Flask用户指南

极客学院出版

前言

Flask 是一个使用 Python 编写的轻量级 Web 应用框架。其 WSGI 工具箱采用 Werkzeug ，模板引擎则使用 Jinja2。

Flask 也被称为 “microframework” ，因为它使用简单的核心，用 extension 增加其他功能。Flask没有默认使用的数据库、窗体验证工具。然而，Flask保留了扩增的弹性，可以用Flask-extension加入这些功能：ORM、窗体验证工具、文件上传、各种开放式身份验证技术。

本文档分成几个部分，推荐您先读《安装》比快速上手文档更详细一点，该文档介绍了如何创建一个完整（尽管很小）的 Flask 应用。如果你想深入研究 Flask ，那么需要阅读《API》。《Flask 方案》中介绍了一些常用的解决方案。

Flask 依赖两个外部库： Jinja2 模板引擎和 Werkzeug WSGI 套件。这两个库的 使用不在本文档的范围内，欲知详情请移步：

- Jinja2 文档
- Werkzeug 文档

适用人群

本课程适用于希望通过 Flask 框架开发 Web 应用的开发人员。

学习前提

我们假定你在学习本课程之前对 Python 语言非常熟悉并掌握基本的 Web 开发知识。

版本信息

书中演示代码基于以下版本：

语言/框架	版本信息
Python	支持Python2.6 以上版本

> 原文地址: http://dormousehole.readthedocs.org/en/latest/
>
> 原文译者: dormouse young dormouse.young@yahoo.com
>
>

目录

1 前言

在使用 Flask 前请阅读本文。希望本文可以回答您有关 Flask 的用途和目的，以及是 否应当使用 Flask 等问题。

“微”是什么意思?

“微”并不代表整个应用只能塞在一个 Python 文件内，当然塞在单一文件内也是可以的。“微”也不代表 Flask 功能不强。微框架中的“微”字表示 Flask 的目标是保持核心既简单而又可扩展。 Flask 不会替你做出许多决定，比如选用何种数据库。类似的决定，如使用何种模板引擎，是非常容易改变的。 Flask 可以变成你任何想要的东西，一切恰到好处，由你做主。

缺省情况下， Flask 不包含数据库抽象层、表单验证或者其他已有的库可以处理的东西。 然而， Flask 通过扩展为你的应用添加这些功能，就如同这些功能是 Flask 原生的一样。 大量的扩展用以支持数据库整合、表单验证、上传处理和各种开放验证等等。Flask 可能是 “微小”的，但它已经为满足您的各种生产需要做出了充足的准备。

配置和惯例

刚起步的时候 Flask 有许多带有合理缺省值的配置值和惯例。按照惯例，模板和静态文件存放在应用的 Python 源代码树的子目录中，名称分别为 templates 和 static 。惯例是可以改变的，但是你大可不必改变，尤其是刚起步的时候。

Flask 可持续发展

一旦你开始使用 Flask ，你会发现有各种各样的扩展可供使用。 Flask 核心开发组会审查扩展，并保证通过检验的扩展可以在最新版本的 Flask 中可用。

随着你的代码库日益壮大，你可以自由地决定设计目标。 Flask 会一直提供一个非常简约而优秀的胶合层，就像 Python 语言一样。你可以自由地使用 SQLAlchemy 执行高级模式，或者使用其他数据库工具，亦可引入非关系数据模型，甚至还可以利用用于 Python 网络接口 WSGI 的非框架工具。

Flask 包含许多可以自定义其行为的钩子。考虑到你的定制需求， Flask 的类专为继承而打造。 如果对这一点感兴趣，请阅读大型应用 (页 0) 。

接下来请阅读安装或者针对高级程序员的前言 。

2

针对高级程序员的前言

Flask 中的本地线程对象

Flask 的设计原则之一是简单的任务不应当使用很多代码，应当可以简单地完成，但同时 又不应当把程序员限制得太死。因此，一些 Flask 的设计思路可能会让某些人觉得吃惊， 或者不可思议。例如， Flask 内部使用本地线程对象，这样就可以不用为了线程安全的 缘故在同一个请求中在函数之间传递对象。这种实现方法是非常便利的，但是当用于附属 注入或者当尝试重用使用与请求挂钩的值的代码时，需要一个合法的环境。 Flask 项目 对于本地线程是直言不讳的，没有一点隐藏的意思，并且在使用本地线程时在代码中进行 了标注和说明。

做网络开发时要谨慎

做网络应用开发时，安全要永记在心。

如果你开发了一个网络应用，那么可能会让用户注册并把他们的数据保存在服务器上。 用户把数据托付给了你。哪怕你的应用只是给自己用的，你也会希望数据完好无损。

不幸的是，网络应用的安全性是千疮百孔的，可以攻击的方法太多了。 Flask 可以防御 现代 Web 应用最常见的安全攻击：跨站代码攻击（ XSS ）。 Flask 和 下层的 Jinja2 模板引擎会保护你免受这种攻击，除非故意把不安全的 HTML 代码放进来。但是安全攻击 的方法依然还有很多。

这里警示你：在 web 开发过程中要时刻注意安全问题。一些安全问题远比想象的要复杂 得多。我们有时会低估程序的弱点，直到被一个聪明人利用这个弱点来攻击我们的程序。 不要以为你的应用不重要，还不足以别人来攻击。没准是自动化机器人用垃圾邮件或恶意 软件链接等东西来填满你宝贵的数据库。

Flask 与其他框架相同，你在开发时必须小心谨慎。

Python 3 的情况

目前， Python 社区正处在改进库的过程中，以便于加强对 Python 语言的新迭代的 支持。虽然现在情况已经有很大改善，但是还是存在一些问题使用户难以下决心现在就 转向 Python 3 。部分原因是 Python 语言中的变动长时间未经审核，还有部分原因是 我们还没有想好底层 API 针对 Python 3 中 unicode 处理方式的变化应该如何改动。

我们强烈建议你在开发过程中使用 Python 2.6 或者 Python 2.7 ，同时打开 Python 3 警告。如果你计划在近期升级到 Python 3 ，那么强烈推荐阅读如何编写向前兼容的 Python 代码 。

如果你一定要使用 Python 3 ，那么请先阅读 Python 3 支持 。

3

安装

Flask 依赖两个外部库： Werkzeug 和 Jinja2 。Werkzeug 是一个 WSGI 套件。 WSGI 是 Web 应用与 多种服务器之间的标准 Python 接口，即用于开发，也用于部署。 Jinja2 是用于渲染 模板的。

那么如何快速在你的计算机上装好所有东西？本节会介绍多种方法，但是最强力的方法是 使用 virtualenv 。因此，我们先说 virtualenv 。

无论使用哪种方法安装，你都会需要 Python 2.6 或更高版本。因此请确保安装了最新的 Python 2.x 版本。在 Python 3 下使用 Flask 请参阅 Python 3 支持 。

virtualenv

如果可以使用 shell，那么可能 Virtualenv 是你在开发环境和生产环境都想使用的东西。

virtualenv 有什么用? 如果你象我一样热爱 Python，那么除了基于 Flask 的项目外还会有其他项目用到 Python 。当项目越来越多时就会面对使用不同版本的 Python 的问题，或者至少会遇到使用不同版本的 Python 库的问题。摆在你面前的是：库常常不能向后兼容，更不幸的是任何成熟的应用都不是零依赖。如果两个项目依赖出现冲突， 怎么办?

Virtualenv 就是救星！它的基本原理是为每个项目安装一套 Python，多套 Python 并存。但它不是真正地安装多套独立的 Python 拷贝，而是使用了一种巧妙的方法让不同的项目处于各自独立的环境中。让我们来看看 virtualenv 是如何运行的!

如果你使用 Mac OS X 或 Linux，那么可以使用下面两条命令中任意一条:

```
$ sudo easy_install virtualenv
```

或更高级的:

```
$ sudo pip install virtualenv
```

上述命令中的任意一条就可以安装好 virtualenv 。也可以使用软件包管理器，在 Ubuntu 系统中可以试试:

```
$ sudo apt-get install python-virtualenv
```

如果你使用 Windows 且无法使用 easy_install，那么你必须先安装它，安装方法详见《 在 Windows 系统中使用 pip 和 distribute 》。安装好以后运行上述命令，但是要去掉 sudo 前缀。

安装完 virtualenv，打开一个 shell，创建自己的环境。我通常创建一个包含 venv 文件夹的项目文件夹:

```
$ mkdir myproject
$ cd myproject
$ virtualenv venv
New python executable in env/bin/python
Installing setuptools.....done.
```

现在，每次需要使用项目时，必须先激活相应的环境。在 OS X 和 Linux 系统中运行:

```
$ . venv/bin/activate
```

Windows 用户请运行下面的命令:

```
$ venv\scripts\activate
```

殊途同归，你现在进入你的 virtualenv（注意查看你的 shell 提示符已经改变了）。

现在可以开始在你的 virtualenv 中安装 Flask 了:

```
$ pip install Flask
```

几秒钟后就安装好了。

系统全局安装

虽然这样做是可行的，虽然我不推荐。只需要以 root 权限运行 pip 就可以了:

```
$ sudo pip install Flask
```

（ Windows 系统中请在管理员 shell 中运行，去掉 sudo ）。

生活在边缘

如果你想要使用最新版的 Flask ，那么有两种方法：要么使用 pip 安装开发版本， 要么使用 git 检索。无论哪种方法，都推荐使用 virtualenv 。

在一个新的 virtualenv 中获得 git 检索，并在开发模式下运行:

```
$ git clone http://github.com/mitsuhiko/flask.git
Initialized empty Git repository in ~/dev/flask/.git/
$ cd flask
$ virtualenv venv --distribute
New python executable in venv/bin/python
Installing distribute.............done.
$ . venv/bin/activate
$ python setup.py develop
...
Finished processing dependencies for Flask
```

上述操作会安装相关依赖库并在 virtualenv 中激活 git 头作为当前版本。然后只要使用 git pull origin 命令就可以安装最新版本的 Flask 了。

如果不使用 git ，那么可以这样获得开发版本:

```
$ mkdir flask
$ cd flask
$ virtualenv venv --distribute
$ . venv/bin/activate
New python executable in venv/bin/python
Installing distribute.............done.
$ pip install Flask==dev
...
Finished processing dependencies for Flask==dev
```

在 Windows 系统中使用 pip 和 distribute

在 Windows 系统中，安装 easy_install 稍微有点麻烦，但还是比较简单的。最简单的 方法是下载并运行 ez_setup.py 文件。最简单的运行文件的方法是打开下载文件所在 文件夹，双击这个文件。

接下来，通过把 Python 代码所在文件夹添加到 PATH 环境变量的方法把 easy_install 命令和其他 Python 代码添加到命令搜索目录。操作方法：用鼠标右键点击桌面上或者开始菜单中的“我的电脑”图标，在弹出的菜单中点击“属性”。然后 点击“高级系统设置”（如果是 Windows XP，则点击“高级”分页）。接着点击“环境变量”按钮，双击“系统变量”一节中的“ Path ”变量。这样就可以添加 Python 代码所在的文件夹了。 注意，与已经存在的值之间要用分号分隔。假设你在缺省路径安装了 Python 2.7，那么就应该添加如下内容:

```
;C:\Python27\Scripts
```

至此安装完成。要检验安装是否正确可以打开命令提示符，并运行 easy_install 命令。如果你使用 Windows Vista 或 Windows 7，并打开了权限控制，会提示你需要管理员权限。

至此，你安装好了 easy_install，接下来就可以用它来安装 pip 了:

```
> easy_install pip
```

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!'

if __name__ == '__main__':
    app.run()
```

把它保存为 hello.py 或其他类似名称并用你的 Python 解释器运行这个文件。请不要使用 flask.py 作为应用名称，这会与 Flask 本身发生冲突。

```
$ python hello.py
 * Running on http://127.0.0.1:5000/
```

现在，在浏览器中打开 `http://127.0.0.1:5000/` ，就可以看到问候页面了。

那么，这些代码是什么意思呢?

1. 首先我们导入了 Flask 类。这个类的实例将会成为我们的 WSGI 应用。
2. 接着我们创建了这个类的实例。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 就可以知道在哪里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. 函数名称可用于生成相关联的 URL ，并返回需要在用户浏览器中显示的信息。
5. 最后，使用 run() 函数来运行本地服务器和我们的应用。 `if __name__ == '__main__'` : 确保服务器只会在使用 Python 解释器运行代码的 情况下运行，而不会在作为模块导入时运行。

按 control-C 可以停止服务器。

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 或信任你网络中的用户，那么可以让服务器被公开访问。只要像这样改变 run() 方法的调用:

```
app.run(host='0.0.0.0')
```

这行代码告诉你的操作系统监听一个公开的 IP 。

调试模式

虽然 run() 方法可以方便地启动一个本地开发服务器，但是每次修改应用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。

打开调试模式有两种方法，一种是在应用对象上设置标志:

```
app.debug = True
app.run()
```

另一种是作为参数传递给 run 方法:

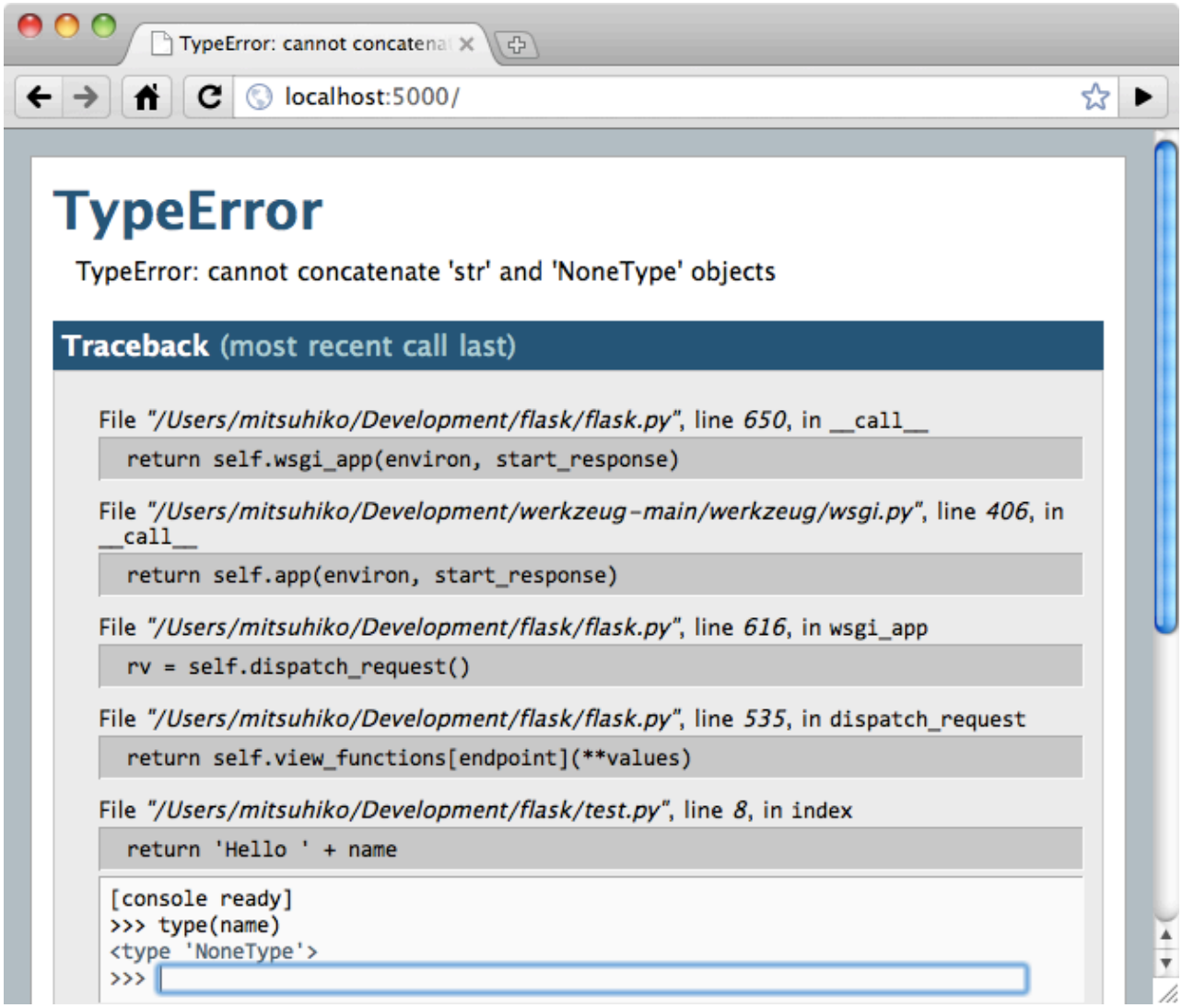
```
app.run(debug=True)
```

两种方法的效果相同。

注意

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** 。

运行的调试器的截图:



想使用其他调试器？请参阅使用调试器 。

路由

现代 web 应用都使用漂亮的 URL ，有助于人们记忆，对于使用网速较慢的移动设备尤其有利。如果用户可以不通过点击首页而直达所需要的页面，那么这个网页会更得到用户的青睐，提高回头率。

如前文所述， route() 装饰器用于把一个函数绑定到一个 URL 。 下面是一些基本的例子:

```
@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 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 ，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' % 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
```

现有的转换器有：

类型	说明
int	接受整数
float	接受浮点数
path	和缺省情况相同，但也接受斜杠

唯一的 URL / 重定向行为

Flask 的 URL 规则都是基于 Werkzeug 的路由模块的。其背后的理念是保证漂亮的外观和唯一的 URL 。这个理念来自于 Apache 和更早期的服务器。

假设有如下两条规则:

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

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

它们看上去很相近，不同之处在于 URL 定义 中尾部的斜杠。第一个例子中 prjects 的 URL 是中规中举的，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。

但是在第二个例子中， 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。

为什么这样做？因为这样可以在省略末尾斜杠时仍能继续相关的 URL 。这种重定向行为与 Apache 和其他服务器一致。同时， URL 仍保持唯一，帮助搜索引擎不重复索引同一页面。

URL 构建

如果可以匹配 URL ，那么 Flask 也可以生成 URL 吗？当然可以。 url_for() 函数就是用于构建指定函数的 URL 的。它把函数名称作为 第一个参数，其余参数对应 URL 中的变量。未知变量将添加到 URL 中作为查询参数。例如：

```
>>> from flask import Flask, url_for
>>> app = Flask(__name__)
>>> @app.route('/')
... def index(): pass
...
>>> @app.route('/login')
... def login(): pass
...
>>> @app.route('/user/<username>')
... def profile(username): pass
...
>>> 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
```

（例子中还使用下文要讲到的 test_request_context() 方法。这个方法的作用是告诉 Flask 我们正在处理一个请求，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。详见下面的本地环境 ）。

为什么不在把 URL 写死在模板中，反而要动态构建？有三个很好的理由：

1. 反向解析通常比硬编码 URL 更直观。同时，更重要的是你可以只在一个地方改变 URL ，而不用到处乱找。
2. URL 创建会为你处理特殊字符的转义和 Unicode 数据，不用你操心。
3. 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。

HTTP 方法

HTTP （ Web 应用使用的协议 ）协议中有访问 URL 的不同方法。缺省情况下，一个路由 只回应 GET 请求，但是可以通过 methods 参数使用不同方法。例如:

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

如果当前使用的是 GET 方法，会自动添加 HEAD ，你不必亲自操刀。同时还会确保 HEAD 请求按照 HTTP RFC （说明 HTTP 协议的文档 ）的要求来处理，因此你可以 完全忽略这部分 HTTP 规范。与 Flask 0.6 一样，OPTIONS 自动为你处理好。

完全不懂 HTTP 方法? 没关系，这里给你速成培训一下:

HTTP 方法（通常也被称为“动作”）告诉服务器一个页面请求要做 什么。以下是常见的方法:

GET

浏览器告诉服务器只要得到页面上的信息并发送这些信息。这可能是最常见的方法。

HEAD

浏览器告诉服务器想要得到信息，但是只要得到信息头 就行了，页面内容不要。 一个应用应该像接受到一个 GET 请求一样运行，但是不传递实际的内容。在 Flask 中，你根本不必理会这个，下层的 Werkzeug 库会为你处理好。

POST

浏览器告诉服务器想要向 URL 发表一些新的信息，服务器必须确保数据被保存好且只保存了一次。 HTML 表单实际上就是使用这个访求向服务器传送数据的。

PUT

与 POST 方法类似，不同的是服务器可能触发多次储存过程而把旧的值覆盖掉。你 可能会问这样做有什么用? 这样做是有原因的。假设在传输过程中连接丢失的情况 下，一个处于浏览器和服务器之间的系统可以在不中断的情况下安全地接收第二次请求。在这种情况下，使用 POST 方法就无法做到了，因为它只被触发一次。

DELETE

删除给定位置的信息。

OPTIONS

为客户端提供一个查询 URL 支持哪些方法的捷径。从 Flask 0.6 开始，自动为你实现了这个方法。 有趣的是在 HTML4 和 XHTML1 中，表单只能使用 GET 和 POST 方法。但是 JavaScript 和未来的 HTML 标准中可以使用其他的方法。此外， HTTP 近来已经变得相当 流行，浏览器不再只是唯一使用 HTTP 的客户端。比如许多版本控制系统也使用 HTTP 。

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。在开发过程中， Flask 也能做好这个工作。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。

使用选定的 '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 自动为你配置的 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 。

模板举例：

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

在模板内部你也可以访问 request 、session 和 g [1] 对象，以及 get_flashed_messages() 函数。

模板在继承使用的情况下尤其有用，其工作原理 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，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。否则请在模板中使用 |safe 过滤器。更多例子参见 Jinja 2 文档。

下面简单介绍一下 Markup 类的工作方式：

```
>>> 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'
```

Changed in version 0.5: 自动转义不再为所有模板开启，只为扩展名为 .html 、 .htm 、 .xml 和 .xhtml 开启。从字符串载入的模板将关闭自动转义。

[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 对象的文档和在 Flask 中使用 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

如果你想了解其工作原理和如何测试，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，但是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。以下是使用两个属性的例子:

```
@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'
    # 如果请求访求是 GET 或验证未通过就会执行下面的代码
    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 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。可以使用请求对象 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，如果能够使用 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')
    # 使用 cookies.get(key) 来代替 cookies[key] ，
    # 以避免当 cookie 不存在时引发 KeyError 。
```

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。

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 MIME 类型的响应对象。以下是转换的规则：

1. 如果视图要返回的是一个响应对象，那么就直接返回它。
2. 如果要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. 如果要返回的是一个元组，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, 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。
4. 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
```

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__)

@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 action="" method="post">
            <p><input type=text name=username>
            <p><input type=submit value=Login>
        </form>
    '''

@app.route('/logout')
def logout():
    # 如果会话中有用户名就删除它。
    session.pop('username', None)
    return redirect(url_for('index'))

 # 设置密钥，复杂一点:

app.secret_key = 'A0Zr98j/3yX R~XHH!jmN]LWX/,?RT'
```

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，困此一个好的密钥应当有足够的随机性。 你的操作系统可以使用一个随机生成器来生成一个好的随机种子：

```
>>> import os
>>> os.urandom(24)
'\xfd{H\xe5<\x95\xf9\xe3\x96.5\xd1\x01O<!\xd5\xa2\xa0\x9fR"\xa1\xa8'
```

只要复制这个随机种子到你的代码中就行了。 基于 cookie 的会话的说明： Flask 会把会话对象中的值储存在 cookie 中。在打开 cookie 的情况下，如果你访问会话对象中没有的值，那么会得到模糊的错误信息：请检查 页面 cookie 的大小是否与网络浏览器所支持的大小一致。

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

New in version 0.3.

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 Python Logger 类， 更多信息详见官方的 logging 文档 。

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)
```

部署到一个网络服务器

准备好发布你的新 Flask 应用了吗？作为本文的一个圆满结尾，你可以立即把应用部署到 一个主机上。下面介绍的是如何把小项目部署到免费主机上。

- 把 Flask 部署到 Heroku
- 把 WSGI 部署到 dotCloud 的 Flask 应用注意点 其他可以部署 Flask 应用的地方：
- 把 Flask 部署到 Webfaction
- 把 Flask 部署到 Google App Engine
- 用 Localtunnel 分离你的本地服务器 如果拥有自己的独立主机，参见《 部署方式 》。

5

教程

想要用 Python 和 Flask 开发应用吗？让我们来边看例子边学习。本教程中我们将会创建 一个微博应用。这个应用只支持单一用户，只能创建文本条目，也没有不能订阅和评论， 但是已经具备一个初学者需要掌握的功能。这个应用只需要使用 Flask 和 SQLite ， SQLite 是 Python 自带的。

如果你想要事先下载完整的源代码或者用于比较，请查看示例源代码 。

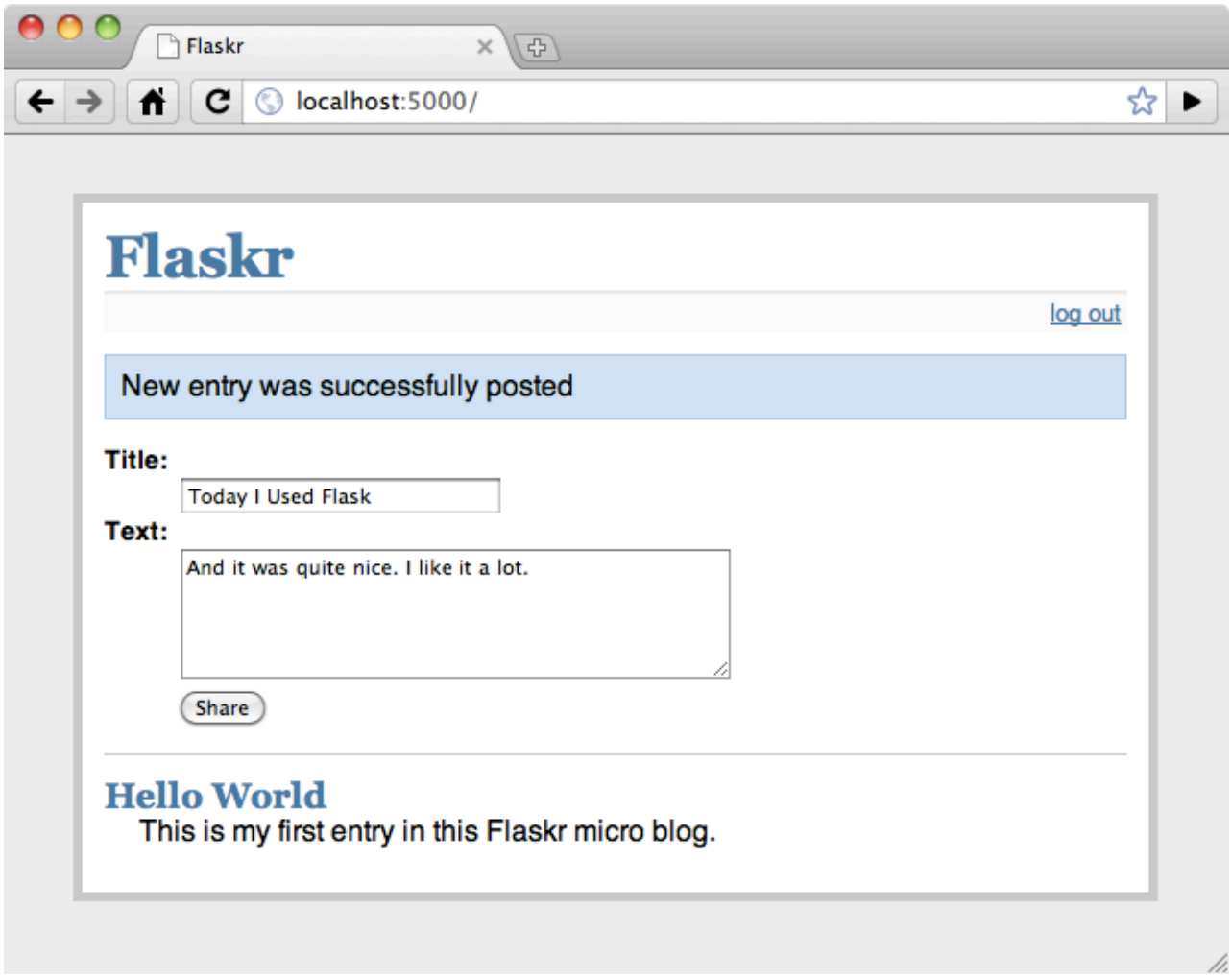
Flaskr 介绍

我们把教程中的博客应用称为 flaskr ， 当然你也可以随便取一个没有 Web-2.0 气息的名字 ;) 它的基本功能如下:

1. 让用户可以根据配置文件中的信息登录和注销。只支持一个用户。
2. 用户登录以后可以添加新的博客条目。条目由文本标题和支持 HTML 代码的内容组成。 因为我们信任用户，所以不对内容中的 HTML 进行净化处理。
3. 页面以倒序（新的在上面）显示所有条目。并且用户登录以后可以在这个页面添加新的条目。

我们直接在应用中使用 SQLite3 ， 因为在这种规模的应用中 SQlite3 已经够用了。如果 是大型应用，那么就有必要使用能够好的处理数据库连接的 SQLAlchemy 了，它可以 同时对应多种数据库，并做其他更多的事情。如果你的数据更适合使用 NoSQL 数据库， 那么也可以考虑使用某种流行的 NoSQL 数据库。

这是教程应用完工后的截图:



步骤 0： 创建文件夹

在开始之前需要为应用创建下列文件夹:

```
/flaskr
    /static
    /templates
```

flaskr 文件夹不是一个 Python 包，只是一个用来存放我们文件的地方。我们将把以后要用到的数据库模式和主模块放在这个文件夹中。 static 文件夹中的文件是用于供应用用户通过 HTTP 访问的文件，主要是 CSS 和 javascript 文件。 Flask 将会在 templates 文件夹中搜索 Jinja2 模板，所有在教程中的模板都放在 templates 文件夹中。

步骤 1： 数据库模式

首先我们要创建数据库模式。本应用只需要使用一张表，并且由于我们使用 SQLite ， 所以这一步非常简单。把以下内容保存为 schema.sql 文件并放在我们上一步创建的 flaskr 文件夹中就行了：

```
drop table if exists entries;
create table entries (
  id integer primary key autoincrement,
  title text not null,
  text text not null
);
```

这个模式只有一张名为 entries 的表，表中的字段为 id 、 title 和 text 。 id 是主键，是自增整数型字段，另外两个字段是非空的字符串型字段。

步骤 2： 应用构建代码

现在我们已经准备好了数据库模式了，下面来创建应用模块。我们把模块命名为 flaskr.py ，并放在 flaskr 文件夹中。为了方便初学者学习，我们把库的导入与相关配置放在了一起。对于小型应用来说，可以把配置直接放在模块中。但是更加清晰的 方案是把配置放在一个独立的 .ini 或 .py 文件中，并在模块中导入配置的值。

在 flaskr.py 文件中:

```
 # all the imports

import sqlite3
from flask import Flask, request, session, g, redirect, url_for, \
     abort, render_template, flash

 # configuration

DATABASE = '/tmp/flaskr.db'
DEBUG = True
SECRET_KEY = 'development key'
USERNAME = 'admin'
PASSWORD = 'default'
```

接着创建真正的应用，并用同一文件中的配置来初始化，在 flaskr.py 文件中:

```
 # create our little application :)

app = Flask(__name__)
app.config.from_object(__name__)
```

from_object() 会查看给定的对象（ 如果该对象是一个字符串就会直接导入它 ），搜索对象中所有变量名均为大字字母的变量。在我们的应用中，已经将配 置写在前面了。你可以把这些配置放到一个独立的文件中。

通常，从一个配置文件中导入配置是比较好的做法，我们使用 from_envvar() 来完成这个工作，把上面的 from_object() 一行替换为:

```
app.config.from_envvar('FLASKR_SETTINGS', silent=True)
```

这样做就可以设置一个 FLASKR_SETTINGS 的环境变量来指定一个配置文件，并根据该文件来重载缺省的配置。 silent 开关的作用是告诉 Flask 如果没有这个环境变量 不要报错。

secret_key（密钥）用于保持客户端会话安全，请谨慎地选择密钥，并尽可能的使 复杂而且不容易被猜到。 DEBUG 标志用于开关交互调试器。因为调试模式允许用户执行服务器上的代码，所以永远不要在生产环境中打开调试模式！

我们还添加了一个方便连接指定数据库的方法。这个方法可以用于在请求时打开连接，也可以用于 Python 交互终端或代码中。以后会派上用场。

```
def connect_db():
    return sqlite3.connect(app.config['DATABASE'])
```

最后，在文件末尾添加以单机方式启动服务器的代码:

```
if __name__ == '__main__':
    app.run()
```

到此为止，我们可以顺利运行应用了。输入以下命令开始运行:

```
python flaskr.py
```

你会看到服务器已经运行的信息，其中包含应用访问地址。

因为我们还没创建视图，所以当你在浏览器中访问这个地址时，会得到一个 404 页面未 找到错误。很快我们就会谈到视图，但我们先要弄好数据库。

外部可见的服务器

想让你的服务器被公开访问？详见外部可见的服务器 。

步骤 3： 创建数据库

如前所述 Flaskr 是一个数据库驱动的应用，更准确地说是一个关系型数据库驱动的 应用。关系型数据库需要一个数据库模式来定义如何储存信息，因此必须在第一次运行 服务器前创建数据库模式。

使用 sqlite3 命令通过管道导入 schema.sql 创建模式:

```
sqlite3 /tmp/flaskr.db < schema.sql
```

上述方法的不足之处是需要额外的 sqlite3 命令，但这个命令不是每个系统都有的。而且还必须提供数据库的路径，容易出错。因此更好的方法是在应用中添加一个数据库初始化函数。

添加的方法是：首先从 contextlib 库中导入 contextlib.closing() 函数，即在 flaskr.py 文件的导入部分添加如下内容:

```
from contextlib import closing
```

接下来，可以创建一个用来初始化数据库的 init_db 函数，其中我们使用了先前创建的 connect_db 函数。把这个初始化函数放在 flaskr.py 文件中的 connect_db 函数 下面:

```
def init_db():
    with closing(connect_db()) as db:
        with app.open_resource('schema.sql', mode='r') as f:
            db.cursor().executescript(f.read())
        db.commit()
```

closing() 帮助函数允许我们在 with 代码块保持数据库连接打开。应用对象的 open_resource() 方法支持也支持这个功能， 可以在 with 代码块中直接使用。这个函数打开一个位于来源位置（ 你的 flaskr 文件夹 ）的文件并允许你读取文件的内容。这里我们用于在数据库连接上执行代码。

当我们连接到数据库时，我们得到一个提供指针的连接对象（ 本例中的 db ）。这个 指针有一个方法可以执行完整的代码。最后我们提供要做的修改。 SQLite 3 和其他事务型数据库只有在显式提交时才会真正提交。

现在可以创建数据库了。打开 Python shell ， 导入，调用函数:

```
>>> from flaskr import init_db
>>> init_db()
```

故障处理

如果出现表无法找到的问题，请检查是否写错了函数名称（应该是 init_db ）， 是否写错了表名（例如单数复数错误）。

步骤 4： 请求数据库连接

现在我们已经学会如何打开并在代码中使用数据库连接，但是如何优雅地在请求时使用它呢？我们会在每一个函数中用到数据库连接，因此有必要在请求之前初始化连接，并在请求之后关闭连接。

Flask 中可以使用 before_request() 、 after_request() 和 teardown_request() 装饰器达到这个目的:

```
@app.before_request
def before_request():
    g.db = connect_db()

@app.teardown_request
def teardown_request(exception):
    db = getattr(g, 'db', None)
    if db is not None:
        db.close()
    g.db.close()
```

使用 before_request() 装饰的函数会在请求之前调用，且不传递 参数。使用 after_request() 装饰的函数会在请求之后调用，且 传递发送给客户端响应对象。它们必须传递响应对象，所以在出错的情况下就不会执行。 因此我们就要用到 teardown_request() 装饰器了。这个装饰器下 的函数在响应对象构建后被调用。它们不允许修改请求，并且它们的返回值被忽略。如果 请求过程中出错，那么这个错误会传递给每个函数；否则传递 None 。

我们把数据库连接保存在 Flask 提供的特殊的 g 对象中。这个对象与 每一个请求是一一对应的，并且只在函数内部有效。不要在其它对象中储存类似信息， 因为在多线程环境下无效。这个特殊的 g 对象会在后台神奇的工作，保证系统正常运行。

若想更好地处理这种资源，请参阅在 Flask 中使用 SQLite 3 。

Hint

我该把这些代码放在哪里?

如果你按教程一步一步读下来，那么可能会疑惑应该把这个步骤和以后的代码放在哪 里？比较有条理的做法是把这些模块级别的函数放在一起，并把新的 before_request 和 teardown_request 函数放在前文的 init_db 函数下面（即按照教程的顺序放置）。

如果你已经晕头转向了，那么你可以参考一下 示例源代码 。在 Flask 中，你可以把应用的所有代码都放在同一个 Python 模块中。但是你没有必要这样做，尤其是当你的应用变大了的时候，更不应当这样。

步骤 5： 视图函数

现在数据库连接弄好了，接着开始写视图函数。我们共需要四个视图函数：

显示条目

这个视图显示所有数据库中的条目。它绑定应用的根地址，并从数据库中读取 title 和 text 字段。 id 最大的记录（最新的条目）在最上面。从指针返回的记录集是一个包含 select 语句查询结果的元组。对于教程应用这样的小应用，做到这样就已经够好了。但是你可能想要把结果转换为字典，具体做法参见简化查询 中的例子。

这个视图会把条目作为字典传递给 `show_entries.html` 模板，并返回渲染结果:

```
@app.route('/')
def show_entries():
    cur = g.db.execute('select title, text from entries order by id desc')
    entries = [dict(title=row[0], text=row[1]) for row in cur.fetchall()]
    return render_template('show_entries.html', entries=entries)
```

添加一个新条目

这个视图可以让一个登录后的用户添加一个新条目。本视图只响应 POST 请求，真正的表单显示在 show_entries 页面中。如果一切顺利，我们会 flash() 一个消息给下一个请求并重定向回到 show_entries 页面:

```
@app.route('/add', methods=['POST'])
def add_entry():
    if not session.get('logged_in'):
        abort(401)
    g.db.execute('insert into entries (title, text) values (?, ?)',
                 [request.form['title'], request.form['text']])
    g.db.commit()
    flash('New entry was successfully posted')
    return redirect(url_for('show_entries'))
```

注意，我们在本视图中检查了用户是否已经登录（即检查会话中是否有 logged_in 键，且对应的值是否为 True）。

安全性建议

请像示例代码一样确保在构建 SQL 语句时使用问号。否则当你使用字符串构建 SQL 时容易遭到 SQL 注入攻击。更多内容参见 在 Flask 中使用 SQLite 3 。

登录和注销

这些函数用于用户登录和注销。登录视图根据配置中的用户名和密码验证用户并在会话中设置 logged_in 键值。如果用户通过验证，键值设为 True ，那么用户会被重定向到 show_entries 页面。另外闪现一个信息，告诉用户已登录成功。如果出现错误，模板会 提示错误信息，并让用户重新登录:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] != app.config['USERNAME']:
            error = 'Invalid username'
        elif request.form['password'] != app.config['PASSWORD']:
            error = 'Invalid password'
        else:
            session['logged_in'] = True
            flash('You were logged in')
            return redirect(url_for('show_entries'))
    return render_template('login.html', error=error)
```

登出视图则正好相反，把键值从会话中删除。在这里我们使用了一个小技巧：如果你使用字典的 pop() 方法并且传递了第二个参数（键的缺省值），那么当字典中有 这个键时就会删除这个键，否则什么也不做。这样做的好处是我们不用检查用户是否已经登录了。

```
@app.route('/logout')
def logout():
    session.pop('logged_in', None)
    flash('You were logged out')
    return redirect(url_for('show_entries'))
```

步骤 6： 模板

现在开始写模板。如果我们现在访问 URL ，那么会得到一个 Flask 无法找到模板文件的 异常。 Flask 使用 Jinja2 模板语法并默认开启自动转义。也就是说除非用 Markup 标记一个值或在模板中使用 |safe 过滤器，否则 Jinja2 会把如 < 或 > 之类的特殊字符转义为与其 XML 等价字符。

我们还使用了模板继承以保存所有页面的布局统一。

请把以下模板放在 templates 文件夹中：

layout.html

这个模板包含 HTML 骨架、头部和一个登录链接（如果用户已登录则变为一个注销链接 ）。如果有闪现信息，那么还会显示闪现信息。{% block body %} 块会被子模板中同名（ body ）的块替换。

session 字典在模板中也可以使用。你可以使用它来检验用户是否已经 登录。注意，在 Jinja 中可以访问对象或字典的不存在的属性和成员。如例子中的 'logged_in' 键不存在时代码仍然能正常运行：

```
<!doctype html>
<title>Flaskr</title>
<link rel=stylesheet type=text/css href="{{ url_for('static', filename='style.css') }}">
<div class=page>
  <h1>Flaskr</h1>
  <div class=metanav>
  {% if not session.logged_in %}
    <a href="{{ url_for('login') }}">log in</a>
  {% else %}
    <a href="{{ url_for('logout') }}">log out</a>
  {% endif %}
  </div>
  {% for message in get_flashed_messages() %}
    <div class=flash>{{ message }}</div>
  {% endfor %}
  {% block body %}{% endblock %}
</div>
```

show_entries.html

这个模板扩展了上述的 layout.html 模板，用于显示信息。注意， for 遍历了我们通过 render_template() 函数传递的所有信息。模板还告诉表单使用 POST 作为 HTTP 方法向 add_entry 函数提交数据：

```
{% extends "layout.html" %}
{% block body %}
  {% if session.logged_in %}
    <form action="{{ url_for('add_entry') }}" method=post class=add-entry>
      <dl>
        <dt>Title:
        <dd><input type=text size=30 name=title>
        <dt>Text:
        <dd><textarea name=text rows=5 cols=40></textarea>
        <dd><input type=submit value=Share>
      </dl>
    </form>
  {% endif %}
  <ul class=entries>
  {% for entry in entries %}
    <li><h2>{{ entry.title }}</h2>{{ entry.text|safe }}
  {% else %}
    <li><em>Unbelievable.  No entries here so far</em>
  {% endfor %}
  </ul>
{% endblock %}
```

login.html

最后是简单显示用户登录表单的登录模板：

```
{% extends "layout.html" %}
{% block body %}
  <h2>Login</h2>
  {% if error %}<p class=error><strong>Error:</strong> {{ error }}{% endif %}
  <form action="{{ url_for('login') }}" method=post>
    <dl>
      <dt>Username:
      <dd><input type=text name=username>
      <dt>Password:
      <dd><input type=password name=password>
      <dd><input type=submit value=Login>
    </dl>
  </form>
{% endblock %}
```

步骤 7： 添加样式

现在万事俱备，只剩给应用添加一些样式了。只要把以下内容保存为 static 文件夹中的 style.css 文件就行了：

```
body           { font-family: sans-serif; background: #eee; }
a, h1, h2      { color: #377ba8; }
h1, h2         { font-family: 'Georgia', serif; margin: 0; }
h1             { border-bottom: 2px solid #eee; }
h2             { font-size: 1.2em; }

.page          { margin: 2em auto; width: 35em; border: 5px solid #ccc;
                 padding: 0.8em; background: white; }
.entries       { list-style: none; margin: 0; padding: 0; }
.entries li    { margin: 0.8em 1.2em; }
.entries li h2  { margin-left: -1em; }
.add-entry     { font-size: 0.9em; border-bottom: 1px solid #ccc; }
.add-entry dl   { font-weight: bold; }
.metanav       { text-align: right; font-size: 0.8em; padding: 0.3em;
                 margin-bottom: 1em; background: #fafafa; }
.flash         { background: #cee5F5; padding: 0.5em;
                 border: 1px solid #aacbe2; }
.error         { background: #f0d6d6; padding: 0.5em; }
```

额外赠送：测试应用

现在你已经完成了整个应用，一切都运行正常。为了方便以后进行完善修改，添加自动测试不失为一个好主意。本教程中的应用已成为测试 Flask 应用文档中演示如何进行 单元测试的例子，可以去看看测试 Flask 应用是多么容易

6

模板

Flask 使用 Jinja2 作为默认模板引擎。你完全可以使用其它模板引擎。但是不管你使用 哪种模板引擎，都必须安装 Jinja2 。因为使用 Jinja2 可以让 Flask 使用更多依赖于 这个模板引擎的扩展。

本文只是简单介绍如何在 Flask 中使用 Jinja2 。如果要详细了解这个模板引擎的语法， 请查阅 Jinja2 模板官方文档 。

Jinja 设置

在 Flask 中， Jinja2 默认配置如下：

- 在扩展名为 .html 、 .htm 、 .xml 和 .xhtml 的模板中开启自动转义。
- 在模板中可以使用 {% autoescape %} 来手动设置是否转义。
- Flask 在 Jinja2 环境中加入一些全局函数和辅助对象，以增强模板的功能。

标准环境

缺省情况下，以下全局变量可以在 Jinja2 模板中使用：

config 当前配置对象（ flask.config ）

New in version 0.6.

Changed in version 0.10: 此版本开始，这个变量总是可用，甚至是在被导入的模板中。

request 当前请求对象（ flask.request ）。在没有活动请求环境情况下渲染模块 时，这个变量不可用。

session 当前会话对象（ flask.session ）。在没有活动请求环境情况下渲染模块 时，这个变量不可用。

g 请求绑定的全局变量（ flask.g ）。在没有活动请求环境情况下渲染模块 时，这个变量不可用。

url_for() flask.url_for() 函数。

get_flashed_messages() flask.get_flashed_messages() 函数。

Jinja 环境行为

这些添加到环境中的变量不是全局变量。与真正的全局变量不同的是这些变量在已导入 的模板的环境中是不可见的。这样做是基于性能的原因，同时也考虑让代码更有条理。

那么对你来说又有什么意义呢？假设你需要导入一个宏，这个宏需要访问请求对象， 那么你有两个选择：

1. 显式地把请求或都该请求有用的属性作为参数传递给宏。
2. 导入 “with context” 宏。
3. 导入方式如下：

```
{% from '_helpers.html' import my_macro with context %}
```

标准过滤器

在 Flask 中的模板中添加了以下 Jinja2 本身没有的过滤器：

tojson()

这个函数可以把对象转换为 JSON 格式。如果你要动态生成 JavaScript ，那么这个函数非常有用。

注意，在 script 标记内部不能转义，因此在 Flask 0.10 之前的版本中，如果要在 script 标记内部使用这个函数必须用 |safe 关闭转义：

```
<script type=text/javascript>
   doSomethingWith({{ user.username|tojson|safe }});
</script>
```

控制自动转义

自动转义是指自动对特殊字符进行转义。特殊字符是指 HTML （ 或 XML 和 XHTML ）中的 & 、 > 、 < 、 " 和 ' 。因为这些特殊字符代表了特殊的意思， 所以如果要在文本中使用它们就必须把它们替换为“实体”。如果不转义，那么用户就 无法使用这些字符，而且还会带来安全问题。（ 参见跨站脚本攻击（ XSS ） ）

有时候，如需要直接把 HTML 植入页面的时候，可能会需要在模板中关闭自动转义功能。这个可以直接植入的 HTML 一般来自安全的来源，例如一个把标记语言转换为 HTML 的转换器。

有三种方法可以控制自动转义：

1. 在 Python 代码中，可以在把 HTML 字符串传递给模板之前，用 Markup 对象封装。一般情况下推荐使用这个方法。
2. 在模板中，使用 |safe 过滤器显式把一个字符串标记为安全的 HTML （ 例如： {{ myvariable|safe }} ）。
3. 临时关闭整个系统的自动转义。

在模板中关闭自动转义系统可以使用 {% autoescape %} 块：

```
{% autoescape false %}
    <p>autoescaping is disabled here
    <p>{{ will_not_be_escaped }}
{% endautoescape %}
```

在这样做的时候，要非常小心块中的变量的安全性。

注册过滤器

有两种方法可以在 Jinja2 中注册你自己的过滤器。要么手动把它们放入应用的 jinja_env 中，要么使用 template_filter() 装饰器。

下面两个例子功能相同，都是倒序一个对象:

```
@app.template_filter('reverse')
def reverse_filter(s):
    return s[::-1]

def reverse_filter(s):
    return s[::-1]
app.jinja_env.filters['reverse'] = reverse_filter
```

装饰器的参数是可选的，如果不给出就使用函数名作为过滤器名。一旦注册完成后，你就可以在模板中像 Jinja2 的内建过滤器一样使用过滤器了。例如，假设在环境中你有一个名为 mylist 的 Pyhton 列表:

```
{% for x in mylist | reverse %}
{% endfor %}
```

环境处理器

环境处理器的作用是把新的变量自动引入模板环境中。环境处理器在模板被渲染前运行， 因此可以把新的变量自动引入模板环境中。它是一个函数，返回值是一个字典。在应用的所有模板中，这个字典将与模板环境合并:

```
@app.context_processor
def inject_user():
    return dict(user=g.user)
```

上例中的环境处理器创建了一个值为 g.user 的 user 变量，并把这个变量加入了 模板环境中。这个例子只是用于说明工作原理，不是非常有用，因为在模板中， g 总是存在的。

传递值不仅仅局限于变量，还可以传递函数（ Python 提供传递函数的功能）:

```
@app.context_processor
def utility_processor():
    def format_price(amount, currency=u'€'):
        return u'{0:.2f}{1}'.format(amount, currency)
    return dict(format_price=format_price)
```

上例中的环境处理器把 format_price 函数传递给了所有模板:

```
{{ format_price(0.33) }}
```

你还可以把 format_price 创建为一个模板过滤器（参见注册过滤器 ），这里只是演示如何在一个环境处理器中传递函数。

7

测试 Flask 应用

> 未经测试的小猫，肯定不是一只好猫。

这句话的出处不详（译者注：这句是译者献给小猫的），也不一定完全正确，但是基本上是正确的。未经测试的应用难于改进现有的代码，因此其开发者会越改进越抓狂。反之， 经过自动测试的代码可以安全的改进，并且如果可以测试过程中立即发现错误。

Flask 提供的测试渠道是公开 Werkzeug 的 Client ，为你 处理本地环境。你可以结合这个渠道使用你喜欢的测试工具。本文使用的测试工具是随着 Python 一起安装好的 unittest 包。

应用

首先，我们需要一个用来测试的应用。我们将使用教程中的应用。如果你还没有这个应用，可以下载示例代码 。

测试骨架

为了测试应用，我们添加了一个新的模块 (flaskr_tests.py) 并创建了如下测试骨架:

```
import os
import flaskr
import unittest
import tempfile

class FlaskrTestCase(unittest.TestCase):

    def setUp(self):
        self.db_fd, flaskr.app.config['DATABASE'] = tempfile.mkstemp()
        flaskr.app.config['TESTING'] = True
        self.app = flaskr.app.test_client()
        flaskr.init_db()

    def tearDown(self):
        os.close(self.db_fd)
        os.unlink(flaskr.app.config['DATABASE'])

if __name__ == '__main__':
    unittest.main()
```

setUp() 方法中会创建一个新的测试客户端并初始化一个新的数据库。在每个独立的测试函数运行前都会调用这个方法。 tearDown() 方法的功能是在测试结束后关闭文件，并在文件系统中删除数据库文件。另外在设置中 TESTING 标志开启的，这意味着在请求时关闭 错误捕捉，以便于在执行测试请求时得到更好的错误报告。

测试客户端会给我们提供一个简单的应用接口。我们可以通过这个接口向应用发送测试请求。客户端还可以追踪 cookies 。

因为 SQLite3 是基于文件系统的，所以我们可以方便地使用临时文件模块来创建一个临时数据库并初始化它。 mkstemp() 函数返回两个东西：一个低级别的文件 句柄和一个随机文件名。这个文件名后面将作为我们的数据库名称。我们必须把句柄保存到 db_fd 中，以便于以后用 os.close() 函数来关闭文件。

如果现在进行测试，那么会输出以下内容:

```
$ python flaskr_tests.py
##  Ran 0 tests in 0.000s

OK
```

虽然没有运行任何实际测试，但是已经可以知道我们的 flaskr 应用没有语法错误。 否则在导入时会引发异常并中断运行。

第一个测试

现在开始测试应用的功能。当我们访问应用的根 URL （ / ）时应该显示 “ No entries here so far ”。我们新增了一个新的测试方法来测试这个功能:

```
class FlaskrTestCase(unittest.TestCase):

    def setUp(self):
        self.db_fd, flaskr.app.config['DATABASE'] = tempfile.mkstemp()
        self.app = flaskr.app.test_client()
        flaskr.init_db()

    def tearDown(self):
        os.close(self.db_fd)
        os.unlink(flaskr.app.config['DATABASE'])

    def test_empty_db(self):
        rv = self.app.get('/')
        assert 'No entries here so far' in rv.data
```

注意，我们的调试函数都是以 test 开头的。这样 unittest 就会自动识别这些是用于测试的函数并运行它们。

通过使用 self.app.get ，可以向应用的指定 URL 发送 HTTP GET 请求，其返回的是一个 ~flask.Flask.response_class 对象。我们可以使用 data 属性来检查应用的返回值（字符串类型）。在本例中，我们检查输出是否包含 'No entries here so far' 。

再次运行测试，会看到通过了一个测试:

```
$ python flaskr_tests.py
## .Ran 1 test in 0.034s

OK
```

登录和注销

我们应用的主要功能必须登录以后才能使用，因此必须测试应用的登录和注销。测试的 方法是使用规定的数据（用户名和密码）向应用发出登录和注销的请求。因为登录和注销后会重定向到别的页面，因此必须告诉客户端使用 follow_redirects 追踪重定向。

在 FlaskrTestCase 类中添加以下两个方法:

```
def login(self, username, password):
    return self.app.post('/login', data=dict(
        username=username,
        password=password
    ), follow_redirects=True)

def logout(self):
    return self.app.get('/logout', follow_redirects=True)
```

现在可以方便地测试登录成功、登录失败和注销功能了。下面为新增的测试代码:

```
def test_login_logout(self):
    rv = self.login('admin', 'default')
    assert 'You were logged in' in rv.data
    rv = self.logout()
    assert 'You were logged out' in rv.data
    rv = self.login('adminx', 'default')
    assert 'Invalid username' in rv.data
    rv = self.login('admin', 'defaultx')
    assert 'Invalid password' in rv.data
```

测试增加条目功能

我们还要测试增加条目功能。添加以下测试代码:

```
def test_messages(self):
    self.login('admin', 'default')
    rv = self.app.post('/add', data=dict(
        title='<Hello>',
        text='<strong>HTML</strong> allowed here'
    ), follow_redirects=True)
    assert 'No entries here so far' not in rv.data
    assert '&lt;Hello&gt;' in rv.data
    assert '<strong>HTML</strong> allowed here' in rv.data
```

这里我们检查了博客内容中允许使用 HTML ，但标题不可以。应用设计思路就是这样的。

运行测试，现在通过了三个测试:

```
$ python flaskr_tests.py
## ...Ran 3 tests in 0.332s

OK
```

关于更复杂的 HTTP 头部和状态码测试参见 MiniTwit 示例 。这个示例的源代码中 包含了更大的测试套件。

其他测试技巧

除了使用上述测试客户端外，还可以在 with 语句中使用 test_request_context() 方法来临时激活一个请求环境。在这个 环境中可以像以视图函数中一样操作 request 、g 和 session 对象。示例:

```
app = flask.Flask(__name__)

with app.test_request_context('/?name=Peter'):
    assert flask.request.path == '/'
    assert flask.request.args['name'] == 'Peter'
```

其他与环境绑定的对象也可以这样使用。

如果你必须使用不同的配置来测试应用，而且没有什么好的测试方法，那么可以考虑使用应用工厂（参见应用工厂）。

注意，在测试请求环境中 before_request() 函数和 after_request() 函数不会被自动调用。但是当调试请求环境离开 with 块时会执行 teardown_request() 函数。如果需要 before_request() 函数和正常情况下一样被调用，那么你必须调用 preprocess_request()

```
app = flask.Flask(__name__)

with app.test_request_context('/?name=Peter'):
    app.preprocess_request()
    ...
```

在这函数中可以打开数据库连接或者根据应用需要打开其他类似东西。

如果想调用 after_request() 函数，那么必须调用 process_response()，并把响应对象传递给它:

```
app = flask.Flask(__name__)

with app.test_request_context('/?name=Peter'):
    resp = Response('...')
    resp = app.process_response(resp)
    ...
```

这个例子中的情况基本没有用处，因为在这种情况下可以直接开始使用测试客户端。

伪造资源和环境

New in version 0.10.

通常情况下，我们会把用户认证信息和数据库连接储存到应用环境或者 flask.g 对象中，并在第一次使用前准备好，然后在断开时删除。假设应用中得到当前用户的代码如下:

```
def get_user():
    user = getattr(g, 'user', None)
    if user is None:
        user = fetch_current_user_from_database()
        g.user = user
    return user
```

在测试时可以很很方便地重载用户而不用改动代码。可以先象下面这样钩接 flask.appcontext_pushed 信号:

```
from contextlib import contextmanager
from flask import appcontext_pushed

@contextmanager
def user_set(app, user):
    def handler(sender, **kwargs):
        g.user = user
    with appcontext_pushed.connected_to(handler, app):
        yield
```

然后使用:

```
from flask import json, jsonify

@app.route('/users/me')
def users_me():
    return jsonify(username=g.user.username)

with user_set(app, my_user):
    with app.test_client() as c:
        resp = c.get('/users/me')
        data = json.loads(resp.data)
        self.assert_equal(data['username'], my_user.username)
```

保持环境

New in version 0.4.

有时候这种情形是有用的：触发一个常规请求，但是保持环境以便于做一点额外的事情。 在 Flask 0.4 之后可以在 with 语句中使用 test_client() 来 实现:

```
app = flask.Flask(__name__)

with app.test_client() as c:
    rv = c.get('/?tequila=42')
    assert request.args['tequila'] == '42'
```

如果你在没有 with 的情况下使用 test_client() ，那么 assert 会出错失败。因为无法在请求之外访问 request 。

访问和修改会话

New in version 0.8.

有时候在测试客户端中访问和修改会话是非常有用的。通常有两方法。如果你想测试会话中 的键和值是否正确，你可以使用 flask.session:

```
with app.test_client() as c:
    rv = c.get('/')
    assert flask.session['foo'] == 42
```

但是这个方法无法修改会话或在请求发出前访问会话。自 Flask 0.8 开始，我们提供了 “会话处理”，用打开测试环境中会话和修改会话，最后保存会话。处理后的会话独立于 后端实际使用的会话:

```
with app.test_client() as c:
    with c.session_transaction() as sess:
        sess['a_key'] = 'a value'

    # 运行到这里时，会话已被保存
```

注意在这种情况下必须使用 sess 对象来代替 flask.session 代理。 sess 对象本身可以提供相同的接口。

8

掌握应用错误

New in version 0.3.

应用出错，服务器出错。或早或晚，你会遇到产品出错。即使你的代码是百分百正确， 还是会时常看见出错。为什么？因为其他相关东西会出错。以下是一些在代码完全正确的条件下服务器出错的情况：

1. 客户端已经中断了请求，但应用还在读取数据。
2. 数据库已经过载，无法处理查询。
3. 文件系统没有空间。
4. 硬盘完蛋了。
5. 后台服务过载。
6. 使用的库出现程序错误。
7. 服务器与另一个系统的网络连接出错。

以上只是你会遇到的问题的一小部分。那么如果处理这些问题呢？如果你的应用运行在生产环境下，那么缺省情况下 Flask 会显示一个简单的出错页面，并把出错情况记录到 logger 。

但要做得还不只这些，下面介绍一些更好的出错处理方法。

报错邮件

如果应用在生产环境（在你的服务器中一般使用生产环境）下运行，那么缺省情况下不会 看到任何日志信息。为什么？因为 Flask 是一个零配置的框架。既然没有配置，那么日志 放在哪里呢？显然， Flask 不能来随便找一个地放给用户存放日志，因为如果用户在这个 位置没有创建文件的权限就糟了。同时，对于大多数小应用来说，没人会去看日志。

事实上，我现在可以负责任地说除非调试一个用户向你报告的错误，你是不会去看应用的 日志文件的。你真下需要的是出错的时候马上给你发封电子邮件，及时提醒你，以便于进行处理。

Flask 使用 Python 内置的日志系统，它可以发送你需要的错误报告电子邮件。以下是 如何配置 Flask 日志记录器发送错误报告电子邮件的例子:

```
ADMINS = ['yourname@example.com']
if not app.debug:
    import logging
    from logging.handlers import SMTPHandler
    mail_handler = SMTPHandler('127.0.0.1',
                               'server-error@example.com',
                               ADMINS, 'YourApplication Failed')
    mail_handler.setLevel(logging.ERROR)
    app.logger.addHandler(mail_handler)
```

这个例子是什么意思？我们创建了一个新的 SMTPHandler 类。这个类会使用邮件服务器 127.0.0.1 向 server-error@example.com 的 ADMINS 发送主题为 “YourApplication Failed” 的 电子邮件。如果你的邮件服务器需要认证，这是可行的，详见 SMTPHandler 的文档。

我们还定义了只报送错误及错误以上级别的信息。因为我们不想得到警告或其他没用的 日志，比如请求处理日志。

在你的产品中使用它们前，请查看一下控制日志格式 ，以了解错误报告邮件的更多 信息，磨刀不误砍柴功。

日志文件

报错邮件有了，可能还需要记录警告信息。这是一个好习惯，有利于除错。请注意，在 核心系统中 Flask 本身不会发出任何警告。因此，在有问题时发出警告只能自力更生了。

虽然有许多日志记录系统，但不是每个系统都能做好基本日志记录的。以下可能是最值得关注的：

- FileHandler – 把日志信息记录到文件系统中的一个文件。
- RotatingFileHandler – 把日志信息记录到文件系统中的一个文件，当信息达到一定数量后反转。
- NTEventLogHandler – 把日志信息记录到 Windows 的事件日志中。 如果你的应用部署在 Windows 下，就用这个吧。
- SysLogHandler – 把日志记录到一个 UNIX 系统日志。 一旦你选定了日志记录器之后，使用方法类似上一节所讲的 SMTP 处理器，只是记录的 级别应当低一点（我推荐 WARNING 级别）：

```
if not app.debug:
    import logging
    from themodule import TheHandlerYouWant
    file_handler = TheHandlerYouWant(...)
    file_handler.setLevel(logging.WARNING)
    app.logger.addHandler(file_handler)
```

控制日志格式

缺省情况下一个处理器只会把信息字符串写入一个文件或把信息作为电子邮件发送给你。 但是一个日志应当记录更多的信息，因些应该认真地配置日志记录器。一个好的日志不光 记录为什么会出错，更重要的是记录错在哪里。

格式化器使用一个格式化字符串作为实例化时的构造参数，这个字符串中的格式变量会在日志记录时自动转化。

举例:

电子邮件

```
from logging import Formatter
mail_handler.setFormatter(Formatter('''
Message type:       %(levelname)s
Location:           %(pathname)s:%(lineno)d
Module:             %(module)s
Function:           %(funcName)s
Time:               %(asctime)s

Message:

%(message)s
'''))
```

日志文件

```
from logging import Formatter
file_handler.setFormatter(Formatter(
    '%(asctime)s %(levelname)s: %(message)s '
    '[in %(pathname)s:%(lineno)d]'
))
```

复杂日志格式

以下是格式化字符串中一种重要的格式变量。注意，这并不包括全部格式变量，更多变更参见 logging 包的官方文档。

格式变量	说明
%(levelname)s	文字形式的日志等级（ 'DEBUG' 、 'INFO' 、 'WARNING' 、 'ERROR' 和 'CRITICAL' ）。
%(pathname)s	调用日志的源文件的完整路径（如果可用）。
%(filename)s	调用日志的源文件文件名。
%(module)s	调用日志的模块名。
%(funcName)s	调用日志的函数名。
%(lineno)d	调用日志的代码的行号（如果可用）。
%(asctime)s	调用日志的时间，缺省格式为 "2003-07-08 16:49:45,896"（逗号后面的数字为 毫秒 ）。通过重载 formatTime() 方法可以改变格式。
%(message)s	日志记录的消息，同 msg % args 。

如果要进一步定制格式，可以使用格式化器的子类。格式化器有三个有趣的方法：

format(): 处理实际的格式化。它接收一个 LogRecord 对象，返回格式化后 的字符串。 formatTime(): 它用于 asctime 格式变量。重载它可以改变时间格式。 formatException() 它用于异常格式化。接收一个 exc_info 元组并且必须返回一个字符串。缺省情况下它够用了，不必重载。 更多信息参见官方文档。

其他库

至此，我们只配置了应用本身的日志记录器。其他库可能同样需要记录日志。例如， SQLAlchemy 在其核心中大量使用日志。在 logging 包中有一个方法可以一次性 地配置所有日志记录器，但我不推荐这么做。因为当你在同一个 Python 解释器中同时运行两个独立的应用时就无法使用不同的日志设置了。

相反，我建议使用 getLogger() 函数来鉴别是哪个日志记录器，并获取相应的处理器:

```
from logging import getLogger
loggers = [app.logger, getLogger('sqlalchemy'),
        getLogger('otherlibrary')]
for logger in loggers:
    logger.addHandler(mail_handler)
    logger.addHandler(file_handler)
```

9

排除应用错误

掌握应用错误 一文所讲的是如何为生产应用设置日志和出错通知。本文要 讲的是部署中配置调试的要点和如何使用全功能的 Python 调试器深挖错误。

有疑问时，请手动运行

在生产环境中，配置应用时出错？如果你可以通过 shell 来访问主机，那么请首先在部署环境中验证是否可以通过 shell 手动运行你的应用。请确保验证时使用的帐户与配置的相同，这样可以排除用户权限引发的错误。你可以在你的生产服务器上，使用 Flask 内建的开发服务器，并且设置 `debug=True` ，这样有助于找到配置问题。但是，请**只能在可控的情况下临时这样做**，绝不能在生产时使用 `debug=True` 。

使用调试器

为了更深入的挖掘错误，追踪代码的执行， Flask 提供一个开箱即用的调试器（参见调试模式 ）。如果你需要使用其他 Python 调试器，请注意调试器之间的干扰问题。在使用你自己的调试器前要做一些参数调整：

- debug – 是否开启调试模式并捕捉异常
- use_debugger – 是否使用 Flask 内建的调试器
- use_reloader – 出现异常后是否重载或者派生进程
- debug 必须设置为 True （即必须捕获异常），另两个随便。

如果你正在使用 Aptana 或 Eclipse 排错，那么 use_debugger 和 use_reloader 都必须设置为 False 。

一个有用的配置模式如下（当然要根据你的应用调整缩进）：

```
FLASK:
    DEBUG: True
    DEBUG_WITH_APTANA: True
```

然后，在应用入口（ main.py ），修改如下：

```
if __name__ == "__main__":
    # 为了让 aptana 可以接收到错误，设置 use_debugger=False
    app = create_app(config="config.yaml")

    if app.debug: use_debugger = True
    try:
        # 如果使用其他调试器，应当关闭 Flask 的调试器。
        use_debugger = not(app.config.get('DEBUG_WITH_APTANA'))
    except:
        pass
    app.run(use_debugger=use_debugger, debug=app.debug,
            use_reloader=use_debugger, host='0.0.0.0')
```

10

配置管理

New in version 0.3.

应用总是需要一定的配置的。根据应用环境不同，会需要不同的配置。比如开关调试 模式、设置密钥以及其他依赖于环境的东西。

Flask 的设计思路是在应用开始时载入配置。你可以在代码中直接硬编码写入配置，对于 许多小应用来说这不一定是一件坏事，但是还有更好的方法。

不管你使用何种方式载入配置，都可以使用 Flask 的 config 属性来操作配置的值。 Flask 本身就使用这个对象来保存 一些配置，扩展也可以使用这个对象保存配置。同时这也是你保存配置的地方。

配置入门

config 实质上是一个字典的子类，可以像字典一样操作:

```
app = Flask(__name__)
app.config['DEBUG'] = True
```

某些配置值还转移到了 Flask 对象中，可以直接通过 Flask 来操作:

```
app.debug = True
```

一次更新多个配置值可以使用 dict.update() 方法:

```
app.config.update(
    DEBUG=True,
    SECRET_KEY='...'
)
```

内置配置变量

以下配置变量由 Flask 内部使用：

DEBUG	开关调试模式
TESTING	开关测试模式
PROPAGATE_EXCEPTIONS	显式开关异常的传播。当 TESTING 或 DEBUG 为真时，总是开启的。
PRESERVE_CONTEXT_ON_EXCEPTION	缺省情况下，如果应用在调试模式下运行， 那么请求环境在发生异常时不会被弹出，以 方便调试器内省数据。可以通过这个配置来 禁止这样做。还可以使用这个配置强制不执行 调试，这样可能有助于调试生产应用（ 风险 大 ）。
SECRET_KEY	密钥
SESSION_COOKIE_NAME	会话 cookie 的名称
SESSION_COOKIE_DOMAIN	会话 cookie 的域。如果没有配置，那么 SERVER_NAME 的所有子域都可以使用 这个 cookie 。
SESSION_COOKIE_PATH	会话 cookie 的路径。如果没有配置，那么 所有 APPLICATION_ROOT 都可以使用 cookie 。如果没有设置 APPLICATION_ROOT ，那么 '/' 可以 使用 cookie 。
SESSION_COOKIE_HTTPONLY	设置 cookie 的 httponly 标志，缺省为 True 。
SESSION_COOKIE_SECURE	设置 cookie 的安全标志，缺省为 False 。
PERMANENT_SESSION_LIFETIME	常驻会话的存活期，其值是一个 datetime.timedelta 对象。 自 Flask 0.8 开始，其值可以是一个整数， 表示秒数。
USE_X_SENDFILE	开关 x-sendfile
LOGGER_NAME	日志记录器的名称
SERVER_NAME	服务器的名称和端口号，用于支持子域（ 如： 'myapp.dev:5000' ）。注意设置为 “ localhost ”没有用，因为 localhost 不 支持子域。设置了 SERVER_NAME 后，在 缺省情况下会启用使用应用环境而不使用请求 环境的 URL 生成。
APPLICATION_ROOT	如果应用不占用整个域或子域，那么可以用 这个配置来设定应用的路径。这个配置还用作 会话 cookie 的路径。如果使用了整个域， 那么这个配置的值应当为 None 。
MAX_CONTENT_LENGTH	这个配置的值单位为字节，如果设置了，那么 Flask 会拒绝超过设定长度的请求，返回一个 413 状态码。

DEBUG	开关调试模式
SEND_FILE_MAX_AGE_DEFAULT	send_static_file()（ 缺省静态文件处理器 ）和 send_file() 使用的缺省缓存 最大存活期控制，以秒为单位。把 get_send_file_max_age() 分别挂勾到 Flask 或 Blueprint 上，可以重载每个 文件的值。缺省值为 43200（ 12 小时 ）。
TRAP_HTTP_EXCEPTIONS	如果设置为 True ，那么 Flask 将不 执行 HTTP 异常的错误处理，而是把它像其它 异常同样对待并把它压入异常堆栈。当你在 必须查找出一个 HTTP 异常来自哪里的情况下 这个 配置比较有用。
TRAP_BAD_REQUEST_ERRORS	Werkzeug 用于处理请求特殊数据的内部数据 结构会引发坏请求异常。同样，许多操作为了一致性会使用一个坏请求隐藏操作失败。在 这种情况下，这个配置可以在调试时辨别到底 为什么会失败。如果这个配置设为 True ，那么就只能得到一个普通的反馈。
PREFERRED_URL_SCHEME	在没有可用的模式的情况下， URL 生成所 使用的 URL 模式。缺省值为 http 。
JSON_AS_ASCII	缺省情况下 Flask 把对象序列化为 ascii-encoded JSON 。如果这个参数值为 False ，那么 Flask 就不会把对象编码 为 ASCII ，只会原样输出返回 unicode 字符 串。 jsonfiy 会自动把对象编码 utf-8 字符用于传输。
JSON_SORT_KEYS	缺省情况下 Flask 会按键值排序 JSON 对象， 这是为了确保字典的哈希种子的唯一性，返回值会保持一致，不会破坏外部 HTTP 缓存。 改变这个参数的值就可以重载缺省的行为， 重载后可能会提高缓存的性能，但是不推荐 这样做。
JSONIFY_PRETTYPRINT_REGULAR	如果这个参数设置为 True（ 缺省值 ）， 并且如果 jsonify 响应不是被一个 XMLHttpRequest 对象请求的（ 由 X-Requested-With 头部控制 ），那么 就会被完美打印。

关于 SERVER_NAME 的更多说明

SERVER_NAME 配置用于支持子域。如果要使用子域，那么就需要这个配置。因为 Flask 在不知道真正服务器名称的情况下无法得知子域。这个配置也用于会话 cookie 。

请记住，不仅 Flask 是在使用子域时有这样的问题，你的浏览器同样如此。大多数 现代浏览器不会允许在没有点的服务器名称上设置跨子域 cookie 。因此，如果你的 服务器名称是 'localhost' ，那么你将不能为 'localhost' 和所有子域设置 cookie 。在这种情况下请选择一个其他服务器名称，如 'myapplication.local' 。并且把名称加上要使用的子域写入主机配置中或者设置 一个本地 bind 。 New in version 0.4: LOGGER_NAME

New in version 0.5: SERVER_NAME

New in version 0.6: MAX_CONTENT_LENGTH

New in version 0.7: PROPAGATE_EXCEPTIONS, PRESERVE_CONTEXT_ON_EXCEPTION

New in version 0.8: TRAP_BAD_REQUEST_ERRORS, TRAP_HTTP_EXCEPTIONS, APPLICATION_ROOT, SESSION_COOKIE_DOMAIN, SESSION_COOKIE_PATH, SESSION_COOKIE_HTTPONLY, SESSION_COOKIE_SECURE

New in version 0.9: PREFERRED_URL_SCHEME

New in version 0.10: JSON_AS_ASCII, JSON_SORT_KEYS, JSONIFY_PRETTYPRINT_REGULAR

使用配置文件

如果把配置放在一个单独的文件中会更有用。理想情况下配置文件应当放在应用包的 外面。这样可以在修改配置文件时不影响应用的打包与分发（ 使用 Distribute 部署 ）。

因此，常见用法如下:

```
app = Flask(__name__)
app.config.from_object('yourapplication.default_settings')
app.config.from_envvar('YOURAPPLICATION_SETTINGS')
```

首先从 `yourapplication.default_settings` 模块载入配置，然后根据 `YOURAPPLICATION_SETTINGS` 环境变量所指向的文件的内容重载配置的值。在 启动服务器前，在 Linux 或 OS X 操作系统中，这个环境变量可以在终端中使用 `export` 命令来设置:

```
$ export YOURAPPLICATION_SETTINGS=/path/to/settings.cfg
$ python run-app.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader...
```

在 Windows 系统中使用内置的 set 来代替:

```
>set YOURAPPLICATION_SETTINGS=\path\to\settings.cfg
```

配置文件本身实质是 Python 文件。只有全部是大写字母的变量才会被配置对象所使用。 因此请确保使用大写字母。

一个配置文件的例子:

```
 # 配置示例

DEBUG = False
SECRET_KEY = '?\xbf,\xb4\x8d\xa3"<\x9c\xb0@\x0f5\xab,w\xee\x8d$0\x13\x8b83'
```

请确保尽早载入配置，以便于扩展在启动时可以访问相关配置。除了从文件载入配置外， 配置对象还有其他方法可以载入配置，详见 Config 对象的文档。

配置的最佳实践

前述的方法的缺点是测试有一点点麻烦。通常解决这个问题没有标准答案，但有些好的好的建议：

1. 在一个函数中创建你的应用并注册“蓝图”。这样就可以使用不同配置创建多个实例，极大方便单元测试。你可以按需载入配置。
2. 不要编写在导入时就访问配置的代码。如果你限制自己只能通过请求访问代码，那么 你可以以后按需配置对象。

开发/生产

大多数应用需要一个以上的配置。最起码需要一个配置用于生产服务器，另一个配置用于 开发。应对这种情况的最简单的方法总是载入一个缺省配置，并把这个缺省配置作为版本 控制的一部分。然后，把需要重载的配置，如前文所述，放在一个独立的文件中:

```
app = Flask(__name__)
app.config.from_object('yourapplication.default_settings')
app.config.from_envvar('YOURAPPLICATION_SETTINGS')
```

然后你只要增加一个独立的 config.py 文件并导出 `YOURAPPLICATION_SETTINGS=/path/to/config.py` 就可了。当然还有其他方法可选， 例如可以使用导入或子类。

在 Django 应用中，通常的做法是在文件的开关增加 from yourapplication.default_settings import * 进行显式地导入，然后手工重载 配置。你还可以通过检查一个 YOURAPPLICATION_MODE 之类的环境变量（变量值设置 为 production 或 development 等等）来导入不同的配置文件。

一个有趣的方案是使用类和类的继承来配置:

```
class Config(object):
    DEBUG = False
    TESTING = False
    DATABASE_URI = 'sqlite://:memory:'

class ProductionConfig(Config):
    DATABASE_URI = 'mysql://user@localhost/foo'

class DevelopmentConfig(Config):
    DEBUG = True

class TestingConfig(Config):
    TESTING = True
```

如果要使用这样的方案，那么必须使用 from_object():

```
app.config.from_object('configmodule.ProductionConfig')
```

配置的方法多种多样，由你定度。以下是一些建议：

- 在版本控制中保存一个缺省配置。要么在应用中使用这些缺省配置，要么先导入缺省配置然后用你自己的配置文件来重载缺省配置。

- 使用一个环境变量来切换不同的配置。这样就可以在 Python 解释器外进行切换，而 根本不用改动代码，使开发和部署更方便，更快捷。如果你经常在不同的项目间 切换，那么你甚至可以创建代码来激活 virtualenv 并导出开发配置。
- 在生产应用中使用 fabric 之类的工具，向服务器分别传送代码和配置。更多细节参见使用 Fabric 部署方案。

实例文件夹

New in version 0.8.

Flask 0.8 引入了实例文件夹。 Flask 花了很长时间才能够直接使用应用文件夹的路径（ 通过 Flask.root_path ）。这也是许多开发者载入应用文件夹外的配置的方法。 不幸的是这种方法只能用于应用不是一个包的情况下，即根路径指向包的内容的情况。

Flask 0.8 引入了一个新的属性： Flask.instance_path 。它指向一个新名词： “实例文件夹”。实例文件夹应当处于版本控制中并进行特殊部署。这个文件夹特别适合存放需要在应用运行中改变的东西或者配置文件。

可以要么在创建 Flask 应用时显式地提供实例文件夹的路径，要么让 Flask 自动探测实例文件夹。显式定义使用 instance_path 参数:

```
app = Flask(__name__, instance_path='/path/to/instance/folder')
```

请记住，这里提供的路径必须是绝对路径。

如果 instance_path 参数没有提供，那么会使用以下缺省位置:

- 未安装的模块:

```
/myapp.py
/instance
```

- 未安装的包:

```
/myapp
   /__init__.py
/instance
```

- 已安装的模块或包:

```
$PREFIX/lib/python2.X/site-packages/myapp
$PREFIX/var/myapp-instance
```

$PREFIX 是你的 Python 安装的前缀。可能是 /usr 或你的 virtualenv 的路径。可以通过打印 sys.prefix 的值来查看当前的前缀的值。

既然可以通过使用配置对象来根据关联文件名从文件中载入配置，那么就可以通过改变与 实例路径相关联的文件名来按需要载入不同配置。在配置文件中的关联路径的行为可以在 “关联到应用的根路径”（ 缺省的 ）和 “关联到实例文件夹”之间变换，具体通过应用构建函数中的 instance_relative_config 来实现:

```
app = Flask(__name__, instance_relative_config=True)
```

以下是一个完整的配置 Flask 的例子，从一个模块预先载入配置，然后从配置文件夹中的 一个配置文件（如果这个文件存在的话）载入要重载的配置:

```
app = Flask(__name__, instance_relative_config=True)
app.config.from_object('yourapplication.default_settings')
app.config.from_pyfile('application.cfg', silent=True)
```

通过 Flask.instance_path 可以找到实例文件夹的路径。 Flask 还提供一个打开实例文件夹中的文件的快捷方法： `Flask.open_instance_resource()` 。

举例说明:

```
filename = os.path.join(app.instance_path, 'application.cfg')
with open(filename) as f:
    config = f.read()

 # 或者通过使用 open_instance_resource:

with app.open_instance_resource('application.cfg') as f:
    config = f.read()
```

11

信号

New in version 0.6.

Flask 自 0.6 版本开始在内部支持信号。信号功能由优秀的 blinker 库提供支持， 如果没有安装该库就无法使用信号功能，但不影响其他功能。

什么是信号? 当核心框架的其他地方或另一个 Flask 扩展中发生动作时，信号通过发送 通知来帮助你解耦应用。简言之，信号允许某个发送者通知接收者有事情发生了。

Flask 自身有许多信号，其他扩展可能还会带来更多信号。请记住，信号使用目的是通知 接收者，不应该鼓励接收者修改数据。你会注意到信号的功能与一些内建的装饰器类似（ 如 request_started 与 before_request() 非常 相似 ），但是它们的工作原理不同。例如核心的 before_request() 处理器以一定的顺序执行，并且可以提前退出请求，返回一个响应。相反，所有的信号处理器是乱序执行的，并且不修改任何数据。

信号的最大优势是可以安全快速的订阅。比如，在单元测试中这些临时订阅十分有用。 假设你想知道请求需要渲染哪个模块，信号可以给你答案。

订阅信号

使用信号的 connect() 方法可以订阅该信号。该方法的 第一个参数是当信号发出时所调用的函数。第二个参数是可选参数，定义一个发送者。 使用 disconnect() 方法可以退订信号。

所有核心 Flask 信号的发送者是应用本身。因此当订阅信号时请指定发送者，除非你真的 想要收听应用的所有信号。当你正在开发一个扩展时，尤其要注意这点。

下面是一个环境管理器的辅助工具，可用于在单元测试中辨别哪个模板被渲染了，哪些变量被传递给了模板:

```
from flask import template_rendered
from contextlib import contextmanager

@contextmanager
def captured_templates(app):
    recorded = []
    def record(sender, template, context, **extra):
        recorded.append((template, context))
    template_rendered.connect(record, app)
    try:
        yield recorded
    finally:
        template_rendered.disconnect(record, app)
```

上例可以在测试客户端中轻松使用:

```
with captured_templates(app) as templates:
    rv = app.test_client().get('/')
    assert rv.status_code == 200
    assert len(templates) == 1
    template, context = templates[0]
    assert template.name == 'index.html'
    assert len(context['items']) == 10
```

为了使 Flask 在向信号中添加新的参数时不发生错误，请确保使用一个额外的 **extra 参数。

在 with 代码块中，所有由 app 渲染的模板会被记录在 templates 变量中。每当有模板被渲染，模板对象及环境就会追加到变量中。

另外还有一个方便的辅助方法（ connected_to() ）。它允许临时把一个使用环境对象的函数订阅到一个信号。因为环境对象的返回值不能被指定，所以必须把列表作为参数:

```
from flask import template_rendered

def captured_templates(app, recorded, **extra):
    def record(sender, template, context):
        recorded.append((template, context))
    return template_rendered.connected_to(record, app)
```

上例可以这样使用:

```
templates = []
with captured_templates(app, templates, **extra):
    ...
    template, context = templates[0]
```

Blinker API 变化

Blinker version 1.1 版本中增加了 connected_to() 方法。

创建信号

如果相要在你自己的应用中使用信号，那么可以直接使用 blinker 库。最常见的,也是最推荐的方法是在自定义的 Namespace 中命名信号:

```
from blinker import Namespace
my_signals = Namespace()
```

接着可以像这样创建新的信号:

```
model_saved = my_signals.signal('model-saved')
```

信号的名称应当是唯一的，并且应当简明以便于调试。可以通过 name 属性获得信号的名称。

扩展开发者注意

如果你正在编写一个 Flask 扩展，并且想要妥善处理 blinker 安装缺失的情况，那么可以使用 flask.signals.Namespace 类。

发送信号

如果想要发送信号，可以使用 send() 方法。它的第一个参数是一个发送者，其他参数要发送给订阅者的东西，其他参数是可选的:

```
class Model(object):
    ...

    def save(self):
        model_saved.send(self)
```

请谨慎选择发送者。如果是一个发送信号的类，请把 self 作为发送者。如果发送信号的是一个随机的函数，那么可以把 current_app._get_current_object() 作为发送者。

传递代理作为发送者

不要把 current_app 作为发送者传递给信号。请使用 current_app._get_current_object() 。因为 current_app 是 一个代理，不是实际的应用对象。

信号与 Flask 的请求环境

信号在接收时，完全支持请求环境 。在 request_started 和 request_finished 本地环境变量 始终可用。因此你可以依赖 flask.g 及其他本地环境变量。 请注意在 发送信号 中所述的限制和 request_tearing_down 信号。

信号订阅装饰器

Blinker 1.1 版本中你还可以通过使用新的 connect_via() 装饰器轻松订阅信号:

```
from flask import template_rendered

@template_rendered.connect_via(app)
def when_template_rendered(sender, template, context, **extra):
    print 'Template %s is rendered with %s' % (template.name, context)
```

核心信号

Flask 中有以下信号:

`flask.template_rendered` 这个信号发送于一个模板被渲染成功后。信号传递的 template 是模板的实例， context 是环境对象是一个字典。

订阅示例:

```
def log_template_renders(sender, template, context, **extra):
    sender.logger.debug('Rendering template "%s" with context %s',
                        template.name or 'string template',
                        context)

from flask import template_rendered
template_rendered.connect(log_template_renders, app)
flask.request_started
```

这个信号发送于请求开始之前，且请求环境设置完成之后。因为请求环境已经绑定， 所以订阅者可以用标准的全局代理，如 request 来操作请求。

订阅示例:

```
def log_request(sender, **extra):
    sender.logger.debug('Request context is set up')

from flask import request_started
request_started.connect(log_request, app)
flask.request_finished
```

这个信号发送于向客户端发送响应之前。信号传递的 response 为将要发送的响应。

订阅示例:

```
def log_response(sender, response, **extra):
    sender.logger.debug('Request context is about to close down.  '
                        'Response: %s', response)

from flask import request_finished
request_finished.connect(log_response, app)
flask.got_request_exception
```

这个信号发送于请求进行中发生异常的时候。它的发送早于标准异常处理介于。 在调试模式下，虽然没有异常处理，但发生异常时也发送这个信号。信号传递的 exception 是异常对象。

订阅示例:

```
def log_exception(sender, exception, **extra):
    sender.logger.debug('Got exception during processing: %s', exception)

from flask import got_request_exception
got_request_exception.connect(log_exception, app)
flask.request_tearing_down
```

这个信号发送于请求崩溃的时候，不管是否引发异常。目前，侦听此信号的函数在一般崩溃处理器后调用，但是没有什么东西可用。

订阅示例:

```
def close_db_connection(sender, **extra):
    session.close()

from flask import appcontext_tearing_down
request_tearing_down.connect(close_db_connection, app)
```

在 Flask 版本 0.9 中，这还会传递一个 exc 关键字参数，如果这个参数存在的话。 这个参数是引发崩溃的异常的引用。

flask.appcontext_tearing_down 当应用环境崩溃时发送这个信号。这个信号总是会发送，甚至是因为一个异常引发的 崩溃。侦听这个信号的函数会在常规崩溃处理器后被调用，但是你无法回馈这个信号。

订阅示例:

```
def close_db_connection(sender, **extra):
    session.close()

from flask import request_tearing_down
appcontext_tearing_down.connect(close_db_connection, app)
```

这还会传递一个 exc 关键字参数，如果这个参数存在的话。这个参数是引发崩溃的异常的引用。

flask.appcontext_pushed 当一个应用的环境被压入时，应用会发送这个信号。这个信号通常用于在单元测试中临时钩接信息。例如可以用于改变 g 对象中现存的资源。

用法示例:

```
from contextlib import contextmanager
from flask import appcontext_pushed

@contextmanager
def user_set(app, user):
    def handler(sender, **kwargs):
        g.user = user
    with appcontext_pushed.connected_to(handler, app):
        yield
```

在测试代码中这样写:

```
def test_user_me(self):
    with user_set(app, 'john'):
        c = app.test_client()
        resp = c.get('/users/me')
        assert resp.data == 'username=john'
New in version 0.10.

appcontext_popped
```

当一个应用的环境被弹出时，应用会发送这个信号。这个信号通常写成 appcontext_tearing_down 信号。

New in version 0.10.

flask.message_flashed 当应用闪现一个消息时会发出这个信号。`message` 参数是消息内容， `category` 参数是消息类别。

订阅示例:

```
recorded = []
def record(sender, message, category, **extra):
    recorded.append((message, category))

from flask import message_flashed
message_flashed.connect(record, app)
New in version 0.10.
```

12

可插拔视图

New in version 0.7.

Flask 0.7 版本引入了可插拔视图。可插拔视图基于使用类来代替函数，其灵感来自于 Django 的通用视图。可插拔视图的主要用途是用可定制的、可插拔的视图来替代部分实现。

基本原理

假设有一个函数用于从数据库中载入一个对象列表并在模板中渲染:

```
@app.route('/users/')
def show_users(page):
    users = User.query.all()
    return render_template('users.html', users=users)
```

上例简单而灵活。但是如果要把这个视图变成一个可以用于其他模型和模板的通用视图， 那么这个视图还是不够灵活。因此，我们就需要引入可插拔的、基于类的视图。第一步， 可以把它转换为一个基础视图:

```
from flask.views import View

class ShowUsers(View):

    def dispatch_request(self):
        users = User.query.all()
        return render_template('users.html', objects=users)

app.add_url_rule('/users/', view_func=ShowUsers.as_view('show_users'))
```

就如你所看到的，必须做的是创建一个 flask.views.View 的子类，并且执行 dispatch_request() 。然后必须通过使用 as_view() 方法把类转换为实际视图函数。传递给函数的 字符串是最终视图的名称。但是这本身没有什么帮助，所以让我们来小小地重构一下:

```
from flask.views import View

class ListView(View):

    def get_template_name(self):
        raise NotImplementedError()

    def render_template(self, context):
        return render_template(self.get_template_name(), **context)

    def dispatch_request(self):
        context = {'objects': self.get_objects()}
        return self.render_template(context)

class UserView(ListView):

    def get_template_name(self):
```

```
        return 'users.html'

    def get_objects(self):
        return User.query.all()
```

这样做对于示例中的小应用没有什么用途，但是可以足够清楚的解释基本原理。当你有一个基础视图类时，问题就来了：类的 self 指向什么？解决之道是：每当请求发出时就创建一个类的新实例，并且根据来自 URL 规则的参数调用 dispatch_request() 方法。类本身根据参数实例化后传递给 as_view() 函数。例如可以这样写一个类:

```
class RenderTemplateView(View):
    def __init__(self, template_name):
        self.template_name = template_name
    def dispatch_request(self):
        return render_template(self.template_name)
```

然后可以这样注册:

```
app.add_url_rule('/about', view_func=RenderTemplateView.as_view(
    'about_page', template_name='about.html'))
```

方法提示

可插拔视图可以像普通函数一样加入应用。加入的方式有两种，一种是使用 route() ，另一种是使用更好的 add_url_rule() 。在加入的视图中应该提供所使用的 HTTP 方法的名称。提供名称的方法是使用 methods 属性:

```
class MyView(View):
    methods = ['GET', 'POST']

    def dispatch_request(self):
        if request.method == 'POST':
            ...
        ...

app.add_url_rule('/myview', view_func=MyView.as_view('myview'))
```

基于方法调度

对于 REST 式的 API 来说，为每种 HTTP 方法提供相对应的不同函数显得尤为有用。使用 flask.views.MethodView 可以轻易做到这点。在这个类中，每个 HTTP 方法 都映射到一个同名函数（函数名称为小写字母）：

```
from flask.views import MethodView

class UserAPI(MethodView):

    def get(self):
        users = User.query.all()
        ...

    def post(self):
        user = User.from_form_data(request.form)
        ...

app.add_url_rule('/users/', view_func=UserAPI.as_view('users'))
```

使用这种方式，不必提供 methods 属性，它会自动使用相应的类方法。

装饰视图

视图函数会被添加到路由系统中，而视图类则不会。因此视图类不需要装饰，只能以手工 使用 as_view() 来装饰返回值:

```
def user_required(f):
    """Checks whether user is logged in or raises error 401."""
    def decorator(*args, **kwargs):
        if not g.user:
            abort(401)
        return f(*args, **kwargs)
    return decorator

view = user_required(UserAPI.as_view('users'))
app.add_url_rule('/users/', view_func=view)
```

自 Flask 0.8 版本开始，新加了一种选择：在视图类中定义装饰的列表:

```
class UserAPI(MethodView):
    decorators = [user_required]
```

请牢记：因为从调用者的角度来看，类的 self 被隐藏了，所以不能在类的方法上单独使用装饰器。

用于 API 的方法视图

网络 API 经常直接对应 HTTP 变量，因此很有必要实现基于 MethodView 的 API 。即多数时候， API 需要把不同的 URL 规则应用到同一个方法视图。例如，假设你需要这样使用一个 user 对象：

|URL| 方法| 说明| |:----|:--- |:--- | |/users/ |GET| 给出一个包含所有用户的列表| |/users/ |POST| 创建一个新用户| |/users/ |GET |显示一个用户| |/users/ |PUT |更新一个用户| |/users/ |DELETE |删除一个用户| 那么如何使用 MethodView 来实现呢？ 方法是使用多个规则对应 到同一个视图。

假设视图是这样的：

```
class UserAPI(MethodView):

    def get(self, user_id):
        if user_id is None:
            # 返回一个包含所有用户的列表
            pass
        else:
            # 显示一个用户
            pass

    def post(self):
        # 创建一个新用户
        pass

    def delete(self, user_id):
        # 删除一个用户
        pass

    def put(self, user_id):
        # update a single user
        pass
```

那么如何把这个视图挂接到路由系统呢？ 方法是增加两个规则并为每个规则显式声明方法：

```
user_view = UserAPI.as_view('user_api')
app.add_url_rule('/users/', defaults={'user_id': None},
                 view_func=user_view, methods=['GET',])
app.add_url_rule('/users/', view_func=user_view, methods=['POST',])
app.add_url_rule('/users/<int:user_id>', view_func=user_view,
                 methods=['GET', 'PUT', 'DELETE'])
```

如果你有许多类似的 API ， 那么可以代码如下：

```
def register_api(view, endpoint, url, pk='id', pk_type='int'):
    view_func = view.as_view(endpoint)
    app.add_url_rule(url, defaults={pk: None},
                     view_func=view_func, methods=['GET',])
    app.add_url_rule(url, view_func=view_func, methods=['POST',])
    app.add_url_rule('%s<%s:%s>' % (url, pk_type, pk), view_func=view_func,
                     methods=['GET', 'PUT', 'DELETE'])

register_api(UserAPI, 'user_api', '/users/', pk='user_id')
```

13 应用环境

New in version 0.9.

Flask 的设计思路之一是代码有两种不同的运行“状态”。一种是“安装”状态，从 Flask 对象被实例化后开始，到第一个请求到来之前。在这种状态下，以下规则 成立：

- 可以安全地修改应用对象。
- 还没有请求需要处理。
- 没有应用对象的引用，因此无法修改应用对象。

相反，在请求处理时，以下规则成立：

- 当请求活动时，本地环境对象（ flask.request 或其他对象 ）指向当前请求。
- 这些本地环境对象可以任意修改。

除了上述两种状态之外，还有位于两者之间的第三种状态。这种状态下，你正在处理应用， 但是没有活动的请求。就类似于，你座在 Python 交互终端前，与应用或一个命令行程序互动。

应用环境是 current_app 本地环境的源动力。

应用环境的作用

应用环境存在的主要原因是，以前过多的功能依赖于请求环境，缺乏更好的处理方案。 Flask 的一个重要设计原则是可以在同一个 Pyhton 进程中运行多个应用。

那么，代码如何找到“正确”的应用呢？以前我们推荐显式地传递应用，但是有可能会引发库与库之间的干扰。

问题的解决方法是使用 current_app 代理。 current_app 是绑定到当前请求的应用的引用。但是没有请求的情况下使用请求环境是一件奢侈在事情，于是就引入了应用环境。

创建一个应用环境

创建应用环境有两种方法。一种是隐式的：当一个请求环境被创建时，如果有需要就会 相应创建一个应用环境。因此，你可以忽略应用环境的存在，除非你要使用它。

另一种是显式的，使用 app_context() 方法:

```
from flask import Flask, current_app

app = Flask(__name__)
with app.app_context():
    # 在本代码块中， current_app 指向应用。
    print current_app.name
```

在 SERVER_NAME 被设置的情况下，应用环境还被 url_for() 函数 使用。这样可以让你在没有请求的情况下生成 URL 。

环境的作用域

应用环境按需创建和消灭，它从来不在线程之间移动，也不被不同请求共享。因此，它是 一个存放数据库连接信息和其他信息的好地方。内部的栈对象被称为 flask._app_ctx_stack 。扩展可以在栈的最顶端自由储存信息，前提是使用唯一 的名称，而 flask.g 对象是留给用户使用的。

更多信息参见 Flask 扩展开发 。

环境的用法

环境的典型用途是缓存一个请求需要预备或使用的资源，例如一个数据库连接。因为环境是一个 Flask 的应用和扩展共享的地方，所以储存的资源必须使用独一无二的名称。

最常见的用法是把资源管理分为以下两个部分：

1. 在环境中缓存一个隐式的资源。
2. 资源释放后的环境解散。

通常会有一个形如 get_X() 函数，这个函数的用途是当资源 X 存在时就返回 这个资源，否则就创建这个资源。还会有一个 teardown_X() 函数用作解散句柄。

这是一个连接数据库的例子:

```
import sqlite3
from flask import g

def get_db():
    db = getattr(g, '_database', None)
    if db is None:
        db = g._database = connect_to_database()
    return db

@app.teardown_appcontext
def teardown_db(exception):
    db = getattr(g, '_database', None)
    if db is not None:
        db.close()
```

第一次调用 get_db() 时，连接将会被建立。建立的过程中隐式地使用了一个 LocalProxy 类:

```
from werkzeug.local import LocalProxy
db = LocalProxy(get_db)
```

这样，用户就可以通过 get_db() 来直接访问 db 了。

14

请求环境

本文讲述 Flask 0.7 版本的运行方式，与旧版本的运行方式基本相同，但也有一些细微的 差别。

建议你在阅读本文之前，先阅读应用环境 。

深入本地环境

假设有一个工具函数，这个函数返回用户重定向的 URL（包括 URL 的 next 参数、或 HTTP 推荐和索引页面）：

```
from flask import request, url_for

def redirect_url():
    return request.args.get('next') or \
           request.referrer or \
           url_for('index')
```

如上例所示，这个函数访问了请求对象。如果你在一个普通的 Python 解释器中运行这个函数，那么会看到如下异常：

```
>>> redirect_url()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'request'
```

这是因为现在我们没有一个可以访问的请求。所以我们只能创建一个请求并绑定到当前环境中。test_request_context 方法可以创建一个 RequestContext：

```
>>> ctx = app.test_request_context('/?next=http://example.com/')
```

这个环境有两种使用方法：一种是使用 with 语句；另一种是调用 push() 和 pop() 方法：

```
>>> ctx.push()
```

现在可以使用请求对象了：

```
>>> redirect_url()
u'http://example.com/'
```

直到你调用 pop：

```
>>> ctx.pop()
```

可以把请求环境理解为一个堆栈，可以多次压入和弹出，可以方便地执行一个像内部重定向之类的东西。

关于在 Python 解释器中使用请求环境的更多内容参见在 Shell 中使用 Flask 。

环境的工作原理

如果深入 Flask WSGI 应用内部，那么会找到类似如下代码:

```
def wsgi_app(self, environ):
    with self.request_context(environ):
        try:
            response = self.full_dispatch_request()
        except Exception, e:
            response = self.make_response(self.handle_exception(e))
        return response(environ, start_response)
```

request_context() 方法返回一个新的 RequestContext 对象，并且使用 with 语句把这个对象绑定到环境。在 with 语句块中，在同一个线程中调用的所有东西可以访问全局请求（ flask.request 或其他 ）。

请求环境的工作方式就像一个堆栈，栈顶是当前活动请求。 push() 把环境压入堆栈中，而 pop() 把环境弹出。弹出的同时，会执行应用的 teardown_request() 函数。

另一件要注意的事情是：请求环境会在压入时自动创建一个应用环境 。在此之前，应用没有应用环境。

回调和错误处理

如果在请求处理的过程中发生错误，那么 Flask 会如何处理呢？自 Flask 0.7 版本之后， 处理方式有所改变。这是为了更方便地反映到底发生了什么情况。新的处理方式非常简单：

1. 在每个请求之前，会执行所有 before_request() 函数。如果 其中一个函数返回一个响应，那么其他函数将不再调用。但是在任何情况下，这个 返回值将会替代视图的返回值。
2. 如果 before_request() 函数均没有响应，那么就会进行正常的 请求处理，匹配相应的视图，返回响应。
3. 接着，视图的返回值会转换为一个实际的响应对象并交给 after_request() 函数处理。在处理过程中，这个对象可能会被 替换或修改。
4. 请求处理的最后一环是执行 teardown_request() 函数。这类 函数在任何情况下都会被执行，甚至是在发生未处理异常或请求预处理器没有执行（ 例如在测试环境下，有时不想执行 ）的情况下。

那么如果出错了会怎么样？在生产模式下，如果一个异常未被主要捕获处理，那么会调用 500 内部服务器处理器。在开发模式下，引发的异常不再被进一步处理，会提交给 WSGI 服务器。因此，需要使用交互调试器来查看调试信息。

Flask 0.7 版本的重大变化是内部服务器错误不再由请求后回调函数来处理，并且请求后 回调函数也不保证一定被执行。这样使得内部调试代码更整洁、更易懂和更容易定制。

同时还引入了新的卸载函数，这个函数在请求结束时一定会执行。

卸载回调函数

卸载回调函数的特殊之处在于其调用的时机是不固定的。严格地说，调用时机取决于其绑定的 RequestContext 对象的生命周期。当请求环境弹出时就 会调用 teardown_request() 函数。

请求环境的生命周期是会变化的，当请求环境位于测试客户端中的 with 语句中或者在 命令行下使用请求环境时，其生命周期会被延长。因此知道生命周期是否被延长是很重要的:

```
with app.test_client() as client:
    resp = client.get('/foo')
    # 到这里还没有调用卸载函数。即使这时响应已经结束，并且已经
    # 获得响应对象，还是不会调用卸载函数。

 # 只有到这里才会调用卸载函数。另外，如果另一个请求在客户端中被

 # 激发，也会调用卸载函数。
```

在使用命令行时，可以清楚地看到运行方式:

```
>>> app = Flask(__name__)
>>> @app.teardown_request
... def teardown_request(exception=None):
...     print 'this runs after request'
...
>>> ctx = app.test_request_context()
>>> ctx.push()
>>> ctx.pop()
this runs after request
>>>
```

记牢记：卸载函数在任何情况下都会被执行，甚至是在请求预处理回调函数没有执行， 但是发生异常的情况下。有的测试系统可能会临时创建一个请求环境，但是不执行 预处理器。请正确使用卸载处理器，确保它们不会执行失败。

关于代理

部分 Flask 提供的对象是其他对象的代理。使用代理的原因是代理对象共享于不同的 线程，它们在后台根据需要把实际的对象分配给不同的线程。

多数情况下，你不需要关心这个。但是也有例外，在下列情况有下，知道对象是一个代理对象是有好处的：

想要执行真正的实例检查的情况。因为代理对象不会假冒被代理对象的对象类型， 因此，必须检查被代理的实际对象（ 参见下面的 _get_current_object ）。 对象引用非常重要的情况（ 例如发送 信号 ）。 如果想要访问被代理的对象，可以使用 _get_current_object() 方法:

```
app = current_app._get_current_object()
my_signal.send(app)
```

出错时的环境保存

不管是否出错，在请求结束时，请求环境会被弹出，并且所有相关联的数据会被销毁。 但是在开发过程中，可能需要在出现异常时保留相关信息。在 Flask 0.6 版本及更早的 版本中，在发生异常时，请求环境不会被弹出，以便于交互调试器提供重要信息。

自 Flask 0.7 版本开始，可以通过设置 `PRESERVE_CONTEXT_ON_EXCEPTION` 配置变量来更好地控制环境的保存。缺省情况下，这个配置变更与 DEBUG 变更关联。如果在调试模式下，那么环境会被保留，而在生产模式下则不保留。

不要在生产环境下强制激活 `PRESERVE_CONTEXT_ON_EXCEPTION` ，因为这会在出现异常 时导致应用内存溢出。但是在调试模式下使用这个变更是十分有用的，你可以获得在生产模式下出错时的环境。

15

使用蓝图的模块化应用

New in version 0.7.

为了在一个或多个应用中，使应用模块化并且支持常用方案， Flask 引入了 蓝图 概念。蓝图可以极大地简化大型应用并为扩展提供集中的注册入口。

Blueprint 对象与 Flask 应用对象的工作方式类似，但不是一个真正 的应用。它更像一个用于构建和扩展应用的蓝图 。

为什么使用蓝图?

Flask 中蓝图有以下用途:

1. 把一个应用分解为一套蓝图。这是针对大型应用的理想方案：一个项目可以实例化一个 应用，初始化多个扩展，并注册许多蓝图。
2. 在一个应用的 URL 前缀和（或）子域上注册一个蓝图。 URL 前缀和（或）子域的参数 成为蓝图中所有视图的通用视图参数（缺省情况下）。
3. 使用不同的 URL 规则在应用中多次注册蓝图。
4. 通过蓝图提供模板过滤器、静态文件、模板和其他工具。蓝图不必执行应用或视图 函数。
5. 当初始化一个 Flask 扩展时，为以上任意一种用途注册一个蓝图。

Flask 中的蓝图不是一个可插拔的应用，因为它不是一个真正的应用，而是一套可以注册 在应用中的操作，并且可以注册多次。那么为什么不使用多个应用对象呢? 可以使用多个 应用对象（参见应用调度 ），但是这样会导致每个应用都使用自己独立的配置，且只能在 WSGI 层中管理应用。

而如果使用蓝图，那么应用会在 Flask 层中进行管理，共享配置，通过注册按需改变应用 对象。蓝图的缺点是一旦应用被创建后，只有销毁整个应用对象才能注销蓝图。

蓝图的概念

蓝图的基本概念是：在蓝图被注册到应用之后，所要执行的操作的集合。当分配请求时， Flask 会把蓝图和视图函数关联起来，并生成两个端点之前的 URL 。

第一个蓝图

以下是一个最基本的蓝图示例。在这里，我们将使用蓝图来简单地渲染静态模板:

```
from flask import Blueprint, render_template, abort
from jinja2 import TemplateNotFound

simple_page = Blueprint('simple_page', __name__,
                        template_folder='templates')

@simple_page.route('/', defaults={'page': 'index'})
@simple_page.route('/<page>')
def show(page):
    try:
        return render_template('pages/%s.html' % page)
    except TemplateNotFound:
        abort(404)
```

当你使用 @simple_page.route 装饰器绑定一个函数时，蓝图会记录下所登记的 show 函数。当以后在应用中注册蓝图时，这个函数会被注册到应用中。另外，它会把 构建 Blueprint 时所使用的名称（ 在本例为 simple_page ）作为函数端点 的前缀。

注册蓝图

可以这样注册蓝图:

```
from flask import Flask
from yourapplication.simple_page import simple_page

app = Flask(__name__)
app.register_blueprint(simple_page)
```

以下是注册蓝图后形成的规则:

```
[<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
 <Rule '/<page>' (HEAD, OPTIONS, GET) -> simple_page.show>,
 <Rule '/' (HEAD, OPTIONS, GET) -> simple_page.show>]
```

第一条很明显，是来自于应用本身的用于静态文件的。后面两条是用于蓝图 simple_page 的 show 函数的。你可以看到，它们的前缀都是蓝图的名称，并且使用一个点（ . ）来分隔。

蓝图还可以挂接到不同的位置:

```
app.register_blueprint(simple_page, url_prefix='/pages')
```

这样就会形成如下规则:

```
[<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
 <Rule '/pages/<page>' (HEAD, OPTIONS, GET) -> simple_page.show>,
 <Rule '/pages/' (HEAD, OPTIONS, GET) -> simple_page.show>]
```

总之，你可以多次注册蓝图，但是不一定每个蓝图都能正确响应。是否能够多次注册实际 上取决于你的蓝图是如何编写的，是否能根据不同的位置做出正确的响应。

蓝图资源

蓝图还可以用于提供资源。有时候，我们仅仅是为了使用一些资源而使用蓝图。

蓝图资源文件夹

和普通应用一样，蓝图一般都放在一个文件夹中。虽然多个蓝图可以共存于同一个文件夹中，但是最好不要这样做。

文件夹由 Blueprint 的第二个参数指定，通常为 `__name__` 。这个参数指定与蓝图相关的逻辑 Python 模块或包。如果这个参数指向的是实际的 Python 包（文件 系统中的一个文件夹），那么它就是资源文件夹。如果是一个模块，那么这个模块包含的 包就是资源文件夹。可以通过 Blueprint.root_path 属性来查看蓝图的资源文件夹:

```
>>> simple_page.root_path
'/Users/username/TestProject/yourapplication'
```

可以使用 open_resource() 函数快速打开这个文件夹中的资源:

```
with simple_page.open_resource('static/style.css') as f:
    code = f.read()
```

静态文件

蓝图的第三个参数是 static_folder 。这个参数用以指定蓝图的静态文件所在的文件夹，它可以是一个绝对路径也可以是相对路径。:

```
admin = Blueprint('admin', __name__, static_folder='static')
```

缺省情况下，路径最右端的部分是在 URL 中暴露的部分。上例中的文件夹为 static ，那么 URL 应该是蓝图加上 `/static` 。蓝图注册为 `/admin` ，那么 静态文件夹就是 `/admin/static` 。

端点的名称是 `blueprint_name.static` ，因此你可以使用和应用中的文件夹一样的方法 来生成其 URL:

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

模板

如果你想使用蓝图来暴露模板，那么可以使用 Blueprint 的 template_folder 参数:

```
admin = Blueprint('admin', __name__, template_folder='templates')
```

和静态文件一样，指向蓝图资源文件夹的路径可以是绝对的也可以是相对的。蓝图中的 模板文件夹会被添加到模板搜索路径中，但其优先级低于实际应用的模板文件夹。这样在 实际应用中可以方便地重载蓝图提供的模板。

假设你的蓝图便于 yourapplication/admin 中，要渲染的模板是 'admin/index.html' ， template_folder 参数值为 templates ，那么真正的 模板文件为： `yourapplication/admin/templates/admin/index.html` 。

创建 URL

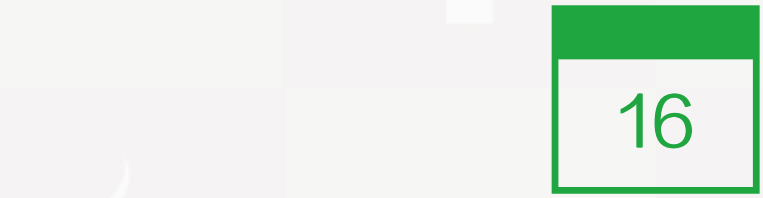
如果要创建页面链接，可以和通常一样使用 url_for() 函数，只是要把蓝图名称作为端点的前缀，并且用一个点（.）来 分隔:

```
url_for('admin.index')
```

另外，如果在一个蓝图的视图函数或者被渲染的模板中需要链接同一个蓝图中的其他端点，那么使用相对重定向，只使用一个点使用为前缀:

```
url_for('.index')
```

如果当前请求被分配到 admin 蓝图端点时，上例会链接到 admin.index 。



16 Flask 扩展

Flask 扩展以各种方式扩展了 Flask 的功能，比如增强对数据库的支持等等。

查找扩展

Flask 扩展都列在 Flask 扩展注册 中，并且可以使用 easy_install 或 pip 下载。如果你把一个扩展作为依赖添加到你的 requirements.rst 或 setup.py 文件，那么它们可以使用一个简单的命令安装或随着应用一起安装。

使用扩展

扩展一般都有说明如何使用的文档，这些文档应该和扩展一起发行。扩展如何运行没有 统一的要求，但是一般在常见位置导入扩展。假设一个扩展称为 Flask-Foo 或 Foo-Flask ，那么总是可以导入 flask.ext.foo:

```
from flask.ext import foo
```

Flask 0.8 以前的版本

如果你正在使用 Flask 0.7 版本或更早版本， flask.ext 包是不存在的。你 必须根据扩展的发行方式导入 flaskext.foo 或 flask_foo 。如果你要开发一个 支持 Flask 0.7 版本或更早版本的应用，那么你应当还是从 flask.ext 包中 导入。我们提供了一个兼容模块用以兼容老版本的 Flask ，你可以从 github 下载： flaskext_compat.py

使用方法如下:

```
import flaskext_compat
flaskext_compat.activate()

from flask.ext import foo
```

一旦 flaskext_compat 模块被激活， flask.ext 就会存在，就可以从这个 包导入扩展。

17

在 Shell 中使用 Flask

New in version 0.3.

喜欢 Python 的原因之一是交互式的 shell，它可以让你实时运行 Python 命令，并且立即得到结果。Flask 本身不带交互 shell，因为它不需要特定的前期设置，只要在 shell 中导入你的应用就可以开始使用了。

有些辅助工具可以让你在 shell 中更舒服。在交互终端中最大的问题是你不会像浏览器一样触发一个请求，这就意味着无法使用 g 和 request 等对象。那么如何在 shell 中测试依赖这些对象的代码呢?

这里有一些有用的辅助函数。请记住，这些辅助函数不仅仅只能用于 shell，还可以用于单元测试和其他需要假冒请求环境的情况下。

在读下去之前最好你已经读过请求环境一节。

创建一个请求环境

在 shell 中创建一个正确的请求环境的最简便的方法是使用 test_request_context 方法。这个方法会创建一个 RequestContext：

```
>>> ctx = app.test_request_context()
```

通常你会使用 with 语句来激活请求对象，但是在 shell 中，可以简便地手动使用 push() 和 pop() 方法：

```
>>> ctx.push()
```

从这里开始，直到调用 pop 之前，你可以使用请求对象：

```
>>> ctx.pop()
```

发送请求前/后动作

仅仅创建一个请求环境还是不够的，需要在请求前运行的代码还是没有运行。比如，在请求前可以会需要转接数据库，或者把用户信息储存在 g 对象中。

使用 preprocess_request() 可以方便地模拟请求前/后动作：

```
>>> ctx = app.test_request_context()
>>> ctx.push()
>>> app.preprocess_request()
```

请记住， preprocess_request() 函数可以会返回一个响应对象。 如果返回的话请忽略它。

如果要关闭一个请求，那么你需要在请求后函数（ 由 process_response() 触发 ）作用于响应对象前关闭：

```
>>> app.process_response(app.response_class())
<Response 0 bytes [200 OK]>
>>> ctx.pop()
```

teardown_request() 函数会在环境弹出后自动执行。我们可以使用 这些函数来销毁请求环境所需要使用的资源（ 如数据库连接 ）。

在 Shell 中玩得更爽

如果你喜欢在 shell 中的感觉，那么你可以创建一个导入有关东西的模块，在模块中还 可以定义一些辅助方法，如初始化数据库或者删除表等等。假设这个模块名为 shelltools ，那么在开始时你可以：

```
>>> from shelltools import *
```

18

Flask 方案

有一些东西是大多数网络应用都会用到的。比如许多应用都会使用关系型数据库和用户 验证，在请求之前连接数据库并得到当前登录用户的信息，在请求之后关闭数据库连接。

更多用户贡献的代码片断和方案参见 Flask 代码片断归档 。

本章包含内容：

- 大型应用
- 应用工厂
- 应用调度
- 实现 API 异常处理
- URL 处理器
- 使用 Distribute 部署
- 使用 Fabric 部署
- 在 Flask 中使用 SQLite 3
- 在 Flask 中使用 SQLAlchemy
- 上传文件
- 缓存
- 视图装饰器
- 使用 WTForms 进行表单验证
- 模板继承
- 消息闪现
- 通过 jQuery 使用 AJAX
- 自定义出错页面
- 惰性载入视图
- 在 Flask 中使用 MongoKit
- 添加一个页面图标
- 流内容
- 延迟的请求回调
- 添加 HTTP 方法重载
- 请求内容校验

- 基于 Celery 的后台任务

大型应用

对于大型应用来说使用包代替模块是一个好主意。使用包非常简单。假设有一个小应用如下:

```
/yourapplication
    /yourapplication.py
    /static
        /style.css
    /templates
        layout.html
        index.html
        login.html
        ...
```

简单的包

要把上例中的小应用装换为大型应用只要在现有应用中创建一个名为 yourapplication 的新文件夹，并把所有东西都移动到这个文件夹内。然后把 yourapplication.py 更名为 `__init__.py` 。（请首先删除所有 .pyc 文件，否则基本上会出问题）

修改完后应该如下例:

```
/yourapplication
    /yourapplication
        /__init__.py
        /static
            /style.css
        /templates
            layout.html
            index.html
            login.html
            ...
```

但是现在如何运行应用呢? 原本的 python `yourapplication/__init__.py` 无法运行了。因为 Python 不希望包内的模块成为启动文件。但是这不是一个大问题，只要在 yourapplication 文件夹旁添加一个 `runserver.py` 文件就可以了，其内容如下:

```
from yourapplication import app
app.run(debug=True)
```

我们从中学到了什么? 现在我们来重构一下应用以适应多模块。只要记住以下几点:

1. Flask 应用对象必须位于 `__init__.py` 文件中。这样每个模块就可以安全地导入了，且 `__name__` 变量会解析到正确的包。
2. 所有视图函数（在顶端有 route() 的）必须在 `__init__.py` 文件中被导入。不是导入对象本身，而是导入视图模块。请 在应用对象创建之后导入视图对象。

`__init__.py` 示例:

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

import yourapplication.views
```

views.py 内容如下:

```
from yourapplication import app

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

最终全部内容如下:

```
/yourapplication
    /runserver.py
    /yourapplication
        /__init__.py
        /views.py
        /static
            /style.css
        /templates
            layout.html
            index.html
            login.html
            ...
```

回环导入

回环导入是指两个模块互相导入，本例中我们添加的 views.py 就与 `__init__.py` 相互依赖。每个 Python 程序员都讨厌回环导入。一般情况下回环导入是个坏主意，但 在这里一点问题都没有。原因是我们没有真正使用 `__init__.py` 中的视图，只是 保证模块被导入，并且我们在文件底部才这样做。

但是这种方式还是有些问题，因为没有办法使用装饰器。要找到解决问题的灵感请参阅大型应用一节。

使用蓝图

对于大型应用推荐把应用分隔为小块，每个小块使用蓝图辅助执行。关于这个主题的介绍 请参阅使用蓝图的模块化应用一节 。

应用工厂

如果你已经在应用中使用了包和蓝图（ 使用蓝图的模块化应用 ），那么还有许多方法可以更 进一步地改进你的应用。常用的方案是导入蓝图后创建应用对象，但是如果在一个函数中创建对象，那么就可以创建多个实例。

那么这样做有什么用呢?

1. 用于测试。可以针对不同的情况使用不同的配置来测试应用。
2. 用于多实例，如果你需要运行同一个应用的不同版本的话。当然你可以在服务器上 使用不同配置运行多个相同应用，但是如果使用应用工厂，那么你可以只使用一个应用进程而得到多个应用实例，这样更容易操控。

那么如何做呢?

基础工厂

方法是在一个函数中设置应用，具体如下:

```
def create_app(config_filename):
    app = Flask(__name__)
    app.config.from_pyfile(config_filename)

    from yourapplication.model import db
    db.init_app(app)

    from yourapplication.views.admin import admin
    from yourapplication.views.frontend import frontend
    app.register_blueprint(admin)
    app.register_blueprint(frontend)

    return app
```

这个方法的缺点是在导入时无法在蓝图中使用应用对象。但是你可以在一个请求中使用它。 如何通过配置来访问应用? 使用 current_app:

```
from flask import current_app, Blueprint, render_template
admin = Blueprint('admin', __name__, url_prefix='/admin')

@admin.route('/')
def index():
    return render_template(current_app.config['INDEX_TEMPLATE'])
```

这里我们在配置中查找模板的名称。

扩展对象初始化时不会绑定到一个应用，应用可以使用 `db.init_app` 来设置扩展。 扩展对象中不会储存特定应用的状态，因此一个扩展可以被多个应用使用。关于扩展设计的更多信息请参阅 Flask 扩展开发 。

当使用 Flask-SQLAlchemy 时，你的 model.py 可能是这样的:

```
from flask.ext.sqlalchemy import SQLAlchemy
 # no app object passed! Instead we use use db.init_app in the factory.

db = SQLAlchemy()

 # create some models
```

使用应用

因此，要使用这样的应用就必须先创建它。下面是一个运行应用的示例 run.py 文件:

```
from yourapplication import create_app
app = create_app('/path/to/config.cfg')
app.run()
```

改进工厂

上面的工厂函数还不是足够好，可以改进的地方主要有以下几点:

1. 为了单元测试，要想办法传入配置，这样就不必在文件系统中创建配置文件。
2. 当设置应用时从蓝图调用一个函数，这样就可以有机会修改属性（如挂接请求前/后 处理器等）。
3. 如果有必要的话，当创建一个应用时增加一个 WSGI 中间件。

应用调度

应用调度是在 WSGI 层面组合多个 WSGI 应用的过程。可以组合多个 Flask 应用，也可以 组合 Flask 应用和其他 WSGI 应用。通过这种组合，如果有必要的话，甚至可以在同一个 解释器中一边运行 Django ，一边运行 Flask 。这种组合的好处取决于应用内部是如何 工作的。

应用调度与模块化的最大不同在于应用调度中的每个 应用是完全独立的，它们以各自的配置运行，并在 WSGI 层面被调度。

说明

下面所有的技术说明和举例都归结于一个可以运行于任何 WSGI 服务器的 application 对象。对于生产环境，参见 部署方式 。对于开发环境， Werkzeug 提供了一个内建开发服务器，它使用 `werkzeug.serving.run_simple()` 来运行:

```
from werkzeug.serving import run_simple
run_simple('localhost', 5000, application, use_reloader=True)
```

注意 run_simple 不能用于生产环境，生产环境服务器参见成熟的 WSGI 服务器 。

为了使用交互调试器，应用和简单服务器都应当处于调试模式。下面是一个简单的 “ hello world ”示例，使用了调试模式和 run_simple:

```
from flask import Flask
from werkzeug.serving import run_simple

app = Flask(__name__)
app.debug = True

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

if __name__ == '__main__':
    run_simple('localhost', 5000, app,
               use_reloader=True, use_debugger=True, use_evalex=True)
```

组合应用

如果你想在同一个 Python 解释器中运行多个独立的应用，那么你可以使用 werkzeug.wsgi.DispatcherMiddleware 。其原理是：每个独立的 Flask 应用都 是一个合法的 WSGI 应用，它们通过调度中间件组合为一个基于前缀调度的大应用。

假设你的主应用运行于 / ，后台接口位于 /backend:

```
from werkzeug.wsgi import DispatcherMiddleware
from frontend_app import application as frontend
from backend_app import application as backend

application = DispatcherMiddleware(frontend, {
    '/backend':     backend
})
```

根据子域调度

有时候你可能需要使用不同的配置来运行同一个应用的多个实例。可以把应用创建过程 放在一个函数中，这样调用这个函数就可以创建一个应用的实例，具体实现参见应用工厂方案。

最常见的做法是每个子域创建一个应用，配置服务器来调度所有子域的应用请求，使用 子域来创建用户自定义的实例。一旦你的服务器可以监听所有子域，那么就可以使用一个很简单的 WSGI 应用来动态创建应用了。

WSGI 层是完美的抽象层，因此可以写一个你自己的 WSGI 应用来监视请求，并把请求分配给你的 Flask 应用。如果被分配的应用还没有创建，那么就会动态创建应用并被登记下来:

```
from threading import Lock

class SubdomainDispatcher(object):

    def __init__(self, domain, create_app):
        self.domain = domain
        self.create_app = create_app
        self.lock = Lock()
        self.instances = {}

    def get_application(self, host):
        host = host.split(':')[0]
        assert host.endswith(self.domain), 'Configuration error'
        subdomain = host[:-len(self.domain)].rstrip('.')
```

```
        with self.lock:
            app = self.instances.get(subdomain)
            if app is None:
                app = self.create_app(subdomain)
                self.instances[subdomain] = app
            return app

    def __call__(self, environ, start_response):
        app = self.get_application(environ['HTTP_HOST'])
        return app(environ, start_response)
```

调度器示例:

```
from myapplication import create_app, get_user_for_subdomain
from werkzeug.exceptions import NotFound

def make_app(subdomain):
    user = get_user_for_subdomain(subdomain)
    if user is None:
        # 如果子域没有对应的用户，那么还是得返回一个 WSGI 应用
        # 用于处理请求。这里我们把 NotFound() 异常作为应用返回，
        # 它会被渲染为一个缺省的 404 页面。然后，可能还需要把
        # 用户重定向到主页。
        return NotFound()

    # 否则为特定用户创建应用
    return create_app(user)

application = SubdomainDispatcher('example.com', make_app)
```

根据路径调度

根据 URL 的路径调度非常简单。上面，我们通过查找 Host 头来判断子域，现在只要查找请求路径的第一个斜杠之前的路径就可以了:

```
from threading import Lock
from werkzeug.wsgi import pop_path_info, peek_path_info

class PathDispatcher(object):

    def __init__(self, default_app, create_app):
        self.default_app = default_app
        self.create_app = create_app
        self.lock = Lock()
```

```
        self.instances = {}

    def get_application(self, prefix):
        with self.lock:
            app = self.instances.get(prefix)
            if app is None:
                app = self.create_app(prefix)
                if app is not None:
                    self.instances[prefix] = app
            return app

    def __call__(self, environ, start_response):
        app = self.get_application(peek_path_info(environ))
        if app is not None:
            pop_path_info(environ)
        else:
            app = self.default_app
        return app(environ, start_response)
```

与根据子域调度相比最大的不同是：根据路径调度时，如果创建函数返回 None ，那么就会回落到另一个应用:

```
from myapplication import create_app, default_app, get_user_for_prefix

def make_app(prefix):
    user = get_user_for_prefix(prefix)
    if user is not None:
        return create_app(user)

application = PathDispatcher(default_app, make_app)
```

实现 API 异常处理

在 Flask 上经常会执行 RESTful API 。开发者首先会遇到的问题之一是用于 API 的内建异常处理不给力，回馈的内容不是很有用。

对于非法使用 API ，比使用 abort 更好的解决方式是实现你自己的异常处理类型， 并安装相应句柄，输出符合用户格式要求的出错信息。

简单的异常类

基本的思路是引入一个新的异常，回馈一个合适的可读性高的信息、一个状态码和一些可选的负载，给错误提供更多的环境内容。

以下是一个简单的示例:

```
from flask import jsonify

class InvalidUsage(Exception):
    status_code = 400

    def __init__(self, message, status_code=None, payload=None):
        Exception.__init__(self)
        self.message = message
        if status_code is not None:
            self.status_code = status_code
        self.payload = payload

    def to_dict(self):
        rv = dict(self.payload or ())
        rv['message'] = self.message
        return rv
```

这样一个视图就可以抛出带有出错信息的异常了。另外，还可以通过 payload 参数以字典的形式提供一些额外的负载。

注册一个错误处理句柄

现在，视图可以抛出异常，但是会立即引发一个内部服务错误。这是因为没有为这个错误处理类注册句柄。句柄增加很容易，例如:

```
@app.errorhandler(InvalidAPIUsage)
def handle_invalid_usage(error):
    response = jsonify(error.to_dict())
    response.status_code = error.status_code
    return response
```

在视图中的用法

以下是如何在视图中使用该功能:

```
@app.route('/foo')
def get_foo():
    raise InvalidUsage('This view is gone', status_code=410)
```

URL 处理器

New in version 0.7.

Flask 0.7 引入了 URL 处理器，其作用是为你处理大量包含相同部分的 URL 。假设你有 许多 URL 都包含语言代码，但是又不想在每个函数中都重复处理这个语言代码，那么就可可以使用 URL 处理器。

在与蓝图配合使用时， URL 处理器格外有用。下面我们分别演示在应用中和蓝图中使用 URL 处理器。

国际化应用的 URL

假设有应用如下:

```
from flask import Flask, g

app = Flask(__name__)

@app.route('/<lang_code>/')
def index(lang_code):
    g.lang_code = lang_code
    ...

@app.route('/<lang_code>/about')
def about(lang_code):
    g.lang_code = lang_code
    ...
```

上例中出现了大量的重复：必须在每一个函数中把语言代码赋值给 g 对象。当然，如果使用一个装饰器可以简化这个工作。但是，当你需要生成由一个函数指向另一个函数的 URL 时，还是得显式地提供语言代码，相当麻烦。

我们使用 url_defaults() 函数来简化这个问题。这个函数可以自动 把值注入到 url_for() 。以下代码检查在 URL 字典中是否存在语言代码， 端点是否需要一个名为 'lang_code' 的值:

```
@app.url_defaults
def add_language_code(endpoint, values):
    if 'lang_code' in values or not g.lang_code:
        return
    if app.url_map.is_endpoint_expecting(endpoint, 'lang_code'):
        values['lang_code'] = g.lang_code
```

URL 映射的 is_endpoint_expecting() 方法可用于检查端点是否需要提供一个语言代码。

上例的逆向函数是 url_value_preprocessor() 。这些函数在请求匹配后立即根据 URL 的值执行代码。它们可以从 URL 字典中取出值，并把取出的值放在 其他地方:

```
@app.url_value_preprocessor
def pull_lang_code(endpoint, values):
    g.lang_code = values.pop('lang_code', None)
```

这样就不必在每个函数中把 lang_code 赋值给 g 了。你还可以作 进一步改进：写一个装饰器把语言代码作为 URL 的前缀。但是更好的解决方式是使用 蓝图。一旦 'lang_code' 从值的字典中弹出，它就不再传送给视图函数了。精简后的代码如下:

```
from flask import Flask, g

app = Flask(__name__)

@app.url_defaults
def add_language_code(endpoint, values):
    if 'lang_code' in values or not g.lang_code:
        return
    if app.url_map.is_endpoint_expecting(endpoint, 'lang_code'):
        values['lang_code'] = g.lang_code

@app.url_value_preprocessor
def pull_lang_code(endpoint, values):
    g.lang_code = values.pop('lang_code', None)

@app.route('/<lang_code>/')
def index():
    ...

@app.route('/<lang_code>/about')
def about():
    ...
```

国际化的蓝图 URL

因为蓝图可以自动给所有 URL 加上一个统一的前缀，所以应用到每个函数就非常方便了。 更进一步，因为蓝图 URL 预处理器不需要检查 URL 是否真的需要要一个 'lang_code' 参数，所以可以去除 url_defaults())函数中的逻辑判断:

```
from flask import Blueprint, g

bp = Blueprint('frontend', __name__, url_prefix='/<lang_code>')

@bp.url_defaults
def add_language_code(endpoint, values):
    values.setdefault('lang_code', g.lang_code)

@bp.url_value_preprocessor
def pull_lang_code(endpoint, values):
    g.lang_code = values.pop('lang_code')

@bp.route('/')
def index():
    ...

@bp.route('/about')
def about():
    ...
```

使用 Distribute 部署

distribute 的前身是 setuptools ，它是一个扩展库，通常用于分发 Python 库和 扩展。它的英文名称的就是“分发”的意思。它扩展了 Python 自带的一个基础模块安装 系统 distutils ，支持多种更复杂的结构，方便了大型应用的分发部署。它的主要特色：

- 支持依赖： 一个库或者应用可以声明其所依赖的其他库的列表。依赖库将被自动 安装。
- 包注册： 可以在安装过程中注册包，这样就可以通过一个包查询其他包的信息。 这套系统最有名的功能是“切入点”，即一个包可以定义一个入口，以便于其他包挂接， 用以扩展包。
- 安装管理： distribute 中的 easy_install 可以为你安装其他库。你也可以使用早晚会替代 easy_install 的 pip ，它除了安装包还可以做更多的事。

Flask 本身，以及其他所有在 cheeseshop 中可以找到的库要么是用 distribute 分发的， 要么是用老的 setuptools 或 distutils 分发的。

在这里我们假设你的应用名称是 yourapplication.py ，且没有使用模块，而是一个[包href="http://dormousehole.readthedocs.org/en/latest/patterns/packages.html#larger-applications) 。 distribute 不支持分发标准模块，因此我们不 讨论模块的问题。关于如何把模块转换为包的信息参见大型应用方案。

使用 distribute 将使发布更复杂，也更加自动化。如果你想要完全自动化处理，请同时阅读使用 Fabric 部署一节。

基础设置脚本

因为你已经安装了 Flask ，所以你应当已经安装了 setuptools 或 distribute 。如果 没有安装，不用怕，有一个 distribute_setup.py 脚本可以帮助你安装。只要下载这个脚本，并在你的 Python 解释器中运行就可以了。

标准声明: 最好使用 virtualenv 。

你的设置代码应用放在 setup.py 文件中，这个文件应当位于应用旁边。这个文件名只是 一个约定，但是最好不要改变，因为大家都会去找这个文件。

是的，即使你使用 distribute ，你导入的包也是 setuptools 。 distribute 完全向后兼容于 setuptools ，因此它使用相同的导入名称。

Flask 应用的基础 setup.py 文件示例如下:

```
from setuptools import setup

setup(
    name='Your Application',
    version='1.0',
    long_description=__doc__,
    packages=['yourapplication'],
    include_package_data=True,
    zip_safe=False,
    install_requires=['Flask']
)
```

请记住，你必须显式的列出子包。如果你要 distribute 自动为你搜索包，你可以使用 find_packages 函数:

```
from setuptools import setup, find_packages

setup(
    ...
    packages=find_packages()
)
```

大多数 setup 的参数可以望文生义，但是 include_package_data 和 zip_safe 可以不容易理解。 include_package_data 告诉 distribute 要搜索一个 MANIFEST.in 文件，把文件内容所匹配的所有条目作为包数据安装。可以通过使用这个参数分发 Python 模块的静态文件和模板（参见分发资源 ）。 zip_safe 标志可用于强制或防止创建 zip 压缩包。通常你不会想要把包安装为 zip 压缩文件，因为一些工具不支持压缩文件，而且压缩文件比较难以调试。

分发资源

如果你尝试安装上文创建的包，你会发现诸如 static 或 templates 之类的文件夹没有被安装。原因是 distribute 不知道要为你添加哪些文件。你要做的是：在你的 setup.py 文件旁边创建一个 MANIFEST.in 文件。这个文件列出了所有应当添加到 tar 压缩包的文件:

```
recursive-include yourapplication/templates *
recursive-include yourapplication/static *
```

不要忘了把 setup 函数的 include_package_data 参数设置为 True！ 否则即使把内容在 MANIFEST.in 文件中全部列出来也没有用。

声明依赖

依赖是在 install_requires 参数中声明的，这个参数是一个列表。列表中的每一项都是一个需要在安装时从 PyPI 获得的包。缺省情况下，总是会获得最新版本的包，但你可以指定最高版本和最低版本。示例:

```
install_requires=[
    'Flask>=0.2',
    'SQLAlchemy>=0.6',
    'BrokenPackage>=0.7,<=1.0'
]
```

我前面提到，依赖包都从 PyPI 获得的。但是如果要从别的地方获得包怎么办呢? 你只要 还是按照上述方法写，然后提供一个可选地址列表就行了:

```
dependency_links=['http://example.com/yourfiles']
```

请确保页面上有一个目录列表，且页面上的链接指向正确的 tar 压缩包。这样 distribute 就会找到文件了。如果你的包在公司内部网络上，请提供指向服务器的 URL 。

安装 / 开发

要安装你的应用（理想情况下是安装到一个 virtualenv ），只要运行带 install 参数的 setup.py 脚本就可以了。它会将你的应用安装到 virtualenv 的 site-packages 文件夹下，同时下载并安装依赖:

```
$ python setup.py install
```

如果你正开发这个包，同时也希望相关依赖被安装，那么可以使用 develop 来代替:

```
$ python setup.py develop
```

这样做的好处是只安装一个指向 site-packages 的连接，而不是把数据复制到那里。这样在开发过程中就不必每次修改以后再运行 install 了。

使用 Fabric 部署

Fabric 是一个 Python 工具，与 Makefiles 类似，但是能够在远程服务器上执行命令。如果与适当的 Python 包（ 大型应用 ）与优良的配置（ 配置管理 ）相结合那么 Fabric 将是在外部服务器上部署 Flask 的利器。

在下文开始之前，有几点需要明确：

- Fabric 1.0 需要要被安装到本地。本教程假设使用的是最新版本的 Fabric 。
- 应用已经是一个包，且有一个可用的 setup.py 文件（ 使用 Distribute 部署 ）。
- 在下面的例子中，我们假设远程服务器使用 mod_wsgi 。当然，你可以使用你自己 喜欢的服务器，但是在示例中我们选择 Apache + mod_wsgi ，因为它们设置方便， 且在没有 root 权限情况下可以方便的重载应用。

创建第一个 Fabfile

fabfile 是控制 Fabric 的东西，其文件名为 fabfile.py ，由 fab 命令执行。在这个文件中定义的所有函数都会被视作 fab 子命令。这些命令将会在一个或多个主机上运行。这些主机可以在 fabfile 中定义，也可以在命令行中定义。本例将在 fabfile 中 定义主机。

下面是第一个例子，比较级。它可以把当前的源代码上传至服务器，并安装到一个预先存在的 virtual 环境：

```
from fabric.api import *

 # 使用远程命令的用户名

env.user = 'appuser'
 # 执行命令的服务器

env.hosts = ['server1.example.com', 'server2.example.com']

def pack():
    # 创建一个新的分发源，格式为 tar 压缩包
    local('python setup.py sdist --formats=gztar', capture=False)

def deploy():
    # 定义分发版本的名称和版本号
    dist = local('python setup.py --fullname', capture=True).strip()
    # 把 tar 压缩包格式的源代码上传到服务器的临时文件夹
```

```
    put('dist/%s.tar.gz' % dist, '/tmp/yourapplication.tar.gz')
    # 创建一个用于解压缩的文件夹，并进入该文件夹
    run('mkdir /tmp/yourapplication')
    with cd('/tmp/yourapplication'):
        run('tar xzf /tmp/yourapplication.tar.gz')
        # 现在使用 virtual 环境的 Python 解释器来安装包
        run('/var/www/yourapplication/env/bin/python setup.py install')
    # 安装完成，删除文件夹
    run('rm -rf /tmp/yourapplication /tmp/yourapplication.tar.gz')
    # 最后 touch .wsgi 文件，让 mod_wsgi 触发应用重载
    run('touch /var/www/yourapplication.wsgi')
```

上例中的注释详细，应当是容易理解的。以下是 fabric 提供的最常用命令的简要说明：

- run – 在远程服务器上执行一个命令
- local – 在本地机器上执行一个命令
- put – 上传文件到远程服务器上
- cd – 在服务器端改变目录。必须与 with 语句联合使用。

运行 Fabfile

那么如何运行 fabfile 呢？答案是使用 fab 命令。要在远程服务器上部署当前版本的代码可以使用这个命令：

```
$ fab pack deploy
```

但是这个命令需要远程服务器上已经创建了 /var/www/yourapplication 文件夹，且 /var/www/yourapplication/env 是一个 virtual 环境。更进一步，服务器上还没有创建配置文件和 .wsgi 文件。那么，我们如何在一个新的服务器上创建一个基础环境呢？

这个问题取决于你要设置多少台服务器。如果只有一台应用服务器（多数情况下），那么在 fabfile 中创建命令有一点多余。当然，你可以这么做。这个命令可以称之为 setup 或 bootstrap 。在使用命令时显式传递服务器名称：

```
$ fab -H newserver.example.com bootstrap
```

设置一个新服务器大致有以下几个步骤：

1. 在 /var/www 创建目录结构：

```
$ mkdir /var/www/yourapplication
$ cd /var/www/yourapplication
$ virtualenv --distribute env
```

1. 上传一个新的 application.wsgi 文件和应用配置文件（如 application.cfg ）到服务器上。
2. 创建一个新的用于 yourapplication 的 Apache 配置并激活它。要确保激活 .wsgi 文件变动监视，这样在 touch 的时候可以自动重载应用。（ 更多信息参见 mod_wsgi (Apache) ）

现在的问题是： application.wsgi 和 application.cfg 文件从哪里来?

WSGI 文件

WSGI 文件必须导入应用，并且还必须设置一个环境变量用于告诉应用到哪里去搜索配置。 示例:

```
import os
os.environ['YOURAPPLICATION_CONFIG'] = '/var/www/yourapplication/application.cfg'
from yourapplication import app
```

应用本身必须像下面这样初始化自己才会根据环境变量搜索配置:

```
app = Flask(__name__)
app.config.from_object('yourapplication.default_config')
app.config.from_envvar('YOURAPPLICATION_CONFIG')
```

这个方法在配置管理一节已作了详细的介绍。

配置文件

上文已谈到，应用会根据 YOURAPPLICATION_CONFIG 环境变量找到正确的配置文件。 因此我们应当把配置文件放在应用可以找到的地方。在不同的电脑上配置文件是不同的， 所以一般我们不对配置文件作版本处理。

一个流行的方法是在一个独立的版本控制仓库为不同的服务器保存不同的配置文件，然后 在所有服务器进行检出。然后在需要的地方使用配置文件的符号链接（例如： /var/www/yourapplication ）。

不管如何，我们这里只有一到两台服务器，因此我们可以预先手动上传配置文件。

第一次部署

现在我们可以进行第一次部署了。我已经设置好了服务器，因此服务器上应当已经有了 virtual 环境和已激活的 apache 配置。现在我们可以打包应用并部署它了:

```
$ fab pack deploy
```

Fabric 现在会连接所有服务器并运行 fabfile 中的所有命令。首先它会打包应用得到一个 tar 压缩包。然后会执行分发，把源代码上传到所有服务器并安装。感谢 setup.py 文件，所需要的依赖库会自动安装到 virtual 环境。

下一步

在前文的基础上，还有更多的方法可以全部署工作更加轻松：

- 创建一个初始化新服务器的 bootstrap 命令。它可以初始化一个新的 virtual 环境、正确设置 apache 等等。
- 把配置文件放入一个独立的版本库中，把活动配置的符号链接放在适当的地方。
- 还可以把应用代码放在一个版本库中，在服务器上检出最新版本后安装。这样你可以方便的回滚到老版本。
- 挂接测试功能，方便部署到外部服务器进行测试。 使用 Fabric 是一件有趣的事情。你会发现在电脑上打出 fab deploy 是非常神奇的。 你可以看到你的应用被部署到一个又一个服务器上。

在 Flask 中使用 SQLite 3

在 Flask 中，你可以方便的按需打开数据库连接，并且在环境解散时关闭这个连接（ 通常是请求结束的时候 ）。

以下是一个在 Flask 中使用 SQLite 3 的例子:

```
import sqlite3
from flask import g

DATABASE = '/path/to/database.db'

def get_db():
    db = getattr(g, '_database', None)
    if db is None:
        db = g._database = connect_to_database()
    return db

@app.teardown_appcontext
def close_connection(exception):
    db = getattr(g, '_database', None)
    if db is not None:
        db.close()
```

为了使用数据库，所有应用都必须准备好一个处于激活状态的环境。使用 get_db 函数可以得到数据库连接。当环境解散时，数据库连接会被切断。

注意：如果你使用的是 Flask 0.9 或者以前的版本，那么你必须使用 flask._app_ctx_stack.top ，而不是 g 。因为 flask.g 对象是绑定到请求的，而不是应用环境。

示例:

```
@app.route('/')
def index():
    cur = get_db().cursor()
    ...
```

> Note 请记住，解散请求和应用环境的函数是一定会被执行的。即使请求前处理器执行失败或根本没有执行，解散函数也会被执行。因此，我们必须保证在关闭数据库连接之前 数据库连接是存在的。

按需连接

上述方式（在第一次使用时连接数据库）的优点是只有在真正需要时才打开数据库连接。 如果你想要在一个请求环境之外使用数据库连接，那么你可以手动在 Python 解释器打开应用环境:

```
with app.app_context():
    # now you can use get_db()
```

简化查询

现在，在每个请求处理函数中可以通过访问 g.db 来得到当前打开的数据库连接。为了简化 SQLite 的使用，这里有一个有用的行工厂函数。该函数会转换每次从数据库返回的 结果。例如，为了得到字典类型而不是元组类型的返回结果，可以这样:

```
def make_dicts(cursor, row):
    return dict((cur.description[idx][0], value)
                for idx, value in enumerate(row))

db.row_factory = make_dicts
```

或者更简单的:

```
db.row_factory = sqlite3.Row
```

此外，把得到游标，执行查询和获得结果组合成一个查询函数不失为一个好办法:

```
def query_db(query, args=(), one=False):
    cur = get_db().execute(query, args)
    rv = cur.fetchall()
    cur.close()
    return (rv[0] if rv else None) if one else rv
```

上述的方便的小函数与行工厂联合使用与使用原始的数据库游标和连接相比要方便多了。

可以这样使用上述函数:

```
for user in query_db('select * from users'):
    print user['username'], 'has the id', user['user_id']
```

只需要得到单一结果的用法:

```
user = query_db('select * from users where username = ?',
                [the_username], one=True)
if user is None:
    print 'No such user'
else:
    print the_username, 'has the id', user['user_id']
```

如果要给 SQL 语句传递参数，请在语句中使用问号来代替参数，并把参数放在一个列表中 一起传递。不要用字符串格式化的方式直接把参数加入 SQL 语句中，这样会给应用带来 SQL 注入的风险。

初始化模式

关系数据库是需要模式的，因此一个应用常常需要一个 schema.sql 文件来创建 数据库。因此我们需要使用一个函数来其于模式创建数据库。下面这个函数可以完成这个任务:

```
def init_db():
    with app.app_context():
        db = get_db()
        with app.open_resource('schema.sql', mode='r') as f:
            db.cursor().executescript(f.read())
        db.commit()
```

可以使用上述函数在 Python 解释器中创建数据库:

```
>>> from yourapplication import init_db
>>> init_db()
```

在 Flask 中使用 SQLAlchemy

许多人喜欢使用 [SQLAlchemyhref="http://www.sqlalchemy.org/) 来访问数据库。建议在你的 Flask 应用中使用包来代替 模块，并把模型放入一个独立的模块中（参见大型应用 ）。虽然这不是必须的，但是很有用。

有四种 SQLAlchemy 的常用方法，下面一一道来：

Flask-SQLAlchemy 扩展

因为 SQLAlchemy 是一个常用的数据库抽象层，并且需要一定的配置才能使用，因此我们为你做了一个处理 SQLAlchemy 的扩展。如果你需要快速的开始使用 SQLAlchemy ，那么推荐你使用这个扩展。

你可以从 PyPI 下载 Flask-SQLAlchemy 。

声明

SQLAlchemy 中的声明扩展是使用 SQLAlchemy 的最新方法，它允许你像 Django 一样， 在一个地方定义表和模型然后到处使用。除了以下内容，我建议你阅读声明的官方文档。

以下是示例 database.py 模块:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import scoped_session, sessionmaker
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine('sqlite:////tmp/test.db', convert_unicode=True)
db_session = scoped_session(sessionmaker(autocommit=False,
                                         autoflush=False,
                                         bind=engine))
Base = declarative_base()
Base.query = db_session.query_property()

def init_db():
    # 在这里导入定义模型所需要的所有模块，这样它们就会正确的注册在
    # 元数据上。否则你就必须在调用 init_db() 之前导入它们。
    import yourapplication.models
    Base.metadata.create_all(bind=engine)
```

要定义模型的话，只要继承上面创建的 Base 类就可以了。你可能会奇怪这里为什么不用理会线程（就像我们在 SQLite3 的例子中一样使用 g 对象）。 原因是 SQLAlchemy 已经用 scoped_session 为我们做好了此 类工作。

如果要在应用中以声明方式使用 SQLAlchemy ，那么只要把下列代码加入应用模块就可以了。 Flask 会自动在请求结束时或者应用关闭时删除数据库会话:

```
from yourapplication.database import db_session

@app.teardown_appcontext
def shutdown_session(exception=None):
    db_session.remove()
```

以下是一个示例模型（放入 models.py 中）:

```
from sqlalchemy import Column, Integer, String
from yourapplication.database import Base

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String(50), unique=True)
    email = Column(String(120), unique=True)

    def __init__(self, name=None, email=None):
        self.name = name
        self.email = email

    def __repr__(self):
        return '<User %r>' % (self.name)
```

可以使用 init_db 函数来创建数据库:

```
>>> from yourapplication.database import init_db
>>> init_db()
```

在数据库中插入条目示例:

```
>>> from yourapplication.database import db_session
>>> from yourapplication.models import User
>>> u = User('admin', 'admin@localhost')
>>> db_session.add(u)
>>> db_session.commit()
```

查询很简单:

```
>>> User.query.all()
[<User u'admin'>]
>>> User.query.filter(User.name == 'admin').first()
<User u'admin'>
```

人工对象关系映射

人工对象关系映射相较于上面的声明方式有优点也有缺点。主要区别是人工对象关系映射 分别定义表和类并映射它们。这种方式更灵活，但是要多些代码。通常，这种方式与声明 方式一样运行，因此请确保把你的应用在包中分为多个模块。

示例 database.py 模块:

```
from sqlalchemy import create_engine, MetaData
from sqlalchemy.orm import scoped_session, sessionmaker

engine = create_engine('sqlite:////tmp/test.db', convert_unicode=True)
metadata = MetaData()
db_session = scoped_session(sessionmaker(autocommit=False,
                                         autoflush=False,
                                         bind=engine))
def init_db():
    metadata.create_all(bind=engine)
```

就像声明方法一样，你需要在请求后或者应用环境解散后关闭会话。把以下代码放入你的应用模块:

```
from yourapplication.database import db_session

@app.teardown_appcontext
def shutdown_session(exception=None):
    db_session.remove()
```

以下是一个示例表和模型（放入 models.py 中）:

```
from sqlalchemy import Table, Column, Integer, String
from sqlalchemy.orm import mapper
from yourapplication.database import metadata, db_session

class User(object):
    query = db_session.query_property()

    def __init__(self, name=None, email=None):
        self.name = name
        self.email = email
```

```
    def __repr__(self):
        return '<User %r>' % (self.name)

users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String(50), unique=True),
    Column('email', String(120), unique=True)
)
mapper(User, users)
```

查询和插入与声明方式的一样。

SQL 抽象层

如果你只需要使用数据库系统（和 SQL ）抽象层，那么基本上只要使用引擎:

```
from sqlalchemy import create_engine, MetaData

engine = create_engine('sqlite:////tmp/test.db', convert_unicode=True)
metadata = MetaData(bind=engine)
```

然后你要么像前文中一样在代码中声明表，要么自动载入它们:

```
users = Table('users', metadata, autoload=True)
```

可以使用 insert 方法插入数据。为了使用事务，我们必须先得到一个连接:

```
>>> con = engine.connect()
>>> con.execute(users.insert(), name='admin', email='admin@localhost')
```

SQLAlchemy 会自动提交。

可以直接使用引擎或连接来查询数据库:

```
>>> users.select(users.c.id == 1).execute().first()
(1, u'admin', u'admin@localhost')
```

查询结果也是类字典元组:

```
>>> r = users.select(users.c.id == 1).execute().first()
>>> r['name']
u'admin'
```

你也可以把 SQL 语句作为字符串传递给 execute() 方法:

```
>>> engine.execute('select * from users where id = :1', [1]).first()
(1, u'admin', u'admin@localhost')
```

关于 SQLAlchemy 的更多信息请移步其官方网站 。

上传文件

是的，这里要谈的是一个老问题：文件上传。文件上传的基本原理实际上很简单，基本上是：

1. 一个带有 enctype=multipart/form-data 的

<

form> 标记，标记中含有一个 Br ow s e... 。2. 应用通过请求对象的 files 字典来访问文件。 3. 使用文件的 save() 方法把文件永久 地保存在文件系统中。

简介

让我们从一个基本的应用开始，这个应用上传文件到一个指定目录，并把文件显示给用户。 以下是应用的引导代码:

```
import os
from flask import Flask, request, redirect, url_for
from werkzeug import secure_filename

UPLOAD_FOLDER = '/path/to/the/uploads'
ALLOWED_EXTENSIONS = set(['txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif'])

app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
```

首先我们导入了一堆东西，大多数是浅显易懂的。werkzeug.secure_filename() 会在稍后解释。UPLOAD_FOLDER 是上传文件要储存的目录，ALLOWED_EXTENSIONS 是允许上传的文件扩展名的集合。接着我们给应用手动添加了一个 URL 规则。一般现在不会做这个，但是为什么这么做了呢? 原因是我们需要服务器（或我们的开发服务器）为我们提供 服务。因此我们只生成这些文件的 URL 的规则。

为什么要限制文件件的扩展名呢? 如果直接向客户端发送数据，那么你可能不会想让用户上传任意文件。否则，你必须确保用户不能上传 HTML 文件，因为 HTML 可能引起 XSS 问题（参见跨站脚本攻击（XSS））。如果服务器可以执行 PHP 文件，那么还必须确保不允许上传 .php 文件。但是谁又会在服务器上安装 PHP 呢，对不? :)

下一个函数检查扩展名是否合法，上传文件，把用户重定向到已上传文件的 URL:

```
def allowed_file(filename):
    return '.' in filename and \
```

```
        filename.rsplit('.', 1)[1] in ALLOWED_EXTENSIONS

@app.route('/', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        file = request.files['file']
        if file and allowed_file(file.filename):
            filename = secure_filename(file.filename)
            file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
            return redirect(url_for('uploaded_file',
                                    filename=filename))
    return '''
    <!doctype html>
    <title>Upload new File</title>
    <h1>Upload new File</h1>
    <form action="" method=post enctype=multipart/form-data>
      <p><input type=file name=file>
         <input type=submit value=Upload>
    </form>
    '''
```

那么 secure_filename() 函数到底是有什么用？有一条原则是“ 永远不要信任用户输入”。这条原则同样适用于已上传文件的文件名。所有提交的表单数据 可能是伪造的，文件名也可以是危险的。此时要谨记：在把文件保存到文件系统之前总是要 使用这个函数对文件名进行安检。

进一步说明

你可以会好奇 secure_filename() 做了哪些工作，如果不使用 它会有什么后果。假设有人把下面的信息作为 filename 传递给你的应用:

```
filename = "../../../../home/username/.bashrc"
```

假设 ../ 的个数是正确的，你会把它和 UPLOAD_FOLDER 结合在一起，那么用户 就可能有能力修改一个服务器上的文件，这个文件本来是用户无权修改的。这需要了解 应用是如何运行的，但是请相信我，黑客都是病态的 :)

现在来看看函数是如何工作的:

```
>>> secure_filename('../../../../home/username/.bashrc')
'home_username_.bashrc'
```

现在还剩下一件事：为已上传的文件提供服务。 Flask 0.5 版本开始我们可以使用一个 函数来完成这个任务:

```
from flask import send_from_directory

@app.route('/uploads/<filename>')
def uploaded_file(filename):
    return send_from_directory(app.config['UPLOAD_FOLDER'],
                               filename)
```

另外，可以把 uploaded_file 注册为 build_only 规则，并使用 SharedDataMiddleware 。这种方式可以在 Flask 老版本中 使用:

```
from werkzeug import SharedDataMiddleware
app.add_url_rule('/uploads/<filename>', 'uploaded_file',
                 build_only=True)
app.wsgi_app = SharedDataMiddleware(app.wsgi_app, {
    '/uploads':  app.config['UPLOAD_FOLDER']
})
```

如果你现在运行应用，那么应该一切都应该按预期正常工作。

改进上传

New in version 0.6.

Flask 到底是如何处理文件上传的呢? 如果上传的文件很小，那么会把它们储存在内存中。 否则就会把它们保存到一个临时的位置（ 通过 tempfile.gettempdir() 可以得到 这个位置 ）。但是，如何限制上传文件的尺寸呢? 缺省情况下， Flask 是不限制上传文件 的尺寸的。可以通过设置配置的 MAX_CONTENT_LENGTH 来限制文件尺寸:

```
from flask import Flask, Request

app = Flask(__name__)
app.config['MAX_CONTENT_LENGTH'] = 16 * 1024 * 1024
```

上面的代码会把尺寸限制为 16 M 。如果上传了大于这个尺寸的文件， Flask 会抛出一个 RequestEntityTooLarge 异常。

Flask 0.6 版本中添加了这个功能。但是通过继承请求对象，在较老的版本中也可以实现这个功能。更多信息请参阅 Werkzeug 关于文件处理的文档。

上传进度条

在不久以前，许多开发者是这样实现上传进度条的：分块读取上传的文件，在数据库中储存上传的进度，然后在客户端通过 JavaScript 获取进度。简而言之，客户端每 5 秒钟向服务器询问一次上传进度。觉得讽刺吗？客户端在明知故问。

现在有了更好的解决方案，更快且更可靠。网络发生了很大变化，你可以在客户端使用 HTML5 、 JAVA 、 Silverlight 或 Flash 获得更好的上传体验。请查看以下库，学习一些优秀的上传的示例：

- Plupload – HTML5, Java, Flash
- SWFUpload – Flash
- JumpLoader – Java

一个更简便的方案

因为所有应用中上传文件的方案基本相同，因此可以使用 Flask-Uploads 扩展来实现 文件上传。这个扩展实现了完整的上传机制，还具有白名单功能、黑名单功能以及其他 功能。

缓存

当你的应用变慢的时候，可以考虑加入缓存。至少这是最简单的加速方法。缓存有什么用？ 假设有一个函数耗时较长，但是这个函数在五分钟前返回的结果还是正确的。那么我们就 可以考虑把这个函数的结果在缓存中存放一段时间。

Flask 本身不提供缓存，但是它的基础库之一 Werkzeug 有一些非常基本的缓存支持。它支持多种缓存后端，通常你需要使用的是 memcached 服务器。

设置一个缓存

与创建 Flask 对象类似，你需要创建一个缓存对象并保持它。如果你 正在使用开发服务器，那么你可以创建一个 SimpleCache 对象。这个对象提供简单的缓存，它把 缓存项目保存在 Python 解释器的内存中:

```
from werkzeug.contrib.cache import SimpleCache
cache = SimpleCache()
```

如果你要使用 memcached ，那么请确保有 memcache 模块支持（你可以从 PyPI 获得模块）和一个正在运行的 memcached 服务器。 连接 memcached 服务器示例:

```
from werkzeug.contrib.cache import MemcachedCache
cache = MemcachedCache(['127.0.0.1:11211'])
```

如果你正在使用 App Engine ，那么你可以方便地连接到 App Engine memcache 服务器:

```
from werkzeug.contrib.cache import GAEMemcachedCache
cache = GAEMemcachedCache()
```

使用缓存

现在的问题是如何使用缓存呢？有两个非常重要的操作： get() 和 set() 。下面展示如何使用它们:

get() 用于从缓存中获得项目，调用时使用 一个字符串作为键名。如果项目存在，那么就会返回这个项目，否则返回 None:

```
rv = cache.get('my-item')
```

set() 用于把项目添加到缓存中。第一个参数是键名，第二个参数是键值。还可以提供一个超时参数，当超过时间后项目会自动删除。

下面是一个完整的例子:

```
def get_my_item():
    rv = cache.get('my-item')
    if rv is None:
        rv = calculate_value()
        cache.set('my-item', rv, timeout=5 * 60)
    return rv
```

视图装饰器

Python 有一个非常有趣的功能：函数装饰器。这个功能可以使网络应用干净整洁。 Flask 中的每个视图都是一个装饰器，它可以被注入额外的功能。你可以已经用过了 route() 装饰器。但是，你有可能需要使用你自己的装饰器。假设有 一个视图，只有已经登录的用户才能使用。如果用户访问时没有登录，则会被重定向到登录页面。这种情况下就是使用装饰器的绝佳机会。

检查登录装饰器

让我们来实现这个装饰器。装饰器是一个返回函数的函数。听上去复杂，其实很简单。只要记住一件事：装饰器用于更新函数的 `__name__` 、 `__module__` 和其他属性。这一点很容易忘记，但是你不必人工更新函数属性，可以使用一个类似于装饰器的函数（ functools.wraps() ）。

下面是检查登录装饰器的例子。假设登录页面为 'login' ，当前用户被储存在 g.user 中，如果还没有登录，其值为 None:

```
from functools import wraps
from flask import g, request, redirect, url_for

def login_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if g.user is None:
            return redirect(url_for('login', next=request.url))
        return f(*args, **kwargs)
    return decorated_function
```

如何使用这个装饰器呢？把这个装饰器放在最靠近函数的地方就行了。当使用更进一步的装饰器时，请记住要把 route() 装饰器放在最外面:

```
@app.route('/secret_page')
@login_required
def secret_page():
    pass
```

缓存装饰器

假设有一个视图函数需要消耗昂贵的计算成本，因此你需要在一定时间内缓存这个视图的 计算结果。这种情况下装饰器是一个好的选择。我们假设你像缓存方案中一样设置了缓存。

下面是一个示例缓存函数。它根据一个特定的前缀（实际上是一个格式字符串）和请求的 当前路径生成缓存键。注意，我们先使用了一个函数来创建装饰器，这个装饰器用于装饰 函数。听起来拗口吧，确实有一点复杂，但是下面的示例代码还是很容易读懂的。

被装饰代码按如下步骤工作

1. 基于基础路径获得当前请求的唯一缓存键。
2. 从缓存中获取键值。如果获取成功则返回获取到的值。
3. 否则调用原来的函数，并把返回值存放在缓存中，直至过期（缺省值为五分钟）。

代码:

```
from functools import wraps
from flask import request

def cached(timeout=5 * 60, key='view/%s'):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            cache_key = key % request.path
            rv = cache.get(cache_key)
            if rv is not None:
                return rv
            rv = f(*args, **kwargs)
            cache.set(cache_key, rv, timeout=timeout)
            return rv
        return decorated_function
    return decorator
```

注意，以上代码假设存在一个可用的实例化的 cache 对象，更多信息参见缓存方案。

模板装饰器

不久前， TurboGear 的人发明了模板装饰器这个通用模式。其工作原理是返回一个字典， 这个字典包含从视图传递给模板的值，模板自动被渲染。以下三个例子的功能是相同的:

```
@app.route('/')
def index():
    return render_template('index.html', value=42)

@app.route('/')
@templated('index.html')
def index():
    return dict(value=42)

@app.route('/')
@templated()
def index():
    return dict(value=42)
```

正如你所见，如果没有提供模板名称，那么就会使用 URL 映射的端点（把点转换为斜杠）加上 '.html' 。如果提供了，那么就会使用所提供的模板名称。当装饰器函数返回 时，返回的字典就被传送到模板渲染函数。如果返回的是 None ，就会使用空字典。如果 返回的不是字典，那么就会直接传递原封不动的返回值。这样就可以仍然使用重定向函数或 返回简单的字符串。

以下是装饰器的代码:

```
from functools import wraps
from flask import request

def templated(template=None):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            template_name = template
            if template_name is None:
                template_name = request.endpoint \
                    .replace('.', '/') + '.html'
            ctx = f(*args, **kwargs)
            if ctx is None:
                ctx = {}
            elif not isinstance(ctx, dict):
                return ctx
            return render_template(template_name, **ctx)
        return decorated_function
    return decorator
```

端点装饰器

当你想要使用 werkzeug 路由系统，以便于获得更强的灵活性时，需要和 Rule 中定义的一样，把端点映射到视图函数。这样就需要 用的装饰器了。例如:

```
from flask import Flask
from werkzeug.routing import Rule

app = Flask(__name__)
app.url_map.add(Rule('/', endpoint='index'))

@app.endpoint('index')
def my_index():
    return "Hello world"
```

使用 WTForms 进行表单验证

当你必须处理浏览器提交的表单数据时，视图代码很快会变得难以阅读。有一些库可以简化这个工作，其中之一便是 WTForms ，下面我们将介绍这个库。如果你必须处理许多表单，那么应当尝试使用这个库。

如果要使用 WTForms ，那么首先要把表单定义为类。我推荐把应用分割为多个模块（ 大型应用 ），并为表单添加一个独立的模块。

使用一个扩展获得大部分 WTForms 的功能

Flask-WTF 扩展可以实现本方案的所有功能，并且还提供一些小的辅助工具。使用这个扩展可以更好的使用表单和 Flask 。你可以从 PyPI 获得这个扩展。

表单

下面是一个典型的注册页面的示例:

```
from wtforms import Form, BooleanField, TextField, PasswordField, validators

class RegistrationForm(Form):
    username = TextField('Username', [validators.Length(min=4, max=25)])
    email = TextField('Email Address', [validators.Length(min=6, max=35)])
    password = PasswordField('New Password', [
        validators.Required(),
        validators.EqualTo('confirm', message='Passwords must match')
    ])
    confirm = PasswordField('Repeat Password')
    accept_tos = BooleanField('I accept the TOS', [validators.Required()])
```

视图

在视图函数中，表单用法示例如下:

```
@app.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm(request.form)
    if request.method == 'POST' and form.validate():
        user = User(form.username.data, form.email.data,
```

```
            form.password.data)
        db_session.add(user)
        flash('Thanks for registering')
        return redirect(url_for('login'))
    return render_template('register.html', form=form)
```

注意，这里我们默认视图使用了 SQLAlchemy（在 Flask 中使用 SQLAlchemy），当然这不是必须的。请根据你的实际情况修改代码。

请记住以下几点：

1. 如果数据是通过 HTTP POST 方法提交的，请根据 form 的值创建表单。如果是通过 GET 方法提交的，则相应的是 args 。
2. 调用 validate() 函数来验证数据。如果验证通过，则函数返回 True ，否则返回 False 。
3. 通过 `form.<NAME>.data` 可以访问表单中单个值。

模板中的表单

现在我们来看看模板。把表单传递给模板后就可以轻松渲染它们了。看一看下面的示例模板就可以知道有多轻松了。 WTForms 替我们完成了一半表单生成工作。为了做得更好， 我们可以写一个宏，通过这个宏渲染带有一个标签的字段和错误列表（如果有的话）。

以下是一个使用宏的示例 _formhelpers.html 模板：

```
{% macro render_field(field) %}
  <dt>{{ field.label }}
  <dd>{{ field(**kwargs)|safe }}
  {% if field.errors %}
    <ul class=errors>
    {% for error in field.errors %}
      <li>{{ error }}</li>
    {% endfor %}
    </ul>
  {% endif %}
  </dd>
{% endmacro %}
```

上例中的宏接受一堆传递给 WTForm 字段函数的参数，为我们渲染字段。参数会作为 HTML 属性插入。例如你可以调用 render_field(form.username, class='username') 来 为输入元素添加一个类。注意： WTForms 返回标准的 Python unicode 字符串，因此我们必须使用 |safe 过滤器告诉 Jinja2 这些数据已经经过 HTML 转义了。

以下是使用了上面的 _formhelpers.html 的 register.html 模板：

```
{% from "_formhelpers.html" import render_field %}
<form method=post action="/register">
  <dl>
    {{ render_field(form.username) }}
    {{ render_field(form.email) }}
    {{ render_field(form.password) }}
    {{ render_field(form.confirm) }}
    {{ render_field(form.accept_tos) }}
  </dl>
  <p><input type=submit value=Register>
</form>
```

更多关于 WTForms 的信息请移步 WTForms 官方网站 。

模板继承

Jinja 最有力的部分就是模板继承。模板继承允许你创建一个基础“骨架”模板。这个模板中包含站点的常用元素，定义可以被子模板继承的 块 。

听起来很复杂其实做起来简单，看看下面的例子就容易理解了。

基础模板

这个模板的名称是 layout.html ，它定义了一个简单的 HTML 骨架，用于显示一个简单的两栏页面。“子”模板的任务是用内容填充空的块：

```
<!doctype html>
<html>
  <head>
    {% block head %}
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
    <title>{% block title %}{% endblock %} - My Webpage</title>
    {% endblock %}
  </head>
  <body>
    <div id="content">{% block content %}{% endblock %}</div>
    <div id="footer">
      {% block footer %}
      &copy; Copyright 2010 by <a href="http://domain.invalid/">you</a>.
      {% endblock %}
    </div>
  </body>
</html>
```

在这个例子中， {% block %} 标记定义了四个可以被子模板填充的块。 block 标记告诉模板引擎这是一个可以被子模板重载的部分。

子模板

子模板示例：

```
{% extends "layout.html" %}
{% block title %}Index{% endblock %}
```

```
{% block head %}
  {{ super() }}
  <style type="text/css">
    .important { color: #336699; }
  </style>
{% endblock %}
{% block content %}
  <h1>Index</h1>
  <p class="important">
    Welcome on my awesome homepage.
{% endblock %}
```

这里 {% extends %} 标记是关键，它告诉模板引擎这个模板“扩展”了另一个模板， 当模板系统评估这个模板时会先找到父模板。这个扩展标记必须是模板中的第一个标记。 如果要使用父模板中的块内容，请使用 {{ super() }} 。

消息闪现

一个好的应用和用户界面都需要良好的反馈。如果用户得不到足够的反馈，那么应用最终会被用户唾弃。 Flask 的闪现系统提供了一个良好的反馈方式。闪现系统的基本工作方式 是：在且只在下一个请求中访问上一个请求结束时记录的消息。一般我们结合布局模板来使用闪现系统。

简单的例子

以下是一个完整的示例:

```
from flask import Flask, flash, redirect, render_template, \
     request, url_for

app = Flask(__name__)
app.secret_key = 'some_secret'

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

@app.route('/login', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] != 'admin' or \
            request.form['password'] != 'secret':
            error = 'Invalid credentials'
        else:
            flash('You were successfully logged in')
            return redirect(url_for('index'))
    return render_template('login.html', error=error)

if __name__ == "__main__":
    app.run()
```

以下是实现闪现的 layout.html 模板:

```
<!doctype html>
<title>My Application</title>
{% with messages = get_flashed_messages() %}
  {% if messages %}
```

```
    <ul class=flashes>
    {% for message in messages %}
      <li>{{ message }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
{% block body %}{% endblock %}
```

以下是 index.html 模板：

```
{% extends "layout.html" %}
{% block body %}
  <h1>Overview</h1>
  <p>Do you want to <a href="{{ url_for('login') }}">log in?</a>
{% endblock %}
```

login 模板：

```
{% extends "layout.html" %}
{% block body %}
  <h1>Login</h1>
  {% if error %}
    <p class=error><strong>Error:</strong> {{ error }}
  {% endif %}
  <form action="" method=post>
    <dl>
      <dt>Username:
      <dd><input type=text name=username value="{{
          request.form.username }}">
      <dt>Password:
      <dd><input type=password name=password>
    </dl>
    <p><input type=submit value=Login>
  </form>
{% endblock %}
```

闪现消息的类别

New in version 0.3.

闪现消息还可以指定类别，如果没有指定，那么缺省的类别为 'message' 。不同的 类别可以给用户提供更好的反馈。例如错误消息可以使用红色背景。

使用 flash() 函数可以指定消息的类别:

```
flash(u'Invalid password provided', 'error')
```

模板中的 get_flashed_messages() 函数也应当返回类别，显示消息的循环也要略作改变:

```
{% with messages = get_flashed_messages(with_categories=true) %}
  {% if messages %}
    <ul class=flashes>
    {% for category, message in messages %}
      <li class="{{ category }}">{{ message }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
```

上例展示如何根据类别渲染消息，还可以给消息加上前缀，如 Error: 。

过滤闪现消息

New in version 0.9.

你可以视情况通过传递一个类别列表来过滤 get_flashed_messages() 的 结果。这个功能有助于在不同位置显示不同类别的消息。

```
{% with errors = get_flashed_messages(category_filter=["error"]) %}
{% if errors %}
<div class="alert-message block-message error">
  <a class="close" href="#">×</a>
  <ul>
    {%- for msg in errors %}
    <li>{{ msg }}</li>
    {% endfor -%}
  </ul>
</div>
{% endif %}
{% endwith %}
```

通过 jQuery 使用 AJAX

jQuery 是一个小型的 JavaScript 库，通常用于简化 DOM 和 JavaScript 的使用。它 是一个非常好的工具，可以通过在服务端和客户端交换 JSON 来使网络应用更具有动态性。

JSON 是一种非常轻巧的传输格式，非常类似于 Python 原语（数字、字符串、字典和列表 ）。 JSON 被广泛支持，易于解析。 JSON 在几年之前开始流行，在网络应用中迅速取代了 XML 。

载入 jQuery

为了使用 jQuery ，你必须先把它下载下来，放在应用的静态目录中，并确保它被载入。 理想情况下你有一个用于所有页面的布局模板。在这个模板的 的底部添加一个 script 语句来载入 jQuery：

```
<script type=text/javascript src="{{
  url_for('static', filename='jquery.js') }}"></script>
```

另一个方法是使用 Google 的 AJAX 库 API 来载入 jQuery：

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>
<script>window.jQuery || document.write('<script src="{{
  url_for('static', filename='jquery.js') }}">\x3C/script>')</script>
```

在这种方式中，应用会先尝试从 Google 下载 jQuery ，如果失败则会调用静态目录中的 备用 jQuery 。这样做的好处是如果用户已经去过使用与 Google 相同版本的 jQuery 的 网站后，访问你的网站时，页面可能会更快地载入，因为浏览器已经缓存了 jQuery 。

我的网站在哪里?

我的网站在哪里? 如果你的应用还在开发中，那么答案很简单：它在本机的某个端口上， 且在服务器的根路径下。但是如果以后要把应用移到其他位置（ 例如< http://example.com/myapp> ）上呢? 在服务端，这个问题不成为问题，可以使用 url_for() 函数来得到答案。但是如果我们使用 jQuery ，那么就不能硬码应用的路径，只能使用动态路径。怎么办?

一个简单的方法是在页面中添加一个 script 标记，设置一个全局变量来表示应用的根 路径。示例：

```
<script type=text/javascript>
  $SCRIPT_ROOT = {{ request.script_root|tojson|safe }};
</script>
```

在 Flask 0.10 版本以前版本中，使用 |safe 是有必要的，是为了使 Jinja 不要转义 JSON 编码的字符串。通常这样做不是必须的，但是在 script 内部我们需要这么做。

进一步说明

在 HTML 中， script 标记是用于声明 CDATA 的，也就是说声明的内容不会被解析。 之间的内容都会被作为脚本处理。这也意味着 在 script 标记之间不会存在任何 </ 。在这里 |tojson 会正确处理问题， 并为你转义斜杠（ {{ ""|tojson|safe }} 会被渲染为 "<\/script>" ）。

在 Flask 0.10 版本中更进了一步，把所有 HTML 标记都用 unicode 转义了，这样使 Flask 自动把 HTML 转换为安全标记。

JSON 视图函数

现在让我们来创建一个服务端函数，这个函数接收两个 URL 参数（两个需要相加的数字 ），然后向应用返回一个 JSON 对象。下面这个例子是非常不实用的，因为一般会在客户端完成类似工作，但这个例子可以简单明了地展示如何使用 jQuery 和 Flask:

```
from flask import Flask, jsonify, render_template, request
app = Flask(__name__)

@app.route('/_add_numbers')
def add_numbers():
    a = request.args.get('a', 0, type=int)
    b = request.args.get('b', 0, type=int)
    return jsonify(result=a + b)

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

正如你所见，我还添加了一个 index 方法来渲染模板。这个模板会按前文所述载入 jQuery 。模板中有一个用于两个数字相加的表单和一个触发服务端函数的链接。

注意，这里我们使用了 get() 方法。它 不会调用失败。如果字典的键不存在，就会返回一个缺省值（这里是 0 ）。更进一步 它还会把值转换为指定的格式（这里是 int ）。在脚本（ API 、 JavaScript 等 ）触发的代码中使用它特别方便，因为在这种情况下不需要特殊的错误报告。

HTML

你的 index.html 模板要么继承一个已经载入 jQuery 和设置好 $SCRIPT_ROOT 变量的 layout.html 模板，要么在模板开头就做好那两件事。下面就是应用的 HTML 示例（ index.html ）。注意，我们把脚本直接放入了 HTML 中。通常更好的方式是放在 独立的脚本文件中：

```
<script type=text/javascript>
  $(function() {
    $('a#calculate').bind('click', function() {
      $.getJSON($SCRIPT_ROOT + '/_add_numbers', {
        a: $('input[name="a"]').val(),
        b: $('input[name="b"]').val()
      }, function(data) {
        $("#result").text(data.result);
      });
      return false;
    });
  });
</script>
<h1>jQuery Example</h1>
<p><input type=text size=5 name=a> +
   <input type=text size=5 name=b> =
   <span id=result>?</span>
<p><a href=# id=calculate>calculate server side</a>
```

这里不讲述 jQuery 运行详细情况，仅对上例作一个简单说明：

1. $(function() { ... }) 定义浏览器在页面的基本部分载入完成后立即执行的. 代码。
2. $('selector') 选择一个元素供你操作。
3. element.bind('event', func) 定义一个用户点击元素时运行的函数。如果函数返回 false ，那么缺省行为就不会起作用（本例为转向 # URL ）。
4. \$.getJSON(url, data, func) 向 url 发送一个 GET 请求，并把 data 对象的内容作为查询参数。一旦有数据返回，它将调用指定的函数，并把返回值作为函数的参数。注意，我们可以在这里使用先前定义的 $SCRIPT_ROOT 变量。

如果你没有一个完整的概念，请从 github 下载示例源代码 。

自定义出错页面

Flask 有一个方便的 abort() 函数，它可以通过一个 HTTP 出错代码退出一个请求。它还提供一个包含基本说明的出错页面，页面显示黑白的文本，很朴素。

用户可以根据错误代码或多或少知道发生了什么错误。

常见出错代码

以下出错代码是用户常见的，即使应用正常也会出现这些出错代码：

404 Not Found 这是一个古老的“朋友，你使用了一个错误的 URL ”信息。这个信息出现得如此频繁，以至于连刚上网的新手都知道 404 代表：该死的，我要看的东西不见了。 一个好的做法是确保 404 页面上有一些真正有用的东西，至少要有一个返回首页的链接。

403 Forbidden 如果你的网站上有某种权限控制，那么当用户访问未获授权内容时应当发送 403 代码。因此请确保当用户尝试访问未获授权内容时得到正确的反馈。

410 Gone 你知道 “404 Not Found” 有一个名叫 “410 Gone” 的兄弟吗? 很少有人使用这个 代码。如果资源以前曾经存在过，但是现在已经被删除了，那么就应该使用 410 代码，而不是 404 。如果你不是在数据库中把文档永久地删除，而只是给文档打了一个删除标记，那么请为用户考虑，应当使用 410 代码，并显示信息告知用户要找的东西已经删除。

500 Internal Server Error 这个代码通常表示程序出错或服务器过载。强烈建议把这个页面弄得友好一点， 因为你的应用迟早 会出现故障的（参见掌握应用错误）。

出错处理器

一个出错处理器是一个函数，就像一个视图函数一样。与视图函数不同之处在于出错处理器 在出现错误时被调用，且传递错误。错误大多数是一个 HTTPException ，但是有一个例外：当出现内部服务器错误 时会把异常实例传递给出错处理器。

出错处理器使用 errorhandler() 装饰器注册，注册时应提供异常的 出代码。请记住， Flask 不会 为你设置出错代码，因此请确保在返回响应时，同时提供 HTTP 状态代码。

以下是一个处理 “404 Page Not Found” 异常的示例:

```
from flask import render_template

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

示例模板：

```
{% extends "layout.html" %}
{% block title %}Page Not Found{% endblock %}
{% block body %}
  <h1>Page Not Found</h1>
  <p>What you were looking for is just not there.
  <p><a href="{{ url_for('index') }}">go somewhere nice</a>
{% endblock %}
```

惰性载入视图

Flask 通常使用装饰器。装饰器简单易用，只要把 URL 放在相应的函数的前面就可以了。 但是这种方式有一个缺点：使用装饰器的代码必须预先导入，否则 Flask 就无法真正找到你的函数。

当你必须快速导入应用时，这就会成为一个问题。在 Google App Engine 或其他系统中， 必须快速导入应用。因此，如果你的应用存在这个问题，那么必须使用集中 URL 映射。

add_url_rule() 函数用于集中 URL 映射，与使用装饰器不同的是你 需要一个设置应用所有 URL 的专门文件。

转换为集中 URL 映射

假设有如下应用:

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

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

@app.route('/user/<username>')
def user(username):
    pass
```

为了集中映射，我们创建一个不使用装饰器的文件（ views.py ）:

```
def index():
    pass

def user(username):
    pass
```

在另一个文件中集中映射函数与 URL:

```
from flask import Flask
from yourapplication import views
app = Flask(__name__)
app.add_url_rule('/', view_func=views.index)
app.add_url_rule('/user/<username>', view_func=views.user)
```

延迟载入

至此，我们只是把视图与路由分离，但是模块还是预先载入了。理想的方式是按需载入 视图。下面我们使用一个类似函数的辅助类来实现按需载入:

```
from werkzeug import import_string, cached_property

class LazyView(object):

    def __init__(self, import_name):
        self.__module__, self.__name__ = import_name.rsplit('.', 1)
        self.import_name = import_name

    @cached_property
    def view(self):
        return import_string(self.import_name)

    def __call__(self, *args, **kwargs):
        return self.view(*args, **kwargs)
```

上例中最重要的是正确设置 `__module__` 和 `__name__`，它被用于在不提供 URL 规则的情况下正确命名 URL 规则。

然后可以这样集中定义 URL 规则:

```
from flask import Flask
from yourapplication.helpers import LazyView
app = Flask(__name__)
app.add_url_rule('/',
                 view_func=LazyView('yourapplication.views.index'))
app.add_url_rule('/user/<username>',
                 view_func=LazyView('yourapplication.views.user'))
```

还可以进一步优化代码：写一个函数调用 add_url_rule() ，加上应用前缀和点符号:

```
def url(url_rule, import_name, **options):
    view = LazyView('yourapplication.' + import_name)
    app.add_url_rule(url_rule, view_func=view, **options)

url('/', 'views.index')
url('/user/<username>', 'views.user')
```

有一件事情要牢记：请求前和请求后处理器必须在第一个请求前导入。

其余的装饰器可以同样用上述方法改写。

在 Flask 中使用 MongoKit

现在使用文档型数据库来取代关系型数据库已越来越常见。本方案展示如何使用 MongoKit ,它是一个用于操作 MongoDB 的文档映射库。

本方案需要一个运行中的 MongoDB 服务器和已安装好的 MongoKit 库。

使用 MongoKit 有两种常用的方法，下面逐一说明：

声明

声明是 MongoKit 的缺省行为。这个思路来自于 Django 或 SQLAlchemy 的声明。

下面是一个示例 app.py 模块:

```
from flask import Flask
from mongokit import Connection, Document

 # configuration

MONGODB_HOST = 'localhost'
MONGODB_PORT = 27017

 # create the little application object

app = Flask(__name__)
app.config.from_object(__name__)

 # connect to the database

connection = Connection(app.config['MONGODB_HOST'],
                        app.config['MONGODB_PORT'])
```

如果要定义模型，那么只要继承 MongoKit 的 Document 类就行了。如果你已经读过 SQLAlchemy 方案，那么可以会奇怪这里为什么没有使用会话，甚至没有定义一个 init_db 函数。一方面是因为 MongoKit 没有类似会话在东西。有时候这样会多写一点代码，但会使它的速度更快。另一方面是因为 MongoDB 是无模式的。这就意味着可以在 插入数据的时候修改数据结构。 MongoKit 也是无模式的，但会执行一些验证，以确保数据的完整性。

以下是一个示例文档（ 把示例内容也放入 app.py ）:

```
def max_length(length):
    def validate(value):
        if len(value) <= length:
            return True
        raise Exception('%s must be at most %s characters long' % length)
    return validate

class User(Document):
    structure = {
        'name': unicode,
        'email': unicode,
    }
    validators = {
        'name': max_length(50),
        'email': max_length(120)
    }
    use_dot_notation = True
    def __repr__(self):
        return '<User %r>' % (self.name)
```

第 18 章 在当前连接中注册用户文档

connection.register([User]) 上例展示如何定义模式（命名结构）和字符串最大长度验证器。上例中还使用了一个 MongoKit 中特殊的 use_dot_notation 功能。缺省情况下， MongoKit 的运作方式和 Python 的字典类似。但是如果 use_dot_notation 设置为 True ，那么就可几乎像其他 ORM 一样使用点符号来分隔属性。

可以像下面这样把条目插入数据库中：

```
>>> from yourapplication.database import connection
>>> from yourapplication.models import User
>>> collection = connection['test'].users
>>> user = collection.User()
>>> user['name'] = u'admin'
>>> user['email'] = u'admin@localhost'
>>> user.save()
```

注意， MongoKit 对于列类型的使用是比较严格的。对于 name 和 email 列，你都不能使用 str 类型，应当使用 unicode 。

查询非常简单：

```
>>> list(collection.User.find())
[<User u'admin'>]
>>> collection.User.find_one({'name': u'admin'})
<User u'admin'>
```

PyMongo 兼容层

如果你只需要使用 PyMongo ，也可以使用 MongoKit 。在这种方式下可以获得最佳的性能。注意，以下示例中，没有 MongoKit 与 Flask 整合的内容，整合的方式参见上文:

```
from MongoKit import Connection

connection = Connection()
```

使用 insert 方法可以插入数据。但首先必须先得到一个连接。这个连接类似于 SQL 界的表。

```
>>> collection = connection['test'].users
>>> user = {'name': u'admin', 'email': u'admin@localhost'}
>>> collection.insert(user)
```

MongoKit 会自动提交。

直接使用集合查询数据库：

```
>>> list(collection.find())
[{u'_id': ObjectId('4c271729e13823182f000000'), u'name': u'admin', u'email': u'admin@localhost'}]
>>> collection.find_one({'name': u'admin'})
{u'_id': ObjectId('4c271729e13823182f000000'), u'name': u'admin', u'email': u'admin@localhost'}
```

查询结果为类字典对象：

```
>>> r = collection.find_one({'name': u'admin'})
>>> r['email']
u'admin@localhost'
```

关于 MongoKit 的更多信息，请移步其官方网站 。

添加一个页面图标

一个“页面图标”是浏览器在标签或书签中使用的图标，它可以给你的网站加上一个唯一 的标示，方便区别于其他网站。

那么如何给一个 Flask 应用添加一个页面图标呢？首先，显而易见的，需要一个图标。 图标应当是 16 X 16 像素的 ICO 格式文件。这不是规定的，但却是一个所有浏览器都 支持的事实上的标准。把 ICO 文件命名为 favicon.ico 并放入静态 文件目录 中。

现在我们要让浏览器能够找到你的图标，正确的做法是在你的 HTML 中添加一个链接。 示例：

```
<link rel="shortcut icon" href="{{ url_for('static', filename='favicon.ico') }}">
```

对于大多数浏览器来说，这样就完成任务了，但是一些老古董不支持这个标准。老的标准是把名为“ favicon.ico ”的图标放在服务器的根目录下。如果你的应用不是挂接在域的 根目录下，那么你需要定义网页服务器在根目录下提供这个图标，否则就无计可施了。 如果你的应用位于根目录下，那么你可以简单地进行重定向:

```
app.add_url_rule('/favicon.ico',
                 redirect_to=url_for('static', filename='favicon.ico'))
```

如果想要保存额外的重定向请求，那么还可以使用 send_from_directory() 函数来写一个视图:

```
import os
from flask import send_from_directory

@app.route('/favicon.ico')
def favicon():
    return send_from_directory(os.path.join(app.root_path, 'static'),
                               'favicon.ico', mimetype='image/vnd.microsoft.icon')
```

上例中的 MIME 类型可以省略，浏览器会自动猜测类型。但是我们在例子中明确定义了， 省去了额外的猜测，反正这个类型是不变的。

上例会通过你的应用来提供图标，如果可能的话，最好配置你的专用服务器来提供图标， 配置方法参见网页服务器的文档。

另见 Wikipedia 上的页面图标词条

流内容

有时候你会需要把大量数据传送到客户端，不在内存中保存这些数据。当你想把运行中产生 的数据不经过文件系统，而是直接发送给客户端时，应当怎么做呢?

答案是使用生成器和直接响应。

基本用法

下面是一个在运行中产生大量 CSV 数据的基本视图函数。其技巧是调用一个内联函数生成 数据，把这个函数传递给一个响应对象:

```
from flask import Response

@app.route('/large.csv')
def generate_large_csv():
    def generate():
        for row in iter_all_rows():
            yield ','.join(row) + '\n'
    return Response(generate(), mimetype='text/csv')
```

每个 yield 表达式被直接传送给浏览器。注意，有一些 WSGI 中间件可能会打断流 内容，因此在使用分析器或者其他工具的调试环境中要小心一些。

模板中的流内容

Jinja2 模板引擎也支持分片渲染模板。这个功能不是直接被 Flask 支持的，因为它太特殊了，但是你可以方便地自己来做:

```
from flask import Response

def stream_template(template_name, **context):
    app.update_template_context(context)
    t = app.jinja_env.get_template(template_name)
    rv = t.stream(context)
    rv.enable_buffering(5)
    return rv

@app.route('/my-large-page.html')
```

```
def render_large_template():
    rows = iter_all_rows()
    return Response(stream_template('the_template.html', rows=rows))
```

上例的技巧是从 Jinja2 环境中获得应用的模板对象，并调用 stream() 来代替 render() ，返回 一个流对象来代替一个字符串。由于我们绕过了 Flask 的模板渲染函数使用了模板对象 本身，因此我们需要调用 update_template_context() ，以确保 更新被渲染的内容。这样，模板遍历流内容。由于每次产生内容后，服务器都会把内容发送给客户端，因此可能需要缓存来保存内容。我们使用了 rv.enable_buffering(size) 来进行缓存。 5 是一个比较明智的缺省值。

环境中的流内容

New in version 0.9.

注意，当你生成流内容时，请求环境已经在函数执行时消失了。 Flask 0.9 为你提供了 一点帮助，让你可以在生成器运行期间保持请求环境:

```
from flask import stream_with_context, request, Response

@app.route('/stream')
def streamed_response():
    def generate():
        yield 'Hello '
        yield request.args['name']
        yield '!'
    return Response(stream_with_context(generate()))
```

如果没有使用 stream_with_context() 函数，那么就会引发 RuntimeError 错误。

延迟的请求回调

Flask 的设计思路之一是：响应对象创建后被传递给一串回调函数，这些回调函数可以修改 或替换响应对象。当请求处理开始的时候，响应对象还没有被创建。响应对象是由一个视图函数或者系统中的其他组件按需创建的。

但是当响应对象还没有创建时，我们如何修改响应对象呢？比如在一个请求前函数中，我们需要根据响应对象设置一个 cookie 。

通常我们选择避开这种情形。例如可以尝试把应用逻辑移动到请求后函数中。但是，有时候这个方法让人不爽，或者让代码变得很丑陋。

变通的方法是把一堆回调函数贴到 g 对象上，并且在请求结束时调用这些回调函数。这样你就可以在应用的任意地方延迟回调函数的执行。

装饰器

下面的装饰器是一个关键，它在 g 对象上注册一个函数列表:

```
from flask import g

def after_this_request(f):
    if not hasattr(g, 'after_request_callbacks'):
        g.after_request_callbacks = []
    g.after_request_callbacks.append(f)
    return f
```

调用延迟的回调函数

至此，通过使用 after_this_request 装饰器，使得函数在请求结束时可以被调用。现在我们来实现这个调用过程。我们把这些函数注册为 after_request() 回调函数:

```
@app.after_request
def call_after_request_callbacks(response):
    for callback in getattr(g, 'after_request_callbacks', ()):
        callback(response)
    return response
```

一个实例

现在我们可以方便地随时随地为特定请求注册一个函数，让这个函数在请求结束时被调用。 例如，你可以在请求前函数中把用户的当前语言记录到 cookie 中:

```
from flask import request

@app.before_request
def detect_user_language():
    language = request.cookies.get('user_lang')
    if language is None:
        language = guess_language_from_request()
        @after_this_request
        def remember_language(response):
            response.set_cookie('user_lang', language)
    g.language = language
 Logo
Table Of Contents
```

添加 HTTP 方法重载

一些 HTTP 代理不支持所有 HTTP 方法或者不支持一些较新的 HTTP 方法（例如 PACTH ）。在这种情况下，可以通过使用完全相反的协议，用一种 HTTP 方法来“代理”另一种 HTTP 方法。

实现的思路是让客户端发送一个 HTTP POST 请求，并设置 X-HTTP-Method-Override 头部为需要的 HTTP 方法（例如 PATCH ）。

通过 HTTP 中间件可以方便的实现:

```
class HTTPMethodOverrideMiddleware(object):
    allowed_methods = frozenset([
        'GET',
        'HEAD',
        'POST',
        'DELETE',
        'PUT',
        'PATCH',
        'OPTIONS'
    ])
    bodyless_methods = frozenset(['GET', 'HEAD', 'OPTIONS', 'DELETE'])

    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        method = environ.get('HTTP_X_HTTP_METHOD_OVERRIDE', '').upper()
        if method in self.allowed_methods:
            method = method.encode('ascii', 'replace')
            environ['REQUEST_METHOD'] = method
        if method in self.bodyless_methods:
            environ['CONTENT_LENGTH'] = '0'
        return self.app(environ, start_response)
```

通过以下代码就可以与 Flask 一同工作了:

```
from flask import Flask

app = Flask(__name__)
app.wsgi_app = HTTPMethodOverrideMiddleware(app.wsgi_app)
```

请求内容校验

请求数据会由不同的代码来处理或者预处理。例如 JSON 数据和表单数据都来源于已经 读取并处理的请求对象，但是它们的处理代码是不同的。这样，当需要校验进来的请求 数据时就会遇到麻烦。因此，有时候就有必要使用一些 API 。

幸运的是可以通过包装输入流来方便地改变这种情况。

下面的例子演示在 WSGI 环境下读取和储存输入数据，得到数据的 SHA1 校验:

```
import hashlib

class ChecksumCalcStream(object):

    def __init__(self, stream):
        self._stream = stream
        self._hash = hashlib.sha1()

    def read(self, bytes):
        rv = self._stream.read(bytes)
        self._hash.update(rv)
        return rv

    def readline(self, size_hint):
        rv = self._stream.readline(size_hint)
        self._hash.update(rv)
        return rv

def generate_checksum(request):
    env = request.environ
    stream = ChecksumCalcStream(env['wsgi.input'])
    env['wsgi.input'] = stream
    return stream._hash
```

要使用上面的类，你只要在请求开始消耗数据之前钩接要计算的流就可以了。（ 按：小心操作 request.form 或类似东西。例如 before_request_handlers 就应当小心不要操作。）

用法示例:

```
@app.route('/special-api', methods=['POST'])
def special_api():
    hash = generate_checksum(request)
    # Accessing this parses the input stream
```

```
files = request.files
# At this point the hash is fully constructed.
checksum = hash.hexdigest()
return 'Hash was: %s' % checksum
```

基于 Celery 的后台任务

Celery 是一个 Python 编写的是一个异步任务队列/基于分布式消息传递的作业队列。 以前它有一个 Flask 的集成，但是从版本 3 开始，它进行了一些内部的重构，已经不需要这个集成了。本文主要说明如何在 Flask 中正确使用 Celery 。本文假设你 已经阅读过了其官方文档中的 Celery 入门

安装 Celery

Celery 在 Python 包索引（ PyPI ）上榜上有名，因此可以使用 pip 或 easy_install 之类标准的 Python 工具来安装:

```
$ pip install celery
```

配置 Celery

你首先需要有一个 Celery 实例，这个实例称为 celery 应用。其地位就相当于 Flask 中 Flask 一样。这个实例被用作所有 Celery 相关事务的入口，例如创建 任务、管理工人等等。因此它必须可以被其他模块导入。

例如，你可以把它放在一个 tasks 模块中。这样不需要重新配置，你就可以使用 tasks 的子类，增加 Flask 应用环境的支持，并钩接 Flask 的配置。

只要如下这样就可以在 Falsk 中使用 Celery 了:

```
from celery import Celery

def make_celery(app):
    celery = Celery(app.import_name, broker=app.config['CELERY_BROKER_URL'])
    celery.conf.update(app.config)
    TaskBase = celery.Task
    class ContextTask(TaskBase):
        abstract = True
        def __call__(self, *args, **kwargs):
            with app.app_context():
                return TaskBase.__call__(self, *args, **kwargs)
    celery.Task = ContextTask
    return celery
```

这个函数创建了一个新的 Celery 对象，使用了应用配置中的 broker ，并从 Flask 配置中升级了 Celery 的其余配置。然后创建了一个任务子类，在一个应用环境中包装了任务执行。

最小的例子

基于上文，以下是一个在 Flask 中使用 Celery 的最小例子:

```
from flask import Flask

app = Flask(__name__)
app.config.update(
    CELERY_BROKER_URL='redis://localhost:6379',
    CELERY_RESULT_BACKEND='redis://localhost:6379'
)
celery = make_celery(app)


@celery.task()
def add_together(a, b):
    return a + b
```

这个任务现在可以在后台调用了:

```
>>> result = add_together.delay(23, 42)
>>> result.wait()
65
```

运行 Celery 工人

至此，如果你已经按上文一步一步执行，你会失望地发现你的 .wait() 不会真正返回。这是因为你还没有运行 celery 。你可以这样以工人方式运行 celery:

```
$ celery -A your_application worker
```

把 your_application 字符串替换为你创建 celery 对像的应用包或模块。

19

部署方式

Flask 应用可以采用多种方式部署。在开发时，你可以使用内置的服务器，但是在生产环境 下你就应当选择功能完整的服务器。下面为你提供几个可用的选择。

除了下面提到的服务器之外，如果你使用了其他的 WSGI 服务器，那么请阅读其文档中与使用 WSGI 应用相关的部分。因为 Flask 应用对象的实质就是一个 WSGI 应用。

如果需要快速体验，请参阅《 快速上手 》中的部署到一个网络服务器 。

mod_wsgi (Apache)

如果你正在使用 Apache 网络服务器，那么建议使用 mod_wsgi 。

小心

请务必把 app.run() 放在 if name == 'main': 内部或者放在单独 的文件中，这样可以保证它不会被调用。因为，每调用一次就会开启一个本地 WSGI 服务器。当我们使用 mod_wsgi 部署应用时，不需要使用本地服务器。

安装 mod_wsgi

可以使用包管理器或编译的方式安装 mod_wsgi 。在 UNIX 系统中如何使用源代码安装请阅读 mod_wsgi 安装介绍 。

如果你使用的是 Ubuntu/Debian ，那么可以使用如下命令安装:

```
# apt-get install libapache2-mod-wsgi
```

在 FreeBSD 系统中，可以通过编译 www/mod_wsgi port 或使用 pkg_add 来安装 mod_wsgi：

```
# pkg_add -r mod_wsgi
```

如果你使用 pkgsrc ，那么可以通过编译 www/ap2-wsgi 包来安装 mod_wsgi 。

如果你遇到子进程段错误的话，不要理它，重启服务器就可以了。

创建一个 .wsgi 文件

为了运行应用，你需要一个 yourapplication.wsgi 文件。这个文件包含 mod_wsgi 开始时需要运行的代码，通过代码可以获得应用对象。文件中的 application 对象就是以后要使用的应用。

对于大多数应用来说，文件包含以下内容就可以了:

```
from yourapplication import app as application
```

如果你的应用没有创建函数，只是一个独立的实例，那么可以直接把实例导入为 application 。

把文件放在一个以后可以找得到的地方（例如 /var/www/yourapplication ），并确保 yourapplication 和所有需要使用的库都位于 pythonpath 中。如果你不想在整个系统中安装，建议使用 virtual python 实例。请记住，最好把应用安装到虚拟环境中。 有一个可选项是在 .wsgi 文件中，在导入前加入路径:

```
import sys
sys.path.insert(0, '/path/to/the/application')
```

配置 Apache

最后一件事是为你的应用创建一个 Apache 配置文件。基于安全原因，在下例中我们告诉 mod_wsgi 使用另外一个用户运行应用:

```
<VirtualHost *>
    ServerName example.com

    WSGIDaemonProcess yourapplication user=user1 group=group1 threads=5
    WSGIScriptAlias / /var/www/yourapplication/yourapplication.wsgi

    <Directory /var/www/yourapplication>
        WSGIProcessGroup yourapplication
        WSGIApplicationGroup %{GLOBAL}
        Order deny,allow
        Allow from all
    </Directory>
</VirtualHost>
```

注意： WSGIDaemonProcess 在 Windows 中不会被执行， 使用上面的配置 Apache 会拒绝运行。在 Windows 系统下，请使用下面内容:

```
<VirtualHost *>
        ServerName example.com
        WSGIScriptAlias / C:\yourdir\yourapp.wsgi
        <Directory C:\yourdir>
                Order deny,allow
                Allow from all
        </Directory>
</VirtualHost>
```

更多信息参见 mod_wsgi 维基 。

故障排除

如果你的应用无法运行，请按以下指导排除故障：

问题： 应用无法运行，出错记录显示 SystemExit ignored 应用文件中有 app.run() 调用，但没有放在 `if __name__ == '__main__'：` 块内。要么把这个调用放入块内，要么把它放在一个单独的 run.py 文件中。

问题： 权限错误 有可以是因为使用了错误的用户运行应用。请检查用户及其所在的组（ WSGIDaemonProcess 的 user 和 group 参数 ）是否有权限访问应用 文件夹。

问题： 打印时应用歇菜 请记住 mod_wsgi 不允许使用 sys.stdout 和 sys.stderr 。把 WSGIRestrictStdout 设置为 off 可以去掉这个保护：

```
WSGIRestrictStdout Off
```

或者你可以在 .wsgi 文件中把标准输出替换为其他的流:

```
import sys
sys.stdout = sys.stderr
```

问题： 访问资源时遇到 IO 错误

你的应用可能是一个独立的 .py 文件，且你把它符号连接到了 site-packages 文件夹。这样是不对的，你应当要么把文件夹放到 pythonpath 中，要么把你的应用转换为一个包。

产生这种错误的原因是对于非安装包来说，模块的文件名用于定位资源，如果使用符号连接的话就会定位到错误的文件名。

支持自动重载

为了辅助部署工具，你可以激活自动重载。这样，一旦 .wsgi 文件有所变动， mod_wsgi 就会自动重新转入所有守护进程。

在 Directory 一节中加入以下指令就可以实现自动重载：

```
WSGIScriptReloading On
```

使用虚拟环境

使用虚拟环境的优点是不必全局安装应用所需要的依赖，这样我们就可以更好地按照自己的需要进行控制。如果要在虚拟环境下使用 mod_wsgi ，那么我们要对 .wsgi 略作改变。

在你的 .wsgi 文件顶部加入下列内容:

```
activate_this = '/path/to/env/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))
```

以上代码会根据虚拟环境的设置载入相关路径。请记住路径必须是绝对路径。

独立 WSGI 容器

有一些用 Python 写的流行服务器可以容纳 WSGI 应用，提供 HTTP 服务。这些服务器是 独立运行的，你可以把代理从你的网络服务器指向它们。如果遇到问题，请阅读代理设置一节。

Gunicorn

Gunicorn ‘Green Unicorn’ 是一个 UNIX 下的 WSGI HTTP 服务器，它是一个移植自 Ruby 的 Unicorn 项目的 pre-fork worker 模型。它既支持 eventlet ， 也支持 greenlet 。在 Gunicorn 上运行 Flask 应用非常简单:

```
gunicorn myproject:app
```

Gunicorn 提供许多命令行参数，可以使用 gunicorn -h 来获得帮助。下面的例子使用 4 worker 进程（ -w 4 ）来运行 Flask 应用，绑定到 localhost 的 4000 端口（ -b 127.0.0.1:4000 ）:

```
gunicorn -w 4 -b 127.0.0.1:4000 myproject:app
```

Tornado

Tornado 是构建 FriendFeed 的服务器和工具的开源版本，具有良好的伸缩性，非阻塞性。得益于其非阻塞的方式和对 epoll 的运用，它可以同步处理数以千计的独立 连接，因此 Tornado 是实时 Web 服务的一个理想框架。用它来服务 Flask 是小事一桩:

```
from tornado.wsgi import WSGIContainer
from tornado.httpserver import HTTPServer
from toranado.ioloop import IOLoop
from yourapplication import app

http_server = HTTPServer(WSGIContainer(app))
http_server.listen(5000)
IOLoop.instance().start()
```

Gevent

Gevent 是一个 Python 并发网络库，它使用了基于 libevent 事件循环的 greenlet 来提供一个高级同步 API:

```
from gevent.wsgi import WSGIServer
from yourapplication import app

http_server = WSGIServer(('', 5000), app)
http_server.serve_forever()
```

Twisted Web

Twisted Web 是一个 Twisted 自带的网络服务器，是一个成熟的、异步的、事件驱动的网络库。 Twisted Web 带有一个标准的 WSGI 容器，该容器可以使用 twistd 工具运行命令行来控制:

```
twistd web --wsgi myproject.app
```

这个命令会运行一个名为 app 的 Flask 应用，其模块名为 myproject 。

与 twistd 工具一样， Twisted Web 支持许多标记和选项。更多信息参见 twistd -h 和 twistd web -h 。例如下面命令在前台运行一个来自 myproject 的应用， 端口为 8080:

```
twistd -n web --port 8080 --wsgi myproject.app
```

代理设置

如果你要在一个 HTTP 代理后面在上述服务器上运行应用，那么必须重写一些头部才行。 通常在 WSGI 环境中经常会出现问题的有两个变量： REMOTE_ADDR 和 HTTP_HOST 。 你可以通过设置你的 httpd 来传递这些头部，或者在中间件中修正这些问题。 Werkzeug 带有一个修复工具可以用于常用的设置，但是你可能需要为特定的设置编写你自己的 WSGI 中间件。

下面是一个简单的 nginx 配置，代理目标是 localhost 8000 端口提供的服务，设置了适当的头部:

```
server {
    listen 80;

    server_name _;

    access_log  /var/log/nginx/access.log;
    error_log  /var/log/nginx/error.log;

    location / {
        proxy_pass         http://127.0.0.1:8000/;
        proxy_redirect     off;

        proxy_set_header   Host             $host;
```

```
    proxy_set_header   X-Real-IP       $remote_addr;
    proxy_set_header   X-Forwarded-For  $proxy_add_x_forwarded_for;
  }
}
```

如果你的 httpd 无法提供这些头部，那么最常用的设置是调用 X-Forwarded-Host 定义的主机和 X-Forwarded-For 定义的远程地址:

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

头部可信问题

请注意，在非代理情况下使用这个中间件是有安全问题的，因为它会盲目信任恶意客户端发来的头部。

如果你要根据另一个头部来重写一个头部，那么可以像下例一样使用修复工具:

```
class CustomProxyFix(object):

    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        host = environ.get('HTTP_X_FHOST', '')
        if host:
            environ['HTTP_HOST'] = host
        return self.app(environ, start_response)

app.wsgi_app = Cus

tomProxyFix(app.wsgi_app)
```

uWSGI

uWSGI 也是部署 Flask 的途径之一,类似的部署途径还有 nginx 、 lighttpd 和 cherokee 。其他部署途径的信息参见 FastCGI 和 独立 WSGI 容器 。使用 uWSGI 协议来部署 WSGI 应用的先决条件是需要一个 uWSGI 服务器。 uWSGI 既是一个协议也是一个服务器。如果作为一个服务器， 它可以服务于 uWSGI 、 FastCGI 和 HTTP 协议。

最流行的 uWSGI 服务器是 uwsgi ，本文将使用它来举例，请先安装它。

小心

请务必把 app.run() 放在 `if __name__ == '__main__':` 内部或者放在单独 的文件中，这样可以保证它不会被调用。因为，每调用一次就会开启一个本地 WSGI 服务器。当我们使用 uWSGI 部署应用时，不需要使用本地服务器。

使用 uwsgi 启动你的应用

uwsgi 是基于 python 模块中的 WSGI 调用的。假设 Flask 应用名称为 myapp.py ， 可以使用以下命令:

```
$ uwsgi -s /tmp/uwsgi.sock --module myapp --callable app
```

或者这个命令也行:

```
$ uwsgi -s /tmp/uwsgi.sock -w myapp:app
```

配置 nginx

一个 nginx 的基本 uWSGI 配置如下:

```
location = /yourapplication { rewrite ^ /yourapplication/; }
location /yourapplication { try_files $uri @yourapplication; }
location @yourapplication {
  include uwsgi_params;
  uwsgi_param SCRIPT_NAME /yourapplication;
  uwsgi_modifier1 30;
  uwsgi_pass unix:/tmp/uwsgi.sock;
}
```

这个配置把应用绑定到 /yourapplication 。如果你想要在根 URL 下运行应用非常简单，因为你不必指出 WSGI PATH_INFO 或让 uwsgi 修改器来使用它:

```
location / { try_files $uri @yourapplication; }
location @yourapplication {
    include uwsgi_params;
    uwsgi_pass unix:/tmp/uwsgi.sock;
}
```

FastCGI

FastCGI 是部署 Flask 的途径之一,类似的部署途径还有 nginx 、 lighttpd 和 cherokee 。其他部署途径的信息参见 uWSGI 和独立 WSGI 容器 。本文讲述的是使用 FastCGI 部署，因此先决条件是要有一个 FastCGI 服务器。 flup 最流行的 FastCGI 服务器之一，我们将会在本文中使用它。在阅读下文之前先安装好 flup 。

小心

请务必把 app.run() 放在 if name == 'main': 内部或者放在单独 的文件中，这样可以保证它不会被调用。因为，每调用一次就会开启一个本地 WSGI 服务器。当我们使用 FastCGI 部署应用时，不需要使用本地服务器。

创建一个 .fcgi 文件

首先你必须创建 FastCGI 服务器配置文件，我们把它命名为 yourapplication.fcgi:

```
 #!/usr/bin/python

from flup.server.fcgi import WSGIServer
from yourapplication import app

if __name__ == '__main__':
    WSGIServer(app).run()
```

如果使用的是 Apache ，那么使用了这个文件之后就可以正常工作了。但是如果使用的是 nginx 或老版本的 lighttpd ，那么需要显式地把接口传递给 FastCGI 服务器，即把接口的路径传递给 WSGIServer:

```
WSGIServer(application, bindAddress='/path/to/fcgi.sock').run()
```

这个路径必须与服务器配置中定义的路径一致。

把这个 yourapplication.fcgi 文件放在一个以后可以找得到的地方，最好是 /var/www/yourapplication 或类似的地方。

为了让服务器可以执行这个文件，请给文件加上执行位，确保这个文件可以执行:

```
 # chmod +x /var/www/yourapplication/yourapplication.fcgi
```

配置 Apache

上面的例子对于基本的 Apache 部署已经够用了，但是你的 .fcgi 文件会暴露在应用的 URL 中，比如 example.com/yourapplication.fcgi/news/ 。有多种方法可以避免出现这中情况。一个较好的方法是使用 ScriptAlias 配置指令:

```
<VirtualHost *>
    ServerName example.com
    ScriptAlias / /path/to/yourapplication.fcgi/
</VirtualHost>
```

如果你无法设置 ScriptAlias ，比如你使用的是一个共享的网络主机，那么你可以使用 WSGI 中间件把 yourapplication.fcgi 从 URL 中删除。你可以这样设置 .htaccess:

```
<IfModule mod_fcgid.c>
   AddHandler fcgid-script .fcgi
   <Files ~ (\.fcgi)>
       SetHandler fcgid-script
       Options +FollowSymLinks +ExecCGI
   </Files>
</IfModule>

<IfModule mod_rewrite.c>
   Options +FollowSymlinks
   RewriteEngine On
   RewriteBase /
   RewriteCond %{REQUEST_FILENAME} !-f
   RewriteRule ^(.*)$ yourapplication.fcgi/$1 [QSA,L]
</IfModule>
```

设置 yourapplication.fcgi:

```
 #!/usr/bin/python

 #: optional path to your local python site-packages folder

import sys
sys.path.insert(0, '<your_local_path>/lib/python2.6/site-packages')

from flup.server.fcgi import WSGIServer
from yourapplication import app

class ScriptNameStripper(object):
```

```
  def __init__(self, app):
    self.app = app

  def __call__(self, environ, start_response):
    environ['SCRIPT_NAME'] = ''
    return self.app(environ, start_response)

app = ScriptNameStripper(app)

if __name__ == '__main__':
  WSGIServer(app).run()
```

配置 lighttpd

一个 lighttpd 的基本 FastCGI 配置如下:

```
fastcgi.server = ("/yourapplication.fcgi" =>
  ((
    "socket" => "/tmp/yourapplication-fcgi.sock",
    "bin-path" => "/var/www/yourapplication/yourapplication.fcgi",
    "check-local" => "disable",
    "max-procs" => 1
  ))
)

alias.url = (
  "/static/" => "/path/to/your/static"
)

url.rewrite-once = (
  "^(/static($|/.*))$" => "$1",
  "^(/.*)$" => "/yourapplication.fcgi$1"
```

请记住启用 FastCGI 、 alias 和 rewrite 模块。以上配置把应用绑定到 /yourapplication 。如果你想要让应用在根 URL 下运行，那么必须使用 LighttpdCGIRootFix 中间件来解决一个 lighttpd 缺陷。

请确保只有应用在根 URL 下运行时才使用上述中间件。更多信息请阅读 FastCGI 和 Python（注意，已经不再需要把一个接口显式传递给 run() 了）。

配置 Nginx

在 Nginx 上安装 FastCGI 应用有一些特殊，因为缺省情况下不传递 FastCGI 参数。

一个 Nginx 的基本 FastCGI 配置如下:

```
location = /yourapplication { rewrite ^ /yourapplication/ last; }
location /yourapplication { try_files $uri @yourapplication; }
location @yourapplication {
   include fastcgi_params;
   fastcgi_split_path_info ^(/yourapplication)(.*)$;
   fastcgi_param PATH_INFO $fastcgi_path_info;
   fastcgi_param SCRIPT_NAME $fastcgi_script_name;
   fastcgi_pass unix:/tmp/yourapplication-fcgi.sock;
}
```

这个配置把应用绑定到 /yourapplication 。如果你想要在根 URL 下运行应用非常简单，因为你不必指出如何计算出 PATH_INFO 和 SCRIPT_NAME:

```
location / { try_files $uri @yourapplication; }
location @yourapplication {
   include fastcgi_params;
   fastcgi_param PATH_INFO $fastcgi_script_name;
   fastcgi_param SCRIPT_NAME "";
   fastcgi_pass unix:/tmp/yourapplication-fcgi.sock;
}
```

运行 FastCGI 进程

Nginx 和其他服务器不会载入 FastCGI 应用，你必须自己载入。Supervisor 可以管理 FastCGI 进程。 在启动时你可以使用其他 FastCGI 进程管理器或写一个脚本来运行 .fcgi 文件，例如使用一个 SysV init.d 脚本。如果是临时使用，你可以在一个 GNU screen 中运行 .fcgi 脚本。运行细节参见 man screen ，同时请注意这是一个手动启动方法， 不会在系统重启时自动启动:

```
$ screen
$ /var/www/yourapplication/yourapplication.fcgi
```

调试

在大多数服务器上， FastCGI 部署难以调试。通常服务器日志只会告诉你类似 “ premature end of headers ”的内容。为了调试应用，查找出错的原因，你必须切换到正确的用户并手动执行应用。

下例假设你的应用是 application.fcgi ，且你的网络服务用户为 www-data:

```
$ su www-data
$ cd /var/www/yourapplication
$ python application.fcgi
Traceback (most recent call last):
  File "yourapplication.fcgi", line 4, in <module>
ImportError: No module named yourapplication
```

上面的出错信息表示“yourapplication”不在 python 路径中。原因可能有：

- 使用了相对路径。在当前工作路径下路径出错。
- 当前网络服务器设置未正确设置环境变量。
- 使用了不同的 python 解释器。

CGI

如果其他的部署方式都不管用，那么就只能使用 CGI 了。 CGI 适应于所有主流服务器， 但是其性能稍弱。

这也是在 Google 的 App Engine 使用 Flask 应用的方法，其执行方式类似于 CGI 环境。

小心

请务必把 app.run() 放在 `if __name__ == '__main__':` 内部或者放在单独 的文件中，这样可以保证它不会被调用。因为，每调用一次就会开启一个本地 WSGI 服务器。当我们使用 CGI 或 App Engine 部署应用时，不需要使用本地服务器。

在使用 CGI 时，你还必须确保代码中不包含任何 print 语句，或者 sys.stdout 被重载，不会写入 HTTP 响应中。

创建一个 .cgi 文件

首先，你需要创建 CGI 应用文件。我们把它命名为 yourapplication.cgi:

```
#!/usr/bin/python

from wsgiref.handlers import CGIHandler
from yourapplication import app

CGIHandler().run(app)
```

服务器设置

设置服务器通常有两种方法。一种是把 .cgi 复制为 cgi-bin （ 并且使用 mod_rewrite 或其他类似东西来改写 URL ）； 另一种是把服务器直接指向文件。

例如，如果使用 Apache ，那么可以把如下内容放入配置中：

```
ScriptAlias /app /path/to/the/application.cgi
```

在共享的网络服务器上，你可能无法变动 Apache 配置。在这种情况下，你可以使用你的公共目录中的 .htaccess 文件。但是 ScriptAlias 指令会失效：

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f # Don't interfere with static files
RewriteRule ^(.*)$ /path/to/the/application.cgi/$1 [L]
```

更多信息参见你所使用的服务器的文档。

20

大型应用

以下是一些建议，当你的代码库日益壮大或者应用需要规划时可以参考。

阅读源代码

Werkzeug（ WSGI ）和 Jinja（ 模板 ）是两个被广泛使用的工具，而 Flask 起源就是 用于展示如何基于这两个工具创建你自己的框架。随着不断地开发， Flask 被越来越多 的人认可了。当你的代码库日益壮大时，不应当仅仅是使用 Flask ，而更应当理解它。 阅读 Flask 的源代码吧。 Flask 的源代码阅读方便，文档公开，有利于你直接使用内部的 API 。 Flask 坚持把上游库的 API 文档化，并文档化自己内部的工具，方便按你的 需要找到挂接点。

挂接，扩展

API 文档随处可见可用重载、挂接点和信号 。你可以定制类似请求或响应对象的自定义类。请深入研究你所使用的 API ，并在 Flask 发行版中有哪些可以立即使用的可定制部分。请研究你的哪些项目可以重构为工具集或 Flask 扩展。你可以在 社区中发现很多 扩展 。如果找不到满意的， 那就写一个你自己的吧。

继承

Flask 类有许多方法专门为继承而设计。你可通过继承 Flask（参见链接的方法文档）快速的添加或者定制行为，并把子类 实例化为一个应用类。这种方法同样适用于应用工厂 。

用中间件包装

应用调度 一文中详细阐述了如何使用中间件。你可以引入中间件来包装你的 Flask 实例，在你的应用和 HTTP 服务器之间的层有所作为。 Werkzeug 包含许多中间件 。

派生

如果以下建议都没有用，那么直接派生 Flask 吧。 Flask 的主要代码都在 Werkzeug 和 Jinja2 这两个库内。这两个库起了主要作用。 Flask 只是把它们粘合在一起而已。对于一个项目来讲，底层框架的切入点很重要。因为如果不重视这一点，那么框架会变得非常 复杂，势必带来陡峭的学习曲线，从而吓退用户。

Flask 并不推崇唯一版本。许多人为了避免缺陷，都使用打过补丁或修改过的版本。这个理念在 Flask 的许可中也有所体现：你不必返回你对框架所做的修改。

分支的缺点是大多数扩展都会失效，因为新的框架会使用不同的导入名称。更进一步： 整合上游的变动将会变得十分复杂，上游变动越多，则整合越复杂。因此，创建分支一般是不得不为之的最后一招。

专家级的伸缩性

对于大多数网络应用来说，最复杂的莫过于对于用户量和数据量提供良好的伸缩性。 Flask 本身具有良好的伸缩性，其伸缩性受限于你的应用代码、所使用的数据储存方式、 Python 实现和应用所运行的服务器。

如果服务器数量增加一倍，你的应用性能就增加一倍，那么就代表伸缩性好。如果伸缩性 不好，那么即使增加服务器的数量，也不会得到更好的性能。伸缩性更差的甚至不支持增加第二台服务器。

Flask 中唯一影响伸缩性的因素是环境本地代理。Flask 中的环境本地代理可以被定义为线程、进程或 greenlet 。如果你的服务器不支持这些，那么 Flask 就不能支持全局代理。 但是，当今主流的服务器都支持线程、进程或 greenlet ，以提高并发性。 Flask 的基础 库 Werkzeug 对于线程、进程或 greenlet 都能够提供良好的支持。

与社区沟通

不管你的代码库是否强大， Flask 开发者总是保持框架的可操作性。如果发现 Flask 有 什么问题，请立即通过邮件列表或 IRC 与社区进行沟通。对于 Flask 及其扩展的开发都 来说，提升其在大型应用中的功能的最佳途径是倾听用户的心声。

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/flask-guide/