

5

Rails 指南

安道/chinakr 译

Rails 指南

安道/chinakr 译

样章

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?	4
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	39
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.....	45
2.1 Active Record 是什么?	45
2.2 Active Record 中的“多约定少配置”原则	46
2.3 创建 Active Record 模型	47
2.4 覆盖命名约定	47
2.5 CRUD: 读写数据	48
2.6 数据验证	49
2.7 回调	50
2.8 迁移	50
第 3 章 Active Record 迁移.....	51
3.1 迁移概述	51
3.2 创建迁移	52
3.3 编写迁移	55
3.4 运行迁移	62
3.5 修改现有的迁移	64
3.6 数据库模式转储	65

3.7 Active Record 和引用完整性	66
3.8 迁移和种子数据	66

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 应用开发变得更有乐趣。

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，也更不容易出问题。
- 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 Mac OS X 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 2.3.0p0
```

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](#)，Mac OS X 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 5.0.0`，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 [configuring#configuring-rails-applications] 。
<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</code> , <code>Gemfile.lock</code>	这两个文件用于指定 Rails 应用所需的 <code>gem</code> 依赖。Bundler <code>gem</code> 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。

文件/文件夹	作用
log/	应用日志文件。
public/	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。
Rakefile	定位并加载可在命令行中执行的任务。这些任务在 Rails 的各个组件中定义。如果要添加自定义任务，请不要修改 Rakefile，真接把自定义任务保存在 lib/tasks 文件夹中即可。
README.md	应用的自述文件，说明应用的用途、安装方法等。
test/	单元测试、固件和其他测试装置。详情请参阅 [testing#a-guide-to-testing-rails-applications] 。
tmp/	临时文件（如缓存和 PID 文件）。
vendor/	包含第三方代码，如第三方 gem。

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`。Mac OS X 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 [Ex-ecJS](#)。

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 Mac OS X，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  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下面这样：

```
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[\[routing#rails-routing-from-the-outside-in\]](#)。

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 `article` 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

  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 ApplicationController 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. reque or API requests, this action would normally respond with 204 No Content: an empty wh that you expected to actually render a template, not... nothing, so we're showing an er 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.variants` 指明伺候的是何种物理设备，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，就不会再看到缺少模板错误，而是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。

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

Extracted source (around line #6):

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

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

关于健壮参数的更多介绍，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，必须放在控制器的私有方法或受保护的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>

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

  <% @articles.each do |article| %>
    <tr>
      <td><%= article.title %></td>
      <td><%= article.text %></td>
      <td><%= link_to 'Show', article_path(article) %></td>
    </tr>
  <% end %>
</table>
```

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[\[active_record_validations#active-record-validations\]](#)。

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>Editing article</h1>

<%= form_for :article, url: article_path(@article), method: :patch 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。

`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

```

```

end

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

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

```

<table>
  <tr>
    <th>Title</th>
    <th>Text</th>
    <th colspan="2"></th>
  </tr>

  <% @articles.each do |article| %>
    <tr>
      <td><%= article.title %></td>
      <td><%= article.text %></td>
      <td><%= link_to 'Show', article_path(article) %></td>
      <td><%= link_to 'Edit', edit_article_path(article) %></td>
    </tr>
  <% end %>
</table>

```

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：



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[\[layouts_and_rendering#layouts-and-rendering-in-rails\]](#)。

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
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 %>
<table>
  <tr>
    <th>Title</th>
    <th>Text</th>
    <th colspan="3"></th>
  </tr>

  <% @articles.each do |article| %>
    <tr>
      <td><%= article.title %></td>
      <td><%= article.text %></td>
      <td><%= link_to 'Show', article_path(article) %></td>
      <td><%= link_to 'Edit', edit_article_path(article) %></td>
      <td><%= link_to 'Destroy', article_path(article),
        method: :delete,
        data: { confirm: 'Are you sure?' } %></td>
    </tr>
  <% 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 `jquery_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。

Listing Articles

[New article](#)

Title

Text

Rails is awesome! It really is. [Show](#) [Edit](#) [Destroy](#)

The page at localhost:30...



Are you sure?

Cancel

OK

图 1-11：确认对话框

提示

关于 jQuery 非侵入式适配器（jQuery UJS）的更多介绍，请参阅 [\[working_with_javascript_in_rails#working-with-javascript-in-rails\]](#)。

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 [\[routing#rails-routing-from-the-outside-in\]](#)。

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 <code>comments</code> 表的迁移文件（你的文件名会包含不同的时间戳）
app/models/comment.rb	<code>Comment</code> 模型文件

文件	用途
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 < ApplicationRecord
  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
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[\[association_basics#active-record-associations\]](#)。

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[\[routing#rails-routing-from-the-outside-in\]](#)。

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

文件/文件夹	用途
test/controllers/comments_controller_test.rb	控制器的测试文件
app/helpers/comments_helper.rb	视图辅助方法文件
app/assets/javascripts/comment.coffee	控制器的 CoffeeScript 文件
app/assets/stylesheets/comment.scss	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
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| %>
  <p>
    <strong>Commenter:</strong>
    <%= comment.commenter %>
  </p>

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

<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>
</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。

Title: Rails is Awesome!

Text: It really is.

Comments

Commenter: A fellow dev

Comment: I agree!!!

Add a comment:

Commenter

Body

Create Comment

[Edit](#) | [Back](#)

图 1-12: 带有评论的文章

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_basic_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

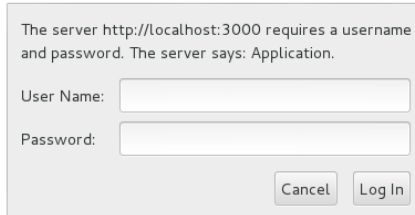
  # 为了行文简洁，省略以下内容
```

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：

Listing Articles

[New article](#)

Title	Text
Rails is awesome! It really is. Show Edit Destroy	

A dialog box for HTTP Basic Authentication. It contains the text: "The server http://localhost:3000 requires a username and password. The server says: Application." Below this text are two input fields: "User Name:" and "Password:". At the bottom right are two buttons: "Cancel" and "Log In".

The server http://localhost:3000 requires a username and password. The server says: Application.

User Name:

Password:

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[\[security#ruby-on-rails-security-guide\]](#)。

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，偶尔还是会出现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
end
```

```

    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)
```

[[active_record_querying#active-record-query-interface](#)]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
```

[[active_record_validations#active-record-validations](#)]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。[\[active_record_callbacks#active-record-callbacks\]](#)会详细介绍回调。

2.8 迁移

Rails 提供了一个 DSL（Domain-Specific Language）用来处理数据库模式，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 `YYYYMMDDHH-MMSS_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/YYYYMMDDHH-MMSS_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, index: true, 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
```

```
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, index: 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用时（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
        >>>SQL
      end
    end
  end
end
```

```

        CHECK (char_length(zipcode) = 5) NO INHERIT;
    SQL
end
dir.down do
    execute <<-SQL
        ALTER TABLE distributors
        DROP CONSTRAINT zipchk
    SQL
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
  end
end

```

```
      # ExampleMigration 中的其他操作无需撤销
    end
  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：

方法	用途
<code>suppress_messages</code>	参数是一个块，抑制块产生的任何输出。

方法	用途
<code>say</code>	接受信息文本作为参数并将其输出。方法的第二个参数是布尔值，用于说明输出结果是否缩进。
<code>say_with_time</code>	输出信息文本以及执行块所花费的时间。如果块返回整数，这个整数会被当作受块操作影响的记录的条数。

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

    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码版本控制系统（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。

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。