

1. 设置程序密钥

session 通过密钥对数据进行签名以加密数据，因此，我们得先设置一个密钥。这里的密钥就是一个具有一定复杂度和随机性的字符串，比如“Drmhze6EPcv0fN_81Bj-nA”。

程序的密钥可以通过 Flask.secret_key 属性或配置变量 SECRET_KEY 设置，比如：

```
app.secret_key = 'secret string'
```

更安全的做法是把密钥写进系统环境变量（在命令行中使用 export 或 set 命令），或是保存在 .env 文件中：

```
SECRET_KEY=secret string
```

然后在程序脚本中使用 os 模块提供的 getenv() 方法获取：

```
import os
# ...
app.secret_key = os.getenv('SECRET_KEY', 'secret string')
```

我们可以在 getenv() 方法中添加第二个参数，作为没有获取到对应环境变量时使用的默认值。



这里的密钥只是示例。在生产环境中，为了安全考虑，你必须使用随机生成的密钥，在第 14 章我们会介绍如何生成随机密钥值。在本书中或相关示例程序中，为了方便会使用诸如 secret string、dev key 之类的占位文字。

2. 模拟用户认证

下面我们会使用 session 模拟用户的认证功能。代码清单 2-5 是用来登入用户的 login 视图。

代码清单 2-5 http/app.py：登入用户

```
from flask import redirect, session, url_for
@app.route('/login')
def login():
    session['logged_in'] = True # 写入session
    return redirect(url_for('hello'))
```

这个登录视图只是简化的示例，在实际的登录中，我们需要在页面上提供登录表单，供用户填写账户和密码，然后在登录视图里验证账户和密码的有效性。session 对象可以像字典一样操作，我们向 session 中添加一个 logged-in cookie，将它的值设为 True，表示用户已认证。

当我们使用 session 对象添加 cookie 时，数据会使用程序的密钥对其进行签名，加密后的数据存储在一块名为 session 的 cookie 里，如图 2-12 所示。

你可以在图 2-12 方框内的 Content 部分看到对应的加密处理后生成的 session 值。使用 session 对象存储的 Cookie，用户可以看到其加密后的值，但无法修改它。因为 session 中的内容使用密钥进行签名，一旦数据被修改，签名的值也会变化。这样在读取时，就会验证失败，对应的 session 值也会随之失效。所以，除非用户知道密钥，否则无法对 session cookie 的值进行修改。

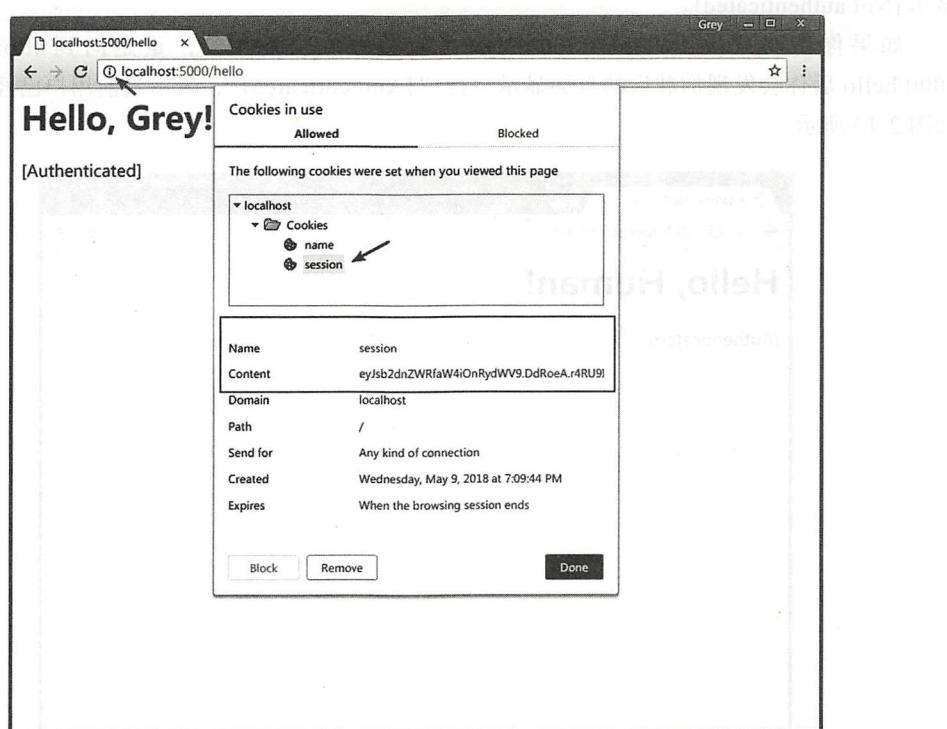


图 2-12 session cookie

当支持用户登录后，我们就可以根据用户的认证状态分别显示不同的内容。在 login 视图的最后，我们将程序重定向到 hello 视图，下面是修改后的 hello 视图：

```
from flask import request, session
@app.route('/')
@app.route('/hello')
def hello():
    name = request.args.get('name')
    if name is None:
        name = request.cookies.get('name', 'Human')
    response = '<h1>Hello, %s!</h1>' % name
    # 根据用户认证状态返回不同的内容
    if 'logged_in' in session:
        response += '[Authenticated]'
    else:
        response += '[Not Authenticated]'
    return response
```

session 中的数据可以像字典一样通过键读取，或是使用 get() 方法。这里我们只是判断 session 中是否包含 logged_in 键，如果有则表示用户已经登录。通过判断用户的认证状态，我们在返回的响应中添加一行表示认证状态的信息：如果用户已经登录，显示 [Authenticated]；否则

显示 [Not authenticated]。

如果你访问 <http://localhost:5000/login>，就会登入当前用户，重定向到 <http://localhost:5000/hello> 后你会发现加载后的页面显示一行 “[Authenticated]”，表示当前用户已经通过认证，如图 2-13 所示。

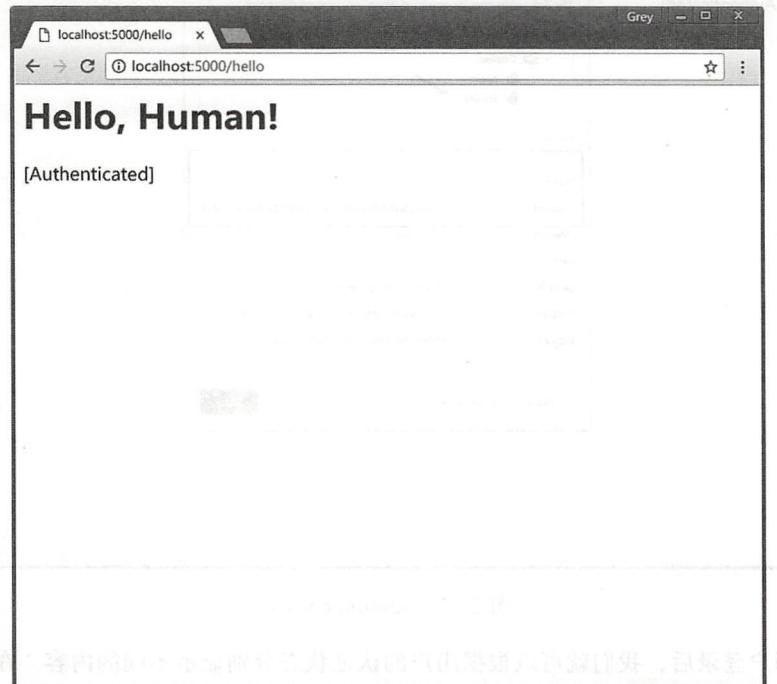


图 2-13 已认证的主页

程序中的某些资源仅提供给登入的用户，比如管理后台，这时我们就可以通过判断 session 是否存在 logged-in 键来判断用户是否认证，代码清单 2-6 是模拟管理后台的 admin 视图。

代码清单 2-6 http/app.py：模拟管理后台

```
from flask import session, abort

@app.route('/admin')
def admin():
    if 'logged_in' not in session:
        abort(403)
    return 'Welcome to admin page.'
```

通过判断 logged_in 是否在 session 中，我们可以实现：如果用户已经认证，会返回一行提示文字，否则会返回 403 错误响应。

登出用户的 logout 视图也非常简单，登出账户对应的实际操作其实就是把代表用户认证的 logged-in cookie 删除，这通过 session 对象的 pop 方法实现，如代码清单 2-7 所示。



代码清单2-7 http/app.py: 登出用户

```
from flask import session
@app.route('/logout')
def logout():
    if 'logged_in' in session:
        session.pop('logged_in')
    return redirect(url_for('hello'))
```

现在访问 `http://localhost:5000/logout` 则会登出用户，重定向后的 /hello 页面的认证状态信息会变为 [Not authenticated]，如图 2-14 所示。

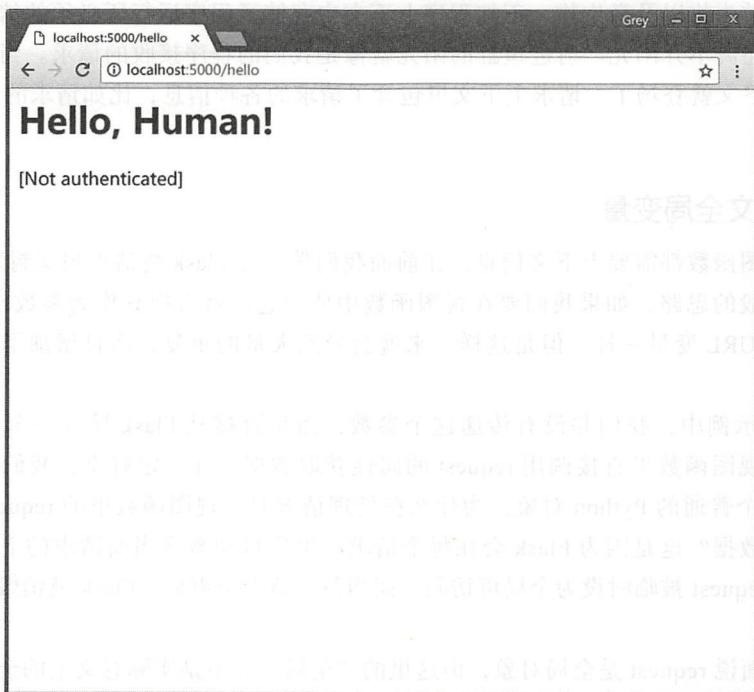


图 2-14 未认证的主页

提 示 默认情况下，`session cookie`会在用户关闭浏览器时删除。通过将 `session.permanent` 属性设为 `True` 可以将 `session` 的有效期延长为 `Flask.permanent_session_lifetime` 属性值对应的 `datetime.timedelta` 对象，也可通过配置变量 `PERSISTENT_SESSION_LIFETIME` 设置，默认为 31 天。

注 意 尽管 `session` 对象会对 `Cookie` 进行签名并加密，但这种方式仅能够确保 `session` 的内容不会被篡改，加密后的数据借助工具仍然可以轻易读取（即使不知道密钥）。因此，绝对不能在 `session` 中存储敏感信息，比如用户密码。



2.4 Flask 上下文

我们可以把编程中的上下文理解为当前环境（environment）的快照（snapshot）。如果把一个 Flask 程序比作一条可怜的生活在鱼缸里的鱼的话，那么它当然离不开身边的环境。

 提示 这里的上下文和阅读文章时的上下文基本相同。如果在某篇文章里单独抽出一句话来看，我们可能会觉得摸不着头脑，只有联系上下文后我们才能正确理解文章。

Flask 中有两种上下文，程序上下文（application context）和请求上下文（request context）。如果鱼想要存活，水是必不可少的元素。对于 Flask 程序来说，程序上下文就是我们的水。水里包含了各种浮游生物以及微生物，正如程序上下文中存储了程序运行所必须的信息；要想健康地活下去，鱼还离不开阳光。射进鱼缸的阳光就像是我们的程序接收的请求。当客户端发来请求时，请求上下文就登场了。请求上下文里包含了请求的各种信息，比如请求的 URL，请求的 HTTP 方法等。

2.4.1 上下文全局变量

每一个视图函数都需要上下文信息，在前面我们学习过 Flask 将请求报文封装在 request 对象中。按照一般的思路，如果我们要在视图函数中使用它，就得把它作为参数传入视图函数，就像我们接收 URL 变量一样。但是这样一来就会导致大量的重复，而且增加了视图函数的复杂度。

在前面的示例中，我们并没有传递这个参数，而是直接从 Flask 导入一个全局的 request 对象，然后在视图函数里直接调用 request 的属性获取数据。你一定好奇，我们在全局导入时 request 只是一个普通的 Python 对象，为什么在处理请求时，视图函数里的 request 就会自动包含对应请求的数据？这是因为 Flask 会在每个请求产生后自动激活当前请求的上下文，激活请求上下文后，request 被临时设为全局可访问。而当每个请求结束后，Flask 就销毁对应的请求上下文。

我们在前面说 request 是全局对象，但这里的“全局”并不是实际意义上的全局。我们可以把这些变量理解为动态的全局变量。

在多线程服务器中，在同一时间可能会有多个请求在处理。假设有三个客户端同时向服务器发送请求，这时每个请求都有各自不同的请求报文，所以请求对象也必然是不同的。因此，请求对象只在各自的线程内是全局的。Flask 通过本地线程（thread local）技术将请求对象在特定的线程和请求中全局可访问。具体内容和应用我们会在后面进行详细介绍。

为了方便获取这两种上下文环境中存储的信息，Flask 提供了四个上下文全局变量，如表 2-12 所示。

 提示 这四个变量都是代理对象（proxy），即指向真实对象的代理。一般情况下，我们不需要太关注其中的区别。在某些特定的情况下，如果你需要获取原始对象，可以对代理对象调用 `_get_current_object()` 方法获取被代理的真实对象。



表 2-12 Flask 中的上下文变量

变 量 名	上 下 文 类 别	说 明
current_app	程序上下文	指向处理请求的当前程序实例
g	程序上下文	替代 Python 的全局变量用法，确保仅在当前请求中可用。用于存储全局数据，每次请求都会重设
request	请求上下文	封装客户端发出的请求报文数据
session	请求上下文	用于记住请求之间的数据，通过签名的 Cookie 实现

我们在前面对 session 和 request 都了解得差不多了，这里简单介绍一下 current_app 和 g。

你在这里也许会疑惑，既然有了程序实例 app 对象，为什么还需要 current_app 变量。在不同的视图函数中，request 对象都表示和视图函数对应的请求，也就是当前请求（current request）。而程序也会有多个程序实例的情况，为了能获取对应的程序实例，而不是固定的某一个程序实例，我们就需要使用 current_app 变量，后面会详细介绍。

因为 g 存储在程序上下文中，而程序上下文会随着每一个请求的进入而激活，随着每一个请求的处理完毕而销毁，所以每次请求都会重设这个值。我们通常会使用它结合请求钩子来保存每个请求处理前所需要的全局变量，比如当前登入的用户对象，数据库连接等。在前面的示例中，我们在 hello 视图中从查询字符串获取 name 的值，如果每一个视图都需要这个值，那么就要在每个视图重复这行代码。借助 g 我们可以将这个操作移动到 before_request 处理函数中执行，然后保存到 g 的任意属性上：

```
from flask import g

@app.before_request
def get_name():
    g.name = request.args.get('name')
```

设置这个函数后，在其他视图中可以直接使用 g.name 获取对应的值。另外，g 也支持使用类似字典的 get()、pop() 以及 setdefault() 方法进行操作。

2.4.2 激活上下文

阳光柔和，鱼儿在水里欢快地游动，这一切都是上下文存在后的美好景象。如果没有上下文，我们的程序只能直挺挺地躺在鱼缸里。在下面这些情况下，Flask 会自动帮我们激活程序上下文：

- 当我们使用 flask run 命令启动程序时。
- 使用旧的 app.run() 方法启动程序时。
- 执行使用 @app.cli.command() 装饰器注册的 flask 命令时。
- 使用 flask shell 命令启动 Python Shell 时。

当请求进入时，Flask 会自动激活请求上下文，这时我们可以使用 request 和 session 变量。另外，当请求上下文被激活时，程序上下文也被自动激活。当请求处理完毕后，请求上下文和程序上下文也会自动销毁。也就是说，在请求处理时这两者拥有相同的生命周期。



结合 Python 的代码执行机制理解，这也就意味着，我们可以在视图函数中或在视图函数内调用的函数 / 方法中使用所有上下文全局变量。在使用 flask shell 命令打开的 Python Shell 中，或是自定义的 flask 命令函数中，我们可以使用 current_app 和 g 变量，也可以手动激活请求上下文来使用 request 和 session。

如果我们在没有激活相关上下文时使用这些变量，Flask 就会抛出 RuntimeError 异常：“RuntimeError: Working outside of application context.” 或是 “RuntimeError: Working outside of request context.”。



提 示 同样依赖于上下文的还有 url_for()、jsonify() 等函数，所以你也只能在视图函数中使用它们。其中 jsonify() 函数内部调用中使用了 current_app 变量，而 url_for() 则需要依赖请求上下文才可以正常运行。

如果你需要在没有激活上下文的情况下使用这些变量，可以手动激活上下文。比如，下面是一个普通的 Python shell，通过 python 命令打开。程序上下文对象使用 app.app_context() 获取，我们可以使用 with 语句执行上下文操作：

```
>>> from app import app
>>> from flask import current_app
>>> with app.app_context():
...     current_app.name
'app'
```

或是显式地使用 push() 方法推送（激活）上下文，在执行完相关操作时使用 pop() 方法销毁上下文：

```
>>> from app import app
>>> from flask import current_app
>>> app_ctx = app.app_context()
>>> app_ctx.push()
>>> current_app.name
'app'
>>> app_ctx.pop()
```

而请求上下文可以通过 test_request_context() 方法临时创建：

```
>>> from app import app
>>> from flask import request
>>> with app.test_request_context('/hello'):
...     request.method
'GET'
```

同样的，这里也可以使用 push() 和 pop() 方法显式地推送和销毁请求上下文。

2.4.3 上下文钩子

在前面我们学习了请求生命周期中可以使用的几种钩子，Flask 也为上下文提供了一个 teardown_appcontext 钩子，使用它注册的回调函数会在程序上下文被销毁时调用，而且通常也会在请求上下文被销毁时调用。比如，你需要在每个请求处理结束后销毁数据库连接：



```
@app.teardown_appcontext
def teardown_db(exception):
    ...
    db.close()
```

使用 `app.teardown_appcontext` 装饰器注册的回调函数需要接收异常对象作为参数，当请求被正常处理时这个参数值将是 `None`，这个函数的返回值将被忽略。

上下文是 Flask 的重要话题，在这里我们也只是简单了解一下，在本书的第三部分，我们会详细了解上下文的实现原理。

2.5 HTTP 进阶实践

在本书的第一部分，从本章开始，每一章的最后都会包含一个“进阶实践”部分，其中介绍的内容我们将会在第二部分的程序实例中使用到。在这一节，我们会接触到一些关于 HTTP 的进阶内容。

2.5.1 重定向回上一个页面

在前面的示例程序中，我们使用 `redirect()` 函数生成重定向响应。比如，在 `login` 视图中，登入用户后我们将用户重定向到 `/hello` 页面。在复杂的应用场景下，我们需要在用户访问某个 URL 后重定向到上一个页面。最常见的情况是，用户单击某个需要登录才能访问的链接，这时程序会重定向到登录页面，当用户登录后合理的行为是重定向到用户登录前浏览的页面，以便用户执行未完成的操作，而不是直接重定向到主页。在示例程序中，我们创建了两个视图函数 `foo` 和 `bar`，分别显示一个 Foo 页面和一个 Bar 页面，如下所示：

```
@app.route('/foo')
def foo():
    return '<h1>Foo page</h1><a href="%s">Do something</a>' % url_for('do_something')

@app.route('/bar')
def bar():
    return '<h1>Bar page</h1><a href="%s">Do something </a>' % url_for('do_something')
```

在这两个页面中，我们都添加了一个指向 `do_something` 视图的链接。这个 `do_something` 视图如下所示：

```
@app.route('/do_something')
def do_something():
    # do something
    return redirect(url_for('hello'))
```

我们希望这个视图在执行完相关操作后能够重定向回上一个页面，而不是固定的 `/hello` 页面。也就是说，如果在 Foo 页面上单击链接，我们希望被重定向回 Foo 页面；如果在 Bar 页面上单击链接，我们则希望返回到 Bar 页面。这一节我们会借助这个例子来介绍这一功能的实现。



1. 获取上一个页面的 URL

要重定向回上一个页面，最关键的是获取上一个页面的 URL。上一个页面的 URL 一般可以通过两种方式获取：

(1) HTTP referer

HTTP referer（起源为referrer在HTTP规范中的错误拼写）是一个用来记录请求发源地址的HTTP首部字段(HTTP_REFERER)，即访问来源。当用户在某个站点单击链接，浏览器向新链接所在的服务器发起请求，请求的数据中包含的HTTP_REFERER字段记录了用户所在的原站点URL。

这个值通常会用来追踪用户，比如记录用户进入程序的外部站点，以此来更有针对性地进行营销。在Flask中，referrer的值可以通过请求对象的referrer属性获取，即request.referrer(正确拼写形式)。现在，do_something视图的返回值可以这样编写：

```
return redirect(request.referrer)
```

但是在很多种情况下，referrer字段会是空值，比如用户在浏览器的地址栏输入URL，或是用户出于保护隐私的考虑使用了防火墙软件或使用浏览器设置自动清除或修改了referrer字段。我们需要添加一个备选项：

```
return redirect(request.referrer or url_for('hello'))
```

(2) 查询参数

除了自动从referrer获取，另一种更常见的方式是在URL中手动加入包含当前页面URL的查询参数，这个查询参数一般命名为next。比如，下面在foo和bar视图的返回值中的URL后添加next参数：

```
from flask import request

@app.route('/foo')
def foo():
    return '<h1>Foo page</h1><a href="%s">Do something and redirect</a>' % url_for('do_something', next=request.full_path)

@app.route('/bar')
def bar():
    return '<h1>Bar page</h1><a href="%s">Do something and redirect</a>' % url_for('do_something', next=request.full_path)
```

在程序内部只需要使用相对URL，所以这里使用request.full_path获取当前页面的完整路径。在do_something视图中，我们获取这个next值，然后重定向到对应的路径：

```
return redirect(request.args.get('next'))
```

用户在浏览器的地址栏直接访问时可以轻易地修改查询参数，为了避免next参数为空的情况，我们也要添加备选项，如果为空就重定向到hello视图：

```
return redirect(request.args.get('next', url_for('hello')))
```

为了覆盖更全面，我们可以将这两种方式搭配起来一起使用：首先获取next参数，如果为



空就尝试获取 referer，如果仍然为空，那么就重定向到默认的 hello 视图。因为在不同视图执行这部分操作的代码完全相同，我们可以创建一个通用的 redirect_back() 函数，如代码清单 2-8 所示。

代码清单2-8 http/app.py：重定向回上一个页面

```
def redirect_back(default='hello', **kwargs):
    for target in request.args.get('next'), request.referrer:
        if target:
            return redirect(target)
    return redirect(url_for(default, **kwargs))
```

通过设置默认值，我们可以在 referer 和 next 为空的情况下重定向到默认的视图。在 do_something 视图中使用这个函数的示例如下所示：

```
@app.route('/do_something_and_redirect')
def do_something():
    # do something
    return redirect_back()
```

2. 对 URL 进行安全验证

虽然我们已经实现了重定向回上一个页面的功能，但安全问题不容小觑，鉴于 referer 和 next 容易被篡改的特性，如果我们不对这些值进行验证，则会形成开放重定向（Open Redirect）漏洞。

以 URL 中的 next 参数为例，next 变量以查询字符串的方式写在 URL 里，因此任何人都可以发给某个用户一个包含 next 变量指向任何站点的链接。举个简单的例子，如果你访问下面的 URL：

```
http://localhost:5000/do-something?next=http://helloflask.com
```

程序会被重定向到 http://helloflask.com。也就是说，如果我们不验证 next 变量指向的 URL 地址是否属于我们的应用内，那么程序很容易就会被重定向到外部地址。你也许还不明白这其中会有什么危险，下面假设的情况也许会给你一个清晰的认识：

假设我们的应用是一个银行业务系统（下面简称网站 A），某个攻击者模仿我们的网站外观做了一个几乎一模一样的网站（下面简称网站 B）。接着，攻击者伪造了一封电子邮件，告诉用户网站 A 账户信息需要更新，然后向用户提供一个指向网站 A 登录页面的链接，但链接中包含一个重定向到网站 B 的 next 变量，比如：http://exampleA.com/login?next=http://maliciousB.com。当用户在 A 网站登录后，如果 A 网站重定向到 next 对应的 URL，那么就会导致重定向到攻击者编写的 B 网站。因为 B 网站完全模仿 A 网站的外观，攻击者就可以在重定向后的 B 网站诱导用户输入敏感信息，比如银行卡号及密码。

确保 URL 安全的关键就是判断 URL 是否属于程序内部，在代码清单 2-9 中，我们创建了一个 URL 验证函数 is_safe_url()，用来验证 next 变量值是否属于程序内部 URL。

代码清单2-9 http/app.py：验证URL安全性

```
from urlparse import urlparse, urljoin # Python3需要从urllib.parse导入
from flask import request

def is_safe_url(target):
```

```

def is_safe_url(target):
    ref_url = urlparse(request.host_url)
    test_url = urljoin(urljoin(request.host_url, target))
    return test_url.scheme in ('http', 'https') and \
        ref_url.netloc == test_url.netloc

```



注意 如果你使用 Python3，那么这里需要从 urllib.parse 模块导入 urlparse 和 urljoin 函数。示例程序仓库中实际的代码做了兼容性处理。

这个函数接收目标 URL 作为参数，并通过 request.host_url 获取程序内的主机 URL，然后使用 urljoin() 函数将目标 URL 转换为绝对 URL。接着，分别使用 urlparse 模块提供的 urlparse() 函数解析两个 URL，最后对目标 URL 的 URL 模式和主机地址进行验证，确保只有属于程序内部的 URL 才会被返回。在执行重定向回上一个页面的 redirect_back() 函数中，我们使用 is_safe_url() 验证 next 和 referer 的值：

```

def redirect_back(default='hello', **kwargs):
    for target in request.args.get('next'), request.referrer:
        if not target:
            continue
        if is_safe_url(target):
            return redirect(target)
    return redirect(url_for(default, **kwargs))

```



附注 关于开放重定向漏洞的更多信息可以访问 https://www.owasp.org/index.php/Unvalidated_Redirects_and_Forwards_Cheat_Sheet 了解。

2.5.2 使用 AJAX 技术发送异步请求

在传统的 Web 应用中，程序的操作都是基于请求响应循环来实现的。每当页面状态需要变动，或是需要更新数据时，都伴随着一个发向服务器的请求。当服务器返回响应时，整个页面会重载，并渲染新页面。

这种模式会带来一些问题。首先，频繁更新页面会牺牲性能，浪费服务器资源，同时降低用户体验。另外，对于一些操作性很强的程序来说，重载页面会显得很不合理。比如我们做了一个 Web 计算器程序，所有的按钮和显示屏幕都很逼真，但当我们单击“等于”按钮时，要等到页面重新加载后才在显示屏幕上看到结果，这显然会严重影响用户体验。我们这一节要学习的 AJAX 技术可以完美地解决这些问题。

1. 认识 AJAX

AJAX 指异步 Javascript 和 XML (Asynchronous JavaScript And XML)，它不是编程语言或通信协议，而是一系列技术的组合体。简单来说，AJAX 基于 XMLHttpRequest (<https://xhr.spec.whatwg.org/>) 让我们可以在不重载页面的情况下和服务器进行数据交换。加上 JavaScript 和 DOM(Document Object Model，文档对象模型)，我们就可以在接收到响应数据后局部更新页面。

而 XML 指的则是数据的交互格式，也可以是纯文本（Plain Text）、HTML 或 JSON。顺便说一句，XMLHttpRequest 不仅支持 HTTP 协议，还支持 FILE 和 FTP 协议。

 提示 AJAX 也常被拼作 Ajax，但是为了和古希腊神话里的英雄 Ajax 区分开来，在本书中将使用全大写形式，即 AJAX。

在 Web 程序中，很多加载数据的操作都可以在客户端使用 AJAX 实现。比如，当用户鼠标向下滚动到底部时在后台发送请求获取数据，然后插入文章；再比如，用户提交表单创建新的待办事项时，在后台将数据发送到服务器端，保存后将新的条目直接插入到页面上。

在这种模式下，我们可以在客户端实现大部分页面逻辑，而服务器端则主要负责处理数据。这样可以避免每次请求都渲染整个页面，这不仅增强了用户体验，也降低了服务器的负载。AJAX 让 Web 程序也可以像桌面程序那样获得更流畅的反应和动态效果。总而言之，AJAX 让 Web 程序更像是程序，而非一堆使用链接和按钮连接起来的网页资源。

以删除某个资源为例，在普通的程序中流程如下：

1) 当“删除”按钮被单击时会发送一个请求，页面变空白，在接收到响应前无法进行其他操作。

2) 服务器端接收请求，执行删除操作，返回包含整个页面的响应。

3) 客户端接收到响应，重载整个页面。

使用 AJAX 技术时的流程如下：

1) 当单击“删除”按钮时，客户端在后台发送一个异步请求，页面不变，在接收到响应前可以进行其他操作。

2) 服务器端接收请求后执行删除操作，返回提示消息或是无内容的 204 响应。

3) 客户端接收到响应，使用 JavaScript 更新页面，移除资源对应的页面元素。

2. 使用 jQuery 发送 AJAX 请求

jQuery 是流行的 JavaScript 库，它包装了 JavaScript，让我们通过更简单的方式编写 JavaScript 代码。对于 AJAX，它提供了多个相关的方法，使用它可以很方便地实现 AJAX 操作。更重要的是，jQuery 处理了不同浏览器的 AJAX 兼容问题，我们只需要编写一套代码，就可以在所有主流的浏览器正常运行。

 提示 使用 jQuery 实现 AJAX 并不是必须的，你可以选择使用原生的 XMLHttpRequest、其他 JavaScript 框架内置的 AJAX 接口，或是使用更新的 Fetch API (<https://fetch.spec.whatwg.org/>) 来发送异步请求。

在示例程序中，我们将使用全局 jQuery 函数 ajax() 发送 AJAX 请求。ajax() 函数是底层函数，有丰富的自定义配置，支持的主要参数如表 2-13 所示。

 附注 完整的可用配置参数列表可以在这里看到：<http://api.jquery.com/jQuery.ajax/#jQuery-ajax-settings>。

表 2-13 ajax() 函数支持的参数

参 数	参数值类型及默认值	说 明
url	字符串；默认为当前页地址	请求的地址
type	字符串；默认为 "GET"	请求的方式，即 HTTP 方法，比如 GET、POST、DELETE 等
data	字符串；无默认值	发送到服务器的数据。会被 jQuery 自动转换为查询字符串
dataType	字符串；默认由 jQuery 自动判断	期待服务器返回的数据类型，可用的值如下："xml" "html" "script" "json" "jsonp" "text"
contentType	字符串；默认为 'application/x-www-form-urlencoded; charset=UTF-8'	发送请求时使用的 content-type，即请求首部的 Content-Type 字段内容
complete	函数；无默认值	请求完成后调用的回调函数
success	函数；无默认值	请求成功后的调用的回调函数
error	函数；无默认值	请求失败后调用的回调函数



jQuery 还提供了其他快捷方法 (shorthand method)：用于发送 GET 请求的 get() 方法和用于发送 POST 请求的 post() 方法，还有直接用于获取 json 数据的 getJson() 以及获取脚本的 getScript() 方法。这些方法都是基于 ajax() 方法实现的。在这里，为了便于理解，使用了底层的 ajax 方法。jQuery 中和 AJAX 相关的方法及其具体用法可以在这里看到：<http://api.jquery.com/category/ajax/>。

3. 返回“局部数据”

对于处理 AJAX 请求的视图函数来说，我们不会返回完整的 HTML 响应，这时一般会返回局部数据，常见的三种类型如下所示：

1. 纯文本或局部 HTML 模板

纯文本可以在 JavaScript 用来直接替换页面中的文本值，而局部 HTML 则可以直接插入页面中，比如返回评论列表：

```
@app.route('/comments/<int:post_id>')
def get_comments(post_id):
    ...
    return render_template('comments.html')
```

2. JSON 数据

JSON 数据可以在 JavaScript 中直接操作：

```
@app.route('/profile/<int:user_id>')
def get_profile(user_id):
    ...
    return jsonify(username=username, bio=bio)
```

在 jQuery 中的 ajax() 方法的 success 回调中，响应主体中的 JSON 字符串会被解析为 JSON

对象，我们可以直接获取并进行操作。

3. 空值

有些时候，程序中的某些接收 AJAX 请求的视图并不需要返回数据给客户端，比如用来删除文章的视图。这时我们可以直接返回空值，并将状态码指定为 204（表示无内容），比如：

```
@app.route('/post/delete/<int:post_id>', methods=['DELETE'])
def delete_post(post_id):
    ...
    return '', 204
```

4. 异步加载长文章示例

在示例程序的对应页面中，我们将显示一篇很长的虚拟文章，文章正文下方有一个“加载更多”按钮，当加载按钮被单击时，会发送一个 AJAX 请求获取文章的更多内容并直接动态插入到文章下方。用来显示虚拟文章的 show_post 视图如代码清单 2-10 所示。

代码清单 2-10 http/app.py：显示虚拟文章

```
from jinja2.utils import generate_lorem_ipsum

@app.route('/post')
def show_post():
    post_body = generate_lorem_ipsum(n=2)  # 生成两段随机文本
    return '''
<h1>A very long post</h1>
<div class="body">%s</div>
<button id="load">Load More</button>
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<script type="text/javascript">
$(function() {
    $('#load').click(function() {
        $.ajax({
            url: '/more',           // 目标URL
            type: 'get',             // 请求方法
            success: function(data){ // 返回2XX响应后触发的回调函数
                $('.body').append(data); // 将返回的响应插入到页面中
            }
        })
    })
})
</script>''' % post_body
```

文章的随机正文通过 Jinja2 提供的 generate_lorem_ipsum() 函数生成，n 参数用来指定段落的数量，默认为 5，它会返回由随机字符组成的虚拟文章。文章下面添加了一个“加载更多”按钮。按钮下面是两个<script></script>代码块，第一个 script 从 CDN 加载 jQuery 资源。

在第二个 script 标签中，我们在代码的最外层创建了一个 \$(function(){ ... }) 函数，这个函数是常见的 \$(document).ready(function() { ... }) 函数的简写形式。这个函数用来在页面 DOM 加载完毕后执行代码，类似传统 JavaScript 中的 window.onload 方法，所以我们通常会将代码包装

在这个函数中。美元符号是 jQuery 的简写，我们通过它来调用 jQuery 提供的多个方法，所以 `$.ajax()` 等同于 `jQuery.ajax()`。

在 `$(function() { ... })` 中，`$('#load')` 被称为选择器，我们在括号中传入目标元素的 id、class 或是其他属性来定位到对应的元素，将其创建为 jQuery 对象。我们传入了“加载更多”按钮的 id 值以定位到加载按钮。在这个选择器上，我们附加了 `.click(function() { ... })`，这会为加载按钮注册一个单击事件处理函数，当加载按钮被单击时就会执行单击事件回调函数。在这个回调函数中，我们使用 `$.ajax()` 方法发送一个 AJAX 请求到服务器，通过 url 将目标 URL 设为“/more”，通过 type 参数将请求的类型设为 GET。当请求成功处理并返回 2XX 响应时（另外还包括 304 响应），会触发 success 回调函数。success 回调函数接收的第一个参数为服务器端返回的响应主体，在这个回调函数中，我们在文章正文（通过 `$('.body')` 选择）底部使用 `append()` 方法插入返回的 data 数据。



提 示 由于篇幅所限，我们不会深入介绍 JavaScript 或 jQuery，你可以阅读其他书籍来学习更多内容。

处理 /more 的视图函数会返回随机文章正文，如下所示：

```
@app.route('/more')
def load_post():
    return generate_lorem_ipsum(n=1)
```

如果你启动了示例程序，那么访问 `http://localhost:5000/post` 可以看到文章页面，当你单击文章下的“Load More”按钮时，浏览器就会在后台发送一个 GET 请求到 /more，这个视图返回的随机字符会被动态插入到文章下方。



附 注 在出版业和设计业，lorem ipsum 指一段常用的无意义的填充文字。以 lorem ipsum 开头的这段填充文本是抽取哲学著作《On the ends of good and evil》中的文段，并对单词进行删改调换而来。

2.5.3 HTTP 服务器端推送

不论是传统的 HTTP 请求 - 响应式的通信模式，还是异步的 AJAX 式请求，服务器端始终处于被动的应答状态，只有在客户端发出请求的情况下，服务器端才会返回响应。这种通信模式被称为客户端拉取（client pull）。在这种模式下，用户只能通过刷新页面或主动单击加载按钮来拉取新数据。

然而，在某些场景下，我们需要的通信模式是服务器端的主动推送（server push）。比如，一个聊天室有很多个用户，当某个用户发送消息后，服务器接收到这个请求，然后把消息推送给聊天室的所有用户。类似这种关注实时性的情况还有很多，比如社交网站在导航栏实时显示新提醒和私信的数量，用户的在线状态更新，股价行情监控、显示商品库存信息、多人游戏、文档协作等。

实现服务器端推送的一系列技术被合称为 HTTP Server Push (HTTP 服务器端推送)，目前常用的推送技术如表 2-14 所示。

表 2-14 常用推送技术

名 称	说 明
传统轮询	在特定的时间间隔内，客户端使用 AJAX 技术不断向服务器发起 HTTP 请求，然后获取新的数据并更新页面
长轮询	和传统轮询类似，但是如果服务器端没有返回数据，那就保持连接一直开启，直到有数据时才返回。取回数据后再次发送另一个请求
Server-Sent Events (SSE)	SSE 通过 HTML5 中的 EventSource API 实现。SSE 会在客户端和服务器端建立一个单向的通道，客户端监听来自服务器端的数据，而服务器端可以在任意时间发送数据，两者建立类似订阅 / 发布的通信模式

按照列出的顺序来说，这几种方式对实时通信的实现越来越完善。当然，每种技术都有各自的优缺点，在具体的选择上，要根据面向的用户群以及程序自身的特点来分析选择。这些技术我们会在本书第二部分的程序实例中逐一介绍。

轮询 (polling) 这类使用 AJAX 技术模拟服务器端推送的方法实现起来比较简单，但通常会造成服务器资源上的浪费，增加服务器的负担，而且会让用户的设备耗费更多的电量 (频繁地发起异步请求)。SSE 效率更高，在浏览器的兼容性方面，除了 Windows IE/Edge，SSE 基本上支持所有主流浏览器，但浏览器通常会限制标签页的连接数量。

 **附注** Server-Sent Event 的最新标准可以在 WHATWG (<https://html.spec.whatwg.org/multipage/server-sent-events.html>) 查看，浏览器的支持情况可以在 Can I use... (<https://caniuse.com/#feat=eventsource>) 查看。

除了这些推送技术，在 HTML5 的 API 中还包含了一个 WebSocket 协议，和 HTTP 不同，它是一种基于 TCP 协议的全双工通信协议 (full-duplex communication protocol)。和前面介绍的服务器端推送技术相比，WebSocket 实时性更强，而且可以实现双向通信 (bidirectional communication)。另外，WebSocket 的浏览器兼容性要强于 SSE。

 **附注** WebSocket 协议在 RFC 6455 (<https://tools.ietf.org/html/rfc6455>) 中定义，浏览器的支持情况可以在 Can I use... (<https://caniuse.com/#feat=websockets>) 查看。

 **附注** 如果你想进一步了解这几种推送技术的区别，StackOverflow 的这篇答案 <https://stackoverflow.com/a/12855533/5511849> 对这几种推送技术进行了对比，并提供了直观的图示。

2.5.4 Web 安全防范

无论是简单的博客，还是大型的社交网站，Web 安全都应该放在首位。Web 安全问题涉及

广泛，我们在这里介绍其中常见的几种攻击（attack）和其他常见的漏洞（vulnerability）。

对于 Web 程序的安全问题，一个首要的原则是：永远不要相信你的用户。大部分 Web 安全问题都是因为没有对用户输入的内容进行“消毒”造成的。

1. 注入攻击

在 OWASP（Open Web Application Security Project，开放式 Web 程序安全项目）发布的最危险的 Web 程序安全风险 Top 10 中，无论是最新的 2017 年的排名，2013 年的排名还是最早的 2010 年，注入攻击（Injection）都位列第一。注入攻击包括系统命令（OS Command）注入、SQL（Structured Query Language，结构化查询语言）注入（SQL Injection）、NoSQL 注入、ORM（Object Relational Mapper，对象关系映射）注入等。我们这里重点介绍的是 SQL 注入。



附注 SQL 是一种功能齐全的数据库语言，也是关系型数据库的通用操作语言。使用它可以对数据库中的数据进行修改、查询、删除等操作；ORM 是用来操作数据库的工具，使用它可以在不手动编写 SQL 语句的情况下操作数据库。



附注 OWASP (<https://www.owasp.org>) 是一个开源的、非盈利的国际性安全组织。在 OWASP 网站的 Top 10 页面中的 Translation Efforts 标签 (https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project) 下可以找到中文版本的 Top 10 报告。顺便说一句，我们在前面提及的开放重定向漏洞曾在 2013 OWASP Top10 中位列第 10：Unvalidated Redirects and Forwards（未经验证的重定向或转发）。

（1）攻击原理

在编写 SQL 语句时，如果直接将用户传入的数据作为参数使用字符串拼接的方式插入到 SQL 查询中，那么攻击者可以通过注入其他语句来执行攻击操作，这些攻击操作包括可以通过 SQL 语句做的任何事：获取敏感数据、修改数据、删除数据库表……

（2）攻击示例

假设我们的程序是一个学生信息查询程序，其中的某个视图函数接收用户输入的密码，返回根据密码查询对应的数据。我们的数据库由一个 db 对象表示，SQL 语句通过 execute() 方法执行：

```
@app.route('/students')
def bobby_table():
    password = request.args.get('password')
    cur = db.execute("SELECT * FROM students WHERE password='%s';" % password)
    results = cur.fetchall()
    return results
```



注意 在实际应用中，敏感数据需要通过表单提交的 POST 请求接收，这里为了便于演示，我们通过查询参数接收。

我们通过查询字符串获取用户输入的查询参数，并且不经过任何处理就使用字符串格式化

的方法拼接到 SQL 语句中。在这种情况下，如果攻击者输入的 password 参数值为 “' or 1=1 --”，即 `http://example.com/students?password=' or 1=1 --`，那么最终视图函数中被执行的 SQL 语句将变为：

```
SELECT * FROM students WHERE password=' or 1=1 --';
```

这时会把 students 表中的所有记录全部查询并返回，也就意味着所有的记录都被攻击者窃取了。更可怕的是，如果攻击者将 password 参数的值设为 “'; drop table users; --”，那么查询语句就会变成：

```
SELECT * FROM students WHERE password=''; drop table students; --;
```

执行这个语句会把 students 表中的所有记录全部删除掉。

 **附注** 在 SQL 中，“;”用来结束一行语句；“--”用来注释后面的语句，类似 Python 中的“#”。

(3) 主要防范方法

- 1) 使用 ORM 可以一定程度上避免 SQL 注入问题，我们将在第 5 章学习使用 ORM。
- 2) 验证输入类型。比如某个视图函数接收整型 id 来查询，那么就在 URL 规则中限制 URL 变量为整型。
- 3) 参数化查询。在构造 SQL 语句时避免使用拼接字符串或字符串格式化（使用百分号或 `format()` 方法）的方式来构建 SQL 语句。而要使用各类接口库提供的参数化查询方法，以内置的 sqlite3 库为例：

```
db.execute('SELECT * FROM students WHERE password=?', password)
```

- 4) 转义特殊字符，比如引号、分号和横线等。使用参数化查询时，各种接口库会为我们做转义工作。

 **附注** 你可以访问 OWASP 的 SQL 注入页面 (https://www.owasp.org/index.php/SQL_Injection) 了解详细的攻击原理介绍的防范措施。

2. XSS 攻击

XSS (Cross-Site Scripting，跨站脚本) 攻击历史悠久，最远可以追溯到 90 年代，但至今仍然是危害范围非常广的攻击方式。在 OWASP TOP 10 中排名第 7。

 **附注** Cross-Site Scripting 的缩写本应是 CSS，但是为了避免和 Cascading Style Sheets 的缩写产生冲突，所以将 Cross (即交叉) 使用交叉形状的 X 表示。

(1) 攻击原理

XSS 是注入攻击的一种，攻击者通过将代码注入被攻击者的网站中，用户一旦访问网页便会执行被注入的恶意脚本。XSS 攻击主要分为反射型 XSS 攻击 (Reflected XSS Attack) 和存储型 XSS 攻击 (Stored XSS Attack) 两类。

(2) 攻击示例

反射型 XSS 又称为非持久型 XSS (Non-Persistent XSS)。当某个站点存在 XSS 漏洞时，这种攻击会通过 URL 注入攻击脚本，只有当用户访问这个 URL 时才会执行攻击脚本。我们在本章前面介绍查询字符串和 cookie 时引入的示例就包含反射型 XSS 漏洞，如下所示：

```
@app.route('/hello')
def hello():
    name = request.args.get('name')
    response = '<h1>Hello, %s!</h1>' % name
```

这个视图函数接收用户通过查询字符串传入的数据，未做任何处理就把它直接插入到返回的响应主体中，返回给客户端。如果某个用户输入了一段 JavaScript 代码作为查询参数 name 的值，如下所示：

```
http://example.com/hello?name=<script>alert('Bingo!');</script>
```

客户端接收的响应将变为下面的代码：

```
<h1>Hello, <script>alert('Bingo!');</script>!</h1>
```

当客户端接收到响应后，浏览器解析这行代码就会打开一个弹窗，如图 2-15 所示。

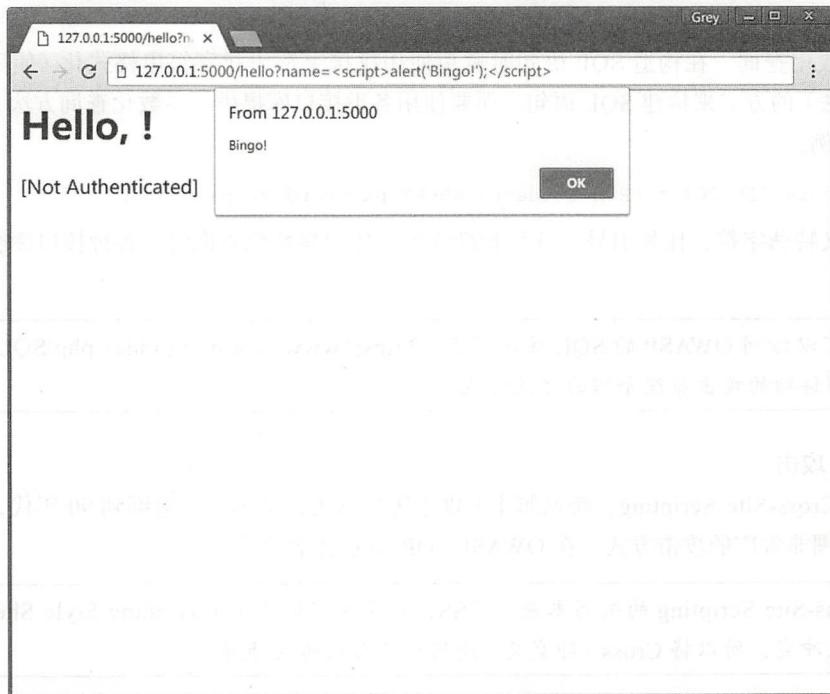


图 2-15 被注入代码后的响应

你觉得一个小弹窗不会造成什么危害？那你就完全错了，能够执行 alert() 函数就意味着通过这种方式可以执行任意 JavaScript 代码。即攻击者通过 JavaScript 几乎能够做任何事情：窃取

用户的 cookie 和其他敏感数据，重定向到钓鱼网站，发送其他请求，执行诸如转账、发布广告信息、在社交网站关注某个用户等。

 提示 即使不插入 JavaScript 代码，通过 HTML 和 CSS（CSS 注入）也可以影响页面正常的输出，篡改页面样式，插入图片等。

如果网站 A 存在 XSS 漏洞，攻击者将包含攻击代码的链接发送给网站 A 的用户 Foo，当 Foo 访问这个链接就会执行攻击代码，从而受到攻击。

存储型 XSS 也被称为持久型 XSS (persistent XSS)，这种类型的 XSS 攻击更常见，危害也更大。它和反射型 XSS 类似，不过会把攻击代码储存到数据库中，任何用户访问包含攻击代码的页面都会被殃及。比如，某个网站通过表单接收用户的留言，如果服务器接收数据后未经处理就存储到数据库中，那么用户可以在留言中插入任意 JavaScript 代码。比如，攻击者在留言中加入一行重定向代码：

```
<script>window.location.href="http://attacker.com";</script>
```

其他任意用户一旦访问留言板页面，就会执行其中的 JavaScript 脚本。那么其他用户一旦访问这个页面就会被重定向到攻击者写入的站点。

(3) 主要防范措施

a. HTML 转义

防范 XSS 攻击最主要的方法是对用户输入的内容进行 HTML 转义，转义后可以确保用户输入的内容在浏览器中作为文本显示，而不是作为代码解析。

 附注 这里的转义和 Python 中的概念相同，即消除代码执行时的歧义，也就是把变量标记的内容标记为文本，而不是 HTML 代码。具体来说，这会把变量中与 HTML 相关的符号转换为安全字符，以避免变量中包含影响页面输出的 HTML 标签或恶意的 JavaScript 代码。

比如，我们可以使用 Jinja2 提供的 escape() 函数对用户传入的数据进行转义：

```
from jinja2 import escape

@app.route('/hello')
def hello():
    name = request.args.get('name')
    response = '<h1>Hello, %s!</h1>' % escape(name)
```

 附注 在 Jinja2 中，HTML 转义相关的功能通过 Flask 的依赖包 MarkupSafe 实现。

调用 escape() 并传入用户输入的数据，可以获得转义后的内容，前面的示例中，用户输入的 JavaScript 代码将被转义为：

```
&lt;script&ampgtalert(&#34;Bingo!&#34;);&lt;/script&ampgt
```

转义后，文本中的特殊字符（比如“>”和“<”）都将被转义为 HTML 实体（character entity），这行文本最终在浏览器中会被显示为文本形式的 `<script>alert('Bingo!')</script>`，如图 2-16 所示。

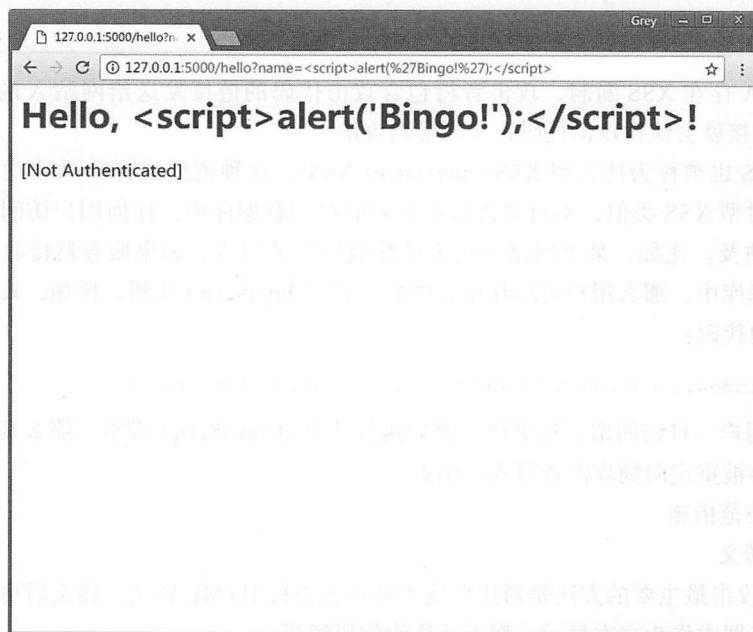


图 2-16 转义后的 JavaScript 代码输出



在 Python 中，如果你想在单引号标记的字符串中显示一个单引号，那么你需要在单引号前添加一个反斜线来转义它，也就是把它标记为普通文本，而不是作为特殊字符解释。在 HTML 中，也存在许多保留的特殊字符，比如大于小于号。如果你想以文本显示这些字符，也需要对其进行转义，即使用 HTML 字符实体表示这些字符。HTML 实体就是一些用来表示保留符号的特殊文本，比如 `<` 表示小于号，`"` 表示双引号。



一般我们不会在视图函数中直接构造返回的 HTML 响应，而是会使用 Jinja2 来渲染包含变量的模板，这部分内容我们将在第 3 章学习。

b. 验证用户输入

XSS 攻击可以在任何用户可定制内容的地方进行，例如图片引用、自定义链接。仅仅转义 HTML 中的特殊字符并不能完全规避 XSS 攻击，因为在某些 HTML 属性中，使用普通的字符也可以插入 JavaScript 代码。除了转义用户输入外，我们还需要对用户的输入数据进行类型验证。在所有接收用户输入的地方做好验证工作。在第 4 章学习表单时，我们会详细介绍表单数据的验证。

以某个程序的用户资料页面为例，我们来演示一下转义无法完全避免的 XSS 攻击。程序允

许用户输入个人资料中的个人网站地址，通过下面的方式显示在资料页面中：

```
<a href="{{ url }}>Website</a>
```

其中 {{ url }} 部分表示会被替换为用户输入的 url 变量值。如果不对 URL 进行验证，那么用户就可以写入 JavaScript 代码，比如 “javascript:alert('Bingo!');”。因为这个值并不包含会被转义的 < 和 >。最终页面上的链接代码会变为：

```
<a href="javascript:alert('Bingo!');">Website</a>
```

当用户单击这个链接时，就会执行被注入的攻击代码。

另外，程序还允许用户自己设置头像图片的 URL。这个图片通过下面的方式显示：

```
 标签就会变为：

```

```

在这里因为 src 中传入了一个错误的 URL，浏览器便会执行 onerror 属性中设置的 JavaScript 代码。

 提示 如果你想允许部分 HTML 标签，比如 <b> 和 <i>，可以使用 HTML 过滤工具对用户输入的数据进行过滤，仅保留少量允许使用的 HTML 标签，同时还要注意过滤 HTML 标签的属性，我们会在本书的第二部分详细了解。

 附注 你可以访问 OWASP 的 XSS 页面（[https://www.owasp.org/index.php/Cross-site\\_Scripting\\_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))）了解详细的攻击原理介绍和防范措施。

### 3. CSRF 攻击

CSRF (Cross Site Request Forgery，跨站请求伪造) 是一种近年来才逐渐被大众了解的网络攻击方式，又被称为 One-Click Attack 或 Session Riding。在 OWASP 上一次 (2013) 的 TOP 10 Web 程序安全风险中，它位列第 8。随着大部分程序的完善，各种框架都内置了对 CSRF 保护的支持，但目前仍有 5% 的程序受到威胁。

#### (1) 攻击原理

CSRF 攻击的大致方式如下：某用户登录了 A 网站，认证信息保存在 cookie 中。当用户访问攻击者创建的 B 网站时，攻击者通过在 B 网站发送一个伪造的请求提交到 A 网站服务器上，让 A 网站服务器误以为请求来自于自己的网站，于是执行相应的操作，该用户的信息便遭到了篡改。总结起来就是，攻击者利用用户在浏览器中保存的认证信息，向对应的站点发送伪造请求。在前面学习 cookie 时，我们介绍过用户认证通过保存在 cookie 中的数据实现。在发送请求时，只要浏览器中保存了对应的 cookie，服务器端就会认为用户已经处于登录状态，而攻击者正是利用了这一机制。为了更便于理解，下面我们举一个实例。

### (2) 攻击示例

假设我们网站是一个社交网站（example.com），简称网站 A；攻击者的网站可以是任意类型的网站，简称网站 B。在我们的网站中，删除账户的操作通过 GET 请求执行，由使用下面的 delete\_account 视图处理：

```
@app.route('/account/delete')
def delete_account():
 if not current_user.authenticated:
 abort(401)
 current_user.delete()
 return 'Deleted!'
```

当用户登录后，只要访问 `http://example.com/account/delete` 就会删除账户。那么在攻击者的网站上，只需要创建一个显示图片的 img 标签，其中的 src 属性加入删除账户的 URL：

```

```

当用户访问 B 网站时，浏览器在解析网页时会自动向 img 标签的 src 属性中的地址发起请求。此时你在 A 网站的登录信息保存在 cookie 中，因此，仅仅是访问 B 网站的页面就会让你的账户被删除掉。

当然，现实中很少有网站会使用 GET 请求来执行包含数据更改的敏感操作，这里只是一个示例。现在，假设我们吸取了教训，改用 POST 请求提交删除账户的请求。尽管如此，攻击者只需要在 B 网站中内嵌一个隐藏表单，然后设置在页面加载后执行提交表单的 JavaScript 函数，攻击仍然会在用户访问 B 网站时发起。

虽然 CSRF 攻击看起来非常可怕，但我们仍然可以采取一些措施来进行防御。下面我们来介绍防范 CSRF 攻击的两种主要方式。

### (3) 主要防范措施

#### a. 正确使用 HTTP 方法

防范 CSRF 的基础就是正确使用 HTTP 方法。在前面我们介绍过 HTTP 中的常用方法。在普通的 Web 程序中，一般只会使用到 GET 和 POST 方法。而且，目前在 HTML 中仅支持 GET 和 POST 方法（借助 AJAX 则可以使用其他方法）。在使用 HTTP 方法时，通常应该遵循下面的原则：

- GET 方法属于安全方法，不会改变资源状态，仅用于获取资源，因此又被称为幂等方法（idempotent method）。页面中所有可以通过链接发起的请求都属于 GET 请求。
- POST 方法用于创建、修改和删除资源。在 HTML 中使用 form 标签创建表单并设置提交方法为 POST，在提交时会创建 POST 请求。



**附注** 在 GET 请求中，查询参数用来传入过滤返回的资源，但是在某些特殊情况下，也可以通过查询参数传递少量非敏感信息。

虽然在实际开发中，通过在“删除”按钮中加入链接来删除资源非常方便，但安全问题应该作为编写代码时的第一考量，应该将这些按钮内嵌在使用了 POST 方法的 form 元素中。正确使用 HTTP 方法后，攻击者就无法通过 GET 请求来修改用户的数据，下面我们会介绍如何保护

GET之外的请求。

#### b. CSRF 令牌校验

当处理非 GET 请求时，要想避免 CSRF 攻击，关键在于判断请求是否来自自己的网站。在前面我们曾经介绍过使用 HTTP referer 获取请求来源，理论上说，通过 referer 可以判断源站点从而避免 CSRF 攻击，但因为 referer 很容易被修改和伪造，所以不能作为主要的防御措施。

除了在表单中加入验证码外，一般的做法是通过在客户端页面中加入伪随机数来防御 CSRF 攻击，这个伪随机数通常被称为 CSRF 令牌（token）。



**附注** 在计算机语境中，令牌（token）指用于标记、验证和传递信息的字符，通常是通过一定算法生成的伪随机数，我们在本书后面会频繁接触到这个词。

在 HTML 中，POST 方法的请求通过表单创建。我们把在服务器端创建的伪随机数（CSRF 令牌）添加到表单中的隐藏字段里和 session 变量（即签名 cookie）中，当用户提交表单时，这个令牌会和表单数据一起提交。在服务器端处理 POST 请求时，我们会对表单中的令牌值进行验证，如果表单中的令牌值和 session 中的令牌值相同，那么就说明请求发自自己的网站。因为 CSRF 令牌在用户向包含表单的页面发起 GET 请求时创建，并且在一定时间内过期，一般情况下攻击者无法获取到这个令牌值，所以我们可以有效地区分出请求的来源是否安全。



**附注** 对于 AJAX 请求，我们可以在 XMLHttpRequest 请求头部添加一个自定义字段 X-CSRFToken 来保存 CSRF 令牌。

我们通常会使用扩展实现 CSRF 令牌的创建和验证工作，比如 Flask-SeaSurf (<https://github.com/maxcountryman/flask-seasurf>)、Flask-WTF 内置的 CSRFProtect (<https://github.com/lepture/flask-wtf>) 等，在后面我们会详细介绍具体的实践内容。



**注意** 如果程序包含 XSS 漏洞，那么攻击者可以使用跨站脚本攻破可能使用的任何跨站请求伪造（CSRF）防御机制，比如使用 JavaScript 窃取 cookie 内容，进而获取 CSRF 令牌。



**附注** 可以访问 OWASP 的 CSRF 页面 ([https://www.owasp.org/index.php/Cross-Site\\_Request\\_Forgery\\_\(CSRF\)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF))) 了解详细的攻击原理介绍的防范措施。

除了这几个攻击方式外，我们还有很多安全问题要注意。比如文件上传漏洞、敏感数据存储、用户认证（authentication）与权限管理等。这些内容我们将在后面的章节陆续介绍。

需要注意的是，虽然本书会介绍如何对常见的攻击和漏洞进行防御和避免，但仍然有许多其他的攻击和漏洞需要读者自己处理。另外，本书的示例程序（包括第一部分和第二部分）仅用于作为功能实现的示例，在安全方面并未按照实际运行的应用进行严格处理。比如，当单个用户出现频繁的登录失败时，应该采取添加验证码或暂时停止接收该用户的登录请求。请阅读 OWASP 或其他相关资料学习更多安全防御技巧。



**附注** 你应该列出一个程序安全项目检查清单，可以参考 OWASP Top 10 或是 CWE (Common Weakness Enumeration, 一般弱点列举) 提供的 Top 25 (<https://cwe.mitre.org/top25/>)。确保你的程序所有的安全项目检查，也可以使用漏洞检查工具来，比如 OWASP 提供的 WebScarab (<https://github.com/OWASP/OWASP-WebScarab>)。

## 2.6 本章小结

HTTP 是各种 Web 程序的基础，本章只是简要介绍了和 Flask 相关的部分，没有涉及 HTTP 底层的 TCP/IP 或 DNS 协议。建议你通过阅读相关书籍来了解完整的 Web 原理，这将有助于编写更完善和安全的 Web 程序。

在下一章，我们会学习使用 Flask 的模板引擎——Jinja2，通过学习运用模板和静态文件，我们可以让程序变得更加丰富和完善。



# 模    板

在第1章里，当用户访问程序的根地址时，我们的视图函数会向客户端返回一行HTML代码。然而，一个完整的HTML页面往往需要几十行甚至上百行代码，如果都写到视图函数里，那可真是个噩梦。这样的代码既不简洁也难于维护，正确的做法是把HTML代码存储在单独的文件中，以便让程序的业务逻辑和表现逻辑分离，即控制器和用户界面的分离。

在动态Web程序中，视图函数返回的HTML数据往往需要根据相应的变量（比如查询参数）动态生成。当HTML代码保存到单独的文件中时，我们没法再使用字符串格式化或拼接字符串的方式来在HTML代码中插入变量，这时我们需要使用模板引擎（template engine）。借助模板引擎，我们可以在HTML文件中使用特殊的语法来标记出变量，这类包含固定内容和动态部分的可重用文件称为模板（template）。

模板引擎的作用就是读取并执行模板中的特殊语法标记，并根据传入的数据将变量替换为实际值，输出最终的HTML页面，这个过程被称为渲染（rendering）。Flask默认使用的模板引擎是Jinja2，它是一个功能齐全的Python模板引擎，除了设置变量，还允许我们在模板中添加if判断，执行for迭代，调用函数等，以各种方式控制模板的输出。对于Jinja2来说，模板可以是任何格式的纯文本文件，比如HTML、XML、CSV、LaTeX等。在这一章，我们会学习Jinja2模板引擎的基本用法和一些常用技巧。

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

```
$ cd demos/template
$ flask run
```

## 3.1 模板基本用法

这一节我们将以一个简单的例子来介绍如何使用Jinja2创建HTML模板，并在视图函数中



渲染模板，最终实现 HTML 响应的动态化。

### 3.1.1 创建模板

假设我们需要编写一个用户的电影清单页面，类似 IMDb 的 watchlist 页面的简易版，模板中要显示用户信息以及用户收藏的电影列表，包含电影的名字和年份。我们首先创建一些虚拟数据用于测试显示效果：

```
user = {
 'username': 'Grey Li',
 'bio': 'A boy who loves movies and music.',
}

movies = [
 {'name': 'My Neighbor Totoro', 'year': '1988'},
 {'name': 'Three Colours trilogy', 'year': '1993'},
 {'name': 'Forrest Gump', 'year': '1994'},
 {'name': 'Perfect Blue', 'year': '1997'},
 {'name': 'The Matrix', 'year': '1999'},
 {'name': 'Memento', 'year': '2000'},
 {'name': 'The Bucket list', 'year': '2007'},
 {'name': 'Black Swan', 'year': '2010'},
 {'name': 'Gone Girl', 'year': '2014'},
 {'name': 'CoCo', 'year': '2017'},
]
```

我们在 templates 目录下创建一个 watchlist.html 作为模板文件，然后使用 Jinja2 支持的语法在模板中操作这些变量，如代码清单 3-1 所示。

代码清单 3-1 template/watchlist.html：电影清单模板

---

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="utf-8">
 <title>{{ user.username }}'s Watchlist</title>
</head>
<body>
 ← Return
 <h2>{{ user.username }}</h2>
 {% if user.bio %}
 <i>{{ user.bio }}</i>
 {% else %}
 <i>This user has not provided a bio.</i>
 {% endif %}

 <h3>{{ user.username }}'s Watchlist ({{ movies|length }}):</h3>

 {% for movie in movies %}
 {{ movie.name }} - {{ movie.year }}
 {% endfor %}

</body>
</html>
```

---



提示 这里创建了一个基础的 HTML 文档结构，关于 HTML5 的结构组成，你可以访问 [https://www.w3.org/wiki/HTML\\_structural\\_elements](https://www.w3.org/wiki/HTML_structural_elements) 了解。

提示 在模板中使用的 &lt; 是我们第 2 章提及的 HTML 实体。HTML 实体除了用来转义 HTML 保留符号外，通常会被用来显示不容易通过键盘输入的字符。这里的 &lt; 会显示为左箭头，另外，我们还经常使用 &copy; 来显示版权标志，你可以访问 <https://dev.w3.org/html5/html-author/charref> 查看所有可用的 HTML 实体。

在模板中添加 Python 语句和表达式时，我们需要使用特定的定界符把它们标示出来。watchlist.html 中涉及的模板语法，我们会在下面逐一介绍。首先，你可以在上面的代码中看到 Jinja2 里常见的三种定界符：

#### (1) 语句

比如 if 判断、for 循环等：

```
{% ... %}
```

#### (2) 表达式

比如字符串、变量、函数调用等：

```
{{ ... }}
```

#### (3) 注释

```
{# ... #}
```

另外，在模板中，Jinja2 支持使用 “.” 获取变量的属性，比如 user 字典中的 username 键值通过 “.” 获取，即 user.username，在效果上等同于 user['username']。

### 3.1.2 模板语法

利用 Jinja2 这样的模板引擎，我们可以将一部分的程序逻辑放到模板中去。简单地说，我们可以在模板中使用 Python 语句和表达式来操作数据的输出。但需要注意的是，Jinja2 并不支持所有 Python 语法。而且出于效率和代码组织等方面的考虑，我们应该适度使用模板，仅把和输出控制有关的逻辑操作放到模板中。

Jinja2 允许你在模板中使用大部分 Python 对象，比如字符串、列表、字典、元组、整型、浮点型、布尔值。它支持基本的运算符号 (+、-、\*、/ 等)、比较符号 (比如 ==、!= 等)、逻辑符号 (and、or、not 和括号) 以及 in、is、None 和布尔值 (True、False)。

Jinja2 提供了多种控制结构来控制模板的输出，其中 for 和 if 是最常用的两种。在 Jinja2 里，语句使用 {% ... %} 标识，尤其需要注意的是，在语句结束的地方，我们必须添加结束标签：

```
{% if user.bio %}
 <i>{{ user.bio }}</i>
{% else %}
 <i>This user has not provided a bio.</i>
{% endif %}
```



在这个 If 语句里，如果 user.bio 已经定义，就渲染 {%- if user.bio %} 和 {%- else %} 之间的内容，否则就渲染 {%- else %} 和 {%- endif %} 之间的默认内容。末尾的 {%- endif %} 用来声明 if 语句的结束，这一行不能省略。

和在 Python 里一样，for 语句用来迭代一个序列：

```

 {% for movie in movies %}
 {{ movie.name }} - {{ movie.year }}
 {% endfor %}

```

和其他语句一样，你需要在 for 循环的结尾使用 endfor 标签声明 for 语句的结束。在 for 循环内，Jinja2 提供了多个特殊变量，常用的循环变量如表 3-1 所示。

表 3-1 常用的 Jinja2 for 循环特殊变量

变 量 名	说 明
loop.index	当前迭代数（从 1 开始计数）
loop.index0	当前迭代数（从 0 开始计数）
loop.revindex	当前反向迭代数（从 1 开始计数）
loop.revindex0	当前反向迭代数（从 0 开始计数）
loop.first	如果是第一个元素，则为 True
loop.last	如果是最后一个元素，则为 True
loop.previtem	上一个迭代的条目
loop.nextitem	下一个迭代的条目
loop.length	序列包含的元素数量



附注 完整的 for 循环变量列表请访问 <http://jinja.pocoo.org/docs/2.10/templates/#for> 查看。

### 3.1.3 渲染模板

渲染一个模板，就是执行模板中的代码，并传入所有在模板中使用的变量，渲染后的结果就是我们要返回给客户端的 HTML 响应。在视图函数中渲染模板时，我们并不直接使用 Jinja2 提供的函数，而是使用 Flask 提供的渲染函数 render\_template()，如代码清单 3-2 所示。

代码清单3-2 template/app.py：渲染HTML模板

```
from flask import Flask, render_template
...
@app.route('/watchlist')
def watchlist():
 return render_template('watchlist.html', user=user, movies=movies)
```

在 render\_template() 函数中，我们首先传入模板的文件名作为参数。如第 1 章项目结构部



分所说，Flask 会在程序根目录下的 templates 文件夹里寻找模板文件，所以这里传入的文件路径是相对于 templates 根目录的。除了模板文件路径，我们还以关键字参数的形式传入了模板中使用的变量值，以 user 为例：左边的 user 表示传入模板的变量名称，右边的 user 则是要传入的对象。

提示 除了 render\_template() 函数，Flask 还提供了一个 render\_template\_string() 函数用来渲染模板字符串。

其他类型的变量通过相同的方式传入。传入 Jinja2 中的变量值可以是字符串、列表和字典，也可以是函数、类和类实例，这完全取决于你在视图函数传入的值。下面是一些示例：

```
<p>这是列表my_list的第一个元素: {{ my_list[0] }}</p>
<p>这是元组my_tuple的第一个元素: {{ my_tuple[0] }}</p>
<p>这是字典my_dict的键为name的值: {{ my_dict['name'] }}</p>
<p>这是函数my_func的返回值: {{ my_func() }}</p>
<p>这是对象my_object调用某方法的返回值: {{ my_object.name() }}</p>
```

如果你想传入函数在模板中调用，那么需要传入函数对象本身，而不是函数调用（函数的返回值），所以仅写出函数名称即可。当把函数传入模板后，我们可以像在 Python 脚本中一样通过添加括号的方式调用，而且你也可以在括号中传入参数。

根据我们传入的虚拟数据，render\_template() 渲染后返回的 HTML 数据如下所示：

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="utf-8">
 <title>Grey Li's Watchlist</title>
</head>
<body>
← Return
<h2>Grey Li</h2>
<i>A boy who loves movies and music.</i>
<h5>Grey Li's Watchlist (10):</h5>

 My Neighbor Totoro - 1988
 Three Colours trilogy - 1993
 Forrest Gump - 1994
 Perfect Blue - 1997
 The Matrix - 1999
 Memento - 2000
 The Bucket list - 2007
 Black Swan - 2010
 Gone Girl - 2014
 CoCo - 2017

</body>
</html>
```

在和渲染前的模板文件对比时你会发现，原模板中所有的 Jinja2 语句、表达式、注释都会在执行后被移除，而所有的变量都会被替换为对应的数据。访问 <http://localhost:5000/watchlist> 即可看到渲染后的页面，如图 3-1 所示。

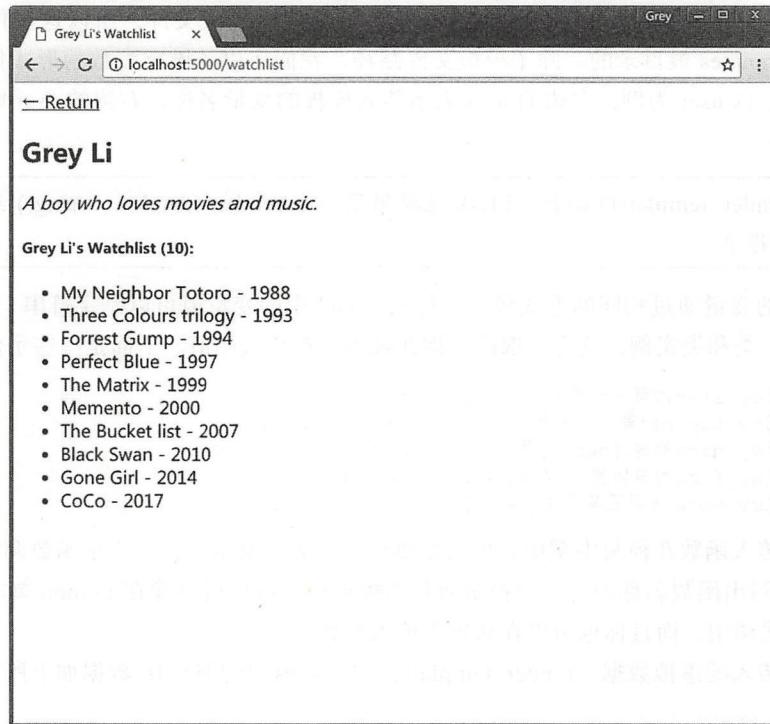


图 3-1 渲染后的页面

## 3.2 模板辅助工具

除了基本语法，Jinja2 还提供了许多方便的工具，这些工具可以让你更方便地控制模板的输出。为了方便测试，我们在示例程序的 templates 目录下创建了一个根页面模板 index.html。返回主页的 index 视图和 watchlist 视图类似：

```
from flask import render_template

@app.route('/')
def index():
 return render_template('index.html')
```

### 3.2.1 上下文

模板上下文包含了很多变量，其中包括我们调用 `render_template()` 函数时手动传入的变量以及 Flask 默认传入的变量。

除了渲染时传入变量，你也可以在模板中定义变量，使用 `set` 标签：

```
{% set navigation = [('/', 'Home'), ('/about', 'About')] %}
```

你也可以将一部分模板数据定义为变量，使用 `set` 和 `endset` 标签声明开始和结束：



```
{% set navigation %}
 Home
 About
{% endset %}
```

## 1. 内置上下文变量

Flask 在模板上下文中提供了一些内置变量，可以在模板中直接使用，如表 3-2 所示。

表 3-2 标准模板全局变量

变 量	说 明
config	当前的配置对象
request	当前的请求对象，在已激活的请求环境下可用
session	当前的会话对象，在已激活的请求环境下可用
g	与请求绑定的全局变量，在已激活的请求环境下可用

## 2. 自定义上下文

如果多个模板都需要使用同一变量，那么比起在多个视图函数中重复传入，更好的方法是能够设置一个模板全局变量。Flask 提供了一个 `app.context_processor` 装饰器，可以用来注册模板上下文处理函数，它可以帮助我们完成统一传入变量的工作。模板上下文处理函数需要返回一个包含变量键值对的字典，如代码清单 3-3 所示。

代码清单3-3 注册模板上下文处理函数

---

```
@app.context_processor
def inject_foo():
 foo = 'I am foo.'
 return dict(foo=foo) # 等同于return {'foo': foo}
```

---

当我们调用 `render_template()` 函数渲染任意一个模板时，所有使用 `app.context_processor` 装饰器注册的模板上下文处理函数（包括 Flask 内置的上下文处理函数）都会被执行，这些函数的返回值会被添加到模板中，因此我们可以在模板中直接使用 `foo` 变量。

 和在 `render_template()` 函数中传入变量类似，除了字符串、列表等数据结构，你也可以传入函数、类或类实例。

除了使用 `app.context_processor` 装饰器，也可以直接将其作为方法调用，传入模板上下文处理函数：

```
...
def inject_foo():
 foo = 'I am foo.'
 return dict(foo=foo)

app.context_processor(inject_foo)
```

使用 `lambda` 可以简化为：



```
app.context_processor(lambda: dict(foo='I am foo.'))
```

### 3.2.2 全局对象

全局对象是指在所有的模板中都可以直接使用的对象，包括在模板中导入的模板，后面我们会详细介绍导入的概念。

#### 1. 内置全局函数

Jinja2 在模板中默认提供了一些全局函数，常用的三个函数如表 3-3 所示。

表 3-3 Jinja2 内置模板全局函数

函 数	说 明
range([start, ]stop[, step])	和 Python 中的 range() 用法相同
lipsum(n=5, html=True, min=20, max=100)	生成随机文本 (lorem ipsum)，可以在测试时用来填充页面。默认生成 5 段 HTML 文本，每段包含 20~100 个单词
dict(**items)	和 Python 中的 dict() 用法相同

 **附注** 这里只列出了部分常用的全局函数，完整的全局函数列表请访问 <http://jinja.pocoo.org/docs/2.10/templates/#list-of-global-functions> 查看。

除了 Jinja2 内置的全局函数，Flask 也在模板中内置了两个全局函数，如表 3-4 所示。

表 3-4 Flask 内置模板全局函数

函 数	说 明
url_for()	用于生成 URL 的函数
get_flashed_messages()	用于获取 flash 消息的函数

 **提示** Flask 除了把 g、session、config、request 对象注册为上下文变量，也将它们设为全局变量，因此可以全局使用。

url\_for() 用来获取 URL，用法和在 Python 脚本中相同。在前面给出的 watchlist.html 模板中，用来返回主页的链接直接写出。在实际的代码中，这个 URL 使用 url\_for() 生成，传入 index 视图的端点：

```
< Return
```

get\_flashed\_messages() 的用法我们会在后面介绍。

#### 2. 自定义全局函数

除了使用 app.context\_processor 注册模板上下文处理函数来传入函数，我们也可以使用 app.template\_global 装饰器直接将函数注册为模板全局函数。比如，代码清单 3-4 把 bar() 函数注册为模板全局函数。



## 代码清单3-4 template/app.py：注册模板全局函数

---

```
@app.template_global()
def bar():
 return 'I am bar.'
```

---

默认使用函数的原名称传入模板，在`app.template_global()`装饰器中使用`name`参数可以指定一个自定义名称。`app.template_global()`仅能用于注册全局函数，后面我们会介绍如何注册全局变量。

**附注** 你可以直接使用`app.add_template_global()`方法注册自定义全局函数，传入函数对象和可选的自定义名称(`name`)，比如`app.add_template_global(your_global_function)`。

---

### 3.2.3 过滤器

在Jinja2中，过滤器(filter)是一些可以用来修改和过滤变量值的特殊函数，过滤器和变量用一个竖线(管道符号)隔开，需要参数的过滤器可以像函数一样使用括号传递。下面是一个对`name`变量使用`title`过滤器的例子：

```
{{ name|title }}
```

这会将`name`变量的值标题化，相当于在Python里调用`name.title()`。再比如，我们在本章开始的示例模板`watchlist.html`中使用`length`获取`movies`列表的长度，类似于在Python中调用`len(movies)`：

```
{{ movies|length }}
```

另一种用法是将过滤器作用于一部分模板数据，使用`filter`标签和`endfilter`标签声明开始和结束。比如，下面使用`upper`过滤器将一段文字转换为大写：

```
{% filter upper %}
 This text becomes uppercase.
{% endfilter %}
```

#### 1. 内置过滤器

Jinja2提供了许多内置过滤器，常用的过滤器如表3-5所示。

表3-5 Jinja2常用内置过滤器

过滤器	说明
<code>default(value, default_value=u'', boolean=False)</code>	设置默认值，默认值作为参数传入，别名为 <code>d</code>
<code>escape(s)</code>	转义HTML文本，别名为 <code>e</code>
<code>first(seq)</code>	返回序列的第一个元素
<code>last(seq)</code>	返回序列的最后一个元素
<code>length(object)</code>	返回变量的长度

(续)

过滤器	说 明
random(seq)	返回序列中的随机元素
safe(value)	将变量值标记为安全，避免转义
trim(value)	清除变量值前后的空格
max(value, case_sensitive=False, attribute=None)	返回序列中的最大值
min(value, case_sensitive=False, attribute=None)	返回序列中的最小值
unique(value, case_sensitive=False, attribute=None)	返回序列中的不重复的值
striptags(value)	清除变量值内的 HTML 标签
urllize (value, trim_url_limit=None, nofollow=False, target=None, rel=None)	将 URL 文本转换为可单击的 HTML 链接
wordcount (s)	计算单词数量
tojson(value, indent=None)	将变量值转换为 JSON 格式
truncate(s, length=255, killwords=False, end='...', leeway=None)	截断字符串，常用于显示文章摘要，length 参数设置截断的长度，killwords 参数设置是否截断单词，end 参数设置结尾的符号



这里只列出了一部分常用的过滤器，完整的列表请访问 <http://jinja.pocoo.org/docs/2.10/templates/#builtin-filters> 查看。

在使用过滤器时，列表中过滤器函数的第一个参数表示被过滤的变量值（value）或字符串（s），即竖线符号左侧的值，其他的参数可以通过添加括号传入。

另外，过滤器可以叠加使用，下面的示例为 name 变量设置默认值，并将其标题化：

```
<h1>Hello, {{ name|default('陌生人')|title }}!</h1>
```

在第 2 章，我们介绍了 XSS 攻击的主要防范措施，其中最主要的是对用户输入的文本进行转义。根据 Flask 的设置，Jinja2 会自动对模板中的变量进行转义，所以我们不用手动使用 escape 过滤器或调用 escape() 函数对变量进行转义。



默认的自动开启转义仅针对 .html、.htm、.xml 以及 .xhtml 后缀的文件，用于渲染模板字符串的 render\_template\_string() 函数也会对所有传入的字符串进行转义。

在确保变量值安全的情况下，这通常意味着你已经对用户输入的内容进行了“消毒”处理。这时如果你想避免转义，将变量作为 HTML 解析，可以对变量使用 safe 过滤器：

```
{{ sanitized_text|safe }}
```

另一种将文本标记为安全的方法是在渲染前将变量转换为 Markup 对象：

```
from flask import Markup
```

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