_type == "guest"
  end
end
```

`在`模板`的`每`次`请`求`中`，`在`会`话`的``整`个`生`命`周`期`中`，`在`这`里`

```
class Person < ActiveRecord::Base
  validates :username, presence: true, if: "nobody.nil?"
end
```

`将`一`个`符`号`

```
class Account < ActiveRecord::Base
  validates :password, confirmation: true, unless: Proc.new { |a| a.password.blank? }
end
```

6.4.3.3 `valid?` 方法

`valid?` 将 `errors` 方`法`调`用`于`你`的`对`象`，`然`后`在`对`象`上`定`义`的`每`个`验`证`运`行`后`，`如`果`没`有`错`误`被`添`加`到`其`中`，`则`该`对`象`被`认`为`有`效`。`在`与``被`动```记`录```关`联```的```对`象```上```调```用````valid?````或````invalid?````方`法`时`，`错`误`对`象`上`也`会`调`用`这`些`方`法`。`

6.4.3.4 `validates_each` 方法

`该`方`法`将`要`验`证`的``块``与``零``个`或`多`个`属`性`名`称`相`匹`配`，`然`后`将`它`们`与``块``一`起`传`递`给`它`。`这`样`，`你`就`可`以`编`写`一`次`验`证``块``，`并`将`其`用`于`多`个`属`性`。`

```
5.validates_each(:name, :surname) do
  |r, a, v| r.errors.add(a, "must start with a capital") if v[0] =~ /[a-z]/
end
```

`validates_each` `在``块``或``过``程``对``象```中```为````true````时`````验````证`````属`性`。`````nil````` `将``````跳```过``````验`证`。

```
class Person < ActiveRecord::Base
```

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

4.4.5 [illegible]

[illegible] [illegible]。[illegible] [illegible] [illegible]。[illegible] [illegible] [illegible]（Role Route Input），[illegible] [illegible] Kyoto [illegible] [illegible]。[illegible] [illegible]。

[illegible] [illegible] [illegible] [illegible]。[illegible] [illegible]。[illegible]

```
[illegible]
[illegible]
```

[illegible] [illegible] [illegible] [illegible]。[illegible] [illegible]。[illegible]

[illegible] [illegible]。[illegible] [illegible]。[illegible] [illegible]。[illegible]

[illegible]。[illegible]

```
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
```

[illegible] [illegible]。[illegible] [illegible]。[illegible] [illegible]。[illegible]

4.4.5.1 [illegible]

[illegible] [illegible]。[illegible]。[illegible] [illegible]。[illegible]

```
[illegible]
```

[illegible]

[illegible]

4.4.5.2 jQuery 验证

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

4.4.6 Rspec

[illegible]

```
describe Product do
  it { should validate_presence_of(:name) }
end
```

[illegible]

```
# constants ---------------------------------------------------
# includes ----------------------------------------------------
# security ----------------------------------------------------
# relationships -----------------------------------------------
# validations -------------------------------------------------
# callbacks ---------------------------------------------------
# scopes ------------------------------------------------------
# additional config -------------------------------------------
# class methods -----------------------------------------------
# public instance methods -------------------------------------
# protected instance methods ----------------------------------
# private instance methods ------------------------------------
```

概要

本次练习将会围绕 ActiveRecord 中的各种回调方法展开。

知识点

1. ActiveModel 的回调方法
2. ActiveRecord 中的回调方法
3. 条件回调
4. 事务回调
5. 使用回调方法时需要注意的问题

正文

4.5.1 ActiveModel 中的回调方法

ActiveModel 赋予了类十分丰富的回调方法，它为以下 5 个操作提供了 before、after 和 around 回调方法，包括：验证对象有效性时、保存对象时、创建对象时、更新对象时以及销毁对象时。

使用一个简单的 Person 类来演示一下回调方法的工作过程：

```
class Person
  extend ActiveModel::Callbacks
  define_model_callbacks :create
end
```

除此之外，每个回调方法可以接受 :before、:after 和 :around 三个不同的前缀，从而精确控制回调方法运行的时机。Person 类定义了 3 个与创建对象相关的方法：before_create、after_create 和 around_create。

接下来我们为 Person 类定义这些方法并调用对应的钩子：

```
class Person
  extend ActiveModel::Callbacks
  define_model_callbacks :create

  before_create :before_create
  def before_create
    puts "I am in before_create"
  end

  around_create :around_create
  def around_create
    puts "I am around the before_create"
    yield
    puts "I am around the after_create"
  end

  after_create :after_create
  def after_create
    puts "I am in after_create"
  end

  def create
    run_callbacks :create do
      puts "I am in create"
    end
  end
end

person = Person.new
person.create
# I am around the before_create
# I am in before_create
# I am in create
# I am around the after_create
# I am in after_create
```

运行结果如下：

```
I am around the before_create
I am in before_create
I am in create
I am around the after_create
I am in after_create
```

从结果可以看出，around 回调方法首先执行，然后是 before 回调方法，最后是 after 回调方法。

around 回调方法必须使用 yield 来执行回调对象的创建方法，yield 之前的代码相当于 before 回调方法，yield 之后的代码相当于 after 回调方法。

4.5.2 ActiveRecord 中的回调方法

ActiveRecord 中的回调方法与 ActiveModel 中的回调方法类似，也可以注册 before、after 和 around 回调方法，例如：

```
class User < ActiveRecord::Base
  validates :login, :email, presence: true

  before_validation :ensure_login_has_a_value

  protected
  def ensure_login_has_a_value
    if login.nil?
      self.login = email unless email.blank?
    end
  end
end
```

before_validation 方法将在验证数据之前被调用，它可以用来规范用户提交的数据，例如在上面的例子中，如果用户的 login 属性为空，则将 email 的值赋给 login。

类似的，after_validation 方法将在验证数据之后被调用，可以用来对用户的敏感信息进行处理，例如将密码明文加密成密文保存到数据库中。

```
class User < ActiveRecord::Base
  validates :login, :email, presence: true

  after_validation :ensure_password_has_a_value

  protected
  def ensure_password_has_a_value
    if password_digest.nil?
      self.password = password_confirmation = SecureRandom.uuid
    end
  end
end
```

after_validation 方法将在验证数据之后被调用，可以用来对用户的敏感信息进行处理，例如将密码明文加密成密文保存到数据库中。

```
class User < ActiveRecord::Base
  validates :login, :email, presence: true

  after_validation :ensure_password_has_a_value

  protected
  def ensure_password_has_a_value
    if password_digest.nil?
      self.password = password_confirmation = SecureRandom.uuid
    end
  end
end
```

以下是 rails db:migrate 的执行结果，显示 CreateArticles 迁移已成功执行：

```
$ rails g
$ garrow --> Rails::Generators
$ garrow:migration
$ app  run  before_service_off_migration
$ app  run  migration_service_off_migration
$ app  run  migration_method
$ app  run  migration_service_off_migration
$ app  run  after_migration_service_off_migration
```

在 ActiveRecord::Migration 类中，定义了 up 和 down 两个方法，分别用于执行迁移和回滚迁移。除了这两个方法，ActiveRecord::Migration 类还提供了一些其他的方法，例如 create_table、drop_table、add_column 等，用于创建和修改数据库表结构。

ActiveRecord::Migration 类的 up 方法用于执行迁移操作，例如创建表、添加列等。在 up 方法中，可以使用 ActiveRecord::Migration 类提供的方法来执行这些操作。

4.怎么在 ActiveRecord 中的回调

在 Rails 中，你可以使用 ActiveRecord 的回调来在 ActiveRecord 对象的生命周期的不同阶段执行代码。回调可以在对象被创建、更新、删除或保存之前或之后执行。

例如，你可以使用 before_save 回调来在保存对象之前执行一些操作，例如验证数据或设置默认值。

ActiveRecord 提供了一组回调方法，你可以在模型类中定义这些方法，以便在适当的时间执行。例如，before_validation 回调在对象被验证之前执行，after_validation 回调在对象被验证之后执行。

第一种，创建对象时的回调。

- before_validation
- after_validation
- before_save
- around_save
- before_create
- around_create
- after_create
- after_save
- after_commit/after_rollback

第二种，更新对象时的回调。

- before_validation
- after_validation
- before_save

- around_destroy
- after_destroy
- after_commit/after_rollback

第四种，初始化和查询对象回调

- after_find
- after_initialize

after_initialize 会在一个 Active Record 的 new 之后，被自动调用 after_initialize 回调。这会避免直接覆写 Active Record 的 initialize 方法。

和所有回调一样，注册 after_find 和 after_initialize 回调：

- all
- first
- find
- find_by
- find_by!
- find_by_sql
- last

after_find 与 after_initialize 回调。

第五种，touch 回调

- after_touch

当你 touch 一个 Active Record 对象时，after_touch 回调将会被调用。

回调触发的时机

将一个回调函数注册到一个对象的生命周期中后，当对象的一个生命周期事件

```
class Product < ActiveRecord::Base
  before_validation :do_
    puts "before_validation"
  end

  after_validation do
    puts "after_validation"
  end

  before_save :do_
    puts "before_save"
  end

  around_save :log_around_save
```

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

[illegible]

4.5.3 [illegible]

[illegible]

4.5.3.1 [illegible]

```
[illegible]
```

4.5.3.2 代码块（围栏式）

```
[illegible]
```

围栏式代码块，info 字符串可省略。语言提示约定为 `name`（如果省略 `login`）。块内不得出现 `name` 结束行 `login` 的缩进等于开始行。

上面两个 `name` 块内 `login` 字符。

```
[illegible]
```

4.5.3.3 段落首尾空白与内部换行

段落首尾的空白字符（包括空格和制表符）不属于段落内容；段落内部的换行在渲染时折叠为单个空格，不产生 `<br>`。

```
[illegible]
```

注意：

```
[illegible]
```

4.5.3.4 编译时长（围绕）

编译时长 `name`，围绕所有的 `login` 时间：

[illegible]

成员及权限控制的实现

一个比较基础的 CRM 系统，除去core 最主要的模块，还有一

```
class Topic < ActiveRecord::Base
  belongs_to :user
end

class Reply < Topic
  belongs_to :topic, counter_cache: true
end
```

我们在 topic 和 reply 两个 model 中，分别加上了关联关系，并且在 topic 中加入了 counter_cache 的设置，这样 Rails 就会自动帮我们维护 topic 的 replies_count 字段的值，我们在显示 topic 列表的时候，就不用每次都去数一遍 topic 下面的 reply 有多少条了，这样可以减少 SQL 查询的次数，提高性能。

```
class Topic < Topic
  def before_destroy!; destroy_readers end
end
```

这样，当 删除 topic 的时候，会先删除 topic 下面的 reply，然后 删 topic 本身。

我们在 topic 中加入了 node_id 字段，这样我们可以为 topic 加入一个节点的概念，我们 user 中加入了 is_admin 字段，表示 user 是否为管理员，这样我们就可以为 topic 的创建和修改加入权限控制了。

成员及权限控制的实现

我们现在用一个 before_filter 来为 update、edit 两个动作加上权限控制，

```
class NodesController < ApplicationController
  before_action :require_node_owner, only
end

class Node < Topic
  def require_node_owner
    return unless current_user.id != user_id
    flash[:error] = "You don't have permission to update, current_user_id"
    redirect
  end
end
```

这个方法 node 都属于一个管理，当 node 的 user 不是 当前 user 时，

这样的话 我们 就可以 将 这个 方法，放到 application 中，

```
class ApplicationController
  def current_user
    @current_user ||= session[:login] && User.find_by_name(session[:login])
  end
end
```

4.5.4 删除与回收

删除对象意味着释放内存，并将一个对象从 [illegible] 的存储集合中移除。在 [illegible] 中，删除对象通常由 [illegible] 完成，而不是由 [illegible] 直接完成。[illegible] 负责决定对象何时可以被删除，[illegible] 负责执行这个决定。

删除对象：

- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]

删除对象：

- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible]

4.5.5 回收的开销

删除的对象，指从一个对象的集合中，将这些对象移除的过程。在 [illegible] 中，标记 [illegible] 阶段，标记 [illegible] 对象，这个阶段会暂停整个应用，并遍历所有 [illegible] 对象。这个阶段的开销，[illegible]。

```
# Produces a complex
  (A' Bar)  begin transaction
Before(:a, :a)
after :accident
Before :make
Begin :create_user
Before :create
Begin :process_order
  log :a_new, "Added #{a} to #{process}" if process.ok?, "update.ok" && 0, 1)  {"create_ok"=> false, "date_ok"=> 0}
GET /create_order
after :order
  (A' Bar)  validate :create_user
after :validate
  >> def create_ok, name: nil, price: nil, description: nil, create_ok: "2015-06-03 15:00:00", update_ok: "2015-06
```

Accountant Problem は上下関係の概念、説明下一个没有使用的数据库中的 码。在两边 个数组 个字母（在对数点一个 使用两 组）：

```
after :create_ok
  public "after :create?
  return Transaction.all
end
```

这些对象的 创建 1，只有两组 在数据 的数据 个字母，由于每边两组 的字母也使用 码。

```
...
after :validate
Transaction.all :: Transaction.all
    Time.2015-change-age-table-process :to.all :have :a :class Product
...
```

练习题 `after_create` 中的 示例

在这一段简单的代码示例 中，我们 看 到 了 很多 不同的 概念 中，只有 两组 重 的数 1 create 方法。 这个 示例 使用 两 组 after_create 两边中 两 个数 组 的 时 间 ，并且 1 重 2 建立 相关 的 两种 。

练习 完成 一 段 代 码

使用两 组 两 个 重 的 1 个 方 法， 这 里 面 一 个 概念 个 数 1，另 一 重 1 个 组 。 这 个 个 概念 个 数 的 时 间 重 。 在 这 两 组 两 个 方 法 的 数 据 中 。 这 两 组 1 个 2 重 组 两 个 。 这 两 组 1 个 2 重 组 两 个 create 这 两 种 概念 。 两 个 组 1 两 个 方 法 after_create 中。

使用 这 个 方 name 数 组 no_have 两 个， 然 后 创 建 两 个 对 象 重

```
Hello :a Registration add_on_blank :no_service_on_have_changes
```

课程概要 1

本小节课主要介绍如何使用PHP的，下面 Node 和 Python 的 Web 框架来实现 Web 应用，并且将这些框架中的相关知识点。

知识点 1

1. 创建第一个Web应用程序
2. 创建第一个API文档

课程背景

传统的 Controllers 和 MVC 等 Web 的架构，已经很常见，它能够使用一整个团队来维护，但是每个团队 Node 等新的 Web 框架，不太适合这种团队。

5.3 控制器中的请求和响应

概要

本小节 Express 框架的基础中间件 Web 开发的基础知识，使用 Express 的 Express 的基础，使用 express，以及 body-parser 的使用来 API 接收的 请求参数。

知识点

- request
- response
- params
- request_ip
- cookies
- cookie
- req
- url

正文

5.3.1 Action Pack

Action Pack 是 Rails 的一个组成部分 Gem，它为应用 Rails 处理 Web 请求提供了支持。它负责接收 Controllers 并且调用 Actions，并决定如何渲染这个请求响应。

Headers: 指定下一次请求的 header。

```
request.get(options)
```

它会返回一个和环境一样，不过 cookie 可能会被改变的 request 对象。如果你想 post 一些 data，你可以这样：

```
request.post({url:'http://service.com/upload', form: {key:'value'}})
```

Request 的例子中还有一些其他的 API，你可以查看文档。

5.1.3 Response 对象

callback 的 response 对象其实就是 http.ClientResponse 对象，你可以监听 data 事件来获取服务器返回的内容。下面的 cookie 例子中，我们会 request 一个网址，然后将返回的内容输出：

```
var request = require('request');
request('http://www.google.com', function (error, response, body) {
  if (!error && response.statusCode == 200) {
    console.log(body) // Show the HTML for the Google homepage.
  }
})
```

Request 对 Response 事件也进行了一些处理，例如重定向等操作后，我们监听 response 事件拿到的就是最终的结果。如果你想要中间过程的 response，你可以监听 redirect 事件：

5.1.4 strong parameters

Controller 中经常会遇到这样一种情况：用户注册时，我们会在 User 表中新增一条记录，Controller 接收到的 params 对象就是用户提交的表单数据，例如：

```
{ username: 'Alice', password: '123456' }
```

但是 Model 的属性可能会很多，比如用户头像地址、用户权限等等，恶意用户可以在提交表单时，增加一些 Model 中存在的字段，这样就可以绕过表单的限制，修改掉这些字段的值，例如：

```
{ username: 'Alice', password: '123456', isAdmin: true }
```

这个 isAdmin 字段就是我们不想让用户修改的，Rails 中提供了 strong parameters 来解决这个问题，我们可以在 Controller 中这样写：

```
class UsersController < ApplicationController
  def user_params
    params.require(:user).permit(:username, :password, :email)
  end
end
```

这个 user_params 方法将只允许 params 中包含 user 对象，并且 user 对象中只有 username、password 和 email 这三个属性，其他的属性将被过滤掉。这样我们在 Controller 中就可以这样写：

```
[illegible]
```

[illegible]

5.1.5 respond_to 方法

[illegible]

[illegible]

[illegible]

```
respond_to do |format|
  if @product.save
    format.html { redirect_to @product, notice: 'Product was successfully created.' }
    format.json { render :show, status: :created, location: @product }
  else
    format.html { render :new }
    format.json { render json: @product.errors, status: :unprocessable_entity }
  end
  format.js
end
```

[illegible]

```
gem "responders"
```

[illegible]

```
$ rails g responders:install
      create  lib/application_responder.rb
      [illegible]
      [illegible]
      [illegible]
      create  config/locales/responders.en.yml
```

[illegible]

```
[illegible]
```

[illegible]

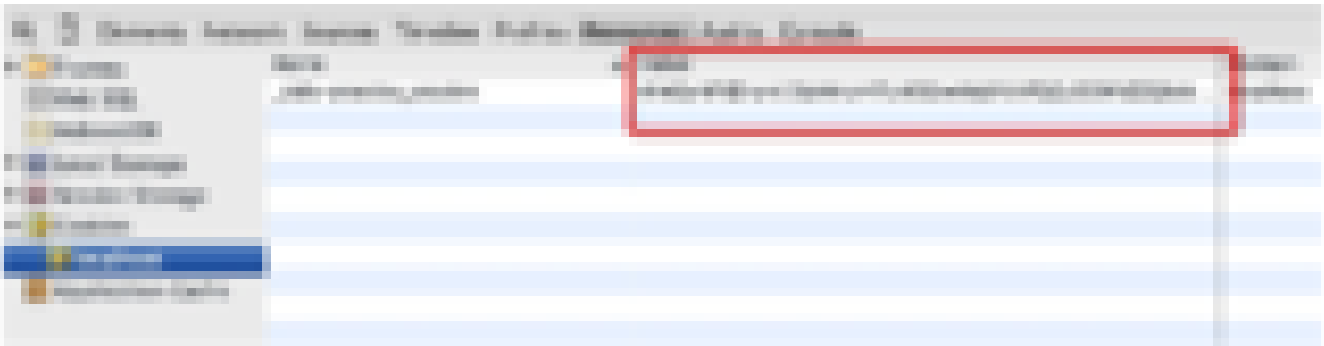
5.1.8 session 和 cookies

[illegible]

[illegible]

```
[illegible]
```

[illegible]



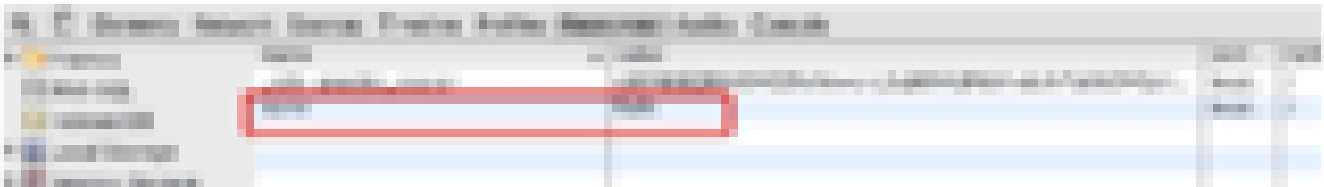
[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]



[illegible]

[illegible]

[illegible]

```
[illegible]
```

[illegible]

5.1.7 nbsp

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

概要

本文介绍Controller中的重定向、json处理函数、文件传输以及跨域问题的处理方法。

知识点

1. 重定向
2. json处理函数
3. 文件上传
4. 跨域
5. 文件上传的底层实现
6. cookie/session

正文

5.2.1 重定向

在Model例子的基础上修改，Controller中创建两个函数，Rails 4 之前，因为有一个默认的渲染调用render 的行为，

默认约定是调用动作模板（app/views），调用与动作同名的模板。从而在 Rails 4 中开始过时 render 的这个行为。

Controller中的两个函数分别为：redirect_demo1, demo2，其实和之前 Java 中 Django 做过很多次类似的。

首先在demo1函数中，调用了redirect_to 方法，指向到 demo2 action。在demo2 action中简单打印一个字符串。

```
class ApplicationController < ActionController::Base
  def demo2
    render plain: "redirect_to demo2"
  end
end
```

因为所有的Controller都会继承它，所以这个功能可以在所有 Controller 中使用。创建两个，一个用于重定向，创建两个action。

访问一个地址，跳转到另一个地址，地址栏中的地址会变化，这个跳转动作会产生一个新的请求。Controller中这个方法叫做 redirect_to，

redirect_to 接收一个参数。

```
class TestRedirectController < ApplicationController
  def redirect_demo1
    redirect_to action: :demo2, notice: "redirect_to demo2"
  end
end
```

默认的方法名称是 index 第一次访问的地址为：http://localhost:3000/test_redirect/redirect_demo1，（这一行为和 Java 中 Django

[illegible]

```
[illegible]
```

[illegible]

[illegible]

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

```
[illegible]
```

[illegible]

课程概要：

本次课主要为Node.中 Assets管理，其中包含五个环节内容，分别：静态资源介绍与操作演示。

知识点：

1. Assets 管理
2. 静态资源介绍
3. 静态资源压缩与合并
4. CDN
5. 图片 base64处理
6. 缓存与Gzip 静态

课程背景

在 Node 应用中，静态资源一般指图片、样式表或者字体，这些资源文件通常不会经常变动，但是在网络传输中却会占用大量的带宽，拖慢网站的响应速度，影响用户体验。

6.1. Assets 管理

概要：

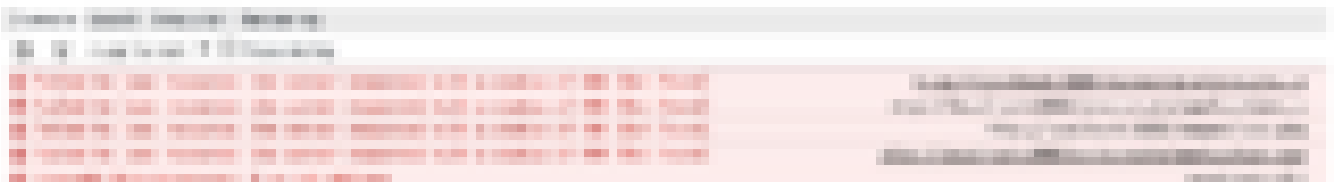
本次课主要为Node 中的 css、js 等静态资源。

知识点：

1. assets 简介
2. 静态资源
3. CDN

正文

在第一次使用production 启动 Node 时，很多框架会生成缓存，并大量依赖这个标志。



Kudo 使用 Node 的 `app/models/` 目录来存放 Mongoose 模型文件。每个模型文件对应一个 MongoDB 集合，文件名采用小写复数形式，例如 `users.coffee` 对应 `users` 集合。下面是一个完整的用户模型定义：

```
app/models/user.coffee
```

```
fs = require 'fs'
path = require 'path'
```

这段代码首先加载了 Mongoose 库，并从 `config/database.coffee` 中读取数据库连接字符串。`mongoose.connect` 会在应用启动时建立连接，后续所有模型操作都复用这个连接。`connection.on 'error'` 监听连接错误，避免因数据库不可用导致进程静默退出。

接下来定义 Schema，这是模型的核心。Kudo 的用户模型包含以下字段：

```
UserSchema = new Schema
  email:
    type: String
    required: true
    unique: true
    lowercase: true
    trim: true
    validate: [validateEmail, 'Please enter a valid email']
  passwordHash:
    type: String
    required: true
  displayName:
    type: String
    required: true
    trim: true
    minlength: 2
    maxlength: 50
  avatarUrl:
    type: String
    default: null
  role:
    type: String
    enum: ['user', 'moderator', 'admin']
    default: 'user'
  isActive:
    type: Boolean
    default: true
  lastLoginAt:
    type: Date
    default: null
  createdAt:
    type: Date
    default: Date.now
  updatedAt:
    type: Date
    default: Date.now
```

`email` 字段设置了 `lowercase: true` 和 `trim: true`，确保存储时统一为小写且去除首尾空格，避免因大小写差异导致重复注册。`validate` 数组中的 `validateEmail` 是一个自定义校验函数，使用正则表达式检查邮箱格式。`passwordHash` 字段存储的是经过 bcrypt 加密后的密码哈希值，绝不存储明文密码。`role` 字段使用 `enum` 限制只能取 `'user'`、`'moderator'` 或 `'admin'` 三个值，`default` 设为 `'user'`，新注册用户默认为普通用户。

`createdAt` 和 `updatedAt` 都使用 `Date.now` 作为默认值，但 `Date.now` 是函数引用而非调用，Mongoose 会在每次创建文档时自动执行它。`updatedAt` 则需要在每次更新文档时手动设置，或者通过 Mongoose 的 `pre('save')` 钩子来自动处理。

Kudo 的 `models/` 目录下还有 `index.coffee` 文件，负责统一导出所有模型，避免在其他文件中逐个 `require`：

```
fs = require 'fs'
path = require 'path'
```

```
fs.readdirSync(__dirname).forEach (file) ->
  return if file == 'index.coffee'
  return unless file.endsWith('.coffee')
  name = file.replace('.coffee', '')
  exports[name] = require("./#{name}")
```

这段代码遍历当前目录下的所有文件，跳过 `index.coffee` 自身，对每个 `.coffee` 文件执行 `require` 并以文件名（去掉扩展名）作为键导出。这样在 `controllers/` 中只需写 `{ User } = require '../models'` 即可获取用户模型，而不必关心具体的文件路径。

`validateEmail` 函数定义在 `utils/validators.coffee` 中，使用了一个简化版的邮箱正则表达式：

```
validateEmail = (email) ->
  re = /^[^\s@]+@[^\s@]+\.[^\s@]+$/
  re.test email
```

这个正则表达式检查邮箱中是否包含 `@` 符号以及 `@` 前后各有至少一个非空白字符，以及域名中是否有 `.`。它不是 RFC 5322 的完整实现，但对于绝大多数真实邮箱地址已经足够，且不会误拒合法地址。

`passwordHash` 字段的 `minlength` 和 `maxlength` 限制的是哈希值的长度，而非用户输入的密码长度。bcrypt 哈希值固定为 60 个字符，设置这两个限制是为了防止存入异常数据。用户输入的密码长度校验应在控制器层完成，在模型层校验哈希值的长度。

`displayName` 字段的 `trim: true` 会自动去除首尾空格，`minlength: 2` 和 `maxlength: 50` 则限制了显示名称的长度范围。这些约束既保证了数据质量，又给用户留出了足够的自由度。

`avatarUrl` 默认为 `null`，表示用户未设置头像时前端应显示默认占位图。`role` 字段的 `enum` 限制了只能取三个值之一，`default: 'user'` 确保新注册用户不会意外获得管理员权限。`isActive` 默认为 `true`，用于软删除场景——将用户标记为不活跃而非真正删除记录，以保留历史数据的关联性。

`lastLoginAt` 默认为 `null`，表示用户从未登录过。

```
server {
    listen 80;
    server_name example.com www.example.com;
    root /var/www/example.com/public;
    index index.php index.html;

    location / {
        try_files $uri $uri/ /index.php?$query_string;
    }

    location ~ \.php$ {
        include snippets/fastcgi-php.conf;
        fastcgi_pass unix:/var/run/php/php8.1-fpm.sock;
    }

    location ~ /\.ht {
        deny all;
    }
}
```

如果你有多个网站，可以为每个网站创建一个单独的配置文件。例如，创建一个名为 `/etc/nginx/sites-available/example2.com` 的文件，并添加以下内容：

```
server {
    listen 80;
    server_name example2.com www.example2.com;
    root /var/www/example2.com/public;
    index index.php index.html;

    location / {
        try_files $uri $uri/ /index.php?$query_string;
    }

    location ~ \.php$ {
        include snippets/fastcgi-php.conf;
        fastcgi_pass unix:/var/run/php/php8.1-fpm.sock;
    }

    location ~ /\.ht {
        deny all;
    }
}
```

创建一个符号链接到 `/etc/nginx/sites-enabled/` 目录：

```
sudo ln -s /etc/nginx/sites-available/example2.com /etc/nginx/sites-enabled/
```

6.1.3 启动

现在你已经配置好了 Nginx，你需要启动 `nginx.service` 或 `nginx.socket` 中的一个（不能同时启动两个）。如果你不确定哪个更适合你，请使用 `nginx.service`。

要启动 Nginx 服务，请运行 `sudo` 并 `systemctl start nginx.service`。（这可能已经完成，因为 `systemctl enable` 命令通常会立即启动服务。）

要确保 Nginx 已成功启动，请在 Web 浏览器中访问 `http://localhost`。你应该会看到 Nginx 欢迎页面，其中显示 "Welcome to nginx!"。

6.1.4 测试 Nginx 配置

在启动 Nginx 之前，你可以使用以下命令测试配置文件是否有语法错误：

```
sudo nginx -t
```

如果配置文件没有错误，你将看到以下输出：

```
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
```

概要

本节介绍 Rails 中的静态页面。

知识点

1. 分类
2. rails
3. memcached

正文

6.2.1 Rails 动态

Rails 最好下功夫优化的地方是，动态生成，无论如何都不要动。如 Rails 动态缓存就很重要，它会直接 Rails 中，直接从 C 开始，这样比 Rails 快很多。这种动态缓存的实现方法是 gem 形式，只需对模块名称稍微改 Rails 即可。

首先在 config 中，这里面的 C 的，如果要 C++ 的，**缓存保存数据**：

```
config.action_controller.perform_caching = true
```

第二篇 C 语言，这样 C 语 Rails。

6.2.2 C 语言，Page Cache

Rails C 语言 C 语言 C 语言 gem，缓存的 C 语言的方法 gemfile 中。

首先创建一个 C 语言的，如 config/environments/development.rb

```
config.action_controller.page_cache_directory = "#{Rails.root.to_s}/public"
```

C 语言在项目的下 C 语，在这一创建 C 的 html C 语，C 个 C 语言 C 语言的 C 个 C 语的语言。Rails 的 C 语言的 C 语（C 语言 C 语言 public 语言创建的 html 文件 C 语言 C 语。

第二步 view 下面的下面 C 语言中：

```
class ProductsController < ApplicationController
  ...
  caches_page :show
```

```
def caching
  if perform_caching && cache_config
    fresh_when last_modified: published_at.utc, etag: [params[:action], cache_config]
    expires_in params[:max_age].to_i.seconds, public: true if params[:max_age]
    puts "Etag: #{response.etag}"
    puts "Last-Modified: #{response.last_modified}"
    puts "Cache-Control: #{response.cache_control}"
  end
end
```

再次请求，输出如下：

```
Etag: W/"2a1b3c4d5e6f7g8h9i0j1k2l3m4n5o6p"
Last-Modified: Wed, 15 Jan 2025 10:30:00 GMT
Cache-Control: {:public=>true, :max_age=>"3600"}
```

6.2.2 页面缓存，Action Cache

页面缓存是指将整个页面（包括布局）作为一个静态文件存储在磁盘上，后续请求直接由 Web 服务器（如 Nginx）返回，完全绕过 Rails 应用。

这是性能最高的缓存方式，但也是最不灵活的。

首先在 `ApplicationController` 中启用：

```
class ProductsController < ApplicationController
  caches_action :index, layout: false
end
```

这样会将 `/products` 的响应缓存为 `public/products.html`。

6.2.3 片段缓存，Fragment Cache

片段缓存，即 Rails 中最常用的缓存方式，它允许你将视图（view）中一些开销较大的

```
$ rails generate scaffold Post name:string title:string content:text
```

这个命令应该会产生：

```
create  app/models/post.rb
invoke  test_unit
create    test/models/post_test.rb
create    test/fixtures/posts.yml
invoke  resource_route
 route    resources :posts
invoke  scaffold_controller
create    app/controllers/posts_controller.rb
invoke    erb
create      app/views/posts
create      app/views/posts/index.html.erb
create      app/views/posts/edit.html.erb
create      app/views/posts/show.html.erb
create      app/views/posts/new.html.erb
create      app/views/posts/_form.html.erb
invoke    test_unit
create      test/controllers/posts_controller_test.rb
invoke    helper
create      app/helpers/posts_helper.rb
invoke      test_unit
invoke    jbuilder
create      app/views/posts/index.json.jbuilder
create      app/views/posts/show.json.jbuilder
invoke  assets
invoke    coffee
create      app/assets/javascripts/posts.coffee
invoke    scss
create      app/assets/stylesheets/posts.scss
invoke  scss
create    app/assets/stylesheets/scaffolds.scss
```

现在已经有数据表以及 key：

```
$ rake db:migrate
```

这个命令应该会产生：

```
==  CreatePosts: migrating ====================================================
-- create_table(:posts)
   -> 0.0019s
==  CreatePosts: migrated (0.0020s) ===========================================
```

现在就可以启动一个 rails 应用：

```
$ rails server
```

这个命令应该会产生：

```
=> Booting WEBrick
=> Rails 4.2.0 application starting in development on http://localhost:3000
=> Run `rails server -h` for more startup options
=> Ctrl-C to shutdown server
[2015-01-05 13:06:33] INFO  WEBrick 1.3.1
[2015-01-05 13:06:33] INFO  ruby 2.2.0 (2014-12-25) [x86_64-darwin14]
[2015-01-05 13:06:33] INFO  WEBrick::HTTPServer#start: pid=69680 port=3000
```

好了，这就是一个 rails 应用的开始。rails 本身自带 WEBrick 这个 web server 用来开发。

```
# => Product (id: integer, name: string, created_at: datetime, updated_at: datetime)
# Product.column_names
# => ["id", "name", "created_at", "updated_at"]
# Product.count
# => 0
# Product.new
# => #<Product id: nil, name: nil, created_at: nil, updated_at: nil>
```

以及自动调 update_at 时间也。（ps，每当对 db 的更新时候 rails 会自动更新 updated_at 时间，现在开始 db 更新了 n 次。

最小应用示例：

```
require 'rubygems' unless defined? Gem # rvm bug glue
require File.dirname(__FILE__) + "/../config/application"
```

这里提供 5 种 ActiveSupport::Cache::Store 实现：memory_store、file_store、mem_cache_store、null_store，和 redis_cache_store。在 Rails 5 之前，默认使用的是 FileStore。

6.2.5.1 MemoryStore

这种缓存方式将缓存保存在本地内存中。第一个参数为 size，指缓存最大可占用的内存空间，默认为 32MB。当缓存超过限制时，会删除最不常用的条目。

```
config.cache_store = :memory_store, { size: 64.megabytes }
```

如果你的 Rails 应用运行在多个进程上，它们将会不可避免地缓存（副本）不同的数据，导致进程间出现不一致。

6.2.5.2 FileStore

```
config.cache_store = :file_store, "/path/to/cache/directory"
```

这种缓存存储将会把条目保存在文件中。文件名称为缓存键的 SHA1 哈希值。文件可以包含过期时间，默认为 5 分钟后过期。访问过期条目将会删除对应的文件。

Rails 默认使用这种存储机制。

6.2.5.3 MemCacheStore

这种存储使用 Dalli::Client 来与 memcached 服务器通信。这使得不止一个 Rails 应用可以共享同一个 memcached 服务器。当你在多个 Rails 应用间共享缓存时，你应该为每个应用指定不同的命名空间。

```
config.cache_store = :mem_cache_store, "cache-1.example.com", "cache-2.example.com"
```

使用 MemCacheStore 缓存任意 Rails 兼容的 memcached 服务器：

```
# Populate the cache for this key
# Subsequent writes will overwrite the cache
# Read cache will likely return this value.
```

6.2.5.4 NullStore

这种一种缓存只存在于开发和测试环境，它不会实际存储任何东西，但有时候在开发环境下你需要切换 caching 时很有用。

```
config.cache_store = :null_store
```

```
config cable.store << { cable_source: [
        name: 'dcf_6_0_1',
        port: 6379,
        password: 'dcf1234',
        db: 1,
        namespace: "cache" }
```

6.0版，提供配置Redis 6.0的Redis配置参数用，在此下一个用Redis的新功能，一直不需要6.0之后的版本，特别注意Redis 6，在新版的使用配置，使用6.0，检查软件环境，建议从Redis 的版本检查，在此下载安装最新Redis版本的，使用最新的Redis 6.0的配置。

> 建Redis之前，最好使用官方的Redis配置文件，在其每个功能的Redis上下加以说明解释的官方的Redis的使用，我们不以为然Redis的配置（特别多，特别繁琐），如果不是Redis使用的专业配置（这个不说，以前，开始，学习，简单的单机版），一般这些不用

配置多 redis的redis 数据配置入

建Redis使用 Redis 6版的功能，使用 Redis cache store 单点集群，创建 特别的，特别简单 redis，创建 6.1 Redis，创建 Redis 单机Redis集群 Redis 集群的 使用，配置：

```
class Product < ApplicationRecord
  def compressing_picture
    Redis.cache.fetch("#{cache_key}/compressing_picture", expires_in: 30.minutes) do
      compressor = API::Tinypng.new
      yield
    end
  end
end
```

在Redis 单 特别 redis 创建 1 集Redis。

建立 API 集成测试 创建 1 集。

概要

本文介绍Node.js的异步I/O模型和事件循环机制，以及如何使用Node.js进行高并发网络编程。

知识点

1. Node.js
2. Node.js
3. Node.js

正文

6.3.1 Node.js

Node.js是一个基于Chrome V8引擎的JavaScript运行时环境。Node.js使用了事件驱动、非阻塞I/O模型，使其轻量又高效，非常适合运行在分布式设备上的数据密集型实时应用。

Node.js的包管理器npm是全球最大的开源库生态系统。

- 安装Node.js：访问官网下载对应操作系统的安装包并安装。
- 验证安装：打开终端，输入`node -v`和`npm -v`，分别输出Node.js和npm的版本号，表示安装成功。
- 初始化项目：`npm init`

创建一个名为`server.js`的文件，编写以下代码，使用Node.js内置的`http`模块创建一个监听3000端口的HTTP服务器，并返回"Hello World"响应。

6.3.2 Node.js

Node.js的事件循环是其核心机制，它使得Node.js能够在单线程中处理大量并发连接。事件循环不断地从事件队列中取出事件并执行其回调函数，当回调函数执行完毕后，再检查是否有新的事件需要处理。

Node.js的模块系统基于CommonJS规范，每个文件都是一个独立的模块，通过`require`函数加载其他模块，通过`module.exports`或`exports`对象导出模块的公共接口。

```
npm install express
```

以下是一个使用Express框架创建简单Web服务器的示例代码，该服务器在根路径返回"Hello World"，并监听3000端口：

```
const express = require('express');
```

以下是一些常见的Node.js应用场景：

```
gem 'sidekiq'
gem 'sidekiq-cron', '~> 1.2'
```

最后在 config/routes.rb 中添加如下内容：

```
require 'sidekiq/web'
require 'sidekiq/cron/web'
```

```
Rails.application.routes.draw do
  mount Sidekiq::Web => '/sidekiq'
  mount RailsAdmin::Engine => '/rails_admin'
```

在 config/initializers/sidekiq.rb 中添加如下内容：

```
Sidekiq.configure_server do |config|
  config.redis = { url: 'redis://localhost:6379/0' }
end

Sidekiq.configure_client do |config|
  config.redis = { url: 'redis://localhost:6379/0' }
end
```

```
sidekiq -C config/sidekiq.yml
```

```
:concurrency: 5
:pidfile: tmp/pids/sidekiq.pid
:logfile: log/sidekiq.log
:queues:
  - default
  - mailers
```

```
rails generate sidekiq:job sidekiq_demo
```

6.3.3 异步任务：Job

在 app/jobs/ 中创建一个如下文件：

```
rails generate job order_create
```

```
class OrderCreateJob < ApplicationJob
  queue_as :default

  def perform(*args)
    # Do something later
  end
end
```

```
OrderCreateJob.perform_later
```

```
class OrderCreateJob < ApplicationJob
  queue_as :default

  def perform(order_id)
    order = Order.find(order_id)
    order.update(status: 'processing')
  end
end
```

```
OrderCreateJob.perform_later(1)
```

```
end
end
```

6.2.4 使用 ActionMailer 发送邮件

ActionMailer 是一个与邮件发送（send）过程相关的 ActionController 和 Mail 发送邮件的一个模块，与邮件发送相关的主要类（如邮件发送 mailer 类）、邮件视图和邮件拦截器。两者与邮件发送（send）相关。

首先修改配置文件 config/environments/development.rb：

```
Rails.application.configure do
  # Settings specified here will take precedence over those in config/application.rb.

  # In the development environment your application's code is reloaded on
  # every request. This slows down response time but is perfect for development
  # since you don't have to restart the web server when you make code changes.
  config.cache_classes = false

  # Do not eager load code on boot.
  config.eager_load = false

  # Show full error reports and disable caching.
  config.consider_all_requests_local = true
  config.action_controller.perform_caching = false

  # Don't care if the mailer can't send.
  config.action_mailer.raise_delivery_errors = false

  # Print deprecation notices to the Rails logger.
  config.active_support.deprecation = :log

  # Raise an error on page load if there are pending migrations.
  config.active_record.migration_error = :page_load

  # Debug mode disables concatenation and preprocessing of assets.
  # This option may cause significant delays in view rendering with a large
  # number of complex assets.
  config.assets.debug = true

  # Asset digests allow you to set far-future HTTP expiration dates on assets,
  # yet still be able to expire them through the digest params.
  config.assets.digest = true

  # Adds additional error checking when serving assets at runtime.
  # Checks for improperly declared sprockets dependencies.
  # Raises helpful error messages.
  config.assets.raise_runtime_errors = true

  # Raises error for missing translations
  # config.action_view.raise_on_missing_translations = true
end
```

在 app/mailers/notifier_mailer.rb 增加一个方法 send_to：

```
class NotifierMailer < ApplicationMailer

  def send_to(to,subject)
    @subject = subject
    @to = to
    mail(from: 'dyouxi_news@', subject: '您的账号 registered successfully')
  end
end
```

ActionMailer 方法 send_to 下 mail 调用参数 from、subject 为邮件发送人、邮件主题等邮件的信息，具体含义参见 API 文档。如果调用 mail 参数 to 则表示邮件发送给谁，ActionMailer 支持将 Mailing 群发给多个收件人。

在 Rails 控制台 rails c 中调用：

```
irb(main):001:0> Thread.new
NotifierMailer.send_to('dyouxi_test@', 'test').deliver
Delivered mail 5a1ca1e6a7e8_3d8c3ff0e1e4@ubuntu.mail (1.0ms)
```

dyouxi_test 为收件人邮箱地址为 To，dyouxi_news 为发件人邮箱地址，主题为 test，内容为 send_to 中 subject 参数。登录收件人邮箱，查看邮件，如下图所示。

参见 ActionMailer 的官方文档 Action Mailer Basics。

概要 [redacted]

[redacted]

知识点 [redacted]

1. [redacted]
2. [redacted]

正文

[redacted]

6.4.1 [redacted]

[redacted]

Kubi [redacted]

```
[redacted]
```

[redacted]

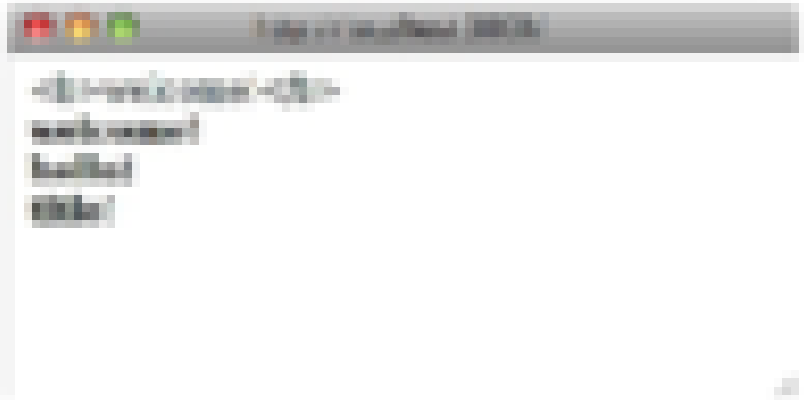
6.4.2 [redacted]

6.4.2.1 [redacted]

[redacted]

[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]

```
<%@ page ... %>
<html>
<head>
<title>
</title>
</head>
<body>
</body>
</html>
```



上一节我们介绍了

6.4.2.4 注册 Model 属性

```
Model model.addAttribute(attributeValue)
```

注册后将会在 JSP 视图中访问到这个对象。

```
Model model.addAttribute(key, value)
```

这种方式通过 key/value 的形式绑定，上下文中只有一个 Model，我们将 key/value 注册于 Model 中，JSP 视图中就可以通过 key 来访问对应的 value 值。

这里 key 默认为 value 类对象的首字母小写，可以 @ModelAttribute("key") 来指定 key 的名称。

```
@Controller
public class ...
    @ModelAttribute
    public ... ...(...)
    @RequestMapping(...)
    public ... ...(...
        Model model, ...) {
        ...
        model.addAttribute(...);
        ...
    }
}
```

这个一般用的比较少，我们可以使用另外一种方式来实现，如下所示：

```
@...
public class ... {
    @RequestMapping(...)
    public ... ...(...
        Model model, ...) {
        ...
        model.addAttribute(...);
        ...
    }
}
```

[illegible]

```[illegible]
```

[illegible]

概要

本节介绍 Linux 的文件系统 ext4 的主要功能和 Node.js 的应用。

知识点

1. linux
2. ext4
3. npm
4. nginx
5. pwsha
6. mmap
7. usermod

正文

从前，有个男孩叫了一个名为 Node.js 的员，他从事过很多份 linux 的工作。

6.5.1 Linux 的主要

这个从属于 Linux 的主要，我们将主从一文说。我们会假设 Node 的理解和掌握的基础知识，我们主要介绍的内容。

Linux 是一个从内核到 CentOS 再到 Ubuntu 等的 Linux。每一个都是一套完整的操作系统，包括 apache、mysql 以及了 Linux 的软件包管理系统 yum)，一个单独的下载和 Linux 的功能。

6.5.2 实际

6.5.2.1 开始前的准备工作

在我们开始，Linux 的安装和一个环境，然后我们来具体study。

> Shell 与 Secure Shell 的区别，最初于 20 世纪 90 年代初由芬兰人 Tatu Ylonen 设计，Shell 主要用于保护通信双方之间的通信安全，Shell 客户端程序与服务器程序进行连接，后来 Shell 协议经过了两个大的版本发展，Shell 1 由于存在设计缺陷而不安全，很快被淘汰，Shell 2 重新设计协议并且不兼容 Shell 1，Shell 2 经过几个小版本的发展，其中 OpenSSH 从 2.3.0 起支持 Shell 1，从 2.9 起默认禁用 Shell 1，从 2.9.3 起彻底删除 Shell 1 协议的支持代码。Shell 协议可以用在 Linux、UNIX、Solaris、Cygwin 上，也可以用在 Windows 上，甚至 IBM 大型机上都可以找到对应的 Shell 客户端或服务器程序。

最后Linux 的 Linux 的环境，我们会 work：

```
sudo nvm-node -h -rm
```

[illegible]

> [illegible]

[illegible]

```
[illegible]
```

[illegible]

[illegible]

```
[illegible]
```

[1] [illegible]

[2] [illegible]

[3] [illegible]

[4] [illegible]

[illegible]

```
[illegible]
```

[illegible]

6.5.2.2 [illegible]

[illegible]

```
[illegible]
```

Nginx 将流量分发到多个 web 服务器，提高了系统的可用性和性能。

接下来，我们将 Node.js 应用程序部署到 Docker 容器中，并使用 Nginx 作为反向代理。首先，创建一个名为 `Dockerfile` 的文件，内容如下：

```
FROM node:14
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 8080
CMD [ "node", "server.js" ]
```

这个 Dockerfile 文件定义了一个基于 Node.js 14 的镜像，将应用程序的依赖项安装到容器中，并将应用程序的代码复制到容器中。最后，它暴露了 8080 端口，并指定了启动应用程序的命令。

接下来，我们将创建一个 Nginx 配置文件，将流量转发到我们的 Node.js 应用程序。创建一个名为 `nginx.conf` 的文件，内容如下：

```
events {
  worker_connections 1024;
}

http {
  upstream app_servers {
    server app1:8080;
    server app2:8080;
    server app3:8080;
  }

  server {
    listen 80;

    location / {
      proxy_pass http://app_servers;
      proxy_set_header Host $host;
      proxy_set_header X-Real-IP $remote_addr;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header X-Forwarded-Proto $scheme;
    }
  }
}
```

这个配置文件定义了一个名为 `app_servers` 的上游服务器，其中包含三个应用程序实例。然后，它将所有传入的请求转发到这些实例，并设置了一些标头以确保请求的正确处理。

现在，我们可以使用 Docker Compose 来启动我们的应用程序和 Nginx 服务器。创建一个名为 `docker-compose.yml` 的文件，内容如下：

```
version: '3'
services:
  app1:
    build: .
    ports:
      - "8081:8080"
  app2:
    build: .
    ports:
      - "8082:8080"
  app3:
    build: .
    ports:
      - "8083:8080"
  nginx:
    image: nginx:latest
    ports:
      - "80:80"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
```

这个 Docker Compose 文件定义了四个服务：三个应用程序实例和一个 Nginx 服务器。每个应用程序实例都将其端口映射到主机上的不同端口，而 Nginx 服务器则将其端口映射到主机上的端口 80。它还将我们的 `nginx.conf` 文件挂载到 Nginx 容器中，以便使用我们的自定义配置。

现在，我们可以使用以下命令启动我们的应用程序和 Nginx 服务器：

```
docker-compose up -d
```

这将在后台启动所有服务，并将它们连接到同一个网络中。现在，我们可以通过访问 http://localhost 来测试我们的应用程序。Nginx 将流量分发到我们的三个应用程序实例中的一个，并将响应返回给客户端。

6.5.5 Puma

Puma 是一个用于 Ruby 应用程序的高性能 Web 服务器，它基于事件驱动的架构，可以处理大量并发请求。Puma 的安装非常简单，只需在 Gemfile 中添加以下行：

```
gem 'puma'
```

然后运行 `bundle install` 命令即可。Puma 的配置文件通常为 `config/puma.rb`，可以在其中设置端口号、线程数、进程数等参数。例如：

```
port ENV.fetch("PORT") { 3000 }
threads_count = ENV.fetch("RAILS_MAX_THREADS") { 5 }
threads threads_count, threads_count
workers ENV.fetch("WEB_CONCURRENCY") { 2 }
preload_app!
```

这个配置文件将端口号设置为环境变量 `PORT` 的值，如果未设置则默认为 3000；将线程数设置为环境变量 `RAILS_MAX_THREADS` 的值，如果未设置则默认为 5；将进程数设置为环境变量 `WEB_CONCURRENCY` 的值，如果未设置则默认为 2；并启用了预加载应用程序的选项。

6.5.6 Rails

在下载并安装好 Rails 框架后，我们可以使用 `rails new` 命令来创建一个新的 Rails 应用程序。例如，创建一个名为 `myapp` 的应用程序：

```
rails new myapp
```

这将在当前目录下创建一个名为 `myapp` 的文件夹，其中包含了 Rails 应用程序的基本结构和文件。

接下来，我们需要配置数据库连接。打开 `myapp/config/database.yml` 文件，根据实际情况修改数据库连接信息，例如：

```
development:
  adapter: mysql2
  encoding: utf8
  database: myapp_development
  pool: 5
  username: root
  password:
  host: localhost
```

这将使用 MySQL 数据库，并将开发环境的数据库名称设置为 `myapp_development`，用户名为 `root`，密码为空，主机为 `localhost`。

现在，我们可以使用 `rails server` 命令来启动应用程序。默认情况下，应用程序将在端口 3000 上运行。我们可以通过访问 http://localhost:3000 来查看应用程序的欢迎页面。

6.5.7 Passenger

概要

本文は、[illegible]について説明する。

正文

Devise

[illegible]。

https://github.com/plataformatec/devise

will_paginate

[illegible]。

https://github.com/mislav/will_paginate

cancan(can)

[illegible]。[illegible]。

https://github.com/CanCanCommunity/cancancan

carrierwave

[illegible]。

https://github.com/carrierwaveuploader/carrierwave

kaminari

[illegible]。

https://github.com/kaminari/kaminari

Active Admin

[illegible]。

https://github.com/activeadmin/activeadmin

Simple Form

[illegible]

[illegible]

https://github.com/[illegible]

[illegible]

[illegible]

https://github.com/[illegible]

[illegible]

[illegible]

https://github.com/[illegible]

[illegible]

[illegible]

https://github.com/[illegible]

写在 Ruby China 社区的帖子