第 2 章 Flask 与 HTTP（部分内容试读）

在第 1 章，我们已经了解了 Flask 的基本知识，如果我们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。

附注 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
```

注意 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，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。

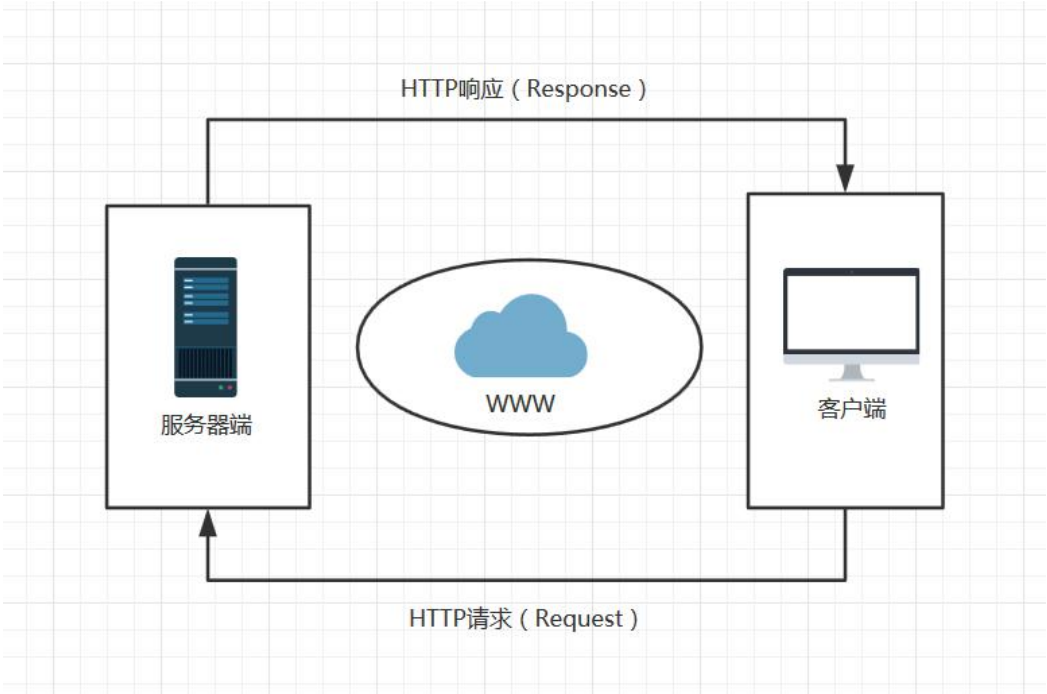

图 2-1 请求响应循环示意图

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。

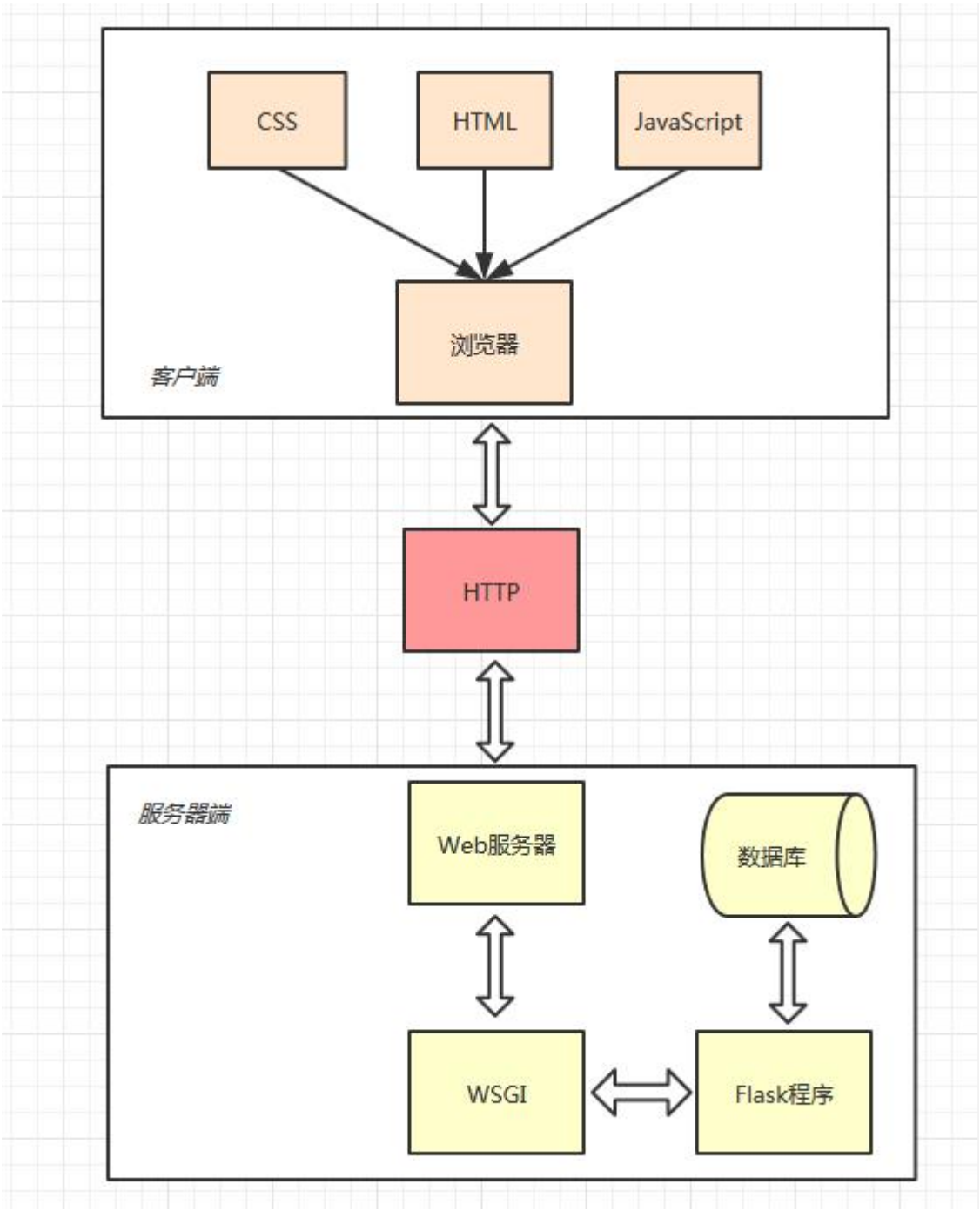

图 2-2 Flask Web 程序工作流程

从上图可以看出，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 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
http://	协议字符串，指定要使用的协议
helloflask.com	服务器的地址（域名）
/hello?name=Grey	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（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，点击任一个请求条目即可看到报文信息，图 2-3 是使用 Chrome 访问本地的示例程序的示例。

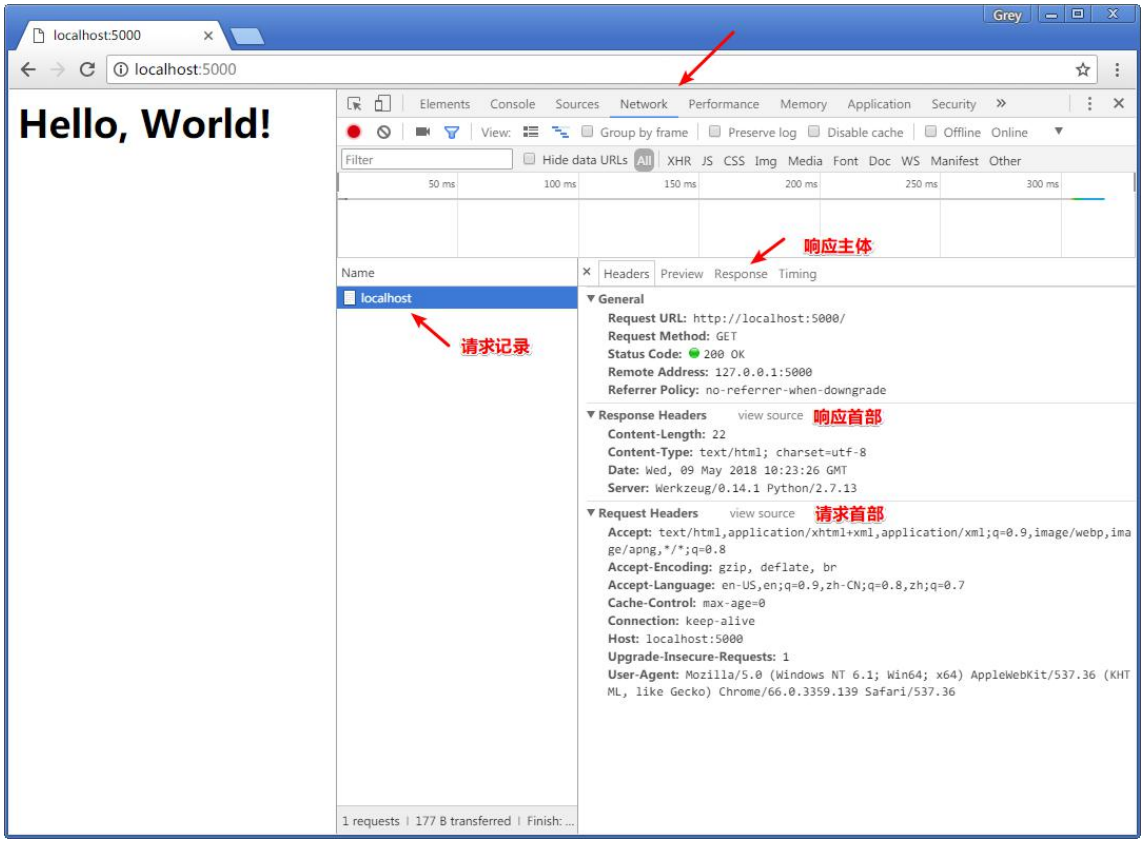

图 2-3 在 Chrome 浏览器中查看请求和响应报文

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
GET	获取资源
POST	传输数据
PUT	传输文件
DELETE	删除资源
HEAD	获得报文首部
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，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
path	u'/hello'
full_path	u'/hello?name=Grey'
host	u'helloflask.com'
host_url	u'http://helloflask.com/ '
base_url	u'http://helloflask.com/hello '
url	u'http://helloflask.com/hello?name=Grey '
url_root	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。类似 files，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 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

提示 Werkzeug 的 MutliDict 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 MutliDict 类，但其值不可更改。具体访问 Werkzeug 文档相关数据结构章节 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，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 对象中类型为 MutliDict 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 requset.args.get('name', 'Human')。

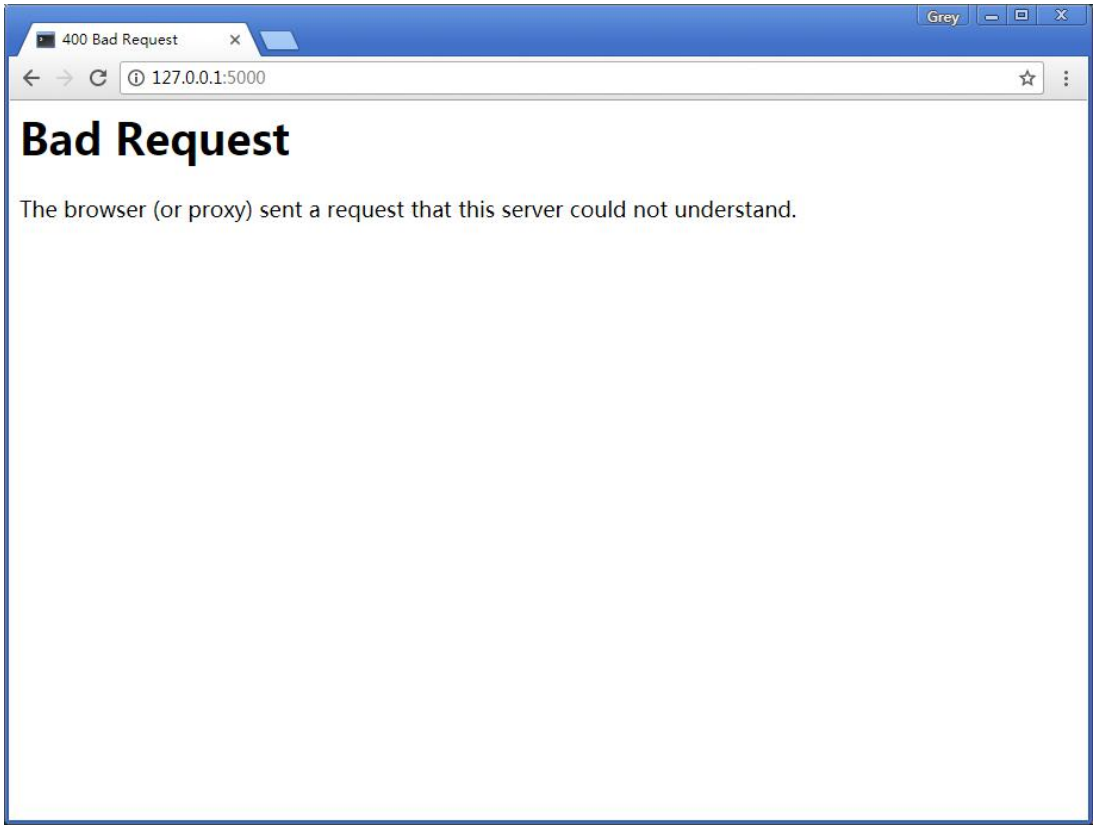

图 2-4 400 错误响应

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的 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。

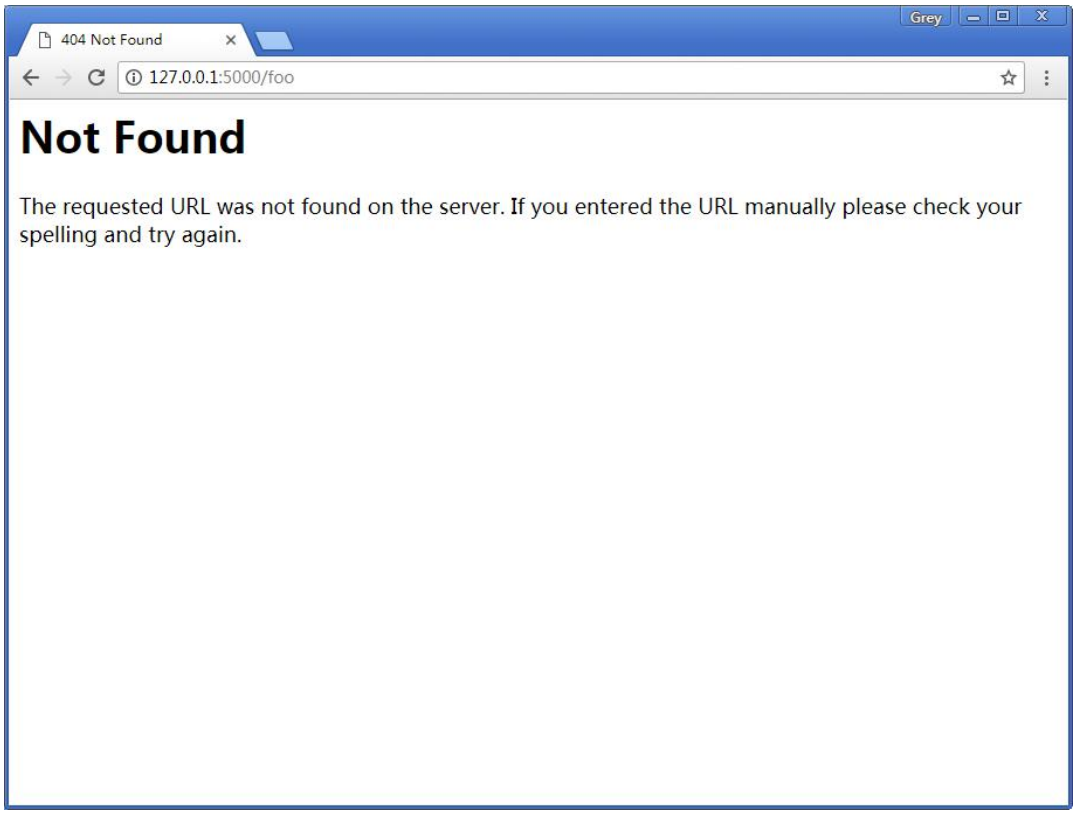

图 2-5 404 错误响应

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，所以视图函数的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。

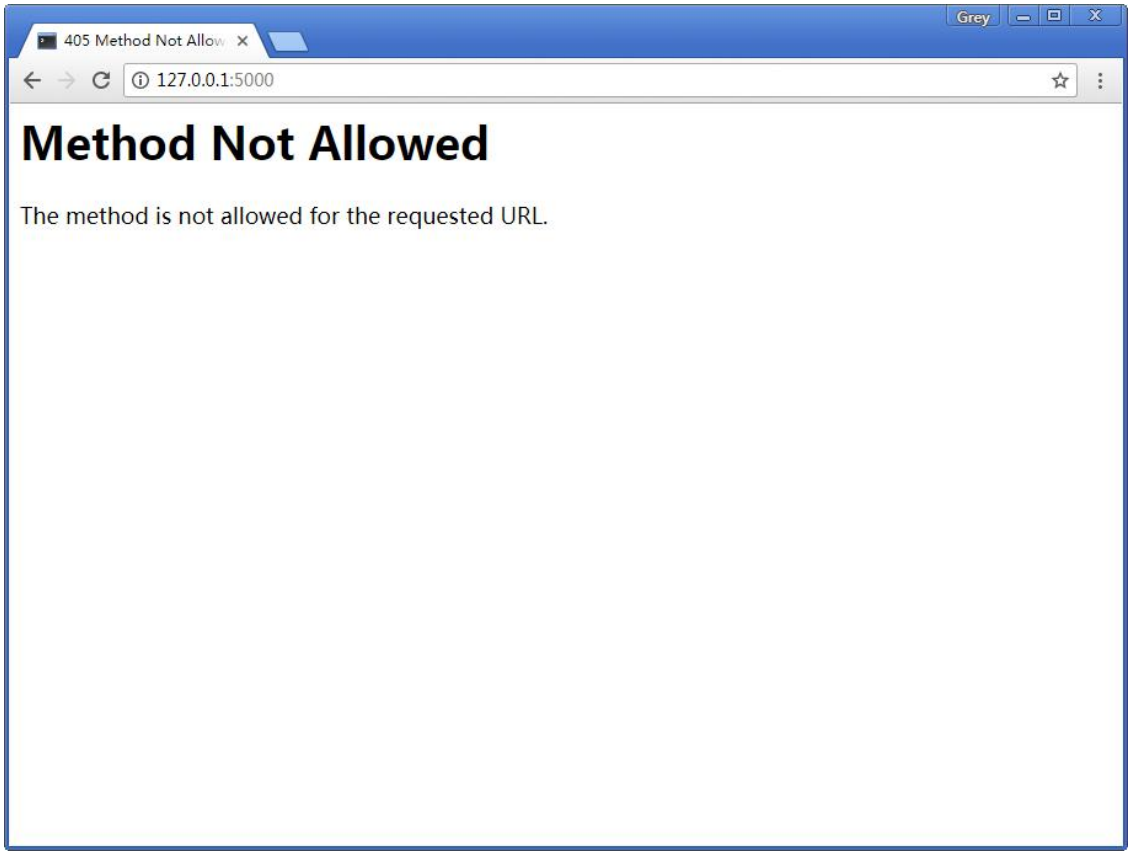

图 2-6 405 错误响应

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，我们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，我们就需要使用 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 提供了一些请求钩子（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。

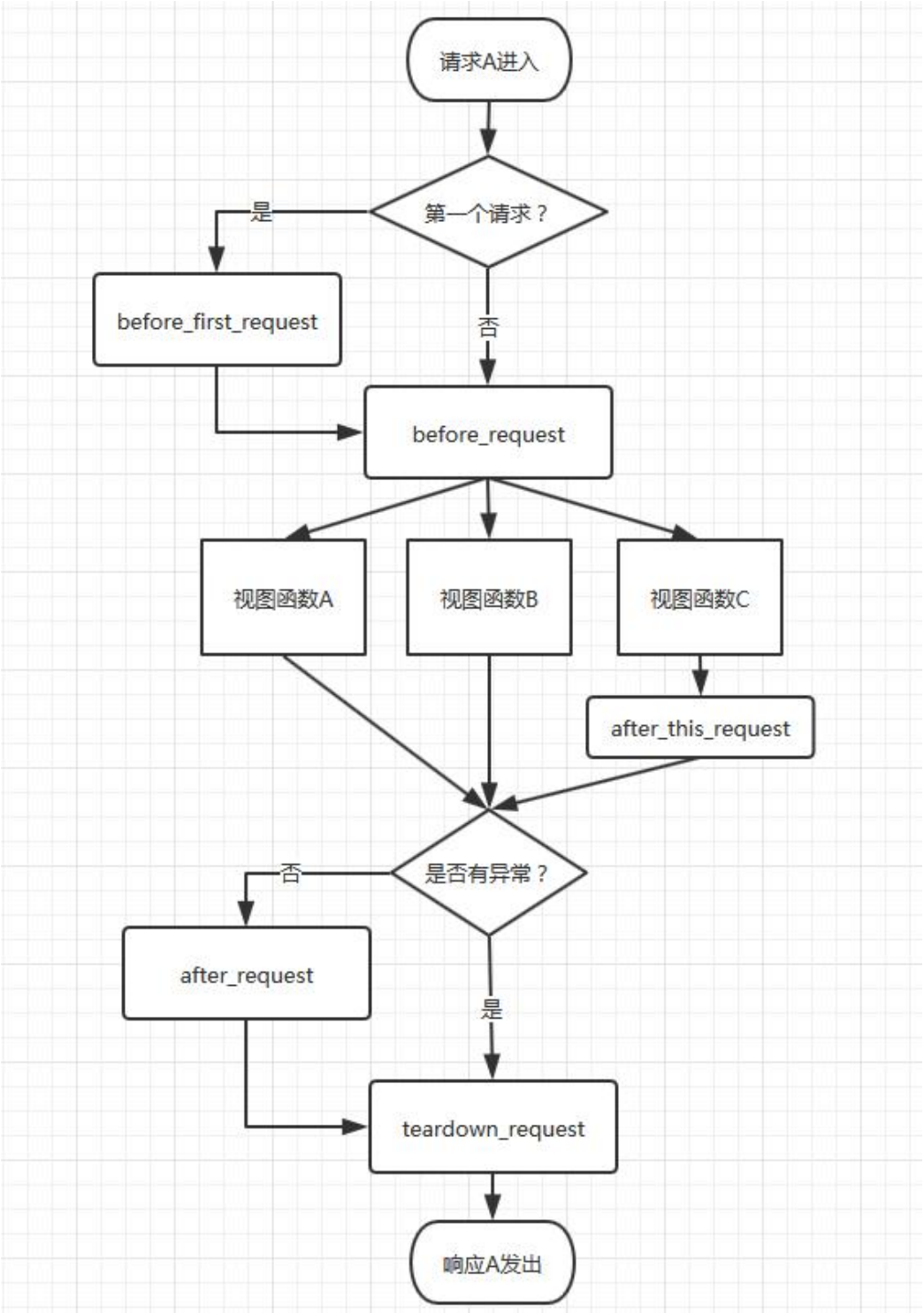

图 2-7 请求处理函数调用示意图

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。

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 钩子必须接受一个响应类对象作为参数，并且返回同一个或是更新后的响应对象。

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	Hello, Human!

响应报文的首部包含了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。浏览器接受到响应后，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。

根据我们在请求一节介绍的内容，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。

完整的说，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，浏览器接受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。

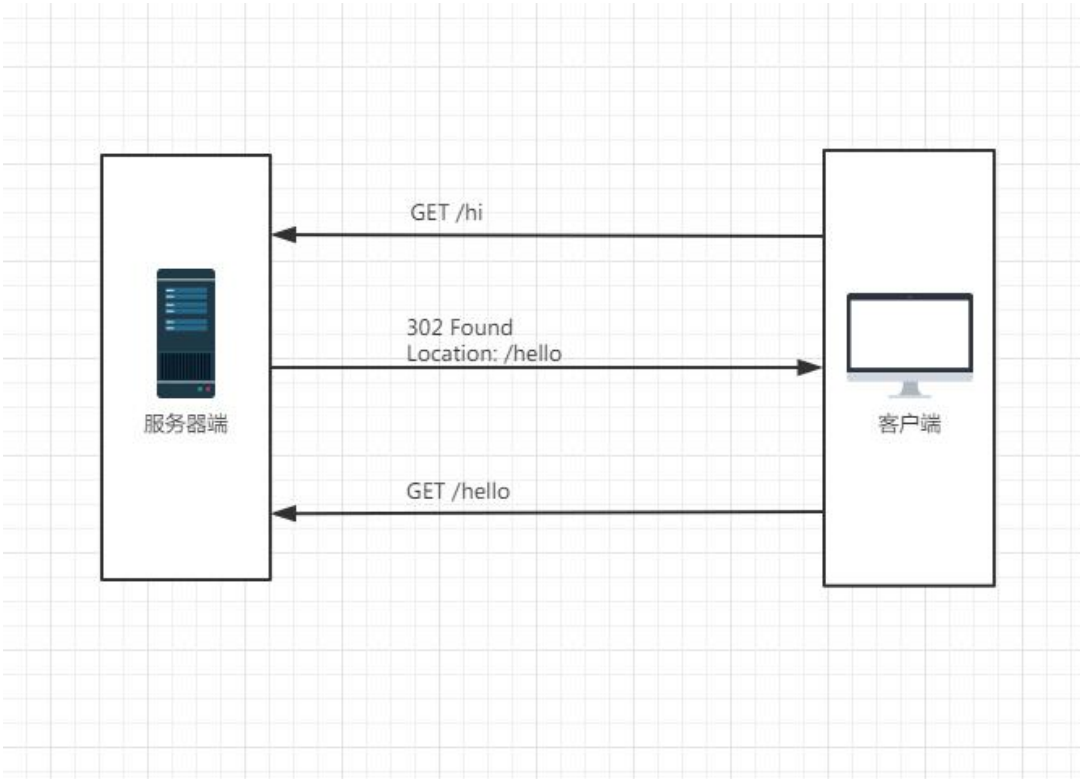

图 2-8 重定向流程示意图

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 redierct(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，你会获得一个 418 错误响应（I'm a teapot），如图 2-9 所示。

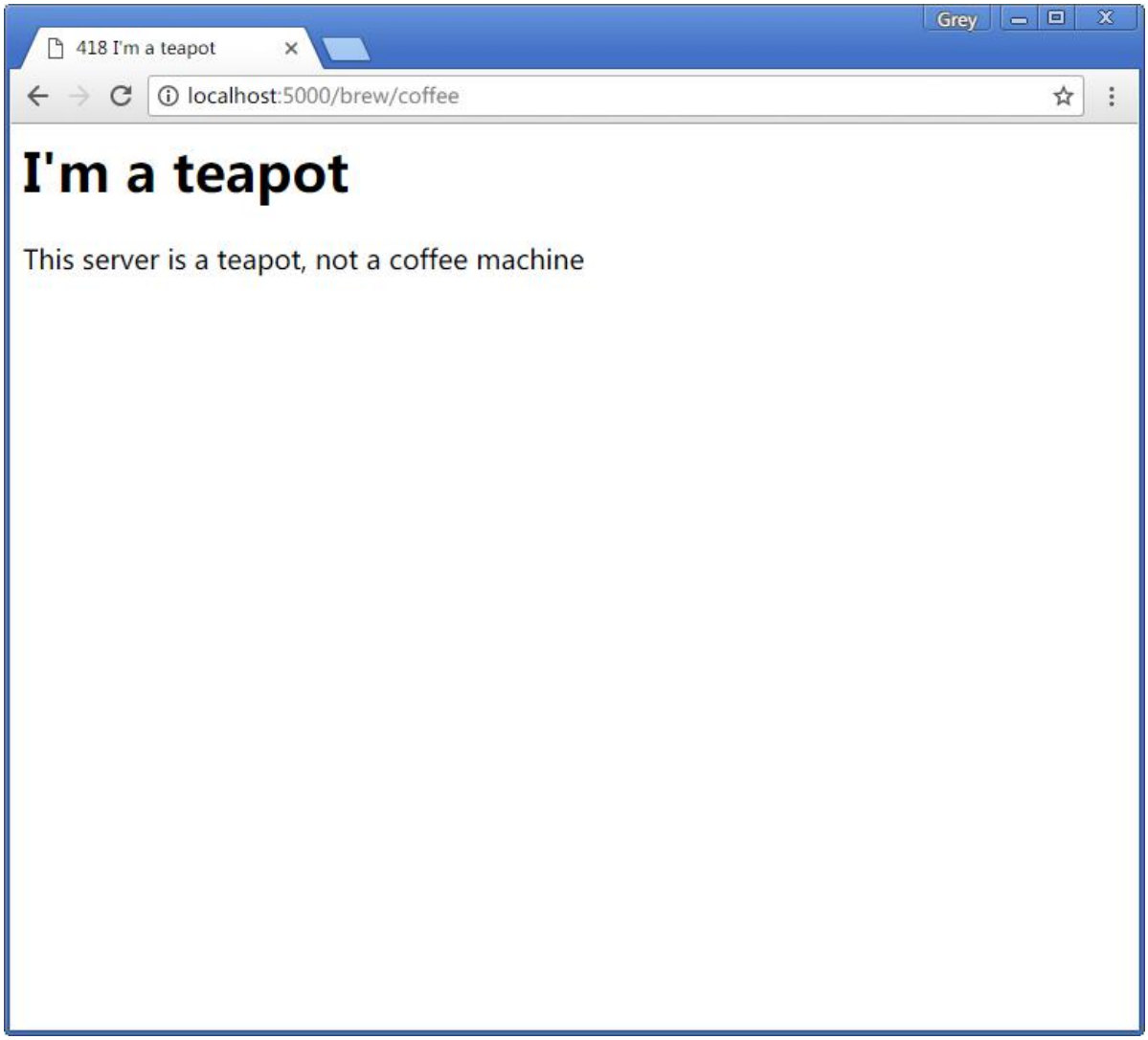

图 2-9 418 错误响应

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 HTCPCP（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 HTCPCP 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，但这并不是必须的。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。

- 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。

- 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。从我们在本书的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 嵌入到页面中，处理起来比较方便。因此，在普通的 HTTP 请求中我们使用 HTTP 作为响应的内容，这也是默认的数据类型。

- 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 提供的信息又太过丰富了，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。

- JOSN

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 更好的兼容。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()函数接受多种形式的参数。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。

试读部分完结。

请访问本书主页 http://helloflask.com/book 了解更多信息。