快速上手

等久了吧？本文会给你好好介绍如何上手 Flask 。 这里假定你已经安装好了 Flask ， 否则请先阅读《 安装 》。

一个最小的应用

一个最小的 Flask 应用如下:

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

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

那么，这些代码是什么意思呢？

1. 首先我们导入了 **Flask** 类。 该类的实例将会成为我们的 WSGI 应用。
2. 接着我们创建一个该类的实例。第一个参数是应用模块或者包的名称。如果你使用 一个单一模块（就像本例），那么应当使用 `__name__` ，因为名称会根据这个 模块是按应用方式使用还是作为一个模块导入而发生变化（可能是 ‘__main__’ ， 也可能是实际导入的名称）。这个参数是必需的，这样 Flask 才能知道在哪里可以 找到模板和静态文件等东西。更多内容详见 **Flask** 文档。
3. 然后我们使用 **route()** 装饰器来告诉 Flask 触发函数的 URL 。
4. 函数名称被用于生成相关联的 URL 。函数最后返回需要在用户浏览器中显示的信息。

把它保存为 `hello.py` 或其他类似名称。请不要使用 `flask.py` 作为应用名称，这会与 Flask 本身发生冲突。

可以使用 **flask** 命令或者 python 的 `-m` 开关来运行这个应用。在 运行应用之前，需要在终端里导出 `FLASK_APP` 环境

```
$ export FLASK_APP=hello.py
$ flask run
 * Running on http://127.0.0.1:5000/
```

如果是在 Windows 下，那么导出环境变量的语法取决于使用的是哪种命令行解释器。 在 Command Prompt 下:

```
C:\path\to\app>set FLASK_APP=hello.py
```

在 PowerShell 下:

```
PS C:\path\to\app> $env:FLASK_APP = "hello.py"
```

还可以使用 **python -m flask**:

```
$ export FLASK_APP=hello.py
$ python -m flask run
 * Running on http://127.0.0.1:5000/
```

这样就启动了一个非常简单的内建的服务器。这个服务器用于测试应该是足够了，但是 用于生产可能是不够的。关于部署的有关内容参见《 部署方式 》。

现在在浏览器中打开 http://127.0.0.1:5000/ ，应该可以看到 Hello World! 字样。

外部可见的服务器:

运行服务器后，会发现只有你自己的电脑可以使用服务，而网络中的其他电脑却 不行。缺省设置就是这样的，因为在调试模式下该应用的用户可以执行你电脑中 的任意 Python 代码。

如果你关闭了调试器或信任你网络中的用户，那么可以让服务器被公开访问。 只要在命令行上简单的加上 `--host=0.0.0.0` 即可:


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

这行代码告诉你的操作系统监听所有公开的 IP 。

如果服务器不能启动怎么办

假如运行 **python -m flask** 命令失败或者 **flask** 命令不存在， 那么可能会有多种原因导致失败。首先应该检查错误信息。

老版本的 Flask

版本低于 0.11 的 Flask ，启动应用的方式是不同的。简单的说就是 **flask** 和 **python -m flask** 命令都无法使用。在这种情况有两个选择：一是升级 Flask 到更新的版本，二是参阅《 开发服务器 》，学习其他 启动服务器的方法。

非法导入名称

`FLASK_APP` 环境变量中储存的是模块的名称，运行 **flask run** 命令就 会导入这个模块。如果模块的名称不对，那么就会出现导入错误。出现错误的时机是在 应用开始的时候。如果调试模式打开的情况下，会在运行到应用开始的时候出现导入 错误。出错信息会告诉你尝试导入哪个模块时出错，为什么会出错。

最常见的错误是因为拼写错误而没有真正创建一个 `app` 对象。

调试模式

（只需要记录出错信息和追踪堆栈？参见 应用错误处理 ）

虽然 **flask** 命令可以方便地启动一个本地开发服务器，但是每次应用代码 修改之后都需要手动重启服务器。这样不是很方便， Flask 可以做得更好。如果你打开 调试模式，那么服务器会在修改应用代码之后自动重启，并且当应用出错时还会提供一个 有用的调试器。

如果需要打开所有开发功能（包括调试模式），那么要在运行服务器之前导出 `FLASK_ENV` 环境变量并把其设置为 `development`:

```
$ export FLASK_ENV=development
$ flask run
```

（在 Windows 下需要使用 `set` 来代替 `export` 。）

这样可以实现以下功能：

1. 激活调试器。
2. 激活自动重载。
3. 打开 Flask 应用的调试模式。

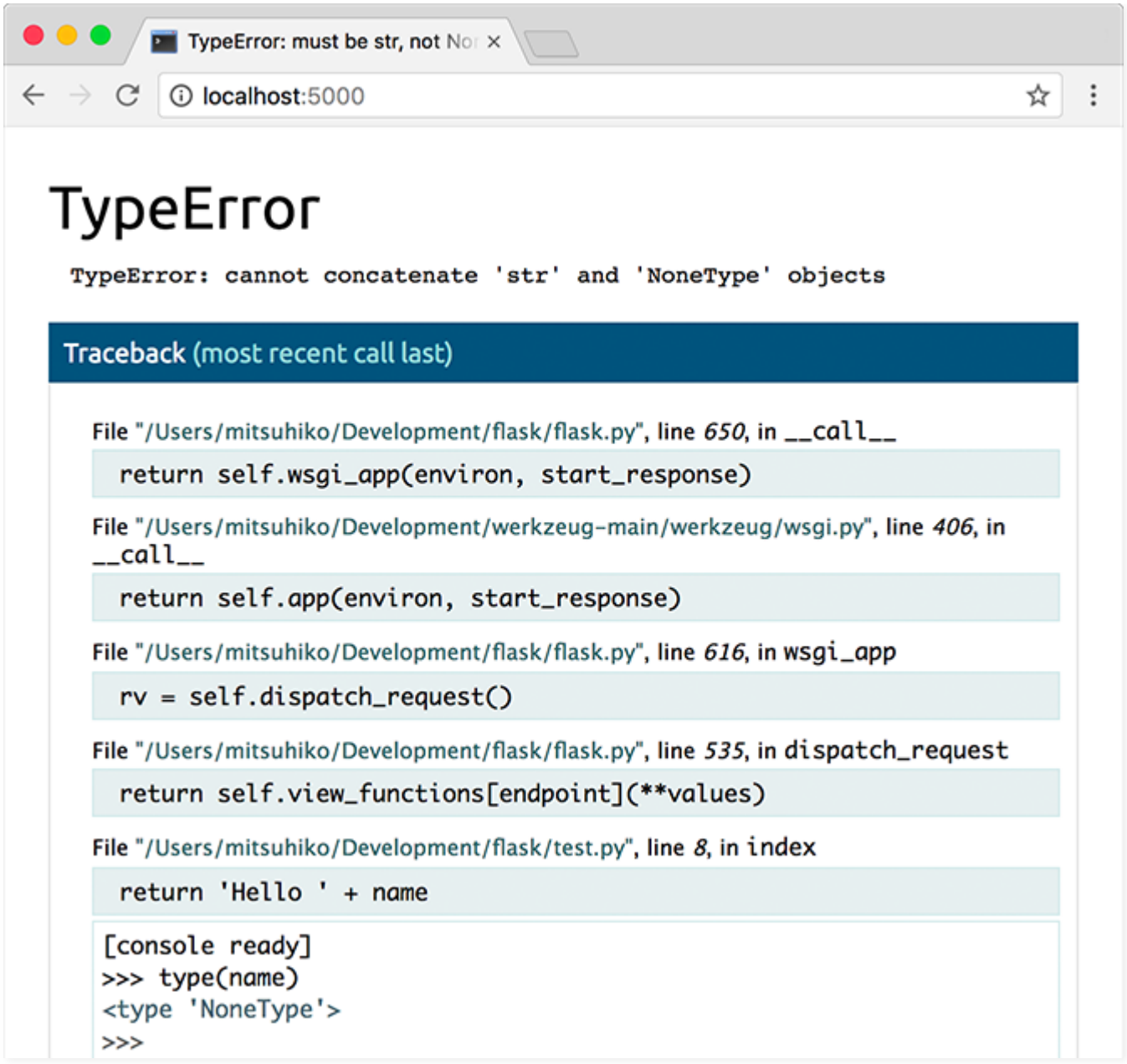
还可以通过导出 `FLASK_DEBUG=1` 来单独控制调试模式的开关。

开发服务器 文档中还有更多的参数说明。

Attention:

虽然交互调试器不能在分布环境下工作（这使得它基本不可能用于生产环境），但是 它允许执行任意代码，这样会成为一个重大安全隐患。因此， **绝对不能在生产环境 中使用调试器** 。

运行中的调试器截图：



更多关于调试器的信息参见 Werkzeug 文档 。

想使用其他调试器？请参阅 使用调试器 。

路由

现代 web 应用都使用有意义的 URL ，这样有助于用户记忆，网页会更得到用户的青睐， 提高回头率。

使用 **route()** 装饰器来把函数绑定到 URL:

```
@app.route('/')
def index():
    return 'Index Page'

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

但是能做的不仅仅是这些！你可以动态变化 URL 的某些部分， 还可以为一个函数指定多个规则。

变量规则

通过把 URL 的一部分标记为 `<variable_name>` 就可以在 URL 中添加变量。标记的 部分会作为关键字参数传递给函数。通过使用 `<converter:variable_name>` ，可以 选择性的加上一个转换器，为变量指定规则。请看下面的例子:

```
@app.route('/user/<username>')
def show_user_profile(username):
    # show the user profile for that user
    return 'User %s' % escape(username)

@app.route('/post/<int:post_id>')
def show_post(post_id):
    # show the post with the given id, the id is an integer
    return 'Post %d' % post_id

@app.route('/path/<path:subpath>')
def show_subpath(subpath):
```

```
    # show the subpath after /path/
    return 'Subpath %s' % escape(subpath)
```

转换器类型：

`string`	（缺省值） 接受任何不包含斜杠的文本
`int`	接受正整数
`float`	接受正浮点数
`path`	类似 `string` ， 但可以包含斜杠
`uuid`	接受 UUID 字符串

唯一的 URL / 重定向行为

以下两条规则的不同之处在于是否使用尾部的斜杠。：

```
@app.route('/projects/')
def projects():
    return 'The project page'

@app.route('/about')
def about():
    return 'The about page'
```

`projects` 的 URL 是中规中矩的，尾部有一个斜杠，看起来就如同一个文件夹。 访问一个没有斜杠结尾的 URL 时 Flask 会自动进行重定向，帮你在尾部加上一个斜杠。

`about` 的 URL 没有尾部斜杠，因此其行为表现与一个文件类似。如果访问这个 URL 时添加了尾部斜杠就会得到一个 404 错误。这样可以保持 URL 唯一，并帮助 搜索引擎避免重复索引同一页面。

URL 构建

url_for() 函数用于构建指定函数的 URL。它把函数名称作为第一个 参数。它可以接受任意个关键字参数，每个关键字参数对应 URL 中的变量。未知变量 将添加到 URL 中作为查询参数。

为什么不在把 URL 写死在模板中，而要使用反转函数 **url_for()** 动态构建？

1. 反转通常比硬编码 URL 的描述性更好。
2. 你可以只在一个地方改变 URL ，而不用到处乱找。
3. URL 创建会为你处理特殊字符的转义和 Unicode 数据，比较直观。
4. 生产的路径总是绝对路径，可以避免相对路径产生副作用。
5. 如果你的应用是放在 URL 根路径之外的地方（如在 `/myapplication` 中，不在 `/` 中）， **url_for()** 会为你妥善处理。

例如，这里我们使用 **test_request_context()** 方法来尝试使用 **url_for()** 。 **test_request_context()** 告诉 Flask 正在处理一个请求，而实际上也许我们正处在交互 Python shell 之中， 并没有真正的请求。参见 本地环境 。

```
from flask import Flask, escape, url_for

app = Flask(__name__)

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

@app.route('/login')
def login():
    return 'login'

@app.route('/user/<username>')
def profile(username):
    return '{}\'s profile'.format(escape(username))

with app.test_request_context():
    print(url_for('index'))
    print(url_for('login'))
    print(url_for('login', next='/'))
    print(url_for('profile', username='John Doe'))
```

```
/
/login
/login?next=/
/user/John%20Doe
```

HTTP 方法

Web 应用使用不同的 HTTP 方法处理 URL 。当你使用 Flask 时，应当熟悉 HTTP 方法。 缺省情况下，一个路由只回应 `GET` 请求。 可以使用 **`route()`** 装饰器的 `methods` 参数来处理不同的 HTTP 方法:

```
from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        return do_the_login()
    else:
        return show_the_login_form()
```

如果当前使用了 GET 方法， Flask 会自动添加 `HEAD` 方法支持，并且同时还会 按照 HTTP RFC 来处理 `HEAD` 请求。同样，`OPTIONS` 也会自动实现。

静态文件

动态的 web 应用也需要静态文件，一般是 CSS 和 JavaScript 文件。理想情况下你的 服务器已经配置好了为你的提供静态文件的服务。但是在开发过程中， Flask 也能做好 这项工作。只要在你的包或模块旁边创建一个名为 `static` 的文件夹就行了。静态文件位于应用的 `/static` 中。

使用特定的 `'static'` 端点就可以生成相应的 URL


```
url_for('static', filename='style.css')
```

这个静态文件在文件系统中的位置应该是 `static/style.css` 。

渲染模板

在 Python 内部生成 HTML 不好玩，且相当笨拙。因为你必须自己负责 HTML 转义， 以确保应用的安全。因此， Flask 自动为你配置 Jinja2 模板引擎。

使用 **`render_template()`** 方法可以渲染模板，你只要提供模板名称和需要 作为参数传递给模板的变量就行了。下面是一个简单的模板渲染例子:

```
from flask import render_template

@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('hello.html', name=name)
```

Flask 会在 `templates` 文件夹内寻找模板。因此，如果你的应用是一个模块， 那么模板文件夹应该在模块旁边；如果是一个包，那么就应该在包里面：

情形 1 : 一个模块:

```
/application.py
/templates
    /hello.html
```

情形 2 : 一个包:

```
/application
    /__init__.py
    /templates
        /hello.html
```

你可以充分使用 Jinja2 模板引擎的威力。更多内容，详见官方 Jinja2 模板文档 。

模板示例：

```
<!doctype html>
<title>Hello from Flask</title>
{% if name %}
  <h1>Hello {{ name }}!</h1>
{% else %}
  <h1>Hello, World!</h1>
{% endif %}
```

在模板内部可以和访问 **get_flashed_messages()** 函数一样访问 **request** 、 **session** 和 **g** [1] 对象。

模板在继承使用的情况下尤其有用。其工作原理参见 模板继承 方案文档。简单的说，模板继承可以使每个页面的特定元素（如页头、导航和页尾） 保持一致。

自动转义默认开启。因此，如果 name 包含 HTML ，那么会被自动转义。如果你可以 信任某个变量，且知道它是安全的 HTML （例如变量来自一个把 wiki 标记转换为 HTML 的模块），那么可以使用 **Markup** 类把它标记为安全的，或者在模板中使用 |safe 过滤器。更多例子参见 Jinja 2 文档。

下面 **Markup** 类的基本使用方法:

```
>>> from flask import Markup
>>> Markup('<strong>Hello %s!</strong>') % '<blink>hacker</blink>'
Markup(u'<strong>Hello &lt;blink&gt;hacker&lt;/blink&gt;!</strong>')
>>> Markup.escape('<blink>hacker</blink>')
Markup(u'&lt;blink&gt;hacker&lt;/blink&gt;')
>>> Markup('<em>Marked up</em> &raquo; HTML').striptags()
u'Marked up \xbb HTML'
```

▶ *Changelog*

[1] 不理解什么是 **g** 对象？它是某个可以根据需要储存信息的 东西。更多信息参见 **g** 对象的文档和 使用 SQLite 3 文档。

操作请求数据

对于 web 应用来说对客户端向服务器发送的数据作出响应很重要。在 Flask 中由全局 对象 **request** 来提供请求信息。如果你有一些 Python 基础，那么 可能 会奇怪：既然这个对象是全局的，怎么还能保持线程安全？答案是本地环境：

本地环境

内部信息:

如果你想了解工作原理和如何使用本地环境进行测试，那么请阅读本节， 否则可以跳过本节。

某些对象在 Flask 中是全局对象，但不是通常意义下的全局对象。这些对象实际上是 特定环境下本地对象的代理。真拗口！但还是很容易理解的。

设想现在处于处理线程的环境中。一个请求进来了，服务器决定生成一个新线程（或者 叫其他什么名称的东西，这个下层的东西能够处理包括线程在内的并发系统）。当 Flask 开始其内部请求处理时会把当前线程作为活动环境，并把当前应用和 WSGI 环境绑定到 这个环境（线程）。它以一种聪明的方式使得一个应用可以在不中断的情况下调用另一个 应用。

这对你有什么用？基本上你可以完全不必理会。这个只有在做单元测试时才有用。在测试 时会遇到由于没有请求对象而导致依赖于请求的代码会突然崩溃的情况。对策是自己创建 一个请求对象并绑定到环境。最简单的单元测试解决方案是使用 **test_request_context()** 环境管理器。通过使用 `with` 语句 可以绑定一个测试请求，以便于交互。例如:

```
from flask import request

with app.test_request_context('/hello', method='POST'):
    # now you can do something with the request until the
    # end of the with block, such as basic assertions:
    assert request.path == '/hello'
    assert request.method == 'POST'
```


另一种方式是把整个 WSGI 环境传递给 **request_context()** 方法:

```
from flask import request

with app.request_context(environ):
    assert request.method == 'POST'
```

请求对象

请求对象在 API 一节中有详细说明这里不细谈（参见 **Request** ）。 这里简略地谈一下最常见的操作。首先，你必须从 `flask` 模块导入请求对象:

```
from flask import request
```

通过使用 **method** 属性可以操作当前请求方法，通过使用 **form** 属性处理表单数据（在 `POST` 或者 `PUT` 请求 中传输的数据）。以下是使用上述两个属性的例子:

```
@app.route('/login', methods=['POST', 'GET'])
def login():
    error = None
    if request.method == 'POST':
        if valid_login(request.form['username'],
                       request.form['password']):
            return log_the_user_in(request.form['username'])
        else:
            error = 'Invalid username/password'
    # the code below is executed if the request method
    # was GET or the credentials were invalid
    return render_template('login.html', error=error)
```

当 `form` 属性中不存在这个键时会发生什么？会引发一个 **KeyError** 。 如果你不像捕捉一个标准错误一样捕捉 **KeyError** ，那么会显示一个 HTTP 400 Bad Request 错误页面。因此，多数情况下你不必处理这个问题。

要操作 URL （如 `?key=value` ）中提交的参数可以使用 **args** 属性:

```
searchword = request.args.get('key', '')
```

用户可能会改变 URL 导致出现一个 400 请求出错页面，这样降低了用户友好度。因此， 我们推荐使用 *get* 或通过捕捉 **KeyError** 来访问 URL 参数。

完整的请求对象方法和属性参见 **Request** 文档。

文件上传

用 Flask 处理文件上传很容易，只要确保不要忘记在你的 HTML 表单中设置 `enctype="multipart/form-data"` 属性就可以了。否则浏览器将不会传送你的文件。

已上传的文件被储存在内存或文件系统的临时位置。你可以通过请求对象 **files** 属性来访问上传的文件。每个上传的文件都储存在这个 字典型属性中。这个属性基本和标准 Python **file** 对象一样，另外多出一个 用于把上传文件保存到服务器的文件系统中的 **save()** 方法。下例展示其如何运作:

```
from flask import request

@app.route('/upload', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        f = request.files['the_file']
        f.save('/var/www/uploads/uploaded_file.txt')
    ...
```

如果想要知道文件上传之前其在客户端系统中的名称，可以使用 **filename** 属性。但是请牢记这个值是 可以伪造的，永远不要信任这个值。如果想要把客户端的文件名作为服务器上的文件名， 可以通过 Werkzeug 提供的 **secure_filename()** 函数:

```
from flask import request
from werkzeug.utils import secure_filename

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

```
    if request.method == 'POST':
        f = request.files['the_file']
        f.save('/var/www/uploads/' + secure_filename(f.filename))
    ...
```

更好的例子参见 上传文件 方案。

Cookies

要访问 cookies ， 可以使用 **cookies** 属性。可以使用响应 对象 的 **set_cookie** 方法来设置 cookies 。请求对象的 **cookies** 属性是一个包含了客户端传输的所有 cookies 的字典。在 Flask 中， 如果使用 会话 ， 那么就不要直接使用 cookies ， 因为 会话 比较安全一些。

读取 cookies:

```
from flask import request

@app.route('/')
def index():
    username = request.cookies.get('username')
    # use cookies.get(key) instead of cookies[key] to not get a
    # KeyError if the cookie is missing.
```

储存 cookies:

```
from flask import make_response

@app.route('/')
def index():
    resp = make_response(render_template(...))
    resp.set_cookie('username', 'the username')
    return resp
```

注意， cookies 设置在响应对象上。通常只是从视图函数返回字符串， Flask 会把它们 转换为响应对象。如果你想显式地转换，那么可以使用 **make_response()** 函数，然后再修改它。

使用 延迟的请求回调 方案可以在没有响应对象的情况下设置一个 cookie 。

同时可以参见 关于响应 。

重定向和错误

使用 **redirect()** 函数可以重定向。使用 **abort()** 可以 更早退出请求，并返回错误代码:

```
from flask import abort, redirect, url_for

@app.route('/')
def index():
    return redirect(url_for('login'))

@app.route('/login')
def login():
    abort(401)
    this_is_never_executed()
```

上例实际上是没有意义的，它让一个用户从索引页重定向到一个无法访问的页面（401 表示禁止访问）。但是上例可以说明重定向和出错跳出是如何工作的。

缺省情况下每种出错代码都会对应显示一个黑白的出错页面。使用 **errorhandler()** 装饰器可以定制出错页面:

```
from flask import render_template

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

注意 **render_template()** 后面的 404 ，这表示页面对就的出错 代码是 404 ，即页面不存在。缺省情况下 200 表示：一切正常。

详见 错误处理 。

关于响应

视图函数的返回值会自动转换为一个响应对象。如果返回值是一个字符串，那么会被 转换为一个包含作为响应体的字符串、一个 200 OK 出错代码 和一个 *text/html* 类型的响应对象。如果返回值是一个字典，那么会调用 **jsonify()** 来产生一个响应。以下是转换的规则：

1. 如果视图返回的是一个响应对象，那么就直接返回它。
2. 如果返回的是一个字符串，那么根据这个字符串和缺省参数生成一个用于返回的 响应对象。
3. 如果返回的是一个字典，那么调用 jsonify 创建一个响应对象。
4. 如果返回的是一个元组，那么元组中的项目可以提供额外的信息。元组中必须至少 包含一个项目，且项目应当由 (response, status) 、 (response, headers) 或者 (response, status, headers) 组成。 status 的值会重载状态代码， headers 是一个由额外头部值组成的列表 或字典。
5. 如果以上都不是，那么 Flask 会假定返回值是一个有效的 WSGI 应用并把它转换为 一个响应对象。

如果想要在视图内部掌控响应对象的结果，那么可以使用 **make_response()** 函数。

设想有如下视图:

```
@app.errorhandler(404)
def not_found(error):
    return render_template('error.html'), 404
```

可以使用 **make_response()** 包裹返回表达式，获得响应对象，并对该对象 进行修改，然后再返回:

```
@app.errorhandler(404)
def not_found(error):
    resp = make_response(render_template('error.html'), 404)
```

```
    resp.headers['X-Something'] = 'A value'
    return resp
```

JSON 格式的 API

JSON 格式的响应是常见的，用 Flask 写这样的 API 是很容易上手的。如果从视图 返回一个 `dict` ，那么它会被转换为一个 JSON 响应。

```
@app.route("/me")
def me_api():
    user = get_current_user()
    return {
        "username": user.username,
        "theme": user.theme,
        "image": url_for("user_image", filename=user.image),
    }
```

如果 `dict` 还不能满足需求，还需要创建其他类型的 JSON 格式响应，可以使用 **`jsonify()`** 函数。该函数会序列化任何支持的 JSON 数据类型。 也可以研究研究 Flask 社区扩展，以支持更复杂的应用。

```
@app.route("/users")
def users_api():
    users = get_all_users()
    return jsonify([user.to_json() for user in users])
```

会话

除了请求对象之外还有一种称为 **`session`** 的对象，允许你在不同请求 之间储存信息。这个对象相当于用密钥签名加密的 cookie ，即用户可以查看你的 cookie ，但是如果没有密钥就无法修改它。

使用会话之前你必须设置一个密钥。举例说明:

```
from flask import Flask, session, redirect, url_for, escape, request

app = Flask(__name__)

# Set the secret key to some random bytes. Keep this really secret!
app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'

@app.route('/')
def index():
    if 'username' in session:
        return 'Logged in as %s' % escape(session['username'])
    return 'You are not logged in'

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        session['username'] = request.form['username']
        return redirect(url_for('index'))
    return '''
        <form method="post">
            <p><input type=text name=username>
            <p><input type=submit value=Login>
        </form>
    '''

@app.route('/logout')
def logout():
    # remove the username from the session if it's there
    session.pop('username', None)
    return redirect(url_for('index'))
```

这里用到的 **escape()** 是用来转义的。如果不使用模板引擎就可以像上例 一样使用这个函数来转义。

如何生成一个好的密钥:

生成随机数的关键在于一个好的随机种子，因此一个好的密钥应当有足够的随机性。 操作系统可以有多种方式基于密码随机生成器来生成随机数据。使用下面的命令 可以快捷的为 **Flask.secret_key** （ 或者 **SECRET_KEY** ）生成值:

```
$ python -c 'import os; print(os.urandom(16))'
b'_5#y2L"F4Q8z\n\xec]/'
```

基于 cookie 的会话的说明： Flask 会取出会话对象中的值，把值序列化后储存到 cookie 中。在打开 cookie 的情况下，如果需要查找某个值，但是这个值在请求中 没有持续储存的话，那么不会得到一个清晰的出错信息。请检查页面响应中的 cookie 的大小是否与网络浏览器所支持的大小一致。

除了缺省的客户端会话之外，还有许多 Flask 扩展支持服务端会话。

消息闪现

一个好的应用和用户接口都有良好的反馈，否则到后来用户就会讨厌这个应用。 Flask 通过闪现系统来提供了一个易用的反馈方式。闪现系统的基本工作原理是在请求结束时 记录一个消息，提供且只提供给下一个请求使用。通常通过一个布局模板来展现闪现的 消息。

flash() 用于闪现一个消息。在模板中，使用 **get_flashed_messages()** 来操作消息。完整的例子参见 消息闪现 。

日志

▶ *Changelog*

有时候可能会遇到数据出错需要纠正的情况。例如因为用户篡改了数据或客户端代码出错 而导致一个客户端代码向服务器发送了明显错误的 HTTP 请求。多数时候在类似情况下 返回 `400 Bad Request` 就没事了，但也有不会返回的时候，而代码还得继续运行 下去。

这时候就需要使用日志来记录这些不正常的东西了。自从 Flask 0.3 后就已经为你配置好 了一个日志工具。

以下是一些日志调用示例:

```
app.logger.debug('A value for debugging')
app.logger.warning('A warning occurred (%d apples)', 42)
app.logger.error('An error occurred')
```

logger 是一个标准的 **Logger** Logger 类， 更多信息详见官方的 **logging** 文档。

更多内容请参阅 应用错误处理 。

集成 WSGI 中间件

如果想要在应用中添加一个 WSGI 中间件，那么可以包装内部的 WSGI 应用。假设为了 解决 lighttpd 的错误，你要使用一个来自 Werkzeug 包的中间件，那么可以这样做:

```
from werkzeug.contrib.fixers import LighttpdCGIRootFix
app.wsgi_app = LighttpdCGIRootFix(app.wsgi_app)
```

使用 Flask 扩展

扩展是帮助完成公共任务的包。例如 Flask-SQLAlchemy 为在 Flask 中轻松使用 SQLAlchemy 提供支持。

更多关于 Flask 扩展的内容请参阅 扩展 。

部署到网络服务器

已经准备好部署你的新 Flask 应用了？请移步 部署方式 。