

提示 如果执行 flask run 命令后显示命令未找到提示 (command not found) 或其他错误，可以尝试使用 python -m flask run 启动服务器，其他命令亦同。

flask run 命令运行的开发服务器默认会监听 `http://127.0.0.1:5000/` 地址（按 Crtl+C 退出），并开启多线程支持。当我们打开浏览器访问这个地址时，会看到网页上显示“Hello, World!”，如图 1-5 所示。

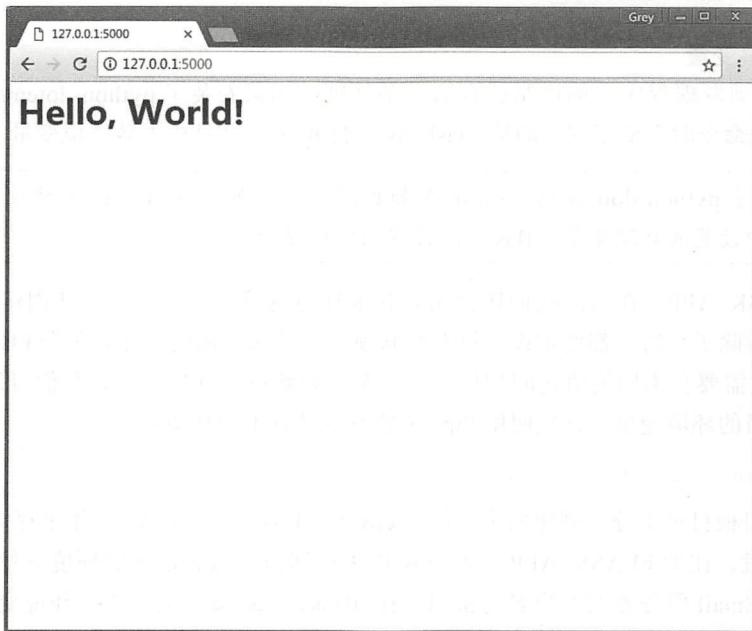


图 1-5 “Hello, World!” 程序主页

提示 `http://127.0.0.1` 即 `localhost`，是指向本地机的 IP 地址，一般用来测试。Flask 默认使用 5000 端口，对于上面的地址，你也可以使用 `http://localhost:5000/`。在本书中这两者会交替使用，除了地址不同外，两者没有实际区别，即域名和 IP 地址的映射关系。

提示 旧的启动开发服务器的方式是使用 `app.run()` 方法，目前已不推荐使用 (deprecated)。

1. 自动发现程序实例

一般来说，在执行 flask run 命令运行程序前，我们需要提供程序实例所在模块的位置。我们在上面可以直接运行程序，是因为 Flask 会自动探测程序实例，自动探测存在下面这些规则：

- 从当前目录寻找 `app.py` 和 `wsgi.py` 模块，并从中寻找名为 `app` 或 `application` 的程序实例。
- 从环境变量 `FLASK_APP` 对应的值寻找名为 `app` 或 `application` 的程序实例。

因为我们的程序主模块命名为 app.py，所以 flask run 命令会自动在其中寻找程序实例。如果你的程序主模块是其他名称，比如 hello.py，那么需要设置环境变量 FLASK_APP，将包含程序实例的模块名赋值给这个变量。Linux 或 macOS 系统使用 export 命令：

```
$ export FLASK_APP=hello
```

在 Windows 系统中使用 set 命令：

```
> set FLASK_APP=hello
```

2. 管理环境变量

Flask 的自动发现程序实例机制还有第三条规则：如果安装了 python-dotenv，那么在使用 flask run 或其他命令时会使用它自动从 .flaskenv 文件和 .env 文件中加载环境变量。

 **附注** 当安装了 python-dotenv 时，Flask 在加载环境变量的优先级是：手动设置的环境变量 >.env 中设置的环境变量 >.flaskenv 设置的环境变量。

除了 FLASK_APP，在后面我们还会用到其他环境变量。环境变量在新创建命令行窗口或重启电脑后就清除了，每次都要重设变量有些麻烦。而且如果你同时开发多个 Flask 程序，这个 FLASK_APP 就需要在不同的值之间切换。为了避免频繁设置环境变量，我们可以使用 python-dotenv 管理项目的环境变量，首先使用 Pipenv 将它安装到虚拟环境：

```
$ pipenv install python-dotenv
```

我们在项目根目录下分别创建两个文件：.env 和 .flaskenv。.flaskenv 用来存储和 Flask 相关的公开环境变量，比如 FLASK_APP；而 .env 用来存储包含敏感信息的环境变量，比如后面我们会用来配置 Email 服务器的账户名与密码。在 .flaskenv 或 .env 文件中，环境变量使用键值对的形式定义，每行一个，以 # 开头的为注释，如下所示：

```
SOME_VAR=1
# 这是注释
FOO="BAR"
```

 **注意** .env 包含敏感信息，除非是私有项目，否则绝对不能提交到 Git 仓库中。当你开发一个新项目时，记得把它的名称添加到 .gitignore 文件中，这会告诉 Git 忽略这个文件。.gitignore 文件是一个名为 .gitignore 的文本文件，它存储了项目中 Git 提交时的忽略文件规则清单。Python 项目的 .gitignore 模板可以参考 <https://github.com/github/gitignore/blob/master/Python.gitignore>。使用 PyCharm 编写程序时会产生一些配置文件，这些文件保存在项目根目录下的 .idea 目录下，关于这些文件的忽略设置可以参考 <https://www.gitignore.io/api/pycharm>。

3. 使用 PyCharm 运行服务器

在 PyCharm 中，虽然我们可以使用内置的命令行窗口执行命令以启动开发服务器，但是在开发时使用 PyCharm 内置的运行功能更加方便。在 2018.1 版本后的专业版添加了 Flask 命令行

支持，在旧版本或社区版中，如果要使用 PyCharm 运行程序，还需要进行一些设置。

首先，在 PyCharm 中，单击菜单栏中的 Run→Edit Configurations 打开运行配置窗口。图 1-6 中标出了在 PyCharm 中设置一个运行配置的具体步骤序号。

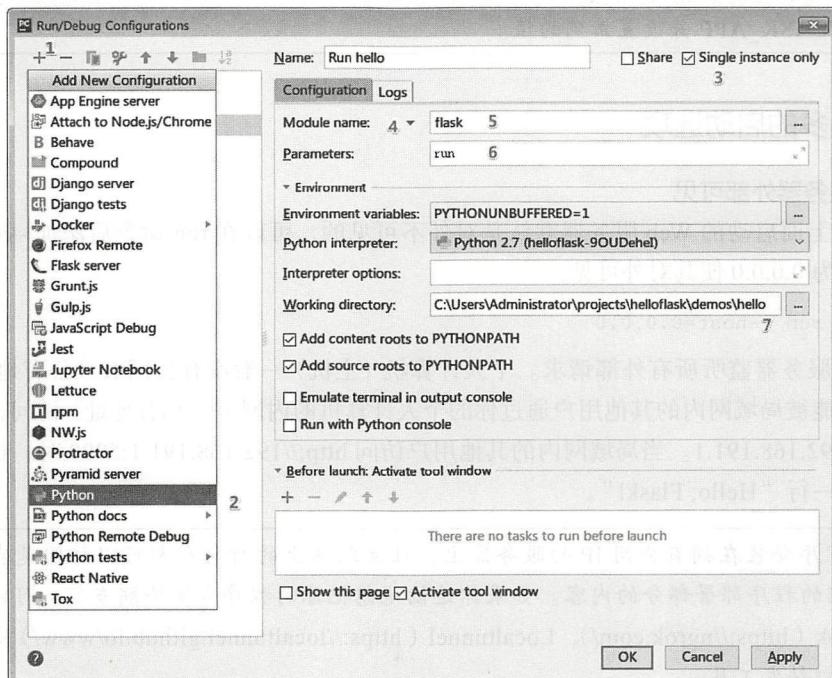


图 1-6 运行 Flask 程序的配置

打开新建配置窗口后，具体的步骤如下所示：

步骤 1 单击左侧的“+”符号打开下拉列表。

步骤 2 新建一个 Python 类型的运行配置（如果你使用的是专业版，则可以直接选择 Flask server），并在右侧的 Name 字段输入一个合适的名称，比如“Run hello”。

步骤 3 勾选“Single instance only”。

步骤 4 将第一项配置字段通过下列选项选为“Module Name”。

步骤 5 填入模块名称 flask。

步骤 6 第二栏的“Parameters”填入要执行的命令 run，你也可以附加其他启动选项。

步骤 7 在“Working directory”字段中选择程序所在的目录作为工作目录。



我们可以单击左上方的复制图标复制一份配置，然后稍加修改就可以用于其他 flask 命令，包括扩展提供的命令，或是我们自定义的命令。

现在单击 Apply 或 OK 保存并关闭窗口。在 PyCharm 右上方选择我们创建的运行配置，然后单击绿色三角形的运行按钮即可启动开发服务器。

 **注 意** 因为本章示例程序的模块名称为 app.py，Flask 会自动从中寻找程序实例，所以我们在 PyCharm 中的运行设置可以正确启动程序。如果你不打算使用 python-dotenv 来管理环境变量，那么需要修改 PyCharm 的运行配置：在 Environment variable 字段中添加环境变量 FLASK_APP 并设置正确的值。

1.3.2 更多的启动选项

1. 使服务器外部可见

我们在上面启动的 Web 服务器默认是对外不可见的，可以在 run 命令后添加 --host 选项将主机地址设为 0.0.0.0 使其对外可见：

```
$ flask run --host=0.0.0.0
```

这会让服务器监听所有外部请求。个人计算机（主机）一般没有公网 IP（公有地址），所以你的程序只能被局域网内的其他用户通过你的个人计算机的内网 IP（私有地址）访问，比如你的内网 IP 为 192.168.191.1。当局域网内的其他用户访问 <http://192.168.191.1:5000> 时，也会看到浏览器里显示一行“Hello, Flask!”。

 **提 示** 把程序安装在拥有公网 IP 的服务器上，让互联网上的所有人都可以访问是我们最后要介绍的程序部署部分的内容。如果你迫切地想把你的程序分享给朋友们，可以考虑使用 ngrok (<https://ngrok.com/>)、Localtunnel (<https://localtunnel.github.io/www/>) 等内网穿透 / 端口转发工具。

2. 改变默认端口

Flask 提供的 Web 服务器默认监听 5000 端口，你可以在启动时传入参数来改变它：

```
$ flask run --port=8000
```

这时服务器会监听来自 8000 端口的请求，程序的主页地址也相应变成了 <http://localhost:8000/>。

 **附 注** 执行 flask run 命令时的 host 和 port 选项也可以通过环境变量 FLASK_RUN_HOST 和 FLASK_RUN_PORT 设置。事实上，Flask 内置的命令都可以使用这种模式定义默认选项值，即“FLASK_<COMMAND>_<OPTION>”，你可以使用 flask --help 命令查看所有可用的命令。

1.3.3 设置运行环境

开发环境（development environment）和生产环境（production environment）是我们后面会频繁接触到的概念。开发环境是指我们在本地编写和测试程序时的计算机环境，而生产环境与开发环境相对，它指的是网站部署上线供用户访问时的服务器环境。

根据运行环境的不同，Flask 程序、扩展以及其他程序会改变相应的行为和设置。为了区分程序运行环境，Flask 提供了一个 FLASK_ENV 环境变量用来设置环境，默认为 production（生产）。在开发时，我们可以将其设为 development（开发），这会开启所有支持开发的特性。为了方便管理，我们将把环境变量 FLASK_ENV 的值写入 .flaskenv 文件中：

```
FLASK_ENV=development
```

现在启动程序，你会看到下面的输出提示：

```
$ flask run
 * Environment: development
 * Debug mode: on
 * Debugger is active!
 * Debugger PIN: 202-005-064
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

在开发环境下，调试模式（Debug Mode）将被开启，这时执行 flask run 启动程序会自动激活 Werkzeug 内置的调试器（debugger）和重载器（reloader），它们会为开发带来很大的帮助。



提示 如果你想单独控制调试模式的开关，可以通过 FLASK_DEBUG 环境变量设置，设为 1 则开启，设为 0 则关闭，不过通常不推荐手动设置这个值。



注意 在生产环境中部署程序时，绝不能开启调试模式。尽管 PIN 码可以避免用户任意执行代码，提高攻击者利用调试器的难度，但并不能确保调试器完全安全，会带来巨大的安全隐患。而且攻击者可能会通过调试信息获取你的数据库结构等容易带来安全问题的信息。另一方面，调试界面显示的错误信息也会让普通用户感到困惑。

1. 调试器

Werkzeug 提供的调试器非常强大，当程序出错时，我们可以在网页上看到详细的错误追踪信息，这在调试错误时非常有用。运行中的调试器如图 1-7 所示。

调试器允许你在错误页面上执行 Python 代码。单击错误信息右侧的命令行图标，会弹出窗口要求输入 PIN 码，也就是在启动服务器时命令行窗口打印出的调试器 PIN 码（Debugger PIN）。输入 PIN 码后，我们可以单击错误堆栈的某个节点右侧的命令行界面图标，这会打开一个包含代码执行上下文信息的 Python Shell，我们可以利用它来进行调试。

2. 重载器

当我们对代码做了修改后，期望的行为是这些改动立刻作用到程序上。重载器的作用就是监测文件变动，然后重新启动开发服务器。当我们修改了脚本内容并保存后，会在命令行看到下面的输出：

```
Detected change in '/path/to/app.py', reloading
* Restarting with stat
```

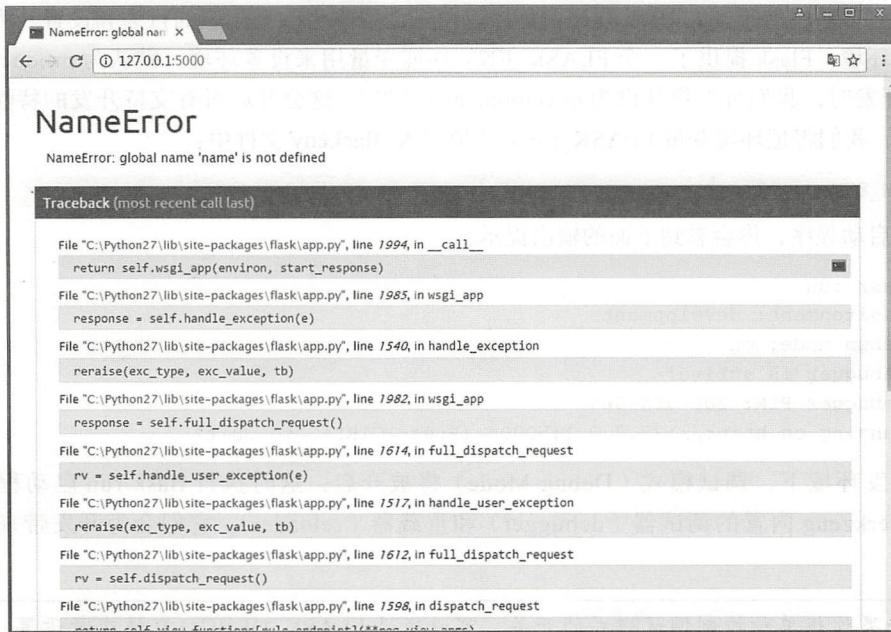


图 1-7 调试器界面

默认会使用 Werkzeug 内置的 stat 重载器，它的缺点是耗电较严重，而且准确性一般。为了获得更优秀的体验，我们可以安装另一个用于监测文件变动的 Python 库 Watchdog，安装后 Werkzeug 会自动使用它来监测文件变动：

```
$ pipenv install watchdog --dev
```

因为这个包只在开发时才会用到，所以我们在安装命令后添加了一个 `--dev` 选项，这用来把这个包声明为开发依赖。在 Pipfile 文件中，这个包会被添加到 `dev-packages` 部分。

不过，如果项目中使用了单独的 CSS 或 JavaScript 文件时，那么浏览器可能会缓存这些文件，从而导致对文件做出的修改不能立刻生效。在浏览器中，我们可以按下 Crtl+F5 或 Shift+F5 执行硬重载（hard reload），即忽略缓存并重载（刷新）页面。

 提示 当在一个新电脑创建运行环境时，使用 `pipenv install` 命令时需要添加额外的 `--dev` 选项 才会安装 `dev-packages` 部分定义的开发依赖包。

1.4 Python Shell

本书有许多操作需要在 Python Shell（即 Python 交互式解释器）里执行。在开发 Flask 程序时，我们并不会直接使用 `python` 命令启动 Python Shell，而是使用 `flask shell` 命令：

```
$ flask shell
App: app [development]
```

```
Instance: Path/to/your/helloflask/instance
>>>
```



注意 和其他 flask 命令相同，执行这个命令前我们要确保程序实例可以被正常找到。

在本书中，如果代码片段前的提示符为三个大于号，即“>>>”，那么就表示这些代码需要在使用 flask shell 命令打开的 Python Shell 中执行。



提示 Python Shell 可以执行 exit() 或 quit() 退出，在 Windows 系统上可以使用 Ctrl+Z 并按 Enter 退出；在 Linux 和 macOS 则可以使用 Ctrl+D 退出。

使用 flask shell 命令打开的 Python Shell 自动包含程序上下文，并且已经导入了 app 实例：

```
>>> app
<Flask 'app'>
>>> app.name
'app'
```



附注 上下文（context）可以理解为环境。为了正常运行程序，一些操作相关的状态和数据需要被临时保存下来，这些状态和数据被统称为上下文。在 Flask 中，上下文有两种，分别为程序上下文和请求上下文，后面我们会详细了解。

1.5 Flask 扩展

在本书中我们将会接触到很多 Flask 扩展。扩展（extension）即使用 Flask 提供的 API 接口编写的 Python 库，可以为 Flask 程序添加各种各样的功能。大部分 Flask 扩展用来集成其他库，作为 Flask 和其他库之间的薄薄一层胶水。因为 Flask 扩展的编写有一些约定，所以初始化的过程大致相似。大部分扩展都会提供一个扩展类，实例化这个类，并传入我们创建的程序实例 app 作为参数，即可完成初始化过程。通常，扩展会在传入的程序实例上注册一些处理函数，并加载一些配置。

以某扩展实现了 Foo 功能为例，这个扩展的名称将是 Flask-Foo 或 Foo-Flask；程序包或模块的命名使用小写加下划线，即 flask_foo（即导入时的名称）；用于初始化的类一般为 Foo，实例化的类实例一般使用小写，即 foo。初始化这个假想中的 Flask-Foo 扩展的示例如下所示：

```
from flask import Flask
from flask_foo import Foo

app = Flask(__name__)
foo = Foo(app)
```

在日常开发中，大多数情况下，我们没有必要重复制造轮子，所以选用扩展可以避免让项目变得臃肿和复杂。尽管使用扩展可以简化操作，快速集成某个功能，但同时也会降低灵活性。如果过度使用扩展，在不需要的地方引入，那么相应也会导致代码不容易维护。更糟糕的是，



质量差的扩展可能还会带来潜在的 Bug，而不同扩展之间也可能会出现冲突。因此，在编写程序时，应该尽量从实际需求出发，只在需要的时候使用扩展，并把扩展的质量和兼容性作为考虑因素，尽量在效率和灵活性之间达到平衡。



附注 早期版本的 Flask 扩展使用 `flaskext.foo` 或 `flask.ext.something` 的形式导入，在实际使用中带来了许多问题，因此 Flask 官方推荐以 `flask_something` 形式导入扩展。在 1.0 版本以后的 Flask 中，旧的扩展导入方式已被移除。

1.6 项目配置

在很多情况下，你需要设置程序的某些行为，这时你就需要使用配置变量。在 Flask 中，配置变量就是一些大写形式的 Python 变量，你也可以称之为配置参数或配置键。使用统一的配置变量可以避免在程序中以硬编码（hard coded）的形式设置程序。

在一个项目中，你会用到许多配置：Flask 提供的配置，扩展提供的配置，还有程序特定的配置。和平时使用变量不同，这些配置变量都通过 Flask 对象的 `app.config` 属性作为统一的接口来设置和获取，它指向的 `Config` 类实际上是字典的子类，所以你可以像操作其他字典一样操作它。



附注 Flask 内置的配置可以访问 Flask 文档的配置章节 (flask.pocoo.org/docs/latest/config/) 查看，扩展提供的配置也可以在对应的文档中查看。

Flask 提供了很多种方式来加载配置。比如，你可以像在字典中添加一个键值对一样来设置一个配置：

```
app.config['ADMIN_NAME'] = 'Peter'
```



注意 配置的名称必须是全大写形式，小写的变量将不会被读取。

使用 `update()` 方法则可以一次加载多个值：

```
app.config.update(  
    TESTING=True,  
    SECRET_KEY='_5#yF4Q8z\\n\\xec'] /'  
)
```

除此之外，你还可以把配置变量存储在单独的 Python 脚本、JSON 格式的文件或是 Python 类中，`config` 对象提供了相应的方法来导入配置，具体我们会在后面了解。

和操作字典一样，读取一个配置就是从 `config` 字典里通过将配置变量的名称作为键读取对应的值：

```
value = app.config['ADMIN_NAME']
```



某些扩展需要读取配置值来完成初始化操作，比如 Flask-Mail，因此我们应该尽量将加载配置的操作提前，最好在程序实例 app 创建后就加载配置。

1.7 URL 与端点

在 Web 程序中，URL 无处不在。如果程序中的 URL 都是以硬编码的方式写出，那么将会大大降低代码的易用性。比如，当你修改了某个路由的 URL 规则，那么程序里对应的 URL 都要一个一个进行修改。更好的解决办法是使用 Flask 提供的 url_for() 函数获取 URL，当路由中定义的 URL 规则被修改时，这个函数总会返回正确的 URL。

调用 url_for() 函数时，第一个参数为端点（endpoint）值。在 Flask 中，端点用来标记一个视图函数以及对应的 URL 规则。端点的默认值为视图函数的名称，至于为什么不直接使用视图函数名，而要引入端点这个概念，我们会在后面了解。

比如，下面的视图函数：

```
@app.route('/')
def index():
    return 'Hello Flask!'
```

这个路由的端点即视图函数的名称 index，调用 url_for('index') 即可获取对应的 URL，即“/”。



在 app.route() 装饰器中使用 endpoint 参数可以自定义端点值，不过我们通常不需要这样做。

如果 URL 含有动态部分，那么我们需要在 url_for() 函数里传入相应的参数，以下面的视图函数为例：

```
@app.route('/hello/<name>')
def greet(name):
    return 'Hello %s!' % name
```

这时使用 url_for('say_hello', name='Jack') 得到的 URL 为“/hello/Jack”。



我们使用 url_for() 函数生成的 URL 是相对 URL（即内部 URL），即 URL 中的 path 部分，比如“/hello”，不包含根 URL。相对 URL 只能在程序内部使用。如果你想要生成供外部使用的绝对 URL，可以在使用 url_for() 函数时，将 _external 参数设为 True，这会生成完整的 URL，比如 http://helloflask.com/hello，在本地运行程序时则会获得 http://localhost:5000/hello。

1.8 Flask 命令

除了 Flask 内置的 flask run 等命令，我们也可以自定义命令。在虚拟环境安装 Flask 后，包含许多内置命令的 flask 脚本就可以使用了。在前面我们已经接触了很多 flask 命令，比如运行



服务器的 flask run，启动 shell 的 flask shell。

通过创建任意一个函数，并为其添加 app.cli.command() 装饰器，我们就可以注册一个 flask 命令。代码清单 1-4 创建了一个自定义的 hello() 命令函数，在函数中我们仍然只是打印一行问候。

代码清单 1-4 hello/app.py：创建自定义命令

```
@app.cli.command()
def hello():
    click.echo('Hello, Human!')
```

函数的名称即为命令名称，这里注册的命令即 hello，你可以使用 flask hello 命令来触发函数。作为替代，你也可以在 app.cli.command() 装饰器中传入参数来设置命令名称，比如 app.cli.command('say-hello') 会把命令名称设置为 say-hello，完整的命令即 flask say-hello。

借助 click 模块的 echo() 函数，我们可以在命令行界面输出字符。命令函数的文档字符串则会作为帮助信息显示（flask hello --help）。在命令行下执行 flask hello 命令就会触发这个 hello() 函数：

```
$ flask hello
Hello, Human!
```

在命令下执行 flask --help 可以查看 Flask 提供的命令帮助文档，我们自定义的 hello 命令也会出现在输出的命令列表中，如下所示：

```
$ flask --help
Usage: flask [OPTIONS] COMMAND [ARGS]...

A general utility script for Flask applications.

...
Options:
--version  Show the flask version
--help      Show this message and exit.

Commands:
hello      Just say hello. # 我们注册的自定义命令
routes     Show the routes for the app. # 显示所有注册的路由
run        Runs a development server.
shell      Runs a shell in the app context.
```



附注 关于自定义命令更多的设置和功能请参考 Click 的官方文档 (<http://click.pocoo.org/6/>)。

1.9 模板与静态文件

一个完整的网站当然不能只返回用户一句“Hello, World!”，我们需要模板（template）和静态文件（static file）来生成更加丰富的网页。模板即包含程序页面的 HTML 文件，静态文件则是需要在 HTML 文件中加载的 CSS 和 JavaScript 文件，以及图片、字体文件等资源文件。默认



情况下，模板文件存放在项目根目录中的 templates 文件夹中，静态文件存放在 static 文件夹下，这两个文件夹需要和包含程序实例的模块处于同一个目录下，对应的项目结构示例如下所示：

```
hello/
  - templates/
  - static/
  - app.py
```

在开发 Flask 程序时，使用 CSS 框架和 JavaScript 库是很常见的需求，而且有很多扩展都提供了对 CSS 框架和 JavaScript 库的集成功能。使用这些扩展时都需要加载对应的 CSS 和 JavaScript 文件，通常这些扩展都会提供一些可以在 HTML 模板中使用的加载方法 / 函数，使用这些方法即可渲染出对应的 link 标签和 script 标签。这些方法一般会直接从 CDN 加载资源，有些提供了手动传入资源 URL 的功能，有些甚至提供了内置的本地资源。

我建议在开发环境下使用本地资源，这样可以提高加载速度。最好自己下载到 static 目录下，统一管理，出于方便的考虑也可以使用扩展内置的本地资源。在过渡到生产环境时，自己手动管理所有本地资源或自己设置 CDN，避免使用扩展内置的资源。这个建议主要基于下面这些考虑因素：

- 鉴于国内的网络状况，扩展默认使用的国外 CDN 可能会无法访问，或访问过慢。
- 不同扩展内置的加载方法可能会加载重复的依赖资源，比如 jQuery。
- 在生产环境下，将静态文件集中在一起更方便管理。
- 扩展内置的资源可能出现版本过旧的情况。

关于模板和静态文件的使用，我们将在第 3 章详细介绍。



附注 CDN 指分布式服务器系统。服务商把你需要的资源存储在分布于不同地理位置的多个服务器，它会根据用户的地理位置来就近分配服务器提供服务（服务器越近，资源传送就越快）。使用 CDN 服务可以加快网页资源的加载速度，从而优化用户体验。对于开源的 CSS 和 JavaScript 库，CDN 提供商通常会免费提供服务。

1.10 Flask 与 MVC 架构

你也许会困惑为什么用来处理请求并生成响应的函数被称为“视图函数（view function）”，其实这个命名并不合理。在 Flask 中，这个命名的约定来自 Werkzeug，而 Werkzeug 中 URL 匹配的实现主要参考了 Routes（一个 URL 匹配库），再往前追溯，Routes 的实现又参考了 Ruby on Rails (<http://rubyonrails.org/>)。在 Ruby on Rails 中，术语 views 用来表示 MVC（Model-View-Controller，模型 – 视图 – 控制器）架构中的 View。

MVC 架构最初是用来设计桌面程序的，后来也被用于 Web 程序，应用了这种架构的 Web 框架有 Django、Ruby on Rails 等。在 MVC 架构中，程序被分为三个组件：数据处理（Model）、用户界面（View）、交互逻辑（Controller）。如果套用 MVC 架构的内容，那么 Flask 中视图函数的名称其实并不严谨，使用控制器函数（Controller Function）似乎更合适些，虽然它也附带处理用户界面。严格来说，Flask 并不是 MVC 架构的框架，因为它没有内置数据模型支持。为了



方便表述，在本书中，使用了 `app.route()` 装饰器的函数仍被称为视图函数，同时会使用“<函数名>视图”（比如 `index` 视图）的形式来代指某个视图函数。

粗略归类，如果想要使用 Flask 来编写一个 MVC 架构的程序，那么视图函数可以作为控制器（Controller），视图（View）则是我们第 3 章将要学习的使用 Jinja2 渲染的 HTML 模板，而模型（Model）可以使用其他库来实现，在第 5 章我们会介绍使用 SQLAlchemy 来创建数据库模型。

1.11 本章小结

本章我们学习了 Flask 程序的运作方式和一些基本概念，这为我们进一步学习打下了基础。下一章，我们会了解隐藏在 Flask 背后的重要角色——HTTP，并学习 Flask 是如何与之进行交互的。



Flask 与 HTTP

在第 1 章，我们已经了解了 Flask 的基本知识，如果想要进一步开发更复杂的 Flask 应用，我们就得了解 Flask 与 HTTP 协议的交互方式。HTTP (Hypertext Transfer Protocol, 超文本传输协议) 定义了服务器和客户端之间信息交流的格式和传递方式，它是万维网 (World Wide Web) 中数据交换的基础。

在这一章，我们会了解 Flask 处理请求和响应的各种方式，并对 HTTP 协议以及其他非常规 HTTP 请求进行简单的介绍。虽然本章的内容很重要，但鉴于内容有些晦涩难懂，如果感到困惑也不用担心，本章介绍的内容你会在后面的实践中逐渐理解和熟悉。如果你愿意，也可以临时跳过本章，等到学习完本书第一部分再回来重读。



HTTP 的详细定义在 RFC 7231~7235 中可以看到。RFC(Request For Comment, 请求评议) 是一系列关于互联网标准和信息的文件，可以将其理解为互联网 (Internet) 的设计文档。完整的 RFC 列表可以在这里看到：<https://tools.ietf.org/rfc/>。

本章的示例程序在 `helloflask/demos/http` 目录下，确保当前工作目录在 `helloflask/demos/http` 下并激活了虚拟环境，然后执行 `flask run` 命令运行程序：

```
$ cd demos/http  
$ flask run
```



第一部分的示例程序都会运行在本地机的 5000 端口，在运行新的示例程序前，请确保没有其他程序在运行。

2.1 请求响应循环

为了更贴近现实，我们以一个真实的 URL 为例：



<http://helloflask.com/hello>

当我们在浏览器中的地址栏中输入这个 URL，然后按下 Enter 时，稍等片刻，浏览器会显示一个问候页面。这背后到底发生了什么？你一定可以猜想到，这背后也有一个类似我们第 1 章编写的程序运行着。它负责接收用户的请求，并把对应的内容返回给客户端，显示在用户的浏览器上。事实上，每一个 Web 应用都包含这种处理模式，即“请求 - 响应循环（Request-Response Cycle）”：客户端发出请求，服务器端处理请求并返回响应，如图 2-1 所示。

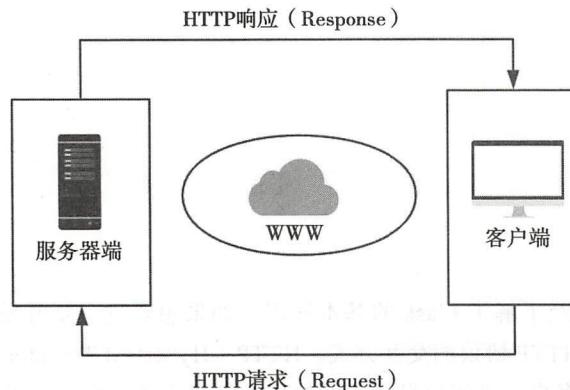


图 2-1 请求响应循环示意图



附注 客户端（Client Side）是指用来提供给用户的与服务器通信的各种软件。在本书中，客户端通常指 Web 浏览器（后面简称浏览器），比如 Chrome、Firefox、IE 等；服务器端（Server Side）则指为用户提供服务的服务器，也是我们的程序运行的地方。

这是每一个 Web 程序的基本工作模式，如果再进一步，这个模式又包含着更多的工作单元，图 2-2 展示了一个 Flask 程序工作的实际流程。

从图 2-2 中可以看出，HTTP 在整个流程中起到了至关重要的作用，它是客户端和服务器端之间沟通的桥梁。

当用户访问一个 URL，浏览器便生成对应的 HTTP 请求，经由互联网发送到对应的 Web 服务器。Web 服务器接收请求，通过 WSGI 将 HTTP 格式的请求数据转换成我们的 Flask 程序能够使用的 Python 数据。在程序中，Flask 根据请求的

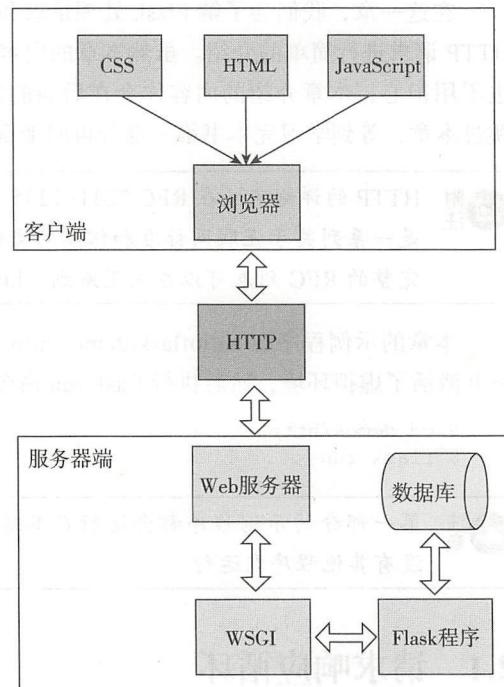


图 2-2 Flask Web 程序工作流程



URL 执行对应的视图函数，获取返回值生成响应。响应依次经过 WSGI 转换生成 HTTP 响应，再经由 Web 服务器传递，最终被发出请求的客户端接收。浏览器渲染响应中包含的 HTML 和 CSS 代码，并执行 JavaScript 代码，最终把解析后的页面呈现在用户浏览器的窗口中。

 提示 关于 WSGI 的更多细节，我们会在第 16 章进行详细介绍。

 提示 这里的服务器指的是处理请求和响应的 Web 服务器，比如我们上一章介绍的开发服务器，而不是指物理层面上的服务器主机。

2.2 HTTP 请求

URL 是一个请求的起源。不论服务器是运行在美国洛杉矶，还是运行在我们自己的电脑上，当我们输入指向服务器所在地址的 URL，都会向服务器发送一个 HTTP 请求。一个标准的 URL 由很多部分组成，以下面这个 URL 为例：

`http://helloflask.com/hello?name=Grey`

这个 URL 的各个组成部分如表 2-1 所示。

表 2-1 URL 组成部分

信 息	说 明
<code>http://</code>	协议字符串，指定要使用的协议
<code>helloflask.com</code>	服务器的地址（域名）
<code>/hello?name=Grey</code>	要获取的资源路径（path），类似 UNIX 的文件目录结构

 附注 这个 URL 后面的 `?name=Grey` 部分是查询字符串（query string）。URL 中的查询字符串用来向指定的资源传递参数。查询字符串从问号？开始，以键值对的形式写出，多个键值对之间使用 & 分隔。

2.2.1 请求报文

当我们在浏览器中访问这个 URL 时，随之产生的一个发向 `http://helloflask.com` 所在服务器的请求。请求的实质是发送到服务器上的一些数据，这种浏览器与服务器之间交互的数据被称为报文（message），请求时浏览器发送的数据被称为请求报文（request message），而服务器返回的数据被称为响应报文（response message）。

请求报文由请求的方法、URL、协议版本、首部字段（header）以及内容实体组成。前面的请求产生的请求报文示意如表 2-2 所示。

表 2-2 请求报文示意表

组成说明	请求报文内容
报文首部：请求行（方法、URL、协议）	GET /hello HTTP/1.1
报文首部：各种首部字段	Host: helloflask.com Connection: keep-alive Cache-Control: max-age=0 User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/59.0.3071.104 Safari/537.36 ...
空行	
报文主体	name=Grey

如果你想看真实的 HTTP 报文，可以在浏览器中向任意一个有效的 URL 发起请求，然后在浏览器的开发者工具（F12）里的 Network 标签中看到 URL 对应资源加载的所有请求列表，单击任一个请求条目即可看到报文信息，图 2-3 是使用 Chrome 访问本地示例程序的示例。

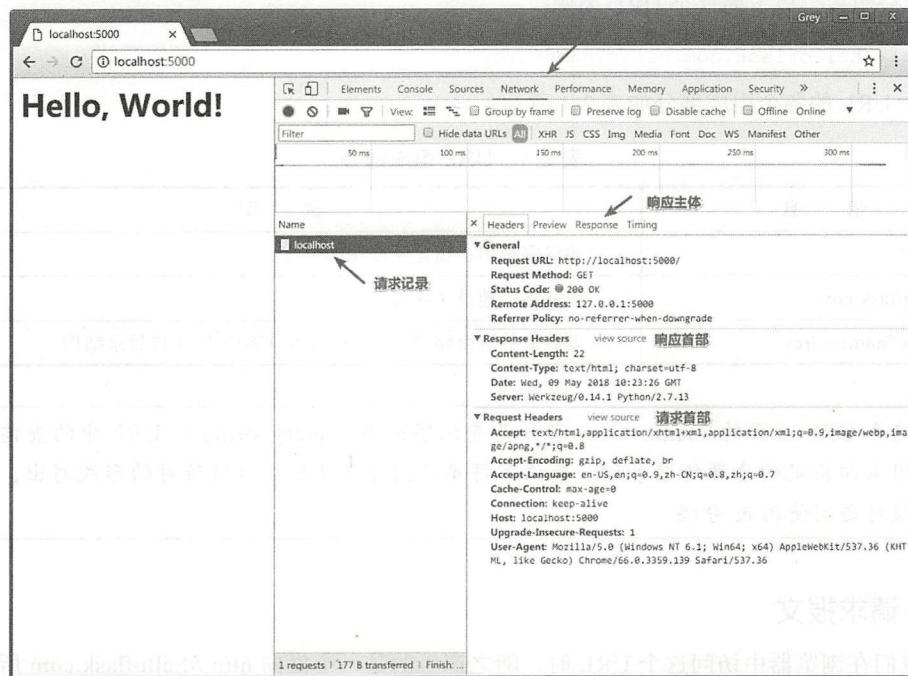


图 2-3 在 Chrome 浏览器中查看请求和响应报文

报文由报文首部和报文主体组成，两者由空行分隔，请求报文的主体一般为空。如果 URL 中包含查询字符串，或是提交了表单，那么报文主体将会是查询字符串和表单数据。

HTTP 通过方法来区分不同的请求类型。比如，当你直接访问一个页面时，请求的方法是 GET；当你在某个页面填写了表单并提交时，请求方法则通常为 POST。表 2-3 是常见的几种

HTTP方法类型。

表 2-3 常见的 HTTP 方法

方 法	说 明	方 法	说 明
GET	获取资源	DELETE	删除资源
POST	传输数据	HEAD	获得报文首部
PUT	传输文件	OPTIONS	询问支持的方法

报文首部包含了请求的各种信息和设置，比如客户端的类型、是否设置缓存、语言偏好等。

 **附注** HTTP 中可用的首部字段列表可以在 <https://www.iana.org/assignments/message-headers/message-headers.xhtml> 看到。请求方法的详细列表和说明可以在 RFC 7231 (<https://tools.ietf.org/html/rfc7231>) 中看到。

如果运行了示例程序，那么当你在浏览器中访问 `http://127.0.0.1:5000/hello` 时，开发服务器会在命令行中输出一条记录日志，其中包含请求的主要信息：

```
127.0.0.1 - - [02/Aug/2017 09:51:37] "GET /hello HTTP/1.1" 200 -
```

2.2.2 Request 对象

现在该让 Flask 的请求对象 `request` 出场了，这个请求对象封装了从客户端发来的请求报文，我们能从它获取请求报文中的所有数据。

 **注意** 请求解析和响应封装实际上大部分是由 Werkzeug 完成的，Flask 子类化 Werkzeug 的请求（Request）和响应（Response）对象并添加了和程序相关的特定功能。在这里为了方便理解，我们先略过不谈。在第 16 章，我们会详细了解 Flask 的工作原理。

和上一节一样，我们先从 URL 说起。假设请求的 URL 是 `http://helloflask.com/hello?name=Grey`，当 Flask 接收到请求后，请求对象会提供多个属性来获取 URL 的各个部分，常用的属性如表 2-4 所示。

表 2-4 使用 `request` 的属性获取请求 URL

属性	值	属性	值
path	u'/hello'	base_url	u'http://helloflask.com/hello'
full_path	u'/hello?name=Grey'	url	u'http://helloflask.com/hello?name=Grey'
host	u'helloflask.com'	url_root	u'http://helloflask.com/'
host_url	u'http://helloflask.com/'		

除了 URL，请求报文中的其他信息都可以通过 `request` 对象提供的属性和方法获取，其中常用的部分如表 2-5 所示。

表 2-5 request 对象常用的属性和方法

属性 / 方法	说 明
args	Werkzeug 的 ImmutableMultiDict 对象。存储解析后的查询字符串，可通过字典方式获取键值。如果你想获取未解析的原生查询字符串，可以使用 query_string 属性
blueprint	当前蓝本的名称，关于蓝本的概念在本书第二部分会详细介绍
cookies	一个包含所有随请求提交的 cookies 的字典
data	包含字符串形式的请求数据
endpoint	与当前请求相匹配的端点值
files	Werkzeug 的 MultiDict 对象，包含所有上传文件，可以使用字典的形式获取文件。使用的键为文件 input 标签中的 name 属性值，对应的值为 Werkzeug 的 FileStorage 对象，可以调用 save() 方法并传入保存路径来保存文件
form	Werkzeug 的 ImmutableMultiDict 对象。与 files 类似，包含解析后的表单数据。表单字段值通过 input 标签的 name 属性值作为键获取
values	Werkzeug 的 CombinedMultiDict 对象，结合了 args 和 form 属性的值
get_data(cache=True, as_text=False, parse_from_data=False)	获取请求中的数据，默认读取为字节字符串 (bytestring)，将 as_text 设为 True 则返回值将是解码后的 unicode 字符串
get_json(self, force=False, silent=False, cache=True)	作为 JSON 解析并返回数据，如果 MIME 类型不是 JSON，返回 None (除非 force 设为 True)；解析出错则抛出 Werkzeug 提供的 BadRequest 异常 (如果未开启调试模式，则返回 400 错误响应，后面会详细介绍)，如果 silent 设为 True 则返回 None；cache 设置是否缓存解析后的 JSON 数据
headers	一个 Werkzeug 的 EnvironHeaders 对象，包含首部字段，可以以字典的形式操作
is_json	通过 MIME 类型判断是否为 JSON 数据，返回布尔值
json	包含解析后的 JSON 数据，内部调用 get_json()，可通过字典的方式获取键值
method	请求的 HTTP 方法
referrer	请求发起的源 URL，即 referer
scheme	请求的 URL 模式 (http 或 https)
user_agent	用户代理 (User Agent, UA)，包含了用户的客户端类型，操作系统类型等信息



Werkzeug 的 MultiDict 类是字典的子类，它主要实现了同一个键对应多个值的情况。比如一个文件上传字段可能会接收多个文件。这时就可以通过 getlist() 方法来获取文件对象列表。而 ImmutableMultiDict 类继承了 MultiDict 类，但其值不可更改。更多内容可访问 Werkzeug 相关数据结构章节 <http://werkzeug.pocoo.org/docs/latest/datastructures/>。

在我们的示例程序中实现了同样的功能。当你访问 <http://localhost:5000/hello?name=Grey> 时，页面加载后会显示“Hello, Grey!”。这说明处理这个 URL 的视图函数从查询字符串中获取了查询参数 name 的值，如代码清单 2-1 所示。

代码清单2-1 获取请求URL中的查询字符串

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/hello')
def hello():
    name = request.args.get('name', 'Flask')      # 获取查询参数name的值
    return '<h1>Hello, %s!</h1>' % name          # 插入到返回值中
```



注意 上面的示例代码包含安全漏洞，在现实中我们要避免直接将用户传入的数据直接作为响应返回，在本章的末尾我们将介绍这个安全漏洞的具体细节和防范措施。

需要注意的是，和普通的字典类型不同，当我们从 `request` 对象的类型为 `MultiDict` 或 `ImmutableMultiDict` 的属性（比如 `files`、`form`、`args`）中直接使用键作为索引获取数据时（比如 `request.args['name']`），如果没有对应的键，那么会返回 HTTP 400 错误响应（Bad Request，表示请求无效），而不是抛出 `KeyError` 异常，如图 2-4 所示。为了避免这个错误，我们应该使用 `get()` 方法获取数据，如果没有对应的值则返回 `None`；`get()` 方法的第二个参数可以设置默认值，比如 `request.args.get('name', 'Human')`。



图 2-4 400 错误响应



提示 如果开启了调试模式，那么会抛出 `BadRequestKeyError` 异常并显示对应的错误堆栈信息，而不是常规的 400 响应。

2.2.3 在 Flask 中处理请求

URL 是指向网络上资源的地址。在 Flask 中，我们需要让请求的 URL 匹配对应的视图函数，视图函数返回值就是 URL 对应的资源。

1. 路由匹配

为了便于将请求分发到对应的视图函数，程序实例中存储了一个路由表（`app.url_map`），其中定义了 URL 规则和视图函数的映射关系。当请求发来后，Flask 会根据请求报文中的 URL（path 部分）来尝试与这个表中的所有 URL 规则进行匹配，调用匹配成功的视图函数。如果没有找到匹配的 URL 规则，说明程序中没有处理这个 URL 的视图函数，Flask 会自动返回 404 错误响应（Not Found，表示资源未找到）。你可以尝试在浏览器中访问 `http://localhost:5000/nothing`，因为我们的程序中没有视图函数负责处理这个 URL，所以你会得到 404 响应，如图 2-5 所示。



图 2-5 404 错误响应

如果你经常上网，那么肯定会对这个错误代码相当熟悉，它表示请求的资源没有找到。和前面提及的 400 错误响应一样，这类错误代码被称为 HTTP 状态码，用来表示响应的状态，具体会在下面详细讨论。

当请求的 URL 与某个视图函数的 URL 规则匹配成功时，对应的视图函数就会被调用。使用 `flask routes` 命令可以查看程序中定义的所有路由，这个列表由 `app.url_map` 解析得到：

```
$ flask routes
Endpoint  Methods  Rule
-----  -----
hello      GET      /hello
go_back   GET      /goback/<int:age>
hi        GET      /hi
```

```
...
static    GET      /static/<path:filename>
```

在输出的文本中，我们可以看到每个路由对应的端点（Endpoint）、HTTP 方法（Methods）和 URL 规则（Rule），其中 static 端点是 Flask 添加的特殊路由，用来访问静态文件，具体我们会在第 3 章学习。

2. 设置监听的 HTTP 方法

在上一节通过 flask routes 命令打印出的路由列表可以看到，每一个路由除了包含 URL 规则外，还设置了监听的 HTTP 方法。GET 是最常用的 HTTP 方法，所以视图函数默认监听的方法类型就是 GET，HEAD、OPTIONS 方法的请求由 Flask 处理，而像 DELETE、PUT 等方法一般不会在程序中实现，在后面我们构建 Web API 时才会用到这些方法。

我们可以在 app.route() 装饰器中使用 methods 参数传入一个包含监听的 HTTP 方法的可迭代对象。比如，下面的视图函数同时监听 GET 请求和 POST 请求：

```
@app.route('/hello', methods=['GET', 'POST'])
def hello():
    return '<h1>Hello, Flask!</h1>'
```

当某个请求的方法不符合要求时，请求将无法被正常处理。比如，在提交表单时通常使用 POST 方法，而如果提交的目标 URL 对应的视图函数只允许 GET 方法，这时 Flask 会自动返回一个 405 错误响应（Method Not Allowed，表示请求方法不允许），如图 2-6 所示。

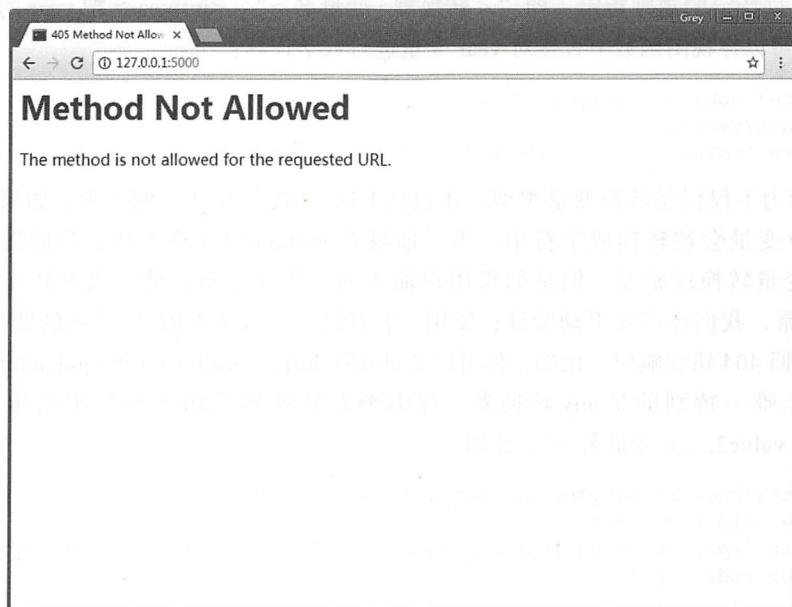


图 2-6 405 错误响应

通过定义方法列表，我们可以为同一个 URL 规则定义多个视图函数，分别处理不同 HTTP 方法的请求，这在本书第二部分构建 Web API 时会用到这个特性。

3. URL 处理

从前面的路由列表中可以看到，除了 /hello，这个程序还包含许多 URL 规则，比如和 go_back 端点对应的 /goback/<int:year>。现在请尝试访问 <http://localhost:5000/goback/34>，在 URL 中加入一个数字作为时光倒流的年数，你会发现加载后的页面中有通过传入的年数计算出的年份：“Welcome to 1984!”。仔细观察一下，你会发现 URL 规则中的变量部分有一些特别，<int:year> 表示为 year 变量添加了一个 int 转换器，Flask 在解析这个 URL 变量时会将其转换为整型。URL 中的变量部分默认类型为字符串，但 Flask 提供了一些转换器可以在 URL 规则里使用，如表 2-6 所示。

表 2-6 Flask 内置的 URL 变量转换器

转 换 器	说 明
string	不包含斜线的字符串（默认值）
int	整型
float	浮点数
path	包含斜线的字符串。static 路由的 URL 规则中的 filename 变量就使用了这个转换器
any	匹配一系列给定值中的一个元素
uuid	UUID 字符串

转换器通过特定的规则指定，即“<转换器：变量名>”。<int:year> 把 year 的值转换为整数，因此我们可以在视图函数中直接对 year 变量进行数学计算：

```
@app.route('goback/<int:year>')
def go_back(year):
    return '<p>Welcome to %d!</p>' % (2018 - year)
```

默认的行为不仅仅是转换变量类型，还包括 URL 匹配。在这个例子中，如果不使用转换器，默认 year 变量会被转换成字符串，为了能够在 Python 中计算天数，我们需要使用 int() 函数将 year 变量转换成整型。但是如果用户输入的是英文字母，就会出现转换错误，抛出 ValueError 异常，我们还需要手动验证；使用了转换器后，如果 URL 中传入的变量不是数字，那么会直接返回 404 错误响应。比如，你可以尝试访问 <http://localhost:5000/goback/tang>。

在用法上唯一特别的是 any 转换器，你需要在转换器后添加括号来给出可选值，即“<any(value1, value2, ...): 变量名>”，比如：

```
@app.route('/colors/<any(blue, white, red):color>')
def three_colors(color):
    return '<p>Love is patient and kind. Love is not jealous or boastful or proud
or rude.</p>'
```

当你在浏览器中访问 <http://localhost:5000/colors/<color>> 时，如果将 <color> 部分替换为 any 转换器中设置的可选值以外的任意字符，均会获得 404 错误响应。

如果你想在 any 转换器中传入一个预先定义的列表，可以通过格式化字符串的方式（使用 % 或是 format() 函数）来构建 URL 规则字符串，比如：

```

colors = ['blue', 'white', 'red']

@app.route('/colors/<any(%s):color>' % str(colors)[1:-1])
...

```

2.2.4 请求钩子

有时我们需要对请求进行预处理（preprocessing）和后处理（postprocessing），这时可以使用Flask提供的一些请求钩子（Hook），它们可以用来注册在请求处理的不同阶段执行的处理函数（或称为回调函数，即Callback）。这些请求钩子使用装饰器实现，通过程序实例app调用，用法很简单：以before_request钩子（请求之前）为例，当你对一个函数附加了app.before_request装饰器后，就会将这个函数注册为before_request处理函数，每次执行请求前都会触发所有before_request处理函数。Flask默认实现的五种请求钩子如表2-7所示。

表2-7 请求钩子

钩子	说明
before_first_request	注册一个函数，在处理第一个请求前运行
before_request	注册一个函数，在处理每个请求前运行
after_request	注册一个函数，如果没有未处理的异常抛出，会在每个请求结束后运行
teardown_request	注册一个函数，即使有未处理的异常抛出，会在每个请求结束后运行。如果发生异常，会传入异常对象作为参数到注册的函数中
after_this_request	在视图函数内注册一个函数，会在这个请求结束后运行

这些钩子使用起来和app.route()装饰器基本相同，每个钩子可以注册任意多个处理函数，函数名并不是必须和钩子名称相同，下面是一个基本示例：

```

@app.before_request
def do_something():
    pass # 这里的代码会在每个请求处理前执行

```

假如我们创建了三个视图函数A、B、C，其中视图C使用了after_this_request钩子，那么当请求A进入后，整个请求处理周期的请求处理函数调用流程如图2-7所示。

下面是请求钩子的一些常见应用场景：

- before_first_request：在玩具程序中，运行程序前我们需要进行一些程序的初始化操作，比如创建数据库表，添加管理员用户。这些工作可以放到使用before_first_request装饰器注册的函数中。
- before_request：比如网站上要记录用户最后在线的时间，可以通过用户最后发送的请求时间来实现。为了避免在每个视图函数都添加更新在线时间的代码，我们可以仅在使用before_request钩子注册的函数中调用这段代码。
- after_request：我们经常在视图函数中进行数据库操作，比如更新、插入等，之后需要将更改提交到数据库中。提交更改的代码就可以放到after_request钩子注册的函数中。

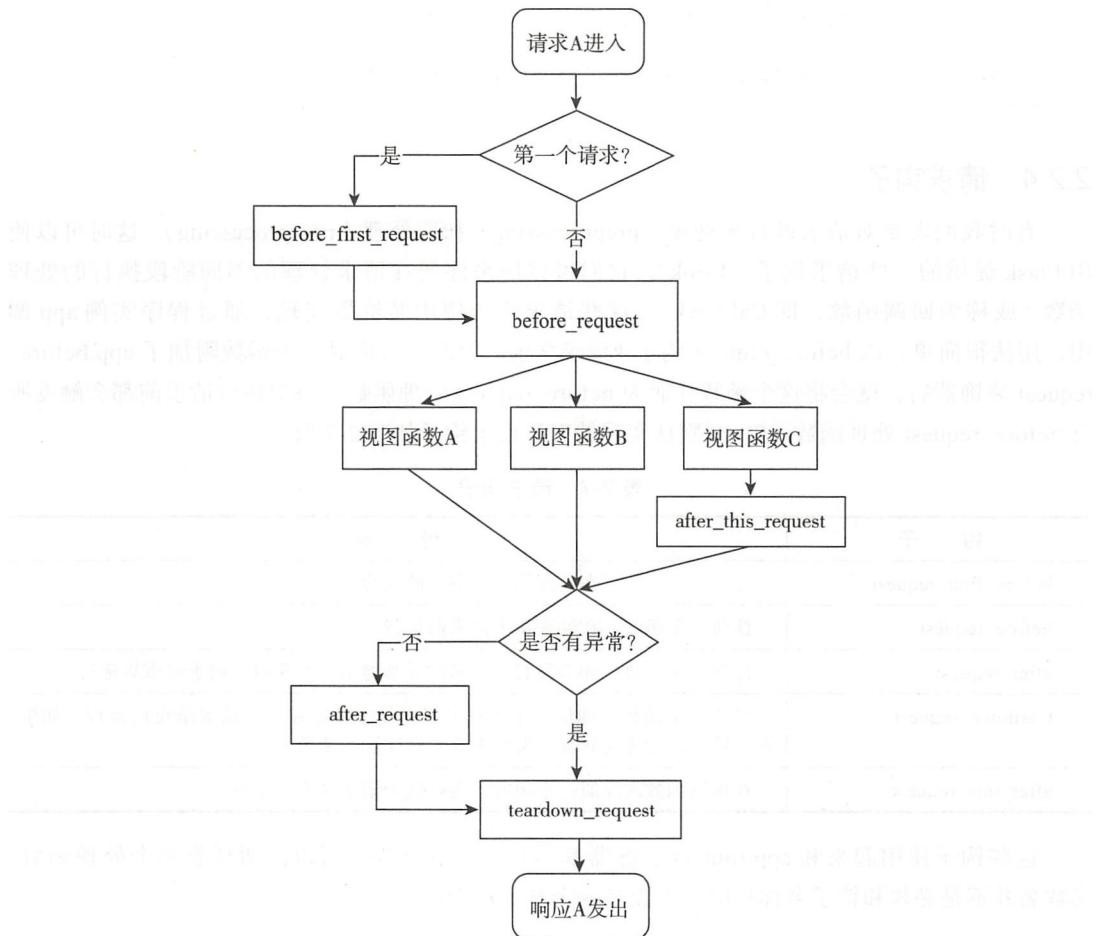


图 2-7 请求处理函数调用示意图

另一种常见的应用是建立数据库连接，通常会有多个视图函数需要建立和关闭数据库连接，这些操作基本相同。一个理想的解决方法是在请求之前（before_request）建立连接，在请求之后（teardown_request）关闭连接。通过在使用相应的请求钩子注册的函数中添加代码就可以实现。这很像单元测试中的 `setUp()` 方法和 `tearDown()` 方法。

注意 `after_request` 钩子和 `after_this_request` 钩子必须接收一个响应类对象作为参数，并且返回同一个或更新后的响应对象。

2.3 HTTP 响应

在 Flask 程序中，客户端发出的请求触发相应的视图函数，获取返回值会作为响应的主体，最后生成完整的响应，即响应报文。

2.3.1 响应报文

响应报文主要由协议版本、状态码 (status code)、原因短语 (reason phrase)、响应首部和响应主体组成。以发向 localhost:5000/hello 的请求为例，服务器生成的响应报文示意如表 2-8 所示。

表 2-8 响应报文

组成说明	响应报文内容
报文首部：状态行（协议、状态码、原因短语）	HTTP/1.1 200 OK
报文首部：各种首部字段	Content-Type: text/html; charset=utf-8 Content-Length: 22 Server: Werkzeug/0.12.2 Python/2.7.13 Date: Thu, 03 Aug 2017 05:05:54 GMT ...
空行	
报文主体	<h1>Hello, Human!</h1>

响应报文的头部包含一些关于响应和服务器的信息，这些内容由 Flask 生成，而我们在视图函数中返回的内容即为响应报文中的主体内容。浏览器接收到响应后，会把返回的响应主体解析并显示在浏览器窗口上。

HTTP 状态码用来表示请求处理的结果，表 2-9 是常见的几种状态码和相应的原因短语。

表 2-9 常见的 HTTP 状态码

类型	状态码	原因短语（用于解释状态码）	说明
成功	200	OK	请求被正常处理
	201	Created	请求被处理，并创建了一个新资源
	204	No Content	请求处理成功，但无内容返回
重定向	301	Moved Permanently	永久重定向
	302	Found	临时性重定向
	304	Not Modified	请求的资源未被修改，重定向到缓存的资源
客户端错误	400	Bad Request	表示请求无效，即请求报文中存在错误
	401	Unauthorized	类似 403，表示请求的资源需要获取授权信息，在浏览器中会弹出认证弹窗
	403	Forbidden	表示请求的资源被服务器拒绝访问
	404	Not Found	表示服务器上无法找到请求的资源或 URL 无效
服务器端错误	500	Internal Server Error	服务器内部发生错误

 提示 当关闭调试模式时，即 FLASK_ENV 使用默认值 production，如果程序出错，Flask 会自动返回 500 错误响应；而调试模式下则会显示调试信息和错误堆栈。

 **附注** 响应状态码的详细列表和说明可以在 RFC 7231 (<https://tools.ietf.org/html/rfc7231>) 中看到。

2.3.1 在 Flask 中生成响应

响应在 Flask 中使用 Response 对象表示，响应报文中的大部分内容由服务器处理，大多数情况下，我们只负责返回主体内容。

根据我们在上一节介绍的内容，Flask 会先判断是否可以找到与请求 URL 相匹配的路由，如果没有则返回 404 响应。如果找到，则调用对应的视图函数，视图函数的返回值构成了响应报文的主体内容，正确返回时状态码默认为 200。Flask 会调用 make_response() 方法将视图函数返回值转换为响应对象。

完整地说，视图函数可以返回最多由三个元素组成的元组：响应主体、状态码、首部字段。其中首部字段可以为字典，或是两元素元组组成的列表。

比如，普通的响应可以只包含主体内容：

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

默认的状态码为 200，下面指定了不同的状态码：

```
@app.route('/hello')
def hello():
    ...
    return '<h1>Hello, Flask!</h1>', 201
```

有时你会想附加或修改某个首部字段。比如，要生成状态码为 3XX 的重定向响应，需要将首部中的 Location 字段设置为重定向的目标 URL：

```
@app.route('/hello')
def hello():
    ...
    return '', 302, {'Location': 'http://www.example.com'}
```

现在访问 <http://localhost:5000/hello>，会重定向到 <http://www.example.com>。在多数情况下，除了响应主体，其他部分我们通常只需要使用默认值即可。

1. 重定向

如果你访问 <http://localhost:5000/hi>，你会发现页面加载后地址栏中的 URL 变为了 <http://localhost:5000/hello>。这种行为被称为重定向（Redirect），你可以理解为网页跳转。在上一节的示例中，状态码为 302 的重定向响应的主体为空，首部中需要将 Location 字段设为重定向的目标 URL，浏览器接收到重定向响应后会向 Location 字段中的目标 URL 发起新的 GET 请求，整个流程如图 2-8 所示。

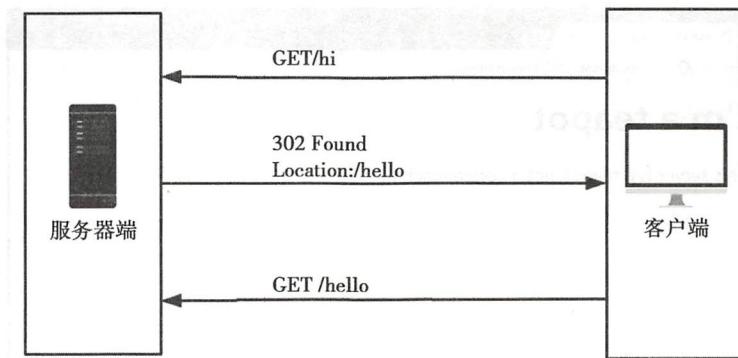


图 2-8 重定向流程示意图

在 Web 程序中，我们经常需要进行重定向。比如，当某个用户在没有经过认证的情况下访问需要登录后才能访问的资源，程序通常会重定向到登录页面。

对于重定向这一类特殊响应，Flask 提供了一些辅助函数。除了像前面那样手动生成 302 响应，我们可以使用 Flask 提供的 `redirect()` 函数来生成重定向响应，重定向的目标 URL 作为第一个参数。前面的例子可以简化为：

```

from flask import Flask, redirect
# ...
@app.route('/hello')
def hello():
    return redirect('http://www.example.com')
    
```

 提示 使用 `redirect()` 函数时，默认的状态码为 302，即临时重定向。如果你想修改状态码，可以在 `redirect()` 函数中作为第二个参数或使用 `code` 关键字传入。

如果要在程序内重定向到其他视图，那么只需在 `redirect()` 函数中使用 `url_for()` 函数生成目标 URL 即可，如代码清单 2-2 所示。

代码清单 2-2 http/app.py：重定向到其他视图

```

from flask import Flask, redirect, url_for
...
@app.route('/hi')
def hi():
    ...
    return redirect(url_for('hello')) # 重定向到/hello

@app.route('/hello')
def hello():
    ...
    
```

2. 错误响应

如果你访问 `http://localhost:5000/brew/coffee`，会获得一个 418 错误响应（I'm a teapot），如图 2-9 所示。



图 2-9 418 错误响应

附注 418 错误响应由 IETF (Internet Engineering Task Force, 互联网工程任务组) 在 1998 年愚人节发布的 HTCP(CP(Hyper Text Coffee Pot Control Protocol, 超文本咖啡壶控制协议) 中定义(玩笑), 当一个控制茶壶的 HTCP(CP 收到 BREW 或 POST 指令要求其煮咖啡时应当回传此错误。

大多数情况下, Flask 会自动处理常见的错误响应。HTTP 错误对应的异常类在 Werkzeug 的 werkzeug.exceptions 模块中定义, 抛出这些异常即可返回对应的错误响应。如果你想手动返回错误响应, 更方便的方法是使用 Flask 提供的 abort() 函数。

在 abort() 函数中传入状态码即可返回对应的错误响应, 代码清单 2-3 中的视图函数返回 404 错误响应。

代码清单 2-3 http/app.py: 返回 404 错误响应

```
from flask import Flask, abort
...
@app.route('/404')
def not_found():
    abort(404)
```

提示 abort() 函数前不需要使用 return 语句, 但一旦 abort() 函数被调用, abort() 函数之后的代码将不会被执行。

 **附注** 虽然我们有必要返回正确的状态码，但这不是必须的。比如，当某个用户没有权限访问某个资源时，返回 404 错误要比 403 错误更加友好。

2.3.2 响应格式

在 HTTP 响应中，数据可以通过多种格式传输。大多数情况下，我们会使用 HTML 格式，这也是 Flask 中的默认设置。在特定的情况下，我们也会使用其他格式。不同的响应数据格式需要设置不同的 MIME 类型，MIME 类型在首部的 Content-Type 字段中定义，以默认的 HTML 类型为例：

```
Content-Type: text/html; charset=utf-8
```

 **附注** MIME 类型（又称为 media type 或 content type）是一种用来标识文件类型的机制，它与文件扩展名相对应，可以让客户端区分不同的内容类型，并执行不同的操作。一般的格式为“类型名 / 子类型名”，其中的子类型名一般为文件扩展名。比如，HTML 的 MIME 类型为“text/html”，png 图片的 MIME 类型为“image/png”。完整的标准 MIME 类型列表可以在这里看到：<https://www.iana.org/assignments/media-types/media-types.xhtml>。

如果你想使用其他 MIME 类型，可以通过 Flask 提供的 make_response() 方法生成响应对象，传入响应的主体作为参数，然后使用响应对象的 mimetype 属性设置 MIME 类型，比如：

```
from flask import make_response

@app.route('/foo')
def foo():
    response = make_response('Hello, World!')
    response.mimetype = 'text/plain'
    return response
```

你也可以直接设置首部字段，比如 response.headers['Content-Type'] = 'text/xml; charset=utf-8'。但操作 mimetype 属性更加方便，而且不用设置字符集（charset）选项。

常用的数据格式有纯文本、HTML、XML 和 JSON，下面我们分别对这几种数据进行简单的介绍和分析。为了对不同的数据类型进行对比，我们将会用不同的数据类型来表示一个便签的内容：Jane 写给 Peter 的一个提醒。

1. 纯文本

MIME 类型：text/plain

示例：

```
Note
to: Peter
from: Jane
heading: Reminder
body: Don't forget the party!
```

事实上，其他几种格式本质上都是纯文本。比如同样是一行包含 HTML 标签的文本“<h1>Hello, Flask!</h1>”，当 MIME 类型设置为纯文本时，浏览器会以文本形式显示“<h1>Hello, Flask!</h1>”；当 MIME 类型声明为 text/html 时，浏览器则会将其作为标题 1 样式的 HTML 代码渲染。

2. HTML

MIME 类型: text/html

示例：

```
<!DOCTYPE html>
<html>
<head></head>
<body>
    <h1>Note</h1>
    <p>to: Peter</p>
    <p>from: Jane</p>
    <p>heading: Reminder</p>
    <p>body: <strong>Don't forget the party!</strong></p>
</body>
</html>
```

HTML (<https://www.w3.org/html/>) 指 Hypertext Markup Language (超文本标记语言)，是最常用的数据格式，也是 Flask 返回响应的默认数据类型。从我们在本书一开始的最小程序中的视图函数返回的字符串，到我们后面会学习的 HTML 模板，都是 HTML。当数据类型为 HTML 时，浏览器会自动根据 HTML 标签以及样式类定义渲染对应的样式。

因为 HTML 常常包含丰富的信息，我们可以直接将 HTML 嵌入页面中，处理起来比较方便。因此，在普通的 HTTP 请求中我们使用 HTML 作为响应的内容，这也是默认的数据类型。

3. XML

MIME 类型: application/xml

示例：

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
    <to>Peter</to>
    <from>Jane</from>
    <heading>Reminder</heading>
    <body>Don't forget the party!</body>
</note>
```

XML (<https://www.w3.org/XML/>) 指 Extensible Markup Language (可扩展标记语言)，它是一种简单灵活的文本格式，被设计用来存储和交换数据。XML 的出现主要就是为了弥补 HTML 的不足：对于仅仅需要数据的请求来说，HTML 提供的信息太过丰富了，而且不易于重用。XML 和 HTML 一样都是标记性语言，使用标签来定义文本，但 HTML 中的标签用于显示内容，而 XML 中的标签只用于定义数据。XML 一般作为 AJAX 请求的响应格式，或是 Web API 的响应格式。

4. JSON

MIME 类型: application/json

示例:

```
{
    "note": {
        "to": "Peter",
        "from": "Jane",
        "heading": "Remider",
        "body": "Don't forget the party!"
    }
}
```

JSON (<http://json.org/>) 指 JavaScript Object Notation (JavaScript 对象表示法), 是一种流行的、轻量的数据交换格式。它的出现又弥补了 XML 的诸多不足: XML 有较高的重用性, 但 XML 相对于其他文档格式来说体积稍大, 处理和解析的速度较慢。JSON 轻量, 简洁, 容易阅读和解析, 而且能和 Web 默认的客户端语言 JavaScript 更好地兼容。JSON 的结构基于“键值对的集合”和“有序的值列表”, 这两种数据结构类似 Python 中的字典 (dictionary) 和列表 (list)。正是因为这种通用的数据结构, 使得 JSON 在同样基于这些结构的编程语言之间交换成为可能。

 提示 示例程序中提供了这一资源的不同格式响应, 你可以访问 <http://localhost:5000/> note/<content_type>, 通过将 content_type 的值依次更改为 text、html、xml 和 json 来获取不同格式的响应。比如, 访问 <http://localhost:5000/note/text> 将得到纯文本格式的响应。

Flask 通过引入 Python 标准库中的 json 模块 (或 simplejson, 如果可用) 为程序提供了 JSON 支持。你可以直接从 Flask 中导入 json 对象, 然后调用 dumps() 方法将字典、列表或元组序列化 (serialize) 为 JSON 字符串, 再使用前面介绍的方法修改 MIME 类型, 即可返回 JSON 响应, 如下所示:

```
from flask import Flask, make_response, json
...
@app.route('/foo')
def foo():
    data = {
        'name': 'Grey Li',
        'gender': 'male'
    }
    response = make_response(json.dumps(data))
    response.mimetype = 'application/json'
    return response
```

不过我们一般并不直接使用 json 模块的 dumps()、load() 等方法, 因为 Flask 通过包装这些方法提供了更方便的 jsonify() 函数。借助 jsonify() 函数, 我们仅需要传入数据或参数, 它会对我们传入的参数进行序列化, 转换成 JSON 字符串作为响应的主体, 然后生成一个响应对象, 并且设置正确的 MIME 类型。使用 jsonify 函数可以将前面的例子简化为这种形式:

```
from flask import jsonify

@app.route('/foo')
def foo():
    return jsonify(name='Grey Li', gender='male')
```

`jsonify()` 函数接收多种形式的参数。你既可以传入普通参数，也可以传入关键字参数。如果你想要更直观一点，也可以像使用 `dumps()` 方法一样传入字典、列表或元组，比如：

```
from flask import jsonify

@app.route('/foo')
def foo():
    return jsonify({'name': 'Grey Li', 'gender': 'male'})
```

上面两种形式的返回值是相同的，都会生成下面的 JSON 字符串：

```
{"gender": "male", "name": "Grey Li"}
```

另外，`jsonify()` 函数默认生成 200 响应，你也可以通过附加状态码来自定义响应类型，比如：

```
@app.route('/foo')
def foo():
    return jsonify(message='Error!'), 500
```

 提示 Flask 在获取请求中的 JSON 数据上也有很方便的解决方案，具体可以参考我们在 Request 对象小节介绍的 `request.get_json()` 方法和 `request.json` 属性。

2.3.3 来一块 Cookie

HTTP 是无状态（stateless）协议。也就是说，在一次请求响应结束后，服务器不会留下任何关于对方状态的信息。但是对于某些 Web 程序来说，客户端的某些信息又必须被记住，比如用户的登录状态，这样才可以根据用户的状态来返回不同的响应。为了解决这类问题，就有了 Cookie 技术。Cookie 技术通过在请求和响应报文中添加 Cookie 数据来保存客户端的状态信息。

 附注 Cookie 指 Web 服务器为了存储某些数据（比如用户信息）而保存在浏览器上的小型文本数据。浏览器会在一定时间内保存它，并在下一次向同一个服务器发送请求时附带这些数据。Cookie 通常被用来进行用户会话管理（比如登录状态），保存用户的个性化信息（比如语言偏好，视频上次播放的位置，网站主题选项等）以及记录和收集用户浏览数据以用来分析用户行为等。

在 Flask 中，如果想要在响应中添加一个 cookie，最方便的方法是使用 `Response` 类提供的 `set_cookie()` 方法。要使用这个方法，我们需要先使用 `make_response()` 方法手动生成一个响应对象，传入响应主体作为参数。这个响应对象默认实例化内置的 `Response` 类。表 2-10 是内置的 `Response` 类常用的属性和方法。

表 2-10 Response 类的常用属性和方法

方法 / 属性	说 明
headers	一个 Werkzeug 的 Headers 对象，表示响应首部，可以像字典一样操作
status	状态码，文本类型
status_code	状态码，整型
mimetype	MIME 类型（仅包括内容类型部分）
set_cookie()	用来设置一个 cookie

 附注 除了表 2-10 中列出的方法和属性外，Response 类同样拥有和 Request 类相同的 get_json() 方法、is_json() 方法以及 json 属性。

set_cookie() 方法支持多个参数来设置 Cookie 的选项，如表 2-11 所示。

表 2-11 set_cookie() 方法的参数

属 性	说 明
key	cookie 的键（名称）
value	cookie 的值
max_age	cookie 被保存的时间数，单位为秒；默认在用户会话结束（即关闭浏览器）时过期
expires	具体的过期时间，一个 datetime 对象或 UNIX 时间戳
path	限制 cookie 只在给定的路径可用，默认为整个域名
domain	设置 cookie 可用的域名
secure	如果设为 True，只有通过 HTTPS 才可以使用
httponly	如果设为 True，禁止客户端 JavaScript 获取 cookie

set_cookie 视图用来设置 cookie，它会将 URL 中的 name 变量的值设置到名为 name 的 cookie 里，如代码清单 2-4 所示。

代码清单 2-4 http/app.py：设置 cookie

```
from flask import Flask, make_response
...
@app.route('/set/<name>')
def set_cookie(name):
    response = make_response(redirect(url_for('hello')))
    response.set_cookie('name', name)
    return response
```

在这个 make_response() 函数中，我们传入的是使用 redirect() 函数生成的重定向响应。set_cookie 视图会在生成的响应报文首部中创建一个 Set-Cookie 字段，即“Set-Cookie: name=Grey; Path=/”。

现在我们查看浏览器中的 Cookie，就会看到多了一块名为 name 的 cookie，其值为我们设

置的“Grey”，如图 2-10 所示。因为过期时间使用默认值，所以会在浏览会话结束时（关闭浏览器）过期。

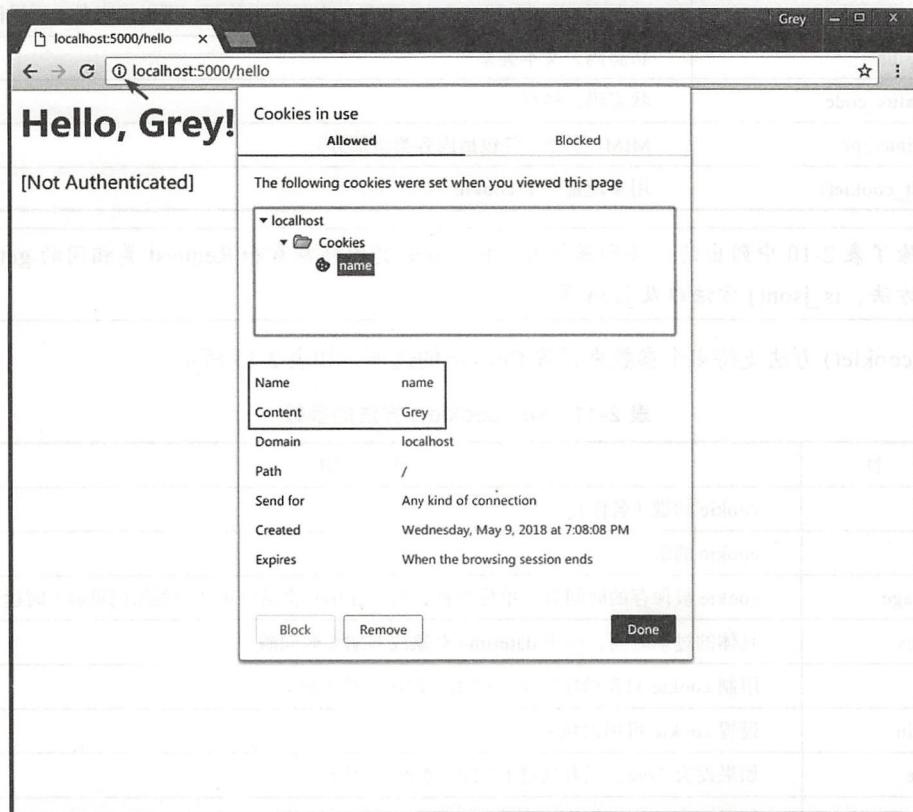


图 2-10 在浏览器中查看 cookie

当浏览器保存了服务器端设置的 Cookie 后，浏览器再次发送到该服务器的请求会自动携带设置的 Cookie 信息，Cookie 的内容存储在请求首部的 Cookie 字段中，整个交互过程由上到下如图 2-11 所示。

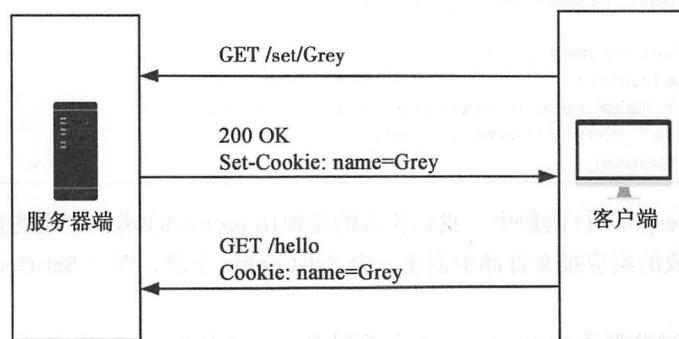


图 2-11 Cookie 设置示意图

在 Flask 中，Cookie 可以通过请求对象的 cookies 属性读取。在修改后的 hello 视图中，如果没有从查询参数中获取到 name 的值，就从 Cookie 中寻找：

```
from flask import Flask, request

@app.route('/')
@app.route('/hello')
def hello():
    name = request.args.get('name')
    if name is None:
        name = request.cookies.get('name', 'Human') # 从Cookie中获取name值
    return '<h1>Hello, %s</h1>' % name
```



注意 这个示例函数同样包含安全漏洞，后面会详细介绍。

这时服务器就可以根据 Cookie 的内容来获得客户端的状态信息，并根据状态返回不同的响应。如果你访问 `http://localhost:5000/set/Grey`，那么就会将名为 name 的 cookie 设为 Grey，重定向到 /hello 后，你会发现返回的内容变成了“Hello, Grey!”。如果你再次通过访问 `http://localhost:5000/set/<name>` 修改 name cookie 的值，那么重定向后的页面返回的内容也会随之改变。

2.3.4 session：安全的 Cookie

Cookie 在 Web 程序中发挥了很大的作用，其中最重要的功能是存储用户的认证信息。我们先来看看基于浏览器的用户认证是如何实现的。当我们使用浏览器登录某个社交网站时，会在登录表单中填写用户名和密码，单击登录按钮后，这会向服务器发送一个包含认证数据的请求。服务器接收请求后会查找对应的账户，然后验证密码是否匹配，如果匹配，就在返回的响应中设置一个 cookie，比如，“`login_user: greyli`”。

响应被浏览器接收后，cookie 会被保存在浏览器中。当用户再次向这个服务器发送请求时，根据请求附带的 Cookie 字段中的内容，服务器上的程序就可以判断用户的认证状态，并识别出用户。

但是这会带来一个问题，在浏览器中手动添加和修改 Cookie 是很容易的事，仅仅通过浏览器插件就可以实现。所以，如果直接把认证信息以明文的方式存储在 Cookie 里，那么恶意用户就可以通过伪造 cookie 的内容来获得对网站的权限，冒用别人的账户。为了避免这个问题，我们需要对敏感的 Cookie 内容进行加密。方便的是，Flask 提供了 session 对象用来将 Cookie 数据加密储存。



附注 在编程中，session 指用户会话（user session），又称为对话（dialogue），即服务器和客户端 / 浏览器之间或桌面程序和用户之间建立的交互活动。在 Flask 中，session 对象用来加密 Cookie。默认情况下，它会把数据存储在浏览器上一个名为 session 的 cookie 里。