# Flask Web 开发实战：入门、进阶与原理解析 (李辉) (Z-Library)

# 初识Flask

## 搭建开发环境

Flask 是使用Python 编写的Web 微框架。Web 框架可以让我们不用关心底层的请求响应处理， 更方便高效地编写Web 程序。因为Flask 核心简单且易千扩展，所以被称作微框架(microframework)。

Flask 有两个主要依赖， 一个是WSGI (Web Server Gateway Interface, Web 服务器网关接口）工具集一Werkzeug (http://werkzeug.pocoo.org/ ），另一个是Jinja2 模板引擎( http: //jinjapocoo.org/ ） 。

打开命令行窗口，切换到合适的目录， 然后使用下面的命令把本书的示例程序仓库复制到本地，并切换进项目根目录：

$ git clone https : //github . com/greyli / helloflask .git

$ cd helloflask

或者在HelloFlask 的GItHub 页面（ https.I/gIthubcom/greyh/helloflask) 单击了Fork 按

丁钮，那么可以使用你自己的GitHub 用户名未替换掉上面的greyli, 这将复制一份派生仓

库，你可以自由地修改和提交代码。

### Pipenv工作流

Pipenv 是基于pip 的Python 包管理工具，它和pip 的用法非常相似，可以看作pip 的加强

版，它的出现解决了旧的pip+virtual en v+requ i rem en ts. txt 的工作方式的弊端。具体来说，它是pip 、和pfile 和Virtualenv 的结合体，它让包安装、包依赖管理和虚拟环境管理更加方便，使用它可以实现高效的Python 项目开发工作流。

1. **安装pip和pipenv**

查看pip是否安装 pip --version

安装指定的包 pip install 包名称

使用pip安装pipenv pip install pipenv

查看pipenv是否安装 pipenv --version

1. **创建虚拟环境**

在Python 中，虚拟环境(virtual enviroment) 就是隔离的Python 解释器环境。通过创建虚拟环境，你可以拥有一个独立的Python 解释器环境。这样做的好处是可以为每一个项目创建独立的Python 解释器环境，因为不同的项目常常会依赖不同版本的库或Python 版本。使用虚拟环境可以保待全局Python 解释器环境的干净，避免包和版本的混乱，并且可以方便地区分和记录每个项目的依赖，以便在新环境下复现依赖环境。

虚拟环境通常使用Virtualenv 来创建，但是为了更方便地管理虚拟环境和依赖包，我们将会使用集成了Virtualenv 的Pipenv 。

确保我们当前工作目录在示例程序项目的根目录，即helloflask 文件夹中， 然后使用pipenv insta ll 命令为当前的项目创建虚拟环境：

为当前项目创建虚拟环境 pipenv install

这会为当前项目创建一个文件夹，其中包含隔离的Python 解释器环境，并且安装pip 、

wheel 、setuptools 等基本的包

默认情况下，和penv 会统一管理所有虚拟环境。在Win!o\_ws 系统中，虚拟环境文件夫会在C: \U sers\Administrator\. virtualenvs\ 目录下创建，而L i nux 或macOS 会在～/ .local/share/virtual envs/ 目录下创建。如果你想在项目目录内创建虚拟环境文件夫，可以设置环境变量PIPENV\_ VENV IN\_PROJECT, 这时名为．venv 的虚拟环境文件夹将在项目根目录被创建。

虚拟环境文件夹的目录名称的形式为“当前项目目录名＋ 一串随机字符＂，比如helloflask-5PaOZfZw

显式激活当前项目的虚拟环境 pipenv shell

不显式激活虚拟环境时，执行命令 pipenv run python hello.py

1. **管理依赖**

在创建虚拟环境时，如果项目根目录下没有和pfile 文件， pip env install 命令还会在项目文件夹根目录下创建和pfile 和Pipfile.lock 文件，前者用来记录项目依赖包列表，而后者记录了固定版本的详细依赖包列表。当我们使用Pipenv 安装／删除／更新依赖包时， 和pfile 以及Pipfile.lock 会自动更新。

查看当前环境下的依赖情况 pipenv graph

在虚拟环境中查看依赖情况 pip list

### 安装Flask

在创建好的虚拟环境中安装Flask pipenv install flask

Pipenv 会自动帮我们管理虚拟环境，所以在执行pip env in s ta ll 安装Pytho n 包时，无论是否激活虚拟环境，包都会安装到虚拟环境中。后面我们都将使用和p en v 安装包，这相当于在激活虚拟环境的情况下使用pip 安装包。只有需要在全局环境下安装／ 更新／ 删除包，我们才会使用pip

更新包 pipenv update flask(不写包名称 则更新该环境中的所有包)

### 集成开发环境(Pycharm)

## Hello,Flask!

切换到示例程序所在目录 cd demo/flask(目录路径)

对于简单的程序来说，程序的主模块一般命令为app.py 。你也可以使用其他名称，比如丁hello.py , 但是要避免使用flask.py, 因为这和F lask 本身冲突。

### 创建程序实例

从flask保重导入Flask类，并实例化为app

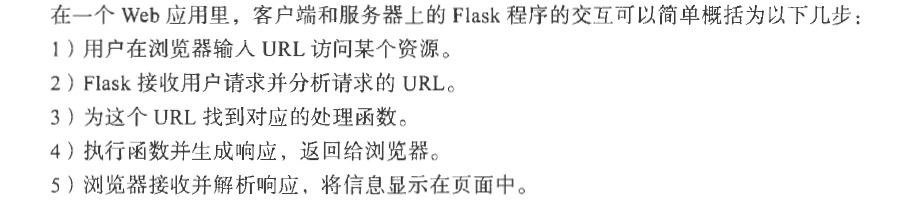
from flask import Flask

app = Flask ( \_ \_name\_ \_ )

传入Flask 类构造方法的第一个参数是模块或包的名称，我们应该使用特殊变量—name—°

Python 会根据所处的模块来赋予—name— 变量相应的值，对千我们的程序来说(app .py) ，这个值为app 。除此之外，这也会帮助Flask 在相应的文件夹里找到需要的资源，比如模板和静态文件。

### 注册路由



在上面这些步骤中，大部分都由F lask 完成，我们要做的只是建立处理请求的函数，并为其

定义对应的URL 规则。只需为函数附加app.route(） 装饰器，并传入URL 规则作为参数，我们就可以让URL 与函数建立关联。这个过程我们称为**注册路由(route)** ，

路由负责管理URL 和函数之间的映射， 而这个函数则被称为视图函数(view function) 。

（注册路由：）定义视图函数，并将其和装饰器绑定起来；

@app.route(‘/’) #绑定装饰器

def index() : #视图函数

return'<hl>Hello, World! </ hl> '

route(） 装饰器的第一个参数是URL 规则，用字符串表示，必须以针杠（/) 开始。这里的URL 是相对URL （又称为内部URL) ，即不包含域名的URL 。以域名www.belloflaskcom 为例， “ /" 对应的是根地址（ 即www.hellofl ask . com ) ，如果把URL 规则改为“ /hello " ，则实际的绝对地址（外部地址）是www.belloflask.com/bello 。

1. **为视图函数绑定多个URL**

一个视图函数可以绑定多个URL, 比如下面的代码把／hi 和/he ll o 都绑定到say\_h e llo(） 函数上，这就会为say\_hello 视图注册两个路由，用户访问这两个URL 均会触发say\_hello() 函数，获得相同的响应。

@app.route('/hi')

@app.route('/hello')

def say\_hello():

return '<h1>Hello, Flask!</h1>'

1. **动态URL**

我们不仅可以为视图函数绑定多个URL, 还可以在URL 规则中添加变盐部分，使用＂ ＜变

量名＞＂ 的形式表示。F l ask 处理请求时会把变量传入视图函数，所以我们可以添加参数获取这个变最值。代码清单1-3 中的视图函数greet(），它的URL 规则包含一个name 变量。

@app.route('/greet', defaults={'name': 'Programmer'}) # 设置默认变量名

@app.route('/greet/<name>')

def greet(name):

return '<h1>Hello, %s!</h1>' % name

## 启动开发服务器

Flask 内置了一个简单的开发服务器（由依赖包Werkzeug 提供），足够在开发和测试阶段

使用。

### Run flask!

Flask 通过依赖包Click 内置了一个CLI ( Command Line Interface , 命令行交互界面）系统。

当我们安装Flask 后，会自动添加一个flask 命令脚本， 我们可以通过flask 命令执行内置命令、扩展提供的命令或是我们自己定义的命令

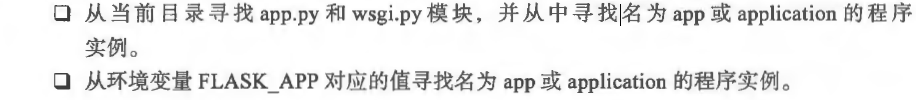
启动内置的开.服务器 flask run

确保执行命令前激活了虚拟环境~ pipenv shell ) ，否则需要使用pipenv run flask run 命令启动开发服务器

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：



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掀FLASK\_APP, 将包含程序实例的模块名赋值给这个变量。

在Wi nd ows 系统中使用set 命令设置环境变量：

set FLASK APP=hello

1. **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。



安装环境变量管理工具到虚拟环境中 pipenv install python-dotenv

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.fl askenv 或e nv 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

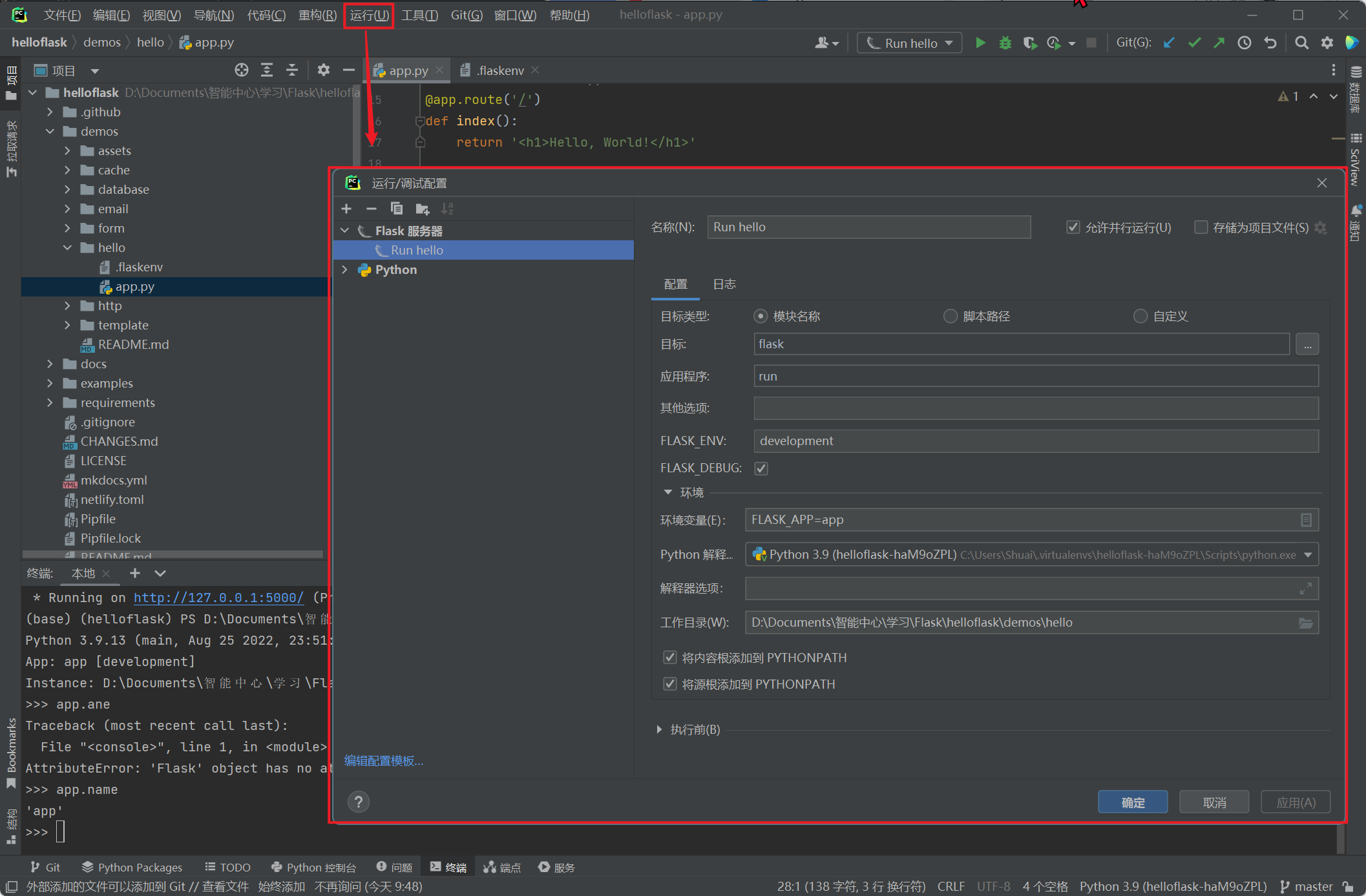
＃这是注释

FOO= "BAR"

.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G it 忽略这个文件。gitignore 文件是一个名为.gitign ore 的文本文件，它存储了项目中G it 提交时的忽略文件规则清单。

Pyth on 项目的g 山gnore 模板可以参考http s: //g ithub .co m/github/g 山gnore/blob/mas ter/Python .g 山gnore 。使用PyC harm 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.gi tignore. io/ api /p yc harm 。

1. **使用Pycharm运行服务器**



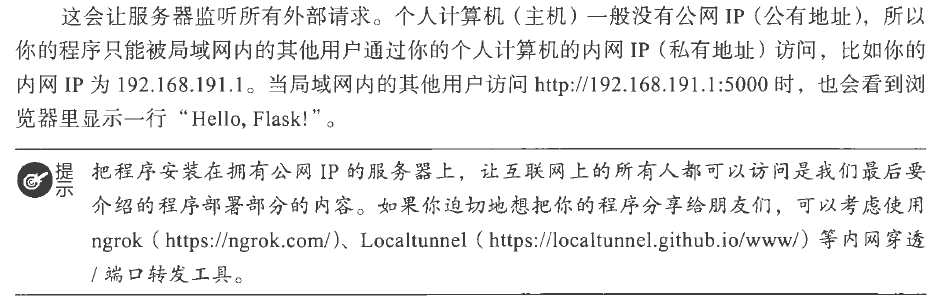


### 其他的启动设置

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：

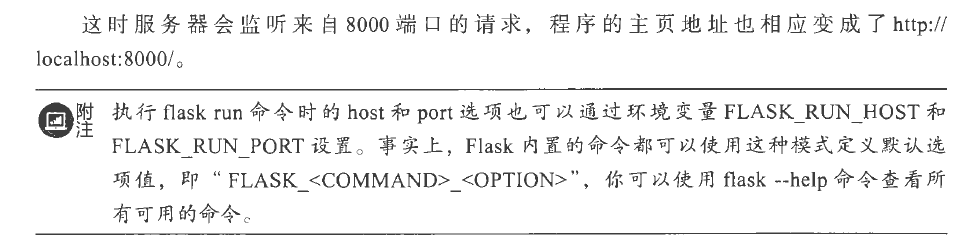
flask run - - host==0.0.0.0



1. **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：

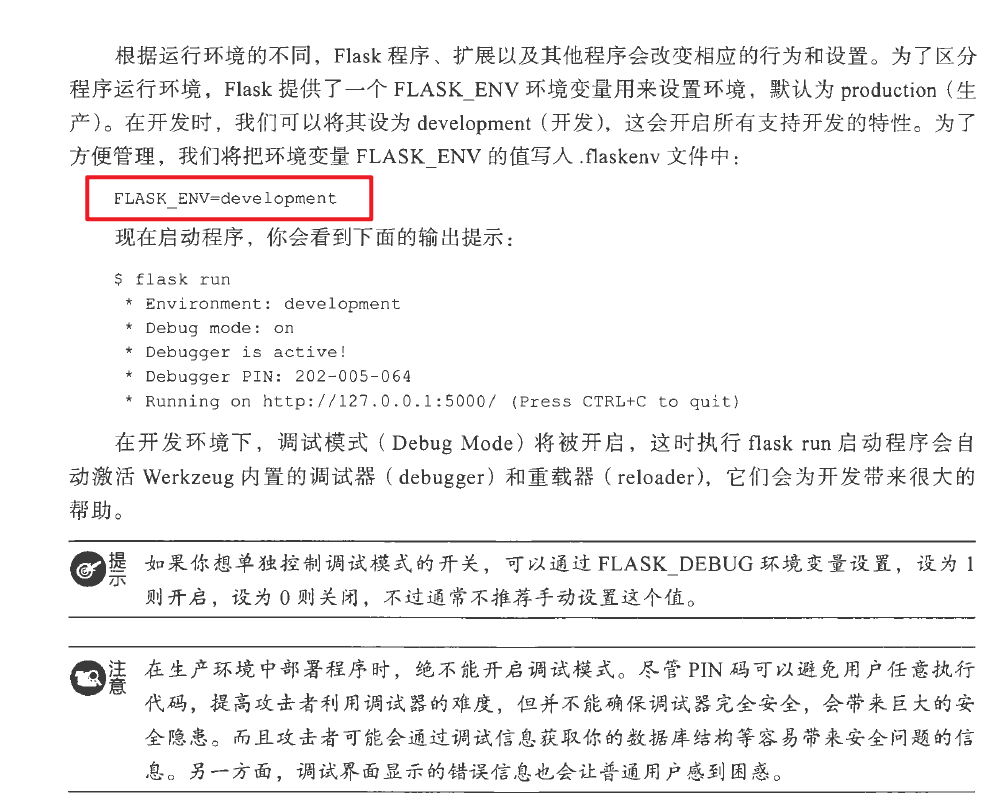
flask run --port=8000



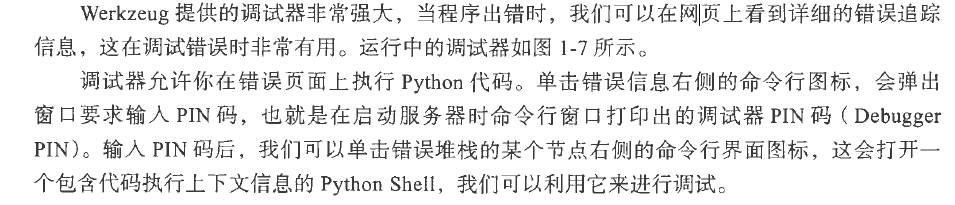
### 设置运行环境

**开发环境**( development e nviroment ) 和**生产环境**(production enviroment ) 是我们后面会频

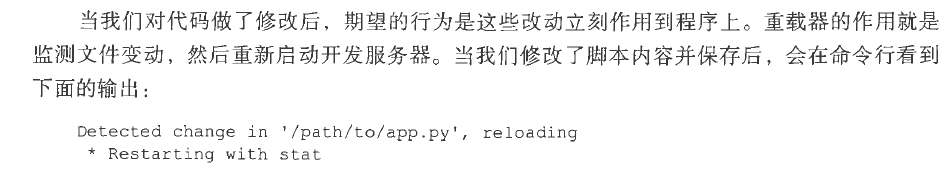
繁接触到的概念。开发环境是指我们在本地编写和测试程序时的计算机环境，而生产环境与开发环境相对，它指的是网站部署上线供用户访问时的服务器环境。

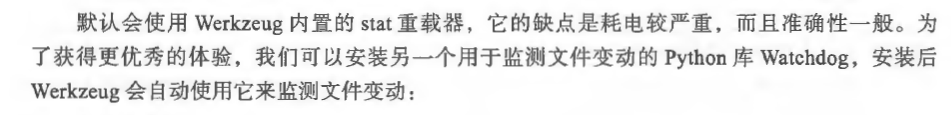


1. **调试器**

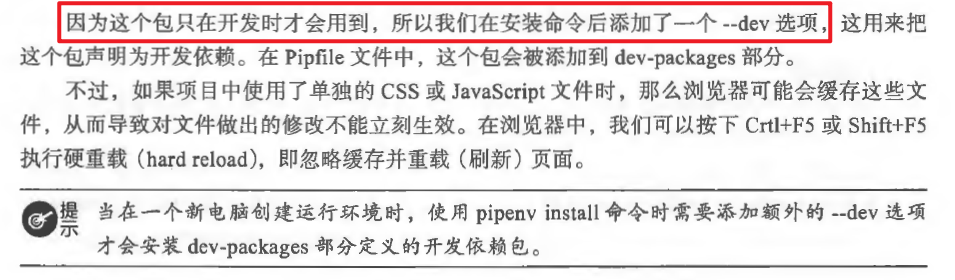


1. **重载器**

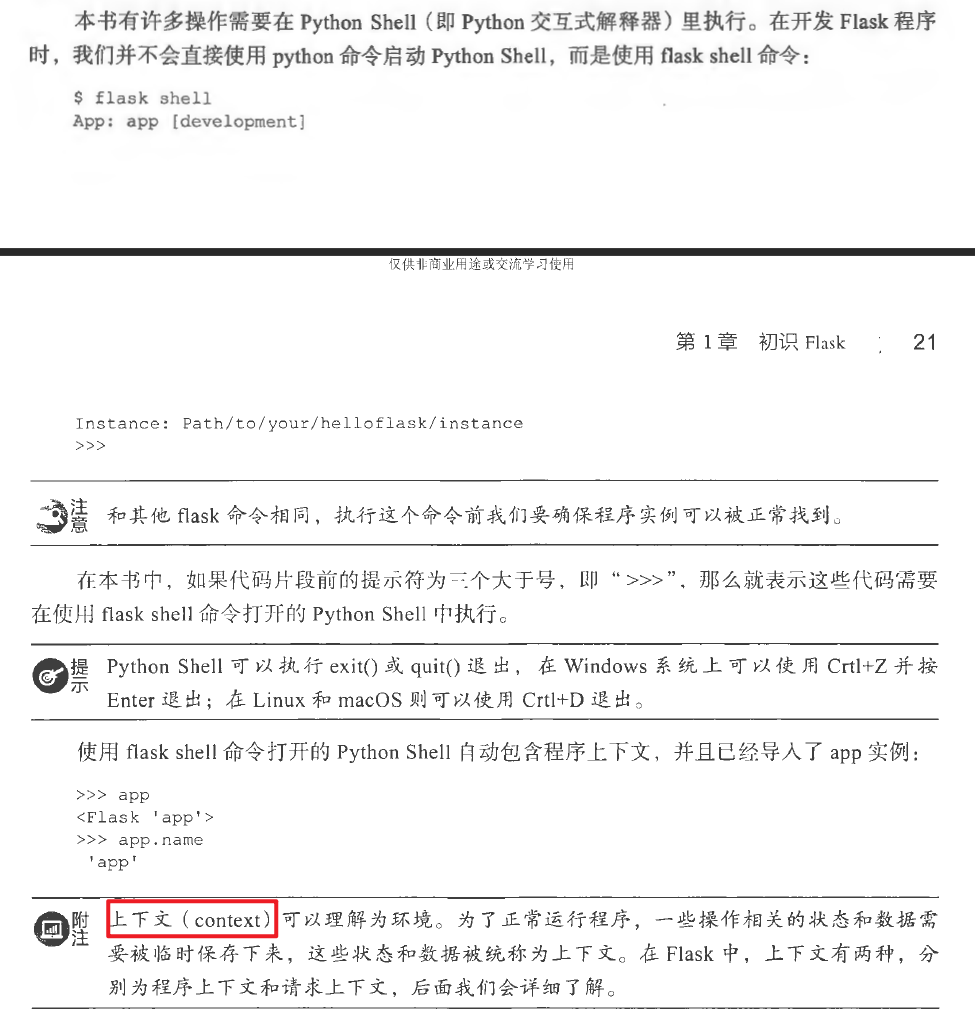




Pipenv install watchdog --dev



## Python Shell



## Flask扩展

**扩展( extension )** 即使用Flask 提供的API 接口编写的Python 库，可以为F lask 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F l ask 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)

## 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F lask 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。

Flask 提供了很多种方式来加载配置。

1. **设置单个配置**

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 . conf ig [ ' ADM IN\_ NAME'] ='Peter '

1. **设置多个配置**

使用update()方法则可以一次加载多个值：

app . config.update(

TESTING=True ,

SECRET\_KEY='\_S#yF4Q8z\n\xec]/'

)

1. **通过文件导入配置**

除此之外，你还可以把配置变擞存储在单独的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confi g 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 ' ]

## URL与端点

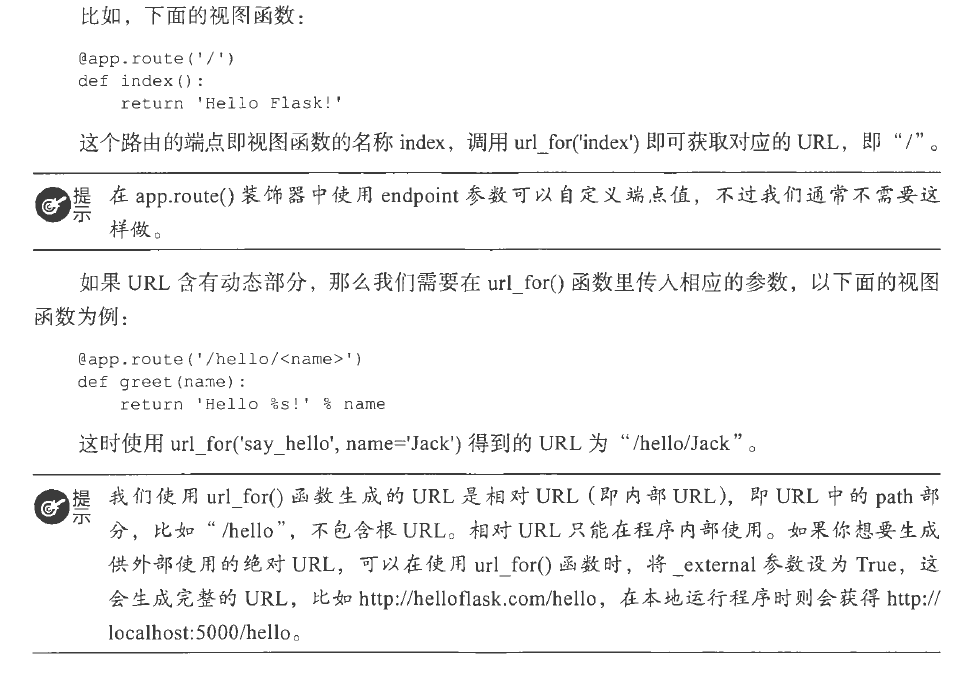
在Web 程序中，URL无处不在。如果程序中的URL 都是以硬编码的方式写出，那么将会

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ur l\_for(）函数时，第一个参数为**端点(endpoint) 值**。在Fl ask 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，至千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。



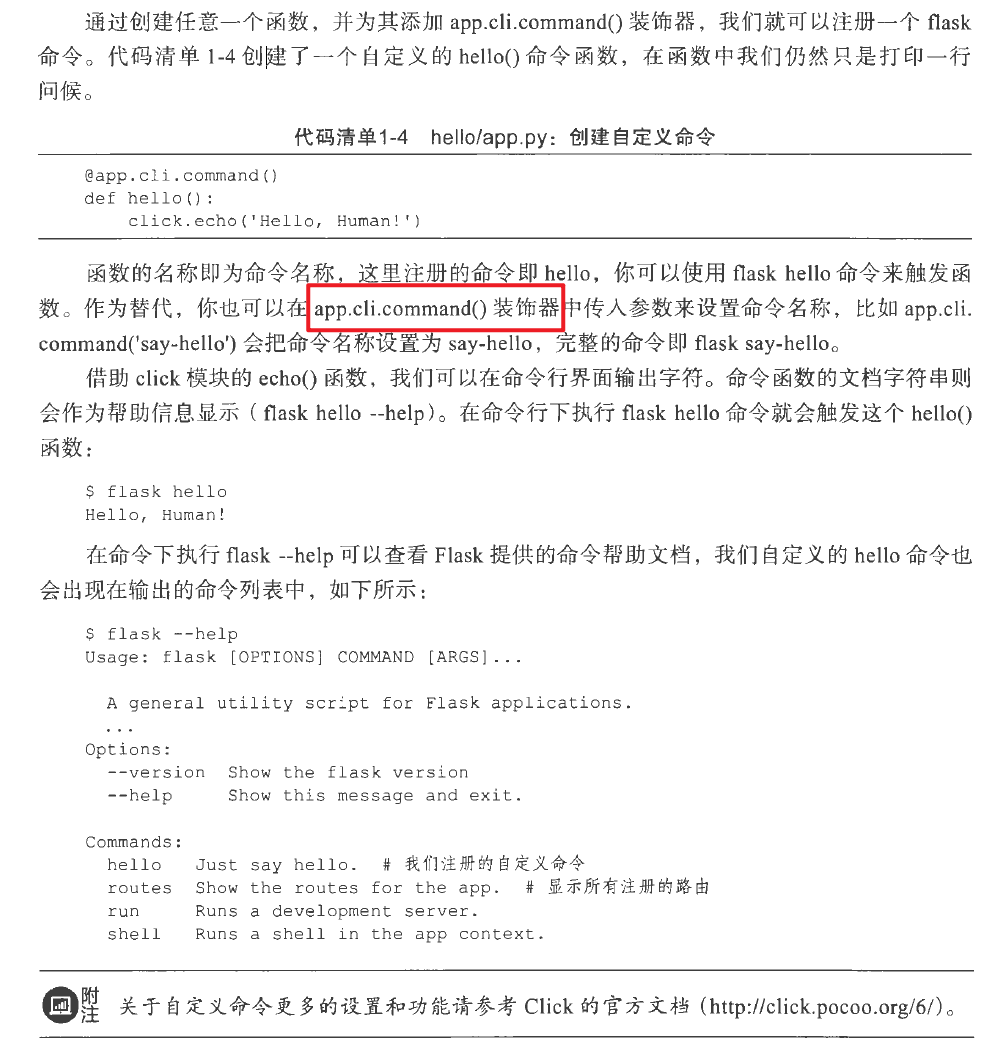
## Flask命令

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。



## 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“He llo , 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：

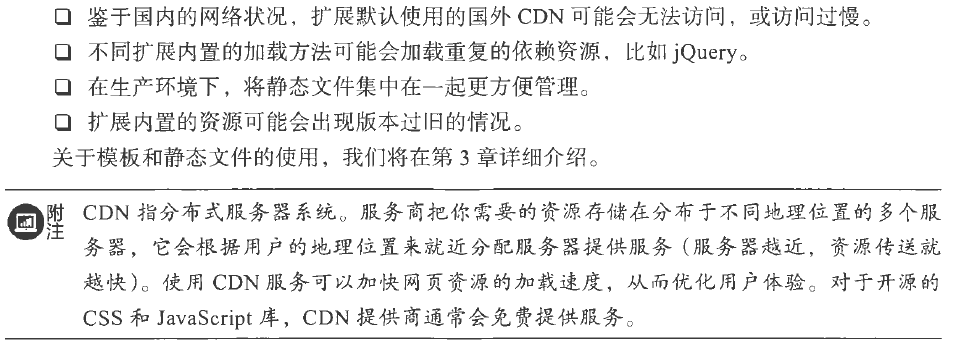


在开发Flask 程序时，使用CSS 框架和JavaScript 库是很常见的需求，而且有很多扩展都提供了对CSS 框架和JavaScript 库的集成功能。使用这些扩展时都需要加载对应的CSS 和

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l ink 标签和script 标签。这些方法一般会直接从C DN 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stat ic 目录

下，统一管理，出千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。



## Flask与MVC架构

MVC 架构最初是用来设计桌面程序的，后来也被用千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(Co ntro ll er Funct i on) 似乎更合适些，虽然它也附带处理用户界面。严格来说， F l ask 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SQLA !chemy 来创建数据库模型。

# Flask与HTTP

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。

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

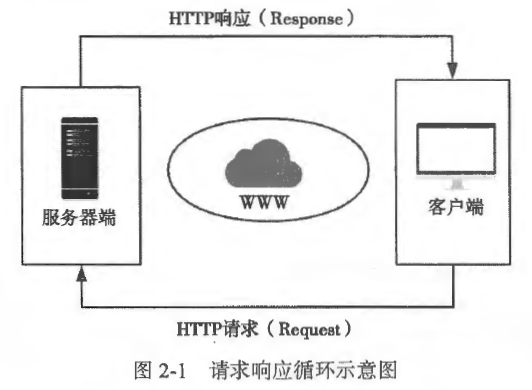
$ flask run

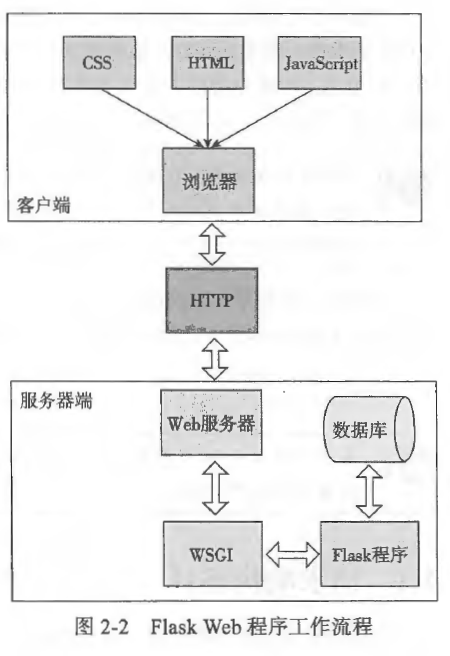
## 请求响应循环

ht tp: //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-ResponseCycle)"**: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。





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F lask 程序能够使用的Python 数据。在程序中， Flask 根据请求的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。

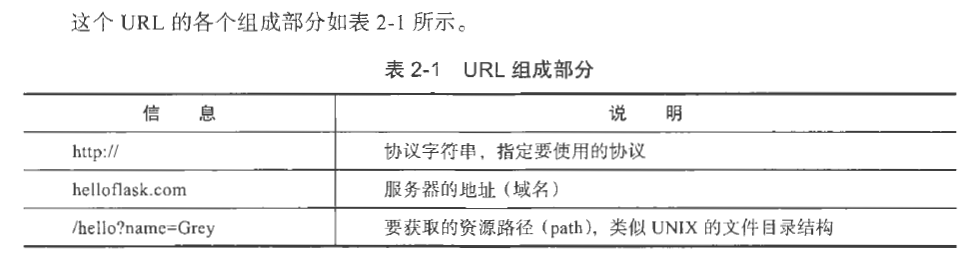
## HTTP请求

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 后面的？name~Grey 部分是查询字符串(query string) 。URL 中的查询宇符串

用未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。

### 请求报文

当我们在浏览器中访问这个URL时，随之产生的是一个发向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(requestmessage)，而服务器返回的数据被称为响应报文(responsemessage)。

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

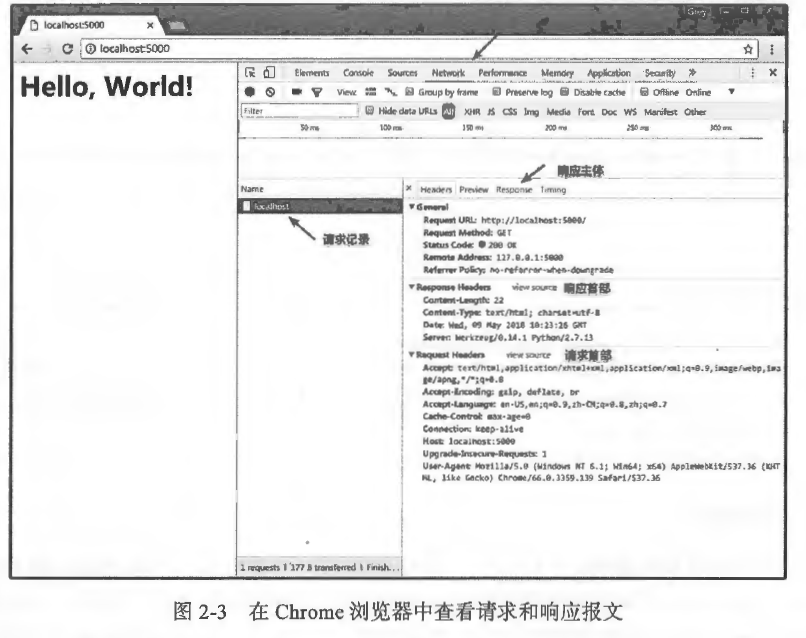
请求产生的请求报文示意如表2-2所示。



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(Fl2)里的Network标签中看到URL对应资源加载的所有请求列表，单

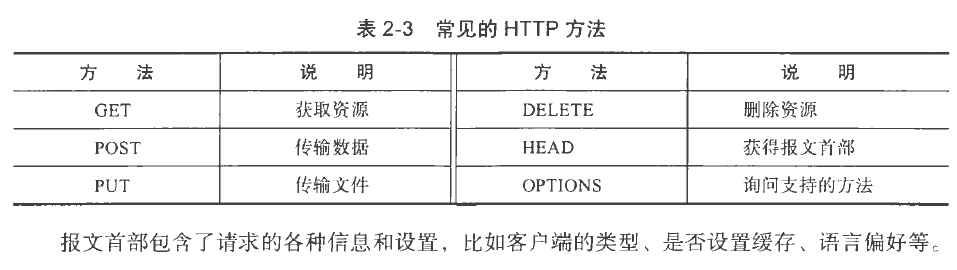
击任一个请求条目即可看到报文信息，图2-3是使用Chrome访问本地示例程序的示例。

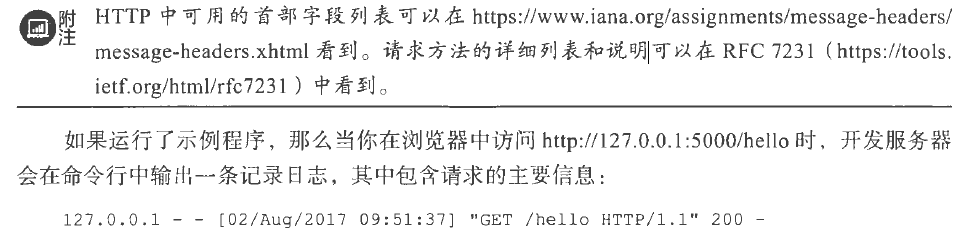


**报文**由报文首部和报文主体组成，两者由空行分隔，请求报文的主体一般为空。

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。





### 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，

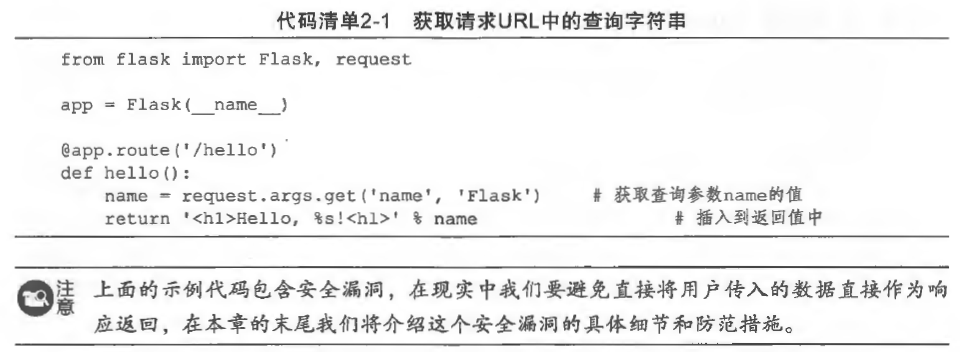
我们能从它获取请求报文中的所有数据。

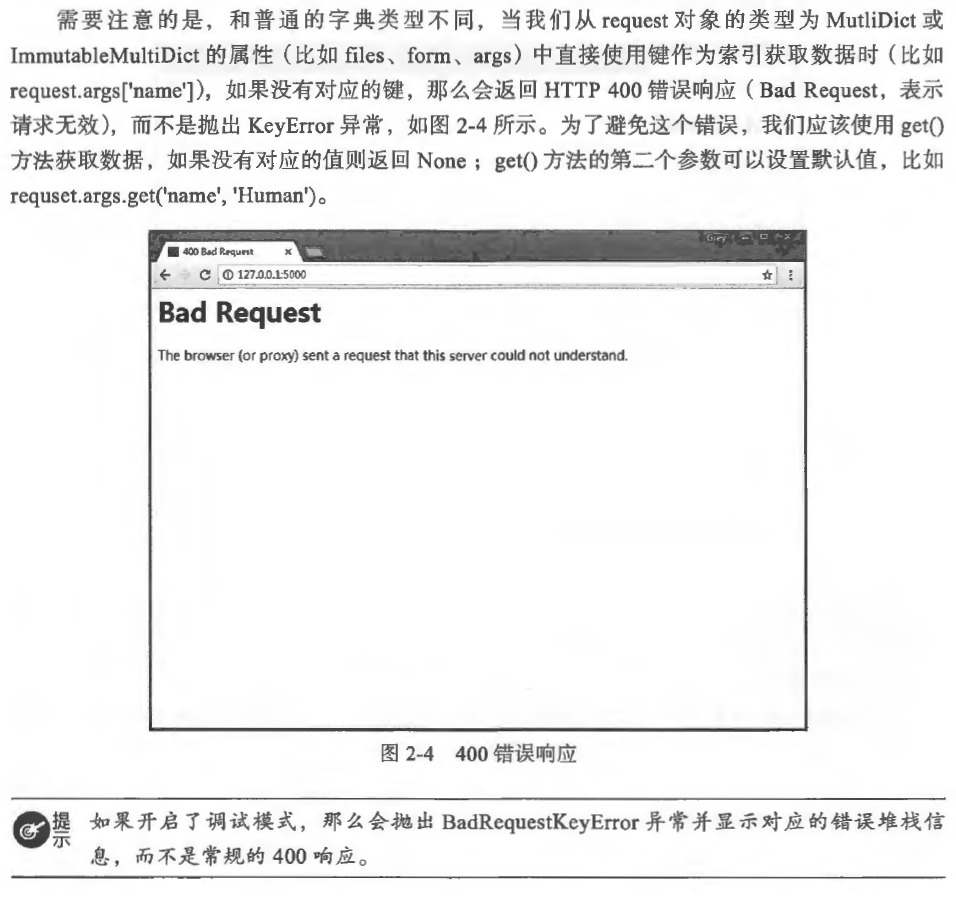




在我们的示例程序中实现了同样的功能。当你访问http://localhost:5000/hello?name=Grey

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





### 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。

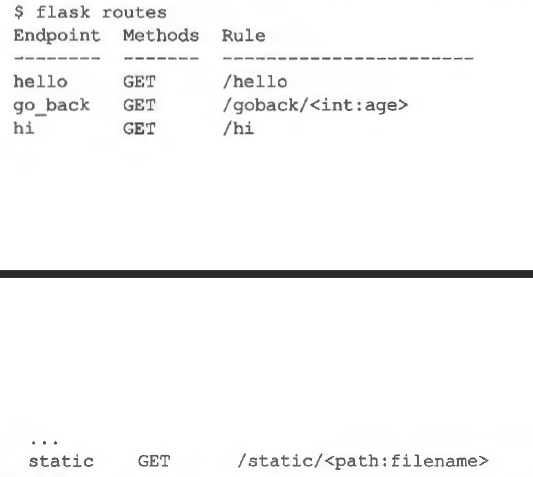
1. **路由匹配**

为了便千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(NotFound,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。



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：



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。

1. **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WebAPI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：

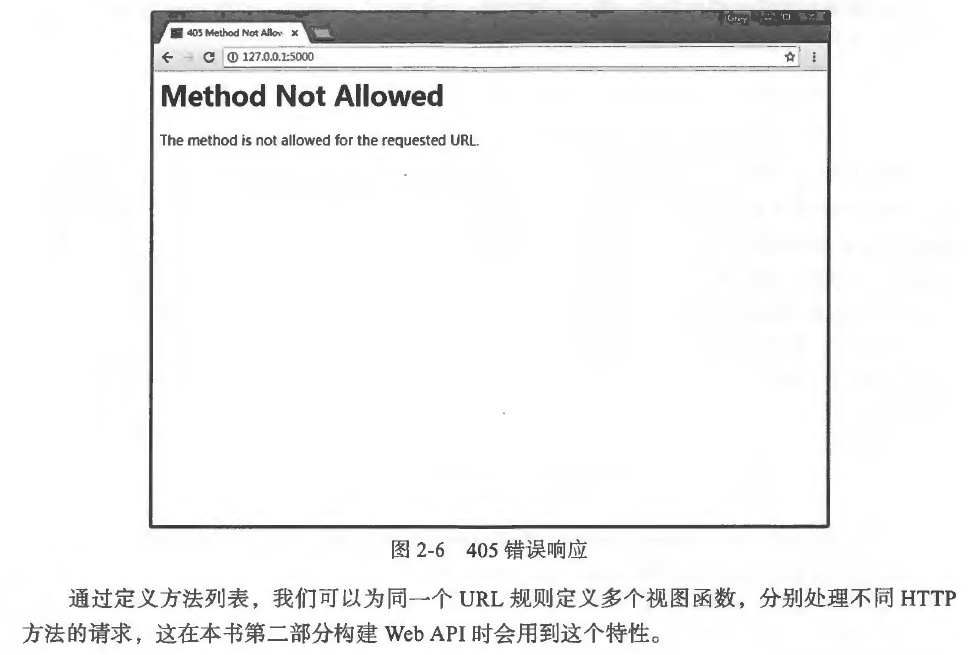
@app.route('/hello',methods=['GET','POST'])

def hello():

return '<hl>Hello,Flask!</hl>'

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



1. **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/fgoback/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：“Welcometo1984!”。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。



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 1</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变最会被转换成字符串，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中传入的变址不是数字，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(valuel, 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 proudor 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

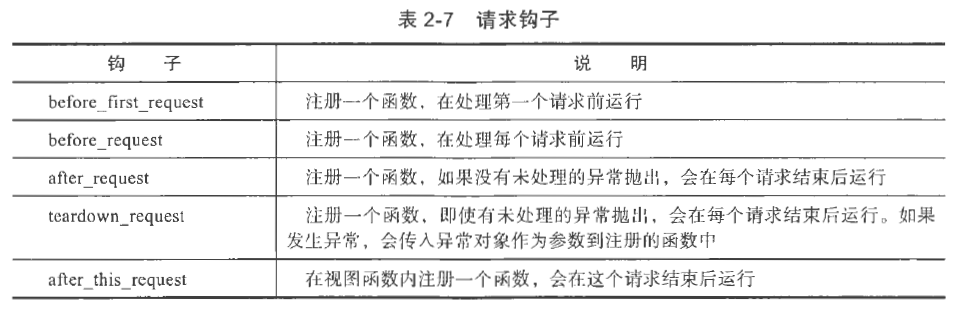
用％或是format(）函数）来构建URL 规则字符串，比如：

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

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

### 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。



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，

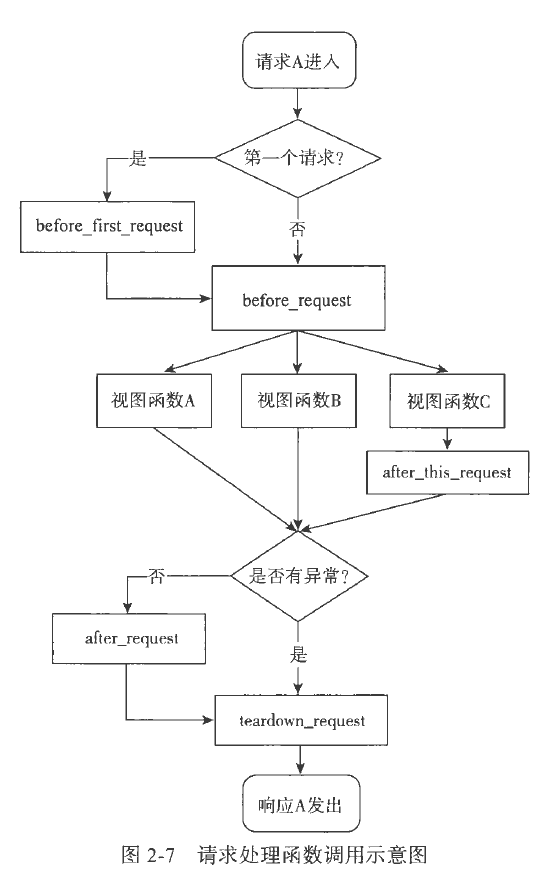
函数名并不是必须和钩子名称相同，下面是一个基本示例：

@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。



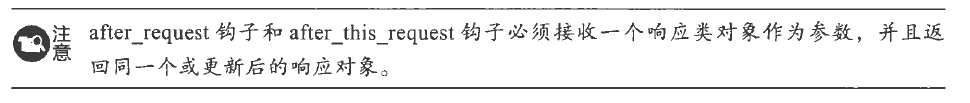
下面是请求钩子的一些常见应用场景：

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。

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。

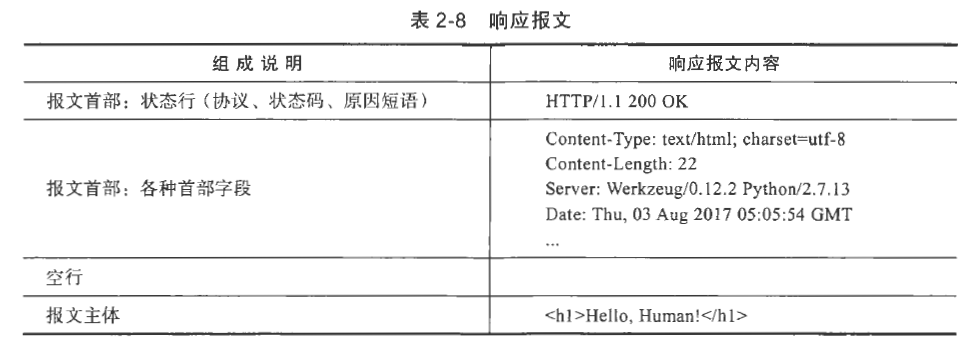


## HTTP响应

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。

### 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。



响应报文的首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。



### 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 ' <hl>Hello, Flask 1</hl> '

默认的状态码为200 , 下面指定了不同的状态码：

@app . route ( ' /hello ' )

def hello () :

retu r n ' <hl>Hello , Flask 1</hl> ' , 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 . rout e ( ' /hel lo ')

def hello() :

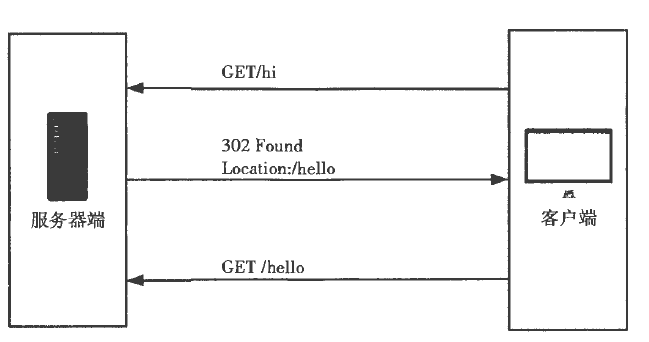
return ' ' , 302 , {' Location' ,' http: //www . example . com' }

现在访问http ://loc a lhos t:5000/h ello , 会重定向到http : //www.exampl e. 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，首部中衙要将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。



在We b 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。

对千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F lask 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：

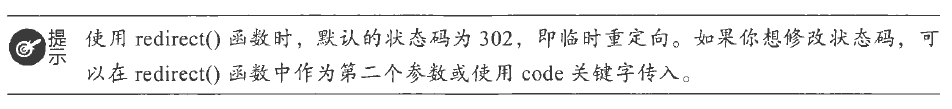
from flask import Flask, redirect

# ...

@app. route (' /hello ')

def hello():

return redirect ( ' http : //www . example . com ')



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。

from flask import Flask , redirect , url for

@app . route ( ' /hi ' )

def hi():

return redierct(url for( ' hello ' )) ＃重定向到／ hello

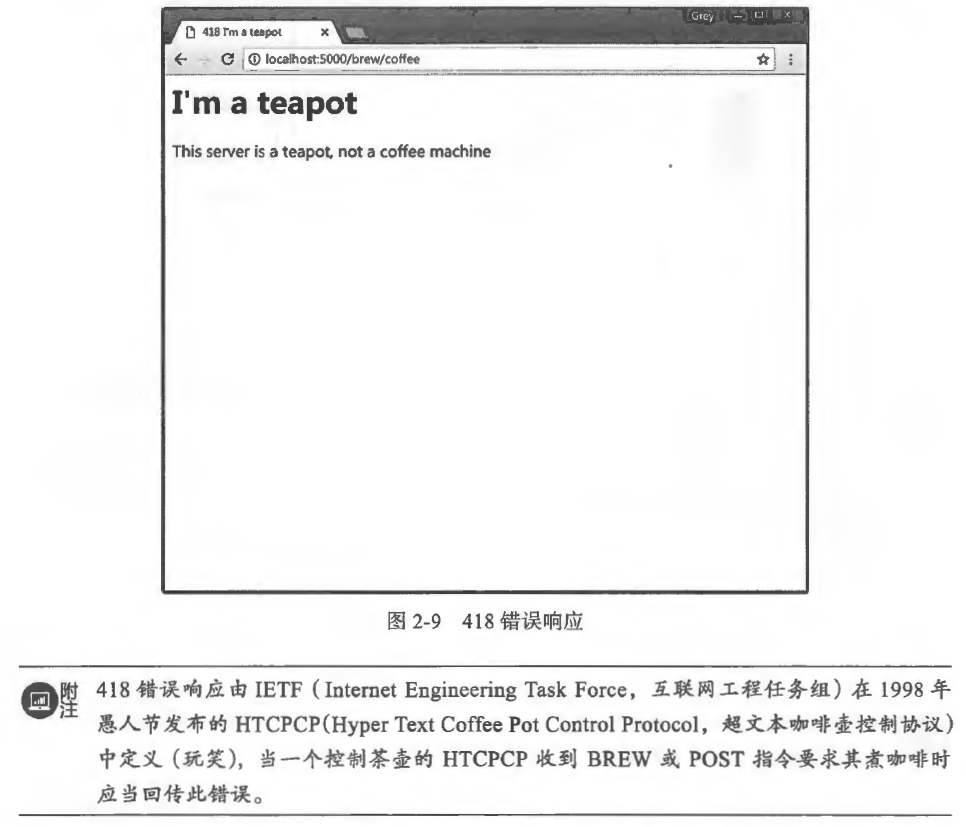
@app . route ( ' /hello ' )

def hello()

…

1. **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'mateapot)，如图2-9所示。



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。

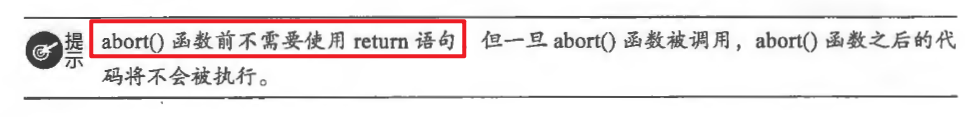
在abort()函数中传入状态码即可返回对应的错误响应，代码清单2- 3 中的视图函数返回404 错误响应。

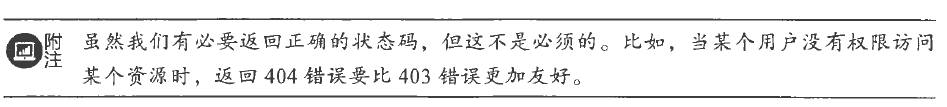
from flask import Flask, abort

@app . route( ' / 404 ' )

def not\_found():

abort(404)

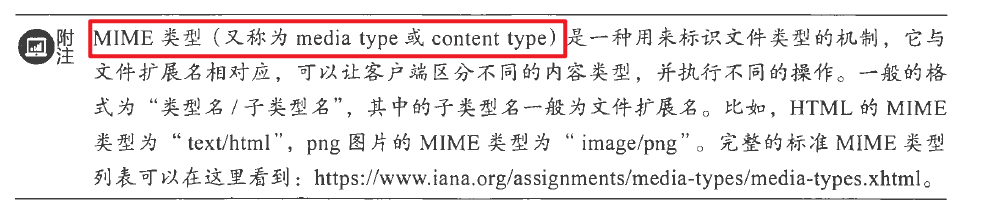




### 响应格式

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



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：

f r om flask i mport make\_response

@app . r oute (' / foo ' )

def foo () :

re sponse = make\_re sponse ( ' Hello, World!')

response .mimetype = ' text/ plai n '

retu rn 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m imetype 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：

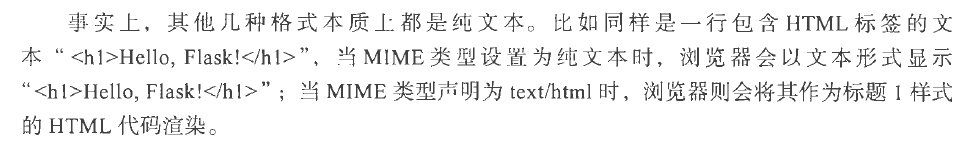
Note

to : Peter

f r om: Jane

heading : Reminder

body : Don ' t forge t the party '



1. **HTML**

**MIME 类型： text/html**

**示例：**

<!DOCTYPE html>

<html>

<head></head>

<body>

<hl>Note</hl>

<p>to : Peter</p>

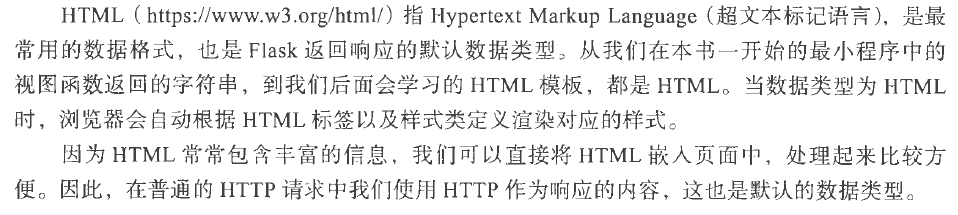
<p>from : Jane</p>

<p>heading: Reminder</p>

<p>body: <strong>Don ' t forget the party1</strong></p>

</body>

</html>



1. **XML**

**MIME 类型： application/x ml**

**示例：**

<?xml version=" l . 0" encoding="UTF-8 "?>

<no te>

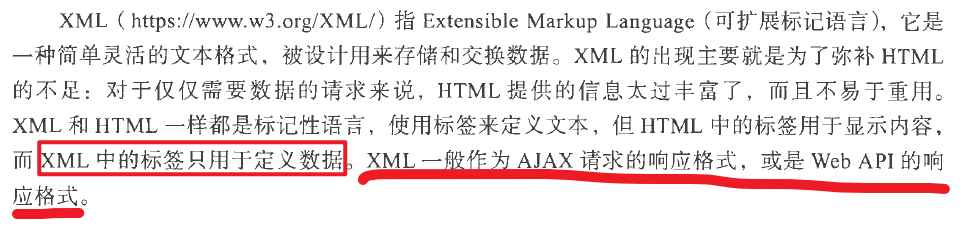
＜七o>Pe 七er</to>

<from>Jane</from>

<headi ng>Reminder</heading>

<body>Don ' t forget 七he party ' </body>

</note>



1. **JSON**

**MIME 类型： application/json**

**示例：**

{"note ": {

"to": "Peter",

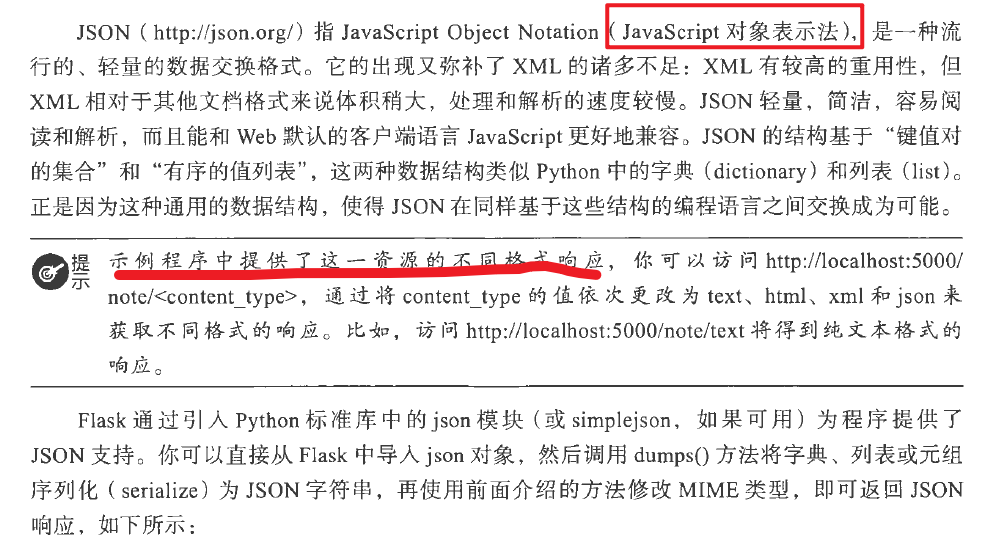
"from " : "Jane ",

"heading ":" Remider",

"body":" Don ' t forget the party '"

}

}



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 = ' applicat ion/json '

return response

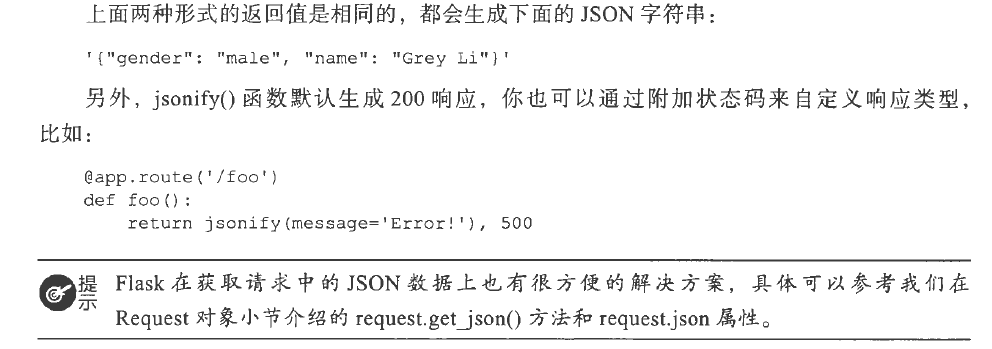
不过我们一般并不直接使用json 模块的dumps(） 、load(） 等方法，因为Flask 通过包装这些方法提供了更方便的jsonify(）函数。借助jsonify(）函数，我们仅需要传入数据或参数， 它会对我们传入的参数进行序列化，转换成JSON 字符串作为响应的主体，然后生成一个响应对象，并且设置正确的MIME 类型。使用jsonify 函数可以将前面的例子简化为这种形式：

from flask import jsonify

@app . route (' /foo ')

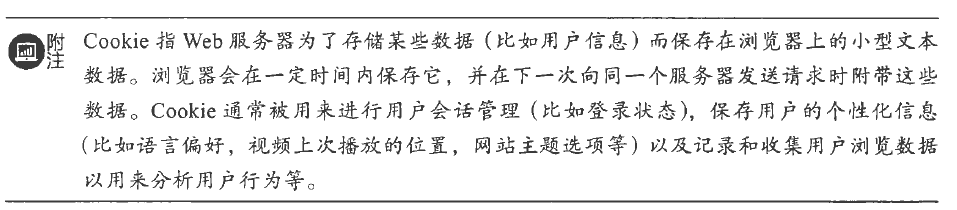
def foo () :

return jsonify(name= ' Grey Li', gender= ' male ')



### 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。



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Respon se 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。



from flask import Flask, make\_response

@app. route (' /set/ <name> ' )

def set\_cooki e(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=/"。