第 1 章

Django 网站开发基础

使用 Django 开发网站开发必须了解 Django 的框架模式和网站的架构原理，若想成为一名合格的网站开发工程师，必须掌握前端基础开发知识（HTML、CSS 和 JavaScript）、Django 的项目搭建和项目开发及调试技巧。

1.1 Django 简史

Django 是一个开放源代码的 Web 应用框架，由 Python 写成，最初用于管理劳伦斯出版集团旗下的一些以新闻内容为主的网站，即 CMS（Content Management System，内容管理系统）软件，于 2005 年 7 月在 BSD（Berkly Softuare Distribution）许可证下发布，这套框架是以比利时的吉卜赛爵士吉他手 Django Reinhardt 来命名的。Django 采用了 MTV 的框架模式，即模型（Model）、模板（Template）和视图（Views），三者之间各自负责不同的职责。

- 模型：数据存取层，处理与数据相关的所有事务，例如如何存取、如何验证有效性、包含哪些行为以及数据之间的关系等。
- 模板：表现层，处理与表现相关的决定，例如如何在页面或其他类型的文档中进行显示。
- 视图：业务逻辑层，存取模型及调取恰当模板的相关逻辑，模型与模板的桥梁。

Django 的主要目的是简便、快速地开发数据库驱动的网站。它强调代码复用，多个组件可以很方便地以插件形式服务于整个框架。Django 有许多功能强大的第三方插件，可以很方便地开发出自己的工具包，这使得 Django 具有很强的可扩展性。此外，Django 还强调快速开发和 DRY（Do Not Repeat Yourself）原则。Django 基于 MTV 的设计十分优美，其具

有以下特点：

- 对象关系映射（Object Relational Mapping，ORM）：通过定义映射类来构建数据模型，将模型与关系数据库连接起来，使用 ORM 框架内置的数据库接口可实现复杂的数据操作。
- URL 设计：开发者可以设计任意的 URL（网站地址），而且还支持使用正则表达式设计。
- 模板系统：提供可扩展的模板语言，模板之间具有可继承性。
- 表单处理：可以生成各种表单模型，而且表单具有有效性检验功能。
- Cache 系统：完善的缓存系统，可支持多种缓存方式。
- Auth 认证系统：提供用户认证、权限设置和用户组功能，功能扩展性强。
- 国际化：内置国际化系统，方便开发出多种语言的网站。
- Admin 后台系统：内置 Admin 后台管理系统，系统扩展性强。

1.2 Django 与 WSGI

在 Python 中，很多 Web 应用框架都支持 WSGI（Web Server Gateway Interface），比如 Django、Flask、Tornado 和 Bottle，等等。WSGI 是 Web 服务器网关接口，这是为 Python 语言定义的 Web 服务器和 Web 应用程序或框架之间的一种简单而通用的接口协议，它是将 Web 服务器（例如 Apache 或 Nginx）的请求转发到后端 Python Web 应用程序或 Web 框架。

可能许多读者搞不清楚 Django、WSGI 和 Web 服务器（Apache 或 Nginx）三者之间的关系，简单来说，Django 是一个 Web 应用框架，WSGI 是定义 Web 应用框架和 Web 服务器的通信协议。一个完整的网站必须包含 Web 服务器、Web 应用框架和数据库。用户通过浏览器访问网址的时候，这个访问操作相当于向网站发送一个 HTTP 请求，网站首先由 Web 服务器接受用户的 HTTP 请求，然后 Web 服务器通过 WSGI 将请求转发到 Web 应用框架进行处理，并得出处理结果，Web 应用框架通过 WSGI 将处理结果返回给 Web 服务器，最后由 Web 服务器将处理结果返回到用户的浏览器，用户即可看到相应的网页内容，如图 1-1 所示。

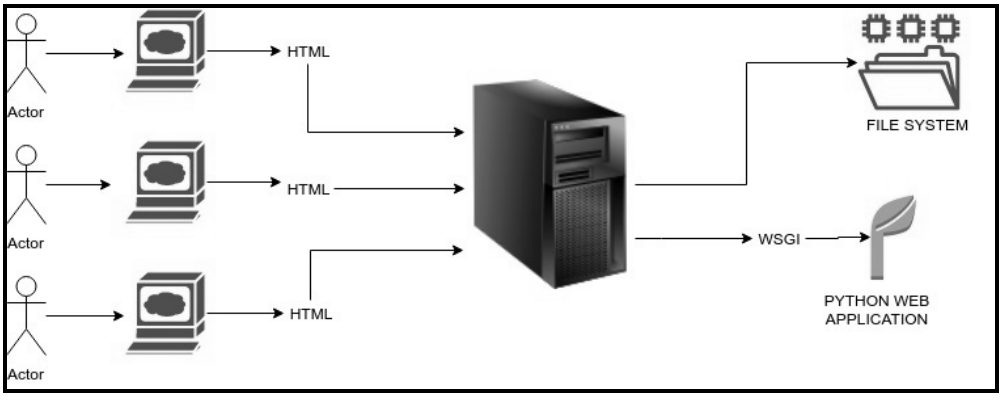


图 1-1　网站运行原理图

WSGI 分为两部分：服务端和应用端，服务端也可以称为网关端（即 uWSGI 或 Gunicorn），应用端也称为框架端（即 Django 或 Flask 的 Web 应用框架）。我们知道 WSGI 是 Web 服务器（即 Apache 或 Nginx）与 Web 应用框架的（即 Django 或 Flask 的 Web 应用框架）的通信规范，它没有具体的实现过程，因此由服务端（即 uWSGI 或 Gunicorn）实现通信过程。换句话说，服务端实现服务器和 Web 应用框架的通信传输，根据实际的网站搭建情况，我们将网站架构分为两级架构和三级架构，如图 1-2 所示。

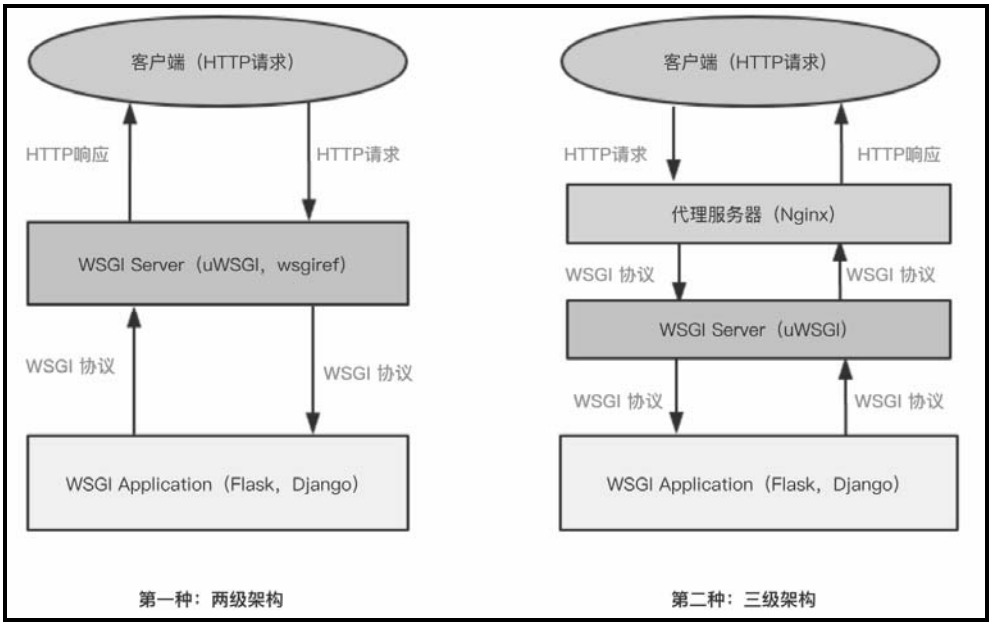


图 1-2　两级架构和三级架构

两级架构是将服务端（即 uWSGI 或 Gunicorn）作为 Web 服务器，许多 Web 框架已经附带了 WSGI 的服务端，比如 Django 和 Flask，因此它们能直接运行启动，但这种架构模式只能在开发阶段使用，在上线阶段是无法适用的，因为服务端性能比不上专业 Web 服务器（即 Apache 或 Nginx）。

三级架构是将服务端作为中间件，实现 Web 服务器和 Web 应用架构的通信，这种架构模式用于上线阶段。

1.3　HTML、CSS 和 JavaScript

网站开发可以分为前端和后端开发，前端开发是指网页设计，我们在浏览器看到网站的图片、文字、音乐视频等内容排版都是由前端开发人员实现的；后端开发是为前端开发提供实际的数据内容和业务逻辑，比如提供文字内容、图片和音乐视频的路径地址等信息。

前端开发人员必须掌握 HTML、CSS 和 JavaScript 的基础语言，这些基础语言上延伸了许多前端框架，比如 jQuery、Bootstrap、Vue、React 和 AngularJS 等。后端开发人员必须掌握一

种或多种后端开发语言、数据库应用原理、Web 服务器应用原理和基础运维技术，目前较为热门的后端开发语言分别有 Java、PHP、Python 和 GO 语言；数据库为 MySQL、MSSQL、Oracle 和 Redis 等。尽管明确划分了网站开发的职责，在实际工作中，特别是一些中小企业，他们也要求后端开发人员必须掌握前端开发技术，但无须精通前端开发，只要掌握基本的应用开发即可，比如调整网站布局或编写简单的 JavaScript 脚本。

我们除了学习使用 Django 开发网站，还需要掌握前端的基础知识，本节将简单讲述 HTML、CSS 和 JavaScript 的基础知识。

1.3.1 HTML

HTML 是超文本标记语言，标准通用标记语言下的一个应用。“超文本”就是指页面内可以包含图片、链接，甚至音乐、程序等非文字元素。超文本标记语言的结构包括“头”部分（Head）和“主体”部分（Body），其中“头”部分提供关于网页的信息，“主体”部分提供网页的具体内容。下面来看一个简单的 HTML 文档的结构：

```
<!DOCTYPE html> # 声明为 HTML5 文档
# HTML 元素是网页的根元素
<html>
# head 元素包含了文档的元（meta）数据
<head>
# meta 元素可提供有关页面的元信息（meta-information），主要是描述和关键词
<meta charset="utf-8">
# title 元素描述了文档的标题
<title>Python</title>
</head>
# body 元素包含了可见的页面内容
<body>
<h1>我的第一个标题</h1> # 定义一个标题
<p>我的第一个段落。</p> # 元素定义一个段落
</body>
</html>
```

一个完整的网页必定以<html></html>为开头和结尾，整个 HTML 可分为两部分：

（1）<head></head>，主要是对网页的描述、图片和 JavaScript 的引用。<head> 元素包含所有的头部标签元素。在 <head>元素中可以插入脚本（scripts）、样式文件（CSS）及各种 meta 信息。该区域可添加的元素标签有<title>、<style>、<meta>、<link>、<script>、<noscript>和<base>。

（2）<body></body>是网页信息的主要载体。该标签下还可以包含很多类别的标签，不同的标签有不同的作用，标签以<>开头，以</>结尾，<>和</>之间的内容是标签的值和属性，每个标签之间可以是相互独立的，也可以是嵌套、层层递进的关系。

根据这两个组成部分就能很容易地分析整个网页的布局。其中，<body></body>是整个 HTML 的重点部分，通过示例讲述如何分析<body></body>：

```
<body>
<h1>我的第一个标题</h1>
<div>
<p> Python</p>
</div>
<h2>
<p>
<a> Python</a>
</p>
</h2>
</body>
```

上述例子分析如下：

（1）<h1>和 是两个不相关的标签，两个标签是相互独立的。

（2） 和<p>是嵌套关系，<p>的上一级标签是 。

（3）<h1>和<p>这两个标签是毫无关系的。

（4）<h2>标签包含一个<p>标签，<p>标签再包含一个<a>标签，一个标签可以包含多个标签在其中。

除上述示例的标签之外，大部分标签都可以在<body></body>中添加，常用的标签如表 1-1 所示。

表 1-1　HTML 常用的标签

HTML 标签	中文释义
img	图片
a	锚
strong	加重（文本）
em	强调（文本）
i	斜体字
b	粗体（文本）
br	换行
div	分隔
span	范围
ol	排序列表
ul	不排序列表
li	列表项目
dl	定义列表
h1~h6	标题 1 到标题 6
p	段落
tr	表格中的一行
th	表格中的表头
td	表格中的一个单元格

1.3.2 CSS

HTML 代码是保存在后缀名为.html 的文件，而 CSS 样式是保存在后缀名为.css 的文件，然后在 HTML 代码中调用 CSS 样式文件。由于 HTML 代码中会存在多个不同的元素，并且每个元素的网页布局各不相同，因此需要使用 CSS 选择器定位每个 HTML 元素，然后再编写相应的 CSS 样式。

CSS 选择器划分了多种类型，同一个 HTML 元素可以使用不同的 CSS 选择器进行定位，实际开发中最常用的 CSS 选择器分别为：类别选择器、标签选择器、ID 选择器、通用选择器和群组选择器，我们将简单讲述如何使用这些 CSS 选择器实现 HTML 元素的网页布局。

为了更好理解 CSS 样式的编写规则，我们将重新定义 HTML 代码，首先在 D 盘中创建文件夹 qd，然后在 qd 文件夹中分别创建 index.html 和 index.css 文件，如图 1-3 所示。

图 1-3 目录结构

然后打开 index.html 文件，在该文件中定义网页元素，详细代码如下。

```
<html>
<head>
<link rel="stylesheet" type="text/css" href="index.css">
</head>
<body>
<h3>这是标题</h3>
<div class="content">
<p>这是正文</P>
<input id="message" placeholder="输入你的留言">
<br>
<button id="submit" >提交</button>
</div>
</body>
</html>
```

上述代码中使用 link 标签引入同一路径的 index.css 文件，link 标签是在 HTML 代码中引入 CSS 文件，使 CSS 文件的样式代码能在 HTML 代码中生效。然后设置了 5 个不同类型的 HTML 标签，分别为<h3>、 、<p>、<input>和<button>，其中 设置了 class 属性，<input>和<button>设置了 id 属性。在设置样式之前，我们使用浏览器查看没有样式效果的 index.html 文件，如图 1-4 所示。

图 1-4　网页效果

下一步将使用类别选择器、标签选择器、ID 选择器、通用选择器和群组选择器分别对这些 HTML 标签进行样式设置。打开 qd 文件夹的 index.css 文件，然后在此文件中分别编写<h3>、 、<p>、<input>和<button>的样式代码，代码如下所示。

```
/*通用选择器*/
* {
font-size:30px
}
 /*标签选择器*/
h3 {
color:blue;
}
 /*类别选择器*/
.content {
text-align:center;
}
 /*ID 选择器*/
#message {
width:500px;
}
 /*群组选择器*/
#submit, p {
color:red;
}
```

上述代码中，我们依次使用通用选择器、标签选择器、类别选择器、ID 选择器和群组选择器设置 index.html 的网页布局，从代码中可以归纳总结 CSS 选择器的语法格式，如下所示。

```
XXX {
attribute:value;
attribute:value;
}
```

CSS 选择器的语法说明如下：

（1）XXX 代表 CSS 选择器的类型。

（2）在 CSS 选择器后面使用空格并添加中括号{}，在中括号{}里面编写具体的样式设置。

（3）样式设置以 attribute:value 表示，attribute 代表样式名称，value 代表该样式设置的数值。多个样式之间使用分号“;”隔开。

（4）如果要对样式添加注释，可以使用“/**/”添加说明。

我们回看 index.css 文件，该文件的样式代码说明如下：

（1）通用选择器：它以符号“*”表示，这是设置整个网页所有元素的样式，用于网页的整体布局。上述代码是将整个网页的字体大小设为 30px。

（2）标签选择器：它以标签名表示，如果网页中有多个相同的标签，那么标签选择器的样式设置都会作用在这些标签。上述代码是将所有 h3 标签的字体颜色设为蓝色。

（3）类别选择器：它以.xxx 表示，其中 xxx 代表标签属性 class 的属性值，这是开发中常用的样式设置之一。使用类别选择器，必须在 HTML 的标签中设置 class 属性，在 class 属性的属性值前面加上实心点“.”即可作为类别选择器。上述代码是将 class="content"的标签放置网页居中位置。

（4）ID 选择器：它以#xxx 表示，其中 xxx 代表标签属性 id 的属性值，这也是开发中常用的样式设置之一。使用 ID 选择器，必须在 HTML 的标签中设置 id 属性，在 id 属性的属性值前面加上井号“#”即可作为 ID 选择器。上述代码是将 id=" message"的标签设置宽度为 500px。

（5）群组选择器：它是将多个 CSS 选择器组合成一个群组，并由这个群组对这些标签进行统一的样式设置，每个 CSS 选择器之间使用逗号隔开。上述代码是分别将 id=" submit"的标签和 p 标签的字体颜色设为红色。

最后保存 index.css 文件的样式代码，在浏览器再次查看 index.html 文件的网页效果，如图 1-5 所示。

图 1-5　网页效果

CSS 样式也可以直接在 html 文件里编写，但在企业开发中，一般都采用 HTML 和 CSS 代码分离，这样便于维护和管理，而且利于开发者阅读。

1.3.3　JavaScript

JavaScript（简称“JS”）是一种具有函数优先的轻量级、解释型的编程语言。它是因为开发 Web 页面的脚本语言而出名的，但是也被用到了很多非浏览器环境中，JavaScript 基于原型编程、多范式的动态脚本语言，并且支持面向对象、命令式和声明式的编程风格。简单来

说，JavaScript 是能被浏览器解释并执行的一种编程语言。

JavaScript 可以在 HTML 文件里编写，但在企业开发中也是采用 HTML 和 JavaScript 代码分离。为了更好地理解 JavaScript 的代码编写方式，我们在 qd 文件夹中新建 index.js 文件，文件夹的目录结构如图 1-6 所示。

图 1-6　目录结构

首先打开 index.html 文件，在 HTML 代码中引入 JS 文件，并为 button 标签添加事件触发，详细代码如下所示。

```
<html>
<head>
<link rel="stylesheet" type="text/css" href="index.css">
<script type="text/javascript" src="index.js"></script>
</head>
<body>
<h3>这是标题</h3>
<div class="content">
<p>这是正文</P>
<input id="message" placeholder="输入你的留言">
<br>
<button id="submit" onclick="getInfo()">提交</button>
</div>
</body>
</html>
```

从上述代码看到，script 标签是在 HTML 代码中引入 JS 文件，使得 JS 文件的 JavaScript 代码能在 HTML 代码中生效。button 标签添加了 onclick 属性，该属性是 JS 的事件触发，当用户单击“提交”按钮的时候，浏览器将会触发事件 onclick 所绑定的函数 getInfo()。

JavaScript 除了事件触发 onclick 之外，还提供了其他的事件触发，如表 1-2 所示。

表 1-2　JavaScript 的事件触发

事件触发	说　明
onabort	图像加载被中断时触发
onblur	元素失去焦点时触发
onchange	用户改变文本内容时触发
onclick	鼠标单击某个标签时触发
ondblclick	鼠标双击某个标签时触发

（续表）

事件触发	说　明
onerror	加载文档或图像时发生某个错误时触发
onfocus	元素获得焦点时触发
onkeydown	某个键盘的键被按下时触发
onkeypress	某个键盘的键被按下或按住时触发
onkeyup	某个键盘的键被松开时触发
onload	某个页面或图像完成加载时触发
onmousedown	某个鼠标按键被按下时触发
onmousemove	鼠标移动时触发
onmouseout	鼠标从某元素移开时触发
onmouseover	鼠标被移到某元素之上时触发
onmouseup	某个鼠标按键被松开时触发
onreset	重置按钮被单击时触发
onresize	窗口或框架被调整尺寸时触发
onselect	文本被选定时触发
onsubmit	提交按钮被单击时触发
onunload	用户退出页面时触发

我们回看 index.html 的 button 标签，由于该标签的事件触发 onclick 绑定了函数 getInfo()，因此下一步在 index.js 里定义函数 getInfo()，函数代码如下：

```
function getInfo(){
var txt = document.getElementById("message").value
if (txt){
    alert("你的留言：" + txt + "，已提交成功")
} else {
    alert("请输入你的留言");
}
}
```

上述代码的 document.getElementById 是获取 id="message"的标签（即 input 标签）的属性 value 的属性值，JavaScript 的 document 对象简称为 DOM 对象，它可以定位某个 HTML 标签并进行操作，从而实现网页的动态效果。document 对象定义了 7 个对象方法，每个对象方法的详细说明如表 1-3 所示。

表 1-3　document 对象方法

document 对象方法	说　明
close()	关闭 document.open()方法打开的输出流，并显示选定的数据
getElementById("xxx")	获取 id=xxx 的第一个 HTML 标签对象

（续表）

document 对象方法	说　明
getElementsByName("xxx")	获取所有 name=xxx 的标签对象，并以数组表示
getElementsByTagName("xxx")	获取所有 xxx 标签对象，并以数组表示
open()	收集 document.write()或 document.writeln()的数据
write()	编写 HTML 或 JavaScript 代码
writeln()	等同 write()方法，但在每个表达式后面自动添加换行符

在实际开发中，我们经常使用 getElementById、getElementsByName 和 getElementsByTagName 方法来定位 HTML 标签，然后再对已定位的 HTML 标签进行操作，由于标签的操作方法较多，本书便不再详细讲述了，有兴趣的读者可自行搜索相关资料。

最后保存 index.js 文件的 JavaScript 代码，在浏览器打开 index.html 文件，在网页的文本框输入内容并点击“提交”按钮，如图 1-7 所示。

图 1-7　网页效果

1.4　搭建开发环境

若想使用 Django 开发网站，我们需要在电脑上安装 Django 的开发环境。首先安装 Python 的开发环境，不同的操作系统有不同的安装方法，关于 Python 的安装就不再详细阐述了，本书的开发环境以 Windows10 操作系统、Python 3.8 版本为例。除了安装 Python 之外，我们还需要安装 Django 和 PyCharm，本节将会讲述如何安装 Django 和 PyCharm。

1.4.1　安装 Django 3

安装 Django 可以使用 pip 指令完成，pip 是 Python 的软件包管理工具，它可以帮助我们安装和卸载 Python 的软件包。在 Windows 中安装 Django，首先按快捷键 Windows+R 打开“运行”对话框，然后在对话框中输入“CMD”并按回车键，进入命令提示符窗口（也称为终端）。在命令提示符窗口输入以下安装指令：

```
pip install Django
```

输入上述指令后按回车键，就会自行下载 Django 最新版本并安装，我们只需等待安装完

成即可。

除了使用 pip 安装之外，还可以从网上下载 Django 的压缩包自行安装。在浏览器上输入网址（www.lfd.uci.edu/~gohlke/pythonlibs/#django）并找到 Django 的下载链接，如图 1-8 所示。

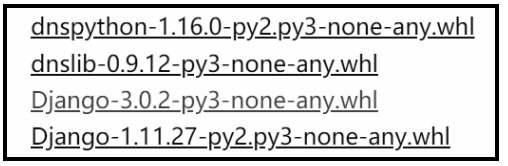


图 1-8　Django 安装包

然后将下载的 whl 文件放到 D 盘，并打开命令提示符窗口，输入以下安装指令：

```
pip install D:\ Django-3.0.2-py3-none-any.whl
```

输入指令后按回车键，等待安装完成的提示即可。完成 Django 的安装后，需要进一步校验安装是否成功，再次进入命令提示符窗口，输入“python”并按回车键，此时进入 Python 交互解释器，在交互解释器下输入校验代码：

```
>>> import django
>>> django.__version__
'3.0.2'
```

从上面返回的结果可以看到，当前安装的 Django 版本为 3.0.2，说明 Django 安装成功。

1.4.2　安装 PyCharm

PyCharm 是一种 Python IDE，它带有一整套可以帮助用户在使用 Python 语言开发时提高其效率的工具，比如调试、语法高亮、项目管理、代码跳转、智能提示、自动完成、单元测试、版本控制等。此外，该 IDE 提供了一些高级功能，例如支持 Django 框架下的专业 Web 开发。

PyCharm 分为专业版和社区版，专业版是收费的，但功能齐全，如果在 PyCharm 里使用 Django 开发网站，建议使用专业版。社区版是免费使用的，但功能十分有限，两者的功能使用权限如图 1-9 所示。

在浏览器中输入下载地址 http://www.jetbrains.com/pycharm/download，可以看到 PyCharm 分别支持 Windows、Linux 和 MacOS 三大系统的使用，版本分为专业版和社区版。本书以在 Windows 下安装 PyCharm 专业版为例，在官网上下载 Windows 的 PyCharm 专业版安装包，双击打开安装包，并根据安装提示完成安装过程即可。

完成 PyCharm 安装后，在桌面上双击 PyCharm 的图标，将其运行启动。初次运行 PyCharm，用户进行简单的设置后会进入软件激活界面，激活方式有三种：Jetbrains 用户激活、激活码和许可服务器。如图 1-10 所示。

	PyCharm Professional Edition	PyCharm Community Edition
Intelligent Python editor	✓	✓
Graphical debugger and test runner	✓	✓
Navigation and Refactorings	✓	✓
Code inspections	✓	✓
VCS support	✓	✓
Scientific tools	✓	
Web development	✓	
Python web frameworks	✓	
Python Profiler	✓	
Remote development capabilities	✓	
Database & SQL support	✓	

图 1-9　专业版和社区版的功能使用权限

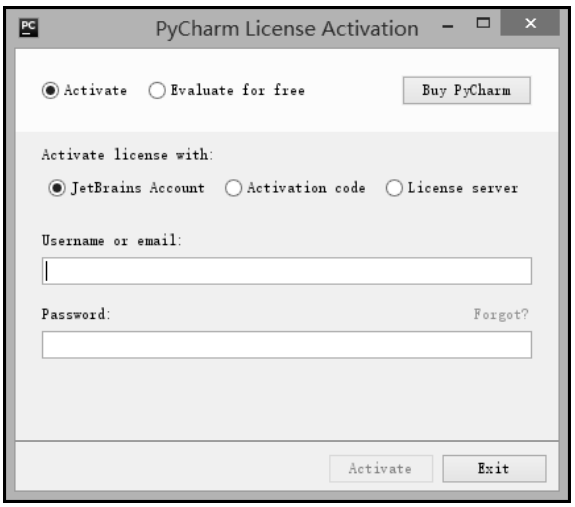


图 1-10　PyCharm 激活界面

1.5　创建 Django 项目

创建 Django 项目可以在终端输入指令完成，也可以在 PyCharm 里创建项目，前者是通过 Django 内置的指令实现，后者是在 PyCharm 的可视化界面完成。

1.5.1　使用内置指令创建项目

一个项目可以理解为一个网站，创建 Django 项目可以在命令提示符窗口输入创建指令完成。打开命令提示符窗口，将当前路径切换到 D 盘并输入项目创建指令：

```
C:\Users\000>d:
```

```
D:\>django-admin startproject MyDjango
```

第一行指令是将当前路径切换到 D 盘；第二行指令是在 D 盘的路径下创建 Django 项目，指令中的“MyDjango”是项目名称，读者可自行命名。项目创建后，可以在 D 盘下看到新创建的文件夹 MyDjango，在 PyCharm 下查看该项目的结构，如图 1-11 所示。

图 1-11　项目目录结构

MyDjango 项目里包含 MyDjango 文件夹和 manage.py 文件，而 MyDjango 文件夹又包含 5 个.py 文件。项目的每个文件说明如下：

- manage.py：命令行工具，内置多种方式与项目进行交互。在命令提示符窗口下，将路径切换到 MyDjango 项目并输入 python manage.py help，可以查看该工具的指令信息。
- __init__.py：初始化文件，一般情况下无须修改。
- asgi.py：用于启动异步通信服务，比如实现在线聊天等异步通信功能。
- settings.py：项目的配置文件，项目的所有功能都需要在该文件中进行配置，配置说明会在下一章详细讲述。
- urls.py：项目的路由设置，设置网站的具体网址内容。
- wsgi.py：全称为 Python Web Server Gateway Interface，即 Python 服务器网关接口，是 Python 应用与 Web 服务器之间的接口，用于 Django 项目在服务器上的部署和上线，一般不需要修改。

从 Django 3.0 开始，新建的项目都会创建 asgi.py 文件，这是将异步通信服务纳入 Django 的内置功能，也是 Django 3.0 的新特性之一。ASGI 是异步网关协议接口，一个介于网络协议服务和 Python 应用之间的标准接口，能够处理多种通用的协议类型，包括 HTTP、HTTP2 和 WebSocket。

WSGI 是基于 HTTP 协议模式，但它不支持 WebSocket，而 ASGI 则是为了解决 WSGI 不支持当前 Web 开发中的一些新的协议标准（比如 WebSocket）。同时，ASGI 不仅支持 WSGI 原有的模式，而且还支持使用 WebSocket，简单来说，ASGI 是 WSGI 的功能扩展。

完成项目的创建后，接着创建项目应用，项目应用简称为 App，相当于网站功能，每个 App 代表网站的一个功能。App 的创建由文件 manage.py 实现，创建指令如下：

```
D:\>cd MyDjango
D:\MyDjango>python manage.py startapp index
```

从 D 盘进入项目 MyDjango，然后使用 python manage.py startapp XXX 创建，其中 XXX 是应用的名称，读者可以自行命名。上述指令创建了网站首页，再次查看项目 MyDjango 的目录结构，如图 1-12 所示。

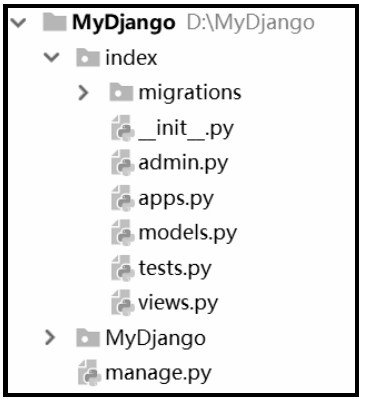


图 1-12　目录结构

从图 1-12 可以看到，项目新建了 index 文件夹，其可作为网站首页。在 index 文件夹中可以看到有多个.py 文件和 migrations 文件夹，说明如下：

- migrations：用于生成数据迁移文件，通过数据迁移文件可自动在数据库里生成相应的数据表。
- __init__.py：index 文件夹的初始化文件。
- admin.py：用于设置当前 App 的后台管理功能。
- apps.py：当前 App 的配置信息，在 Django 1.9 版本后自动生成，一般情况下无须修改。
- models.py：定义数据库的映射类，每个类可以关联一张数据表，实现数据持久化，即 MTV 里面的模型（Model）。
- tests.py：自动化测试的模块，用于实现单元测试。
- views.py：视图文件，处理功能的业务逻辑，即 MTV 里面的视图（Views）。

完成项目和 App 的创建后，最后在命令提示符窗口输入以下指令启动项目：

```
C:\Users\000>d:
D:\>cd MyDjango
D:\MyDjango>python manage.py runserver 8001
```

将命令提示符窗口的路径切换到项目的路径，输入运行指令 python manage.py runserver 8001，如图 1-13 所示。其中 8001 是端口号，如果在指令里没有设置端口，端口就默认为 8000。最后在浏览器中输入 http://127.0.0.1:8001/，可看到项目的运行情况，如图 1-14 所示。

```
D:\MyDjango>python manage.py runserver 8001
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 17 unapplied migration(s). Your project may not
 auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
February 11, 2020 - 12:04:15
Django version 3.0.2, using settings 'MyDjango.settings'
Starting development server at http://127.0.0.1:8001/
Quit the server with CTRL-BREAK.
```

图 1-13　输入运行指令

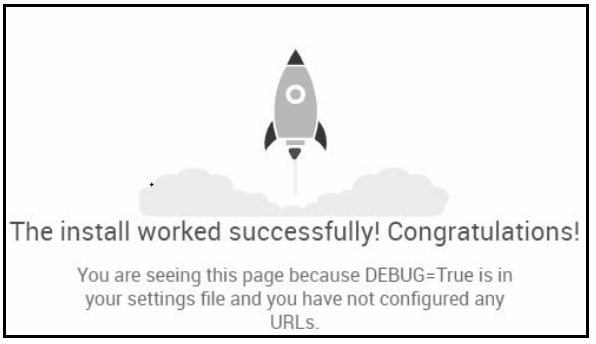


图 1-14　项目运行结果

1.5.2　使用 PyCharm 创建项目

除了在命令提示符窗口创建项目之外，还可以在 PyCharm 中创建项目。PyCharm 必须为专业版才能创建与调试 Django 项目，社区版是不支持此功能的。打开 PyCharm 并在左上方单击 File→New Project，创建新项目，如图 1-15 所示。

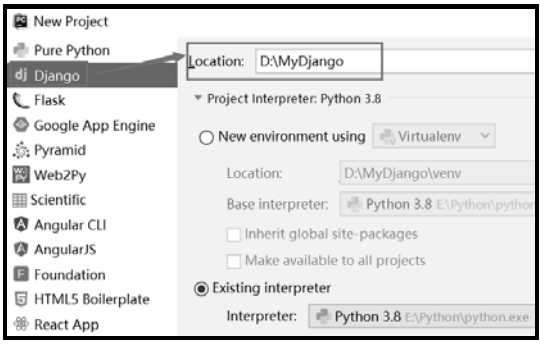


图 1-15　创建 Django

项目创建后，可以看到目录结构多出了 templates 文件夹，该文件夹用于存放 HTML 模板文件，如图 1-16 所示。

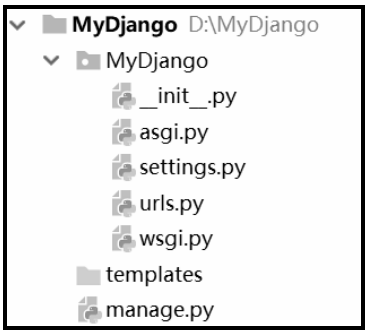


图 1-16　目录结构

接着创建 App 应用，可以在 PyCharm 的 Terminal 中输入创建指令，创建指令与命令提示符窗口中输入的指令是相同的，如图 1-17 所示。

图 1-17　创建 App

完成项目和 App 的创建后，启动项目。如果项目是由 PyCharm 创建的，就直接单击“运行”按钮启动项目，如图 1-18 所示。

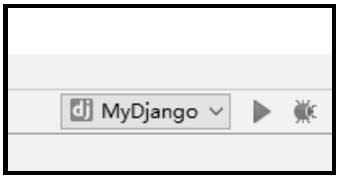


图 1-18　启动 Django

如果项目是在命令提示符窗口创建的，想要在 PyCharm 启动项目，而 PyCharm 没有运行脚本，就需要对该项目创建运行脚本，如图 1-19 所示。

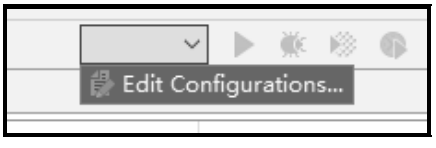


图 1-19　创建运行脚本

然后单击图 1-19 中的 Edit Configurations 就会出现 Run/Debug Configurations 界面，单击该界面左上方的 + 并选择 Django server，最后输入脚本名字，单击 OK 按钮即可创建运行脚本，如图 1-20 所示。

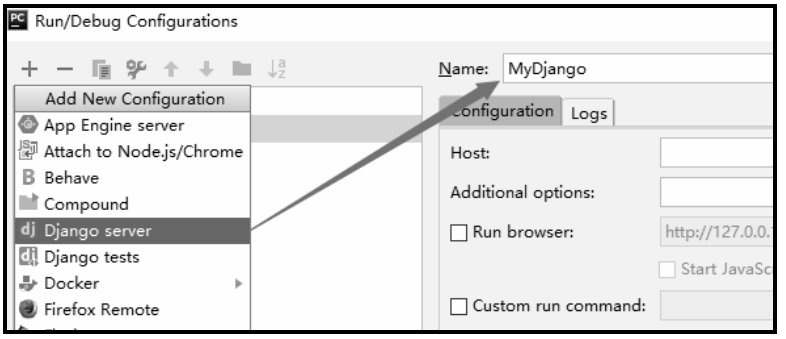


图 1-20　运行脚本

1.6　程序调试技巧

在开发网站的过程中，为了确保功能可以正常运行及验证是否实现开发需求，开发人员需要对已实现的功能进行调试。Django 的调试方式分为 PyCharm 断点调试和调试异常。

1.6.1　PyCharm 的 Debug 模式

我们知道，PyCharm 调试 Django 开发的项目，PyCharm 的版本必须为专业版，而社区版是不具备 Web 开发功能的。使用 PyCharm 启动 Django 的时候，可以发现 PyCharm 上带有爬虫的按钮，该按钮用于开启 Django 的 Debug 调试模式，如图 1-21 所示。

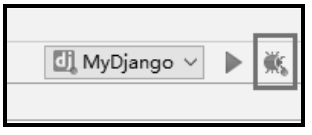


图 1-21　调试按钮

单击图 1-21 中的调试按钮（带有爬虫的按钮），即可开启调试模式，在 PyCharm 的正下方可以看到相关的调试信息，如图 1-22 所示。

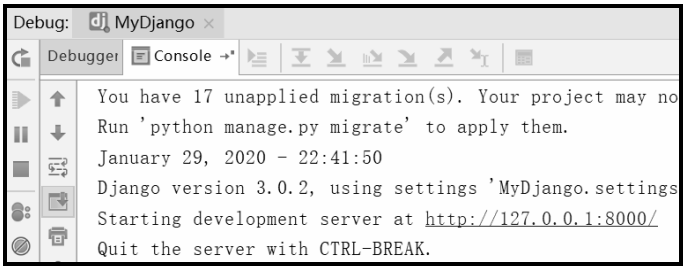


图 1-22　调试信息

从图 1-22 的调试界面可以看到有多个操作按钮，我们将常用的调试按钮的功能说明以表格的形式表示，如表 1-4 所示。

表 1-4　常用的调试按钮的功能说明

按　钮	说　明
Console	显示项目的运行信息
Debugger	显示程序的对象信息
	重新运行项目
	继续往下执行程序，直到下一个断点才暂停程序
	暂停当前运行的程序
	停止程序的运行
	查看所有断点信息
	清空 Console 的信息
	程序断点后，执行下一行的代码
	显示当前断点的位置

我们通过简单的示例来讲述如何使用 PyCharm 的调试模式。以 MyDjango 项目为例，在 index 文件夹的 views.py 文件里，视图函数 index 添加变量 value 并且在返回值 return 处设置断点，如图 1-23 所示。

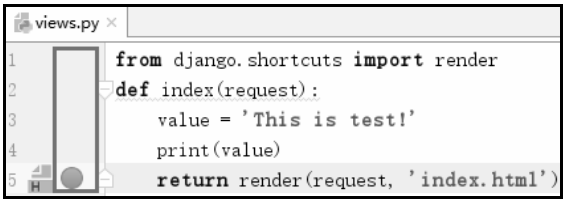


图 1-23　设置断点

设置断点是在图 1-23 的方框里单击一下即可出现红色的圆点，该圆点代表断点设置，当项目开启调试模式并运行到断点所在的代码位置，程序就会暂停运行。

开启 MyDjango 项目的调试模式并在浏览器上访问 127.0.0.1:8000，在 PyCharm 正下方的调试界面里可以看到相关的代码信息，如图 1-24 所示。

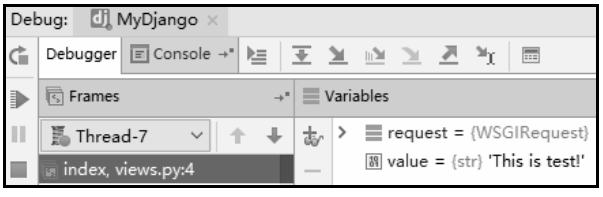


图 1-24　代码信息

调试界面 Debugger 的 Frames 是当前断点的程序所依赖的程序文件，单击某个文件，Variables 就会显示当前文件的程序所生成的对象信息。

单击 按钮，PyCharm 就会自动往下执行程序，直到下一个断点才暂停程序；单击 按

钮，PyCharm 只会执行当前暂停位置的下一步代码，这样可以清晰地看到每行代码的执行情况。这两个按钮是断点调试最为常用的，它们能让开发者清晰地了解代码的执行情况和运行逻辑。

如果程序在运行过程中出现异常或者代码中设有输出功能（如 print），这些信息就可以在 PyCharm 正下方调试界面的 Console 里查看，如图 1-25 所示。

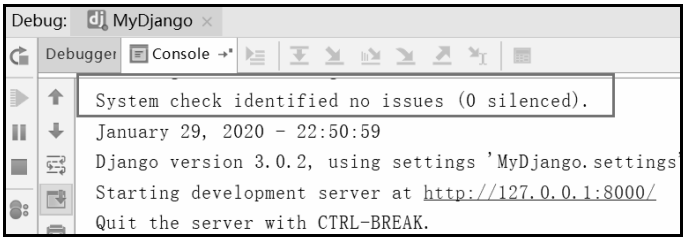


图 1-25 输出信息

启动项目的时候，从图 1-25 的运行窗口可看到“System check identified no issues (0 silenced)”信息，该信息表示 Django 对项目里所有的代码语法进行检测，如果代码语法存在错误，在启动的过程中就会报出相关的异常信息。

图 1-25 中的“This is test!”是视图函数 index 的“print(value)”代码的输出结果；"GET / HTTP/1.1" 200 代表浏览器成功访问 127.0.0.1:8000，其中 200 代表 HTTP 的状态码。

注 意

断点调试无法在模板文件（templates的 index.html）设置断点，因此无法对模板文件进行调试，只能通过 PyCharm的调试界面 Console或浏览器开发者工具进行调试。

1.6.2 异常提示进行调试

PyCharm 的调试模式无法调试模板文件，而模板文件需要使用 Django 的模板语法，若想调试模板文件，则最有效的方法是查看 PyCharm 或浏览器提示的异常信息。

调试异常需要根据项目运行时所产生的异常信息进行分析，使用浏览器访问路由地址的时候，如果出现异常信息，就可以直接查看异常信息找出错误位置。比如在 templates 的模板文件 index.html 里添加错误的代码，如下所示：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello World</title>
</head>
<body>
    {# 添加错误代码 static#}
    {% static %}
    <span>Hello World!!</span>
```

```
</body>
</html>
```

当运行 MyDjango 项目并在浏览器访问 127.0.0.1:8000 的时候，PyCharm 正下方的调试界面 Console 就会出现异常信息，从异常信息中可以找到具体的异常位置，如图 1-26 所示。

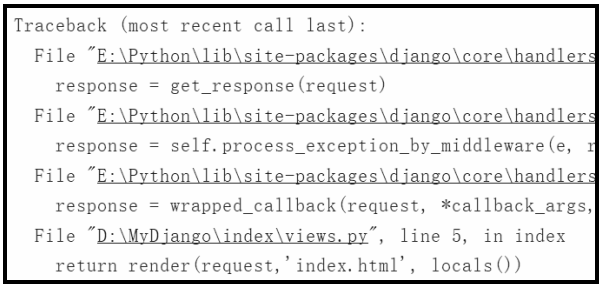


图 1-26　异常信息

除了在 PyCharm 正下方的调试界面 Console 中查看异常信息外，还可以在浏览器上分析异常信息，比如模板文件 index.html 的错误语法，Django 还能标记出错位置，便于开发者调试和跟踪，如图 1-27 所示。

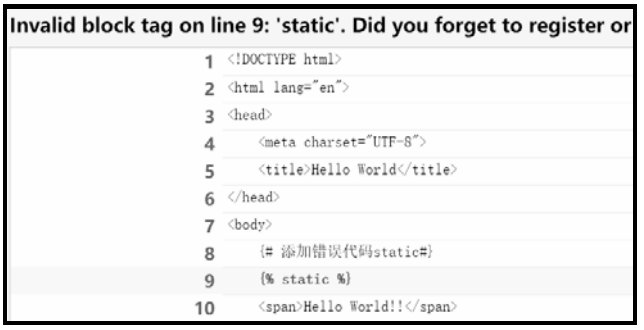


图 1-27　异常信息

还有一种常见的情况是网页能正常显示，但网页内容出现部分缺失，对于这种情况，只能使用浏览器的开发者工具对网页进行分析处理。以 templates 的模板文件 index.html 为例，对其添加正确的代码，但在网页里出现内容缺失，如下所示：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello World</title>
</head>
<body>
    {# 添加正确代码，但不出现在网页 #}
    <div>Hi,{{ value }}</div>
    <span>Hello World!!</span>
</body>
```

```
</html>
```

再次启动 MyDjango 项目并在浏览器访问 127.0.0.1:8000 的时候，浏览器能正常访问网页，但无法显示{{ value }}的内容，打开浏览器的开发者工具可以看到，{{ value }}的内容是不存在的，如图 1-28 所示。

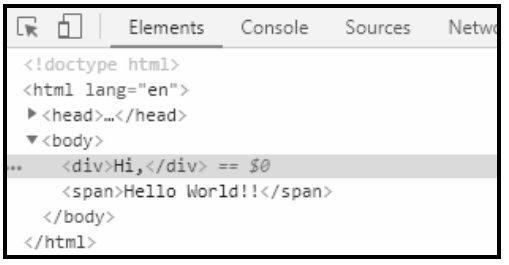


图 1-28 开发者工具

此外，浏览器的开发者工具对于调试 Ajax 和 CSS 样式非常有用。通过生成的网页内容进行分析来反向检测代码的合理性是常见的手段之一，这是通过校验结果与开发需求是否一致的方法来调试项目功能的。

1.7 本章小结

Django 是一个开放源代码的 Web 应用框架，由 Python 写成，最初用于管理劳伦斯出版集团旗下的一些以新闻内容为主的网站，即 CMS 软件，于 2005 年 7 月在 BSD 许可证下发布，这套框架是以比利时的吉卜赛爵士吉他手 Django Reinhardt 来命名的。Django 采用了 MTV 的框架模式，即模型（Model）、模板（Template）和视图（Views），三者之间各自负责不同的职责。

- 模型：数据存取层，处理与数据相关的所有事务，例如如何存取、如何验证有效性、包含哪些行为以及数据之间的关系等。
- 模板：表现层，处理与表现相关的决定，例如如何在页面或其他类型的文档中进行显示。
- 视图：业务逻辑层，存取模型及调取恰当模板的相关逻辑，模型与模板的桥梁。

Django 是一个 Web 应用框架，WSGI 是定义 Web 应用框架和 Web 服务器（Apache 或 Nginx）的通信协议。一个完整的网站必须包含 Web 服务器、Web 应用框架和数据库。用户通过浏览器访问网址的时候，这个访问操作相当于向网站发送一个 HTTP 请求，网站首先由 Web 服务器接受用户的 HTTP 请求，然后 Web 服务器通过 WSGI 接口将请求转发到 Web 应用框架进行处理，并得出处理结果，Web 应用框架通过 WSGI 接口将处理结果返回给 Web 服务器，最后由 Web 服务器将处理结果返回到用户的浏览器，用户即可看到相应的网页内容。

网站开发可以分为前端和后端开发，前端开发是指网页设计，我们在浏览器看到网站的图片、文字、音乐、视频等内容排版都是由前端开发人员实现；后端开发是为前端开发提供

实际的数据内容，比如提供文字内容、图片和音乐及视频的路径地址等信息。

前端开发人员必须掌握 HTML、CSS 和 JavaScript 的基础语言，在这些基础语言上延伸了许多前端框架，比如 jQuery、Bootstrap、Vue、React 和 AngularJS 等。后端开发人员必须掌握后端开发语言、数据库应用原理、Web 服务器应用原理和基础运维技术，目前较为热门的后端开发语言分别有 Java、PHP、Python 和 GO 语言；数据库为 MySQL、MSSQL、Oracle 和 Redis 等。尽管明确划分了网站开发的职责，在实际工作中，特别是一些中小企业，他们要求后端开发人员必须掌握前端开发技术，无须精通前端开发，但必须掌握基本的应用开发，比如网站的布局调整或编写简单的 JavaScript 脚本。

Django 的安装建议使用 pip 执行安装，安装的方法如下：

```
# 方法一
pip install Django
# 方法二
pip install D:\ Django-3.0.2-py3-none-any.whl
```

两种不同的安装方法都是使用 pip 执行的，唯一的不同之处在于前者在安装过程中会从互联网下载安装包，而后者直接对本地已下载的安装包进行解压安装。Django 安装完成后，在 Python 交互解释器模式校验安装是否成功：

```
>>> import django
>>> django.__version__
'3.0.2'
```

Django 的目录结构以及含义如下：

- manage.py：命令行工具，内置多种方式与项目进行交互。在命令提示符窗口下，将路径切换到 MyDjango 项目并输入 python manage.py help，可以查看该工具的指令信息。
- __init__.py：初始化文件，一般情况下无须修改。
- asgi.py：用于启动异步通信服务，比如实现在线聊天等异步通信功能。
- settings.py：项目的配置文件，项目的所有功能都需要在该文件中进行配置，配置说明会在下一章详细讲述。
- urls.py：项目的路由设置，设置网站的具体网址内容。
- wsgi.py：全称为 Python Web Server Gateway Interface，即 Python 服务器网关接口，是 Python 应用与 Web 服务器之间的接口，用于 Django 项目在服务器上的部署和上线，一般不需要修改。
- migrations：用于生成数据迁移文件，通过数据迁移文件可自动在数据库里生成相应的数据表。
- __init__.py：index 文件夹的初始化文件。
- admin.py：用于设置当前 App 的后台管理功能。
- apps.py：当前 App 的配置信息，在 Django 1.9 版本后自动生成，一般情况下无须修改。
- models.py：定义数据库的映射类，每个类可以关联一张数据表，实现数据持久化，

即 MTV 里面的模型（Model）。

- tests.py：自动化测试的模块，用于实现单元测试。
- views.py：视图文件，处理功能的业务逻辑，即 MTV 里面的视图（Views）。

此外，作为 Web 开发者必须掌握 Django 的功能调试方法，学会如何使用 PyCharm 调试项目以及分析项目在运行过程中出现的异常信息。

第 2 章

商城的设计说明与配置

网站开发隶属于软件工程，开发流程为：需求分析→设计说明（细分为概要设计和详细设计）→代码编写→程序测试→软件交付→客户验收→后期维护。本章分别从需求分析、设计说明的角度研究如何分析客户需求，并根据客户需求设计网站架构。

2.1 需求分析

假设我们是一家软件开发公司，公司员工分别有需求工程师、网页设计师、前端工程师、后端工程师和测试工程师，现有一名客户想开发自家的购物平台，该客户拥有实体门店，专售母婴产品。大多数情况下，客户对网站平台的开发流程只有表面的认知，他们不会提出详细的需求，只会说出他们的目的，比如说“我只想有一个自家的购物平台，能让我的客户在线购买产品，好像淘宝那样就行了。”在实际开发中，我们肯定不能直接仿造淘宝交付给客户，毕竟客户有自己的实体门店，应结合门店现有的业务流程定制购物平台。

对于客户的精简需求，需求工程师需要深入了解客户的具体需求，比如了解客户现有的顾客量、产品类型、实体店的进销存管理方式等因素，这些都会影响网站设计模式，例如现有的顾客数量需要考虑网站的并发量、产品类型影响网站页面设计（如商品详细页的布局设计）、实体店的进销存管理方式影响商品库存管理，是否考虑缺货提醒、预售功能等。

需求工程师根据客户的实际情况，梳理并归纳以下需求要点：

（1）网站需要提供搜索功能，便于用户搜索商品。
（2）搜索结果需要根据销量、价格、上架时间和收藏数量进行排序。
（3）商品详情应有尺寸、原价、活动价、图片展示、收藏功能和购买功能。
（4）用户购买后应看到订单信息，订单信息包括支付金额、购买时间和订单状态。

（5）商品购买应支持在线支付，如支付宝或微信支付等功能。

（6）目前顾客数量约有 3000 人，实体店暂无进销存系统。

在需求分析阶段，需求工程师要不断地与客户反复交流，并将交流结果以 Demo 的方式展示给客户，直到客户确认无误为止。在此阶段，需求工程师需要使用简单的绘图软件完成 Demo 设计，比如 Axure 或 Visio 等软件。除此之外，需求工程师还要将收集的需求信息编写成需求规格说明书。

2.2 设计说明

当我们完成客户的需求分析之后，下一步是执行系统的设计说明，它包括了概要设计和详细设计。概要设计划分为系统总体结构设计、数据结构及数据库设计、概要设计文档说明；详细设计是对系统每个功能模块进行算法设计、业务逻辑处理、网页界面设计、代码设计等具体的实现过程。

在概要设计阶段，系统总体结构设计需要由需求工程师和开发人员共同商议，针对用户需求来商量如何设计系统各个功能模块以及各个模块的数据结构。我们根据用户需求，网站的概要设计说明如下：

（1）网站首页应设有导航栏，并且所有功能展示在导航栏，在导航栏的下面展示各类的热销商品，当单击商品图片时即可进入商品详细页面，导航栏上方设有搜索框，便于用户搜索相关商品。

（2）商品列表页将所有商品以一定规则排序展示，用户可以从销量、价格、上架时间和收藏数量设置商品的排序方式，并且在页面的左侧设置分类列表，选择某一分类可以筛选出相应的商品信息。

（3）商品详细页展示某一商品的主图、名称、规格、数量、详细介绍、购买按钮和收藏按钮，并在商品详细介绍的左侧设置了热销商品列表。

（4）购物车页面只能在用户已登录的情况下才能访问，它是将用户选购的商品以列表形式展示，列表的每行数据包含了商品图片、名称、单价、数量、合计和删除操作，用户可以增减商品的购买数量，并且能自动计算费用。

（5）个人中心页面用于展示用户的基本信息及订单信息，只能在用户已登录的情况下才能访问。

（6）用户登录注册页面是共用一个页面，如果用户账号已存在，则对用户账号密码验证并登录，如果用户不存在，则对当前的账号密码进行注册处理。

（7）数据库使用 MySQL 5.7 以上版本，数据表除了 Django 内置数据表之外，还需定义商品信息表、商品类别表、购物车信息表和订单信息表。

从概要设计看到，我们大概搭建了网站的整体架构，下一步是在整体架构的基础上完善各个功能模块的细节内容。在详细设计中，网站开发最主要的是完成网页设计和数据库的数据结构，如果某些功能涉及复杂的逻辑业务，还需编写相应的算法。

根据概要设计说明，分别对网站的网页设计和数据库的数据结构进行具体设计说明。一共设计了 6 个网页界面，每个网页界面的设计说明如下：

网站首页一共划分了 5 个不同的功能区域：商品搜索功能、网站导航、广告轮播、商品分类热销、网站尾部，如图 2-1 所示，每个功能的设计说明如下：

（1）商品搜索功能：用户输入关键字并单击搜索按钮，网站将进行数据查询处理，将符合条件的商品在商品列表页展示；如果没有输入关键字的情况下单击搜索按钮，网站直接访问商品列表页并展示所有的商品信息。

（2）网站导航：设有首页、所有商品、购物车和个人中心的地址链接，每个链接分别对应网站首页、商品列表页、购物车页面和个人中心页面。

（3）广告轮播：以图片形式展示，用于商品的广告宣传。

（4）商品分类热销：分为今日必抢和分类商品。今日必抢是在所有商品中获取前十名销量最高的商品进行排序；分类商品是在某分类的商品中获取前五名销量最高的商品进行排序。

（5）网站尾部：这是每个网站的基本架构，用于说明网站的基本信息，如备案信息、售后服务、联系我们等。

图 2-1　网站首页

商品列表页分为 4 个功能区域：商品搜索功能、网站导航、商品分类和商品列表信息，如图 2-2 所示，每个功能的设计说明如下：

（1）商品分类：当用户选择某一分类的时候，网站会筛选出对应的商品信息并在右侧的商品列表信息展示。

（2）商品列表信息：提供了销量、价格、上架时间和收藏数量的排序方式，商品默认以销量排序，并设置分页功能，每一页只显示 6 条商品信息。

图 2-2　商品列表页

商品详细页分为 5 个功能区：商品搜索功能、网站导航、商品基本信息、商品详细介绍和热销推荐，如图 2-3 所示，每个功能的设计说明如下：

（1）商品基本信息：包含了商品的规格、名称、价格、主图、购买数量、收藏按钮和购买按钮。收藏按钮使用 JavaScript 脚本完成收藏功能，购买按钮将商品信息和购买数量添加到购物车。

（2）商品详细介绍：以图片形式展示，用于描述商品的细节内容。

（3）热销推荐：在所有商品中（排除当前商品之外）获取并展示前五名销量最高的商品。

图 2-3　商品详细页

购物车页面分为 3 个功能区域：商品搜索功能、网站导航、商品的购买费用核算，如图 2-4 所示。商品的购买费用核算允许用户编辑商品的购买数量、选择购买的商品和删除商品，结算按钮根据购买信息自动跳转到支付页面。

图 2-4　购物车页面

个人中心页面分为 4 个功能区域：商品搜索功能、网站导航、用户基本信息和订单信息，如图 2-5 所示，用户基本信息和订单信息的设计说明如下：

（1）用户基本信息：在网页的左侧位置，展示了用户的头像、名称和登录时间，按钮功能分别有购物车页面链接和退出登录。

（2）订单信息：以数据列表展示，每行数据包含了订单编号、价格、购买时间和状态，并设置分页功能，每一页显示 7 条订单信息。

图 2-5　个人中心页面

用户登录注册页面分为 3 个功能区域：商品搜索功能、网站导航、登录注册表单，如图 2-6 所示。登录注册表单是共用一个网页表单，如果用户账号已存在，则对用户账号密码验证并登录，如果用户不存在，则对当前的账号密码进行注册处理。

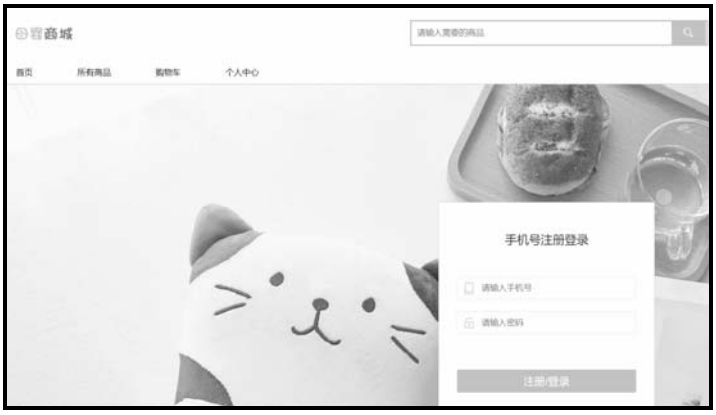


图 2-6　用户登录注册页面

从网站的 6 个页面看到，每个页面的设计和布局都需要数据支持，比如商品的规格、名称、价格、主图等数据信息。由于 Django 内置了用户管理功能，已为我们提供了用户信息表，因此我们只需定义商品信息表、商品类别表、购物车信息表和订单信息表，每个数据表的数据结构如表 2-1 所示。

表 2-1　商品信息表的数据结构

表 字 段	字段类型	含　义
id	Int 类型，长度为 11	主键
name	Varchar 类型，长度为 100	商品名称
sezes	Varchar 类型，长度为 100	商品规格
types	Varchar 类型，长度为 100	商品类型
price	Float 类型	商品价格
discount	Float 类型	折后价格
stock	Int 类型	存货数量
sold	Int 类型	已售数量
likes	Int 类型	收藏数量
created	Date 类型	上架日期
img	Varchar 类型，长度为 100	商品主图
details	Varchar 类型，长度为 100	商品描述

从表 2-1 看到，商品信息表负责记录商品的基本信息，其中商品主图和商品描述是以文件路径的形式记录在数据库中的。一般来说，如果网站中涉及文件的存储和使用，那么数据库最好记录文件的路径地址。若将文件内容以二进制的数据格式写入数据库，则会对数据库造成一定的压力，从而降低网站的响应速度。

商品信息表的字段 types 是代表商品类型，每一个商品类型都记录在商品类别表中，因此商品类别表的数据结构如表 2-2 所示。

表 2-2　商品类别表的数据结构

表 字 段	字段类型	含　义
id	Int 类型，长度为 11	主键
firsts	Varchar 类型，长度为 100	一级分类
seconds	Varchar 类型，长度为 100	二级分类

商品类别表分为一级分类和二级分类，它的设计是由商品列表页的商品分类决定，如图 2-2 所示，比如图 2-2 的“奶粉辅食”作为一级分类，该分类下设置了二级分类（进口奶粉、宝宝辅食、营养品），而商品信息表的字段 types 来自商品类别表的二级分类字段 seconds。

虽然商品信息表的字段 types 来自商品类别表的二级分类字段 seconds，但两个数据表之间并没有设置外键关系，这样的设计方式能够降低两个数据表之间的耦合性。如果网站需要改造成微服务架构或分布式架构，这种设计方式符合微服务或分布式的拆分要求。

购物车信息表的数据来自于商品信息表，为了简化表字段数量，我们在购物车信息表设置字段 commodityInfos_id，该字段是商品信息表的主键 id，从而使商品信息表和购物车信息表构成数据关联，这种方式不仅能简化字段数量，当商品信息发生改动，购物车的商品信息也能及时更新。此外，购物车信息表还需要设置字段 user_id，该字段是 Django 内置用户表的主键 id，用于区分不同用户的购物车信息，因此购物车信息表的数据结构如表 2-3 所示。

表 2-3　购物车信息表的数据结构

表 字 段	字段类型	含　义
id	Int 类型，长度为 11	主键
quantity	Int 类型，长度为 11	购买数量
commodityInfos_id	Int 类型，长度为 11	商品信息表的主键 id
user_id	Int 类型，长度为 11	Django 内置用户表的主键 id

当购物车信息表的商品执行结算操作的时候，结算费用将写入订单信息表的字段 price，并且还需要根据不同的用户区分相应的订单信息，因此订单信息表的数据结构如表 2-4 所示。

表 2-4　订单信息表的数据结构

表 字 段	字段类型	含　义
id	Int 类型，长度为 11	主键
price	Float 类型，长度为 11	订单总价
created	Int 类型，长度为 11	订单创建时间
user_id	Date 类型	Django 内置用户表的主键 id
state	Varchar 类型，长度为 20	订单状态

综合上述，我们将商品信息表、商品类别表、购物车信息表、订单信息表和 Django 内置用户表的数据关系进行整理，如图 2-7 所示。

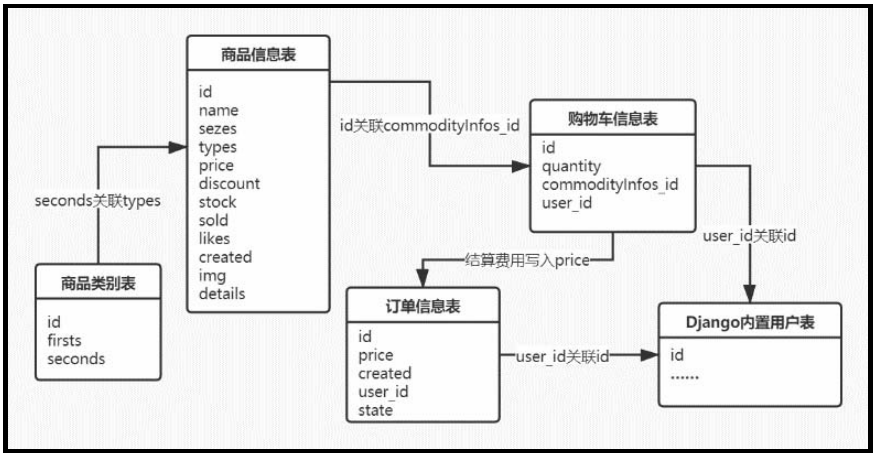


图 2-7　数据表的数据关系

2.3　搭建项目开发环境

当我们了解整个项目的开发设计之后，下一步是根据设计内容编写相应的功能代码。开始搭建网站之前，还需要确认使用哪种开发技术完成项目开发，比如网站的前后端是否分离，前后端分别采用哪些框架实现等。本项目采用前后端不分离模式开发，后端使用 Django 3.0 + MySQL 8.0 实现，前端使用 layui 框架 + jQuery 实现网页设计。

前后端不分离模式要求前端开发人员提供静态的 HTML 模板，并且 HTML 模板实现简单的 JavaScript 脚本功能，如果涉及 Ajax 异步数据传输，则需要在开发阶段中与后端人员相互调试 API 接口的数据结构。

我们将项目命名为 babys，在 Windows 的 CMD 窗口输入 Django 的项目创建指令，然后在新建的项目中创建项目应用（App）index、commodity 和 shopper，具体操作如下所示。

```
# 将 CMD 当前路径切换到 F 盘
C:\WINDOWS\system32>f:
# 在 F 盘创建项目 babys
F:\>django-admin startproject babys
# 将路径切换到项目 babys
F:\>cd babys
# 分别创建项目应用（App）index、commodity 和 shopper
F:\babys>python manage.py startapp index
F:\babys>python manage.py startapp commodity
F:\babys>python manage.py startapp shopper
```

打开项目 babys，分别创建文件夹 media、pstatic 和 templates，整个项目的目录结构如图 2-8 所示。

图 2-8　目录结构

整个项目共有 7 个文件夹和 1 个文件，每个文件夹和文件的功能说明如下：

（1）babys 文件夹与项目名相同，该文件夹下含有文件__init__.py、asgi.py、settings.py、urls.py 和 wsgi.py

（2）commodity 是 Django 创建的项目应用（App），文件夹含有__init__.py、admin.py、apps.py、models.py、tests.py 和 views.py 文件，它主要实现网站的商品列表页和商品详细页。

（3）index 是 Django 创建的项目应用（App），文件夹含有的文件与项目应用（App）commodity 相同，它主要实现网站首页。

（4）media 是网站的媒体资源，用于存放商品的主图和详细介绍图。

（5）pstatic 是网站的静态资源，用于存放网站的静态资源文件，如 CSS、JavaScript 和网站界面图片。

（6）shopper 也是 Django 创建的项目应用（App），它主要实现网站的购物车页面、个人中心页面、用户登录注册页面、在线支付功能等。

（7）templates 用于存放 HTML 模板文件，即网站的网页文件。

（8）manage.py 是项目的命令行工具，内置多种方法与项目进行交互。在命令提示符窗口下，将路径切换到项目 babys 并输入 python manage.py help，可以查看该工具的指令信息。

由于文件夹 media、pstatic 和 templates 是我们自行创建的，因此还需要在这些文件夹中添加前端提供的 HTML 静态模板，详细的添加说明如下：

```
# media 文件夹分别创建文件夹 details 和 imgs
media
  |—details
  |—imgs
# pstatic 文件夹分别创建文件夹 css、img、js 和 layui
# css、img、js 和 layui 文件夹含有多个文件
pstatic
  |—css
    |—main.css
  |—img
    |—多张网站页面的设计图
  |—js
```

```
      |—car.js
      |—mm.js
    |—layui
      |—layui 框架的源码文件
# templates 文件夹存放 6 个 HTML 文件
templates
  |—index.html（网站首页）
  |—login.html（用户注册登录页面）
  |—commodity.html（商品列表页面）
  |—details.html（商品详细页面）
  |—shopcart.html（购物车页面）
  |—shopper.html（个人中心页面）
```

至此，我们已完成项目 babys 的整体架构搭建，整个搭建过程可以分为两个步骤，说明如下：

（1）使用指令创建 Django 项目，并在新建的项目下创建相应的项目应用（App）。

（2）根据前端提供的 HTML 静态模板，分别创建文件夹 media、pstatic 和 templates，并将 HTML 静态模板的 CSS、JavaScript 和 HTML 文件分别放置在文件夹 pstatic 和 templates。

2.4 项目的功能配置

由于文件夹 media、pstatic 和 templates 是我们自行创建的，Django 在运行中无法识别这些文件夹的具体作用，因此，我们还需要在 Django 的配置文件 settings.py 中添加这些文件夹，使 Django 在运行中能识别这些文件夹的作用。

使用 PyCharm 打开项目 babys，然后打开 babys 文件夹的 settings.py 文件，如图 2-9 所示。

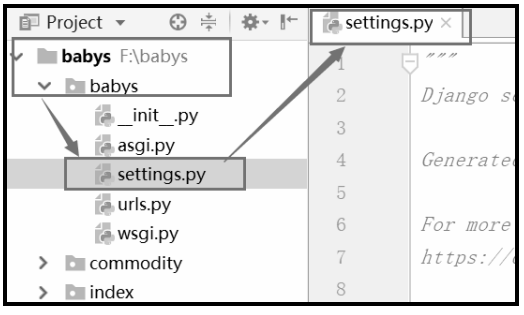


图 2-9 打开 settings.py

Django 已为我们设置了一些默认的配置信息，比如项目路径、密钥配置、域名访问权限、App 列表和中间件等。以项目 babys 为例，settings.py 的默认配置如下：

```
import os
# 项目路径
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
```

```
# 密钥配置
SECRET_KEY = 'tq%piy+c3pntex*%)7m&1xoo1&1hb6cp2v42eyqaj)%f%jxc&5'
# 调试模式
DEBUG = True
# 域名访问权限
ALLOWED_HOSTS = []
# App 列表
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
# 中间件
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
# 路由入口设置
ROOT_URLCONF = 'babys.urls'
# 模板配置
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
# WSGI 配置
WSGI_APPLICATION = 'babys.wsgi.application'
```

```
    # 数据库配置
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.sqlite3',
            'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
        }
    }
    # 内置 Auth 认证的功能配置
    AUTH_PASSWORD_VALIDATORS = [
        {
        'NAME':
        'django.contrib.auth.password_validation.
UserAttributeSimilarityValidator',
        },
        {
        'NAME':'django.contrib.auth.password_validation.
MinimumLengthValidator',
        },
        {
        'NAME':'django.contrib.auth.password_validation.
CommonPasswordValidator',
        },
        {
        'NAME':'django.contrib.auth.password_validation.
NumericPasswordValidator',
        },
    ]
    # 国际化与本地化配置
    LANGUAGE_CODE = 'en-us'
    TIME_ZONE = 'UTC'
    USE_I18N = True
    USE_L10N = True
    USE_TZ = True
    # 静态资源配置
    STATIC_URL = '/static/'
```

上述代码列出了 13 个配置信息，每个配置信息的说明如下：

（1）项目路径 BASE_DIR：主要通过 os 模块读取当前项目在计算机系统的具体路径，该代码在创建项目时自动生成，一般情况下无须修改。

（2）密钥配置 SECRET_KEY：这是一个随机值，在项目创建的时候自动生成，一般情况下无须修改。主要用于重要数据的加密处理，提高项目的安全性，避免遭到攻击者恶意破坏。密钥主要用于用户密码、CSRF 机制和会话 Session 等数据加密。

- 用户密码：Django 内置一套 Auth 认证系统，该系统具有用户认证和存储用户信息等

功能，在创建用户的时候，将用户密码通过密钥进行加密处理，保证用户的安全性。

- CSRF 机制：该机制主要用于表单提交，防止窃取网站的用户信息来制造恶意请求。
- 会话 Session：Session 的信息存放在 Cookie 中，以一串随机的字符串表示，用于标识当前访问网站的用户身份，记录相关用户信息。

（3）调试模式 DEBUG：该值为布尔类型。如果在开发调试阶段，那么应设置为 True，在开发调试过程中会自动检测代码是否发生更改，根据检测结果执行是否刷新重启系统。如果项目部署上线，那么应将其改为 False，否则会泄漏项目的相关信息。

（4）域名访问权限 ALLOWED_HOSTS：设置可访问的域名，默认值为空列表。当 DEBUG 为 True 并且 ALLOWED_HOSTS 为空列表时，项目只允许以 localhost 或 127.0.0.1 在浏览器上访问。当 DEBUG 为 False 时，ALLOWED_HOSTS 为必填项，否则程序无法启动，如果想允许所有域名访问，可设置 ALLOW_HOSTS = ['*']。

（5）App 列表 INSTALLED_APPS：告诉 Django 有哪些 App。在项目创建时已有 admin、auth 和 sessions 等配置信息，这些都是 Django 内置的应用功能，各个功能说明如下：

- admin：内置的后台管理系统。
- auth：内置的用户认证系统。
- contenttypes：记录项目中所有 model 元数据（Django 的 ORM 框架）。
- sessions：Session 会话功能，用于标识当前访问网站的用户身份，记录相关用户信息。
- messages：消息提示功能。
- staticfiles：查找静态资源路径。

（6）中间件 MIDDLEWARE：这是一个用来处理 Django 的请求（Request）和响应（Response）的框架级别的钩子，它是一个轻量、低级别的插件系统，用于在全局范围内改变 Django 的输入和输出。

（7）路由入口设置 ROOT_URLCONF：告诉 Django 从哪个文件查找整个项目的路由信息（路由信息即我们定义的网址信息），默认值是与项目同名的文件夹的 urls.py 文件，即 babys 文件夹的 urls.py。

（8）模板配置 TEMPLATES：主要配置模板的解析引擎、模板的存放路径地址以及 Django 内置功能的模板使用配置信息。

（9）WSGI 配置 WSGI_APPLICATION：告诉 Django 如何查找 WSGI 文件，并从 WSGI 文件启动并运行 Django 系统服务，默认值是与项目同名的文件夹的 wsgi.py 文件，即 babys 文件夹的 wsgi.py。

（10）数据库配置 DATABASES：配置数据的连接信息，如连接数据库的模块、数据库名称、数据库的账号密码等，默认连接 sqlite 数据库。

（11）内置 Auth 认证的功能配置 AUTH_PASSWORD_VALIDATORS：主要实现 Django 的 Auth 认证系统的内置功能。

（12）国际化与本地化配置：包含配置属性 LANGUAGE_CODE、TIME_ZONE、

USE_I18N、USE_L10N、USE_TZ，主要实现网站的语言设置、不同时区的时间设置等。

（13）静态资源配置 STATIC_URL：设置静态文件的路径信息。

在网站开发阶段中，我们经常对配置文件 settings.py 的 INSTALLED_APPS、MIDDLEWARE、TEMPLATES、DATABASES 和 STATIC_URL 进行配置，从而完成网站的开发过程，而配置属性 DEBUG 和 ALLOWED_HOSTS 则用于网站上线阶段。

上述配置属性是 Django 默认的功能配置，在实际开发中，可根据实际情况适当添加或删除相应的功能配置。

2.4.1 添加项目应用

我们在项目 babys 添加了项目应用（App）index、commodity 和 shopper，但 Django 在运行过程中依然无法识别新增的项目应用（App），因此还需在 Django 的配置文件 settings.py 添加我们新增的项目应用（App）。在 App 列表 INSTALLED_APPS 分别添加 index、commodity 和 shopper，添加信息如下：

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'index',
    'commodity',
    'shopper'
]
```

在 App 列表 INSTALLED_APPS 添加项目应用（App）不用考虑添加顺序，一般情况下，新增的项目应用写在 App 列表 INSTALLED_APPS 末端，并且以字符串格式表示。

2.4.2 设置模板信息

在 Web 开发中，模板是一种较为特殊的 HTML 文档。这个 HTML 文档嵌入了一些能够让 Django 识别的变量和指令，然后由 Django 的模板引擎解析这些变量和指令，生成完整的 HTML 网页并返回给用户浏览。模板是 Django 里面的 MTV 框架模式的 T 部分，配置模板路径是为了告诉 Django 在解析模板时，如何找到模板所在的位置。

一般情况下，项目的根目录文件夹 templates 通常存放共用的模板文件，能为各个 App 的模板文件调用，这个模式符合代码重复使用的原则。我们已在项目 babys 创建了文件夹 templates，它是用来存放 Django 模板文件的，在配置文件 settings.py 的配置属性 TEMPLATES 添加文件夹 templates，配置信息如下：

```
TEMPLATES = [
    {
```

```
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates'),],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

模板配置以列表格式表示，每个元素具有不同的含义，其含义说明如下：

- BACKEND：定义模板引擎，用于识别模板里面的变量和指令。内置的模板引擎有 Django Templates 和 jinja2.Jinja2，每个模板引擎都有自己的变量和指令语法。
- DIRS：设置模板所在的路径，告诉 Django 在哪个地方查找模板的位置，默认为空列表。
- APP_DIRS：是否在 App 里查找模板文件。
- OPTIONS：用于填充在 RequestContext 的上下文（模板里面的变量和指令），一般情况下不做任何修改。

模板文件夹也可以在项目应用（App）里面创建，比如在项目应用 index 中创建模板文件夹 temps，那么在 TEMPLATES 的配置属性 DIRS 添加 os.path.join(BASE_DIR, 'index/temp')，其中 index/temp 代表项目应用 index 的模板文件夹 temps；并且配置属性 APP_DIRS 必须设置为 True，否则 Django 无法从项目应用中查找模板文件。

2.4.3　添加中间件

中间件（Middleware）是一个用来处理 Django 请求（Request）和响应（Response）的框架级别的钩子，它是一个轻量、低级别的插件系统，用于在全局范围内改变 Django 的输入和输出。

当用户在网站中进行某个操作时，这个过程是用户向网站发送 HTTP 请求（Request），而网站会根据用户的操作返回相关的网页内容，这个过程称为响应处理（Response）。从请求到响应的过程中，当 Django 接收到用户的请求时，首先经过中间件处理请求信息，执行相关的处理，然后将处理结果返回给用户。中间件的执行流程如图 2-10 所示。

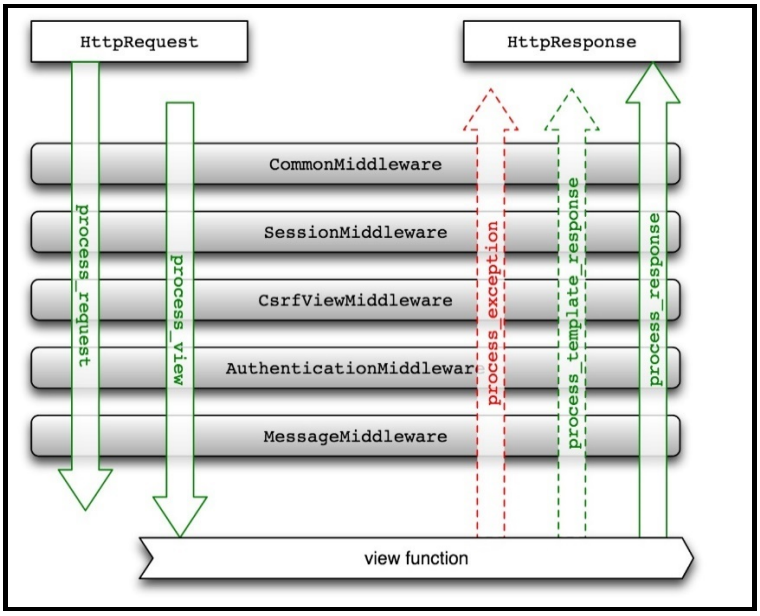


图 2-10　中间件的执行流程

从图 2-10 中能清晰地看到，中间件的作用是处理用户请求信息和返回响应内容。开发者可以根据自己的开发需求自定义中间件，只要将自定义的中间件添加到配置属性 MIDDLEWARE 中即可激活。

一般情况下，Django 默认的中间件配置均可满足大部分的开发需求。我们在项目的 MIDDLEWARE 中添加 LocaleMiddleware 中间件，使得 Django 内置的功能支持中文显示，代码如下：

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    # 添加中间件 LocaleMiddleware
    'django.middleware.locale.LocaleMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

配置属性 MIDDLEWARE 的数据格式为列表类型，每个中间件的设置顺序是固定的，如果随意变更中间件很容易导致程序异常。每个中间件的说明如下：

- SecurityMiddleware：内置的安全机制，保护用户与网站的通信安全。
- SessionMiddleware：会话 Session 功能。
- LocaleMiddleware：国际化和本地化功能。
- CommonMiddleware：处理请求信息，规范化请求内容。

- CsrfViewMiddleware：开启 CSRF 防护功能。
- AuthenticationMiddleware：开启内置的用户认证系统。
- MessageMiddleware：开启内置的信息提示功能。
- XFrameOptionsMiddleware：防止恶意程序单击劫持。

2.4.4　配置数据库

默认情况下，Django 支持使用 PostgreSQL、MySQL、Sqlite3 和 Oracle 数据库，如果要使用其他的数据库，如 MSSQL 或 Redis 等，需要自行安装第三方插件。配置属性 DATABASES 是设置项目所使用的数据库信息，不同的数据库需要设置不同的数据库引擎，数据库引擎用于实现项目与数据库的连接，Django 提供了 4 种数据库引擎：

- 'django.db.backends.postgresql'
- 'django.db.backends.mysql'
- 'django.db.backends.sqlite3'
- 'django.db.backends.oracle'

在创建项目的时候，Django 已默认使用 Sqlite3 数据库，配置文件 settings.py 的配置信息如下所示：

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

项目创建之后，如果没有修改配置属性 DATABASES，当启动并运行 Django 时，Django 会自动在项目的目录下创建数据库文件 db.sqlite3，如图 2-11 所示。

图 2-11　目录结构

由于项目 babys 需要使用 MySQL 数据库，因此在配置属性 DATABASES 中设置 MySQL 的连接信息。在配置数据库信息之前，首先确保本地计算机已安装 MySQL 数据库系统，然后再安装 MySQL 的连接模块，Django 可以使用 mysqlclient 和 pymysql 模块实现 MySQL 连接。

mysqlclient 模块可以使用 pip 指令安装，打开命令提示符窗口并输入安装指令 pip install mysqlclient，然后等待模板安装完成即可。

如果使用 pip 在线安装 mysqlclient 的过程中出现错误，还可以选择 whl 文件安装。在浏览器中访问 www.lfd.uci.edu/~gohlke/pythonlibs/#mysqlclient 并下载与 Python 版本相匹配的 mysqlclient 文件。我们将 mysqlclient 文件下载保存在 D 盘，然后打开命令提示符窗口，使用 pip 完成 whl 文件的安装，如下所示：

```
pip install D:\mysqlclient-1.4.6-cp38-cp38-win_amd64.whl
```

完成 mysqlclient 模块的安装后，在项目的配置文件 settings.py 中配置 MySQL 数据库连接信息，代码如下：

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'babys',
        'USER':'root',
        'PASSWORD':'1234',
        'HOST':'127.0.0.1',
        'PORT':'3306',
    }
}
```

为了验证数据库连接信息是否正确，我们使用数据库可视化工具 Navicat Premium 打开本地的 MySQL 数据库。在本地的 MySQL 数据库创建数据库 babys，如图 2-12 所示。

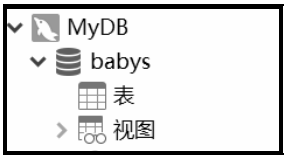


图 2-12　数据库 babys

刚创建的数据库 babys 是一个空白的数据库，接着在 PyCharm 的 Terminal 界面下输入 Django 操作指令 python manage.py migrate 来创建 Django 内置功能的数据表。

因为 Django 自带了内置功能，如 Admin 后台系统、Auth 用户系统和会话机制等功能，这些功能都需要借助数据表实现，所以该操作指令可以将内置的迁移文件生成数据表，如图 2-13 所示。

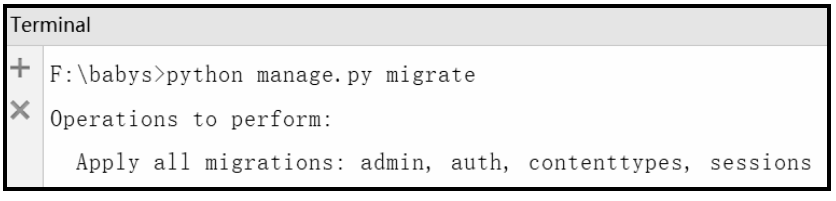


图 2-13　创建数据表

最后在数据库可视化工具 Navicat Premium 里查看数据库 babys 是否生成相应的数据表，如图 2-14 所示。

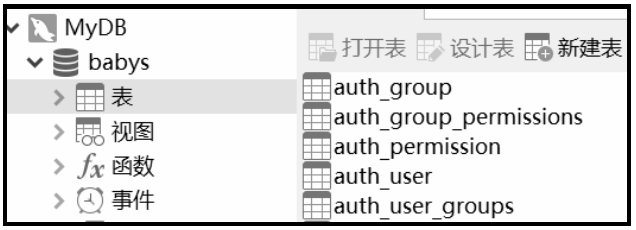


图 2-14　查看数据表

使用 mysqlclient 连接 MySQL 数据库时，Django 对 mysqlclient 版本有要求，打开 Django 的源码查看 mysqlclient 的版本要求，如图 2-15 所示。

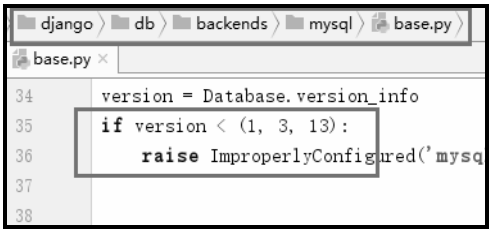


图 2-15　mysqlclient 版本要求

一般情况下，使用 pip 安装 mysqlclient 模块都能符合 Django 的使用要求。如果在开发过程中发现 Django 提示 mysqlclient 过低，那么可以对 Django 的源码进行修改，将图 2-15 的 if 条件判断注释即可。

除了使用 mysqlclient 模块连接 MySQL 之外，还可以使用 pymysql 模块连接 MySQL 数据库。pymysql 模块的安装使用 pip 在线安装即可，在命令提示符窗口下输入 pip install pymysql 指令并等待安装完成即可。

pymysql 模块安装成功后，项目配置文件 settings.py 的数据库配置信息无须修改，只要在 babys 文件夹的__init__.py 中设置数据库连接模块即可，代码如下：

```
# babys 文件夹的__init__.py
import pymysql
pymysql.install_as_MySQLdb()
```

若要验证 pymysql 模块连接 MySQL 数据库的功能，建议读者先将 mysqlclient 模块卸载，这样能排除干扰因素，而验证方式与 mysqlclient 模块连接 MySQL 的验证方式一致。记得在验证之前，务必将数据库 babys 的数据表删除，具体的验证过程不再重复讲述。

值得注意的是，如果读者使用的 MySQL 是 8.0 以上版本，在 Django 连接 MySQL 数据库时会提示 django.db.utils.OperationalError 的错误信息，这是因为 MySQL 8.0 版本的密码加密方式发生了改变，8.0 版本的用户密码采用的是 CHA2 加密方式。

为了解决这个问题，我们通过 SQL 语句将 8.0 版本的加密方式改回原来的加密方式，这样可以解决 Django 连接 MySQL 数据库的错误问题。在 MySQL 的可视化工具中运行以下 SQL

语句：

```
    # newpassword 是已设置的用户密码
    ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY
'newpassword';
    FLUSH PRIVILEGES;
```

Django 除了支持 PostgreSQL、SQLite3、MySQL 和 Oracle 之外，还支持 SQL Server 和 MongoDB 的连接。由于不同的数据库有不同的连接方式，因此此处不过多介绍，本书主要以 MySQL 连接为例，若需了解其他数据库的连接方式，可自行搜索相关资料。

2.4.5 配置静态资源

静态资源的配置分别由配置属性 STATIC_URL、STATICFILES_DIRS 和 STATIC_ROOT 完成，默认情况下，Django 只配置了配置属性 STATIC_URL。一个项目在开发过程中肯定需要使用 CSS 和 JavaScript 文件，这些静态文件的存放路径主要在配置文件 settings.py 设置，Django 默认的配置信息如下：

```
    # Static files (CSS, JavaScript, Images)
    # https://docs.djangoproject.com/en/2.0/howto/static-files/
    STATIC_URL = '/static/'
```

上述配置是设置静态资源的路由地址，其作用是使浏览器能成功访问 Django 的静态资源。默认情况下，Django 只能识别项目应用（App）的 static 文件夹里面的静态资源。当项目启动时，Django 会从项目应用（App）里面查找相关的资源文件，查找功能主要由 App 列表 INSTALLED_APPS 的 staticfiles 实现。

Django 在调试模式（DEBUG=True）下只能识别项目应用（App）的 static 文件夹里面的静态资源，并且项目应用（App）的 static 文件夹在创建项目应用的时候不会自动生成，开发者还需要自行在项目应用（App）里面创建 static 文件夹，如果该文件夹改为其他名字，Django 将无法识别；若将 static 文件夹放在 babys 的项目目录下，则 Django 也是无法识别的。

由于 STATIC_URL 的特殊性，在开发中会造成诸多不便，比如将静态文件夹存放在项目的根目录或者定义多个静态文件夹等。以项目 babys 为例，若想在网页上正常访问静态资源文件，可以将文件夹 pstatic 写入资源集合 STATICFILES_DIRS，在配置文件 settings.py 添加并设置配置属性 STATICFILES_DIRS，该属性以列表或元组的形式表示，设置方式如下：

```
    # 添加并设置配置属性 STATICFILES_DIRS
    STATICFILES_DIRS = (
        os.path.join(BASE_DIR, 'pstatic'),
    )
```

如果项目中有多个静态资源文件夹，并且这些文件夹不是在项目应用（App）里面；或者项目应用（App）的静态文件夹名称不是 static，那么我们只需在配置属性 STATICFILES_DIRS 添加对应的文件夹即可

静态资源配置还有 STATIC_ROOT，其作用是在服务器上部署项目，实现服务器和项目

之间的映射。STATIC_ROOT 主要收集整个项目的静态资源并存放在一个新的文件夹，然后由该文件夹与服务器之间构建映射关系。STATIC_ROOT 的配置如下：

```
STATIC_ROOT = os.path.join(BASE_DIR, 'AllStatic')
```

当项目的配置属性 DEBUG 设为 True 的时候，Django 会自动提供静态文件代理服务，此时整个项目处于开发阶段，因此无须使用 STATIC_ROOT。当配置属性 DEBUG 设为 False 的时候，意味着项目进入生产环境，Django 不再提供静态文件代理服务，此时需要在项目的配置文件中设置 STATIC_ROOT。

设置 STATIC_ROOT 需要使用 Django 操作指令 collectstatic 来收集所有的静态资源，这些静态资源会保存在 STATIC_ROOT 所设置的文件夹里。关于 STATIC_ROOT 的使用会在后续的章节详细讲述。

2.4.6　配置媒体资源

一般情况下，STATIC_URL 是设置静态文件的路由地址，如 CSS 样式文件、JavaScript 文件以及常用图片等。对于一些经常变动的资源，通常将其存放在媒体资源文件夹，如用户头像、商品主图、商品详细介绍图等。

媒体资源和静态资源是可以同时存在的，而且两者可以独立运行，互不影响，而媒体资源只有配置属性 MEDIA_URL 和 MEDIA_ROOT。以项目 babys 为例，新建的文件夹 media 是用来存放媒体资源文件的，在配置文件 settings.py 分别设置 MEDIA_URL 和 MEDIA_ROOT，使 Django 在运行的时候能够自动识别媒体资源文件夹 media，详细的设置方式如下：

```
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

配置属性设置后，还需要将 media 文件夹注册到 Django 里，让 Django 知道如何找到媒体文件，否则无法在浏览器中访问该文件夹的文件信息。打开 babys 文件夹的 urls.py 文件，为媒体文件夹 media 添加相应的路由地址，代码如下：

```
# babys 文件夹的 urls.py
from django.contrib import admin
from django.urls import path, re_path
from django.views.static import serve
from django.conf import settings

urlpatterns = [
    path('admin/', admin.site.urls),
    # 配置媒体资源的路由信息
    re_path('media/(?P<path>.*)', serve,
     {'document_root':settings.MEDIA_ROOT}, name='media'),
]
```

最后，我们启动运行项目 babys，在浏览器中分别访问 http://127.0.0.1:8000/static/css/main.css 和 http://127.0.0.1:8000/media/imgs/p1.jpg，前者是访问项目 babys 的静态资源文件夹

pstatic 的文件夹 CSS 的样式文件 main.css，后者是访问媒体资源文件夹 media 的文件夹 imgs 的图片文件 p1.jpg，如图 2-16 所示。

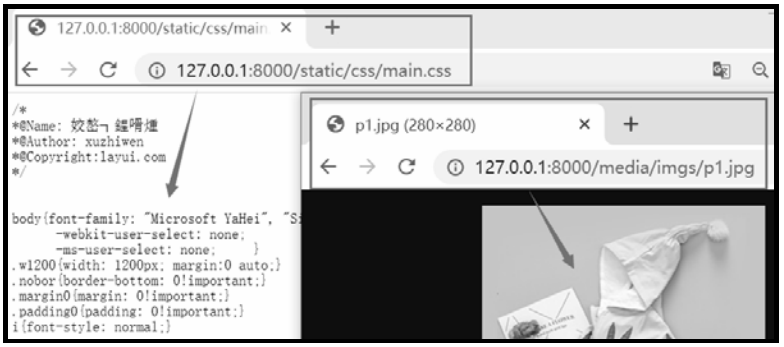


图 2-16　静态资源和媒体资源

2.5　内置指令

我们已掌握了 Django 的项目创建、项目开发调试和基本的功能配置，本节将讲述 Django 内置指令的详细作用，只有了解各个操作指令的功能，才能为我们的项目开发提供巨大的便利和帮助。

在 PyCharm 的 Terminal 或者 Windows 的 CMD 窗口（CMD 窗口路径必须在项目路径下，如项目 babys 的路径）中输入指令 python manage.py help 并按回车键，即可看到相关的指令信息，以 PyCharm 的 Terminal 为例，如图 2-17 所示。

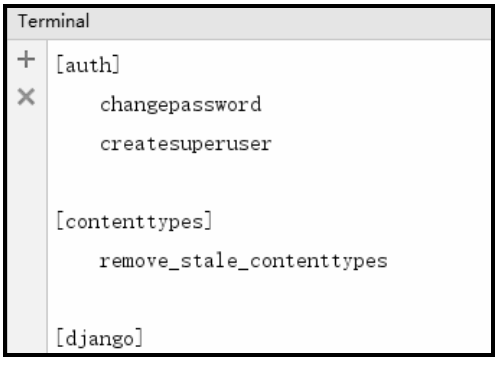


图 2-17　Django 指令信息

Django 的操作指令共有 30 条，每条指令的说明这里以表格形式展示，如表 2-5 所示。

表 2-5　Django 操作指令说明

指　令	说　明
changepassword	修改内置用户表的用户密码
createsuperuser	为内置用户表创建超级管理员账号

（续表）

指　令	说　明
remove_stale_contenttypes	删除数据库中已不使用的数据表
check	检测整个项目是否存在异常问题
compilemessages	编译语言文件，用于项目的区域语言设置
createcachetable	创建缓存数据表，为内置的缓存机制提供存储功能
dbshell	进入 Django 配置的数据库，可以执行数据库的 SQL 语句
diffsettings	显示当前 settings.py 的配置信息与默认配置的差异
dumpdata	导出数据表的数据并以 JSON 格式存储，如 python manage.py dumpdata index > data.json，这是 index 的模型所对应的数据导出，并保存在 data.json 文件中
flush	清空数据表的数据信息
inspectdb	获取项目所有模型的定义过程
loaddata	将数据文件导入数据表，如 python manage.py loaddatadata.json
makemessages	创建语言文件，用于项目的区域语言设置
makemigrations	从模型对象创建数据迁移文件并保存在 App 的 migrations 文件夹
migrate	根据迁移文件的内容，在数据库里生成相应的数据表
sendtestemail	向指定的收件人发送测试的电子邮件
shell	进入 Django 的 Shell 模式，用于调试项目功能
showmigrations	查看当前项目的所有迁移文件
sqlflush	查看清空数据库的 SQL 语句脚本
sqlmigrate	根据迁移文件内容输出相应的 SQL 语句
sqlsequencereset	重置数据表递增字段的索引值
squashmigrations	对迁移文件进行压缩处理
startapp	创建项目应用 App
startproject	创建新的 Django 项目
test	运行 App 里面的测试程序
testserver	新建测试数据库并使用该数据库运行项目
clearsessions	清除会话 Session 数据
collectstatic	收集所有的静态文件
findstatic	查找静态文件的路径信息
runserver	在本地计算机上启动 Django 项目

表 2-5 简单讲述了 Django 操作指令的作用，对于刚接触 Django 的读者来说，可能并不理解每个指令的具体作用，本节只对这些指令进行概述，读者只需要大概了解，在后续的学习中会具体讲述这些指令的使用方法。此外，有兴趣的读者也可以参考官方文档（docs.djangoproject.com/zh-hans/3.0/ref/django-admin/）。

2.6 本章小结

系统总体结构设计需要由需求工程师和开发人员共同商议，针对用户需求来商量如何设计系统各个功能模块以及各个模块的数据结构。本章商城网站的概要设计如下：

（1）网站首页应设有导航栏，并且所有功能展示在导航栏，在导航栏的下面展示各类的热销商品，当单击商品图片即可进入商品详细页面，导航栏上方设有搜索框，便于用户搜索相关商品。

（2）商品列表页将所有商品以一定的规则排序展示，用户可以从销量、价格、上架时间和收藏数量设置商品的排序方式，并且在页面的左侧设置分类列表，选择某一分类可以筛选出相应的商品信息。

（3）商品详细页展示某一商品的主图、名称、规格、数量、详细介绍、购买按钮和收藏按钮，并在商品详细介绍的左侧设置了热销商品列表。

（4）购物车页面只能在用户已登录的情况下才能访问，它是将用户选购的商品以列表形式展示，列表的每行数据包含了商品图片、名称、单价、数量、合计和删除操作，用户可以增减商品的购买数量，并且能自动计算费用。

（5）个人中心页面是展示用户的基本信息及订单信息，只能在用户已登录的情况下访问。

（6）用户登录和注册页面共用一个页面，如果用户账号已存在，则对用户账号密码验证并登录，如果用户不存在，则对当前的账号密码进行注册处理。

（7）数据库使用 MySQL 5.7 以上版本，数据表除了 Django 内置数据表之外，还需定义商品信息表、商品类别表、购物车信息表和订单信息表

整个项目共有 7 个文件夹和 1 个文件，每个文件夹和文件的功能说明如下：

（1）babys 文件夹与项目名相同，该文件夹下含有文件__init__.py、asgi.py、settings.py、urls.py 和 wsgi.py

（2）commodity 是 Django 创建的项目应用（App），文件夹含有__init__.py、admin.py、apps.py、models.py、tests.py 和 views.py 文件，它主要实现网站的商品列表页和商品详细页。

（3）index 是 Django 创建的项目应用（App），该文件夹含有的文件与项目应用（App）commodity 相同，它主要实现网站首页。

（4）shopper 也是 Django 创建的项目应用（App），它主要实现网站的购物车页面、个人中心页面、用户登录注册页面、在线支付功能等。

（5）media 是网站的媒体资源，用于存放商品的主图和详细介绍图。

（6）pstatic 是网站的静态资源，用于存放网站的静态资源文件，如 CSS、JavaScript 和网站界面图片。

（7）templates 用于存放 HTML 模板文件，即网站的网页文件。

（8）manage.py 是项目的命令行工具，内置多种方法与项目进行交互。在命令提示符窗

口下，将路径切换到项目 babys 并输入 python manage.py help，可以查看该工具的指令信息。

Django 已为我们设置了一些默认的配置信息，比如项目路径、密钥配置、域名访问权限、App 列表和中间件等，每个配置说明如下：

（1）项目路径 BASE_DIR：主要通过 os 模块读取当前项目在计算机系统的具体路径，该代码在创建项目时自动生成，一般情况下无须修改。

（2）密钥配置 SECRET_KEY：这是一个随机值，在项目创建的时候自动生成，一般情况下无须修改。主要用于重要数据的加密处理，提高项目的安全性，避免遭到攻击者恶意破坏。密钥主要用于用户密码、CSRF 机制和会话 Session 等数据加密。

（3）调试模式 DEBUG：该值为布尔类型。如果在开发调试阶段，那么应设置为 True，在开发调试过程中会自动检测代码是否发生更改，根据检测结果执行是否刷新重启系统。如果项目部署上线，那么应将其改为 False，否则会泄漏项目的相关信息。

（4）域名访问权限 ALLOWED_HOSTS：设置可访问的域名，默认值为空列表。当 DEBUG 为 True 并且 ALLOWED_HOSTS 为空列表时，项目只允许以 localhost 或 127.0.0.1 在浏览器上访问。当 DEBUG 为 False 时，ALLOWED_HOSTS 为必填项，否则程序无法启动，如果想允许所有域名访问，可设置 ALLOW_HOSTS = ['*']。

（5）App 列表 INSTALLED_APPS：告诉 Django 有哪些 App。在项目创建时已有 admin、auth 和 sessions 等配置信息，这些都是 Django 内置的应用功能。

（6）中间件 MIDDLEWARE：这是一个用来处理 Django 请求（Request）和响应（Response）的框架级别的钩子，它是一个轻量、低级别的插件系统，用于在全局范围内改变 Django 的输入和输出。

（7）路由入口设置 ROOT_URLCONF：告诉 Django 从哪个文件查找整个项目的路由信息，默认值是与项目同名的文件夹的 urls.py 文件，即 babys 文件夹的 urls.py。

（8）模板配置 TEMPLATES：主要配置模板的解析引擎、模板的存放路径地址以及 Django 内置功能的模板使用配置信息。

（9）WSGI 配置 WSGI_APPLICATION：告诉 Django 如何查找 WSGI 文件，并从 WSGI 文件启动并运行 Django 系统服务，默认值是与项目同名的文件夹的 wsgi.py 文件，即 babys 文件夹的 wsgi.py。

（10）数据库配置 DATABASES：配置数据的连接信息，如连接数据库的模块、数据库名称、数据库的账号密码等，默认连接 SQLite 数据库。

（11）内置 Auth 认证的功能配置 AUTH_PASSWORD_VALIDATORS：主要实现 Django 的 Auth 认证系统的内置功能。

（12）国际化与本地化配置：包含配置属性 LANGUAGE_CODE、TIME_ZONE、USE_I18N、USE_L10N、USE_TZ，主要实现网站的语言设置、不同时区的时间设置等。

（13）静态资源配置 STATIC_URL：设置静态文件的路径信息。

在 PyCharm 的 Terminal 或者 Windows 的 CMD 窗口（CMD 窗口路径必须在项目路径下，如项目 babys 的路径）中输入指令 python manage.py help 并按回车键。Django 的操作指令共有 30 条，读者必须了解每个指令的具体作用。

第 3 章

商城网址的规划与设计

路由称为 URL（Uniform Resource Locator，统一资源定位符），也可以称为 URLconf，是对可以从互联网上得到的资源位置和访问方法的一种简洁的表示，是互联网上标准资源的地址。互联网上的每个文件都有一个唯一的路由，用于指出网站文件的路径位置。简单地说，路由可视为我们常说的网址，每个网址代表不同的网页。

3.1 设置路由分发规则

一个完整的路由包含：路由地址、视图函数（或者视图类）、路由变量和路由命名。其中基本的信息必须有：路由地址和视图函数（或者视图类），路由地址即我们常说的网址；视图函数（或者视图类）即项目应用（App）的 views.py 文件所定义的函数或类；路由变量和路由命名是路由的变量和命名设置，使路由具有动态变化和命名引用功能。（动态变化是指一个路由地址按照某个规律演变多种不同的路由地址；命名引用是指在视图、模型等其他项目文件使用路由命名生成相应的路由地址）

在默认情况下，设置路由地址是在项目同名的文件夹的 urls.py 文件里实现，这也是由配置文件 settings.py 的 ROOT_URLCONF 决定，以项目 babys 为例，配置属性 ROOT_URLCONF 指向 babys 文件夹的 urls.py，如图 3-1 所示。

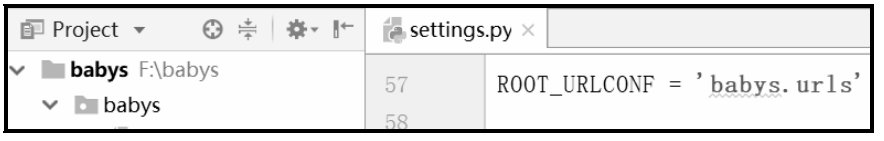


图 3-1　配置属性 ROOT_URLCONF

一个项目中可能设有多个项目应用（App），而 babys 文件夹的 urls.py 是定义项目所有路由地址的总入口，如果项目中所有路由地址都在 babys 文件夹的 urls.py 中定义，当项目功能规模越来越大的时候，babys 文件夹的 urls.py 定义的路由地址就会越来越多，从而造成难以管理的问题。

为了更好区分各个项目应用（App）的路由地址，我们在 babys 文件夹的 urls.py 中分别为每个项目应用（App）定义一条路由入口。首先在每个项目应用（App）的文件夹里创建 urls.py 文件，以项目 babys 的项目应用 index 为例，其目录结构如图 3-2 所示。

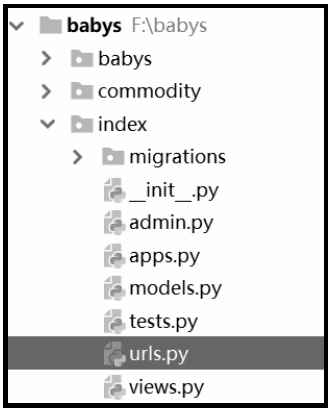


图 3-2　目录结构

除了项目应用 index 之外，我们还要在项目应用 shopper 和 commodity 分别创建新的 urls.py 文件。然后在 PyCharm 里打开 babys 文件夹的 urls.py 文件，将项目应用 index、shopper 和 commodity 新建的 urls.py 添加到 babys 文件夹的 urls.py，添加方法由 Django 内置函数 path 和 include 实现，详细代码如下：

```
    # babys 文件夹的 urls.py
    from django.contrib import admin
    from django.urls import path, include, re_path
    from django.views.static import serve
    from django.conf import settings

    urlpatterns = [
        path('admin/', admin.site.urls),
    # 添加项目应用 index、commodity 和 shopper 的 urls.py
        path('', include(('index.urls', 'index'), namespace='index')),
        path('commodity', include(('commodity.urls', 'commodity'),
         namespace='commodity')),
        path('shopper',include(('shopper.urls', 'shopper'),
namespace='shopper')),
        # 配置媒体资源的路由信息
        re_path('media/(?P<path>.*)', serve,
    {'document_root':settings.MEDIA_ROOT}, name='media'),
    ]
```

babys 文件夹的 urls.py 定义了 5 条路由信息，分别是 Admin 站点管理、首页地址（项目应用 index 的 urls.py）、商品信息（项目应用 commodity 的 urls.py）、购物车信息（项目应用 shopper 的 urls.py）和媒体资料。其中，Admin 站点管理在创建项目时已自动生成，一般情况下无须更改；媒体资料的路由地址是在第 2.4.6 节配置的。整个 babys 文件夹的 urls.py 的代码说明如下：

- from django.contrib import admin：导入内置 Admin 功能模块。
- from django.urls import path,include：导入 Django 的路由函数模块。
- urlpatterns：代表整个项目的路由集合，以列表格式表示，每个元素代表一条路由信息。
- path('admin/', admin.site.urls)：设定 Admin 管理系统的路由信息。'admin/'代表 127.0.0.1:8000/admin 的路由地址，admin 后面的斜杠是路径分隔符，其作用等同于计算机中文件目录的斜杠符号；admin.site.urls 指向内置 Admin 功能所自定义的路由信息，可以在 Python 目录 Lib\site-packages\django\contrib\admin\sites.py 中找到具体的定义过程。
- path('', include(('index.urls', 'index'), namespace='index'))：路由地址为“/”，即 127.0.0.1:8000，通常是网站的首页；路由函数 include 是将该路由地址分发给项目应用 index 的 urls.py 处理。
- path('commodity', include(('commodity.urls', 'commodity'), namespace='commodity'))：路由地址为“/commodity”，即 127.0.0.1:8000/commodity，这是商品详细和商品列表页面；路由函数 include 是将该路由地址分发给项目应用 commodity 的 urls.py 处理。
- path('shopper', include(('shopper.urls', 'shopper'), namespace='shopper'))：路由地址为“/shopper”，即 127.0.0.1:8000/shopper，这是购物车、个人中心、用户注册等页面；路由函数 include 是将该路由地址分发给项目应用 shopper 的 urls.py 处理。
- re_path('media/(?P<path>.*)', serve, {'document_root':settings.MEDIA_ROOT}, name='media')：路由地址为“/media/xxx”，即 127.0.0.1:8000/media/xxx，这是媒体资源定义的路由地址，路由函数 re_path 表示允许在路由地址里面设置正则表达式。

从 babys 文件夹的 urls.py 定义的路由信息得知，每个项目应用（App）的路由地址交给项目应用的 urls.py 自行管理，这是路由的分发规则，使路由按照一定的规则进行分类管理。整个路由设计模式的工作原理说明如下：

（1）当运行 babys 项目时，Django 从 babys 文件夹的 urls.py 找到各个项目应用（App）的 urls.py，然后读取每个项目应用（App）的 urls.py 定义的路由信息，从而生成完整的路由列表。

（2）用户在浏览器上访问某个路由地址时，Django 就会收到该用户的请求信息。

（3）Django 从当前请求信息中获取路由地址，并在路由列表里匹配相应的路由信息，再执行路由信息所指向的视图函数（或视图类），从而完成整个请求响应过程。

3.2　路由分发详解

从 3.1 节看到，我们设置项目路由分发功能的时候，除了使用内置函数 path 和 include 之外，还在路由中设置了参数 namespace，该参数是可选参数，是 Django 设置路由的命名空间。

路由函数 include 设有参数 arg 和 namespace ，参数 arg 指向项目应用 App 的 urls.py 文件，其数据格式以元组或字符串表示；可选参数 namespace 是路由的命名空间。

若要对路由设置参数 namespace，则参数 arg 必须以元组格式表示，并且元组的长度必须为 2。以路由 path('', include(('index.urls', 'index'), namespace='index')) 为例，参数 arg 为 ('index.urls', 'index')，参数的每个元素说明如下：

- 第一个元素为项目应用的 urls.py 文件，比如('index.urls','index')的“index.urls”，这是代表项目应用 index 的 urls.py 文件。
- 第二个元素可以自行命名，但不能为空，一般情况下是以项目应用的名称进行命名，如('index.urls','index')的“index”是以项目应用 index 进行命名的。

如果路由设置参数 namespace 并且参数 arg 为字符串或元组长度不足 2 的时候，比如我们将首页的路由分发设为 path('', include(('index.urls'), namespace='index'))，当运行项目的时候，Django 就会提示错误信息，如图 3-3 所示。

```
  File "F:\babys\babys\urls.py", line 23, in <module>
    path('', include(('index.urls'), namespace='index')),
  File "E:\Python\lib\site-packages\django\urls\conf.py", line 38, in include
    raise ImproperlyConfigured(
django.core.exceptions.ImproperlyConfigured: Specifying a namespace in include()
```

图 3-3　运行结果

路由函数 include 的作用是将当前路由分配到某个项目应用的 urls.py 文件，而项目应用的 urls.py 文件可以设置多条路由，这种情况类似计算机上的文件夹 A，并且该文件夹下包含多个子文件夹，而 Django 的命名空间 namespace 相当于对文件夹 A 进行命名。

假设项目路由设计为：在 babys 文件夹的 urls.py 定义 3 条路由，每条路由都使用路由函数 include，并分别命名为 A、B、C，每条路由对应某个项目应用的 urls.py 文件，并且每个项目应用的 urls.py 文件里定义若干条路由。

根据上述的路由设计模式，将 babys 文件夹的 urls.py 视为计算机上的 D 盘，在 D 盘下有 3 个文件夹，分别命名为 A、B、C，每个项目应用的 urls.py 所定义的若干条路由可视为这 3 个文件夹里面的文件。在这种情况下，Django 的命名空间 namespace 等同于文件夹 A、B、C 的文件名。

Django 的命名空间 namespace 可以为我们快速定位某个项目应用的 urls.py，再结合路由

命名 name 就能快速地从项目应用的 urls.py 找到某条路由的具体信息，这样就能有效管理整个项目的路由列表。有关路由函数 include 的定义过程，可以在 Python 安装目录下找到源码（Lib\site-packages\django\urls\conf.py）进行解读。

3.3 设置商城的路由地址

我们已在 babys 文件夹的 urls.py 分别为项目应用 index、shopper 和 commodity 设置路由分发功能，本节将会在项目应用 index、shopper 和 commodity 的 urls.py 定义网站首页、商品列表页、商品详细页、购物车页面、个人中心页面和用户登录注册页面的路由地址。

首先打开项目应用 index 的 urls.py，在该文件中定义网站首页的路由地址，定义方法如下：

```
# index 的 urls.py
from django.urls import path
from .views import *

urlpatterns = [
    path('', indexView, name='index'),
]
```

上述代码中，我们只在项目应用 index 的 urls.py 定义了路由地址 index，路由地址由 Django 内置函数 path 完成定义过程，函数 path 设置了 3 个参数，每个参数的说明如下：

（1）第一个参数为空字符串，这是设置具体的路由地址，由于 babys 文件夹的 urls.py 的路由分发为 path('', include(('index.urls', 'index'), namespace='index'))，即代表网址 127.0.0.1:8000，而 index 的 urls.py 定义的路由地址 index 设为空字符串，那么路由地址 index 的网址为 127.0.0.1:8000。

（2）第二个参数为 indexView，这是指向项目应用 index 的 views.py 的某个视图函数或视图类，当用户在浏览器访问 127.0.0.1:8000 的时候，Django 将接收到一个 HTTP 请求，从该请求中获取路由地址并与自身的路由列表进行匹配，如果路由地址匹配成功，Django 将 HTTP 请求交给路由地址指向的某个视图函数或视图类进行业务处理。

（3）第三个参数为 name='index'，这是函数 path 的可选参数，该参数是命名路由地址。实际开发中必须为每个路由地址进行命名，可以在视图或模板中使用路由名称生成相应的路由地址。

下一步打开项目应用 commodity 的 urls.py，在该文件中定义商品列表页和商品详细页的路由地址，详细的定义过程如下：

```
# 项目应用 commodity 的 urls.py
from django.urls import path
from .views import *
```

```
urlpatterns = [
    path('.html', commodityView, name='commodity'),
    path('/detail.<int:id>.html', detailView, name='detail'),
]
```

上述代码分别定义了商品列表页的路由地址 commodity 和商品详细页的路由地址 detail，路由地址的定义说明如下：

（1）项目应用 commodity 的 urls.py 路由空间是 path('commodity', include(('commodity.urls', 'commodity'), namespace='commodity'))，因此路由 commodity 为 127.0.0.1:8000/commodity.html，路由 detail 为 127.0.0.1:8000/commodity/detail/id.html。

（2）路由 detail 设置了路由变量 id，该变量以整数型表示，它可以代表 1、2、3……等整数，变量 id 对应商品信息表的主键 id，通过改变变量 id 的数值可以查看不同商品的详细介绍。

（3）路由地址的末端设置了“.html”，这是一种伪静态 URL 技术，可将网址设置为静态网址，用于 SEO 搜索引擎的爬取，如百度、谷歌等。此外，在末端设置“.html”是为变量 id 设置终止符，假如末端没有设置“.html”，并且路由变量为字符串类型，在浏览器上输入无限长的字符串，路由也能正常访问。

（4）路由 commodity 和 detail 的业务逻辑处理分别指向项目应用 commodity 的 views.py 定义的视图函数 commodityView 和 detailView。

最后打开项目应用 shopper 的 urls.py，在该文件中定义个人中心页、购物车信息页、用户登录注册页和用户注销的路由地址，详细代码如下：

```
# 项目应用 shopper 的 urls.py
from django.urls import path
from .views import *

urlpatterns = [
    path('.html', shopperView, name='shopper'),
    path('/login.html', loginView, name='login'),
    path('/logout.html', logoutView, name='logout'),
    path('/shopcart.html', shopcartView, name='shopcart'),
]
```

上述代码定义了 4 条路由地址，每个路由所对应的功能说明如下：

（1）路由 shopper 代表个人中心页，它的路由空间是 path('shopper', include(('shopper.urls', 'shopper'), namespace='shopper'))，因此路由地址为 127.0.0.1:8000/shopper.html，个人中心页的业务逻辑由项目应用 shopper 的 views.py 定义的视图函数 shopperView 实现。

（2）路由 login 代表用户登录注册页，路由地址为 127.0.0.1:8000/shopper/login.html，它的业务逻辑由项目应用 shopper 的 views.py 定义的视图函数 loginView 实现。

（3）路由 logout 实现个人中心的用户注销功能，路由地址为 127.0.0.1:8000/shopper/logout.html，它的业务逻辑由项目应用 shopper 的 views.py 定义的视图函数 logoutView 实现。

（4）路由 shopcart 代表购物车信息页，路由地址为 127.0.0.1:8000/shopper/shopcart.html，它的业务逻辑由项目应用 shopper 的 views.py 定义的视图函数 shopcartView 实现。

3.4 路由的定义规则

在 3.3 节看到，我们已在项目应用 index、shopper 和 commodity 的 urls.py 中定义了网站首页、商品列表页、商品详细页、购物车页面、个人中心页面、用户注销和用户登录注册页面的路由地址。综合分析得知，路由地址的定义规则如下：

（1）每个 urls.py 文件的路由地址必须在列表 urlpatterns 里定义，换句话说，每个 urls.py 必须设有一个列表 urlpatterns，该列表是用于定义路由信息。

（2）每条路由是由函数 path 定义，函数 path 设置了 3 个参数：第一个参数是设置具体的路由地址；第二个参数是指向项目应用的 views.py 的某个视图函数或视图类，负责处理路由的业务逻辑；第三个参数为 name='index'，这是函数 path 的可选参数，该参数是命名路由地址。

（3）如果函数 path 第二个参数使用内置函数 include，该路由是实现路由分发功能。也就是说，如果函数 path 的第二个参数是函数 include，该路由为路由分发；如果函数 path 的第二个参数是项目应用的 views.py 的视图类或视图函数，该路由为网站的路由地址。

函数 path 是 Django 2.0 以上版本定义的内置函数，如果开发环境是 Django 1.X 版本，那么路由定义应使用函数 url。从参数的角度分析，函数 path 和函数 url 的参数设置是相同的，只不过函数 url 定义的路由地址需设置路由符号 ^和 $。^代表当前路由地址的相对路径；$代表当前路由地址的终止符。

我们以项目应用 commodity 的 urls.py 为例，对路由 commodity 和 detail 使用的函数 url 进行定义，定义过程如下所示。

```
# 项目应用 commodity 的 urls.py
from django.conf.urls import url
from .views import *

urlpatterns = [
    url('^.html$', commodityView, name='commodity'),
    url('^/detail.(?P<id>\d+).html$', detailView, name='detail'),
]
```

从上述代码看到，函数 url 要为每个路由地址设置路由符号^和$，而且路由变量 id 应使用正则表达式表示（如(?P<id>\d+)）。综上所述，Django 1 的路由规则是使用 Django 的 url 函数实现路由定义，并且路由地址设有路由符号^和$，读者需要区分路由符号^和$的作用与使用规则，在某种程度上，它比 Django 2 版本复杂并且代码可读性差，因此 Django 1 的路由规则应该会在 Django 以后的新版本里逐渐淘汰。

3.5　路由变量与正则表达式

路由 detail 在路由地址里设置了路由变量 id，通过动态改变路由变量 id 的数值就能生成相应的商品详细介绍页面，Django 的路由变量分为字符类型、整型、slug 和 uuid，最为常用的是字符类型和整型。各个类型说明如下：

- 字符类型：匹配任何非空字符串，但不含斜杠。如果没有指定类型，就默认使用该类型。
- 整型：匹配 0 和正整数。
- slug：可理解为注释、后缀或附属等概念，常作为路由的解释性字符。可匹配任何 ASCII 字符以及连接符和下画线，能使路由更加清晰易懂。比如网页的标题是“13 岁的孩子”，其路由地址可以设置为“13-sui-de-hai-zi”。
- uuid：匹配一个 uuid 格式的对象。为了防止冲突，规定必须使用“-”并且所有字母必须小写，例如 075194d3-6885-417e-a8a8-6c931e272f00。

在路由中，如果使用函数 path 定义路由，那么路由变量则使用变量符号“<>”定义。在括号里面以冒号划分为两部分，冒号前面代表的是变量的数据类型，冒号后面代表的是变量名，变量名可自行命名，如果没有设置变量的数据类型，就默认为字符类型。比如路由变量<year>、<int:month>和<slug:day>，变量说明如下：

- <year>：变量名为 year，数据格式为字符类型，与<str:year>的含义一样。
- <int:month>：变量名为 month，数据格式为整型。
- <slug:day>：变量名为 day，数据格式为 slug。

除了在路由地址设置变量外，Django 还支持在路由地址外设置变量（路由的可选变量），比如在路由 detail 的路由中添加可选变量 user，如下所示：

```
# 项目应用 commodity 的 urls.py
from django.urls import path
from .views import *

urlpatterns = [
    path('.html', commodityView, {'user': 'admin'}, name='commodity'),
    path('/detail.<int:id>.html', detailView, name='detail'),
]
```

从上述代码可以看出，可选变量 user 的设置规则如下：

- 可选变量只能以字典的形式表示。
- 设置的可选变量只能在视图函数中读取和使用。
- 字典的一个键值对代表一个可选变量，键值对的键代表变量名，键值对的值代表变

量值。

- 变量值没有数据格式限制，可以为某个实例对象、字符串或列表（元组）等。
- 可选变量必须在视图函数（视图类）和参数 name 之间。

不管我们在路由地址中添加路由变量或者添加可选变量，只要路由信息里设置了变量，都必须在对应的视图函数里设置对应的函数参数，并且函数参数必须与路由信息的变量名一一对应。

虽然路由变量可以使用字符类型、整型、slug 和 uuid 表示，但某些路由变量会因为业务需求或实际情况而设置一定的范围值，比如变量<int:year>，该变量代表年份，年份都是由 4 位数字组成，而整型的数值可以长达 10 位。为了进一步规范路由变量的数据格式，可以使用正则表达式限制路由变量的取值范围，示例如下：

```
    # 某项目应用的 urls.py
    from django.urls import re_path
    from . import views
    urlpatterns = [
    re_path('(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/(?P<day>[0-9]{2}).html',
views.mydate)
    ]
```

路由的正则表达式是由路由函数 re_path 定义的，其作用是对路由变量进行截取与判断，正则表达式是以小括号为单位的，每个小括号的前后可以使用斜杠或者其他字符将其分隔与结束。以上述代码为例，分别将变量 year、month 和 day 以斜杠隔开，每个变量以一个小括号为单位，在小括号内，可分为 3 部分，以(?P<year>[0-9]{4})为例。

- ?P 是固定格式，字母 P 必须为大写。
- <year>为变量名。
- [0-9]{4}是正则表达式的匹配模式，代表变量的长度为 4，只允许取 0~9 的值。

3.6 本章小结

路由称为 URL（Uniform Resource Locator，统一资源定位符），也可以称为 URLconf，是对可以从互联网上得到的资源位置和访问方法的一种简洁的表示，是互联网上标准资源的地址。互联网上的每个文件都有一个唯一的路由，用于指出网站文件的路径位置。简单地说，路由可视为我们常说的网址，每个网址代表不同的网页。

一个完整的路由包含：路由地址、视图函数（或者视图类）、路由变量和路由命名。其中基本的信息必须有：路由地址和视图函数（或者视图类），路由地址即我们常说的网址；视图函数（或者视图类）即项目应用（App）的 views.py 文件所定义的函数或类；路由变量和路由命名是路由的变量和命名设置，使路由具有动态变化和命名引用功能。（动态变化是指一个路由地址按照某个规律演变多种不同的路由地址；命名引用是指在视图、模型等其他项目文

件使用路由命名生成相应的路由地址）

路由函数 include 的作用是将当前路由分配到某个项目应用的 urls.py 文件，而项目应用的 urls.py 文件可以设置多条路由，这种情况类似计算机上的文件夹 A，并且该文件夹下包含多个子文件夹，而 Django 的命名空间 namespace 相当于对文件夹 A 进行命名。

假设项目路由设计为：在 babys 文件夹的 urls.py 定义 3 条路由，每条路由都使用路由函数 include，并分别命名为 A、B、C，每条路由对应某个项目应用的 urls.py 文件，并且每个项目应用的 urls.py 文件里定义若干条路由。

根据上述的路由设计模式，将 babys 文件夹的 urls.py 视为计算机上的 D 盘，在 D 盘下有 3 个文件夹，分别命名为 A、B、C，每个项目应用的 urls.py 所定义的若干条路由可视为这 3 个文件夹里面的文件。在这种情况下，Django 的命名空间 namespace 等同于文件夹 A、B、C 的文件名。

路由函数 path 是定义项目的路由信息，定义规则如下：

（1）每个 urls.py 文件的路由地址必须在列表 urlpatterns 里定义，换句话说，每个 urls.py 必须设有一个列表 urlpatterns，该列表是用于定义路由信息。

（2）每条路由是由函数 path 定义，函数 path 设置了 3 个参数：第一个参数是设置具体的路由地址；第二个参数是指向项目应用的 views.py 的某个视图函数或视图类，负责处理路由的业务逻辑；第三个参数为 name='index'，这是函数 path 的可选参数，该参数是命名路由地址。

（3）如果函数 path 第二个参数使用内置函数 include，该路由是实现路由分发功能。也就是说，如果函数 path 的第二个参数是函数 include，该路由为路由分发；如果函数 path 的第二个参数是项目应用的 views.py 的视图类或视图函数，该路由为网站的路由地址。

Django 的路由变量分为字符类型、整型、slug 和 uuid，最为常用的是字符类型和整型。各个类型说明如下：

- 字符类型：匹配任何非空字符串，但不含斜杠。如果没有指定类型，就默认使用该类型。
- 整型：匹配 0 和正整数。
- slug：可理解为注释、后缀或附属等概念，常作为路由的解释性字符。可匹配任何 ASCII 字符以及连接符和下画线，能使路由更加清晰易懂。比如网页的标题是“13 岁的孩子”，其路由地址可以设置为“13-sui-de-hai-zi”。
- uuid：匹配一个 uuid 格式的对象。为了防止冲突，规定必须使用“-”并且所有字母必须小写，例如 075194d3-6885-417e-a8a8-6c931e272f00。

路由的正则表达式是由路由函数 re_path 定义的，其作用是对路由变量进行截取与判断，正则表达式是以小括号为单位的，每个小括号的前后可以使用斜杠或者其他字符将其分隔与结束。以上述代码为例，分别将变量 year、month 和 day 以斜杠隔开，每个变量以一个小括号为单位，在小括号内，可分为 3 部分，以(?P<year>[0-9]{4})为例：

- ?P 是固定格式，字母 P 必须为大写。
- <year>为变量名。
- [0-9]{4}是正则表达式的匹配模式，代表变量的长度为4，只允许取0~9的值。

第 4 章

商城的数据模型搭建与使用

Django 对各种数据库提供了很好的支持，包括 PostgreSQL、MySQL、SQLite 和 Oracle，而且为这些数据库提供了统一的 API 方法，这些 API 统称为 ORM 框架。通过使用 Django 内置的 ORM 框架可以实现数据库连接和读写操作。

ORM 框架是一种程序技术，用于实现面向对象编程语言中不同类型系统的数据之间的转换。从效果上说，它创建了一个可在编程语言中使用的“虚拟对象数据库”，通过对虚拟对象数据库的操作从而实现对目标数据库的操作，虚拟对象数据库与目标数据库是相互对应的。

4.1 定义商城的数据模型

在 2.2 节中，我们已设计了项目 babys 的数据结构，用户信息表是由 Django 内置用户管理功能定义，除此之外，项目还需要定义商品信息表、商品类别表、购物车信息表和订单信息表。我们将商品信息表和商品类别表定义在项目应用 commodity 的 models.py；购物车信息表和订单信息表定义在项目应用 shopper 的 models.py。

首先打开项目应用 commodity 的 models.py 文件，在文件中定义模型 Types 和 CommodityInfos，它们以类的形式表示，并且继承父类 Model，详细的定义过程如下：

```
# 项目应用 commodity 的 models.py
from django.db import models

class Types(models.Model):
    id = models.AutoField(primary_key=True)
    firsts = models.CharField('一级类型', max_length=100)
```

```
    seconds = models.CharField('二级类型', max_length=100)

    def __str__(self):
        return str(self.id)

    class Meta:
        verbose_name = '商品类型'
        verbose_name_plural = '商品类型'

class CommodityInfos(models.Model):
    id = models.AutoField(primary_key=True)
    name = models.CharField('商品名称', max_length=100)
    sezes = models.CharField('颜色规格', max_length=100)
    types = models.CharField('商品类型', max_length=100)
    price = models.FloatField('商品价格')
    discount = models.FloatField('折后价格')
    stock = models.IntegerField('存货数量')
    sold = models.IntegerField('已售数量')
    likes = models.IntegerField('收藏数量')
    created = models.DateField('上架日期', auto_now_add=True)
    img = models.FileField('商品主图', upload_to=r'imgs')
    details = models.FileField('商品介绍', upload_to=r'details')

    def __str__(self):
        return str(self.id)

    class Meta:
        verbose_name = '商品信息'
        verbose_name_plural = '商品信息'
```

最后再打开项目应用 shopper 的 models.py 文件，在文件中定义模型 CartInfos 和 OrderInfos，模型的定义过程与模型 Types 和 CommodityInfos 相似，如下所示：

```
# 项目应用 shopper 的 models.py
from django.db import models

STATE = (
    ('待支付', '待支付'),
    ('已支付', '已支付'),
    ('发货中', '发货中'),
    ('已签收', '已签收'),
    ('退货中', '退货中'),
)

class CartInfos(models.Model):
    id = models.AutoField(primary_key=True)
```

```
    quantity = models.IntegerField('购买数量')
    commodityInfos_id = models.IntegerField('商品 ID')
    user_id = models.IntegerField('用户 ID')

    def __str__(self):
        return str(self.id)

    class Meta:
        verbose_name = '购物车'
        verbose_name_plural = '购物车'

class OrderInfos(models.Model):
    id = models.AutoField(primary_key=True)
    price = models.FloatField('订单总价')
    created = models.DateField('创建时间', auto_now_add=True)
    user_id = models.IntegerField('用户 ID')
    state = models.CharField('订单状态', max_length=20, choices=STATE)

    def __str__(self):
        return str(self.id)

    class Meta:
        verbose_name = '订单信息'
        verbose_name_plural = '订单信息'
```

从模型的定义过程分析归纳得知，模型定义可以分为三部分，每个部分的功能说明如下：

（1）定义模型字段，每个模型字段对应数据表的某个表字段，字段以 aa= models.bb(cc) 格式表示，比如 id = models.AutoField(primary_key=True)，其中 id 为模型字段名称，它与数据表的表字段相互对应；models.AutoField 是设置字段的数据类型，常用类型有整型、字符型或浮点型等；primary_key=True 是设置字段属性，例如字段是否为表主键、限制内容长度、设置默认值等。

在实际开发中，我们需要定义不同的字段类型来满足各种开发需求，因此 Django 划分了多种字段类型，在源码目录 django\db\models\fields 的__init__.py 和 files.py 文件里找到各种模型字段，说明如下：

- AutoField：自增长类型，数据表的字段类型为整数，长度为 11 位。
- BigAutoField：自增长类型，数据表的字段类型为 bigint，长度为 20 位。
- CharField：字符类型。
- BooleanField：布尔类型。
- CommaSeparatedIntegerField：用逗号分隔的整数类型。
- DateField：日期（Date）类型。
- DateTimeField：日期时间（Datetime）类型。

- Decimal：十进制小数类型。
- EmailField：字符类型，存储邮箱格式的字符串。
- FloatField：浮点数类型，数据表的字段类型变成 Double 类型。
- IntegerField：整数类型，数据表的字段类型为 11 位的整数。
- BigIntegerField：长整数类型。
- IPAddressField：字符类型，存储 Ipv4 地址的字符串。
- GenericIPAddressField：字符类型，存储 Ipv4 和 Ipv6 地址的字符串。
- NullBooleanField：允许为空的布尔类型。
- PositiveIntegerFiel：正整数的整数类型。
- PositiveSmallIntegerField：小正整数类型，取值范围为 0~32767。
- SlugField：字符类型，包含字母、数字、下画线和连字符的字符串。
- SmallIntegerField：小整数类型，取值范围为-32,768~+32,767。
- TextField：长文本类型。
- TimeField：时间类型，显示时分秒 HH:MM[:ss[.uuuuuu]]。
- URLField：字符类型，存储路由格式的字符串。
- BinaryField：二进制数据类型。
- FileField：字符类型，存储文件路径的字符串。
- ImageField：字符类型，存储图片路径的字符串。
- FilePathField：字符类型，从特定的文件目录选择某个文件。

在不同的字段类型中，我们还可以设置字段的基本属性，比如 primary_key=True 是将字段设置为主键，每个字段都具有共同的基本属性，如下所示：

- verbose_name：默认为 None，在 Admin 站点管理设置字段的显示名称。
- primary_key：默认为 False，若为 True，则将字段设置成主键。
- max_length：默认为 None，设置字段的最大长度。
- unique：默认为 False，若为 True，则设置字段的唯一属性。
- blank：默认为 False，若为 True，则字段允许为空值，数据库将存储空字符串。
- null：默认为 False，若为 True，则字段允许为空值，数据库表现为 NULL。
- db_index：默认为 False，若为 True，则以此字段来创建数据库索引。
- default：默认为 NOT_PROVIDED 对象，设置字段的默认值。
- editable：默认为 True，允许字段可编辑，用于设置 Admin 的新增数据的字段。
- serialize：默认为 True，允许字段序列化，可将数据转化为 JSON 格式。
- unique_for_date：默认为 None，设置日期字段的唯一性。
- unique_for_month：默认为 None，设置日期字段月份的唯一性。
- unique_for_year：默认为 None，设置日期字段年份的唯一性。
- choices：默认为空列表，设置字段的可选值。
- help_text：默认为空字符串，用于设置表单的提示信息。
- db_column：默认为 None，设置数据表的列名称，若不设置，则将字段名作为数据表的列名。

- db_tablespace：默认为 None，如果字段已创建索引，那么数据库的表空间名称将作为该字段的索引名。注意：部分数据库不支持表空间。
- auto_created：默认为 False，若为 True，则自动创建字段，用于一对一的关系模型。
- validators：默认为空列表，设置字段内容的验证函数。
- error_messages：默认为 None，设置错误提示。

（2）重写函数__str__()，这是设置模型的返回值，默认情况下，返回值为模型名+主键。函数__str__可用于外键查询，比如模型 A 设有外键字段 F，外键字段 F 关联模型 B，当查询模型 A 时，外键字段 F 会将模型 B 的函数__str__返回值作为字段内容。

需要注意的是，函数__str__只允许返回字符类型的字段，如果字段是整型或日期类型的，就必须使用 Python 的 str()函数将其转化成字符类型。

（3）重写 Meta 选项，这是设置模型的常用属性，一共设有 19 个属性，每个属性的说明如下：

- abstract：若设为 True，则该模型为抽象模型，不会在数据库里创建数据表。
- app_label：属性值为字符串，将模型设置为指定的项目应用，比如将 index 的 models.py 定义的模型 A 指定到其他 App 里。
- db_table：属性值为字符串，设置模型所对应的数据表名称。
- db_teblespace：属性值为字符串，设置模型所使用数据库的表空间。
- get_latest_by：属性值为字符串或列表，设置模型数据的排序方式。
- managed：默认值为 True，支持 Django 命令执行数据迁移；若为 False，则不支持数据迁移功能。
- order_with_respect_to：属性值为字符串，用于多对多的模型关系，指向某个关联模型的名称，并且模型名称必须为英文小写。比如模型 A 和模型 B，模型 A 的一条数据对应模型 B 的多条数据，两个模型关联后，当查询模型 A 的某条数据时，可使用 get_b_order()和 set_b_order()来获取模型 B 的关联数据，这两个方法名称的 b 为模型名称小写，此外 get_next_in_order()和 get_previous_in_order()可以获取当前数据的下一条和上一条的数据对象。
- ordering：属性值为列表，将模型数据以某个字段进行排序。
- permissions：属性值为元组，设置模型的访问权限，默认设置添加、删除和修改的权限。
- proxy：若设为 True，则为模型创建代理模型，即克隆一个与模型 A 相同的模型 B。
- required_db_features：属性值为列表，声明模型依赖的数据库功能。比如['gis_enabled']，表示模型依赖 GIS 功能。
- required_db_vendor：属性值为列表，声明模型支持的数据库，默认支持 SQLite、PostgreSQL，MySQL 和 Oracle。
- select_on_save：数据新增修改算法，通常无须设置此属性，默认值为 False。
- indexes：属性值为列表，定义数据表的索引列表。
- unique_together：属性值为元组，多个字段的联合唯一，等于数据库的联合约束。
- verbose_name：属性值为字符串，设置模型直观可读的名称并以复数形式表示。

- verbose_name_plural：与 verbose_name 相同，以单数形式表示。
- label：只读属性，属性值为 app_label.object_name，如 index 的模型 PersonInfo，值为 index.PersonInfo。
- label_lower：与 label 相同，但其值为字母小写，如 index.personinfo。

综上所述，模型字段、函数__str__和 Meta 选项是模型定义的基本要素，模型字段的类型、函数__str__和 Meta 选项的属性设置需由开发需求而定。在定义模型时，还可以在模型里定义相关函数，比如 get_absolute_url()，当视图类没有设置属性 success_url 时，视图类的重定向路由地址将由模型定义的 get_absolute_url()提供。

4.2　数据迁移创建数据表

数据迁移是将项目里定义的模型生成相应的数据表，首次在项目里定义模型时，项目所配置的数据库里并没有创建任何数据表，想要通过模型创建数据表，可使用 Django 的操作指令完成创建过程。以项目 babys 为例，在配置文件 settings.py 的 DATABASES 属性配置 MySQL 数据库连接信息，连接本地的 MySQL 数据库系统，如下所示：

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'babys',
        'USER': 'root',
        'PASSWORD': '1234',
        'HOST': '127.0.0.1',
        'PORT': '3306',
    }
}
```

下一步是注释项目应用 index、commodity 和 shopper 的 urls.py 定义的路由信息，由于网站的路由地址尚未定义相应的视图函数，因此无法使用 Django 内置指令创建数据表。然后在 PyCharm 的 Terminal 输入 Django 的操作指令，如下所示：

```
F:\babys>python manage.py makemigrations
Migrations for 'commodity':
  commodity\migrations\0001_initial.py
    - Create model CommodityInfos
    - Create model Types
Migrations for 'shopper':
  shopper\migrations\0001_initial.py
    - Create model CartInfos
    - Create model OrderInfos
```

当 makemigrations 指令执行成功后，在项目应用 commodity 和 shopper 的 migrations 文件

夹里分别创建 0001_initial.py 文件，如果项目里有多个 App，并且每个项目应用的 models.py 文件里定义了模型对象，当首次执行 makemigrations 指令时，Django 就在每个项目应用的 migrations 文件夹里创建 0001_initial.py 文件。打开查看项目应用 commodity 的 migrations 的 0001_initial.py 文件，文件内容如图 4-1 所示。

```
class Migration(migrations.Migration):
    initial = True
    dependencies = [
    ]
    operations = [
        migrations.CreateModel(
            name='CommodityInfos',
            fields=[
                ('id', models.AutoField(primary_key=True, serialize=False)),
                ('name', models.CharField(max_length=100, verbose_name='商品名称')),
                ('sezes', models.CharField(max_length=100, verbose_name='颜色规格')),
```

图 4-1　0001_initial.py 文件内容

0001_initial.py 文件将 models.py 定义的模型生成数据表的脚本代码，该文件的脚本代码可被 migrate 指令执行，migrate 指令会根据脚本代码的内容在数据库里创建相应的数据表，只要在 PyCharm 的 Terminal 窗口下输入 migrate 指令即可完成数据表的创建，代码如下：

```
    F:\babys>python manage.py migrate
    Operations to perform:
      Apply all migrations: admin, auth, commodity, contenttypes, sessions,
shopper
    Running migrations:
      Applying contenttypes.0001_initial... OK
```

当指令运行完成后，使用数据库可视化工具打开数据库就能看到新建的数据表，以数据库可视化工具 Navicat Premium 为例，如图 4-2 所示。其中数据表 commodity_types 由项目应用 commodity 定义的模型 Types 创建，而其他数据表是 Django 内置的功能所使用的数据表，分别是会话 Session、用户认证管理和 Admin 后台系统等。

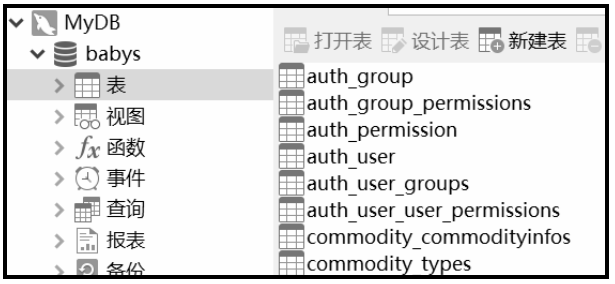
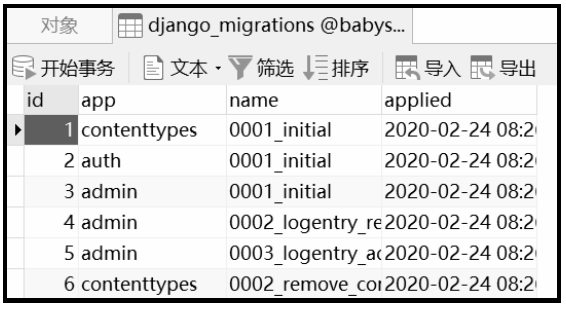


图 4-2　数据表

在开发过程中，开发者因为开发需求而经常调整数据表的结构，比如新增功能、优化现有功能等。假如在上述例子里新增模型 Payment 及其数据表，为了保证不影响现有的数据表，如何通过新增的模型创建相应的数据表？

针对上述问题，我们只需在某个项目应用的 models.py 里定义新的模型 Payment，然后再

次执行 makemigrations 和 migrate 指令即可。第二次执行 makemigrations 的时候，新定义的模型会在项目应用的 migrations 文件夹里创建新的 py 文件，当再次运行 migrate 指令的时候，Django 会执行 migrations 文件夹中新建的 py 文件，每次执行 migrate 的操作记录都会记录在内置数据表 django_migrations 中，如图 4-3 所示。



id	app	name	applied
1	contenttypes	0001_initial	2020-02-24 08:2
2	auth	0001_initial	2020-02-24 08:2
3	admin	0001_initial	2020-02-24 08:2
4	admin	0002_logentry_re	2020-02-24 08:2
5	admin	0003_logentry_a	2020-02-24 08:2
6	contenttypes	0002_remove_co	2020-02-24 08:2

图 4-3　数据表 django_migrations

除了新建模型及其数据表之外，makemigrations 和 migrate 指令还支持模型的修改，从而修改相应的数据表结构，比如新增、修改和删除数据表的某个字段。字段的增删改只需在已有的模型中重新定义即可，然后再次执行 makemigrations 和 migrate 指令。

但在已有模型中新增字段，模型字段必须将属性 null 和 blank 设为 True 或者为模型字段设置默认值（设置属性 default），否则执行 makemigrations 指令会提示字段修复信息，如图 4-4 所示。

```
F:\babys>python manage.py makemigrations
You are trying to add a non-nullable field 'titl
ws).
Please select a fix:
 1) Provide a one-off default now (will be set
 2) Quit, and let me add a default in models.py
```

图 4-4　字段修复信息

migrate 指令还可以单独执行某个 .py 文件，首次在项目中使用 migrate 指令时，Django 会默认创建内置功能的数据表，如果只想执行项目应用 commodity 的 migrations 文件夹的某个 .py 文件，那么可以在 migrate 指令里指定文件名，指令如下：

```
F:\babys>python manage.py migrate commodity 0001_initial
Operations to perform:
  Target specific migration: 0001_initial, from commodity
Running migrations:
  Applying commodity.0001_initial... OK
```

在 migrate 指令末端设置项目应用名称 commodity 和 migrations 文件夹的 0001_initial 文件名，三者（migrate 指令、项目应用名称 commodity 和 0001_initial 文件名）之间使用空格隔开即可，指令执行完成后，数据库只有数据表 django_migrations、commodity_types 和

commodity_commodityinfos，如图 4-5 所示。

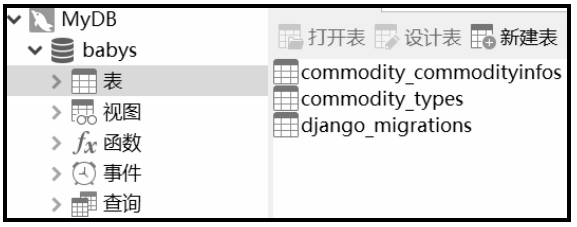


图 4-5　数据表信息

我们知道，migrate 指令根据 migrations 文件夹的 .py 文件创建数据表，但在数据库里，数据表的创建和修改离不开 SQL 语句的支持，因此 Django 提供了 sqlmigrate 指令，该指令能将 .py 文件转化成相应的 SQL 语句。以项目应用 commodity 的 0001_initial.py 文件为例，在 PyCharm 的 Terminal 窗口输入 sqlmigrate 指令，指令末端必须设置项目应用名称和 migrations 文件夹的某个 .py 文件名，三者之间使用空格隔开即可，指令输出结果如图 4-6 所示。

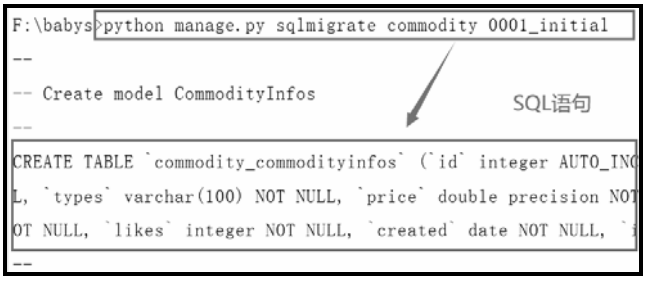


图 4-6　sqlmigrate 指令

除此之外，Django 还提供了很多数据迁移指令，如 squashmigrations、inspectdb、showmigrations、sqlflush、sqlsequencereset 和 remove_stale_contenttypes，这些指令在 2.5 节里已说明过了，此处不再重复讲述。

4.3　数据的导入与导出

在实际开发过程中，我们经常对数据库的数据进行导入和导出操作，比如网站重构、数据分析和网站分布式部署等。一般情况下，我们使用数据库可视化工具来实现数据的导入和导出，以 Navicat Premium 为例，打开某个数据表，单击“导入”或“导出”按钮，按照操作提示即可完成，如图 4-7 所示。

使用数据库可视化工具导入某个表的数据时，如果当前数据表设有外键字段，就必须将外键字段关联的数据表的数据导入，再执行当前数据表的数据导入操作，否则数据无法导入成功。因为外键字段指向它所关联的数据表，如果关联的数据表没有数据，外键字段就无法与关联的数据表生成数据关系，从而使当前数据表的数据导入失败。

图 4-7　数据的导入与导出

除了使用数据库可视化工具实现数据的导入与导出之外，Django 还为我们提供了操作指令（loaddata 和 dumpdata）来实现数据的导入与导出操作。以项目 babys 为例，在数据表 commodity_types 和 commodity_commodityinfos 中分别添加数据，如图 4-8 所示。

图 4-8　数据表 commodity_types 和 commodity_commodityinfos

在 PyCharm 的 Terminal 窗口输入 dumpdata 指令，将整个项目的数据从数据库里导出并保存到 data.json 文件，其指令如下：

```
F:\babys>python manage.py dumpdata>data.json
```

dumpdata 指令末端使用了符号“>”和文件名 data.json，这是将项目所有的数据都存放在 data.json 文件中，并且 data.json 的文件路径在项目的根目录（与项目的 manage.py 文件在同一个路径），如图 4-9 所示。

图 4-9　目录结构

如果只想导出某个项目应用的所有数据或者项目应用里某个模型的数据，那么可在 dumpdata 指令末端设置项目名称或项目名称的某个模型名称，指令如下：

```
# 导出项目应用 commodity 的所有模型的数据
```

```
python manage.py dumpdata commodity>data.json
# 导出项目应用 commodity 的模型 Types 的数据
python manage.py dumpdata commodity.Types>types.json
```

一般情况下，使用 dumpdata 指令导出的数据文件都存放在项目的根目录，因为在输入指令时，PyCharm 的 Terminal 窗口的命令行所在路径为项目的根目录，若想更换存放路径，则可改变命令行的当前路径，比如将数据文件存放在 D 盘，其指令如下：

```
# 将命令行路径切换到 D 盘
F:\babys>cd ..
# 命令行在 D 盘路径下使用项目 babys 的 manage 文件执行 dumpdata 指令
F:\>python babys/manage.py dumpdata>data.json
```

若想将导出的数据文件重新导入数据库里，则可使用 loaddata 指令完成，该指令使用方式相对单一，只需在指令末端设置需要导入的文件名即可：

```
F:\babys>python manage.py loaddata data.json
Installed 68 object(s) from 1 fixture(s)
```

loaddata 指令根据数据文件的 model 属性来确定当前数据所属的数据表，并将数据插入数据表，从而完成数据导入。

一般情况下，数据的导出和导入最好以整个项目或整个项目应用的数据为单位，因为数据表之间可能存在外键关联，如果只导入某张数据表的数据，就必须考虑该数据表是否设有外键，并且外键所关联的数据表是否已有数据。

4.4 使用 QuerySet 操作数据

Django 对数据库的数据进行增、删、改操作是借助内置 ORM 框架所提供的 API 方法实现的，简单来说，ORM 框架的数据操作 API 是在 QuerySet 类里面定义的，然后由开发者自定义的模型对象调用 QuerySet 类，从而实现数据操作。

4.4.1 新增数据

Django 提供了多种数据新增方法，开发者可以根据实际情况以及个人使用习惯选择某一种新增方式。为了更好地演示数据库的增、删、改操作，在项目 babys 使用 Shell 模式（启动命令行和执行脚本）进行讲述，该模式方便开发人员开发和调试程序。在 PyCharm 的 Terminal 下开启 Shell 模式，输入 python manage.py shell 指令即可，如图 4-10 所示。

```
F:\babys>python manage.py shell
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 
Type "help", "copyright", "credits" or "license"
(InteractiveConsole)
```

图 4-10　Shell 模式

在 Shell 模式下，若想对数据表 commodity_types 新增数据，则可输入以下代码实现：

```
>>> from commodity.models import Types
>>> t = Types()
>>> t.firsts = '童装'
>>> t.seconds = '女装'
>>> t.save()
# 数据新增后，查看新增数据的主键 id
>>> t.id
```

上述代码是对项目应用 commodity 的模型 Types 进行实例化，再对实例化对象的属性进行赋值，从而实现数据表 commodity_types 的数据新增，代码说明如下：

（1）从项目应用 commodity 的 models.py 文件中导入模型 Types。

（2）对模型 Types 声明并实例化，生成对象 t。

（3）对对象 t 的属性进行逐一赋值，对象 t 的属性来自于模型 Types 所定义的字段。完成赋值后，再由对象 t 调用 save 方法进行数据保存。

代码运行结束后，在数据表 commodity_types 里查看数据的新增情况，如图 4-11 所示。

图 4-11　数据入库

除了上述方法外，数据新增还有以下 3 种常见方法，代码如下：

```
# 方法一
# 使用 create 方法实现数据新增
>>> t = Types.objects.create(firsts='儿童用品', seconds='婴儿车')
# 数据新增后，获取新增数据的主键 id
>>> t.id
# 方法二
# 同样使用 create 方法，但数据以字典格式表示
>>> d = dict(firsts='奶粉辅食', seconds='磨牙饼干')
>>> t = Types.objects.create(**d)
# 数据新增后，获取新增数据的主键 id
```

```
>>> t.id
# 方法三
# 在实例化时直接设置属性值
>>> t = Types(firsts='儿童早教', seconds='童话故事')
>>> t.save()
# 数据新增后，获取新增数据的主键 id
>>> t.id
```

在执行数据新增时，为了保证数据的有效性，我们需要对数据进行去重判断，确保数据不会重复新增。以往的方案都是对数据表进行查询操作，如果查询的数据不存在，就执行数据新增操作。为了简化这一过程，Django 提供了 get_or_create 方法，使用如下：

```
>>> d = dict(firsts='奶粉辅食', seconds='营养品')
>>> t = Types.objects.get_or_create(**d)
# 数据新增后，获取新增数据的主键 id
>>> t[0].id
```

get_or_create 根据每个模型字段的值与数据表的数据进行判断，判断方式如下：

- 只要有一个模型字段的值与数据表的数据不相同（除主键之外），就会执行数据新增操作。
- 如果每个模型字段的值与数据表的某行数据完全相同，就不执行数据新增，而是返回这行数据的数据对象，比如对上述的字典 d 重复执行 get_or_create，第一次是执行数据新增（若执行结果显示为 True，则代表数据新增），第二次是返回数据表已有的数据信息(若执行结果显示为 False，则数据表已存在数据，不再执行数据新增)，如图 4-12 所示。

```
>>> d = dict(firsts='奶粉辅食', seconds='营养品')
>>> Types.objects.get_or_create(**d)
(<Types: 4>, True)
>>> Types.objects.get_or_create(**d)
(<Types: 4>, False)
```

图 4-12　执行结果

除了 get_or_create 之外，Django 还定义了 update_or_create 方法，这是判断当前数据在数据表里是否存在，若存在，则进行更新操作，否则在数据表里新增数据，使用说明如下：

```
# 第一次是新增数据
>>> d = dict(firsts='儿童早教', seconds='儿童玩具')
>>> t = Types.objects.update_or_create(**d)
>>> t
(<Types: 5>, True)
# 第二次是修改数据
>>> t = Types.objects.update_or_create(**d,defaults={'firsts': '教育资料'})
>>> t[0].title
```

update_or_create 是根据字典 d 的内容查找数据表的数据，如果能找到相匹配的数据，就执行数据修改，修改内容以字典格式传递给参数 defaults 即可；如果在数据表找不到匹配的数据，就将字典 d 的数据新增到数据表里。

如果要对某个模型执行数据批量新增操作，那么可以使用 bulk_create 方法实现，只需将数据对象以列表或元组的形式传入 bulk_create 方法即可：

```
>>> t1 = Types(firsts='儿童用品', seconds='湿纸巾')
>>> t2 = Types(firsts='儿童用品', seconds='纸尿裤')
>>> ojb_list = [t1, t2]
>>> Types.objects.bulk_create(ojb_list)
```

在使用 bulk_create 之前，数据类型为模型 Types 的实例化对象，并且在实例化过程中设置每个字段的值，最后将所有实例化对象放置在列表或元组里，以参数的形式传递给 bulk_create，从而实现数据的批量新增操作。

4.4.2　更新数据

更新数据的步骤与数据新增的步骤大致相同，唯一的区别在于数据对象来自数据表，因此需要执行一次数据查询，查询结果以对象的形式表示，并将对象的属性进行赋值处理，代码如下：

```
>>> t = Types.objects.get(id=1)
>>> t.firsts = '儿童用品'
>>> t.save()
```

上述代码获取数据表 commodity_types 里主键 id 等于 1 的数据对象 t，然后修改数据对象 t 的 firsts 属性，从而完成数据修改操作。打开数据表 commodity_types 查看数据修改情况，如图 4-13 所示。

图 4-13　数据表 commodity_types

除此之外，我们还可以使用 update 方法实现数据更新，使用方法如下：

```
# 批量更新一条或多条数据，查询方法使用 filter
# filter 以列表格式返回，查询结果可能是一条或多条数据
>>> Types.objects.filter(id=1).update(seconds='男装')
# 更新数据以字典格式表示
>>> d= dict(seconds='童鞋')
>>> Types.objects.filter(id=1).update(**d)
```

```
# 不使用查询方法，默认对全表的数据进行更新
>>> Types.objects.update(firsts='母婴用品')
# 使用内置 F 方法实现数据的自增或自减
# F 方法还可以在 annotate 或 filter 方法里使用
>>> from django.db.models import F
>>> t = Types.objects.filter(id=1)
# 将 id 字段原有的数据自增加 10，自增或自减的字段必须为数字类型
>>> t.update(id=F('id')+10)
```

在 Django 2.2 或以上版本新增了数据批量更新方法 bulk_update，它的使用与批量新增方法 bulk_create 相似，使用说明如下：

```
# 新增两行数据
>>> t1 = Types.objects.create(firsts='奶粉辅食', seconds='纸尿片')
>>> t2 = Types.objects.create(firsts='儿童用品', seconds='进口奶粉')
# 修改字段 firsts 和 seconds 的数据
>>> t1.firsts = '儿童用品'
>>> t2.seconds = '婴儿车'
# 批量修改字段 firsts 和 seconds 的数据
>>> Types.objects.bulk_update([t1,t2],fields=['firsts','seconds'])
```

4.4.3 删除数据

删除数据有 3 种方式：删除数据表的全部数据、删除一行数据和删除多行数据，实现方式如下：

```
# 删除数据表中的全部数据
>>> Types.objects.all().delete()
# 删除一条 id 为 1 的数据
>>> Types.objects.get(id=1).delete()
# 删除多条数据
>>> Types.objects.filter(firsts='儿童用品').delete()
```

删除数据的过程中，如果删除的数据设有外键字段，就会同时删除外键关联的数据。比如我们在项目应用 index 的 models.py 定义模型 PersonInfo 和 Vocation，如下所示：

```
from django.db import models

class PersonInfo(models.Model):
    id = models.AutoField(primary_key=True)
    name = models.CharField(max_length=20)
    age = models.IntegerField()
    hireDate = models.DateField()

    def __str__(self):
        return self.name
    class Meta:
```

```
            verbose_name = '人员信息'

    class Vocation(models.Model):
        id = models.AutoField(primary_key=True)
        job = models.CharField(max_length=20)
        title = models.CharField(max_length=20)
        payment = models.IntegerField(null=True, blank=True)

name=models.ForeignKey(PersonInfo,on_delete=models.CASCADE,related_name='ps')

        def __str__(self):
            return str(self.id)
        class Meta:
            verbose_name = '职业信息'
```

如果删除模型 PersonInfo 里主键等于 3 的数据（简称为数据 A），那么在模型 Vocation 中，有些数据（简称为数据 B）关联了数据 A，在删除数据 A 时，也会同时删除数据 B。假设模型 Vocation 中有 3 行数据（数据 B）关联了模型 PersonInfo 的数据 A，数据删除过程如下所示：

```
>>> PersonInfo.objects.get(id=3).delete()
# 删除结果，共删除 4 条数据
# 其中 Vocation 删除了 3 条数据，PersonInfo 删除了 1 条数据
>>> (4, {'index.Vocation': 3, 'index.PersonInfo': 1})
```

从模型 Vocation 得知，外键字段的参数 on_delete 用于设置数据删除模式，比如模型 Vocation 的外键字段 name 设为 CASCADE 模式，不同的删除模式会影响数据删除结果，说明如下：

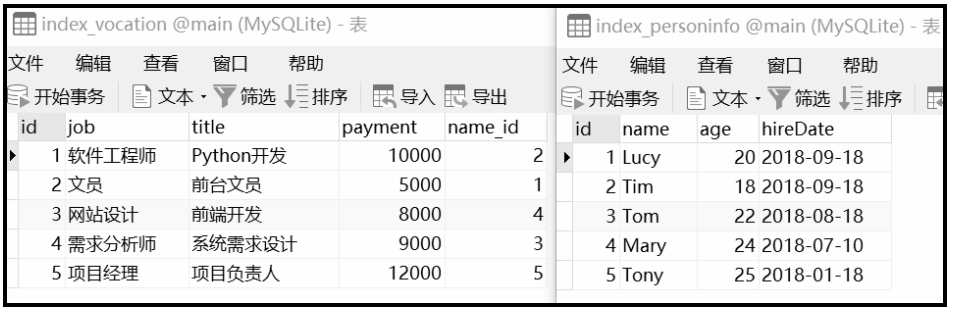
- PROTECT 模式：如果删除的数据设有外键字段并且关联其他数据表的数据，就提示数据删除失败。
- SET_NULL 模式：执行数据删除并把其他数据表的外键字段设为 Null，外键字段必须将属性 Null 设为 True，否则提示异常。
- SET_DEFAULT 模式：执行数据删除并把其他数据表的外键字段设为默认值。
- SET 模式：执行数据删除并把其他数据表的外键字段关联其他数据。
- DO_NOTHING 模式，不做任何处理，删除结果由数据库的删除模式决定。

4.4.4　查询单表数据

在更新数据时，往往只修改某行数据的内容，因此在更新数据之前还要对模型进行查询操作，确定数据表某行的数据对象，最后才执行数据更新操作。我们知道数据库设有多种数据查询方式，如单表查询、多表查询、子查询和联合查询等，而 Django 的 ORM 框架对不同的查询方式定义了相应的 API 方法。

以数据表 index_personinfo 和 index_vocation（即 4.4.3 节定义的模型 PersonInfo 和 Vocation）

为例，并为数据表 index_personinfo 和 index_vocation 添加数据，数据内容如图 4-14 所示。



index_vocation @main (MySQLite) - 表

文件 编辑 查看 窗口 帮助

开始事务 文本 筛选 排序 导入 导出

id	job	title	payment	name_id
1	软件工程师	Python开发	10000	2
2	文员	前台文员	5000	1
3	网站设计	前端开发	8000	4
4	需求分析师	系统需求设计	9000	3
5	项目经理	项目负责人	12000	5

index_personinfo @main (MySQLite) - 表

文件 编辑 查看 窗口 帮助

开始事务 文本 筛选 排序

id	name	age	hireDate
1	Lucy	20	2018-09-18
2	Tim	18	2018-09-18
3	Tom	22	2018-08-18
4	Mary	24	2018-07-10
5	Tony	25	2018-01-18

图 4-14　数据表 index_personinfo 和 index_vocation

然后在项目 babys 的 Shell 模式下使用 ORM 框架提供的 API 方法实现数据查询，代码如下：

```
>>> from index.models import *
# 全表查询
# SQL: Select * from index_vocation，数据以列表返回
>>> v = Vocation.objects.all()
# 查询第一条数据，序列从 0 开始
>>>v[0].job

# 查询前 3 条数据
# SQL: Select * from index_vocation LIMIT 3
# SQL 语句的 LIMIT 方法，在 Django 中使用列表截取即可
>>> v = Vocation.objects.all()[:3]
>>> v
<QuerySet [<Vocation: 1>, <Vocation: 2>, <Vocation: 3>]>

# 查询某个字段
# SQL: Select job from index_vocation
# values 方法，数据以列表返回，列表元素以字典表示
>>> v = Vocation.objects.values('job')
>>> v[1]['job']

# values_list 方法，数据以列表返回，列表元素以元组表示
>>> v = Vocation.objects.values_list('job')[:3]
>>> v
<QuerySet [('软件工程师',), ('文员',), ('网站设计',)]>

# 使用 get 方法查询数据
# SQL: Select*from index_vocation where id=2
>>> v = Vocation.objects.get(id=2)
>>>v.job
```

```
# 使用 filter 方法查询数据，注意区分 get 和 filter 的差异
>>> v = Vocation.objects.filter(id=2)
>>>v[0].job

# SQL 的 and 查询主要在 filter 里面添加多个查询条件
>>> v = Vocation.objects.filter(job='网站设计', id=3)
>>> v
<QuerySet [<Vocation: 3>]>
#filter 的查询条件可设为字典格式
>>> d=dict(job='网站设计', id=3)
>>> v = Vocation.objects.filter(**d)

# SQL 的 or 查询，需要引入 Q，编写格式：Q(field=value)|Q(field=value)
#多个 Q 之间使用“|”隔开即可
# SQL: Select * from index_vocation where job='网站设计' or id=9
>>> from django.db.models import Q
>>> v = Vocation.objects.filter(Q(job='网站设计')|Q(id=4))
>>> v
<QuerySet [<Vocation: 3>, <Vocation: 4>]>

# SQL 的不等于查询，在 Q 查询前面使用“~”即可
# SQL 语句：SELECT * FROM index_vocation WHERE NOT (job='网站设计')
>>> v = Vocation.objects.filter(~Q(job='网站设计'))
>>> v
<QuerySet [<Vocation: 1>,<Vocation: 2>,<Vocation: 4>,<Vocation: 5>]>
#还可以使用 exclude 实现不等于查询
>>> v = Vocation.objects.exclude(job='网站设计')
>>> v
<QuerySet [<Vocation: 1>,<Vocation: 2>,<Vocation: 4>,<Vocation: 5>]>

# 使用 count 方法统计查询数据的数据量
>>> v = Vocation.objects.filter(job='网站设计').count()

# 去重查询，distinct 方法无须设置参数，去重方式根据 values 设置的字段执行
# SQL: Select DISTINCT job from index_vocation where job = '网站设计'
>>> v = Vocation.objects.values('job').filter(job='网站设计').distinct()
>>> v
<QuerySet [{'job': '网站设计'}]>

# 根据字段 id 降序排列，降序只要在 order_by 里面的字段前面加"-"即可
# order_by 可设置多字段排列，如 Vocation.objects.order_by('-id', 'job')
>>> v = Vocation.objects.order_by('-id')
>>> v

# 聚合查询，实现对数据值求和、求平均值等。由 annotate 和 aggregate 方法实现
```

```
# annotate 类似于 SQL 里面的 GROUP BY 方法
#如果不设置 values，默认对主键进行 GROUP BY 分组
# SQL: Select job,SUM(id) AS 'id__sum' from index_vocation GROUP BY job
>>> from django.db.models import Sum, Count
>>> v = Vocation.objects.values('job').annotate(Sum('id'))
>>> print(v.query)

# aggregate 是计算某个字段的值并只返回计算结果
# SQL: Select COUNT(id) AS 'id_count' from index_vocation
>>> from django.db.models import Count
>>> v = Vocation.objects.aggregate(id_count=Count('id'))
>>> v
{'id_count': 5}

# union、intersection 和 difference 语法
# 每次查询结果的字段必须相同
# 第一次查询结果 v1
>>> v1 = Vocation.objects.filter(payment__gt=9000)
>>> v1
<QuerySet [<Vocation: 1>, <Vocation: 5>]>
# 第二次查询结果 v2
>>> v2 = Vocation.objects.filter(payment__gt=5000)
>>> v2
<QuerySet [<Vocation: 1>,<Vocation: 3>,<Vocation: 4>,<Vocation: 5>]>
# 使用 SQL 的 UNION 来组合两个或多个查询结果的并集
# 获取两次查询结果的并集
>>> v1.union(v2)
<QuerySet [<Vocation: 1>, <Vocation: 3>, <Vocation: 5>]>
# 使用 SQL 的 INTERSECT 来获取两个或多个查询结果的交集
# 获取两次查询结果的交集
>>> v1.intersection(v2)
<QuerySet [<Vocation: 1>, <Vocation: 5>]>
# 使用 SQL 的 EXCEPT 来获取两个或多个查询结果的差
# 以 v2 为目标数据，去除 v1 和 v2 的共同数据
>>> v2.difference(v1)
<QuerySet [<Vocation: 3>, <Vocation: 4>]>
```

上述例子讲述了开发中常用的数据查询方法，但有时需要设置不同的查询条件来满足多方面的查询要求。上述的查询条件 filter 和 get 是使用等值的方法来匹配结果。若想使用大于、不等于或模糊查询的匹配方法，则可在查询条件 filter 和 get 里使用表 4-1 所示的匹配符实现。

表 4-1　匹配符的使用及说明

匹 配 符	使　用	说　明
__exact	filter(job__exact='开发')	精确等于，如 SQL 的 like '开发'
__iexact	filter(job__iexact='开发')	精确等于并忽略大小写
__contains	filter(job__contains='开发')	模糊匹配，如 SQL 的 like '%荣耀%'
__icontains	filter(job__icontains='开发')	模糊匹配，忽略大小写
__gt	filter(id__gt=5)	大于
__gte	filter(id__gte=5)	大于等于
__lt	filter(id__lt=5)	小于
__lte	filter(id__lte=5)	小于等于
__in	filter(id__in=[1,2,3])	判断是否在列表内
__startswith	filter(job__startswith='开发')	以.....开头
__istartswith	filter(job__istartswith='开发')	以.....开头并忽略大小写
__endswith	filter(job__endswith='开发')	以.....结尾
__iendswith	filter(job__iendswith='开发')	以.....结尾并忽略大小写
__range	filter(job__range='开发')	在.....范围内
__year	filter(date__year=2018)	日期字段的年份
__month	filter(date__month=12)	日期字段的月份
__day	filter(date__day=30)	日期字段的天数
__isnull	filter(job__isnull=True/False)	判断是否为空

从表 4-1 中可以看到，只要在查询的字段末端设置相应的匹配符，就能实现不同的数据查询方式。例如在数据表 index_vocation 中查询字段 payment 大于 8000 的数据，在 Shell 模式下使用匹配符__gt 执行数据查询，代码如下：

```
>>> from index.models import *
>>> v = Vocation.objects.filter(payment__gt=8000)
>>> v
<QuerySet [<Vocation: 1>,<Vocation: 4>,<Vocation: 5>]>
```

综上所述，在查询数据时可以使用查询条件 get 或 filter 实现，但是两者的执行过程存在一定的差异，说明如下：

- 查询条件 get：查询字段必须是主键或者唯一约束的字段，并且查询的数据必须存在，如果查询的字段有重复值或者查询的数据不存在，程序就会抛出异常信息。
- 查询条件 filter：查询字段没有限制，只要该字段是数据表的某一字段即可。查询结果以列表形式返回，如果查询结果为空（查询的数据在数据表中找不到），就返回空列表。

4.4.5 查询多表数据

在日常的开发中，常常需要对多张数据表同时进行数据查询。多表查询需要在数据表之间建立表关系才能够实现。一对多或一对一的表关系是通过外键实现关联的，而多表查询分为正向查询和反向查询。

以模型 PersonInfo 和 Vocation 为例，模型 Vocation 定义的外键字段 name 关联到模型 PersonInfo。如果查询对象的主体是模型 Vocation，通过外键字段 name 去查询模型 PersonInfo 的关联数据，那么该查询称为正向查询；如果查询对象的主体是模型 PersonInfo，要查询它与模型 Vocation 的关联数据，那么该查询称为反向查询。无论是正向查询还是反向查询，两者的实现方法大致相同，代码如下：

```
# 正向查询
# 查询模型 Vocation 某行数据对象 v
>>> v = Vocation.objects.filter(id=1).first()
# v.name 代表外键 name
# 通过外键 name 去查询模型 PersonInfo 所对应的数据
>>> v.name.hireDate

# 反向查询
# 查询模型 PersonInfo 某行数据对象 p
>>> p = PersonInfo.objects.filter(id=2).first()
# 方法一
# vocation_set 的返回值为 queryset 对象，即查询结果
# vocation_set 的 vocation 为模型 Vocation 的名称小写
# 模型 Vocation 的外键字段 name 不能设置参数 related_name
# 若设置参数 related_name，则无法使用 vocation_set
>>> v = p.vocation_set.first()
>>> v.job
# 方法二
# 由模型 Vocation 的外键字段 name 的参数 related_name 实现
# 外键字段 name 必须设置参数 related_name 才有效，否则无法查询
# 将外键字段 name 的参数 related_name 设为 ps
>>> v = p.ps.first()
>>> v.job
```

正向查询和反向查询还能在查询条件（filter 或 get）里使用，这种方式用于查询条件的字段不在查询对象里，比如查询对象为模型 Vocation，查询条件是模型 PersonInfo 的某个字段，对于这种查询可以采用以下方法实现：

```
# 正向查询
# name__name，前面的 name 是模型 Vocation 的字段 name
# 后面的 name 是模型 PersonInfo 的字段 name，两者使用双下划线连接
>>> v = Vocation.objects.filter(name__name='Tim').first()
# v.name 代表外键 name
```

```
    >>> v.name.hireDate

    # 反向查询
    # 通过外键 name 的参数 related_name 实现反向条件查询
    # ps 代表外键 name 的参数 related_name
    # job 代表模型 Vocation 的字段 job
    p = PersonInfo.objects.filter(ps__job='网站设计').first()
    # 通过参数 related_name 反向获取模型 Vocation 的数据
    >>> v = p.personinfo.first()
    >>> v.job
```

无论是正向查询还是反向查询，它们在数据库里需要执行两次 SQL 查询，第一次是查询某张数据表的数据，再通过外键关联获取另一张数据表的数据信息。为了减少查询次数，提高查询效率，我们可以使用 select_related 或 prefetch_related 方法实现，该方法只需执行一次 SQL 查询就能实现多表查询。

select_related 主要针对一对一和一对多关系进行优化，它是使用 SQL 的 JOIN 语句进行优化的，通过减少 SQL 查询的次数来进行优化和提高性能，其使用方法如下：

```
    # select_related 方法，参数为字符串格式
    # 以模型 PersonInfo 为查询对象
    # select_related 使用 LEFT OUTER JOIN 方式查询两个数据表
    # 查询模型 PersonInfo 的字段 name 和模型 Vocation 的字段 payment
    # select_related 参数为 ps，代表外键字段 name 的参数 related_name
    # 若要得到其他数据表的关联数据，则可用双下划线“__”连接字段名
    # 双下划线“__”连接字段名必须是外键字段名或外键字段参数 related_name
    >>>
p=PersonInfo.objects.select_related('ps').values('name','ps__payment')
    # 查看 SQL 查询语句
    >>> print(p.query)

    # 以模型 Vocation 为查询对象
    # select_related 使用 INNER JOIN 方式查询两个数据表
    # select_related 的参数为 name，代表外键字段 name
    >>> v=Vocation.objects.select_related('name').values('name','name__age')
    # 查看 SQL 查询语句
    >>> print(v.query)

    # 获取两个模型的数据，以模型 Vocation 的 payment 大于 8000 为查询条件
    >>> v=Vocation.objects.select_related('name').
    filter(payment__gt=8000)
    # 查看 SQL 查询语句
    >>> print(v.query)
    # 获取查询结果集的首个元素的字段 age 的数据
    # 通过外键字段 name 定位模型 PersonInfo 的字段 age
    >>> v[0].name.age
```

除此之外，select_related 还可以支持 3 个或 3 个以上的数据表同时查询，以下面的例子进行说明。

```
# index 的 models.py
from django.db import models
# 省份信息表
class Province(models.Model):
    name = models.CharField(max_length=10)
    def __str__(self):
        return str(self.name)

# 城市信息表
class City(models.Model):
    name = models.CharField(max_length=5)
    province = models.ForeignKey(Province, on_delete=models.CASCADE)
    def __str__(self):
        return str(self.name)

# 人物信息表
class Person(models.Model):
    name = models.CharField(max_length=10)
    living = models.ForeignKey(City, on_delete=models.CASCADE)
    def __str__(self):
        return str(self.name)
```

在上述模型中，模型 Person 通过外键 living 关联模型 City，模型 City 通过外键 province 关联模型 Province，从而使 3 个模型形成一种递进关系。我们对上述新定义的模型执行数据迁移并在数据表里插入数据，如图 4-15 所示。

id	name
1	广东省
2	浙江省
3	海南省

id	name	province_id
1	广州	1
2	苏州	2
3	杭州	2
4	海口	3
5	深圳	1

id	name	living_id
1	Lily	1
2	Tom	2
3	Lucy	3
4	Tim	4
5	Mary	5

图 4-15　数据表信息

例如，查询 Tom 现在所居住的省份，首先通过模型 Person 和模型 City 查出 Tom 所居住的城市，然后通过模型 City 和模型 Province 查询当前城市所属的省份。因此，select_related 的实现方法如下：

```
>>> p=Person.objects.select_related('living__province').get(name='Tom')
>>> p.living.province
<Province: 浙江省>
```

从上述例子可以发现，通过设置 select_related 的参数值可实现 3 个或 3 个以上的多表查

询。例子中的参数值为 living__province，参数值说明如下：

- living 是模型 Person 的外键字段，该字段指向模型 City。
- province 是模型 City 的外键字段，该字段指向模型 Province。

两个外键字段之间使用双下画线连接，在查询过程中，模型 Person 的外键字段 living 指向模型 City，再从模型 City 的外键字段 province 指向模型 Province，从而实现 3 个或 3 个以上的多表查询。

prefetch_related 和 select_related 的设计目的很相似，都是为了减少 SQL 查询的次数，但是实现的方式不一样。select_related 是由 SQL 的 JOIN 语句实现的，但是对于多对多关系，使用 select_related 会增加数据查询时间和内存占用；而 prefetch_related 是分别查询每张数据表，然后由 Python 语法来处理它们之间的关系，因此对于多对多关系的查询，prefetch_related 更有优势。

我们在项目应用 index 的 models.py 里定义模型 Performer 和 Program，分别代表人员信息和节目信息，然后对模型执行数据迁移，生成相应的数据表，模型定义如下：

```
# index 的 models.py
from django.db import models
class Performer(models.Model):
    id = models.IntegerField(primary_key=True)
    name = models.CharField(max_length=20)
    nationality = models.CharField(max_length=20)
    def __str__(self):
        return str(self.name)

class Program(models.Model):
    id = models.IntegerField(primary_key=True)
    name = models.CharField(max_length=20)
    performer = models.ManyToManyField(Performer)
    def __str__(self):
        return str(self.name)
```

数据迁移成功后，在数据表 index_performer 和 index_program 中分别添加人员信息和节目信息，然后在数据表 index_program_performer 中设置多对多关系，如图 4-16 所示。

id	name	nationality
1	Lily	USA
2	Lilei	CHINA
3	Tom	US
4	Hanmei	CHINA

id	name
1	喜洋洋
2	小猪佩奇
3	白雪公主
4	小王子

id	program_id	performer_id
1	1	1
2	1	2
3	1	3
4	1	4

图 4-16　数据表信息

例如，查询“喜洋洋”节目有多少个人员参与演出，首先从节目表 index_program 里找出“喜洋洋”的数据信息，然后通过外键字段 performer 获取参与演出的人员信息，实现过程

如下：

```
# 查询模型 Program 的某行数据
>>> p=Program.objects.prefetch_related('performer').
filter(name='喜洋洋').first()
# 根据外键字段 performer 获取当前数据的多对多或一对多关系
>>> p.performer.all()
```

从上述例子看到，prefetch_related 的使用与 select_related 有一定的相似之处。如果是查询一对多关系的数据信息，那么两者皆可实现，但 select_related 的查询效率更佳。除此之外，Django 的 ORM 框架还提供很多 API 方法，可以满足开发中各种复杂的需求，由于篇幅有限，就不再一一介绍了，有兴趣的读者可在官网上查阅。

4.5 执行原生 SQL 语句

Django 在查询数据时，大多数查询都能使用 ORM 提供的 API 方法，但对于一些复杂的查询可能难以使用 ORM 的 API 方法实现，因此 Django 引入了 SQL 语句的执行方法，有以下 3 种实现方法。

- extra：结果集修改器，一种提供额外查询参数的机制。
- raw：执行原始 SQL 并返回模型实例对象。
- execute：直接执行自定义 SQL。

extra 适合用于 ORM 难以实现的查询条件，将查询条件使用原生 SQL 语法实现，此方法需要依靠模型对象，在某程度上可防止 SQL 注入。在 PyCharm 里打开 extra 源码，如图 4-17 所示，它一共定义了 6 个参数，每个参数说明如下：

- select：添加新的查询字段，即新增并定义模型之外的字段。
- where：设置查询条件。
- params：如果 where 设置了字符串格式化%s，那么该参数为 where 提供数值。
- tables：连接其他数据表，实现多表查询。
- order_by：设置数据的排序方式。
- select_params：如果 select 设置字符串格式化%s，那么该参数为 select 提供数值。

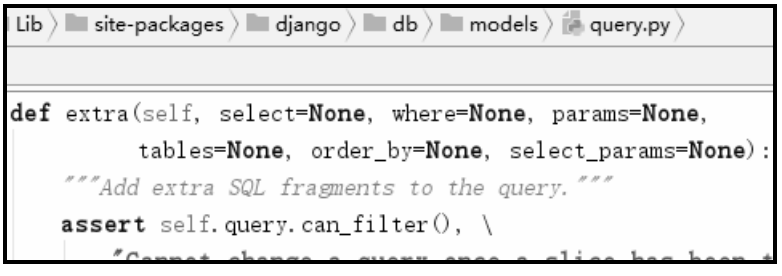


图 4-17　extra 源码

上述参数都是可选参数，我们可根据实际情况选择所需的参数。以模型 Vocation 为例，使用 extra 实现数据查询，代码如下：

```
# 查询字段 job 等于‘网站设计’的数据
# params 为 where 的%s 提供数值
>>> Vocation.objects.extra(where=["job=%s"],params=['网站设计'])
<QuerySet [<Vocation: 3>]>

# 新增查询字段 seat，select_params 为 select 的%s 提供数值
>>> v=Vocation.objects.extra(select={"seat":"%s"},
select_params=['seatInfo'])
>>> print(v.query)

# 连接数据表 index_personinfo
>>> v=Vocation.objects.extra(tables=['index_personinfo'])
>>> print(v.query)
```

下一步分析 raw 的语法，它和 extra 所实现的功能是相同的，只能实现数据查询操作，并且也要依靠模型对象，但从使用角度来说，raw 更为直观易懂。在 PyCharm 里打开 raw 源码，如图 4-18 所示，它一共定义了 4 个参数，每个参数说明如下：

- raw_query：SQL 语句。
- params：如果 raw_query 设置字符串格式化%s，那么该参数为 raw_query 提供数值。
- translations：为查询的字段设置别名。
- using：数据库对象，即 Django 所连接的数据库。

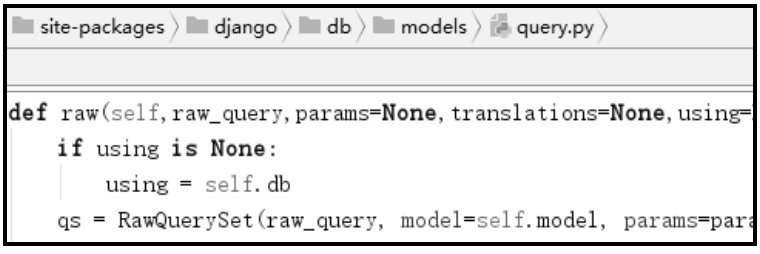


图 4-18　raw 源码

上述参数只有 raw_query 是必选参数，其他参数可根据需求自行选择。我们以模型 Vocation 为例，使用 raw 实现数据查询，代码如下：

```
>>> v = Vocation.objects.raw('select * from index_vocation')
>>> v[0]
<Vocation: 1>
```

最后分析 execute 的语法，它执行 SQL 语句无须经过 Django 的 ORM 框架。我们知道 Django 连接数据库需要借助第三方模块实现连接过程，如 MySQL 的 mysqlclient 模块和 SQLite 的 sqlite3 模块等，这些模块连接数据库之后，可通过游标的方式来执行 SQL 语句，而 execute 就是使用这种方式执行 SQL 语句，使用方法如下：

```
>>> from django.db import connection
>>> cursor=connection.cursor()
# 执行 SQL 语句
>>> cursor.execute('select * from index_vocation')
# 读取第一行数据
>>> cursor.fetchone()
# 读取所有数据
>>> cursor.fetchall()
```

execute 能够执行所有的 SQL 语句，但很容易受到 SQL 注入攻击，一般情况下不建议使用这种方式实现数据操作。尽管如此，它能补全 ORM 框架所缺失的功能，如执行数据库的存储过程。

4.6 本章小结

Django 对各种数据库提供了很好的支持，包括 PostgreSQL、MySQL、SQLite 和 Oracle，而且为这些数据库提供了统一的 API 方法，这些 API 统称为 ORM 框架。通过使用 Django 内置的 ORM 框架可以实现数据库连接和读写操作。

ORM 框架是一种程序技术，用于实现面向对象编程语言中不同类型系统的数据之间的转换。从效果上说，它创建了一个可在编程语言中使用的“虚拟对象数据库”，通过对虚拟对象数据库的操作从而实现对目标数据库的操作，虚拟对象数据库与目标数据库是相互对应的。

数据迁移是根据模型在数据库里创建相应的数据表，这一过程由 Django 内置的操作指令 makemigrations 和 migrate 实现，此外还讲述了数据迁移常见的错误以及其他数据迁移指令。

数据导入与导出是对数据表的数据执行导入与导出操作，确保开发阶段、测试阶段和项目上线的数据互不影响。

新增数据由模型实例化对象调用内置方法实现数据新增，比如单数据新增调用 create，查询与新增调用 get_or_create，修改与新增调用 update_or_create，批量新增调用 bulk_create。

更新数据必须执行一次数据查询，再对查询结果进行修改操作，常用方法有：模型实例化、update 方法和批量更新 bulk_update。

删除数据必须执行一次数据查询，再对查询结果进行删除操作，若删除的数据设有外键字段，则删除结果由外键的删除模式决定。

数据查询分为单表查询和多表查询，Django 提供多种不同查询的 API 方法，以满足开发需求。

执行 SQL 语句有 3 种方法实现：extra、raw 和 execute，其中 extra 和 raw 只能实现数据查询，具有一定的局限性；而 execute 无须经过 ORM 框架处理，能够执行所有的 SQL 语句，但很容易受到 SQL 注入攻击。