

李辉 (Grey Li)

Flask入门教程

使用Python和Flask开发你的第一个Web程序



目录

简介	1.1
前言	1.2
第 1 章：准备工作	1.3
第 2 章：Hello, Flask!	1.4
第 3 章：模板	1.5
第 4 章：静态文件	1.6
第 5 章：数据库	1.7
第 6 章：模板优化	1.8
第 7 章：表单	1.9
第 8 章：用户认证	1.10
第 9 章：测试	1.11
第 10 章：组织你的代码	1.12
第 11 章：部署上线	1.13
小挑战	1.14
后记	1.15

Flask 入门教程

这是一本 Flask 入门教程，提供了入门 Flask 所需的最少信息，你可以跟随本书自己动手开发一个简单的 [Watchlist 程序](http://helloflask.com/tutorial)。本书主页为 <http://helloflask.com/tutorial>。

关于作者

我叫李辉，我是《Flask Web 开发实战》的作者，Pallets Team 成员。你可以在我[的个人主页](#)了解更多关于我的信息。

目录

- 前言
- 第 1 章：准备工作
- 第 2 章：Hello, Flask!
- 第 3 章：模板
- 第 4 章：静态文件
- 第 5 章：数据库
- 第 6 章：模板优化
- 第 7 章：表单
- 第 8 章：用户认证
- 第 9 章：测试
- 第 10 章：组织你的代码
- 第 11 章：部署上线
- 小挑战
- 后记

版权信息

书名：Flask 入门教程

副书名：使用 Python 和 Flask 开发你的第一个 Web 程序

作者：[李辉](#)

简介

版本：2.0

发布时间：2019.12.6

© 2018 李辉（Grey Li） / HelloFlask.com

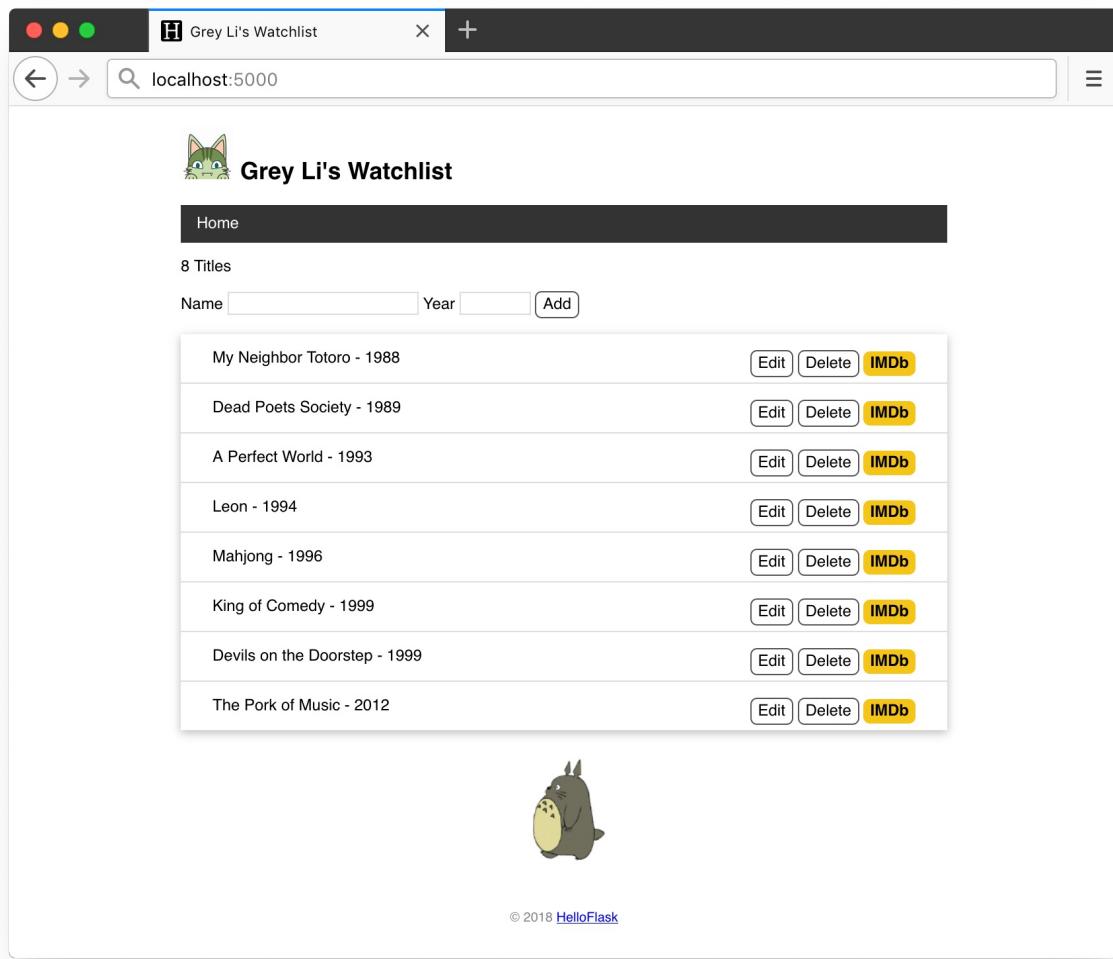
本书采用 [CC BY-NC-ND 3.0](#) 协议授权，禁止商用、演绎后分发或无署名转载。

前言

Flask 是一个使用 Python 语言编写的 Web 框架，它可以让你高效的编写 Web 程序。Web 程序即“网站”或“网页程序”，是指可以通过浏览器进行交互的程序。我们日常使用浏览器访问的豆瓣、知乎、百度等网站都是 Web 程序。

通过这本书，你会学到 Flask 开发的基础知识，并开发出一个简单的 Watchlist（观影清单）程序。

当你想要完成一个比较大的目标时，通常你会把这个目标分解成多个小目标，然后逐一去完成。开发程序也是这样，在一开始就编写出像豆瓣、IMDB 这样的程序恐怕不太现实，但是我们可以先模仿其中的一小部分。我们要完成的 Watchlist 程序就是一个很好的开始。在功能上，这个程序可以看做是简化版的 IMDB Watchlist / 豆瓣豆单：你可以添加、删除和修改你收藏的电影信息。



你可以访问 <http://watchlist.helloflask.com/> 查看示例程序的在线 Demo。

本书特点

- 基于 Flask 最新的 1.0.2 版本
- 使用一个 Watchlist 程序作为示例
- 复原完整的开发流程
- 只提供入门所需的最少信息
- 优化术语解释，更容易理解

阅读方法

本书复原了编写这个 Watchlist 程序的完整流程，包括每一行代码块，每一个需要执行的命令。在阅读时，你需要自己输入每一个代码和命令，检查输出是否和书中一致。在这个过程中，你也可以对它进行一些调整。比如，示例程序的界面语言使用了英文，你可以修改为中文或是其他语言。对于页面布局和样式，你也可以自由修改。

在本书的最后，你会把你自己的 Watchlist 部署到互联网上，让任何人都可以访问。

讨论与求助

如果你想和其他同学交流 Flask、Web 开发等相关话题，或是在学习中遇到了问题，想要寻求帮助，下面是一些好去处：

- [HelloFlask 论坛](#)
- [HelloFlask 微信群](#)
- [HelloFlask QQ 群 \(419980814\)](#)
- [HelloFlask Telegram 群组 \(@helloflask\)](#)

反馈与勘误

欢迎通过下面的方式提出反馈、建议和勘误：

- 在源码仓库[创建 Issue](#)。

- 在 [HelloFlask 论坛](#)发布帖子，并选择“Flask 入门教程”分类。
- 在专栏对应的连载文章下面撰写评论。

相关资源

- 本书主页：<http://helloflask.com/tutorial>
- 本书论坛：<https://discuss.helloflask.com>
- 本书源码：<https://github.com/greyli/flask-tutorial>
- 本书示例程序源码：<https://github.com/greyli/watchlist>
- 示例程序在线 Demo：<http://watchlist.helloflask.com>

付费支持

本书采取自愿付费原则，价格为 10 元。如果你愿意，可以通过付费来支持我，让我有更多的时间和动力写作 Flask 教程和文章。你可以通过支付宝账号 withlihui@gmail.com 转账，或是扫描下面的二维码付款。

支付宝二维码 / 微信二维码



第 1 章：准备工作

在通过这本书学习 Flask 开发前，我假设你了解了 Python 和 HTML 的基础知识。你的 Python 版本可以是 2.7，也可以是 3.3 及以上版本。电脑的操作系统可以是 Windows，也可以是 macOS 或 Linux。

安装编辑器和浏览器

对于编辑器来说，每个人都有不同的偏好，你可以自由选择。可以选择功能丰富的 IDE（集成开发环境），比如 PyCharm；也可以选择相对轻量的编辑器，比如 Atom 或 Sublime Text。浏览器建议使用 Firefox 或 Chrome。

使用命令行

在本书中，你需要使用命令行窗口来执行许多操作。你可以使用 Windows 下的 cmd.exe，或是 macOS 和 Linux 下的终端（Terminal）。下面我们执行一个最简单的 whoami 命令（即 Who Am I？）：

```
$ whoami  
greyli
```

这个命令会打印出当前计算机用户的名称。其他常用的命令还有 cd 命令，用来切换目录（change directory）； mkdir 命令，用来创建目录（make directory）。在不同的操作系统上，执行某个操作的命令可能会有所不同，在必要的地方，书里会进行提示。

我们先来为我们的程序创建一个文件夹：

```
$ mkdir watchlist  
$ cd watchlist
```

除非特别说明，从现在开始，本书假设你的工作目录将是在项目的根目录，即 watchlist/ 目录。

为了确保你已经正确安装了 Python，可以执行下面的命令测试是否有报错：

```
$ python --version  
Python 2.7.11
```

对于 Windows 用户，请使用 [cmdre](#)（一个基于 ConEmu 实现的终端模拟器）来代替系统自带的 cmd.exe，或是使用安装 Git for Windows 后（下一节）附带的 Git Bash。cmdre 集成了 Git Bash，支持一些在 Linux 或 macOS 下才能使用的命令（程序），比如 ls、cat、nano、ssh 等，这些命令我们在后面会用到。

使用 Git

Git 是一个流行的版本控制工具，我们可以用它来记录程序源码和文件的变动情况，或是在编程时进行多人协作，你可以把它看做一个优雅的代码变动备份工具。

如果你还不熟悉 Git 也没关系，可以先按照书中的命令去做，有时间再去了解原理。现在要做的第一件事就是在你的电脑上[安装 Git](#)（可以执行 git --help 命令检查是否已经安装，没有提示“命令未找到（Command not found）”则表示已安装）。

安装后可以在命令行先使用使用下面的命令查看版本，没有报错则表示已正确安装：

```
$ git --version  
git version 2.17.1
```

为了让 Git 知道你是谁，以便在提交代码到版本仓库的时候进行记录，使用下面的命令设置你的信息：

```
$ git config --global user.name "Grey Li" # 替换成你的名字  
$ git config --global user.email "withlihui@gmail.com" # 替换成  
你的邮箱地址
```

现在为我们的项目文件夹创建一个 Git 仓库，这会在我们的项目根目录创建一个 .git 文件夹：

```
$ git init
Initialized empty Git repository in ~/watchlist/.git/
```

Git 默认会追踪项目文件夹（或者说代码仓库）里所有文件的变化，但是有些无关紧要的文件不需要记录变化，我们在项目根目录创建一个 `.gitignore` 文件，在文件中写入忽略文件的规则。因为文件内容比较简单，我们直接在命令使用 `nano` 来创建：

```
$ nano .gitignore
```

在 `nano` 编辑界面写入常见的可忽略文件规则：

```
*.pyc
*~
__pycache__
.DS_Store
```

使用 `Control + O` 和 `Enter` 键保存，然后按下 `Control + X` 键退出。在后续章节，对于简单的文件，都会使用 `nano` 创建，这部分操作你也可以使用编辑器来完成。

将程序托管到 GitHub（可选）

这一步是可选的，将程序托管到 GitHub、GitLab 或是 BitBucket 等平台上，可以更方便的备份、协作和部署。这些托管平台作为 Git 服务器，你可以为本地仓库创建远程仓库。

首先要注册一个 GitHub 账户，点击访问[注册页面](#)，根据指示完成注册流程。登录备用。

设置 SSH 密钥

一般情况下，当推送本地改动到远程仓库时，需要输入用户名和密码。因为传输通常是通过 SSH 加密，所以可以通过设置 SSH 密钥来省去验证账号的步骤。

首先使用下面的命令检查是否已经创建了 SSH 密钥：

```
$ cat ~/.ssh/id_rsa.pub
```

如果显示“No such file or directory”，就使用下面的命令生成 SSH 密钥对，否则复制输出的值备用：

```
$ ssh-keygen
```

一路按下 Enter 采用默认值，最后会在用户根目录创建一个 .ssh 文件夹，其中包含两个文件，id_rsa 和 id_rsa.pub，前者是私钥，不能泄露出去，后者是公钥，用于认证身份，就是我们要保存到 GitHub 上的密钥值。再次使用前面提到的命令获得文件内容：

```
$ cat ~/.ssh/id_rsa.pub  
ssh-rsa AAAAB3Nza...省略 N 个字符...3aph book@greyli
```

选中并复制输出的内容，访问 GitHub 的 [SSH 设置页面](#)（导航栏头像 - Settings - SSH and GPG keys），点击 New SSH key 按钮，将复制的内容粘贴到 Key 输入框里，再填一个标题，比如“My PC”，最后点击“Add SSH key”按钮保存。

创建远程仓库

访问[新建仓库页面](#)（导航栏“+” - New repository），在“Repository name”处填写仓库名称，这里填“watchlist”即可，接着选择仓库类型（公开或私有）等选项，最后点击“Create repository”按钮创建仓库。

因为我们已经提前创建了本地仓库，所以需要指定仓库的远程仓库地址（如果没有创建，则可以直接将远程仓库克隆到本地）：

```
$ git remote add origin git@github.com:greyli/watchlist.git #  
注意更换地址中的用户名
```

这会为本地仓库关联一个名为“origin”的远程仓库，注意将仓库地址中的“**greyli**”换成你的 **GitHub** 用户名。

创建虚拟环境

虚拟环境是独立于 Python 全局环境的 Python 解释器环境，使用它的好处如下：

- 保持全局环境的干净
- 指定不同的依赖版本
- 方便记录和管理依赖

我们将使用 Python 3 内置的 `venv` 模块创建虚拟环境。

如果你使用 Python 2，则需要安装 `virtualenv` 作为替代：

```
$ pip install virtualenv
```

或：

```
$ sudo pip install virtualenv
```

然后使用下面的命令即可为当前项目创建一个虚拟环境：

```
$ python -m venv env
```

Python 2 用户使用：

```
$ virtualenv env
```

上述命令的最后一个参数是虚拟环境名称，你可以自由定义，比如 `venv`、`env`、`.venv`，或是 `项目名-venv`，这里使用了 `env`。

这会在当前目录创建一个包含 Python 解释器环境的虚拟环境文件夹，名称为 `env`。

创建虚拟环境后，我们可以使用下面的命令来激活虚拟环境，如下所示（执行 `deactivate` 可以退出虚拟环境）：

```
$ env\Scripts\activate # Windows
```

或

```
$ . env/bin/activate # Linux 或 macOS
```

这时命令提示符前会显示虚拟环境的名称，表示已经激活成功：

```
(env) $
```

注意除了 Git 相关命令外，除非特别说明，本书后续的所有命令均需要在激活虚拟环境后执行。

提示 建议为 pip 更新 PyPI 源，改为使用国内的 PyPI 镜像源以提高下载速度，具体见[这篇文章](#)。

安装 Flask

激活虚拟环境后，使用下面的命令来安装 Flask：

```
(env) $ pip install flask
```

这会把 Flask 以及相关的一些依赖包安装到对应的虚拟环境。

提示 如果你没有使用虚拟环境，记得将 Flask 更新到最新版本（`pip install -U flask`）。

本章小结

当你进行到这里，就意味这我们已经做好学习和开发 Flask 程序的全部准备了。使用 `git status` 命令可以查看当前仓库的文件变动状态：

```
$ git status
```

下面让我们将文件改动提交进 Git 仓库，并推送到在 GitHub 上创建的远程仓库：

```
$ git add .
$ git commit -m "I'm ready!"
$ git push -u origin master # 如果你没有把仓库托管到 GitHub，则跳过这条命令，后面章节亦同
```

这里最后一行命令添加了 `-u` 参数，会将推送的目标仓库和分支设为默认值，后续的推送直接使用 `git push` 命令即可。在 GitHub 上，你可以通过 <https://github.com/你的用户名/watchlist> 查看你的仓库内容。

提示 你可以在 GitHub 上查看本书示例程序的对应 commit：[1b6fe4a](#)。

进阶提示

- 阅读 MDN 的《Web 入门教程》（了解 HTML、CSS、JavaScript）。
- 阅读短教程《Git 简明指南》。
- 如果你打算开源你的程序，在项目根目录中添加一个 README.md（自述文件）和 LICENSE（授权声明）是很有必要的。详情可以访问 [Open Source Guides](#) 了解。

第 2 章：Hello, Flask!

追溯到最初，Flask 诞生于 Armin Ronacher 在 2010 年愚人节开的一个玩笑。后来，它逐渐发展成为一个成熟的 Python Web 框架，越来越受到开发者的喜爱。目前它在 GitHub 上是 Star 数量最多的 Python Web 框架，没有之一。

Flask 是典型的微框架，作为 Web 框架来说，它仅保留了核心功能：请求响应处理和模板渲染。这两类功能分别由 Werkzeug (WSGI 工具库) 完成和 Jinja (模板渲染库) 完成，因为 Flask 包装了这两个依赖，我们暂时不用深入了解它们。

主页

这一章的主要任务就是为我们的程序编写一个简单的主页。主页的 URL 一般就是根地址，即 /。当用户访问根地址的时候，我们需要返回一行欢迎文字。这个任务只需要下面几行代码就可以完成：

`app.py`：程序主页

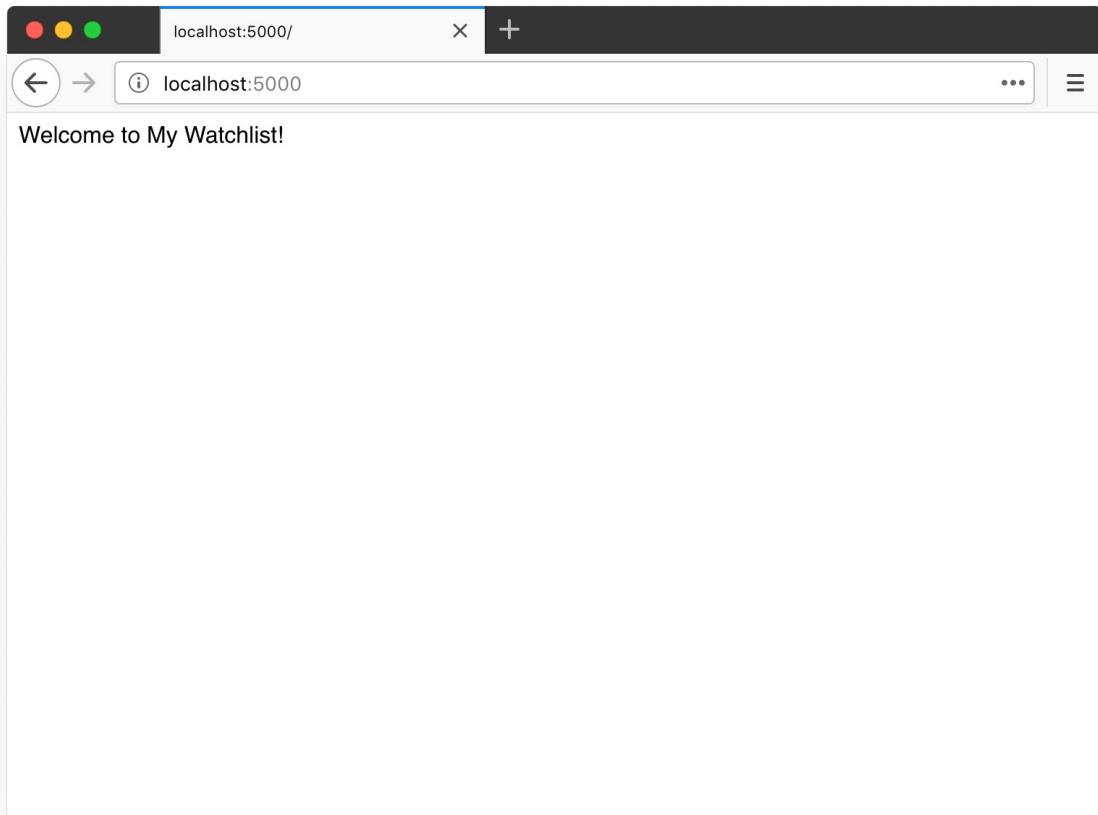
```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello():
    return 'Welcome to My Watchlist!'
```

按照惯例，我们把程序保存为 `app.py`，确保当前目录是项目的根目录，并且激活了虚拟环境，然后在命令行窗口执行 `flask run` 命令启动程序（按下 Control + C 可以退出）：

```
(env) $ flask run
* Serving Flask app "app.py"
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

现在打开浏览器，访问 <http://localhost:5000> 即可访问我们的程序主页，并看到我们在程序里返回的问候语，如下图所示：



执行 `flask run` 命令时，Flask 会使用内置的开发服务器来运行程序。这个服务器默认监听本地机的 5000 端口，也就是说，我们可以通过在地址栏输入 <http://127.0.0.1:5000> 或是 <http://localhost:5000> 访问程序。

注意 内置的开发服务器只能用于开发时使用，部署上线的时候要换用性能更好的服务器，我们会在最后一章学习。

解剖时间

下面我们来分解这个 Flask 程序，了解它的基本构成。

首先我们从 `flask` 包导入 `Flask` 类，通过实例化这个类，创建一个程序对象 `app`：

```
from flask import Flask
app = Flask(__name__)
```

接下来，我们要注册一个处理函数，这个函数是处理某个请求的处理函数，Flask 官方把它叫做视图函数（view function），你可以理解为“请求处理函数”。

所谓的“注册”，就是给这个函数戴上一个装饰器帽子。我们使用 `app.route()` 装饰器来为这个函数绑定对应的 URL，当用户在浏览器访问这个 URL 的时候，就会触发这个函数，获取返回值，并把返回值显示到浏览器窗口：

```
@app.route('/')
def hello():
    return 'Welcome to My Watchlist!'
```

提示 为了便于理解，你可以把 Web 程序看作是一堆这样的视图函数的集合：编写不同的函数处理对应 URL 的请求。

填入 `app.route()` 装饰器的第一个参数是 URL 规则字符串，这里的 `/` 指的是根地址。

我们只需要写出相对地址，主机地址、端口号等都不需要写出。所以说，这里的 `/` 对应的是主机名后面的路径部分，完整 URL 就是 <http://localhost:5000/>。如果我们这里定义的 URL 规则是 `/hello`，那么完整 URL 就是 <http://localhost:5000/hello>。

整个请求的处理过程如下所示：

1. 当用户在浏览器地址栏访问这个地址，在这里即 <http://localhost:5000/>
2. 服务器解析请求，发现请求 URL 匹配的 URL 规则是 `/`，因此调用对应的处理函数 `hello()`
3. 获取 `hello()` 函数的返回值，处理后返回给客户端（浏览器）

4. 浏览器接受响应，将其显示在窗口上

提示 在 Web 程序的语境下，虽然客户端可能有多种类型，但在本书里通常是指浏览器。

程序发现机制

如果你把上面的程序保存成其他的名字，比如 `hello.py`，接着执行 `flask run` 命令会返回一个错误提示。这是因为 Flask 默认会假设你把程序存储在名为 `app.py` 或 `wsgi.py` 的文件中。如果你使用了其他名称，就要设置系统环境变量 `FLASK_APP` 来告诉 Flask 你要启动哪个程序。

Flask 通过读取这个环境变量值对应的模块寻找要运行的程序实例，你可以把它设置成下面这些值：

- 模块名
- Python 导入路径
- 文件目录路径

管理环境变量

现在在启动 Flask 程序的时候，我们通常要和两个环境变量打交道：`FLASK_APP` 和 `FLASK_ENV`。因为我们的程序现在的名字是 `app.py`，暂时不需要设置 `FLASK_APP`；`FLASK_ENV` 用来设置程序运行的环境，默认为 `production`。在开发时，我们需要开启调试模式（`debug mode`）。调试模式可以通过将系统环境变量 `FLASK_ENV` 设为 `development` 来开启。调试模式开启后，当程序出错，浏览器页面上会显示错误信息；代码出现变动后，程序会自动重载。

为了不用每次打开新的终端会话都要设置环境变量，我们安装用来管理系统环境变量的 `python-dotenv`：

```
(env) $ pip install python-dotenv
```

当 `python-dotenv` 安装后，Flask 会从项目根目录的 `.flaskenv` 和 `.env` 文件读取环境变量并设置。我们分别使用文本编辑器创建这两个文件，或是使用更方便的 `touch` 命令创建：

```
$ touch .env .flaskenv
```

.flaskenv 用来存储 Flask 命令行系统相关的公开环境变量；而 .env 则用来存储敏感数据，不应该提交进 Git 仓库，我们把文件名 .env 添加到 .gitignore 文件的结尾（新建一行）来让 Git 忽略它。你可以使用编辑器打开这个文件，然后添加下面这一行内容：

```
.env
```

在新创建的 .flaskenv 文件里，我们写入一行 FLASK_ENV=development ，将环境变量 FLASK_ENV 的值设为 development ，以便开启调试模式：

```
FLASK_ENV=development
```

实验时间

在这个小节，我们可以通过做一些实验，来扩展和加深对本节内容的理解。

修改视图函数返回值

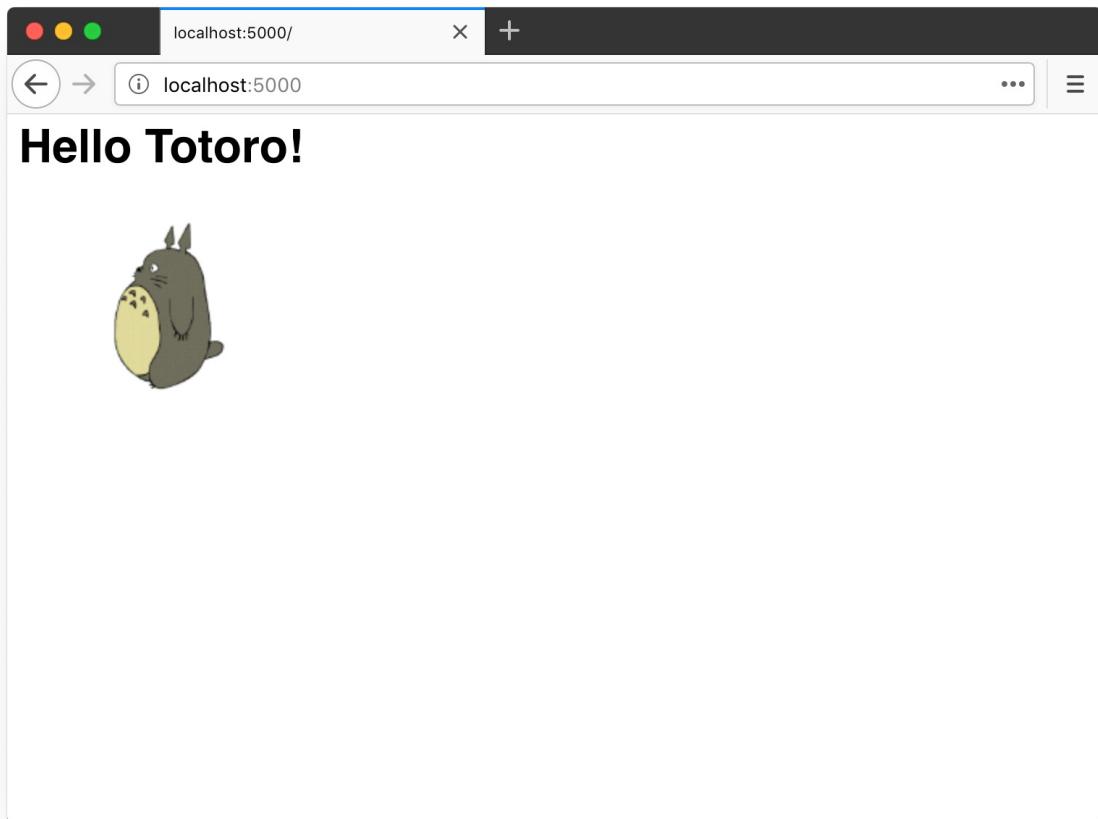
首先，你可以自由修改视图函数的返回值，比如：

```
@app.route('/')
def hello():
    return u'欢迎来到我的 Watchlist !'
```

返回值作为响应的主体，默认会被浏览器作为 HTML 格式解析，所以我们可以添加一个 HTML 元素标记：

```
@app.route('/')
def hello():
    return '<h1>Hello Totoro!</h1>'
```

保存修改后，只需要在浏览器里刷新页面，你就会看到页面上的内容也会随之变化。



修改 URL 规则

另外，你也可以自由修改传入 `app.route` 装饰器里的 URL 规则字符串，但要注意以斜线 `/` 作为开头。比如：

```
@app.route('/home')
def hello():
    return 'Welcome to My Watchlist!'
```

保存修改，这时刷新浏览器，则会看到一个 404 错误提示，提示页面未找到（Page Not Found）。这是因为视图函数的 URL 改成了 `/home`，而我们刷新后访问的地址仍然是旧的 `/`。如果我们把访问地址改成 <http://localhost:5000/home>，就会正确看到返回值。

一个视图函数也可以绑定多个 URL，这通过附加多个装饰器实现，比如：

```
@app.route('/')
@app.route('/index')
@app.route('/home')
def hello():
    return 'Welcome to My Watchlist!'
```

现在无论是访问 <http://localhost:5000/>、<http://localhost:5000/home> 还是 <http://localhost:5000/index> 都可以看到返回值。

在前面，我们之所以把传入 `app.route` 装饰器的参数称为 URL 规则，是因为我们也可以在 URL 里定义变量部分。比如下面这个视图函数会处理所有类似 `/user/<name>` 的请求：

```
@app.route('/user/<name>')
def user_page(name):
    return 'User page'
```

不论你访问 <http://localhost:5000/user/greyli>，还是 <http://localhost:5000/user/peter>，抑或是 <http://localhost:5000/user/甲>，都会触发这个函数。通过下面的方式，我们也可以在视图函数里获取到这个变量值：

```
@app.route('/user/<name>')
def user_page(name):
    return 'User: %s' % name
```

修改视图函数名？

最后一个可以修改的部分就是视图函数的名称了。首先，视图函数的名字是自由定义的，和 URL 规则无关。和定义其他函数或变量一样，只需要让它表达出所要处理页面的含义即可。

除此之外，它还有一个重要的作用：作为代表某个路由的端点（endpoint），同时用来生成 URL。对于程序内的 URL，为了避免手写，Flask 提供了一个 `url_for` 函数来生成 URL，它接受的第一个参数就是端点值，默认认为视图函数的名称：

```
from flask import url_for

# ...

@app.route('/')
def hello():
    return 'Hello'

@app.route('/user/<name>')
def user_page(name):
    return 'User: %s' % name

@app.route('/test')
def test_url_for():
    # 下面是一些调用示例（请在命令行窗口查看输出的 URL）：
    print(url_for('hello')) # 输出：/
    # 注意下面两个调用是如何生成包含 URL 变量的 URL 的
    print(url_for('user_page', name='greyli')) # 输出：/user/gre
    yli
    print(url_for('user_page', name='peter')) # 输出：/user/peter

    print(url_for('test_url_for')) # 输出：/test
    # 下面这个调用传入了多余的关键字参数，它们会被作为查询字符串附加到 URL
    后面。
    print(url_for('test_url_for', num=2)) # 输出：/test?num=2
    return 'Test page'
```

实验过程中编写的代码可以删掉，也可以保留，但记得为根地址返回一行问候，这可是我们这一章的任务。

本章小结

这一章我们为程序编写了主页，同时学习了 Flask 视图函数的基本编写方式。结束前，让我们提交代码：

```
$ git add .
$ git commit -m "Add minimal home page"
$ git push
```

为了保持简单，我们统一在章节最后一次提交所有改动。在现实世界里，通常会根据需要分为多个 `commit`；同样的，这里使用 `-m` 参数给出简单的提交信息。在现实世界里，你可能需要撰写更完整的提交信息。

提示 你可以在 GitHub 上查看本书示例程序的对应 `commit`：[eca06dc](#)。

进阶提示

- 如果你使用 Python 2.7，为了使程序正常工作，需要在脚本首行添加编码声明
`# -*- coding: utf-8-*-`，并在包含中文的字符串前面添加 `u` 前缀。本书中对于包含中文的字符串均添加了 `u` 前缀，这在 Python 3 中并不需要。
- 对于 URL 变量，Flask 还支持在 URL 规则字符串里对变量设置处理器，对变量进行预处理。比如 `/user/<int:number>` 会将 URL 中的 `number` 部分处理成整型，同时这个变量值接收传入数字。
- 因为 Flask 的上下文机制，有一些变量和函数（比如 `url_for` 函数）只能在特定的情况下才能正确执行，比如视图函数内。我们先暂时不用纠结，后面再慢慢了解。
- 名字以 `.` 开头的文件默认会被隐藏，执行 `ls` 命令时会看不到它们，这时你可以使用 `ls -f` 命令来列出所有文件。
- 了解 HTTP 基本知识将会有助于你了解 Flask 的工作原理。
- 阅读文章 [《互联网是如何工作的》](#)。
- 阅读文章 [《从HTTP请求 - 响应循环探索Flask的基本工作方式》](#)。
- 如果你是 [《Flask Web 开发实战》](#) 的读者，这部分的进阶内容可以在第 1 章 [《初识 Flask》](#) 和第 2 章 [《HTTP 和 Flask》](#) 找到。

第3章：模板

在一般的 Web 程序里，访问一个地址通常会返回一个包含各类信息的 HTML 页面。因为我们的程序是动态的，页面中的某些信息需要根据不同的情况进行调整，比如对登录和未登录用户显示不同的信息，所以页面需要在用户访问时根据程序逻辑动态生成。

我们把包含变量和运算逻辑的 HTML 或其他格式的文本叫做模板，执行这些变量替换和逻辑计算工作的过程被称为渲染，这个工作由我们这一章要学习使用的模板渲染引擎——Jinja2 来完成。

按照默认的设置，Flask 会从程序实例所在模块同级目录的 `templates` 文件夹中寻找模板，我们的程序目前存储在项目根目录的 `app.py` 文件里，所以我们要在项目根目录创建这个文件夹：

```
$ mkdir templates
```

模板基本语法

在社交网站上，每个人都只有一个主页，借助 Jinja2 就可以写出一个通用的模板：

```
<h1>{{ username }}的个人主页</h1>
{% if bio %}
    <p>{{ bio }}</p>  {# 这里的缩进只是为了可读性，不是必须的 #-}
{% else %}
    <p>自我介绍为空。</p>
{% endif %}  {# 大部分 Jinja 语句都需要声明关闭 #}
```

Jinja2 的语法和 Python 大致相同，你在后面会陆续接触到一些常见的用法。在模板里，你需要添加特定的定界符将 Jinja2 语句和变量标记出来，下面是三种常用的定界符：

- `{{ ... }}` 用来标记变量。
- `{% ... %}` 用来标记语句，比如 `if` 语句，`for` 语句等。
- `{# ... #}` 用来写注释。

模板中使用的变量需要在渲染的时候传递进去，具体我们后面会了解。

编写主页模板

我们先在 `templates` 目录下创建一个 `index.html` 文件，作为主页模板。主页需要显示电影条目列表和个人信息，代码如下所示：

`templates/index.html`：主页模板

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>{{ name }}'s Watchlist</title>
</head>
<body>
    <h2>{{ name }}'s Watchlist</h2>
    {# 使用 length 过滤器获取 movies 变量的长度 #}
    <p>{{ movies|length }} Titles</p>
    <ul>
        {% for movie in movies %}  {# 迭代 movies 变量 #}
            <li>{{ movie.title }} - {{ movie.year }}</li>  {# 等同于
movie['title'] #}
        {% endfor %}  {# 使用 endfor 标签结束 for 语句 #}
    </ul>
    <footer>
        <small>&copy; 2018 <a href="http://helloflask.com/tutorial">HelloFlask</a></small>
    </footer>
</body>
</html>
```

为了方便对变量进行处理，Jinja2 提供了一些过滤器，语法形式如下：

```
{{ 变量 | 过滤器 }}
```

左侧是变量，右侧是过滤器名。比如，上面的模板里使用 `length` 过滤器来获取 `movies` 的长度，类似 Python 里的 `len()` 函数。

提示 访问 <http://jinja.pocoo.org/docs/2.10/templates/#list-of-built-in-filters> 查看所有可用的过滤器。

准备虚拟数据

为了模拟页面渲染，我们需要先创建一些虚拟数据，用来填充页面内容：

app.py：定义虚拟数据

```
name = 'Grey Li'  
movies = [  
    {'title': 'My Neighbor Totoro', 'year': '1988'},  
    {'title': 'Dead Poets Society', 'year': '1989'},  
    {'title': 'A Perfect World', 'year': '1993'},  
    {'title': 'Leon', 'year': '1994'},  
    {'title': 'Mahjong', 'year': '1996'},  
    {'title': 'Swallowtail Butterfly', 'year': '1996'},  
    {'title': 'King of Comedy', 'year': '1999'},  
    {'title': 'Devils on the Doorstep', 'year': '1999'},  
    {'title': 'WALL-E', 'year': '2008'},  
    {'title': 'The Pork of Music', 'year': '2012'},  
]
```

渲染主页模板

使用 `render_template()` 函数可以把模板渲染出来，必须传入的参数为模板文件名（相对于 `templates` 根目录的文件路径），这里即 `'index.html'`。为了让模板正确渲染，我们还要把模板内部使用的变量通过关键字参数传入这个函数，如下所示：

app.py：返回渲染好的模板作为响应

```
from flask import Flask, render_template

# ...

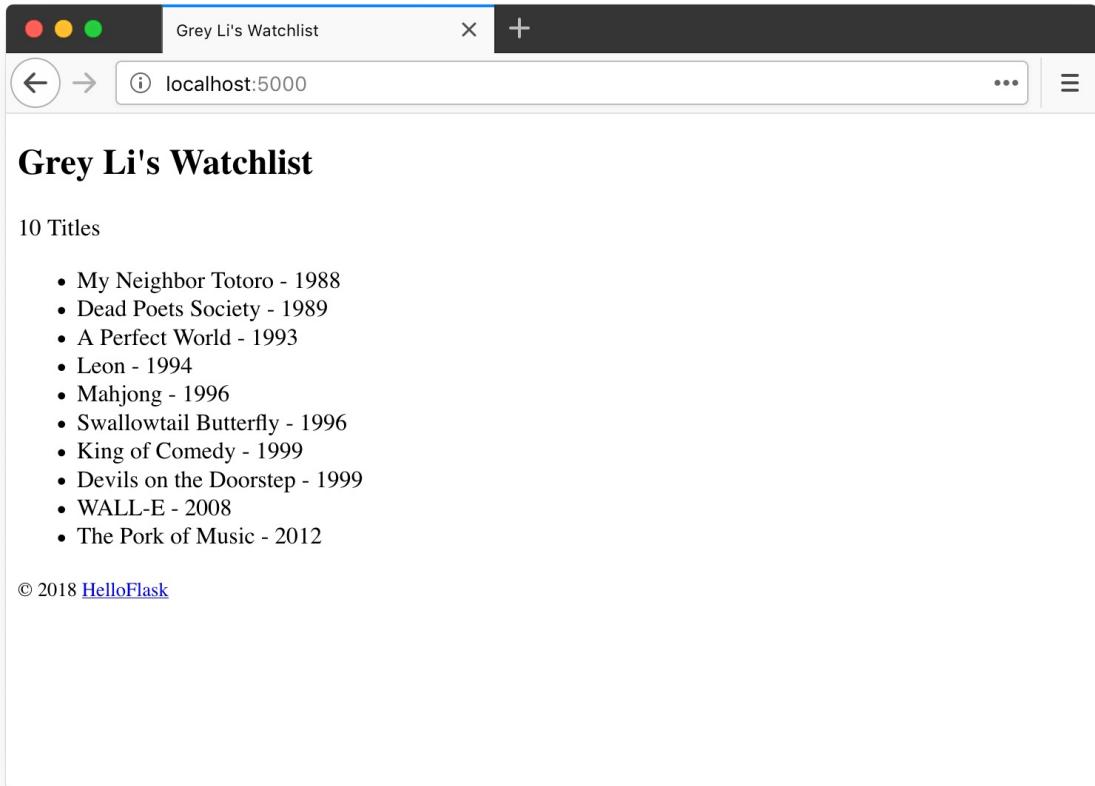
@app.route('/')
def index():
    return render_template('index.html', name=name, movies=movie
s)
```

为了更好的表示这个视图函数的作用，我们把原来的函数名 `hello` 改为 `index`，意思是“索引”，即主页。

在传入 `render_template()` 函数的关键字参数中，左边的 `movies` 是模板中使用的变量名称，右边的 `movies` 则是该变量指向的实际对象。这里传入模板的 `name` 是字符串，`movies` 是列表，但能够在模板里使用的不只这两种 Python 数据结构，你也可以传入元组、字典、函数等。

`render_template()` 函数在调用时会识别并执行 `index.html` 里所有的 Jinja2 语句，返回渲染好的模板内容。在返回的页面中，变量会被替换为实际的值（包括定界符），语句（及定界符）则会在执行后被移除（注释也会一并移除）。

现在访问 <http://localhost:5000/> 看到的程序主页如下图所示：



本章小结

这一章我们编写了一个简单的主页。结束前，让我们提交代码：

```
$ git add .
$ git commit -m "Add index page"
$ git push
```

提示 你可以在 GitHub 上查看本书示例程序的对应 commit：[8537d98](#)。

进阶提示

- 使用 [Faker](#) 可以实现自动生成虚拟数据，它支持丰富的数据类型，比如时间、人名、地名、随机字符等等……
- 除了过滤器，[Jinja2](#) 还在模板中提供了一些测试器、全局函数可以使用；除此之外，还有更丰富的控制结构支持，有一些我们会在后面学习到，更多的内容

则可以访问 [Jinja2 文档](#) 学习。

- 如果你是《[Flask Web 开发实战](#)》的读者，模板相关内容可以在第 3 章《模板》找到，[Faker](#) 相关内容可以在第 7 章找到。

第 4 章：静态文件

静态文件（static files）和我们的模板概念相反，指的是内容不需要动态生成的文件。比如图片、CSS 文件和 JavaScript 脚本等。

在 Flask 中，我们需要创建一个 static 文件夹来保存静态文件，它应该和程序模块、templates 文件夹在同一目录层级，所以我们在项目根目录创建它：

```
$ mkdir static
```

生成静态文件 URL

在 HTML 文件里，引入这些静态文件需要给出资源所在的 URL。为了更加灵活，这些文件的 URL 可以通过 Flask 提供的 `url_for()` 函数来生成。

在第 2 章的最后，我们学习过 `url_for()` 函数的用法，传入端点值（视图函数的名称）和参数，它会返回对应的 URL。对于静态文件，需要传入的端点值是 `static`，同时使用 `filename` 参数来传入相对于 static 文件夹的文件路径。

假如我们在 static 文件夹的根目录下面放了一个 `foo.jpg` 文件，下面的调用可以获取它的 URL：

```

```

花括号部分的调用会返回 `/static/foo.jpg`。

提示 在 Python 脚本里，`url_for()` 函数需要从 `flask` 包中导入，而在模板中则可以直接使用，因为 Flask 把一些常用的函数和对象添加到了模板上下文（环境）里。

添加 Favicon

Favicon (favourite icon) 是显示在标签页和书签栏的网站头像。你需要准备一个 ICO、PNG 或 GIF 格式的图片，大小一般为 16×16 、 32×32 、 48×48 或 64×64 像素。把这个图片放到 static 目录下，然后像下面这样在 HTML 模板里引入它：

templates/index.html：引入 Favicon

```
<head>
  ...
  <link rel="icon" href="{{ url_for('static', filename='favicon.ico') }}">
</head>
```

保存后刷新页面，即可在浏览器标签页上看到这个图片。

添加图片

为了让页面不那么单调，我们来添加两个图片：一个是显示在页面标题旁边的头像，另一个是显示在页面底部的龙猫动图。我们在 `static` 目录下面创建一个子文件夹 `images`，把这两个图片都放到这个文件夹里：

```
$ cd static
$ mkdir images
```

创建子文件夹并不是必须的，这里只是为了更好的组织同类文件。同样的，如果你有多个 `CSS` 文件，也可以创建一个 `css` 文件夹来组织他们。下面我们在页面模板中添加这两个图片，注意填写正确的文件路径：

templates/index.html：添加图片

```
<h2>
  
```

提示 这两张图片你可以自己替换为任意的图片（注意更新文件名），也可以在示例程序的 [GitHub 仓库](#) 下载。

添加 CSS

虽然添加了图片，但页面还是非常简陋，因为我们还没有添加 CSS 定义。下面在 static 目录下创建一个 CSS 文件 style.css，内容如下：

static/style.css：定义页面样式

```
/* 页面整体 */
body {
    margin: auto;
    max-width: 580px;
    font-size: 14px;
    font-family: Helvetica, Arial, sans-serif;
}

/* 脚注 */
footer {
    color: #888;
    margin-top: 15px;
    text-align: center;
    padding: 10px;
}

/* 头像 */
.avatar {
    width: 40px;
}

/* 电影列表 */
.movie-list {
    list-style-type: none;
    padding: 0;
    margin-bottom: 10px;
    box-shadow: 0 2px 5px 0 rgba(0, 0, 0, 0.16), 0 2px 10px 0 rgba(0, 0, 0, 0.12);
}

.movie-list li {
    padding: 12px 24px;
    border-bottom: 1px solid #ddd;
```

```
}

.movie-list li:last-child {
    border-bottom:none;
}

.movie-list li:hover {
    background-color: #f8f9fa;
}

/* 龙猫图片 */
.totoro {
    display: block;
    margin: 0 auto;
    height: 100px;
}
```

接着在页面的 `<head>` 标签内引入这个 CSS 文件：

`templates/index.html`：引入 CSS 文件

```
<head>
    ...
    <link rel="stylesheet" href="{{ url_for('static', filename='
style.css') }}" type="text/css">
</head>
```

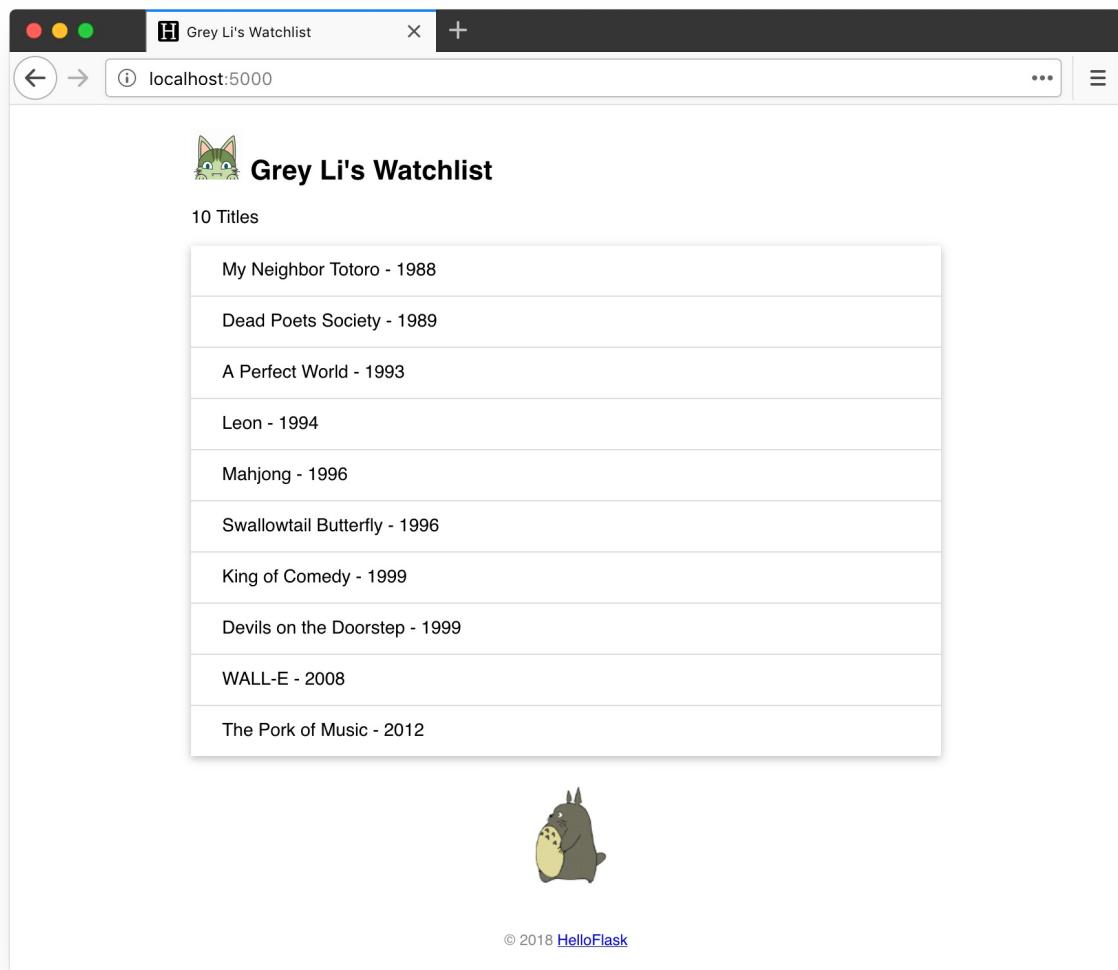
最后要为对应的元素设置 `class` 属性值，以便和对应的 CSS 定义关联起来：

`templates/index.html`：添加 `class` 属性

```
<h2>
    
    {{ name }}'s Watchlist
</h2>
...
<ul class="movie-list">
    ...
</ul>

```

最终的页面如下图所示（你可以自由修改 CSS 定义，我已经尽力了）：



本章小结

主页现在基本成型了，接下来我们会慢慢完成程序的功能。结束前，让我们提交代码：

```
$ git add .
$ git commit -m "Add static files"
$ git push
```

| 提示 你可以在 GitHub 上查看本书示例程序的对应 commit：[e51c579](#)。

进阶提示

- 如果你对 CSS 很头疼，可以借助前端框架来完善页面样式，比如 [Bootstrap](#)、[Semantic-UI](#)、[Foundation](#) 等。它们提供了大量的 CSS 定义和动态效果，使用起来非常简单。
- 扩展 [Bootstrap-Flask](#) 可以简化在 Flask 项目里使用 Bootstrap 4 的步骤。

第 5 章：数据库

大部分程序都需要保存数据，所以不可避免要使用数据库。用来操作数据库的数据库管理系统（DBMS）有很多选择，对于不同类型的程序，不同的使用场景，都会有不同的选择。在这个教程中，我们选择了属于关系型数据库管理系统（RDBMS）的 [SQLite](#)，它基于文件，不需要单独启动数据库服务器，适合在开发时使用，或是在数据库操作简单、访问量低的程序中使用。

使用 SQLAlchemy 操作数据库

为了简化数据库操作，我们将使用 [SQLAlchemy](#)——一个 Python 数据库工具（ORM，即对象关系映射）。借助 SQLAlchemy，你可以通过定义 Python 类来表示数据库里的一张表（类属性表示表中的字段 / 列），通过对这个类进行各种操作来代替写 SQL 语句。**这个类我们称之为模型类，类中的属性我们将称之为字段。**

Flask 有大量的第三方扩展，这些扩展可以简化和第三方库的集成工作。我们下面将使用一个叫做 [Flask-SQLAlchemy](#) 的官方扩展来集成 SQLAlchemy。

首先安装它：

```
(env) $ pip install flask-sqlalchemy
```

大部分扩展都需要执行一个“初始化”操作。你需要导入扩展类，实例化并传入 Flask 程序实例：

```
from flask_sqlalchemy import SQLAlchemy # 导入扩展类

app = Flask(__name__)

db = SQLAlchemy(app) # 初始化扩展，传入程序实例 app
```

设置数据库 URI

为了设置 Flask、扩展或是我们程序本身的一些行为，我们需要设置和定义一些配置变量。Flask 提供了一个统一的接口来写入和获取这些配置变量：`Flask.config` 字典。配置变量的名称必须使用大写，写入配置的语句一般会放到扩展类实例化语句之前。

下面写入了一个 `SQLALCHEMY_DATABASE_URI` 变量来告诉 SQLAlchemy 数据库连接地址：

```
import os

# ...

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////' + os.path.join(app.root_path, 'data.db')
```

注意 这个配置变量的最后一个单词是 `URI`，而不是 `URL`。

对于这个变量值，不同的 DBMS 有不同的格式，对于 SQLite 来说，这个值的格式如下：

`sqlite:///` 数据库文件的绝对地址

数据库文件一般放到项目根目录即可，`app.root_path` 返回程序实例所在模块的路径（目前来说，即项目根目录），我们使用它来构建文件路径。数据库文件的名称和后缀你可以自由定义，一般会使用 `.db`、`.sqlite` 和 `.sqlite3` 作为后缀。

另外，如果你使用 Windows 系统，上面的 `URI` 前缀部分需要写入三个斜线（即 `sqlite:///`）。在本书的示例程序代码里，做了一些兼容性处理，另外还新设置了一个配置变量，实际的代码如下：

`app.py`：数据库配置

```
import os
import sys

from flask import Flask

WIN = sys.platform.startswith('win')
if WIN: # 如果是 Windows 系统，使用三个斜线
    prefix = 'sqlite:///'
else: # 否则使用四个斜线
    prefix = 'sqlite:////'

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = prefix + os.path.join(app.root_path, 'data.db')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False # 关闭对模型修改的监控
# 在扩展类实例化前加载配置
db = SQLAlchemy(app)
```

如果你固定在某一个操作系统上进行开发，部署时也使用相同的操作系统，那么可以不用这么做，直接根据你的需要写出前缀即可。

提示 你可以访问 [Flask 文档的配置页面](#)查看 Flask 内置的配置变量；同样的，在 [Flask-SQLAlchemy 文档的配置页面](#)可以看到 Flask-SQLAlchemy 提供的配置变量。

创建数据库模型

在 Watchlist 程序里，目前我们有两类数据要保存：用户信息和电影条目信息。下面分别创建了两个模型类来表示这两张表：

`app.py`：创建数据库模型

```

class User(db.Model): # 表名将会是 user (自动生成，小写处理)
    id = db.Column(db.Integer, primary_key=True) # 主键
    name = db.Column(db.String(20)) # 名字

class Movie(db.Model): # 表名将会是 movie
    id = db.Column(db.Integer, primary_key=True) # 主键
    title = db.Column(db.String(60)) # 电影标题
    year = db.Column(db.String(4)) # 电影年份

```

模型类的编写有一些限制：

- 模型类要声明继承 `db.Model`。
- 每一个类属性（字段）要实例化 `db.Column`，传入的参数为字段的类型，下面的表格列出了常用的字段类。
- 在 `db.Column()` 中添加额外的选项（参数）可以对字段进行设置。比如，`primary_key` 设置当前字段是否为主键。除此之外，常用的选项还有 `nullable`（布尔值，是否允许为空值）、`index`（布尔值，是否设置索引）、`unique`（布尔值，是否允许重复值）、`default`（设置默认值）等。

常用的字段类型如下表所示：

字段类	说明
<code>db.Integer</code>	整型
<code>db.String(size)</code>	字符串，size 为最大长度，比如 <code>db.String(20)</code>
<code>db.Text</code>	长文本
<code>db.DateTime</code>	时间日期，Python <code>datetime</code> 对象
<code>db.Float</code>	浮点数
<code>db.Boolean</code>	布尔值

创建数据库表

模型类创建后，还不能对数据库进行操作，因为我们还没有创建表和数据库文件。下面在 Python Shell 中创建了它们：

```
(env) $ flask shell  
>>> from app import db  
>>> db.create_all()
```

打开文件管理器，你会发现项目根目录下出现了新创建的数据库文件 `data.db`。这个文件不需要提交到 Git 仓库，我们在 `.gitignore` 文件最后添加一行新规则：

```
*.db
```

如果你改动了模型类，想重新生成表模式，那么需要先使用 `db.drop_all()` 删除表，然后重新创建：

```
>>> db.drop_all()  
>>> db.create_all()
```

注意这会一并删除所有数据，如果你想在不破坏数据库内的数据的前提下变更表的结构，需要使用数据库迁移工具，比如集成了 Alembic 的 [Flask-Migrate](#) 扩展。

提示 上面打开 Python Shell 使用的是 `flask shell` 命令，而不是 `python`。使用这个命令启动的 Python Shell 激活了“程序上下文”，它包含一些特殊变量，这对于某些操作是必须的（比如上面的 `db.create_all()` 调用）。请记住，后续的 Python Shell 都会使用这个命令打开。

和 `flask shell` 类似，我们可以编写一个自定义命令来自动执行创建数据库表操作：

`app.py`：自定义命令 `initdb`

```
import click

@app.cli.command() # 注册为命令
@click.option('--drop', is_flag=True, help='Create after drop.')
# 设置选项
def initdb(drop):
    """Initialize the database."""
    if drop: # 判断是否输入了选项
        db.drop_all()
    db.create_all()
    click.echo('Initialized database.') # 输出提示信息
```

默认情况下，函数名称就是命令的名字，现在执行 `flask initdb` 命令就可以创建数据库表：

```
(env) $ flask initdb
```

使用 `--drop` 选项可以删除表后重新创建：

```
(env) $ flask initdb --drop
```

创建、读取、更新、删除

在前面打开的 Python Shell 里，我们来测试一下常见的数据库操作。你可以跟着示例代码来操作，也可以自由练习。

创建

下面的操作演示了如何向数据库中添加记录：

```
>>> from app import User, Movie # 导入模型类
>>> user = User(name='Grey Li') # 创建一个 User 记录
>>> m1 = Movie(title='Leon', year='1994') # 创建一个 Movie 记录
>>> m2 = Movie(title='Mahjong', year='1996') # 再创建一个 Movie
记录
>>> db.session.add(user) # 把新创建的记录添加到数据库会话
>>> db.session.add(m1)
>>> db.session.add(m2)
>>> db.session.commit() # 提交数据库会话，只需要在最后调用一次即可
```

提示 在实例化模型类的时候，我们并没有传入 `id` 字段（主键），因为 SQLAlchemy 会自动处理这个字段。

最后一行 `db.session.commit()` 很重要，只有调用了这一行才会真正把记录提交进数据库，前面的 `db.session.add()` 调用是将改动添加进数据库会话（一个临时区域）中。

读取

通过对模型类的 `query` 属性调用可选的过滤方法和查询方法，我们就可以获取到对应的单个或多个记录（记录以模型类实例的形式表示）。查询语句的格式如下：

```
<模型类>.query.<过滤方法（可选）>.<查询方法>
```

下面是一些常用的过滤方法：

过滤方法	说明
<code>filter()</code>	使用指定的规则过滤记录，返回新产生的查询对象
<code>filter_by()</code>	使用指定规则过滤记录（以关键字表达式的形式），返回新产生的查询对象
<code>order_by()</code>	根据指定条件对记录进行排序，返回新产生的查询对象
<code>group_by()</code>	根据指定条件对记录进行分组，返回新产生的查询对象

下面是一些常用的查询方法：

查询方法	说明
all()	返回包含所有查询记录的列表
first()	返回查询的第一条记录，如果未找到，则返回None
get(id)	传入主键值作为参数，返回指定主键值的记录，如果未找到，则返回None
count()	返回查询结果的数量
first_or_404()	返回查询的第一条记录，如果未找到，则返回404错误响应
get_or_404(id)	传入主键值作为参数，返回指定主键值的记录，如果未找到，则返回404错误响应
paginate()	返回一个Pagination对象，可以对记录进行分页处理

下面的操作演示了如何从数据库中读取记录，并进行简单的查询：

```
>>> from app import Movie # 导入模型类
>>> movie = Movie.query.first() # 获取 Movie 模型的第一个记录（返回
模型类实例）
>>> movie.title # 对返回的模型类实例调用属性即可获取记录的各字段数据
'Leon'
>>> movie.year
'1994'
>>> Movie.query.all() # 获取 Movie 模型的所有记录，返回包含多个模型类
实例的列表
[<Movie 1>, <Movie 2>]
>>> Movie.query.count() # 获取 Movie 模型所有记录的数量
2
>>> Movie.query.get(1) # 获取主键值为 1 的记录
<Movie 1>
>>> Movie.query.filter_by(title='Mahjong').first() # 获取 title
字段值为 Mahjong 的记录
<Movie 2>
>>> Movie.query.filter(Movie.title=='Mahjong').first() # 等同于
上面的查询，但使用不同的过滤方法
<Movie 2>
```

提示 我们在说 Movie 模型的时候，实际指的是数据库中的 movie 表。表的实际名称是模型类的小写形式（自动生成），如果你想自己指定表名，可以定义 `_tablename_` 属性。

对于最基础的 `filter()` 过滤方法，SQLAlchemy 支持丰富的查询操作符，具体可以访问[文档相关页面](#)查看。除此之外，还有更多的查询方法、过滤方法和数据库函数可以使用，具体可以访问文档的 [Query API](#) 部分查看。

更新

下面的操作更新了 `Movie` 模型中主键为 `2` 的记录：

```
>>> movie = Movie.query.get(2)
>>> movie.title = 'WALL-E' # 直接对实例属性赋予新的值即可
>>> movie.year = '2008'
>>> db.session.commit() # 注意仍然需要调用这一行来提交改动
```

删除

下面的操作删除了 `Movie` 模型中主键为 `1` 的记录：

```
>>> movie = Movie.query.get(1)
>>> db.session.delete(movie) # 使用 db.session.delete() 方法删除
记录，传入模型实例
>>> db.session.commit() # 提交改动
```

在程序里操作数据库

经过上面的一番练习，我们可以在 `Watchlist` 里进行实际的数据库操作了。

在主页视图读取数据库记录

因为设置了数据库，负责显示主页的 `index` 可以从数据库里读取真实的数据：

```
@app.route('/')
def index():
    user = User.query.first() # 读取用户记录
    movies = Movie.query.all() # 读取所有电影记录
    return render_template('index.html', user=user, movies=movies)
```

在 `index` 视图中，原来传入模板的 `name` 变量被 `user` 实例取代，模板 `index.html` 中的两处 `name` 变量也要相应的更新为 `user.name` 属性：

```
{{ user.name }}'s Watchlist
```

生成虚拟数据

因为有了数据库，我们可以编写一个命令函数把虚拟数据添加到数据库里。下面是用来生成虚拟数据的命令函数：

`app.py`：创建自定义命令 `forge`

```

import click

@app.cli.command()
def forge():
    """Generate fake data."""
    db.create_all()

    # 全局的两个变量移动到这个函数内
    name = 'Grey Li'
    movies = [
        {'title': 'My Neighbor Totoro', 'year': '1988'},
        {'title': 'Dead Poets Society', 'year': '1989'},
        {'title': 'A Perfect World', 'year': '1993'},
        {'title': 'Leon', 'year': '1994'},
        {'title': 'Mahjong', 'year': '1996'},
        {'title': 'Swallowtail Butterfly', 'year': '1996'},
        {'title': 'King of Comedy', 'year': '1999'},
        {'title': 'Devils on the Doorstep', 'year': '1999'},
        {'title': 'WALL-E', 'year': '2008'},
        {'title': 'The Pork of Music', 'year': '2012'},
    ]

    user = User(name=name)
    db.session.add(user)
    for m in movies:
        movie = Movie(title=m['title'], year=m['year'])
        db.session.add(movie)

    db.session.commit()
    click.echo('Done.')

```

现在执行 `flask forge` 命令就会把所有虚拟数据添加到数据库里：

```
(env) $ flask forge
```

本章小结

本章我们学习了使用 SQLAlchemy 操作数据库，后面你会慢慢熟悉相关的操作。结束前，让我们提交代码：

```
$ git add .
$ git commit -m "Add database support with Flask-SQLAlchemy"
$ git push
```

提示 你可以在 GitHub 上查看本书示例程序的对应 commit：[4d2442a](#)。

进阶提示

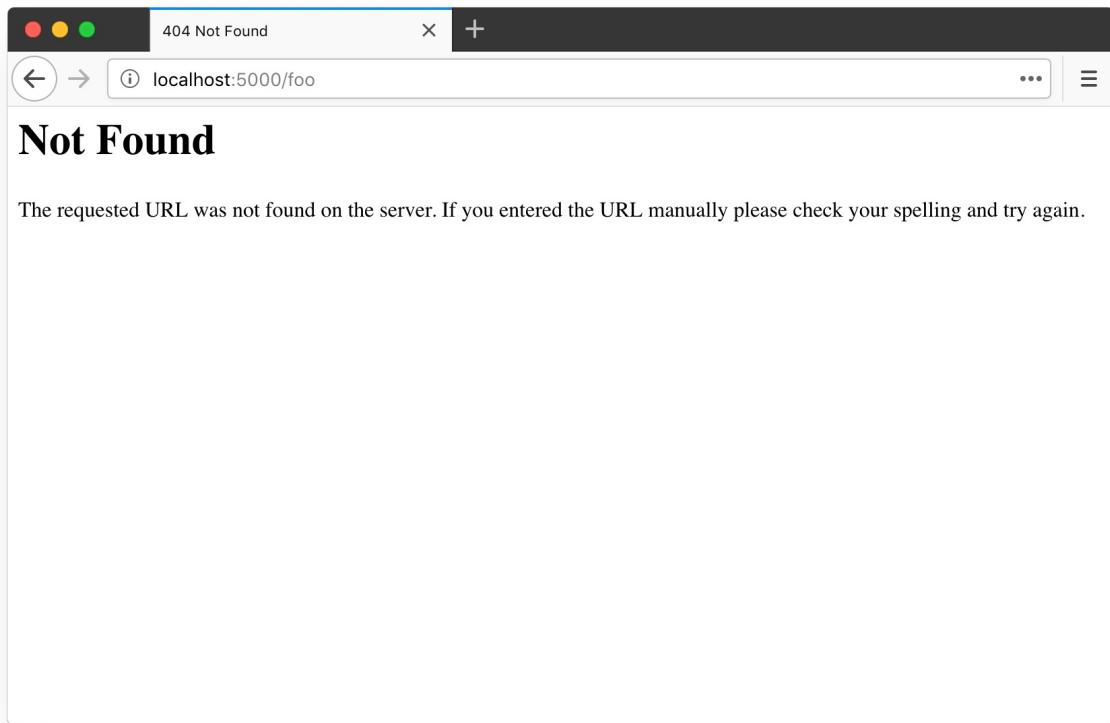
- 在生产环境，你可以更换更合适的 DBMS，因为 SQLAlchemy 支持多种 SQL 数据库引擎，通常只需要改动非常少的代码。
- 我们的程序只有一个用户，所以没有将 User 表和 Movie 表建立关联。访问 [Flask-SQLAlchemy 文档的“声明模型”章节](#)可以看到相关内容。
- 阅读 [SQLAlchemy 官方文档](#)和教程详细了解它的用法。注意我们在这里使用 [Flask-SQLAlchemy](#) 来集成它，所以用法和单独使用 SQLAlchemy 有一些不同。作为参考，你可以同时阅读 [Flask-SQLAlchemy 官方文档](#)。
- 如果你是《[Flask Web 开发实战](#)》的读者，第 5 章详细介绍了 SQLAlchemy 和 [Flask-Migrate](#) 的使用，第 8 章和第 9 章引入了更复杂的模型关系和查询方法。

第6章：模板优化

这一章我们会继续完善模板，学习几个非常实用的模板编写技巧，为下一章实现创建、编辑电影条目打下基础。

自定义错误页面

为了引出相关知识点，我们首先要为 Watchlist 编写一个错误页面。目前的程序中，如果你访问一个不存在的 URL，比如 /hello，Flask 会自动返回一个 404 错误响应。默认的错误页面非常简陋，如下图所示：



在 Flask 程序中自定义错误页面非常简单，我们先编写一个 404 错误页面模板，如下所示：

`templates/404.html`：404 错误页面模板

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>{{ user.name }}'s Watchlist</title>
    <link rel="icon" href="{{ url_for('static', filename='favicon.ico') }}">
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}" type="text/css">
</head>
<body>
    <h2>
        
        {{ user.name }}'s Watchlist
    </h2>
    <ul class="movie-list">
        <li>
            Page Not Found - 404
            <span class="float-right">
                <a href="{{ url_for('index') }}">Go Back</a>
            </span>
        </li>
    </ul>
    <footer>
        <small>&copy; 2018 <a href="http://helloflask.com/tutorial">HelloFlask</a></small>
    </footer>
</body>
</html>

```

接着使用 `app.errorhandler()` 装饰器注册一个错误处理函数，它的作用和视图函数类似，当 404 错误发生时，这个函数会被触发，返回值会作为响应主体返回给客户端：

`app.py` : 404 错误处理函数

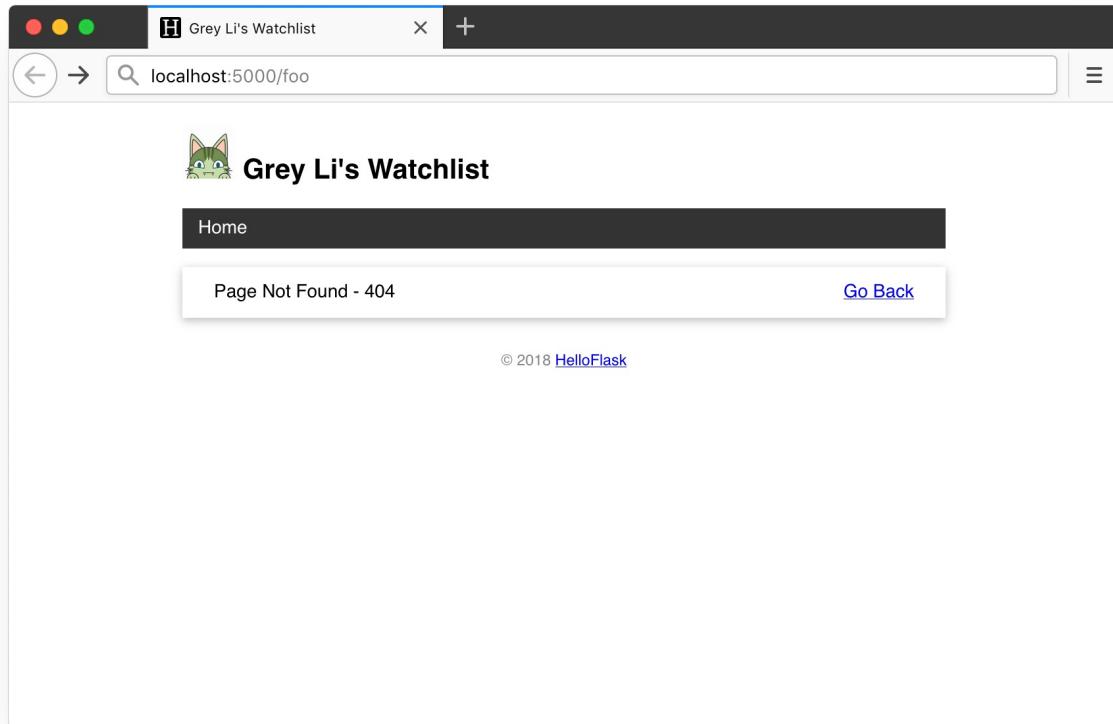
```

@app.errorhandler(404) # 传入要处理的错误代码
def page_not_found(e): # 接受异常对象作为参数
    user = User.query.first()
    return render_template('404.html', user=user), 404 # 返回模板和状态码

```

提示 和我们前面编写的视图函数相比，这个函数返回了状态码作为第二个参数，普通的视图函数之所以不用写出状态码，是因为默认会使用 200 状态码，表示成功。

这个视图返回渲染好的错误模板，因为模板中使用了 `user` 变量，这里也要一并传入。现在访问一个不存在的 URL，会显示我们自定义的错误页面：



编写完这部分代码后，你会发现两个问题：

- 错误页面和主页都需要使用 `user` 变量，所以在对应的处理函数里都要查询数据库并传入 `user` 变量。因为每一个页面都需要获取用户名显示在页面顶部，如果有更多的页面，那么每一个对应的视图函数都要重复传入这个变量。
- 错误页面模板和主页模板有大量重复的代码，比如 `<head>` 标签的内容，页首的标题，页脚信息等。这种重复不仅带来不必要的工作量，而且会让修改变得

得更加麻烦。举例来说，如果页脚信息需要更新，那么每个页面都要一一进行修改。

显而易见，这两个问题有更优雅的处理方法，下面我们就一一了解。

模板上下文处理函数

对于多个模板内都需要使用的变量，我们可以使用 `app.context_processor` 装饰器注册一个模板上下文处理函数，如下所示：

`app.py`：模板上下文处理函数

```
@app.context_processor
def inject_user(): # 函数名可以随意修改
    user = User.query.first()
    return dict(user=user) # 需要返回字典，等同于return {'user': user}
```

这个函数返回的变量（以字典键值对的形式）将会统一注入到每一个模板的上下文环境中，因此可以直接在模板中使用。

现在我们可以删除 404 错误处理函数和主页视图函数中的 `user` 变量定义，并删除在 `render_template()` 函数里传入的关键字参数：

```

@app.context_processor
def inject_user():
    user = User.query.first()
    return dict(user=user)

@app.errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404

@app.route('/')
def index():
    movies = Movie.query.all()
    return render_template('index.html', movies=movies)

```

同样的，后面我们创建的任意一个模板，都可以在模板中直接使用 `user` 变量。

使用模板继承组织模板

对于模板内容重复的问题，Jinja2 提供了模板继承的支持。这个机制和 Python 类继承非常类似：我们可以定义一个父模板，一般会称之为基模板（`base template`）。基模板中包含完整的 HTML 结构和导航栏、页首、页脚都通用部分。在子模板里，我们可以使用 `extends` 标签来声明继承自某个基模板。

基模板中需要在实际的子模板中追加或重写的部分则可以定义成块（`block`）。块使用 `block` 标签创建，`{% block 块名称 %}` 作为开始标记，`{% endblock %}` 或 `{% endblock 块名称 %}` 作为结束标记。通过在子模板里定义一个同样名称的块，你可以向基模板的对应块位置追加或重写内容。

编写基础模板

下面是新编写的基模板 `base.html`：

`templates/base.html`：基模板

```

<!DOCTYPE html>
<html lang="en">
<head>
    {% block head %}
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{{ user.name }}'s Watchlist</title>
    <link rel="icon" href="{{ url_for('static', filename='favicon.ico') }}">
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}" type="text/css">
    {% endblock %}
</head>
<body>
    <h2>
        
        {{ user.name }}'s Watchlist
    </h2>
    <nav>
        <ul>
            <li><a href="{{ url_for('index') }}">Home</a></li>
        </ul>
    </nav>
    {% block content %}{% endblock %}
    <footer>
        <small>&copy; 2018 <a href="http://helloflask.com/tutorial">HelloFlask</a></small>
    </footer>
</body>
</html>

```

在基模板里，我们添加了两个块，一个是包含 `<head></head>` 内容的 `head` 块，另一个是用来在子模板中插入页面主体内容的 `content` 块。在复杂的项目里，你可以定义更多的块，方便在子模板中对基模板的各个部分插入内容。另外，块的名字没有特定要求，你可以自由修改。

在编写子模板之前，我们先来看一下基模板中的两处新变化。

第一处，我们添加了一个新的 `<meta>` 元素，这个元素会设置页面的视口，让页面根据设备的宽度来自动缩放页面，让移动设备拥有更好的浏览体验：

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

第二处，新的页面添加了一个导航栏：

```
<nav>
    <ul>
        <li><a href="{{ url_for('index') }}>Home</a></li>
    </ul>
</nav>
```

导航栏对应的 CSS 代码如下所示：

```
nav ul {  
    list-style-type: none;  
    margin: 0;  
    padding: 0;  
    overflow: hidden;  
    background-color: #333;  
}  
  
nav li {  
    float: left;  
}  
  
nav li a {  
    display: block;  
    color: white;  
    text-align: center;  
    padding: 8px 12px;  
    text-decoration: none;  
}  
  
nav li a:hover {  
    background-color: #111;  
}
```

编写子模板

创建了基模板后，子模板的编写会变得非常简单。下面是新的主页模板（index.html）：

templates/index.html：继承基模板的主页模板

```
{% extends 'base.html' %}

{% block content %}
<p>{{ movies|length }} Titles</p>
<ul class="movie-list">
    {% for movie in movies %}
        <li>{{ movie.title }} - {{ movie.year }}
            <span class="float-right">
                <a class="imdb" href="https://www.imdb.com/find?q={{ movie.title }}" target="_blank" title="Find this movie on IMDb">
                    IMDb
                </a>
            </span>
        </li>
    {% endfor %}
</ul>

{% endblock %}
```

第一行使用 `extends` 标签声明扩展自模板 `base.html`，可以理解成“这个模板继承自 `base.html`”。接着我们定义了 `content` 块，这里的内容会插入到基模板中 `content` 块的位置。

提示 默认的块重写行为是覆盖，如果你想向父块里追加内容，可以在子块中使用 `super()` 声明，即 `{{ super() }}`。

404 错误页面的模板类似，如下所示：

`templates/404.html`：继承基模板的 404 错误页面模板

```
{% extends 'base.html' %}

{% block content %}
<ul class="movie-list">
    <li>
        Page Not Found - 404
        <span class="float-right">
            <a href="{{ url_for('index') }}">Go Back</a>
        </span>
    </li>
</ul>
{% endblock %}
```

添加 IMDb 链接

在主页模板里，我们还为每一个电影条目右侧添加了一个 IMDb 链接：

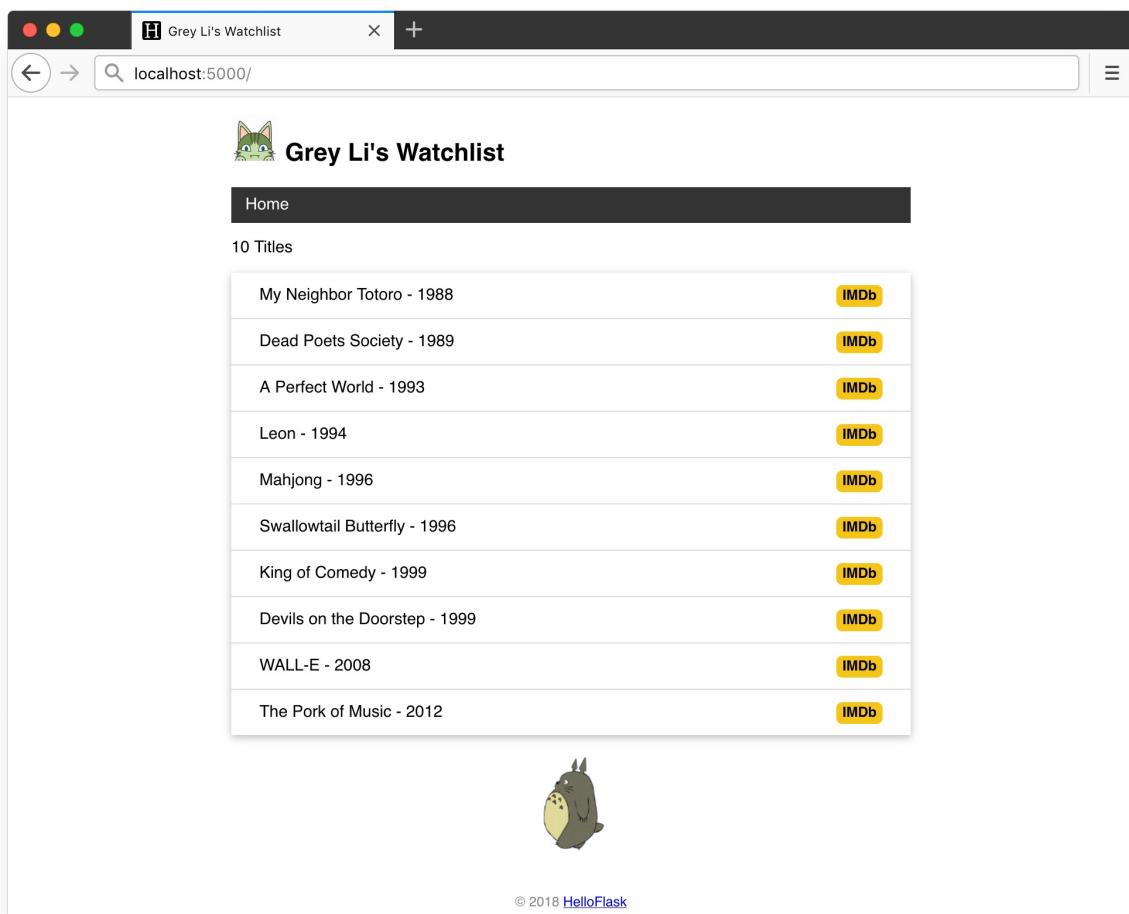
```
<span class="float-right">
    <a class="imdb" href="https://www.imdb.com/find?q={{ movie.title }}" target="_blank" title="Find this movie on IMDb">IMDb</a>
</span>
```

这个链接的 `href` 属性的值为 IMDb 搜索页面的 URL，搜索关键词通过查询参数 `q` 传入，这里传入了电影的标题。

对应的 CSS 定义如下所示：

```
.float-right {  
    float: right;  
}  
  
.imdb {  
    font-size: 12px;  
    font-weight: bold;  
    color: black;  
    text-decoration: none;  
    background: #F5C518;  
    border-radius: 5px;  
    padding: 3px 5px;  
}
```

现在，我们的程序主页如下所示：



本章小结

本章我们主要学习了 Jinja2 的模板继承机制，去掉了大量的重复代码，这让后续的模板编写工作变得更加轻松。结束前，让我们提交代码：

```
$ git add .
$ git commit -m "Add base template and error template"
$ git push
```

提示 你可以在 GitHub 上查看本书示例程序的对应 commit：3bca489。

进阶提示

- 本章介绍的自定义错误页面是为了引出两个重要的知识点，因此并没有着重介绍错误页面本身。这里只为 404 错误编写了自定义错误页面，对于另外两个常见的错误 400 错误和 500 错误，你可以自己试着为它们编写错误处理函数和对应的模板。
- 因为示例程序的语言和电影标题使用了英文，所以电影网站的搜索链接使用了 IMDb，对于中文，你可以使用豆瓣电影或时光网。以豆瓣电影为例，它的搜索链接为 https://movie.douban.com/subject_search?search_text=关键词，对应的 `href` 属性即 `https://movie.douban.com/subject_search?search_text={{ movie.title }}`。
- 因为基模板会被所有其他页面模板继承，如果你在基模板中使用了某个变量，那么这个变量也需要使用模板上下文处理函数注入到模板里。

第7章：表单

在HTML页面里，我们需要编写表单来获取用户输入。一个典型的表单如下所示：

```
<form method="post">  <!-- 指定提交方法为 POST -->
    <label for="name">名字</label>
    <input type="text" name="name" id="name"><br>  <!-- 文本输入框
-->
    <label for="occupation">职业</label>
    <input type="text" name="occupation" id="occupation"><br>  <
!-- 文本输入框 -->
    <input type="submit" name="submit" value="登录">  <!-- 提交按
钮 -->
</form>
```

编写表单的HTML代码有下面几点需要注意：

- 在`<form>`标签里使用`method`属性将提交表单数据的HTTP请求方法指定为POST。如果不指定，则会默认使用GET方法，这会将表单数据通过URL提交，容易导致数据泄露，而且不适用于包含大量数据的情况。
- `<input>`元素必须要指定`name`属性，否则无法提交数据，在服务器端，我们也要通过这个`name`属性值来获取对应字段的数据。

提示 填写输入框标签文字的`<label>`元素不是必须的，只是为了辅助鼠标用户。当使用鼠标点击标签文字时，会自动激活对应的输入框，这对复选框来说比较有用。`for`属性填入要绑定的`<input>`元素的`id`属性值。

创建新条目

创建新条目可以放到一个新的页面来实现，也可以直接在主页实现。这里我们采用后者，首先在主页模板里添加一个表单：

`templates/index.html`：添加创建新条目表单

```
<p>{{ movies|length }} Titles</p>
<form method="post">
    Name <input type="text" name="title" autocomplete="off" required>
    Year <input type="text" name="year" autocomplete="off" required>
    <input class="btn" type="submit" name="submit" value="Add">
</form>
```

在这两个输入字段中，`autocomplete` 属性设为 `off` 来关闭自动完成（按下输入框不显示历史输入记录）；另外还添加了 `required` 标志属性，如果用户没有输入内容就按下了提交按钮，浏览器会显示错误提示。

两个输入框和提交按钮相关的 CSS 定义如下：

```
/* 覆盖某些浏览器对 input 元素定义的字体 */
input[type=submit] {
    font-family: inherit;
}

input[type=text] {
    border: 1px solid #ddd;
}

input[name=year] {
    width: 50px;
}

.btn {
    font-size: 12px;
    padding: 3px 5px;
    text-decoration: none;
    cursor: pointer;
    background-color: white;
    color: black;
    border: 1px solid #555555;
    border-radius: 5px;
}

.btn:hover {
    text-decoration: none;
    background-color: black;
    color: white;
    border: 1px solid black;
}
```

接下来，我们需要考虑如何获取提交的表单数据。

处理表单数据

默认情况下，当表单中的提交按钮被按下，浏览器会创建一个新的请求，默认发往当前 URL（在 `<form>` 元素使用 `action` 属性可以自定义目标 URL）。

因为我们在模板里为表单定义了 POST 方法，当你输入数据，按下提交按钮，一个携带输入信息的 POST 请求会发往根地址。接着，你会看到一个 405 Method Not Allowed 错误提示。这是因为处理根地址请求的 index 视图默认只接受 GET 请求。

提示 在 HTTP 中，GET 和 POST 是两种最常见的请求方法，其中 GET 请求用来获取资源，而 POST 则用来创建 / 更新资源。我们访问一个链接时会发送 GET 请求，而提交表单通常会发送 POST 请求。

为了能够处理 POST 请求，我们需要修改一下视图函数：

```
@app.route('/', methods=['GET', 'POST'])
```

在 app.route() 装饰器里，我们可以用 methods 关键字传递一个包含 HTTP 方法字符串的列表，表示这个视图函数处理哪种方法类型的请求。默认只接受 GET 请求，上面的写法表示同时接受 GET 和 POST 请求。

两种方法的请求有不同的处理逻辑：对于 GET 请求，返回渲染后的页面；对于 POST 请求，则获取提交的表单数据并保存。为了在函数内加以区分，我们添加一个 if 判断：

app.py：创建电影条目

```

from flask import request, url_for, redirect, flash

# ...

@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST': # 判断是否是 POST 请求
        # 获取表单数据
        title = request.form.get('title') # 传入表单对应输入字段的
name 值
        year = request.form.get('year')
        # 验证数据
        if not title or not year or len(year) > 4 or len(title)
> 60:
            flash('Invalid input.') # 显示错误提示
            return redirect(url_for('index')) # 重定向回主页
        # 保存表单数据到数据库
        movie = Movie(title=title, year=year) # 创建记录
        db.session.add(movie) # 添加到数据库会话
        db.session.commit() # 提交数据库会话
        flash('Item created.') # 显示成功创建的提示
        return redirect(url_for('index')) # 重定向回主页

    user = User.query.first()
    movies = Movie.query.all()
    return render_template('index.html', user=user, movies=movies)

```

在 `if` 语句内，我们编写了处理表单数据的代码，其中涉及 3 个新的知识点，下面来一一了解。

请求对象

Flask 会在请求触发后把请求信息放到 `request` 对象里，你可以从 `flask` 包导入它：

```
from flask import request
```

因为它在请求触发时才会包含数据，所以你只能在视图函数内部调用它。它包含请求相关的所有信息，比如请求的路径（`request.path`）、请求的方法（`request.method`）、表单数据（`request.form`）、查询字符串（`request.args`）等等。

在上面的 `if` 语句中，我们首先通过 `request.method` 的值来判断请求方法。在 `if` 语句内，我们通过 `request.form` 来获取表单数据。`request.form` 是一个特殊的字典，用表单字段的 `name` 属性值可以获取用户填入的对应数据：

```
if request.method == 'POST':
    title = request.form.get('title')
    year = request.form.get('year')
```

flash 消息

在用户执行某些动作后，我们通常在页面上显示一个提示消息。最简单的实现就是在视图函数里定义一个包含消息内容的变量，传入模板，然后在模板里渲染显示它。因为这个需求很常用，Flask 内置了相关的函数。其中 `flash()` 函数用来在视图函数里向模板传递提示消息，`get_flashed_messages()` 函数则用来在模板中获取提示消息。

`flash()` 的用法很简单，首先从 `flask` 包导入 `flash` 函数：

```
from flask import flash
```

然后在视图函数里调用，传入要显示的消息内容：

```
flash('Item Created.')
```

`flash()` 函数在内部会把消息存储到 Flask 提供的 `session` 对象里。`session` 用来在请求间存储数据，它会把数据签名后存储到浏览器的 `Cookie` 中，所以我们需要设置签名所需的密钥：

```
app.config['SECRET_KEY'] = 'dev' # 等同于 app.secret_key = 'dev'
```

提示 这个密钥的值在开发时可以随便设置。基于安全的考虑，在部署时应该设置为随机字符，且不应该明文写在代码里，在部署章节会详细介绍。

下面在基模板（`base.html`）里使用 `get_flashed_messages()` 函数获取提示消息并显示：

```
<!-- 插入到页面标题上方 -->
{% for message in get_flashed_messages() %}
    <div class="alert">{{ message }}</div>
{% endfor %}
<h2>...</h2>
```

`alert` 类为提示消息增加样式：

```
.alert {
    position: relative;
    padding: 7px;
    margin: 7px 0;
    border: 1px solid transparent;
    color: #004085;
    background-color: #cce5ff;
    border-color: #b8daff;
    border-radius: 5px;
}
```

通过在 `<input>` 元素内添加 `required` 属性实现的验证（客户端验证）并不完全可靠，我们还要在服务器端追加验证：

```
if not title or not year or len(year) > 4 or len(title) > 60:
    flash('Invalid input.') # 显示错误提示
    return redirect(url_for('index'))
# ...
flash('Item created.') # 显示成功创建的提示
```

提示 在真实世界里，你会进行更严苛的验证，比如对数据去除首尾的空格。一般情况下，我们会使用第三方库（比如 [WTForms](#)）来实现表单数据的验证工作。

如果输入的某个数据为空，或是长度不符合要求，就显示错误提示“Invalid input.”，否则显示成功创建的提示“Item Created.”。

重定向响应

重定向响应是一类特殊的响应，它会返回一个新的 URL，浏览器在接受到这样的响应后会向这个新 URL 再次发起一个新的请求。Flask 提供了 `redirect()` 函数来快捷生成这种响应，传入重定向的目标 URL 作为参数，比如

```
redirect('http://helloflask.com')。
```

根据验证情况，我们发送不同的提示消息，最后都把页面重定向到主页，这里的主页 URL 均使用 `url_for()` 函数生成：

```
if not title or not year or len(year) > 4 or len(title) > 60:  
    flash('Invalid title or year!')  
    return redirect(url_for('index')) # 重定向回主页  
flash('Item created.')  
return redirect(url_for('index')) # 重定向回主页
```

编辑条目

编辑的实现和创建类似，我们先创建一个用于显示编辑页面和处理编辑表单提交请求的视图函数：

`app.py`：编辑电影条目

```

@app.route('/movie/edit/<int:movie_id>', methods=['GET', 'POST'])

def edit(movie_id):
    movie = Movie.query.get_or_404(movie_id)

    if request.method == 'POST': # 处理编辑表单的提交请求
        title = request.form['title']
        year = request.form['year']

        if not title or not year or len(year) > 4 or len(title) > 60:
            flash('Invalid input.')
            return redirect(url_for('edit', movie_id=movie_id))

    # 重定向回对应的编辑页面

    movie.title = title # 更新标题
    movie.year = year # 更新年份
    db.session.commit() # 提交数据库会话
    flash('Item updated.')
    return redirect(url_for('index')) # 重定向回主页

    return render_template('edit.html', movie=movie) # 传入被编辑的电影记录

```

这个视图函数的 URL 规则有一些特殊，如果你还有印象的话，我们在第 2 章的《实验时间》部分曾介绍过这种 URL 规则，其中的 `<int:movie_id>` 部分表示 URL 变量，而 `int` 则是将变量转换成整型的 URL 变量转换器。在生成这个视图的 URL 时，我们还需要传入对应的变量，比如 `url_for('edit', movie_id=2)` 会生成 `/movie/edit/2`。

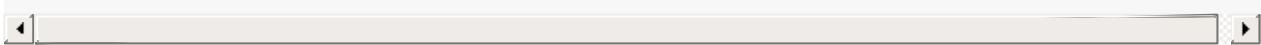
`movie_id` 变量是电影条目记录在数据库中的主键值，这个值用来在视图函数里查询到对应的电影记录。查询的时候，我们使用了 `get_or_404()` 方法，它会返回对应主键的记录，如果没有找到，则返回 404 错误响应。

为什么要在最后把电影记录传入模板？既然我们要编辑某个条目，那么必然要在输入框里提前把对应的数据放进去，以便于进行更新。在模板里，通过表单 `<input>` 元素的 `value` 属性即可将它们提前写到输入框里。完整的编辑页面模板如下所示：

templates/edit.html：编辑页面模板

```
{% extends 'base.html' %}

{% block content %}
<h3>Edit item</h3>
<form method="post">
    Name <input type="text" name="title" autocomplete="off" required value="{{ movie.title }}>
    Year <input type="text" name="year" autocomplete="off" required value="{{ movie.year }}>
    <input class="btn" type="submit" name="submit" value="Update">
</form>
{% endblock %}
```

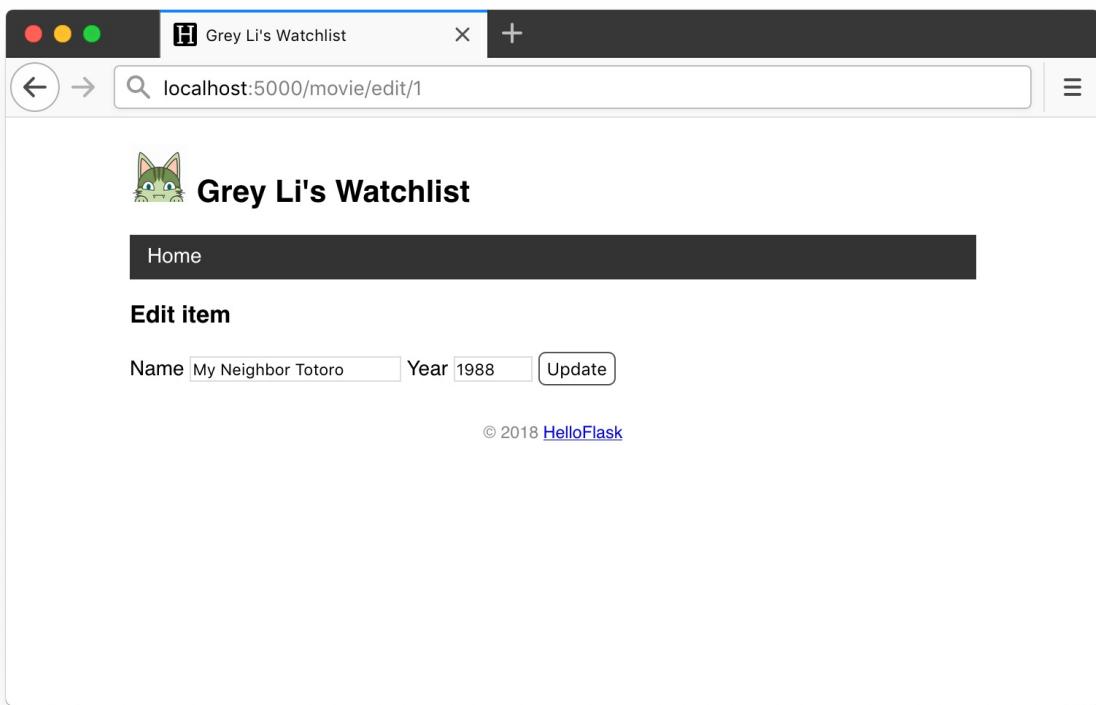


最后在主页每一个电影条目右侧都添加一个指向该条目编辑页面的链接：

index.html：编辑电影条目的链接

```
<span class="float-right">
    <a class="btn" href="{{ url_for('edit', movie_id=movie.id) }}>Edit</a>
    ...
</span>
```

点击某一个电影条目的编辑按钮打开的编辑页面如下图所示：



删除条目

因为不涉及数据的传递，删除条目的实现更加简单。首先创建一个视图函数执行删除操作，如下所示：

app.py：删除电影条目

```
@app.route('/movie/delete/<int:movie_id>', methods=['POST']) #  
限定只接受 POST 请求  
def delete(movie_id):  
    movie = Movie.query.get_or_404(movie_id) # 获取电影记录  
    db.session.delete(movie) # 删除对应的记录  
    db.session.commit() # 提交数据库会话  
    flash('Item deleted.')  
    return redirect(url_for('index')) # 重定向回主页
```

为了安全的考虑，我们一般会使用 POST 请求来提交删除请求，也就是使用表单来实现（而不是创建删除链接）：

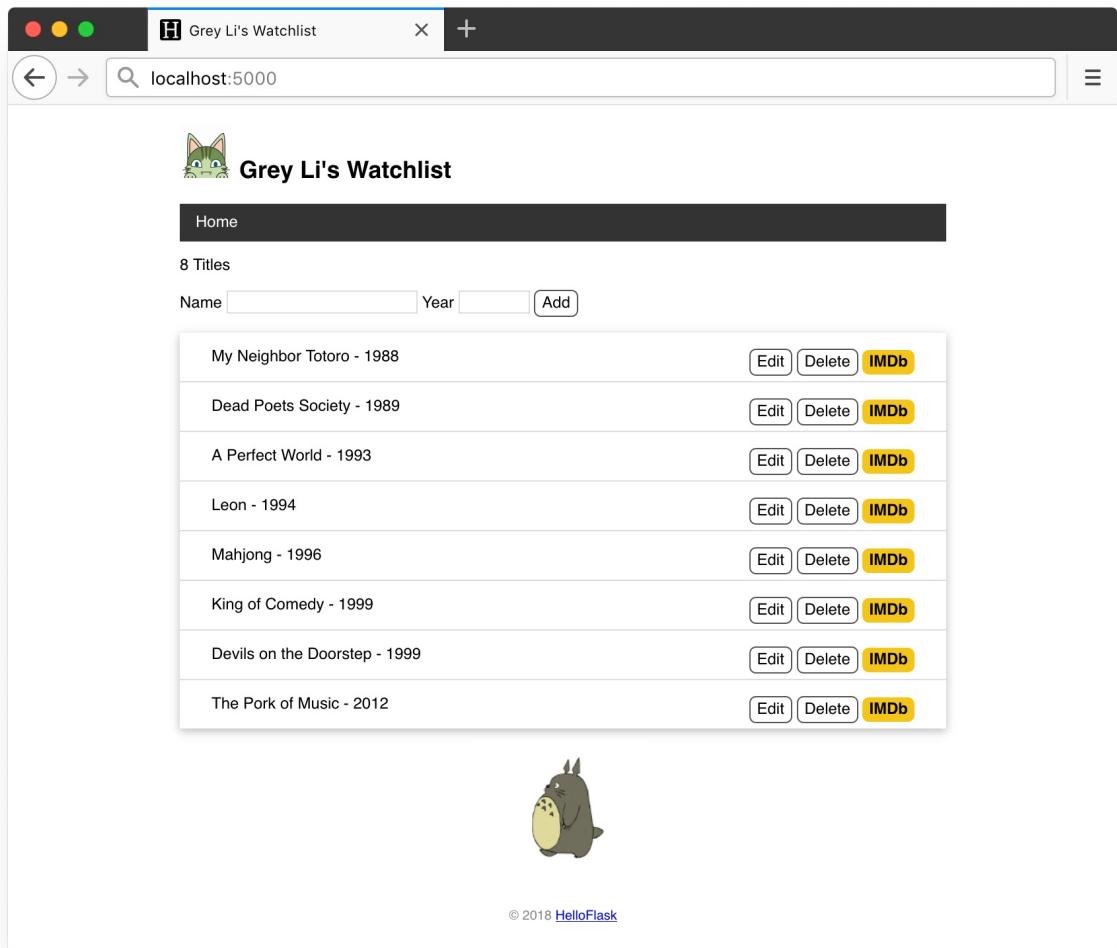
index.html：删除电影条目表单

```
<span class="float-right">
    ...
    <form class="inline-form" method="post" action="{{ url_for('
delete', movie_id=movie.id) }}>
        <input class="btn" type="submit" name="delete" value="De
lete" onclick="return confirm('Are you sure?')">
    </form>
    ...
</span>
```

为了让表单中的删除按钮和旁边的编辑链接排成一行，我们为表单元素添加了下面的 CSS 定义：

```
.inline-form {
    display: inline;
}
```

最终的程序主页如下图所示：



本章小结

本章我们完成了程序的主要功能：添加、编辑和删除电影条目。结束前，让我们提交代码：

```
$ git add .
$ git commit -m "Create, edit and delete item by form"
$ git push
```

提示 你可以在 GitHub 上查看本书示例程序的对应 commit：[84e766f](#)。在后续的 [commit](#) 里，我们为另外两个常见的 HTTP 错误：400（Bad Request）和 500（Internal Server Error）错误编写了错误处理函数和对应的模板，前者会在请求格式不符要求时返回，后者则会在程序内部出现任意错误时返回（关闭调试模式的情况下）。

进阶提示

- 从上面的代码可以看出，手动验证表单数据既麻烦又不可靠。对于复杂的程序，我们一般会使用集成了 WTForms 的扩展 [Flask-WTF](#) 来简化表单处理。通过编写表单类，定义表单字段和验证器，它可以自动生成表单对应的 HTML 代码，并在表单提交时验证表单数据，返回对应的错误消息。更重要的，它还内置了 CSRF（跨站请求伪造）保护功能。你可以阅读 [Flask-WTF 文档](#) 和 [Hello, Flask!](#) 专栏上的 [表单系列文章](#) 了解具体用法。
- CSRF 是一种常见的攻击手段。以我们的删除表单为例，某恶意网站的页面中内嵌了一段代码，访问时会自动发送一个删除某个电影条目的 POST 请求到我们的程序。如果我们访问了这个恶意网站，就会导致电影条目被删除，因为我们的程序没法分辨请求发自哪里。解决方法通常是在表单里添加一个包含随机字符串的隐藏字段，同时在 Cookie 中也创建一个同样的随机字符串，在提交时通过对比两个值是否一致来判断是否是用户自己发送的请求。在我们的程序中没有实现 CSRF 保护。
- 使用 [Flask-WTF](#) 时，表单类在模板中的渲染代码基本相同，你可以编写宏来渲染表单字段。如果你使用 [Bootstrap](#)，那么扩展 [Bootstrap-Flask](#) 内置了多个表单相关的宏，可以简化渲染工作。
- 你可以把删除按钮的行内 JavaScript 代码改为事件监听函数，写到单独的 JavaScript 文件里。再进一步，你也可以使用 JavaScript 来监听点击删除按钮的动作，并发送删除条目的 POST 请求，这样删除按钮就可以使用普通 `<a>` 标签（CSRF 令牌存储在元素属性里），而不用创建表单元素。
- 如果你是《[Flask Web 开发实战](#)》的读者，第 4 章介绍了表单处理的各个方面，包括表单类的编写和渲染、错误消息显示、自定义错误消息语言、文件和多文件上传、富文本编辑器等等。

第 8 章：用户认证

目前为止，虽然程序的功能大部分已经实现，但还缺少一个非常重要的部分——用户认证保护。页面上的编辑和删除按钮是公开的，所有人都可以看到。假如我们现在把程序部署到网络上，那么任何人都可以执行编辑和删除条目的操作，这显然是不合理的。

这一章我们会为程序添加用户认证功能，这会把用户分成两类，一类是管理员，通过用户名和密码登入程序，可以执行数据相关的操作；另一个是访客，只能浏览页面。在此之前，我们先来看看密码应该如何安全的存储到数据库中。

安全存储密码

把密码明文存储在数据库中是极其危险的，假如攻击者窃取了你的数据库，那么用户的账号和密码就会被直接泄露。更保险的方式是对每个密码进行计算生成独一无二的密码散列值，这样即使攻击者拿到了散列值，也几乎无法逆向获取到密码。

Flask 的依赖 Werkzeug 内置了用于生成和验证密码散列值的函数，`werkzeug.security.generate_password_hash()` 用来为给定的密码生成密码散列值，而 `werkzeug.security.check_password_hash()` 则用来检查给定的散列值和密码是否对应。使用示例如下所示：

```
>>> from werkzeug.security import generate_password_hash, check_
password_hash
>>> pw_hash = generate_password_hash('dog') # 为密码 dog 生成密码
散列值
>>> pw_hash # 查看密码散列值
'pbkdf2:sha256:50000$mm9UPTRI$ee68ebc71434a4405a28d34ae3f170757f
b424663dc0ca15198cb881edc0978f'
>>> check_password_hash(pw_hash, 'dog') # 检查散列值是否对应密码 dog
True
>>> check_password_hash(pw_hash, 'cat') # 检查散列值是否对应密码 cat
False
```

我们在存储用户信息的 `User` 模型类添加 `username` 字段和 `password_hash` 字段，分别用来存储登录所需的用户名和密码散列值，同时添加两个方法来实现设置密码和验证密码的功能：

```
from werkzeug.security import generate_password_hash, check_password_hash

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(20))
    username = db.Column(db.String(20)) # 用户名
    password_hash = db.Column(db.String(128)) # 密码散列值

    def set_password(self, password): # 用来设置密码的方法，接受密码作为参数
        self.password_hash = generate_password_hash(password) # 将生成的密码保持到对应字段

    def validate_password(self, password): # 用于验证密码的方法，接受密码作为参数
        return check_password_hash(self.password_hash, password)
        # 返回布尔值
```

因为模型（表结构）发生变化，我们需要重新生成数据库（这会清空数据）：

```
(env) $ flask initdb --drop
```

生成管理员账户

因为程序只允许一个人使用，没有必要编写一个注册页面。我们可以编写一个命令来创建管理员账户，下面是实现这个功能的 `admin()` 函数：

```

import click

@app.cli.command()
@click.option('--username', prompt=True, help='The username used
to login.')
@click.option('--password', prompt=True, hide_input=True, confirmation_prompt=True, help='The password used to login.')
def admin(username, password):
    """Create user."""
    db.create_all()

    user = User.query.first()
    if user is not None:
        click.echo('Updating user...')
        user.username = username
        user.set_password(password) # 设置密码
    else:
        click.echo('Creating user...')
        user = User(username=username, name='Admin')
        user.set_password(password) # 设置密码
        db.session.add(user)

    db.session.commit() # 提交数据库会话
    click.echo('Done.')

```

使用 `click.option()` 装饰器设置的两个选项分别用来接受输入用户名和密码。执行 `flask admin` 命令，输入用户名和密码后，即可创建管理员账户。如果执行这个命令时账户已存在，则更新相关信息：

```

(env) $ flask admin
Username: greyli
Password: 123 # hide_input=True 会让密码输入隐藏
Repeat for confirmation: 123 # confirmation_prompt=True 会要求二次确认输入
Updating user...
Done.

```

使用 Flask-Login 实现用户认证

扩展 [Flask-Login](#) 提供了实现用户认证需要的各类功能函数，我们将使用它来实现程序的用户认证，首先来安装它：

```
(env) $ pip install flask-login
```

这个扩展的初始化步骤稍微有些不同，除了实例化扩展类之外，我们还要实现一个“用户加载回调函数”，具体代码如下所示：

app.py：初始化 *Flask-Login*

```
from flask_login import LoginManager

login_manager = LoginManager(app) # 实例化扩展类

@login_manager.user_loader
def load_user(user_id): # 创建用户加载回调函数，接受用户 ID 作为参数
    user = User.query.get(int(user_id)) # 用 ID 作为 User 模型的
    主键查询对应的用户
    return user # 返回用户对象
```

Flask-Login 提供了一个 `current_user` 变量，注册这个函数的目的是，当程序运行后，如果用户已登录，`current_user` 变量的值会是当前用户的用户模型类记录。

另一个步骤是让存储用户的 `User` 模型类继承 `Flask-Login` 提供的 `UserMixin` 类：

```
from flask_login import UserMixin

class User(db.Model, UserMixin):
    # ...
```

继承这个类会让 `User` 类拥有几个用于判断认证状态的属性和方法，其中最常用的是 `is_authenticated` 属性：如果当前用户已经登录，那么 `current_user.is_authenticated` 会返回 `True`，否则返回 `False`。有了 `current_user` 变量和这几个验证方法和属性，我们可以很轻松的判断当前用户的认证状态。

登 录

登录用户使用 `Flask-Login` 提供的 `login_user()` 函数实现，需要传入用户模型类对象作为参数。下面是用于显示登录页面和处理登录表单提交请求的视图函数：

`app.py`：用户登录

```
from flask_login import login_user

# ...

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        if not username or not password:
            flash('Invalid input.')
            return redirect(url_for('login'))

        user = User.query.first()
        # 验证用户名和密码是否一致
        if username == user.username and user.validate_password(
password):
            login_user(user) # 登入用户
            flash('Login success.')
            return redirect(url_for('index')) # 重定向到主页

            flash('Invalid username or password.') # 如果验证失败，显示错误消息
            return redirect(url_for('login')) # 重定向回登录页面

    return render_template('login.html')
```

下面是包含登录表单的登录页面模板：

templates/login.html：登录页面

```
{% extends 'base.html' %}

{% block content %}
<h3>Login</h3>
<form method="post">
    Username<br>
    <input type="text" name="username" required><br><br>
    Password<br>
    <!-- 密码输入框的 type 属性使用 password，会将输入值显示为圆点 -->
    <input type="password" name="password" required><br><br>
    <input class="btn" type="submit" name="submit" value="Submit"
>
</form>
{% endblock %}
```

登出

和登录相对，登出操作则需要调用 `logout_user()` 函数，使用下面的视图函数实现：

```
from flask_login import login_required, logout_user

# ...

@app.route('/logout')
@login_required # 用于视图保护，后面会详细介绍
def logout():
    logout_user() # 登出用户
    flash('Goodbye.')
    return redirect(url_for('index')) # 重定向回首页
```

实现了登录和登出后，我们先来看看认证保护，最后再把对应这两个视图函数的登录/登出链接放到导航栏上。

认证保护

在 Web 程序中，有些页面或 URL 不允许未登录的用户访问，而页面上有些内容则需要对未登陆的用户隐藏，这就是认证保护。

视图保护

在视图保护层面来说，未登录用户不能执行下面的操作：

- 访问编辑页面
- 访问设置页面
- 执行注销操作
- 执行删除操作
- 执行添加新条目操作

对于不允许未登录用户访问的视图，只需要为视图函数附加一个 `login_required` 装饰器就可以将未登录用户拒之门外。以删除条目视图为例：

```
@app.route('/movie/delete/<int:movie_id>', methods=['POST'])
@login_required # 登录保护
def delete(movie_id):
    movie = Movie.query.get_or_404(movie_id)
    db.session.delete(movie)
    db.session.commit()
    flash('Item deleted.')
    return redirect(url_for('index'))
```

添加了这个装饰器后，如果未登录的用户访问对应的 URL，Flask-Login 会把用户重定向到登录页面，并显示一个错误提示。为了让这个重定向操作正确执行，我们还需要把 `login_manager.login_view` 的值设为我们程序的登录视图端点（函数名）：

```
login_manager.login_view = 'login'
```

提示 如果你需要的话，可以通过设置 `login_manager.login_message` 来自定义错误提示消息。

编辑视图同样需要附加这个装饰器：

```
@app.route('/movie/edit/<int:movie_id>', methods=['GET', 'POST'])

@login_required
def edit(movie_id):
    # ...
```

创建新条目的操作稍微有些不同，因为对应的视图同时处理显示页面的 GET 请求和创建新条目的 POST 请求，我们仅需要禁止未登录用户创建新条目，因此不能使用 `login_required`，而是在函数内部的 POST 请求处理代码前进行过滤：

```
from flask_login import login_required, current_user

# ...

@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        if not current_user.is_authenticated: # 如果当前用户未认证
            return redirect(url_for('index')) # 重定向到主页
    # ...
```

最后，我们为程序添加一个设置页面，支持修改用户名字：

`app.py`：支持设置用户名字

```
from flask_login import login_required, current_user

# ...

@app.route('/settings', methods=['GET', 'POST'])
@login_required
def settings():
    if request.method == 'POST':
        name = request.form['name']

        if not name or len(name) > 20:
            flash('Invalid input.')
            return redirect(url_for('settings'))

        current_user.name = name
        # current_user 会返回当前登录用户的数据库记录对象
        # 等同于下面的用法
        # user = User.query.first()
        # user.name = name
        db.session.commit()
        flash('Settings updated.')
        return redirect(url_for('index'))

    return render_template('settings.html')
```

下面是对应的模板：

templates/settings.html：设置页面模板

```
{% extends 'base.html' %}

{% block content %}
<h3>Settings</h3>
<form method="post">
    Your Name <input type="text" name="name" autocomplete="off"
    required value="{{ current_user.name }}>
    <input class="btn" type="submit" name="submit" value="Save">
</form>
{% endblock %}
```

模板内容保护

认证保护的另一形式是页面模板内容的保护。比如，不能对未登录用户显示下列内容：

- 创建新条目表单
- 编辑按钮
- 删除按钮

这几个元素的定义都在首页模板（index.html）中，以创建新条目表单为例，我们在表单外部添加一个 `if` 判断：

```
<!-- 在模板中可以直接使用 current_user 变量 -->
{% if current_user.is_authenticated %}
<form method="post">
    Name <input type="text" name="title" autocomplete="off" required>
    Year <input type="text" name="year" autocomplete="off" required>
    <input class="btn" type="submit" name="submit" value="Add">
</form>
{% endif %}
```

在模板渲染时，会先判断当前用户的登录状态

`(current_user.is_authenticated)`。如果用户没有登录
`(current_user.is_authenticated 返回 False)`，就不会渲染表单部分的

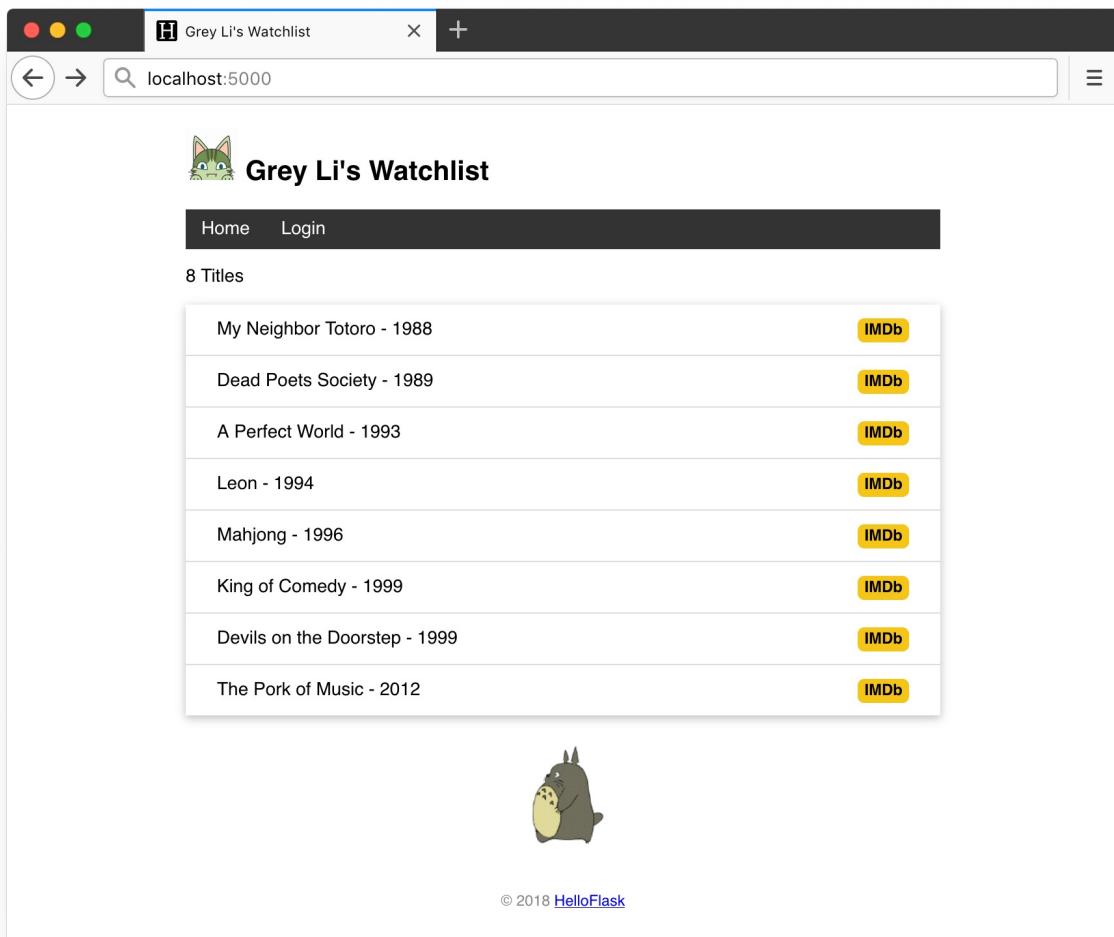
HTML代码，即上面代码块中`{% if ... %}`和`{% endif %}`之间的代码。类似的还有编辑和删除按钮：

```
{% if current_user.is_authenticated %}
    <a class="btn" href="{{ url_for('edit', movie_id=movie.id) }">
}">Edit</a>
    <form class="inline-form" method="post" action="{{ url_for(
.delete', movie_id=movie.id) }}">
        <input class="btn" type="submit" name="delete" value="Delete" onclick="return confirm('Are you sure?')">
    </form>
{% endif %}
```

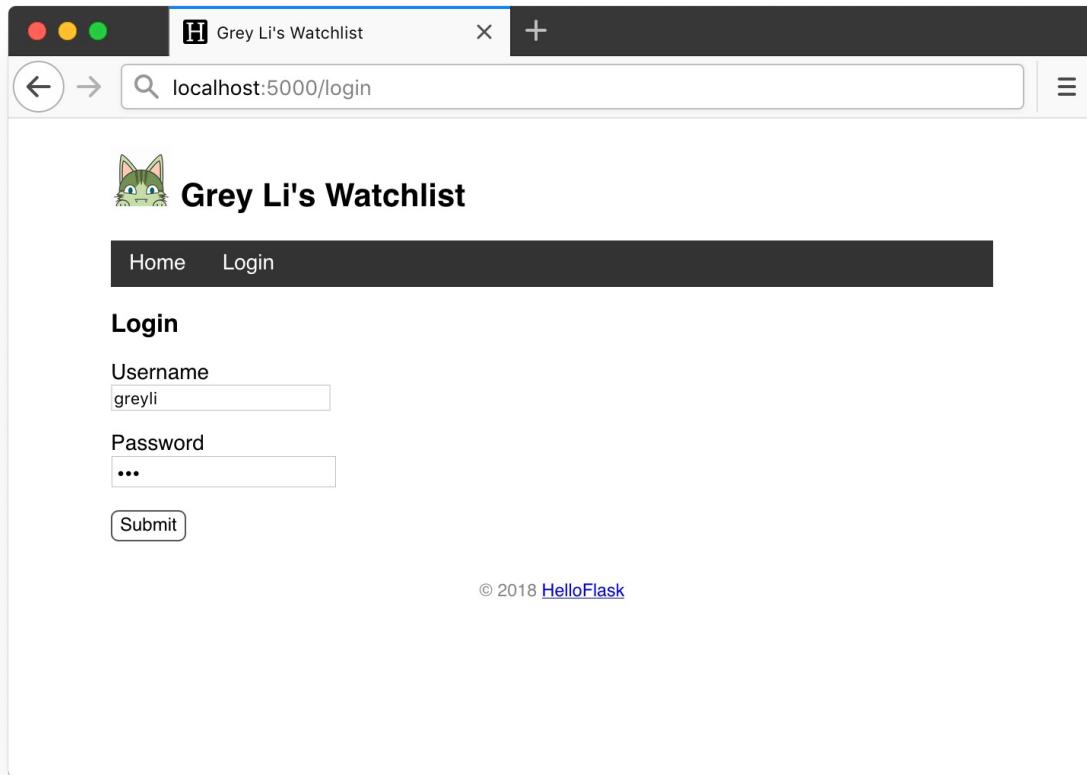
有些地方则需要根据登录状态分别显示不同的内容，比如基模板（base.html）中的导航栏。如果用户已经登录，就显示设置和登出链接，否则显示登录链接：

```
{% if current_user.is_authenticated %}
    <li><a href="{{ url_for('settings') }}>Settings</a></li>
    <li><a href="{{ url_for('logout') }}>Logout</a></li>
{% else %}
    <li><a href="{{ url_for('login') }}>Login</a></li>
{% endif %}
```

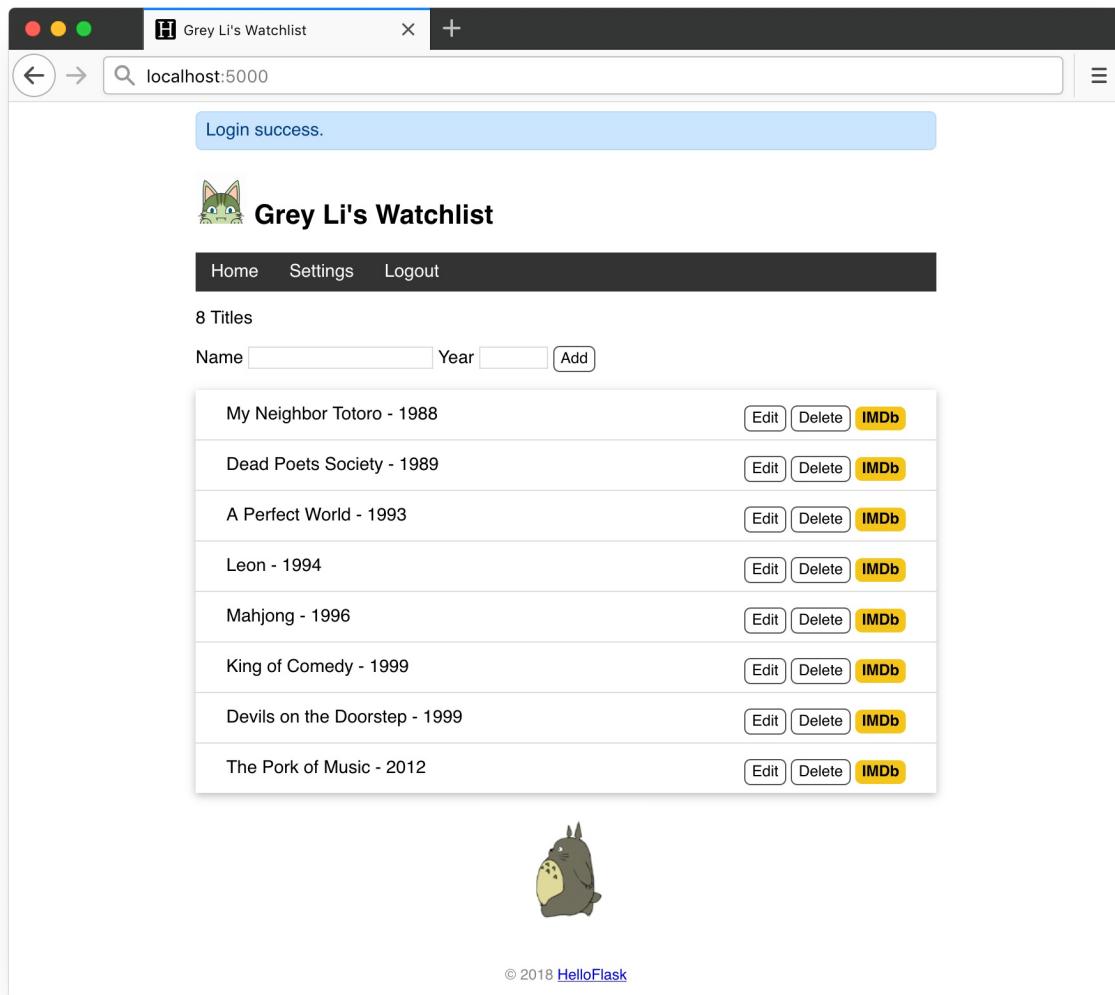
现在的程序中，未登录用户看到的主页如下所示：



在登录页面，输入用户名和密码登入：



登录后看到的主页如下所示：



本章小结

添加用户认证后，在功能层面，我们的程序基本算是完成了。结束前，让我们提交代码：

```
$ git add .
$ git commit -m "User authentication with Flask-Login"
$ git push
```

提示 你可以在 GitHub 上查看本书示例程序的对应 commit：[6c60b7d](#)。

进阶提示

- 访问 [Flask-Login 文档](#) 了解更多细节和用法。
- 如果你是《[Flask Web 开发实战](#)》的读者，第 2 章通过一个示例介绍了用户认证的实现方式；第 8 章包含对 Flask-Login 更详细的介绍。

第9章：测试

在此之前，每次为程序添加了新功能，我们都要手动在浏览器里访问程序进行测试。除了测试新添加的功能，你还要确保旧的功能依然正常工作。在功能复杂的大程序里，如果每次修改代码或添加新功能后手动测试所有功能，那会产生很大的工作量。另一方面，手动测试并不可靠，重复进行测试操作也很枯燥。

基于这些原因，为程序编写自动化测试就变得非常重要。

注意 为了便于介绍，本书统一在这里介绍关于测试的内容。在实际的项目开发中，你应该在开发每一个功能后立刻编写相应的测试，确保测试通过后再开发下一个功能。

单元测试

单元测试指对程序中的函数等独立单元编写的测试，它是自动化测试最主要的形式。这一章我们将会使用 Python 标准库中的测试框架 `unittest` 来编写单元测试，首先通过一个简单的例子来了解一些基本概念。假设我们编写了下面这个函数：

```
def sayhello(to=None):
    if to:
        return 'Hello, %s!' % to
    return 'Hello!'
```

下面是我们为这个函数编写的单元测试：

```

import unittest

from module_foo import sayhello


class SayHelloTestCase(unittest.TestCase): # 测试用例

    def setUp(self): # 测试固件
        pass

    def tearDown(self): # 测试固件
        pass

    def test_sayhello(self): # 第 1 个测试
        rv = sayhello()
        self.assertEqual(rv, 'Hello!')

    def test_sayhello_to_somebody(self) # 第 2 个测试
        rv = sayhello(to='Grey')
        self.assertEqual(rv, 'Hello, Grey!')


if __name__ == '__main__':
    unittest.main()

```

测试用例继承 `unittest.TestCase` 类，在这个类中创建的以 `test_` 开头的方法将会被视为测试方法。

内容为空的两个方法很特殊，它们是测试固件，用来执行一些特殊操作。比如 `setUp()` 方法会在每个测试方法执行前被调用，而 `tearDown()` 方法则会在每一个测试方法执行后被调用（注意这两个方法名称的大小写）。

如果把执行测试方法比作战斗，那么准备弹药、规划战术的工作就要在 `setUp()` 方法里完成，而打扫战场则要在 `tearDown()` 方法里完成。

每一个测试方法（名称以 `test_` 开头的方法）对应一个要测试的函数 / 功能 / 使用场景。在上面我们创建了两个测试方法，`test_sayhello()` 方法测试 `sayhello()` 函数，`test_sayhello_to_somebody()` 方法测试传入参数时的 `sayhello()` 函数。

在测试方法里，我们使用断言方法来判断程序功能是否正常。以第一个测试方法为例，我们先把 `sayhello()` 函数调用的返回值保存为 `rv` 变量（`return value`），然后使用 `self.assertEqual(rv, 'Hello!')` 来判断返回值内容是否符合预期。如果断言方法出错，就表示该测试方法未通过。

下面是一些常用的断言方法：

- `assertEqual(a, b)`
- `assertNotEqual(a, b)`
- `assertTrue(x)`
- `assertFalse(x)`
- `assertIs(a, b)`
- `assertIsNot(a, b)`
- `assertIsNone(x)`
- `assertIsNotNone(x)`
- `assertIn(a, b)`
- `assertNotIn(a, b)`

这些方法的作用从方法名称上基本可以得知。

假设我们把上面的测试代码保存到 `test_sayhello.py` 文件中，通过执行 `python test_sayhello.py` 命令即可执行所有测试，并输出测试的结果、通过情况、总耗时等信息。

测试 Flask 程序

回到我们的程序，我们在项目根目录创建一个 `test_watchlist.py` 脚本来存储测试代码，我们先编写测试固件和两个简单的基础测试：

`test_watchlist.py`：测试固件

```
import unittest

from app import app, db, Movie, User


class WatchlistTestCase(unittest.TestCase):

    def setUp(self):
        # 更新配置
        app.config.update(
            TESTING=True,
            SQLALCHEMY_DATABASE_URI='sqlite:///memory:'
        )
        # 创建数据库和表
        db.create_all()
        # 创建测试数据，一个用户，一个电影条目
        user = User(name='Test', username='test')
        user.set_password('123')
        movie = Movie(title='Test Movie Title', year='2019')
        # 使用 add_all() 方法一次添加多个模型类实例，传入列表
        db.session.add_all([user, movie])
        db.session.commit()

        self.client = app.test_client() # 创建测试客户端
        self.runner = app.test_cli_runner() # 创建测试命令运行器

    def tearDown(self):
        db.session.remove() # 清除数据库会话
        db.drop_all() # 删除数据库表

    # 测试程序实例是否存在
    def test_app_exist(self):
        self.assertIsNotNone(app)

    # 测试程序是否处于测试模式
    def test_app_is_testing(self):
        self.assertTrue(app.config['TESTING'])
```

某些配置，在开发和测试时通常需要使用不同的值。在 `setUp()` 方法中，我们更新了两个配置变量的值，首先将 `TESTING` 设为 `True` 来开启测试模式，这样在出错时不会输出多余信息；然后将 `SQLALCHEMY_DATABASE_URI` 设为 `'sqlite:///memory:'`，这会使用 SQLite 内存型数据库，不会干扰开发时使用的数据库文件。你也可以使用不同文件名的 SQLite 数据库文件，但内存型数据库速度更快。

接着，我们调用 `db.create_all()` 创建数据库和表，然后添加测试数据到数据库中。在 `setUp()` 方法最后创建的两个类属性分别为测试客户端和测试命令运行器，前者用来模拟客户端请求，后者用来触发自定义命令，下一节会详细介绍。

在 `tearDown()` 方法中，我们调用 `db.session.remove()` 清除数据库会话并调用 `db.drop_all()` 删除数据库表。测试时的程序状态和真实的程序运行状态不同，所以需要调用 `db.session.remove()` 来确保数据库会话被清除。

测试客户端

`app.test_client()` 返回一个测试客户端对象，可以用来模拟客户端（浏览器），我们创建类属性 `self.client` 来保存它。对它调用 `get()` 方法就相当于浏览器向服务器发送 GET 请求，调用 `post()` 则相当于浏览器向服务器发送 POST 请求，以此类推。下面是两个发送 GET 请求的测试方法，分别测试 404 页面和主页：

`test_watchlist.py`：测试固件

```

class WatchlistTestCase(unittest.TestCase):
    # ...
    # 测试 404 页面
    def test_404_page(self):
        response = self.client.get('/nothing') # 传入目标 URL
        data = response.get_data(as_text=True)
        self.assertIn('Page Not Found - 404', data)
        self.assertIn('Go Back', data)
        self.assertEqual(response.status_code, 404) # 判断响应状态码

    # 测试主页
    def test_index_page(self):
        response = self.client.get('/')
        data = response.get_data(as_text=True)
        self.assertIn('Test\'s Watchlist', data)
        self.assertIn('Test Movie Title', data)
        self.assertEqual(response.status_code, 200)

```

调用这类方法返回包含响应数据的响应对象，对这个响应对象调用 `get_data()` 方法并把 `as_text` 参数设为 `True` 可以获取 `Unicode` 格式的响应主体。我们通过判断响应主体中是否包含预期的内容来测试程序是否正常工作，比如 `404` 页面响应是否包含 `Go Back`，主页响应是否包含标题 `Test's Watchlist`。

接下来，我们要测试数据库操作相关的功能，比如创建、更新和删除电影条目。这些操作对应的请求都需要登录账户后才能发送，我们先编写一个用于登录账户的辅助方法：

`test_watchlist.py`：测试辅助方法

```

class WatchlistTestCase(unittest.TestCase):
    # ...
    # 辅助方法，用于登入用户
    def login(self):
        self.client.post('/login', data=dict(
            username='test',
            password='123'
        ), follow_redirects=True)

```

在 `login()` 方法中，我们使用 `post()` 方法发送提交登录表单的 POST 请求。和 `get()` 方法类似，我们需要先传入目标 URL，然后使用 `data` 关键字以字典的形式传入请求数据（字典中的键为表单 `<input>` 元素的 `name` 属性值），作为登录表单的输入数据；而将 `follow_redirects` 参数设为 `True` 可以跟随重定向，最终返回的会是重定向后的响应。

下面是测试创建、更新和删除条目的测试方法：

`test_watchlist.py`：测试创建、更新和删除条目

```
class WatchlistTestCase(unittest.TestCase):
    # ...
    # 测试创建条目
    def test_create_item(self):
        self.login()

        # 测试创建条目操作
        response = self.client.post('/', data=dict(
            title='New Movie',
            year='2019'
        ), follow_redirects=True)
        data = response.get_data(as_text=True)
        self.assertIn('Item created.', data)
        self.assertIn('New Movie', data)

        # 测试创建条目操作，但电影标题为空
        response = self.client.post('/', data=dict(
            title='',
            year='2019'
        ), follow_redirects=True)
        data = response.get_data(as_text=True)
        self.assertNotIn('Item created.', data)
        self.assertIn('Invalid input.', data)

        # 测试创建条目操作，但电影年份为空
        response = self.client.post('/', data=dict(
            title='New Movie',
            year=''
        ), follow_redirects=True)
        data = response.get_data(as_text=True)
```

```
    self.assertNotIn('Item created.', data)
    self.assertIn('Invalid input.', data)

# 测试更新条目
def test_update_item(self):
    self.login()

    # 测试更新页面
    response = self.client.get('/movie/edit/1')
    data = response.get_data(as_text=True)
    self.assertIn('Edit item', data)
    self.assertIn('Test Movie Title', data)
    self.assertIn('2019', data)

    # 测试更新条目操作
    response = self.client.post('/movie/edit/1', data=dict(
        title='New Movie Edited',
        year='2019'
    ), follow_redirects=True)
    data = response.get_data(as_text=True)
    self.assertIn('Item updated.', data)
    self.assertIn('New Movie Edited', data)

    # 测试更新条目操作，但电影标题为空
    response = self.client.post('/movie/edit/1', data=dict(
        title='',
        year='2019'
    ), follow_redirects=True)
    data = response.get_data(as_text=True)
    self.assertNotIn('Item updated.', data)
    self.assertIn('Invalid input.', data)

    # 测试更新条目操作，但电影年份为空
    response = self.client.post('/movie/edit/1', data=dict(
        title='New Movie Edited Again',
        year=''
    ), follow_redirects=True)
    data = response.get_data(as_text=True)
    self.assertNotIn('Item updated.', data)
    self.assertNotIn('New Movie Edited Again', data)
```

```

        self.assertIn('Invalid input.', data)

# 测试删除条目
def test_delete_item(self):
    self.login()

    response = self.client.post('/movie/delete/1', follow_redirects=True)
    data = response.get_data(as_text=True)
    self.assertIn('Item deleted.', data)
    self.assertNotIn('Test Movie Title', data)

```

在这几个测试方法中，大部分的断言都是在判断响应主体是否包含正确的提示消息和电影条目信息。

登录、登出和认证保护等功能的测试如下所示：

test_watchlist.py：测试认证相关功能

```

class WatchlistTestCase(unittest.TestCase):
    # ...
    # 测试登录保护
    def test_login_protect(self):
        response = self.client.get('/')
        data = response.get_data(as_text=True)
        self.assertNotIn('Logout', data)
        self.assertNotIn('Settings', data)
        self.assertNotIn('<form method="post">', data)
        self.assertNotIn('Delete', data)
        self.assertNotIn('Edit', data)

    # 测试登录
    def test_login(self):
        response = self.client.post('/login', data=dict(
            username='test',
            password='123'
        ), follow_redirects=True)
        data = response.get_data(as_text=True)
        self.assertIn('Login success.', data)
        self.assertIn('Logout', data)

```

```
    self.assertIn('Settings', data)
    self.assertIn('Delete', data)
    self.assertIn('Edit', data)
    self.assertIn('<form method="post">', data)

    # 测试使用错误的密码登录
    response = self.client.post('/login', data=dict(
        username='test',
        password='456'
    ), follow_redirects=True)
    data = response.get_data(as_text=True)
    self.assertNotIn('Login success.', data)
    self.assertIn('Invalid username or password.', data)

    # 测试使用错误的用户名登录
    response = self.client.post('/login', data=dict(
        username='wrong',
        password='123'
    ), follow_redirects=True)
    data = response.get_data(as_text=True)
    self.assertNotIn('Login success.', data)
    self.assertIn('Invalid username or password.', data)

    # 测试使用空用户名登录
    response = self.client.post('/login', data=dict(
        username='',
        password='123'
    ), follow_redirects=True)
    data = response.get_data(as_text=True)
    self.assertNotIn('Login success.', data)
    self.assertIn('Invalid input.', data)

    # 测试使用空密码登录
    response = self.client.post('/login', data=dict(
        username='test',
        password=''
    ), follow_redirects=True)
    data = response.get_data(as_text=True)
    self.assertNotIn('Login success.', data)
    self.assertIn('Invalid input.', data)
```

```
# 测试登出
def test_logout(self):
    self.login()

    response = self.client.get('/logout', follow_redirects=True)
    data = response.get_data(as_text=True)
    self.assertIn('Goodbye.', data)
    self.assertNotIn('Logout', data)
    self.assertNotIn('Settings', data)
    self.assertNotIn('Delete', data)
    self.assertNotIn('Edit', data)
    self.assertNotIn('<form method="post">', data)

# 测试设置
def test_settings(self):
    self.login()

    # 测试设置页面
    response = self.client.get('/settings')
    data = response.get_data(as_text=True)
    self.assertIn('Settings', data)
    self.assertIn('Your Name', data)

    # 测试更新设置
    response = self.client.post('/settings', data=dict(
        name='Grey Li',
    ), follow_redirects=True)
    data = response.get_data(as_text=True)
    self.assertIn('Settings updated.', data)
    self.assertIn('Grey Li', data)

    # 测试更新设置，名称为空
    response = self.client.post('/settings', data=dict(
        name='',
    ), follow_redirects=True)
    data = response.get_data(as_text=True)
    self.assertNotIn('Settings updated.', data)
    self.assertIn('Invalid input.', data)
```

测试命令

除了测试程序的各个视图函数，我们还需要测试自定义命令。`app.test_cli_runner()` 方法返回一个命令运行器对象，我们创建类属性 `self.runner` 来保存它。通过对它调用 `invoke()` 方法可以执行命令，传入命令函数对象，或是使用 `args` 关键字直接给出命令参数列表。`invoke()` 方法返回的命令执行结果对象，它的 `output` 属性返回命令的输出信息。下面是我们为各个自定义命令编写的测试方法：

`test_watchlist.py`：测试自定义命令行命令

```
# 导入命令函数
from app import app, db, Movie, User, forge, initdb


class WatchlistTestCase(unittest.TestCase):
    # ...
    # 测试虚拟数据
    def test_forge_command(self):
        result = self.runner.invoke(forge)
        self.assertIn('Done.', result.output)
        self.assertEqual(Movie.query.count(), 0)

    # 测试初始化数据库
    def test_initdb_command(self):
        result = self.runner.invoke(initdb)
        self.assertIn('Initialized database.', result.output)

    # 测试生成管理员账户
    def test_admin_command(self):
        db.drop_all()
        db.create_all()
        result = self.runner.invoke(args=['admin', '--username',
                                         'grey', '--password', '123'])
        self.assertIn('Creating user...', result.output)
        self.assertIn('Done.', result.output)
        self.assertEqual(User.query.count(), 1)
        self.assertEqual(User.query.first().username, 'grey')
```

```

        self.assertTrue(User.query.first().validate_password('12
3'))

# 测试更新管理员账户
def test_admin_command_update(self):
    # 使用 args 参数给出完整的命令参数列表
    result = self.runner.invoke(args=['admin', '--username',
'peter', '--password', '456'])
    self.assertIn('Updating user...', result.output)
    self.assertIn('Done.', result.output)
    self.assertEqual(User.query.count(), 1)
    self.assertEqual(User.query.first().username, 'peter')
    self.assertTrue(User.query.first().validate_password('45
6'))

```

在这几个测试中，大部分的断言是在检查执行命令后的数据库数据是否发生了正确的变化，或是判断命令行输出（`result.output`）是否包含预期的字符。

运行测试

最后，我们在程序结尾添加下面的代码：

```

if __name__ == '__main__':
    unittest.main()

```

使用下面的命令执行测试：

```
(env) $ python test_watchlist.py
```

```
.....
```

```
-----
```

```
-----
```

```
Ran 15 tests in 2.942s
```

```
OK
```

如果测试出错，你会看到详细的错误信息，进而可以有针对性的修复对应的程序代码，或是调整测试方法。

测试覆盖率

为了让程序更加强壮，你可以添加更多、更完善的测试。那么，如何才能知道程序里有哪些代码还没有被测试？整体的测试覆盖率情况如何？我们可以使用 [Coverage.py](#) 来检查测试覆盖率，首先安装它：

```
(env) $ pip install coverage
```

使用下面的命令执行测试并检查测试覆盖率：

```
(env) $ coverage run --source=app test_watchlist.py
```

因为我们只需要检查程序脚本 `app.py` 的测试覆盖率，所以使用 `--source` 选项来指定要检查的模块或包。

最后使用下面的命令查看覆盖率报告：

```
$ coverage report
Name      Stmts   Miss  Cover
-----
app.py      146      5    97%
```

从上面的表格可以看出，一共有 146 行代码，没测试到的代码有 5 行，测试覆盖率为 97%。

你还可以使用 `coverage html` 命令获取详细的 HTML 格式的覆盖率报告，它会在当前目录生成一个 `htmlcov` 文件夹，打开其中的 `index.html` 即可查看覆盖率报告。点击文件名可以看到具体的代码覆盖情况，如下图所示：

```

1 # -*- coding: utf-8 -*-
2 import os
3 import sys
4
5 import click
6 from flask import Flask, render_template, request, url_for, redirect, flash
7 from flask_sqlalchemy import SQLAlchemy
8 from flask_login import LoginManager, login_user, login_required, logout_user, current_user, UserMixin
9 from werkzeug.security import generate_password_hash, check_password_hash
10
11 # SQLite URI compatible
12 WIN = sys.platform.startswith('win')
13 if WIN:
14     prefix = 'sqlite:///'
15 else:
16     prefix = 'sqlite:///'
17
18 app = Flask(__name__)
19 app.config['SECRET_KEY'] = 'dev'
20 app.config['SQLALCHEMY_DATABASE_URI'] = prefix + os.path.join(app.root_path, 'data.db')
21 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
22
23 db = SQLAlchemy(app)
24 login_manager = LoginManager(app)
25
26
27 @login_manager.user_loader
28 def load_user(user_id):
29     user = User.query.get(int(user_id))
30     return user
31
32

```

同时在 `.gitignore` 文件后追加下面两行，忽略掉生成的覆盖率报告文件：

```

htmlcov/
.coverage

```

本章小结

通过测试后，我们就可以准备上线程序了。结束前，让我们提交代码：

```

$ git add .
$ git commit -m "Add unit test with unittest"
$ git push

```

提示 你可以在 GitHub 上查看本书示例程序的对应 commit：[66dc487](#)。

进阶提示

- 访问 Coverage.py 文档（<https://coverage.readthedocs.io>）或执行 coverage help 命令来查看更多用法。
- 使用标准库中的 unittest 编写单元测试并不是唯一选择，你也可以使用第三方测试框架，比如非常流行的 pytest。
- 如果你是《Flask Web 开发实战》的读者，第 12 章详细介绍了测试 Flask 程序的相关知识，包括使用 Selenium 编写用户界面测试，使用 Flake8 检查代码质量等。

第 10 章：组织你的代码

虽然我们的程序开发已经完成，但随着功能的增多，把所有代码放在 `app.py` 里会让后续的开发和维护变得麻烦。这一章，我们要对项目代码进行一次重构，让项目组织变得更加合理。

`Flask` 对项目结构没有固定要求，你可以使用单脚本，也可以使用包。这一章我们会学习使用包来组织程序。

先来看看我们目前的项目文件结构：

```
└── .flaskenv
└── app.py
└── test_watchlist.py
└── static
    ├── favicon.ico
    ├── images
    │   ├── avatar.png
    │   └── totoro.gif
    └── style.css
└── templates
    ├── 400.html
    ├── 404.html
    ├── 500.html
    ├── base.html
    ├── edit.html
    ├── index.html
    ├── login.html
    └── settings.html
```

使用包组织代码

我们会创建一个包，然后把 `app.py` 中的代码按照类别分别放到多个模块里。下面是我们需要执行的一系列操作（这些操作你也可以使用文件管理器或编辑器完成）：

```
$ mkdir watchlist # 创建作为包的文件夹
$ mv static templates watchlist # 把 static 和 templates 文件夹移动到 watchlist 文件夹内
$ cd watchlist # 切换进包目录
$ touch __init__.py views.py errors.py models.py commands.py # 创建多个模块
```

我们把这个包称为程序包，包里目前包含的模块和作用如下表所示：

模块	作用
<code>__init__.py</code>	包构造文件，创建程序实例
<code>views.py</code>	视图函数
<code>errors.py</code>	错误处理函数
<code>models.py</code>	模型类
<code>commands.py</code>	命令函数

提示 除了包构造文件外，其他的模块文件名你可以自由修改，比如 `views.py` 也可以叫 `routes.py`。

创建程序实例，初始化扩展的代码放到包构造文件里（`__init__.py`），如下所示：

```
import os
import sys

from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager

# ...

app = Flask(__name__)
app.config['SECRET_KEY'] = 'dev'
# 注意更新这里的路径，把 app.root_path 添加到 os.path.dirname() 中
# 以便把文件定位到项目根目录
app.config['SQLALCHEMY_DATABASE_URI'] = prefix + os.path.join(os
    .path.dirname(app.root_path), 'data.db')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy(app)
login_manager = LoginManager(app)

@login_manager.user_loader
def load_user(user_id):
    from watchlist.models import User
    user = User.query.get(int(user_id))
    return user

login_manager.login_view = 'login'

@app.context_processor
def inject_user():
    from watchlist.models import User
    user = User.query.first()
    return dict(user=user)

from watchlist import views, errors, commands
```

在构造文件中，为了让视图函数、错误处理函数和命令函数注册到程序实例上，我们需要在这里导入这几个模块。但是因为这几个模块同时也要导入构造文件中的程序实例，为了避免循环依赖（A 导入 B，B 导入 A），我们把这一行导入语句放到

构造文件的结尾。同样的，`load_user()` 函数和 `inject_user()` 函数中使用的模型类也在函数内进行导入。

其他代码则按照分类分别放到各自的模块中，这里不再给出具体代码，你可以参考[源码仓库](#)。在移动代码之后，注意添加并更新导入语句，比如使用下面的导入语句来导入程序实例和扩展对象：

```
from watchlist import app, db
```

使用下面的导入语句来导入模型类：

```
from watchlist.models import User, Movie
```

以此类推。

组织模板

模块文件夹 `templates` 下包含了多个模板文件，我们可以创建子文件夹来更好的组织它们。下面的操作创建了一个 `errors` 子文件夹，并把错误页面模板都移动到这个 `errors` 文件夹内（这些操作你也可以使用文件管理器或编辑器完成）：

```
$ cd templates # 切换到 templates 目录
$ mkdir errors # 创建 errors 文件夹
$ mv 400.html 404.html 500.html errors # 移动错误页面模板到 errors
文件夹
```

因为错误页面放到了新的路径，所以我们需要修改代码中的 3 处模板文件路径，以 `404` 错误处理函数为例：

```
@app.errorhandler(400)
def bad_request(e):
    return render_template('errors/400.html'), 400
```

单元测试

你也可以将测试文件拆分成多个模块，创建一个 `tests` 包来存储这些模块。但是因为目前的测试代码还比较少，暂时不做改动，只需要更新导入语句即可：

```
from watchlist import app, db
from watchlist.models import Movie, User
from watchlist.commands import forge, initdb
```

因为要测试的目标改变，测试时的 `--source` 选项的值也要更新为包的名称

`watchlist`：

```
(env) $ coverage run --source=watchlist test_watchlist.py
```

提示 你可以创建配置文件来预先定义 `--source` 选项，避免每次执行命令都给出这个选项，具体可以参考[文档配置文件章节](#)。

现在的测试覆盖率报告会显示包内的多个文件的覆盖率情况：

Name	Stmts	Miss	Cover
watchlist__init__.py	25	1	96%
watchlist\commands.py	35	1	97%
watchlist\errors.py	8	2	75%
watchlist\models.py	16	0	100%
watchlist\views.py	77	2	97%
TOTAL	161	6	96%

启动程序

因为我们使用包来组织程序，不再是 Flask 默认识别的 `app.py`，所以在启动开发服务器前需要使用环境变量 `FLASK_APP` 来给出程序实例所在的模块路径。因为我们的程序实例在包构造文件内，所以直接写出包名称即可。在 `.flaskenv` 文件中添加下面这行代码：

```
FLASK_APP=watchlist
```

最终的项目文件结构如下所示：

```
└── .flaskenv
└── test_watchlist.py
└── watchlist # 程序包
    ├── __init__.py
    ├── commands.py
    ├── errors.py
    ├── models.py
    ├── views.py
    ├── static
    │   ├── favicon.ico
    │   ├── images
    │   │   ├── avatar.png
    │   │   └── totoro.gif
    │   └── style.css
    └── templates
        ├── base.html
        ├── edit.html
        ├── errors
        │   ├── 400.html
        │   ├── 404.html
        │   └── 500.html
        ├── index.html
        ├── login.html
        └── settings.html
```

本章小结

对我们的程序来说，这样的项目结构已经足够了。但对于大型项目，你可以使用蓝本和工厂函数来进一步组织程序。结束前，让我们提交代码：

```
$ git add .
$ git commit -m "Organize application with package"
$ git push
```

提示 你可以在 GitHub 上查看本书示例程序的对应 commit：[f705408](#)。

进阶提示

- 蓝本类似于子程序的概念，借助蓝本你可以把程序不同部分的代码分离开（比如按照功能划分为用户认证、管理后台等多个部分），即对程序进行模块化处理。每个蓝本可以拥有独立的子域名、URL 前缀、错误处理函数、模板和静态文件。
- 工厂函数就是创建程序的函数。在工厂函数内，我们先创建程序实例，并在函数内完成初始化扩展、注册视图函数等一系列操作，最后返回可以直接运行的程序实例。工厂函数可以接受配置名称作为参数，在内部加载对应的配置文件，这样就可以实现按需创建加载不同配置的程序实例，比如在测试时调用工厂函数创建一个测试用的程序实例。
- 如果你是《Flask Web 开发实战》的读者，第 7 章介绍了使用包组织程序，第 8 章介绍了大型项目结构以及如何使用蓝本和工厂函数组织程序。

第 11 章：部署上线

在这个教程的最后一章，我们将会把程序部署到互联网上，让网络中的其他所有人都可以访问到。

Web 程序通常有两种部署方式：传统部署和云部署。传统部署指的是在使用物理主机或虚拟主机上部署程序，你通常需要在一个 Linux 系统上完成所有的部署操作；云部署则是使用其他公司提供的云平台，这些平台为你设置好了底层服务，包括 Web 服务器、数据库等等，你只需要上传代码并进行一些简单设置即可完成部署。这一章我们会介绍使用云平台 [PythonAnywhere](#) 来部署程序。

部署前的准备

首先，我们需要生成一个依赖列表，方便在部署环境里安装。使用下面的命令把当前依赖列表写到一个 requirements.txt 文件里：

```
(env) $ pip freeze > requirements.txt
```

对于某些配置，生产环境下需要使用不同的值。为了让配置更加灵活，我们把需要在生产环境下使用的配置改为优先从环境变量中读取，如果没有读取到，则使用默认值：

```
app.config['SECRET_KEY'] = os.getenv('SECRET_KEY', 'dev')
app.config['SQLALCHEMY_DATABASE_URI'] = prefix + os.path.join(os
    .path.dirname(app.root_path), os.getenv('DATABASE_FILE', 'data.d
    b'))
```

以第一个配置变量为例，`os.getenv('SECRET_KEY', 'dev')` 表示读取系统环境变量 `SECRET_KEY` 的值，如果没有获取到，则使用 `dev`。

注意 像密钥这种敏感信息，保存到环境变量中要比直接写在代码中更加安全。

对于第二个配置变量，我们仅改动了最后的数据库文件名。在示例程序里，因为我们部署后将继续使用 SQLite，所以只需要为生产环境设置不同的数据库文件名，否则的话，你可以像密钥一样设置优先从环境变量读取整个数据库 URL。

在部署程序时，我们不会使用 Flask 内置的开发服务器运行程序，因此，对于写到 .env 文件的环境变量，我们需要手动使用 `python-dotenv` 导入。下面在项目根目录创建一个 `wsgi.py` 脚本，在这个脚本中加载环境变量，并导入程序实例以供部署时使用：

`wsgi.py`：手动设置环境变量并导入程序实例

```
import os

from dotenv import load_dotenv

dotenv_path = os.path.join(os.path.dirname(__file__), '.env')
if os.path.exists(dotenv_path):
    load_dotenv(dotenv_path)

from watchlist import app
```

这两个环境变量的具体定义，我们将在远程服务器环境创建新的 `.env` 文件写入。

最后让我们把改动提交到 Git 仓库，并推送到 GitHub 上的远程仓库：

```
$ git add .
$ git commit -m "Ready to deploy"
$ git push
```

提示 你可以在 GitHub 上查看本书示例程序的对应 commit：[92eabc8](#)。

使用 PythonAnywhere 部署程序

首先访问[注册页面](#)注册一个免费账户。注册时填入的用户名将作为你的程序域名的子域部分，以及分配给你的 Linux 用户名。比如，如果你的用户名为 `greyli`，最终为你分配的程序域名就是 <http://greyli.pythonanywhere.com/>。

注册完成后会有一个简单的教程，你可以跳过，也可以跟着了解一下基本用法。管理面板主页如下所示：

The screenshot shows the PythonAnywhere dashboard. At the top, there's a logo and navigation links: Dashboard, Consoles, Files, Web, Tasks, Databases. A welcome message "Welcome back, greyl" is on the right. Below, CPU and File storage usage are shown. The main area has four sections: "Recent Consoles" (empty), "Recent Files" (empty), "Recent Notebooks" (empty, with a note about Jupyter support), and "All Web apps" (empty). Buttons for "Open another file", "Browse files", "Open Web tab", and "More..." are visible.

导航栏包含几个常用的链接，可以打开其他面板：

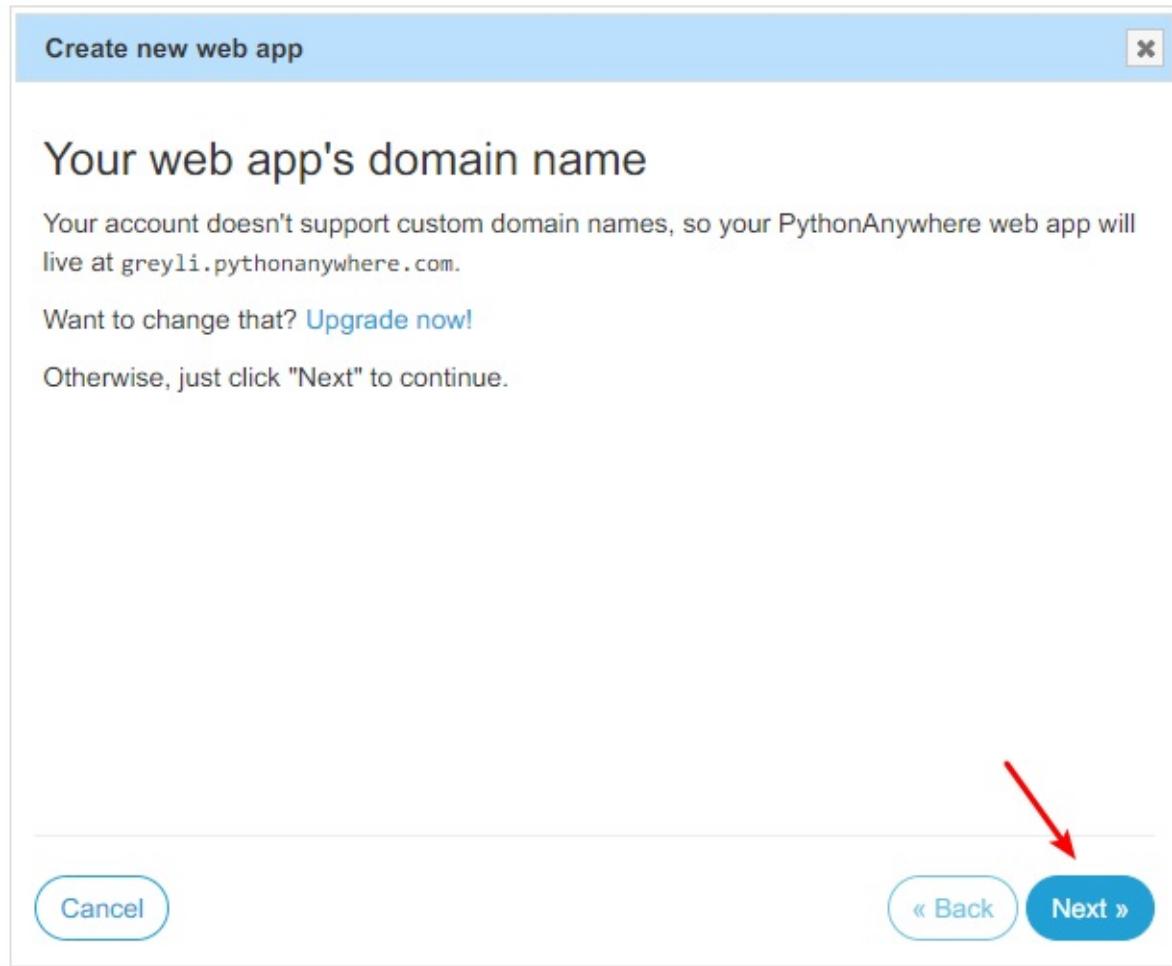
- **Consoles**（控制台）：可以打开 Bash、Python Shell、MySQL 等常用的控制台
- **Files**（文件）：创建、删除、编辑、上传文件，你可以在这里直接修改代码
- **Web**：管理 Web 程序
- **Tasks**（任务）：创建计划任务
- **Databases**（数据库）：设置数据库，免费账户可以使用 MySQL

这些链接对应页面的某些功能也可以直接在管理面板主页打开。

我们需要先来创建一个 Web 程序，你可以点击导航栏的 Web 链接，或是主页上的“Open Web tab”按钮打开 Web 面板：

The screenshot shows the "Web" panel. It features a large blue button "Add a new web app" with a red arrow pointing to it from the left. To the right, a message says "You have no web apps" and provides instructions: "To create a PythonAnywhere-hosted web app, click the 'Add a new web app' button to the left."

点击“Add a new web app”按钮创建 Web 程序，第一步提示升级账户后可以自定义域名，我们直接点击“Next”按钮跳到下一步：



这一步选择 Web 框架，为了获得更灵活的控制，选择手动设置（Manual configuration）：

Create new web app ×

Select a Python Web framework

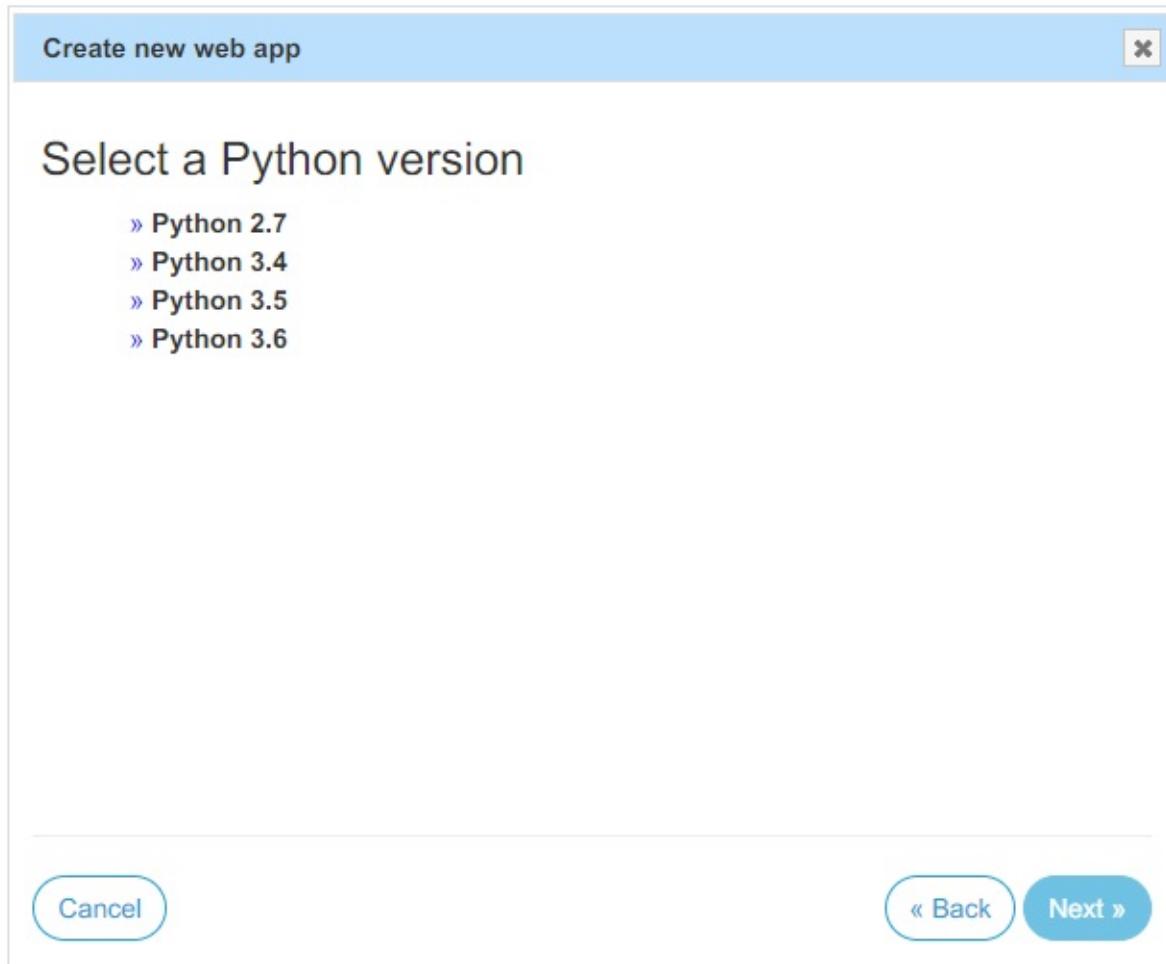
...or select "Manual configuration" if you want detailed control.

- » Django
- » web2py
- » Flask
- » Bottle
- » **Manual configuration (including virtualenvs)**

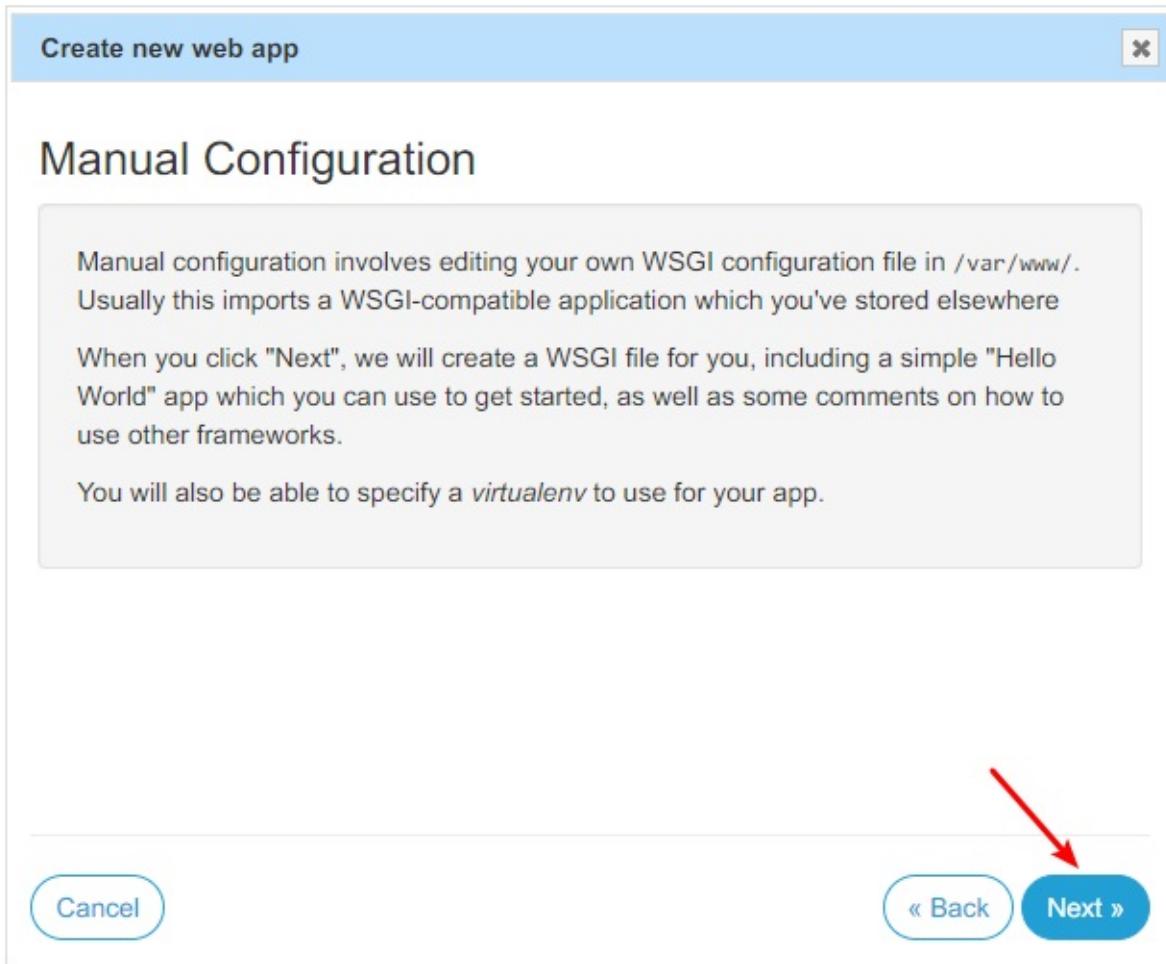
What other frameworks should we have here? Send us some feedback using the link at the top of the page!

[Cancel](#) [« Back](#) [Next »](#)

接着选择你想使用的 Python 版本：



最后点击“Next”按钮即可完成创建 Web 程序流程：



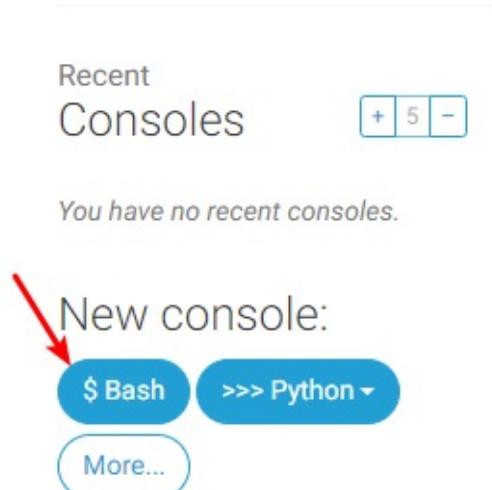
接下来我们需要进行一系列程序初始化操作，最后再回到 Web 面板进行具体的设置。

初始化程序运行环境

我们首先要考虑把代码上传到 PythonAnywhere 的服务器上。上传代码一般有两种方式：

- 从 GitHub 拉取我们的程序
- 在本地将代码存储为压缩文件，然后在 Files 标签页上传压缩包

因为我们的代码已经推送到 GitHub 上，这里将采用第一种方式。首先通过管理面板主页的“Bash”按钮或是 Consoles 面板下的“Bash”链接创建一个命令行会话：



在命令行下输入下面的命令：

```
$ git clone https://github.com/greyli/watchlist # 注意替换 Git 仓库地址
$ cd watchlist # 切换进程序仓库
```

这会把程序代码克隆到 PythonAnywhere 为你分配的用户目录中，路径即 `/home/你的 PythonAnywhere 用户名/你的仓库名称`，比如 `/home/greyli/watchlist`。

注意替换 `git clone` 命令后的 Git 地址，将 `greyli` 替换为你的 GitHub 用户名，将 `watchlist` 替换为你的仓库名称。

提示 如果你在 GitHub 上的仓库类型为私有仓库，那么需要将 PythonAnywhere 服务器的 SSH 密钥添加到 GitHub 账户中，具体参考第 1 章“设置 SSH 密钥”小节。

下面我们在项目根目录创建 `.env` 文件，并写入生产环境下需要设置的两个环境变量。其中，密钥（`SECRET_KEY`）的值是随机字符串，我们可以使用 `uuid` 模块来生成：

```
$ python3
>>> import uuid
>>> uuid.uuid4().hex
'3d6f45a5fc12445dbac2f59c3b6c7cb1'
```

复制生成的随机字符备用，接着创建 .env 文件：

```
$ nano .env
```

写入设置密钥和数据库名称的环境变量：

```
SECRET_KEY=3d6f45a5fc12445dbac2f59c3b6c7cb1  
DATABASE_FILE=data-prod.db
```

最后安装依赖并执行初始化操作：

```
$ python3 -m venv env # 创建虚拟环境  
$ . env/bin/activate # 激活虚拟环境  
(env) $ pip install -r requirements.txt # 安装所有依赖  
(env) $ flask initdb # 初始化数据库  
(env) $ flask admin # 创建管理员账户
```

先不要关闭这个标签页，后面我们还要在这里执行一些命令。点击右上角的菜单按钮，并在浏览器的新标签页打开 Web 面板。

设置并启动程序

代码部分我们已经设置完毕，接下来进行一些简单设置就可以启动程序了。

代码

回到 Web 标签页，先来设置 Code 部分的配置：

Code:

What your site is running.

Source code:	/home/greyli/watchlist/
Working directory:	/home/greyli/watchlist/
WSGI configuration file:	/var/www/greyli_pythonanywhere_com_wsgi.py
Python version:	3.6

点击源码（Source code）和工作目录（Working directory）后的路径并填入项目根目录，目录规则为“/home/用户名/项目文件夹名”。

点击 WSGI 配置文件（WSGI configuration file）后的链接打开编辑页面，删掉这个文件内的所有内容，填入下面的代码：

```
import sys

path = '/home/greyli/watchlist' # 路径规则为 /home/你的用户名/项目
文件夹名
if path not in sys.path:
    sys.path.append(path)

from wsgi import app as application
```

完成后点击绿色的 Save 按钮或按下 Ctrl+S 保存修改，点击右上角的菜单按钮返回 Web 面板。

PythonAnywhere 会自动从这个文件里导入名称为 `application` 的程序实例，所以我们从项目目录的 `wsgi` 模块中导入程序实例 `app`，并将名称映射为 `application`。

虚拟环境

为了让程序正确运行，我们需要在 Virtualenv 部分填入虚拟环境文件夹的路径：

Virtualenv:

Use a virtualenv to get different versions of flask, django etc from our default system ones. [More info here](#). You need to Reload your web app to activate it; NB - will do nothing if the virtualenv does not exist.

[Enter path to a virtualenv, if desired](#)

对应我们的项目就是 `/home/greyli/watchlist/env/`，注意替换其中的用户名、项目名称和虚拟环境名称部分。点击 Virtualenv 部分的红色字体链接，填入并保存。

静态文件

静态文件可以交给 PythonAnywhere 设置的服务器来处理，这样会更高效。要让 PythonAnywhere 处理静态文件，我们只需要在 **Static files** 部分指定静态文件 URL 和对应的静态文件文件夹目录，如下所示：

Static files:

Files that aren't dynamically generated by your code, like CSS, JavaScript or uploaded files, can be served much faster straight off the disk if you specify them here. You need to **Reload your web app** to activate any changes you make to the mappings below.

URL	Directory	Delete
/static/	/home/greyli/watchlist/watchlist/static/	
<i>Enter URL</i>	<i>Enter path</i>	

注意更新目录中的用户名和项目文件夹名称。

启动程序

一切就绪，点击绿色的重载按钮即可让配置生效：

Configuration for greyli.pythonanywhere.com

Reload:

 Reload greyli.pythonanywhere.com



现在访问你的程序网址“<https://用户名.pythonanywhere.com>”（Web 面板顶部的链接），比如<https://greyli.pythonanywhere.com> 即可访问程序。

最后还要注意的是，免费账户需要每三个月点击一次黄色的激活按钮（在过期前你会收到提醒邮件）：

Best before date:

We're happy to host your free website – and keep it free – for as long as you want to keep it running, but you'll need to log in at least once every three months and click the "Run until 3 months from today" button below. We'll send you an email a week before the site is disabled so that you don't forget to do that. [See here for more details.](#)

This site will be disabled on **Tuesday 30 April 2019**

Run until 3 months from today

Paying users' sites stay up forever without any need to log in to keep them running.



更新部署后的程序

当你需要更新程序时，流程和部署类似。在本地完成更新，确保程序通过测试后，将代码推送到 GitHub 上的远程仓库。登录到 PythonAnywhere，打开一个命令行会话（Bash），切换到项目目录，使用 git pull 命令从远程仓库拉取更新：

```
$ cd watchlist  
$ git pull
```

然后你可以执行一些必要的操作，比如安装新的依赖等等。最后在 Web 面板点击绿色的重载（Reload）按钮即可完成更新。

本章小结

程序部署上线以后，你可以考虑继续为它开发新功能，也可以从零编写一个新的程序。虽然本书即将接近尾声，但你的学习之路才刚刚开始，因为本书只是介绍了 Flask 入门所需的基础知识，你还需要进一步学习。在后记中，你可以看到进一步学习的推荐读物。

接下来，有一个挑战在等着你。

进阶提示

- 因为 PythonAnywhere 支持在线管理文件、编辑代码、执行命令，你可以在学习编程的过程中使用它来在线开发 Web 程序。

- PythonAnywhere 的 Web 面板还有一些功能设置：`Log files` 部分可以查看你的程序日志，`Traffic` 部分显示了你的程序访问流量情况，`Security` 部分可以为你的程序程序开启强制启用 HTTPS 和密码保护。
- 如果你是《Flask Web 开发实战》的读者，第 14 章详细介绍了部署 Flask 程序的两种方式，传统部署和云部署。

小挑战

经过本书的学习，你应该有能力独立开发一个简单的 Web 程序了。所以这里有一个小挑战：为你的 Watchlist 添加一个留言板功能，效果类似 [SayHello](#)。

下面是一些编写提示：

- 编写表示留言的模型类，更新数据库表
- 创建留言页面的模板
- 在模板中添加留言表单
- 添加显示留言页面的视图函数
- 在显示留言页面的视图函数编写处理表单的代码
- 生成一些虚拟数据进行测试
- 编写单元测试
- 更新到部署后的程序
- 可以参考 [SayHello 源码](#)

如果在完成这个挑战的过程中遇到了困难，可以在 [HelloFlask 论坛](#) 发起讨论（设置帖子分类为“Flask 入门教程”）。除此之外，你可以在后记查看更多讨论的去处。

后记

恭喜，你已经完成了整个 Flask 入门教程。不出意外的话，你也编写了你的第一个 Web 程序，并把它部署到了互联网上。这是一件值得纪念的事，它可以作为你的编程学习路上的一个小小的里程碑。继续加油！

留言 & 打卡

如果你完成了这个教程，可以在 HelloFlask 论坛上的[这个帖子](#)留言打卡，欢迎分享你的心得体会和经验总结。如果你对这本书有什么建议，也可以在这里进行留言反馈。

进阶阅读

说来惭愧，在这本教程几乎每一章的结尾，我都会提到《[Flask Web 开发实战](#)》，每次写到这里，我都觉得自己好像在写“问候家明”。所以，最合适的进阶读物我已经推荐过很多次了。除了这本书，其他的进阶读物如下：

- [Flask 官方文档](#)
- [Flask Mega-Tutorial](#)
- [知乎专栏 Hello, Flask!](#)

未完待续

你喜欢这本书以及这本书的写作模式吗？

如果有足够的人喜欢的话，或许我会考虑写一本包含 Flask 进阶知识的《[Flask 进阶教程](#)》。按照设想，在这个进阶教程里，这个 Watchlist 程序变成一个支持多人注册和使用的简化版豆瓣电影 / [IMDb](#)。同时介绍各类常用扩展的使用和 Flask 上下文、蓝本等进阶知识。

再或者，这个教程或许会升级为一本完整的书，使用类似的编写模式，引入一个更加丰富有趣的程序，包含优化后的入门知识和 Flask 进阶内容。

如果你期待这样一本进阶教程 / 书的出现，欢迎让我知道。你可以在[打卡 & 留言贴](#)发布留言，或是直接发邮件（withlihui@gmail.com）告诉我。