导读

本电子书为 Flask 官方文档的中文译本，最早由网友亦念、云尔和 atupal 发布于网络，由汇智网编目整理，是网上流传最广的 Flask 文档之一。

毫无疑问，对于开发者而言，官方文档是最权威的文档，但往往也最枯燥的，从官方文档开始使开发者的学习效率大打折扣。为了弥补这一遗憾，汇智网推出了适合初学者快速上手的在线互动式 Flask 开发课程，课程创作团队在深入研读 Flask 源码的基础上，引导学习者循序渐进地理解 Flask 中的诸多知识难点。读者可以通过以下链接访问《深入浅出 Flask》在线教程：http://xc.hubwiz.com/course/562427361bc20c980538e26f?affid=flaskom7878

教程预置了开发环境。进入教程后，可以在每一个知识点立刻进行同步实践，而不必在开发环境的搭建上浪费时间：

汇智网带来的是一种全新的交互式学习方式，可以极大提高学习编程的效率和学习效果：

汇智网课程内容已经覆盖以下的编程技术：

Node.js、MongoDB、JavaScript、C、C#、PHP、Python、Angularjs、Ionic、React、UML、redis、mySQL、Nginx、CSS、HTML、Flask、Gulp、Mocha、Git、Meteor、Canvas、zebra、Typescript、Material Design Lite、ECMAScript、Elasticsearch、Mongoose、jQuery、d3.js、django、cheerio、SVG、phoneGap、Bootstrap、jQueryMobile、Saas、YAML、Vue.js、webpack、Firebird，jQuery EasyUI，ruby，asp.net，c++，Express ，Spark.....

引子

1、欢迎使用 Flask

欢迎阅读 Flask 文档。 本文档分为几个部分。我推荐您先从*安装*开始，之后再浏览 *快速入门* 章节。 *教程* 比快速入门更详细地介绍了如何用 Flask 创建一个完整的应用（虽然很小）。 想要深入了解 Flask 内部细节，请查阅 *API* 文档。*Flask 代码模式* 章节介绍了一些常见模式。

Flask 依赖两个外部库： Jinja2 模板引擎和 Werkzeug WSGI 工具集。此文档不包含这两个库的文档。要细读它们的文档，请点击下面的链接：

- Jinja2 文档
- Werkzeug 文档

2、前言

请在使用 Flask 前阅读。希望本文能回答你一些关于 Flask 的用途和目标以及 Flask 适用情境的问题。

2.1“微”是什么意思？

“微”(micro) 并不表示你需要把整个 Web 应用塞进单个 Python 文件（虽然确实可以 ），也不意味着 Flask 在功能上有所欠缺。微框架中的“微”意味着 Flask 旨在保持核心简单而易于扩展。Flask 不会替你做出太多决策——比如使用何种数据库。而那些 Flask 所选择的——比如使用何种模板引擎——则很容易替换。除此之外的一切都由可由你掌握。如此，Flask 可以与您珠联璧合。

默认情况下，Flask 不包含数据库抽象层、表单验证，或是其它任何已有多种库可以胜任的功能。然而，Flask 支持用扩展来给应用添加这些功能，如同是 Flask 本身实现的一样。众多的扩展提供了数据库集成、表单验证、上传处理、各种各样的开放认证技术等功能。Flask 也许是“微小”的，但它已准备好在需求繁杂的生产环境中投入使用。

2.2 配置与惯例

Flask 繁多的配置选项在初始状况下都有一个明智的默认值，并会遵循一些惯例。 例如，按照惯例，模板和静态文件分别存储在应用 Python 源

代码树下的子目录 *templates* 和 *static* 里。虽然这个配置可以修改，但你通常不必这么做， 尤其是在刚开始的时候。

2.3 与 Flask 共成长

当你配置好并运行 Flask，你会发现社区中有许多可以集成到生产环境项目的扩展。Flask 核心团队会审阅这些扩展，确保经过检验的扩展在未来版本中仍能适用。

随着你的代码库逐渐壮大，你仍可自由把握项目的设计决策。Flask 会继续尽可能提供的一个非常简单的胶水层，这也是 Python 应该提供的东西。你可以在 SQLAlchemy 或其它数据库工具中实现更高级的模式，酌情引入非关系型数据持久化， 也可以从框架无关的 WSGI——Python 的 Web 接口——工具中获益。

Flask 里有许多钩子用于定制行为。若是需要深层次的定制，可以直接继承 Flask 类。 如果你对此有兴趣，请阅读 *聚沙成塔* 章节。如果你好奇 Flask 的设计原则，请查阅 *Flask 中的设计决策* 章节。

3、给有经验程序员的前言

3.1Flask 中的线程局部变量

Flask 的设计抉择之一就是让简单的任务保持简单；它们的实现不应采用大量的代码， 并且不应对你做出限制。为此，我们选择了一些可能让某些人觉得惊讶或异端的设计。例如，Flask 内部使用线程局部的对象，这样你不必在请求内的函数间传递对象来保证线程安全。这个方法很方便，但为了实现依赖注入，或尝试重用含有与请求挂钩的值的代码之时，需要一个有效的请求环境（Request Context）。

3.2Web 开发危机四伏

快乐码 Web，安全记心间。

你相当可能编写允许用户在你的服务器上注册并留下数据的 Web 应用。即便你是这唯一的用户，也会在应用中留下数据。用户们把数据托付给你，你当然更希望这些数据被妥善安全地保存。

不幸的是，攻陷 Web 应用的手段五花八门。Flask 可保护你免受一个在现代 Web 应用中最常见的安全问题的困扰：跨站脚本攻击（XSS）。Flask 和底层的 Jinja2 模板引擎已经为你应付得足够好，除非你蓄意把不安全的 HTML 标记为安全。但仍有很多导致安全问题的可能。

本文档会在 Web 开发中那些需要注意安全的方面警示你。一些安全上的顾虑远比人们想象的复杂，我们所有人都会有低估漏洞被利用的可能性的

时候——直到一个精明的攻击者找出利用我们程序的方法。而且，不要侥幸认为你的应用没有重要到足够吸引攻击者。取决于攻击的类型，有时候会是自动化的僵尸机器来检测如何在你数据库中填充垃圾内容、恶意程序链接或之类东西。

开发者必须在为需求编写代码时留心安全隐患，在这点上，Flask 与其它框架没有区别。

3.3Python 3 的状态

Python 社区目前处于改善库对 Python 新版本支持的进程中。而当前大力改进中的处境仍有一些问题，使得用户难以迁移到 Python 3 。这些问题一部分是因为长时间没有回顾语言中的变化，一部分也是因为我们没有找出低层 API 应该如何做出修改来适应 Python 3 中 Unicode 的变化。

我们强烈建议在开发时使用 Python 2.6 和 2.7 ，并激活 Python 3 警告。如果你计划在近期升级到 Python 3 ，我们强烈推荐你阅读 如何编写向后兼容的 Python 代码 。

继续阅读 *安装* 或 *快速入门* 。

一、安装

Flask 依赖两个外部库：Werkzeug 和 Jinja2 。 Werkzeug 是一个 WSGI（在 Web 应用和多种服务器之间的标准 Python 接口）工具集。Jinja2 负责渲染模板。

那么如何在你的电脑上安装这一切？虽说条条大道通罗马，但是最强大的方式是 virtualenv ，所以我们首先来看它。

你首先需要 Python 2.6 或更高的版本，所以请确认有一个最新的 Python 2.x 安装。 在 Python 3 中使用 Flask 请参考：*Python 3 支持*。

1.1 virtualenv

你很可能想在开发中用上 virtualenv，如果你有生产环境的 shell 权限，你同样会乐于在生产环境中使用它。

virtualenv 解决了什么问题？如果你像我一样喜欢 Python，不仅会在采用 Flask 的 Web 应用中用上 virtualenv，在别的项目中你也会想用上它。你拥有的项目越多，同时使用不同版本的 Python 工作的可能性也就越大，或者起码需要不同版本的 Python 库。悲惨现实是：常常会有库破坏向后兼容性，然而正经应用不采用外部库的可能微乎其微。当在你的项目中，出现两个或更多依赖性冲突时，你会怎么做？

virtualenv 拯救世界！virtualenv 为每个不同项目提供一份 Python 安装。它并没有真正安装多个 Python 副本，但是它确实提供了一种巧妙的方式来让各项目环境保持独立。让我们来看看 virtualenv 是怎么工作的。

如果你在 Mac OS X 或 Linux 下，下面两条命令可能会适用:

```
$ sudo easy_install virtualenv
```

或更好的:

```
$ sudo pip install virtualenv
```

上述的命令会在你的系统中安装 virtualenv。它甚至可能会存在于包管理器中， 如果你用的是 Ubuntu，可以尝试:

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

如果你用的是 Windows ，而且没有 *easy_install* 命令，那么你必须先安装这个命令。查阅 *Windows 下的 pip 和 distribute* 章节了解如何安装。之后，运行上述的命令，但是要去掉 *sudo* 前缀。

virtualenv 安装完毕后，你可以立即打开 shell 然后创建你自己的环境。我通常创建一个项目文件夹，并在其下创建一个 *venv* 文件夹

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

现在，无论何时你想在某个项目上工作，只需要激活相应的环境。在 OS X 和 Linux 上，执行如下操作:

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

下面的操作适用 Windows:

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

无论通过哪种方式，你现在应该已经激活了 virtualenv（注意你的 shell 提示符显示的是当前活动的环境）。

现在你只需要键入以下的命令来激活 virtualenv 中的 Flask:

```
$ pip install Flask
```

几秒钟后，一切都搞定了。

1.2 全局安装

这样也是可以的，虽然我不推荐。只需要以 root 权限运行 *pip*:

```
$ sudo pip install Flask
```

（在 Windows 上，在管理员权限的命令提示符中去掉 *sudo* 运行这条命令 。）

1.3 活在边缘

如果你需要最新版本的 Flask，有两种方法：你可以使用 *pip* 拉取开发版本， 或让它操作一个 git checkout。无论哪种方式，依然推荐使用 virtualenv。

在一个全新的 virtualenv 中 git checkout 并运行在开发模式下:

```
$ 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
```

这会拉取依赖并激活 git head 作为 virtualenv 中的当前版本。然后你只需要执行 `gitpull origin` 来升级到最新版本。

没有 git 时，获取开发版本的替代操作:

```
$ 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
```

1.4Windows 下的 *pip* 和 *distribute*

在 Windows 下， *easy_install* 的安装稍微有点麻烦，但还是相当简单。最简单的方法是下载 *distribute_setup.py* 文件并运行它。运行这个文件最简单的方法就是打开你的下载文件夹并且双击这个文件。

下一步，把你的 Python 安装中的 Scripts 文件夹添加到 *PATH* 环境变量来， 这样 *easy_install* 命令和其它 Python 脚本就加入到了命令行自动搜索的路径。做法是：右键单击桌面上或是“开始”菜单中的“我的电脑”图标，选择“属性”， 然后单击“高级系统设置”（在 Windows XP 中，单击“高级”选项卡），然后单击 “环境变量”按钮，最后双击“系统变量”栏中的“Path”变量，并加入你的 Python 解释器的 Scripts 文件夹。确保你用分号把它和现有的值分隔开。假设你使用 Python 2.7 且为默认目录，添加下面的值:

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

如此，你就搞定了！打开命令提示符并执行 `easy_install` 测试它是否正常工作。如果你开启了 Windows Vista 或 Windows 7 中的用户账户控制，它应该会提示你使用管理员权限。

现在你有了 `easy_install`， 你可以用它来安装 `pip`:

```
> easy_install pip
```

二、快速入门

迫不及待要开始了吗？本页提供了一个很好的 Flask 介绍，并假定你已经安装好了 Flask。如果没有，请跳转到 *安装* 章节。

2.1 一个最小的应用

一个最小的 Flask 应用看起来会是这样:

```
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*（或是类似的），然后用 Python 解释器来运行。 确保你的应用文件名不是 *flask.py*，因为这将与 Flask 本身冲突。

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

现在访问 http://127.0.0.1:5000/，你会看见 Hello World 问候。

那么，这段代码做了什么？

1. 首先，我们导入了 `Flask` 类。这个类的实例将会是我们的 WSGI 应用程序。
2. 接下来，我们创建一个该类的实例，第一个参数是应用模块或者包的名称。 如果你使用单一的模块（如本例），你应该使用 *__name__* ，因为模块的名称将会因其作为单独应用启动还是作为模块导入而有不同(也即是 `'__main__'` 或实际的导入名)。这是必须的，这样 Flask 才知道到哪去找模板、静态文件等等。详情见 `Flask` 的文档。
3. 然后，我们使用 `route()` 装饰器告诉 Flask 什么样的 URL 能触发我们的函数。
4. 这个函数的名字也在生成 URL 时被特定的函数采用，这个函数返回我们想要显示在用户浏览器中的信息。
5. 最后我们用 `run()` 函数来让应用运行在本地服务器上。 其中 `if __name__ =='__main__':` 确保服务器只会在该脚本被 Python 解释器直接执行的时候才会运行，而不是作为模块导入的时候。

欲关闭服务器，按 Ctrl+C。

外部可访问的服务器

如果你运行了这个服务器，你会发现它只能从你自己的计算机上访问，网络中其它任何的地方都不能访问。在调试模式下，用户可以在你的计算机上执行任意 Python 代码。因此，这个行为是默认的。

如果你禁用了 *debug* 或信任你所在网络的用户，你可以简单修改调用 `run()` 的方法使你的服务器公开可用，如下:

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

这会让操作系统监听所有公网 IP。

2.2 调试模式

虽然 `run()` 方法适用于启动本地的开发服务器，但是你每次修改代码后都要手动重启它。这样并不够优雅，而且 Flask 可以做到更好。如果你启用了调试支持，服务器会在代码修改后自动重新载入，并在发生错误时提供一个相当有用的调试器。

有两种途径来启用调试模式。一种是直接在应用对象上设置:

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

另一种是作为 run 方法的一个参数传入:

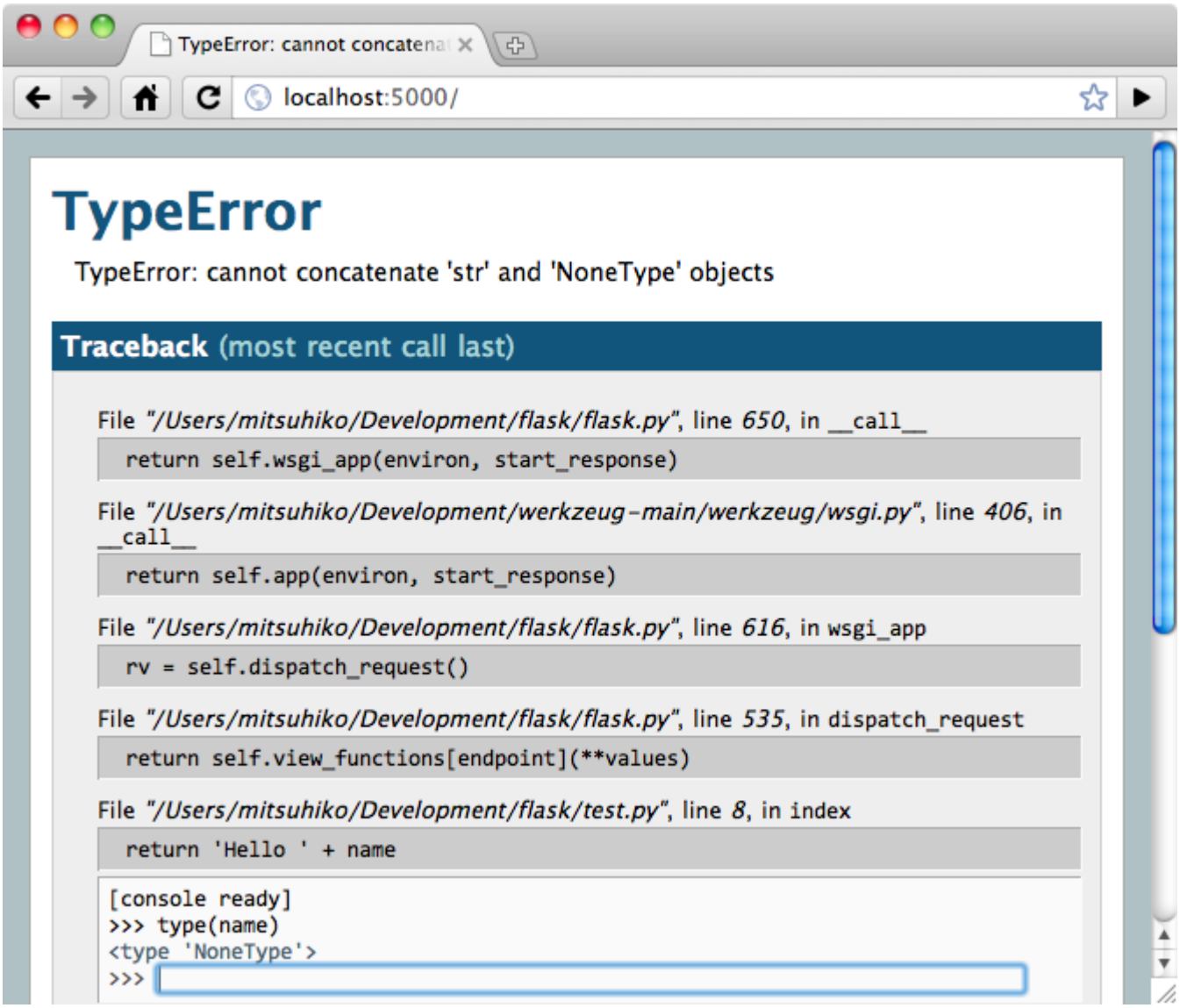
```
app.run(debug=True)
```

两种方法的效果完全相同。

注意

尽管交互式调试器在允许 fork 的环境中无法正常使用（也即在生产服务器上正常使用几乎是不可能的），但它依然允许执行任意代码。这使它成为一个巨大的安全隐患，因此它 **绝对不能用于生产环境** 。

运行中的调试器截图：



想用其它的调试器？ 参见 *调试器操作* 。

2.3 路由

现代 Web 应用的 URL 十分优雅，易于人们辨识记忆，这一点对于那些面向使用低速网络连接移动设备访问的应用特别有用。如果可以不访问索引页，而是直接访问想要的那个页面，他们多半会笑逐颜开而再度光顾。

如上所见， `route()` 装饰器把一个函数绑定到对应的 URL 上。

这里是一些基本的例子:

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


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

但是，不仅如此！你可以构造含有动态部分的 URL，也可以在一个函数上附着多个规则。

2.3.1 变量规则

要给 URL 添加变量部分，你可以把这些特殊的字段标记为 `<variable_name>` ， 这个部分将会作为命名参数传递到你的函数。规则可以用 `<converter:variable_name>` 指定一个可选的转换器。这里有一些不错的例子:

```
@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
```

转换器有下面几种：

int	接受整数
float	同 *int*，但是接受浮点数
path	和默认的相似，但也接受斜线

唯一 URL / 重定向行为

Flask 的 URL 规则基于 Werkzeug 的路由模块。这个模块背后的思想是基于 Apache 以及更早的 HTTP 服务器主张的先例，保证优雅且唯一的 URL。

以这两个规则为例：

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

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

虽然它们看起来着实相似，但它们结尾斜线的使用在 URL *定义* 中不同。第一种情况中，指向 *projects* 的规范 URL 尾端有一个斜线。这种感觉很

像在文件系统中的文件夹。访问一个结尾不带斜线的 URL 会被 Flask 重定向到带斜线的规范 URL 去。

然而，第二种情况的 URL 结尾不带斜线，类似 UNIX-like 系统下的文件的路径名。访问结尾带斜线的 URL 会产生一个 404 “Not Found” 错误。

这个行为使得在遗忘尾斜线时，允许关联的 URL 接任工作，与 Apache 和其它的服务器的行为并无二异。此外，也保证了 URL 的唯一，有助于避免搜索引擎索引同一个页面两次。

2.3.2 构造 URL

如果 Flask 能匹配 URL，那么 Flask 可以生成它们吗？当然可以。你可以用`url_for()`来给指定的函数构造 URL。它接受函数名作为第一个参数，也接受对应 URL 规则的变量部分的命名参数。未知变量部分会添加到 URL 末尾作为查询参数。这里有一些例子:

```
>>> 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
```

（这里也用到了 `test_request_context()` 方法，下面会解释。即使我们正在通过 Python 的 shell 进行交互，它依然会告诉 Flask 要表现为正在处理一个请求。请看下面的解释。 *环境局部变量* ）

为什么你要构建 URL 而非在模板中硬编码？这里有三个绝妙的理由：

1. 反向构建通常比硬编码的描述性更好。更重要的是，它允许你一次性修改 URL， 而不是到处边找边改。
2. URL 构建会转义特殊字符和 Unicode 数据，免去你很多麻烦。
3. 如果你的应用不位于 URL 的根路径（比如，在 `/myapplication` 下，而不是 `/` ）， `url_for()` 会妥善处理这个问题。

2.3.3HTTP 方法

HTTP （与 Web 应用会话的协议）有许多不同的访问 URL 方法。默认情况下，路由只回应 *GET* 请求，但是通过 `route()` 装饰器传递 *methods* 参数可以改变这个行为。这里有一些例子：

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

如果存在 *GET*，那么也会替你自动地添加 *HEAD*，无需干预。它会确保遵照 HTTP RFC（描述 HTTP 协议的文档）处理 *HEAD* 请求，所以你可以完全忽略这部分的 HTTP 规范。同样，自从 Flask 0.6 起， 也实现了 *OPTIONS* 的自动处理。

你不知道一个 HTTP 方法是什么？不必担心，这里会简要介绍 HTTP 方法和它们为什么重要：

HTTP 方法（也经常被叫做“谓词”）告知服务器，客户端想对请求的页面 *做* 些什么。下面的都是非常常见的方法：

GET

浏览器告知服务器：只 *获取* 页面上的信息并发给我。这是最常用的方法。

HEAD

浏览器告诉服务器：欲获取信息，但是只关心 *消息头* 。应用应像处理 *GET* 请求一样来处理它，但是不分发实际内容。在 Flask 中你完全无需 人工 干预，底层的 Werkzeug 库已经替你打点好了。

POST

浏览器告诉服务器：想在 URL 上 *发布* 新信息。并且，服务器必须确保 数据已存储且仅存储一次。这是 HTML 表单通常发送数据到服务器的方法。

PUT

类似 *POST* 但是服务器可能触发了存储过程多次，多次覆盖掉旧值。你可 能会问这有什么用，当然这是有原因的。考虑到传输中连接可能会丢失，在 这种 情况下浏览器和服务器之间的系统可能安全地第二次接收请求，而 不破坏其它东西。因为 *POST* 它只触发一次，所以用 *POST* 是不可能的。

DELETE

删除给定位置的信息。

OPTIONS

给客户端提供一个敏捷的途径来弄清这个 URL 支持哪些 HTTP 方法。 从 Flask 0.6 开始，实现了自动处理。

有趣的是，在 HTML4 和 XHTML1 中，表单只能以 *GET* 和 *POST* 方法提交到服务器。但是 JavaScript 和未来的 HTML 标准允许你使用其它所有的方法。此外，HTTP 最近变得相当流行，浏览器不再是唯一的 HTTP 客户端。比如，许多版本控制系统就在使用 HTTP。

2.4 静态文件

动态 web 应用也会需要静态文件，通常是 CSS 和 JavaScript 文件。理想状况下， 你已经配置好 Web 服务器来提供静态文件，但是在开发中，Flask 也可以做到。 只要在你的包中或是模块的所在目录中创建一个名为 *static* 的文件夹，在应用中使用 */static* 即可访问。

给静态文件生成 URL ，使用特殊的 `'static'` 端点名:

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

这个文件应该存储在文件系统上的 `static/style.css` 。

2.5 模板渲染

用 Python 生成 HTML 十分无趣，而且相当繁琐，因为你必须手动对 HTML 做转义来保证应用的安全。为此，Flask 配备了 Jinja2 模板引擎。

你可以使用 `render_template()` 方法来渲染模板。你需要做的一切就是将模板名和你想作为关键字的参数传入模板的变量。这里有一个展示如何渲染模板的简例:

```
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* 文件夹里寻找模板。所以，如果你的应用是个模块，这个文件夹应该与模块同级；如果它是一个包，那么这个文件夹作为包的子目录:

情况 1: 模块:

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

情况 2: 包:

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

关于模板，你可以发挥 Jinja2 模板的全部实例。更多信息请见 Jinja2 模板文档 。

这里有一个模板实例：

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

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

模板继承让模板用起来相当顺手。如欲了解继承的工作机理，请跳转到 *模板继承* 模式的文档。最起码，模板继承能使特定元素 （比如页眉、导航栏和页脚）可以出现在所有的页面。

自动转义功能默认是开启的，所以如果 *name* 包含 HTML ，它将会被自动转义。如果你能信任一个变量，并且你知道它是安全的（例如一个模块把 Wiki 标记转换为 HTML），你可以用 `Markup` 类或 `|safe` 过滤器在模板中把它标记为安全的。在 Jinja 2 文档中，你会看到更多的例子。

这里是一个 `Markup` 类如何使用的简单介绍:

```
>>> 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'
```

在 0.5 版更改: 自动转义不再在所有模板中启用。下列扩展名的模板会触发自动转义： `.html` 、 `.htm` 、 `.xml` 、 `.xhtml` 。从字符串加载的模板会禁用自动转义。

[1]	不确定 g 对象是什么？它允许你按需存储信息， 查看（ g ）对象的文档和 *在 Flask 中使用 SQLite 3* 的文档以获取更多信息。

2.6 访问请求数据

对于 Web 应用，与客户端发送给服务器的数据交互至关重要。在 Flask 中由全局的 `request` 对象来提供这些信息。如果你有一定的 Python 经验，你会好奇，为什么这个对象是全局的，为什么 Flask 还能保证线程安全。答案是环境作用域：

2.6.1 环境局部变量

内幕

如果你想理解其工作机制及如何利用环境局部变量实现自动化测试，请阅读此节，否则可跳过。

Flask 中的某些对象是全局对象，但却不是通常的那种。这些对象实际上是特定环境的局部对象的代理。虽然很拗口，但实际上很容易理解。

想象一下处理线程的环境。一个请求传入，Web 服务器决定生成一个新线程（ 或者别的什么东西，只要这个底层的对象可以胜任并发系统，而不仅仅是线程）。 当 Flask 开始它内部的请求处理时，它认定当前线程是活动的环境，并绑定当前的应用和 WSGI 环境到那个环境上（线程）。它的实现很巧妙，能保证一个应用调用另一个应用时不会出现问题。

所以，这对你来说意味着什么？除非你要做类似单元测试的东西，否则你基本上可以完全无视它。你会发现依赖于一段请求对象的代码，因没有请求对象无法正常运行。解决方案是，自行创建一个请求对象并且把它绑定到环境中。单元测试的最简单的解决方案是：

用 `test_request_context()` 环境管理器。结合 *with* 声明，绑定一个测试请求，这样你才能与之交互。下面是一个例子：

```
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'
```

另一种可能是：传递整个 WSGI 环境给 `request_context()` 方法：

```
from flask import request


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

2.6.2 请求对象

API 章节对请求对象作了详尽阐述（参见 `request` ），因此这里不会赘述。此处宽泛介绍一些最常用的操作。首先从 *flask* 模块里导入它：

```
from flask import request
```

当前请求的 HTTP 方法可通过 `method` 属性来访问。通过:attr:*~flask.request.form* 属性来访问表单数据（*POST* 或 *PUT* 请求提交的数据）。这里有一个用到上面提到的那两个属性的完整实例:

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

当访问 *form* 属性中的不存在的键会发生什么？会抛出一个特殊的 `KeyError` 异常。你可以像捕获标准的 `KeyError` 一样来捕获它。 如果你不这么做，它会显示一个 HTTP 400 Bad Request 错误页面。所以，多数情况下你并不需要干预这个行为。

你可以通过 `args` 属性来访问 URL 中提交的参数 （`?key=value`）:

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

我们推荐用 *get* 来访问 URL 参数或捕获 *KeyError*，因为用户可能会修改 URL，向他们展现一个 400 bad request 页面会影响用户体验。

欲获取请求对象的完整方法和属性清单，请参阅 `request` 的文档。

2.6.3 文件上传

用 Flask 处理文件上传很简单。只要确保你没忘记在 HTML 表单中设置 `enctype="multipart/form-data"` 属性，不然你的浏览器根本不会发送文件。

已上传的文件存储在内存或是文件系统中一个临时的位置。你可以通过请求对象的 `files` 属性访问它们。每个上传的文件都会存储在这个字典里。它表现近乎为一个标准的 Python `file` 对象，但它还有一个 `save()` 方法，这个方法允许你把文件保存到服务器的文件系统上。这里是一个用它保存文件的例子:

```
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')
    ...
```

如果你想知道上传前文件在客户端的文件名是什么，你可以访问 `filename` 属性。但请记住， 永远不要信任这个值，这个值是可以伪造

的。如果你要把文件按客户端提供的文件名存储在服务器上，那么请把它传递给 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))
    ...
```

一些更好的例子，见 *上传文件* 模式。

2.6.4Cookies

你可以通过 `cookies` 属性来访问 Cookies，用响应对象的 `set_cookie` 方法来设置 Cookies。请求对象的 `cookies` 属性是一个内容为客户端提交的所有 Cookies 的字典。如果你想使用会话，请不要直接使用 Cookies，请参考 *会话* 一节。在 Flask 中，已经注意处理了一些 Cookies 安全细节。

读取 cookies:

```
from flask import request

@app.route('/')
```

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

存储 cookies:

```
from flask import make_response

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

可注意到的是，Cookies 是设置在响应对象上的。由于通常视图函数只是返回字符串，之后 Flask 将字符串转换为响应对象。如果你要显式地转换，你可以使用`make_response()`函数然后再进行修改。

有时候你想设置 Cookie，但响应对象不能醋在。这可以利用 *延迟请求回调* 模式实现。

为此，也可以阅读 *关于响应* 。

2.7 重定向和错误

你可以用`redirect()` 函数把用户重定向到其它地方。放弃请求并返回错误代码，用`abort()` 函数。这里是一个它们如何使用的例子:

```
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()
```

这是一个相当无意义的例子因为用户会从主页重定向到一个不能访问的页面 （401 意味着禁止访问），但是它展示了重定向是如何工作的。

默认情况下，错误代码会显示一个黑白的错误页面。如果你要定制错误页面， 可以使用 `errorhandler()` 装饰器:

```
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()` 调用之后的 `404` 。这告诉 Flask，该页的错误代码是 404 ，即没有找到。默认为 200，也就是一切正常。

2.8 关于响应

视图函数的返回值会被自动转换为一个响应对象。如果返回值是一个字符串， 它被转换为该字符串为主体的、状态码为 `200 OK``的 、 MIME 类型是 ``text/html` 的响应对象。Flask 把返回值转换为响应对象的逻辑是这样：

1. 如果返回的是一个合法的响应对象，它会从视图直接返回。
2. 如果返回的是一个字符串，响应对象会用字符串数据和默认参数创建。
3. 如果返回的是一个元组，且元组中的元素可以提供额外的信息。这样的元组必须是 `(response, status, headers)` 的形式，且至少包含一个元素。 *status* 值会覆盖状态代码， *headers* 可以是一个列表或字典，作为额外的消息标头值。
4. 如果上述条件均不满足， Flask 会假设返回值是一个合法的 WSGI 应用程序，并转换为一个请求对象。

如果你想在视图里操纵上述步骤结果的响应对象，可以使用 `make_response()` 函数。

譬如你有这样一个视图:

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

你只需要把返回值表达式传递给 `make_response()`，获取结果对象并修改，然后再返回它:

```
@app.errorhandler(404)
def not_found(error):
    resp = make_response(render_template('error.html'), 404)
    resp.headers['X-Something'] = 'A value'
    return resp
```

2.9 会话

除请求对象之外，还有一个 `session` 对象。它允许你在不同请求间存储特定用户的信息。它是在 Cookies 的基础上实现的，并且对 Cookies 进行密钥签名。这意味着用户可以查看你 Cookie 的内容，但却不能修改它，除非用户知道签名的密钥。

要使用会话，你需要设置一个密钥。这里介绍会话如何工作:

```
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():
    # remove the username from the session if it's there
    session.pop('username', None)
    return redirect(url_for('index'))

# set the secret key.  keep this really secret:
app.secret_key = 'A0Zr98j/3yX R~XHH!jmN]LWX/,?RT'
```

这里提到的 **escape()** 可以在你模板引擎外做转义（如同本例）。

如何生成强壮的密钥

随机的问题在于很难判断什么是真随机。一个密钥应该足够随机。你的操作系统可以基于一个密钥随机生成器来生成漂亮的随机值，这个值可以用来做密钥:

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

把这个值复制粘贴进你的代码中，你就有了密钥。

使用基于 cookie 的会话需注意: Flask 会将你放进会话对象的值序列化至 Cookies。如果你发现某些值在请求之间并没有持久存在，然而确实已经启用了 Cookies，但也没有得到明确的错误信息。这时，请检查你的页面响应中的 Cookies 的大小，并与 Web 浏览器所支持的大小对比。

2.10 消息闪现

反馈，是良好的应用和用户界面的重要构成。如果用户得不到足够的反馈，他们很可能开始厌恶这个应用。 Flask 提供了消息闪现系统，可以简单地给用户反馈。 消息闪现系统通常会在请求结束时记录信息，并在下一个（且仅在下一个）请求中访问记录的信息。展现这些消息通常结合要模板布局。

使用 `flash()` 方法可以闪现一条消息。要操作消息本身，请使用 `get_flashed_messages()` 函数，并且在模板中也可以使用。完整的例子见 *消息闪现* 部分。

2.11 日志记录

0.3 新版功能.

有时候你会处于这样一种境地，你处理的数据本应该是正确的，但实际上不是。 比如，你会有一些向服务器发送请求的客户端代码，但请求显然是畸形的。这可能是用户篡改了数据，或是客户端代码的粗制滥造。大多数情况下，正常地返回 `400 Bad Request` 就可以了，但是有时候不能这么做，并且要让代码继续运行。

你可能依然想要记录下，是什么不对劲。这时日志记录就派上了用场。从 Flask 0.3 开始，Flask 就已经预置了日志系统。

这里有一些调用日志记录的例子:

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

附带的 `logger` 是一个标准日志类 `Logger`，所以更多信息请查阅 logging 的文档。

2.12 整合 WSGI 中间件

如果你想给你的应用添加 WSGI 中间件，你可以封装内部 WSGI 应用。例如若是你想用 Werkzeug 包中的某个中间件来应付 lighttpd 中的 bugs，可以这样做:

```
from werkzeug.contrib.fixers import LighttpdCGIRootFix

app.wsgi_app = LighttpdCGIRootFix(app.wsgi_app)
```

2.13 部署到 Web 服务器

准备好部署你的 Flask 应用了？你可以立即部署到托管平台来圆满完成快速入门，以下厂商均向小项目提供免费的方案:

- 在 Heroku 上部署 Flask
- 在 dotCloud 上部署 Flask 附 Flask 的具体说明

托管 Flask 应用的其它选择:

- 在 Webfaction 上部署 Flask
- 在 Google App Engine 上部署 Flask
- 用 Localtunnel 共享你的本地服务器

如果你有自己的主机，并且准备自己托管，参见 *部署选择* 章节。

三、教程

想要用 Python 和 Flask 开发一个应用？在此，你将有机会通过实例来学习。 在本教程中，我们会创建一个简单的微博客应用。它只支持单用户和纯文本条目，并且没有推送或评论功能，但是它仍然有你需要开始的一切。我们将使用 Flask ，采用 Python 自带的 SQLite 数据库，所以你不需要其它的东西。

如果你想预先拿到完整源码或是用于对照，请查看 示例源码 。

3.1 介绍 Flaskr

在本教程中，我们把我们的这个博客应用称为 flaskr ，也可以选一个不那么 web 2.0 的名字 ;) 。基本上，我们希望它能做这些事情：

1. 允许用户用配置文件里指定的凭证登入登出。只支持一个用户。

2. 当用户登入后，可以向页面添加条目。条目标题是纯文本，正文可以是一些 HTML 。因信任这里的用户，这部分 HTML 不做审查。
3. 页面倒序显示所有条目（后来居上），并且用户登入后可以在此添加新条目。

我们将会在应用中直接采用 SQLite3 ，因为它足以应付这种规模的应用。对于更大型的应用，就有必要使用 SQLAlchemy，它能更加智能地处理数据库连接、允许你一次连接不同的关系数据库等等。如果你的数据更适合 NoSQL，你也可以考虑流行的 NoSQL 数据库，。

这里是一个应用最终效果的截图:

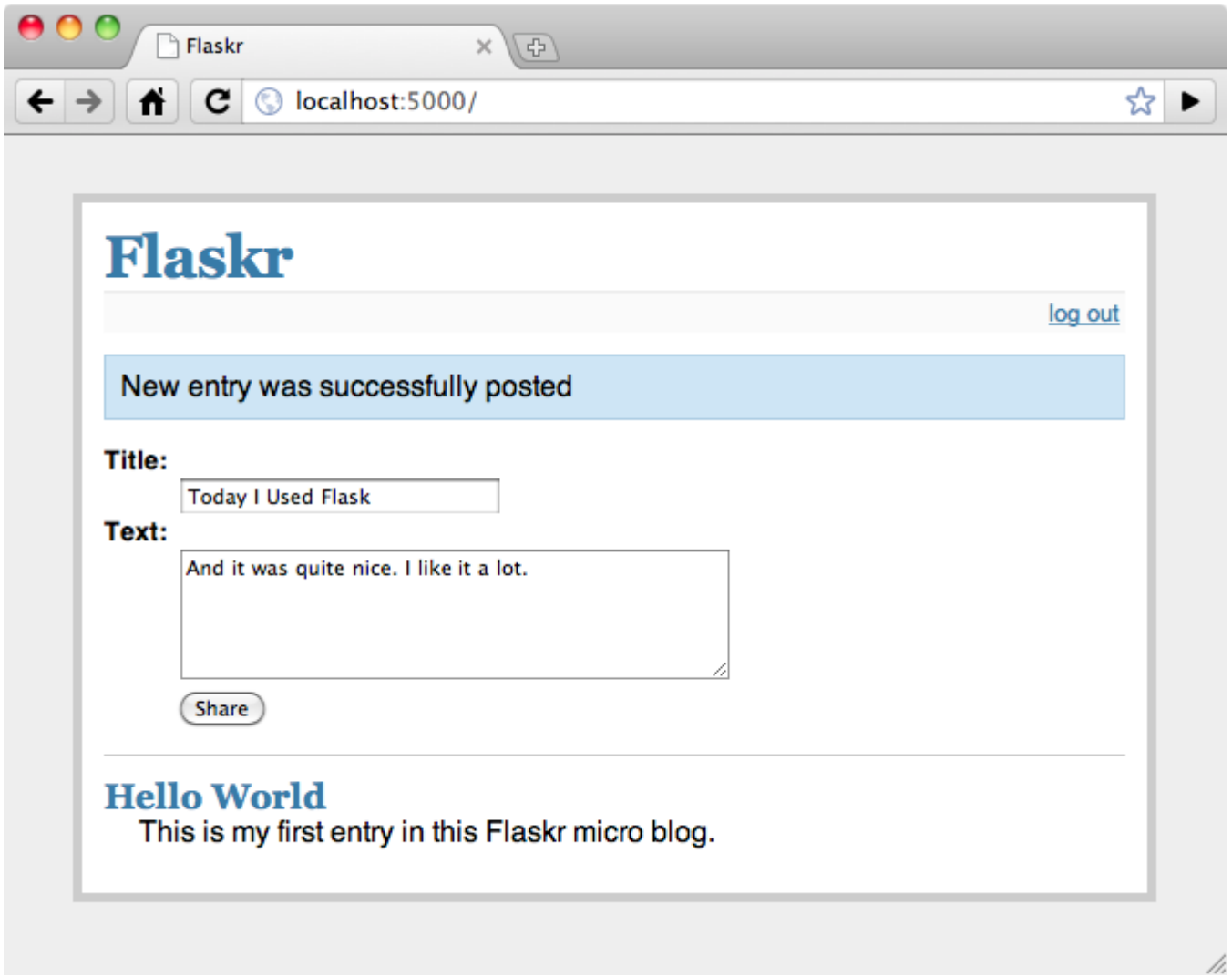
3.2 步骤 0: 创建文件夹

在我们真正开始之前，让我们创建这个应用所需的文件夹:

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

flaskr 文件夹不是一个 Python 包，只是个我们放置文件的地方。在接下来的步骤中，我们会直接把数据库模式和主模块放在这个目录中。 用户可以通过 *HTTP* 访问 *static* 文件夹中的文件，也即存放 css 和 javascript 文件的地方。Flask 会在 *templates* 文件夹里寻找 Jinja2 模板，之后教程中创建的模板将会放在这个文件夹里。

阅读 *步骤 1: 数据库模式* 以继续。



阅读 *步骤 0：创建文件夹* 以继续。

3.3 步骤 1：数据库模式

首先我们要创建数据库模式。对于这个应用来说，一张表就足够了，而且只需支持 SQLite，所以会很简单。只需要把下面的内容放进一个名为 *schema.sql* 的文件，放在刚才创建的 *flaskr* 文件夹中：

```
drop table if exists entries;
create table entries (
```

```
  id integer primary key autoincrement,
  title string not null,
  text string not null
);
```

这个模式包含一个名为 *entries* 的表，该表中的每行都包含一个 *id* 、一个 *title* 和一个 *text* 。 *id* 是一个自增的整数，也是主键；其余的两个是字符串，且不允许为空。

阅读 *步骤 2：应用设置代码* 以继续。

3.4 步骤 2: 应用设置代码

现在我们已经有了数据库模式，我们可以创建应用的模块了。让我们把它叫做 *flaskr.py*，并放置在 *flaskr* 目录下。我们从添加所需的导入语句和添加配置部分开始。对于小型应用，可以直接把配置放在主模块里，正如我们现在要做的一样。但更简洁的方案是创建独立的 *.ini* 或 *.py* 文件，并载入或导入里面的值。

首先在 *flaskr.py* 里导入:

```
# all the imports
import os
import sqlite3
from flask import Flask, request, session, g, redirect, url_for, abort, \
     render_template, flash
```

`Config` 对象的用法如同字典，所以我们可以用新值更新它。

数据库路径

操作系统有进程当前工作目录的概念。不幸的是，你在 Web 应用中不能依赖此概念，因为你可能会在相同的进程中运行多个应用。

为此，提供了 `app.root_path` 属性以获取应用的路径。配合 `os.path` 模块使用，轻松可达任意文件。在本例中，我们把数据库放在根目录下。

对于实际生产环境的应用，推荐使用 *实例文件夹* 。

通常，加载一个单独的、环境特定的配置文件是个好主意。Flask 允许你导入多份配置，并且使用最后的导入中定义的设置。这使得配置设定过程更可靠。 `from_envvar()` 可用于达此目的。

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

只需设置一个名为 `FLASKR_SETTINGS` 的环境变量，指向要加载的配置文件。启用静默模式告诉 Flask 在没有设置该环境变量的情况下噤声。

此外，你可以使用配置对象上的 `from_object()` 方法，并传递一个模块的导入名作为参数。Flask 会从这个模块初始化变量。 注意，只有名称全为大写字母的变量才会被采用。

secret_key 是保证客户端会话的安全的要点。正确选择一个尽可能难猜测、尽可能复杂的密钥。调试标志关系交互式调试器的开启。 *永远不要在生产系统中激活调试模式* ，因为它将允许用户在服务器上执行代码。

我们还添加了一个让连接到指定数据库变得很简单的方法，这个方法用于在请求时开启一个数据库连接，并且在交互式 Python shell 和脚本中也能使用。这为以后的操作提供了相当的便利。我们创建了一个简单的 SQLite 数据库的连接，并让它用 `sqlite3.Row` 表示数据库中的行。这使得我们可以通过字典而不是元组的形式访问行:

```
def connect_db():
    """Connects to the specific database."""
    rv = sqlite3.connect(app.config['DATABASE'])
    rv.row_factory = sqlite3.Row
    return rv
```

最后，如果我们想要把这个文件当做独立应用来运行，我们只需在可启动服务器文件的末尾添加这一行:

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

如此我们便可以开始顺利运行这个应用，使用如下命令:

```
python flaskr.py
```

你将会看见有消息告诉你访问该服务器的地址。

当你在浏览器中访问服务器遇到一个 404 page not found 错误时， 是因为我们还没有任何视图。我们之后再来关注视图。首先我们应该让数据库工作起来。

外部可见的服务器

想要你的服务器公开可见？

> *外部可见的服务器* 一节有更多信息。

阅读 *步骤 4: 创建数据库* 以继续。

3.5 步骤 3: 数据库连接

我们已经创建了一个能建立数据库连接的函数 *connect_db*，但它本身并不是很有用。总是创建或关闭数据库连接是相当低效的，所以我们会让连接保持更长时间。因为数据库连接封装了事务，我们也需要确保同一时刻只有一个请求使用这个连接。那么，如何用 Flask 优雅地实现呢？

这该是应用环境上场的时候了。那么，让我们开始吧。

Flask 提供了两种环境（Context）：应用环境（Application Context）和请求环境（Request Context）。暂且你所需了解的是，不同环境有不同的特殊变量。例如 `request` 变量与当前请求的请求对象有关， 而 `g` 是与当前应用环境有关的通用变量。我们在之后会深入了解它们。

现在你只需要知道可以安全地在 `g` 对象存储信息。

那么你何时把数据库连接存放到它上面？你可以写一个辅助函数。这个函数首次调用的时候会为当前环境创建一个数据库连接，调用成功后返回已经建立好的连接:

```
def get_db():
    """Opens a new database connection if there is none yet for the
```

```
    current application context.
    """
    if not hasattr(g, 'sqlite_db'):
        g.sqlite_db = connect_db()
    return g.sqlite_db
```

于是现在我们知道如何连接到数据库，但如何妥善断开连接呢？为此，Flask 提供了 `teardown_appcontext()` 装饰器。它将在每次应用环境销毁时执行:

```
@app.teardown_appcontext
def close_db(error):
    """Closes the database again at the end of the request."""
    if hasattr(g, 'sqlite_db'):
        g.sqlite_db.close()
```

`teardown_appcontext()` 标记的函数会在每次应用环境销毁时调用。这意味着什么？本质上，应用环境在请求传入前创建，每当请求结束时销毁。销毁有两种原因：一切正常（错误参数会是 *None* ）或发生异常，后者情况中，错误会被传递给销毁时函数。

好奇这些环境的意义？阅读 *应用上下文* 文档了解更多。

阅读 *步骤 4: 创建数据库* 以继续。

提示

我该把这些代码放在哪？

如果你一直遵循教程，你应该会问从此以后的步骤产生的代码放在什么地方。逻辑上来讲，应该按照模块来组织函数，即把你新的 `get_db` 和 `close_db` 函数放在之前的 `connect_db` 函数下面（逐行复刻教程）。

如果你需要来找准定位，可以看一下 示例源码 是怎么组织的。 在 Flask 中，你可以把你应用中所有的代码放在一个 Python 模块里。但你无需这么做，而且在你的应用 *规模扩大* 以后，这显然不妥。

3.6 步骤 4: 创建数据库

正如之前介绍的，Flaskr 是一个数据库驱动的应用，更准确的说法是，一个由关系数据库系统驱动的应用。关系数据库系统需要一个模式来决定存储信息的方式。所以在第一次开启服务器之前，要点是创建模式。

可以通过管道把 *schema.sql* 作为 *sqlite3* 命令的输入来创建这个模式，命令为如下:

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

这种方法的缺点是需要安装 sqlite3 命令，而并不是每个系统都有安装。而且你必须提供数据库的路径，否则将报错。用函数来初始化数据库是个不错的想法。

3.7 步骤 5: 视图函数

现在数据库连接已经正常工作，我们终于可以开始写视图函数了。我们一共需要写四个:

3.7.1 显示条目

这个视图显示数据库中存储的所有条目。它绑定在应用的根地址，并从数据库查询出文章的标题和正文。id 值最大的条目（最新的条目）会显示在最上方。从指针返回的行是按 select 语句中声明的列组织的元组。这对像我们这样的小应用已经足够了， 但是你可能会想把它转换成字典。如果你对这方面有兴趣，请参考 *简化查询* 的例子。

视图函数会将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)
```

3.7.2 添加条目

这个视图允许已登入的用户添加新条目，并只响应 *POST* 请求，实际的表单显示在 *show_entries* 页。如果一切工作正常，我们会用 `flash()` 向下一次请求发送提示消息，并重定向回 *show_entries* 页:

```
@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'))
```

注意这里的用户登入检查（ *logged_in* 键在会话中存在，并且为 *True* ）

安全提示

确保像上面例子中一样，使用问号标记来构建 SQL 语句。否则，当你使用格式化字符串构建 SQL 语句时，你的应用容易遭受 SQL 注入。 更多请见 *在 Flask 中使用 SQLite 3* 。

3.7.3 登入和登出

这些函数用来让用户登入登出。登入通过与配置文件中的数据比较检查用户名和密码， 并设定会话中的 *logged_in* 键值。如果用户成功登入，那么这个键值会被设为 *True* ，并跳转回 *show_entries* 页。此外，会有消息闪现来提示用户登入成功。 如果发生一个错误，模板会通知，并提示重新登录。

```
@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)
```

登出函数，做相反的事情，从会话中删除 *logged_in* 键。我们这里使用了一个简洁的方法：如果你使用字典的 `pop()` 方法并传入第二个参数（默认），这个方法会从字典中删除这个键，如果这个键不存在则什么都不做。这很有用，因为我们不需要检查用户是否已经登入。

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

继续 *步骤 6：模板* 。

要这么做，我们可以创建一个名为 *init_db* 的函数来初始化数据库。 让我们首先看看代码。只需要把这个函数放在 *flaskr.py* 里的 *connect_db* 函数的后面:

```
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()
```

那么，这段代码会发生什么？还记得吗？上个章节中提到，应用环境在每次请求传入时创建。这里我们并没有请求，所以我们需要手动创建一个应用环境。 `g` 在应用环境外无法获知它属于哪个应用，因为可能会有多个应用同时存在。

`with app.app_context()` 语句为我们建立了应用环境。在 with 语句的内部， `g` 对象会与 `app` 关联。在语句的结束处，会释放这个关联兵执行所有销毁函数。这意味着数据库连接在提交后断开。

应用对象的 `open_resource()` 方法是一个很方便的辅助函数，可以打开应用提供的资源。这个函数从资源所在位置（ 你的 *flaskr* 文件夹）打开文件，并允许你读取它。我们在此用它来在数据库连接上执行脚本。

SQLite 的数据库连接对象提供了一个游标对象。游标上有一个方法可以执行完整的脚本。最后我们只需提交变更。SQLite 3 和其它支持事务的数据库只会在你显式提交的时候提交。

现在可以在 Python shell 导入并调用这个函数来创建数据库:

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

故障排除

如果你遇到了表无法找到的异常，请检查你是否确实调用过 *init_db* 函数并且表的名称是正确的（比如弄混了单数和复数）。

阅读 *步骤 5: 视图函数* 以继续。

3.8 步骤 6: 模板

接下来我们应该创建模板了。如果我们现在请求 URL，只会得到 Flask 无法找到模板的异常。 模板使用 Jinja2 语法并默认开启自动转义。这意味着除非你使用 `Markup` 标记或在模板中使用 `|safe` 过滤器，否则 Jinja 2 会确保特殊字符，比如 `<` 或 `>` 被转义为等价的 XML 实体。

我们也会使用模板继承在网站的所有页面中重用布局。

将下面的模板放在 *templates* 文件夹里:

3.8.1layout.html

这个模板包含 HTML 主体结构、标题和一个登入链接（用户已登入则提供登出）。 如果有，它也会显示闪现消息。 `{% block body %}` 块可以被子模板中相同名字的块（ `body`）替换。

`session` 字典在模板中也是可用的。你可以用它来检查用户是否已登入。注意，在 Jinja 中你可以访问不存在的对象/字典属性或成员。比如下面的代码， 即便 `'logged_in'` 键不存在，仍然可以正常工作:

```
<!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>
```

3.8.2show_entries.html

这个模板继承了上面的 *layout.html* 模板来显示消息。注意 *for* 循环会遍历并输出所有 `render_template()` 函数传入的消息。我们还告诉表单使用 *HTTP* 的 *POST* 方法提交信息到 *add_entry* 函数:

```
{% 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 %}
```

3.8.3login.html

最后是登入模板，只是简单地显示一个允许用户登入的表单:

```
{% 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 %}
```

继续 *步骤 7: 添加样式*。

3.9 步骤 7: 添加样式

现在其它的一切都可以正常工作，是时候给应用添加样式了。只需在之前创建的 *static* 文件夹中创建一个名为 *style.css* 的样式表:

```
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; }
```

继续 *福利: 应用测试* 。

3.10 福利: 应用测试

现在你应该完成你的应用，并且一切都按预期运转正常，对于简化未来的修改，添加自动测试不是一个坏主意。上面的应用将作为文档中 *测试 Flask 应用* 节的例子来演示如何进行单元测试。去看看测试 Flask 应用是多么简单的一件事。

四、模板

Flask 使用 Jinja 2 作为模板引擎。当然，你也可以自由使用其它的模板引擎，但运行 Flask 本身仍然需要 Jinja2 依赖 ，这对启用富扩展是必要的，扩展可以依赖 Jinja2 存在。

本节只是快速地介绍 Jinja2 是如何集成到 Flask 中的。更多关于 Jinja2 语法本身的信息， 请参考官方文档 Jinja2 模板引擎 。

4.1Jinja 配置

Jinja 2 默认配置如下:

- 所有扩展名为 `.html` 、 `.htm` 、 `.xml` 以及 `.xhtml` 的模板会开启自动转义
- 模板可以利用 `{% autoescape %}` 标签选择自动转义的开关。
- Flask 在 Jinja2 上下文中插入了几个全局函数和助手，另外还有一些目前默认的值

4.2 标准上下文

下面的全局变量默认在 Jinja2 模板中可用:

config

当前的配置对象（`flask.config`）

0.6 新版功能.

在 0.10 版更改: 现在这总是可用的，甚至在导入的模版里。

request

当前的请求对象 (`flask.request`)。当模版不是在活动的请求上下文中渲染时这个变量不可用。

session

当前的会话对象 (`flask.session`)。当模版不是在活动的请求上下文中渲染时这个变量不可用。

g

请求相关的全局变量 (`flask.g`)。当模版不是在活动的请求上下文中渲染时这个变量不可用。

url_for()

`flask.url_for()` 函数

get_flashed_messages()

`flask.get_flashed_messages()` 函数

Jinja 上下文行为

这些变量被添加到了请求的上下文中，而非全局变量。区别在于，他们默认不会在导入模板的上下文中出现。这样做，一方面是考虑到性能，另一方面是为了让事情显式透明。

这对你来说意味着什么？如果你想要导入一个需要访问请求对象的宏，有两种可能的方法:

1. 显式地传入请求或请求对象的属性作为宏的参数。
2. 与上下文一起（with context）导入宏。

与上下文中一起（with context）导入的方式如下:

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

4.3 标准过滤器

这些过滤器在 Jinja2 中可用，也是 Jinja2 自带的过滤器:

tojson()

这个函数把给定的对象转换为 JSON 表示，如果你要动态生成 JavaScript 这里有一个非常有用的例子。

注意 *script* 标签里的东西不应该被转义，因此如果你想在 *script* 标签里使用它， 请使用 `|safe` 来禁用转义，:

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

4.4 控制自动转义

自动转义的概念是自动转义特殊字符。 HTML （或 XML ，因此也有 XHTML ）意义下的特殊字符是 `&` ， `>` ， `<` ， `"` 以及 `'` 。因为这些字符在文档中表示它们特定的含义，如果你想在文本中使用它们，应该把它们替

换成相应的“实体”。不这么做不仅会导致用户疲于在文本中使用这些字符，也会导致安全问题。 （见 *跨站脚本攻击（XSS）*）

虽然你有时会需要在模板中禁用自动转义，比如在页面中显式地插入 HTML ， 可以是一个来自于 markdown 到 HTML 转换器的安全输出。

我们有三种可行的解决方案:

- 在传递到模板之前，用 `Markup` 对象封装 HTML 字符串。一般推荐这个方法。
- 在模板中，使用 `|safe` 过滤器显式地标记一个字符串为安全的 HTML （ `{{myvariable|safe }}` ）。
- 临时地完全禁用自动转义系统。

在模板中禁用自动转义系统，可以使用 `{%autoescape %}` 块:

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

无论何时，都请务必格外小心这里的变量。

4.5 注册过滤器

如果你要在 Jinja2 中注册你自己的过滤器，你有两种方法。你可以把它们手动添加到应用的 `jinja_env` 或者使用 `template_filter()` 装饰器。

下面两个例子作用相同，都是反转一个对象：

```
@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
```

在使用装饰器的情况下，如果你想以函数名作为过滤器名，参数是可选的。注册之后，你可以在模板中像使用 Jinja2 内置过滤器一样使用你的过滤器，例如你在上下文中有一个名为 *mylist* 的 Python 列表：

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

4.6 上下文处理器

Flask 上下文处理器自动向模板的上下文中插入新变量。上下文处理器在模板渲染之前运行，并且可以在模板上下文中插入新值。上下文处理器是一个返回字典的函数，这个字典的键值最终将传入应用中所有模板的上下文：

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

上面的上下文处理器使得模板可以使用一个名为 *user* ，值为 *g.user* 的变量。 不过这个例子不是很有意思，因为 *g* 在模板中本来就是可用的，但它解释了上下文处理器是如何工作的。

变量不仅限于值，上下文处理器也可以使某个函数在模板中可用（由于 Python 允许传递函数）：

```
@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)
```

上面的上下文处理器使得 *format_price* 函数在所有模板中可用:

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

你也可以构建 *format_price* 为一个模板过滤器（见 *注册过滤器* ）， 但这展示了上下文处理器传递函数的工作过程。

五、测试 Flask 应用

没有经过测试的东西都是不完整的

这一箴言的起源已经不可考了，尽管他不是完全正确的，但是仍然离真理不远。没有测试过的应用将会使得提高现有代码质量很困难，二不测试应用程序的开发者，会显得特别多疑。如果一个应用拥有自动化测试，那么您就可以安全的修改然后立刻知道是否有错误。

Flask 提供了一种方法用于测试您的应用，那就是将 Werkzeug 测试 `Client` 暴露出来，并且为您操作这些内容的本地上下文变量。然后您就可以将自己最喜欢的测试解决方案应用于其上了。 在这片文档中，我们将会使用 Python 自带的 `unittest` 包。

5.1 应用程序

首先，我们需要一个应用来测试，我们将会使用 *教程* 这里的应用来演示。如果您还没有获取它，请从 *the examples* 这里查找源码。

5.2 测试的大框架

为了测试这个引用，我们添加了第二个模块(*flaskr_tests.py*)， 并且创建了一个框架如下:

```
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()
```

在 **setUp()** 方法的代码创建了一个新的测试客户端并且初始化了一个新的数据库。这个函数将会在每次独立的测试函数运行之前运行。要在测试之后删除这个数据库，我们在 **tearDown()** 函数当中关闭这个文件，并将它从文件系统中删除。同时，在初始化的时候 `TESTING` 配置标志被激活，这将会使得处理请求时的错误捕捉失效，以便于您在进行对应用发出请求的测试时获得更好的错误反馈。

这个测试客户端将会给我们一个通向应用的简单接口，我们可以激发对向应用发送请求的测试，并且此客户端也会帮我们记录 Cookie 的动态。

因为 SQLite3 是基于文件系统的，我们可以很容易的使用临时文件模块来创建一个临时的数据库并初始化它，函数 `mkstemp()` 实际上完成了两件事情：它返回了一个底层的文件指针以及一个随机的文件名，后者我们用作数据库的名字。我们只需要将 *db_fd* 变量保存起来，就可以使用 *os.close* 方法来关闭这个文件。

如果我们运行这套测试，我们应该会得到如下的输出:

```
$ python flaskr_tests.py


----------------------------------------------------------------------
Ran 0 tests in 0.000s


OK
```

虽然现在还未进行任何实际的测试，我们已经可以知道我们的 flaskr 程序没有语法错误了。否则，在 import 的时候就会抛出一个致死的错误了。

5.3 第一个测试

是进行第一个应用功能的测试的时候了。让我们检查当我们访问根路径(/)时应用程序是否正确地返回了了“No entries here so far” 字样。为此，我们添加了一个新的测试函数到我们的类当中， 如下面的代码所示:

```
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.DATABASE)


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

注意到我们的测试函数以 *test* 开头，这允许 **unittest** 模块自动识别出哪些方法是一个测试方法，并且运行它。

通过使用 *self.app.get* 我们可以发送一个 HTTP *GET* 请求给应用的某个给定路径。返回值将会是一个 **response_class** 对象。我们可以使用 **data** 属性来检查程序的返回值(以字符串类型)。在这里，我们检查 `'No entries here so far'` 是不是输出内容的一部分。

再次运行，您应该看到一个测试成功通过了:

```
$ python flaskr_tests.py
.
```

```
----------------------------------------------------------------------
Ran 1 test in 0.034s

OK
```

5.4 登陆和登出

我们应用的大部分功能只允许具有管理员资格的用户访问。所以我们需要一种方法来帮助我们的测试客户端登陆和登出。为此，我们向登陆和登出页面发送一些请求，这些请求都携带了表单数据（用户名和密码），因为登陆和登出页面都会重定向，我们将客户端设置为 *follow_redirects* 。

将如下两个方法加入到您的 *FlaskrTestCase* 类:

```
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)
```

现在我们可以轻松的测试登陆和登出是正常工作还是因认证失败而出错，添加新的测试函数到类中:

```
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
```

5.5 测试消息的添加

我们同时应该测试消息的添加功能是否正常，添加一个新的测试方法如下：

```
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
```

这里我们测试计划的行为是否能够正常工作，即在正文中可以出现 HTML 标签，而在标题中不允许。

运行这个测试，我们应该得到三个通过的测试：

```
$ python flaskr_tests.py
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.332s


OK
```

关于请求的头信息和状态值等更复杂的测试，请参考 MiniTwit Example ，在这个例子的源代码里包含一套更长的测试。

5.6 其他测试技巧

除了如上文演示的使用测试客户端完成测试的方法，也有一个 `test_request_context()`方法可以配合 *with* 语句用于激活一个临时的请求上下文。通过它，您可以访问 `request`、`g` 和 `session` 类的对象，就像在视图中一样。 这里有一个完整的例子示范了这种用法:

```
app = flask.Flask(__name__)


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

所有其他的和上下文绑定的对象都可以使用同样的方法访问。

如果您希望测试应用在不同配置的情况下的表现，这里似乎没有一个很好的方法，考虑使用应用的工厂函数(参考 *应用程序的工厂函数*)

注意，尽管你在使用一个测试用的请求环境，函数 `before_request()` 以及 `after_request()` 都不会自动运行。 然而，`teardown_request()` 函数在测试请求的上下文离开 *with* 块的时候会执行。如果您希望 `before_request()` 函数仍然执行。 您需要手动调用 `preprocess_request()` 方法:

```
app = flask.Flask(__name__)

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

这对于打开数据库连接或者其他类似的操作来说，很可能是必须的，这视您应用的设计方式而定。

如果您希望调用 `after_request()` 函数， 您需要使用 `process_response()` 方法。 这个方法需要您传入一个 response 对象:

```
app = flask.Flask(__name__)

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

这通常不是很有效，因为这时您可以直接转向使用测试客户端。

5.7 伪造资源和上下文

0.10 新版功能.

在应用上下文或 `flask.g` 对象上存储用户认证信息和数据库连接非常常见。一般的模式是在第一次使用对象时，把对象放在应用上下文或 `flask.g` 上面，而在请求销毁时移除对象。试想一下例如下面的获取当前用户的代码:

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

对于测试，这样易于从外部覆盖这个用户，而不用修改代码。连接 `flask.appcontext_pushed` 信号可以很容易地完成这个任务:

```
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
```

并且之后使用它:

```
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)
```

5.8 保存上下文

0.4 新版功能.

有时，激发一个通常的请求，但是将当前的上下文保存更长的时间，以便于附加的内省发生是很有用的。 在 Flask 0.4 中，通过 `test_client()` 函数和 *with* 块的使用可以实现:

```
app = flask.Flask(__name__)


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

如果您仅仅使用 `test_client()` 方法，而不使用 *with* 代码块， *assert* 断言会失败，因为 *request* 不再可访问(因为您试图在非真正请求中时候访问它)。

5.9 访问和修改 Sessions

0.8 新版功能.

有时，在测试客户端里访问和修改 Sesstions 可能会非常有用。 通常有两种方法实现这种需求。如果您仅仅希望确保一个 Session 拥有某个特定的键，且此键的值是某个特定的值，那么您可以只保存起上下文，并且访问 `flask.session`:

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

但是这样做并不能使您修改 Session 或在请求发出之前访问 Session。从 Flask 0.8 开始，我们提供一个叫做“Session 事务”的东西用于模拟适当的调用，从而在测试客户端的上下文中打开一个 Session，并用于修改。在事务的结尾，Session 将被恢复为原来的样子。这些都独立于 Session 的后端使用:

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

```
# once this is reached the session was stored
```

注意到，在此时，您必须使用这个 `sess` 对象而不是调用 `flask.session` 代理，而这个对象本身提供了同样的接口。

六、记录应用错误

0.3 新版功能.

应用故障，服务器故障。早晚你会在产品中看见异常。即使你的代码是 100% 正确的， 你仍然会不时看见异常。为什么？因为涉及的所有一切都会出现故障。这里给出一些完美正确的代码导致服务器错误的情况:

- 客户端在应用读取到达数据时，提前终止请求
- 数据库服务器超载，并无法处理查询
- 文件系统满了
- 硬盘损坏
- 后端服务器超载
- 你所用的库出现程序错误
- 服务器的网络连接或其它系统故障

而且这只是你可能面对的问题的简单情形。那么，我们应该怎么处理这一系列问题？ 默认情况下，如果你的应用在以生产模式运行， Flask 会显示一个非常简单的页面并记录异常到 `logger` 。

但是你还可以做些别的，我们会介绍一些更好的设置来应对错误。

6.1 错误邮件

如果你的应用在生产模式下运行（会在你的服务器上做），默认情况下，你不会看见任何日志消息。为什么会这样？Flask 试图实现一个零配置框架。如果没有配置，日志会存放在哪？猜测不是个好主意，因为它猜测的位置可能不是一个用户有权创建日志文件的地方。而且，对于大多数小型应用，不会有人关注日志。

事实上，我现在向你保证，如果你给应用错误配置一个日志文件，你将永远不会去看它，除非在调试问题时用户向你报告。你需要的应是异常发生时的邮件，然后你会得到一个警报，并做些什么。

Flask 使用 Python 内置的日志系统，而且它确实向你发送你可能需要的错误邮件。 这里给出你如何配置 Flask 日志记录器向你发送报告异常的邮件:

```
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)
```

那么刚刚发生了什么？我们创建了一个新的 `SMTPHandler` 来用监听 `127.0.0.1` 的邮件服务器向所有的 *ADMINS* 发送发件人为 *server-error@example.com*，主题为“YourApplication Failed”的邮件。如果你的邮件服务器需要凭证，这些功能也被提供了。详情请见 `SMTPHandler` 的文档。

我们同样告诉处理程序只发送错误和更重要的消息。因为我们的确不想收到警告或是其它没用的，每次请求处理都会发生的日志邮件。

你在生产环境中运行它之前，请参阅 *控制日志格式* 来向错误邮件中置放更多的信息。这会让你少走弯路。

6.2 记录到文件

即便你收到了邮件，你可能还是想记录警告。当调试问题的时候，收集更多的信息是个好主意。请注意 Flask 核心系统本身不会发出任何警告，所以在古怪的事情发生时发出警告是你的责任。

在日志系统的方框外提供了一些处理程序，但它们对记录基本错误并不是都有用。最让人感兴趣的可能是下面的几个:

- `FileHandler` - 在文件系统上记录日志
- `RotatingFileHandler` - 在文件系统上记录日志， 并且当消息达到一定数目时，会滚动记录

- `NTEventLogHandler` - 记录到 Windows 系统中的系统事件日志。如果你在 Windows 上做开发，这就是你想要用的。
- `SysLogHandler` - 发送日志到 Unix 的系统日志

当你选择了日志处理程序，像前面对 SMTP 处理程序做的那样，只要确保使用一个低级的设置（我推荐 *WARNING* ）：

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

6.3 控制日志格式

默认情况下，错误处理只会把消息字符串记录到文件或邮件发送给你。一个日志记录应存储更多的信息，这使得配置你的日志记录器包含那些信息很重要，如此你会对错误发生的原因，还有更重要的——错误在哪发生，有更好的了解。

格式可以从一个格式化字符串实例化。注意回溯（tracebacks）会被自动加入到日志条目后，你不需要在日志格式的格式化字符串中这么做。

这里有一些配置实例:

6.3.1 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
'''))
```

6.3.2 日志文件

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

6.3.3 复杂日志格式

这里给出一个用于格式化字符串的格式变量列表。注意这个列表并不完整，完整的列表请翻阅 **logging** 包的官方文档。

格式	描述
`%(levelname)s`	消息文本的记录等级 (`'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, `'CRITICAL'`).
`%(pathname)s`	发起日志记录调用的源文件的完整路径（如果可用）
`%(filename)s`	路径中的文件名部分
`%(module)s`	模块（文件名的名称部分）
`%(funcName)s`	包含日志调用的函数名
`%(lineno)d`	日志记录调用所在的源文件行的行号（如果可用）
`%(asctime)s`	*LogRecord* 创建时的人类可读的时间。默认情况下，格式为 `"2003-07-08 16:49:45,896"`（逗号后的数字时间的毫秒部分）。这可以通过继承 :class:~logging.Formatter，并重载 **`formatTime()`** 改变。
`%(message)s`	记录的消息，视为 `msg % args`

如果你想深度定制日志格式，你可以继承 **`Formatter`** 。 **`Formatter`** 有三个需要关注的方法:

`format()`:

处理实际上的格式。需要一个 **`LogRecord`** 对象作为参数，并必须返回一个格式化字符串。

`formatTime()`:

控制 *asctime* 格式。如果你需要不同的时间格式，可以重载这个函数。

`formatException()`

控制异常的格式。需要一个 **`exc_info`** 元组作为参数，并必须返 回一个字符串。默认的通常足够好，你不需要重载它。

更多信息请见其官方文档。

6.4 其它的库

至此，我们只配置了应用自己建立的日志记录器。其它的库也可以记录它们。例如， SQLAlchemy 在它的核心中大量地使用日志。而在 `logging` 包中有一个方法可以一次性配置所有的日志记录器，我不推荐使用它。可能存在一种情况，当你想要在同一个 Python 解释器中并排运行多个独立的应用时，则不可能对它们的日志记录器做不同的设置。

作为替代，我推荐你找出你有兴趣的日志记录器，用 `getLogger()` 函数来获取日志记录器，并且遍历它们来附加处理程序:

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

七、调试应用错误

对于生产应用，按照 *记录应用错误* 中的描述来配置你应用的日志记录和通知。这个章节讲述了调试部署配置和深入一个功能强大的 Python 调试器的要点。

7.1 有疑问时，手动运行

在配置你的应用到生产环境时时遇到了问题？如果你拥有主机的 shell 权限，验证你是否可以在部署环境中手动用 shell 运行你的应用。确保在同一用户账户下运行配置好的部署来解决权限问题。你可以使用 Flask 内置的开发服务器并设置 *debug=True*， 这在捕获配置问题的时候非常有效，但是 **请确保在可控环境下临时地这么做。** 不要在生产环境中使用 *debug=True* 运行。

7.2 调试器操作

为了深入跟踪代码的执行，Flask 提供了一个方框外的调试器（见 *调试模式*）。 如果你想用其它的 Python 调试器，请注意相互的调试器接口。你需要设置下面的参数来使用你中意的调试器:

- `debug` - 是否开启调试模式并捕获异常
- `use_debugger` - 是否使用内部的 Flask 调试器
- `use_reloader` - 是否在异常时重新载入并创建子进程

debug 必须为 True （即异常必须被捕获）来允许其它的两个选项设置为任何值。

如果你使用 Aptana/Eclipse 来调试，你会需要把 `use_debugger` 和 `user_reloader` 都设置为 False 。

一个可能有用的配置模式就是在你的 config.yaml 中设置为如下（当然，自行更改为适用你应用的）：

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

然后在你应用的入口（ main.py ），你可以写入下面的内容:

```
if __name__ == "__main__":
    # To allow aptana to receive errors, set use_debugger=False
    app = create_app(config="config.yaml")

    if app.debug: use_debugger = True
    try:
        # Disable Flask's debugger if external debugger is requested
        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')
```

八、配置处理

0.3 新版功能.

应用会需要某种配置。你可能会需要根据应用环境更改不同的设置，比如切换调试模式、设置密钥、或是别的设定环境的东西。

Flask 被设计为需要配置来启动应用。你可以在代码中硬编码配置，这对于小的应用并不坏，但是有更好的方法。

跟你如何载入配置无关，会有一个可用的配置对象保存着载入的配置值: `Flask` 对象的 `config` 属性。这是 Flask 自己放置特定配置值的地方，也是扩展可以存储配置值的地方。但是，你也可以把自己的配置保存到这个对象里。

8.1 配置基础

`config` 实际上继承于字典，并且可以像修改字典一样修改它:

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

给定的配置值会被推送到 `Flask` 对象中，所以你可以在那里读写它们:

```
app.debug = True
```

你可以使用 `dict.update()` 方法来一次性更新多个键:

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

8.2 内置的配置值

下列配置值是 Flask 内部使用的:

DEBUG	启用/禁用调试模式
TESTING	启用/禁用测试模式
PROPAGATE_EXCEPTIONS	显式地允许或禁用异常的传播。如果没有设置或显式地设置为 *None*，当 TESTING 或 DEBUG 为真时，这个值隐式地为 true.
PRESERVE_CONTEXT_ON_EXCEPTION	默认情况下，如果应用工作在调试模式，请求上下文不会在异常时出栈来允许调试器内省。 这可以通过这个键来禁用。你同样可以用这个设定来强制启用它，即使没有调试执行，这对调试生产应用很有用（但风险也很大）
SECRET_KEY	密钥
SESSION_COOKIE_NAME	会话 cookie 的名称。
SESSION_COOKIE_DOMAIN	会话 cookie 的域。如果不设置这个值，则 cookie 对 SERVER_NAME 的全部子域名有效
SESSION_COOKIE_PATH	会话 cookie 的路径。如果不设置这个值，且没有给 '/' 设置过，则 cookie 对 APPLICATION_ROOT 下的所有路径有效。
SESSION_COOKIE_HTTPONLY	控制 cookie 是否应被设置 httponly 的标志，默认为 *True*
SESSION_COOKIE_SECURE	控制 cookie 是否应被设置安全标志，默认为 *False*
PERMANENT_SESSION_LIFETIME	以 **datetime.timedelta** 对象控制长期会话的生存时间。从 Flask 0.8 开始，也可以用整数来表示秒。
SESSION_REFRESH_EACH_REQUEST	这个标志控制永久会话如何刷新。如果被设置为 *True*（这是默认值），每一个请求 cookie 都会被刷新。如果设置为 *False*，只有当 cookie 被修改后才会发送一个 *set-cookie* 的标头。非永久会话不会受到这个配置项的影响 。
USE_X_SENDFILE	启用/禁用 x-sendfile
LOGGER_NAME	日志记录器的名称
SERVER_NAME	服务器名和端口。需要这个选项来支持子域名（例如：'myapp.dev:5000'）。注意 localhost 不支持子域名，所以把这个选项设置为 “localhost” 没有意义。设置 SERVER_NAME 默认会允许在没有请求上下文而仅有应用上下文时生成 URL

APPLICATION_ROOT	如果应用不占用完整的域名或子域名，这个选项可以被设置为应用所在的路径。这个路径也会用于会话 cookie 的路径值。如果直接使用域名，则留作 None
MAX_CONTENT_LENGTH	如果设置为字节数， Flask 会拒绝内容长度大于此值的请求进入，并返回一个 413 状态码
SEND_FILE_MAX_AGE_DEFAULT:	默认缓存控制的最大期限，以秒计，在 **flask.Flask.send_static_file()**(默认的静态文件处理器)中使用。对于单个文件分别在 **Flask** 或 **Blueprint** 上使用 **get_send_file_max_age()**来覆盖这个值。默认为 43200（12 小时）。
TRAP_HTTP_EXCEPTIONS	如果这个值被设置为 True，Flask 不会执行 HTTP 异常的错误处理，而是像对待其它异常一样， 通过异常栈让它冒泡地抛出。这对于需要找出 HTTP 异常源头的可怕调试情形是有用的。
TRAP_BAD_REQUEST_ERRORS	Werkzeug 处理请求中的特定数据的内部数据结构会抛出同样也是“错误的请求”异常的特殊的 key errors 。同样地，为了保持一致，许多操作可以显式地抛出 BadRequest 异常。因为在调试中，你希望准确地找出异常的原因，这个设置用于在这些情形下调试。如果这个值被设置为 True，你只会得到常规的回溯。
PREFERRED_URL_SCHEME	生成 URL 的时候如果没有可用的 URL 模式话将使用这个值。默认为 http
JSON_AS_ASCII	默认情况下 Flask 使用 ascii 编码来序列化对象。如果这个值被设置为 False， Flask 不会将其编码为 ASCII，并且按原样输出，返回它的 unicode 字符串。比如 jsonfiy 会自动地采用 utf-8 来编码它然后才进行传输。
JSON_SORT_KEYS	默认情况下 Flask 按照 JSON 对象的键的顺序来序来序列化它。这样做是为了确保键的顺序不会受到字典的哈希种子的影响，从而返回的值每次都是一致的，不会造成无用的额外 HTTP 缓存。你可以通过修改这个配置的值来覆盖默认的操作。但这是不被推荐的做法因为这个默认的行为可能会给你在性能的代价上带来改善。
JSONIFY_PRETTYPRINT_REGULAR	如果这个配置项被 True （默认值）， 如果不是 XMLHttpRequest 请求的话（由 X-Requested-With 标头控制） json 字符

	串的返回值会被漂亮地打印出来。

关于 SERVER_NAME 的更多

SERVER_NAME 用于子域名支持。因为 Flask 在得知现有服务器名之前不能猜测出子域名部分，所以如果你想使用子域名，这个选项是必要的，并且也用于会话 cookie 。

请注意，不只是 Flask 有不知道子域名是什么的问题，你的 web 浏览器也会这样。现代 web 浏览器不允许服务器名不含有点的跨子域名 cookie 。所以如果你的服务器名是 'localhost'，你不能在 'localhost' 和它的每个子域名下设置 cookie 。请选择一个合适的服务器名，像 'myapplication.local'， 并添加你想要的服务器名 + 子域名到你的 host 配置或设置一个本地 绑定 。

0.4 新版功能: LOGGER_NAME

0.5 新版功能: SERVER_NAME

0.6 新版功能: MAX_CONTENT_LENGTH

0.7 新版功能: PROPAGATE_EXCEPTIONS, PRESERVE_CONTEXT_ON_EXCEPTION

0.8 新版功能: TRAP_BAD_REQUEST_ERRORS, TRAP_HTTP_EXCEPTIONS, APPLICATION_ROOT, SESSION_COOKIE_DOMAIN, SESSION_COOKIE_PATH, SESSION_COOKIE_HTTPONLY, SESSION_COOKIE_SECURE

0.9 新版功能: PREFERRED_URL_SCHEME

0.10 新版功

能: `JSON_AS_ASCII`, `JSON_SORT_KEYS`, `JSONIFY_PRETTYPRINT_REGULAR`

1.0 新版功能: `SESSION_REFRESH_EACH_REQUEST`

8.3 从文件配置

如果你能在独立的文件里存储配置，理想情况是存储在当前应用包之外，它将变得更有用。这使得通过各式包处理工具（ *部署和分发* ）打包和分发你的应用成为可能，并在之后才修改配置文件。

则一个常见模式为如下:

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

首先从 *yourapplication.default_settings* 模块加载配置，然后用 **`YOURAPPLICATION_SETTINGS`** 环境变量指向的文件的内容覆盖其值。 在 Linux 或 OS X 上，这个环境变量可以在服务器启动之前 ，在 shell 中用 export 命令设置:

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

在 Windows 下则使用其内置的 *set* 命令:

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

配置文件其实是 Python 文件。只有大写名称的值才会被存储到配置对象中。所以请确保你在配置键中使用了大写字母。

这里是一个配置文件的例子:

```
# Example configuration

DEBUG = False

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

确保足够早载入配置，这样扩展才能在启动时访问配置。配置对象上也有其它方法来从多个文件中载入配置。完整的参考请阅读 `Config` 对象的文档。

8.4 配置的最佳实践

之前提到的建议的缺陷是它会使得测试变得有点困难。基本上，这个问题没有单一的 100% 解决方案，但是你可以注意下面的事项来改善体验:

1. 在函数中创建你的应用，并在上面注册蓝图。这样你可以用不同的配置来创建多个应用实例，以此使得单元测试变得很简单。你可以用这样的方法来按需传入配置。
2. 不要写出在导入时需要配置的代码。如果你限制只在请求中访问配置，你可以在之后按需重新配置对象。

8.5 开发 / 生产

大多数应用不止需要一份配置。生产服务器和开发期间使用的服务器应该各有一份单独的配置。处理这个的最简单方法是，使用一份默认的总会被载入的配置，和一部分版本控制，以及独立的配置来像上面提到的例子中必要的那样覆盖值:

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

然后你只需要添加一个独立的 *config.py* 文件然后 export`YOURAPPLICATION_SETTINGS=/path/to/config.py` 。不过，也有其它可选的方式。 例如你可以使用导入或继承。

在 Django 世界中流行的是在文件顶部，显式地使用 `fromyourapplication.default_settings import *` 导入配置文件，并手动覆盖更改。你也可以检查一个类似 `YOURAPPLICATION_MODE` 的环境变量来设置 *production* ， *development* 等等，并导入基于此的不同的硬编码文件。

一个有意思的模式是在配置中使用类和继承:

```
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
```

启用这样的配置你需要调用 `from_object()`

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

管理配置文件有许多方式，这取决于你。这里仍然给出一个好建议的列表:

- 在版本控制中保留一个默认的配置。向配置中迁移这份默认配置，或者在覆盖配置值前，在你自己的配置文件中导入它。
- 使用环境变量来在配置间切换。这样可以在 Python 解释器之外完成，使开发和部署更容易，因为你可以在不触及代码的情况下快速简便地切换配置。如果你经常在不同的项目中作业，你甚至可以创建激活一个 virtualenv 并导出开发配置的脚本。
- 使用 fabric 之类的工具在生产环境中独立地向生产服务器推送代码和配置。 参阅 *使用 Fabric 部署* 模式来获得更详细的信息。

8.6 实例文件夹

0.8 新版功能.

Flask 0.8 引入了示例文件夹。 Flask 在很长时间使得直接引用相对应用文件夹的路径成为可能(通过 `Flask.root_path`)。这也是许多开发者加载存储在载入应用旁边的配置的方法。不幸的是，这只会在应用不是包，即根路径指向包内容的情况下才能工作。

在 Flask 0.8 中，引入了 `Flask.instance_path` 并提出了“实例文件夹”的新概念。实例文件夹被为不使用版本控制和特定的部署而设计。这是放置运行时更改的文件和配置文件的最佳位置。

你可以在创建 Flask 应用时显式地提供实例文件夹的路径，也可以让 Flask 自动找到它。对于显式的配置，使用 *instance_path* 参数:

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

请注意给出的 *一定* 是绝对路径。

如果 *instance_path* 参数没有赋值，会使用下面默认的位置:

- 未安装的模块:

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

- 未安装的包:

```
- /myapp
```

- /__init__.py
- /instance

    - 已安装的包或模块:

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

    `$PREFIX` 是你 Python 安装的前缀。这个前缀可以是 `/usr` 或者你的 virtualenv 的路径。你可以打印 `sys.prefix` 的值来查看前缀被设置成了什么。

既然配置对象提供从相对文件名来载入配置的方式，那么我们也使得它从相对实例路径的文件名加载成为可能，如果你想这样做。配置文件中的相对路径的行为可以在“相对应用的根目录”（默认）和“相对实例文件夹”中切换，后者通过应用构造函数的 *instance_relative_config* 开关实现:

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

这里有一个配置 Flask 来从模块预载入配置并覆盖配置文件夹中配置文件（如果存在）的完整例子:

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

实例文件夹的路径可以在 **`Flask.instance_path`** 找到。 Flask 也提供了一个打开实例文件夹中文件的捷径，就是 **`Flask.open_instance_resource()`** 。

两者的使用示例:

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

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

九、信号

0.6 新版功能.

从 Flask 0.6 开始， Flask 集成了信号支持。这个支持由 blinker 库提供，并且当它不可用时会优雅地退回。

什么是信号？信号通过发送发生在核心框架的其它地方或 Flask 扩展的动作时的通知来帮助你解耦应用。简而言之，信号允许特定的发送端通知订阅者发生了什么。

Flask 提供了几个信号，其它的扩展可能会提供更多。另外，请注意信号倾向于通知订阅者，而不应该鼓励订阅者修改数据。你会注意到，信号似乎和一些内置的装饰器做同样的事情（例如： `request_started` 与 `before_request()` 十分相似）。然而它们工作的方式是有差异的。譬如核心的 `before_request()` 处理程序以特定的顺序执行，并且可以在返回响应之前放弃请求。相比之下，所有的信号处理器执行的顺序没有定义，并且不修改任何数据。

信号之于其它处理器最大的优势是你可以在一秒钟的不同的时段上安全地订阅。譬如这些临时的订阅对单元测试很有用。比如说你想要知道哪个模板被作为请求的一部分渲染：信号允许你完全地了解这些。

9.1 订阅信号

你可以使用信号的 `connect()` 方法来订阅信号。该函数的第一个参数是信号发出时要调用的函数，第二个参数是可选的，用于确定信号的发送端。退订一个信号，可以使用 `disconnect()` 方法。

对于所有的核心 Flask 信号，发送端都是发出信号的应用。当你订阅一个信号，请确保也提供一个发送端，除非你确实想监听全部应用的信号。这在你开发一个扩展的时候尤其正确。

比如这里有一个用于在单元测试中找出哪个模板被渲染和传入模板的变量的助手上下文管理器:

```
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)
```

这可以很容易地与一个测试客户端配对:

```
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
```

确保订阅使用了一个额外的 `**extra` 参数，这样当 Flask 对信号引入新参数时你的调用不会失败。

代码中，从 *with* 块的应用 *app* 中流出的渲染的所有模板现在会被记录到 *templates* 变量。无论何时模板被渲染，模板对象和上下文中都会被添加到它里面。

此外，也有一个方便的助手方法（ **`connected_to()`** ），它允许你临时地把函数订阅到信号并使用信号自己的上下文管理器。因为这个上下文管理器的返回值不能由我们决定，所以必须把列表作为参数传入:

```
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)
```

上面的例子会看起来是这样:

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

Blinker API 变更

`connected_to()` 方法出现于 Blinker 1.1 。

9.2 创建信号

如果你想要在自己的应用中使用信号，你可以直接使用 blinker 库。最常见的用法是在自定义的 `Namespace` 中命名信号。这也是大多数时候推荐的做法:

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

现在你可以这样创建新的信号:

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

这里使用唯一的信号名，简化调试。可以用 `name` 属性来访问信号名。

给扩展开发者

如果你在编写一个 Flask 扩展并且你想优雅地在没有 blinker 安装时退化，你可以用 `flask.signals.Namespace` 这么做。

9.3 发送信号

如果你想要发出信号，调用 `send()` 方法可以做到。 它接受发送端作为第一个参数，和一些推送到信号订阅者的可选关键字参数:

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

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

永远尝试选择一个合适的发送端。如果你有一个发出信号的类，把 *self* 作为发送端。如果你从一个随机的函数发出信号，

把 `current_app._get_current_object()` 作为发送端。

传递代理作为发送端

永远不要向信号传递 `current_app` 作为发送端，使用 `current_app._get_current_object()` 作为替代。这样的原因是， `current_app` 是一个代理，而不是真正的应用对象。

9.4 信号与 Flask 的请求上下文

信号在接收时，完全支持 *请求上下文*。上下文本地的变量在 `request_started` 和 `request_finished` 一贯可用， 所以你可以信任 `flask.g` 和其它需要的东西。注意 *发送信号* 和 `request_tearing_down` 信号中描述的限制。

9.5 基于装饰器的信号订阅

你可以在 Blinker 1.1 中容易地用新的 `connect_via()` 装饰器订阅信号:

```
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)
```

9.6 核心信号

下列是 Flask 中存在的信号:

flask.**template_rendered**

当模板成功渲染的时候，这个信号会发出。这个信号与模板实例 *template* 和上下文的字典（名为 *context* ）一起调用。

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**

这个信号在处建立请求上下文之外的任何请求处理开始前发送。因为请求上下文已经被约束，订阅者可以用 `request` 之类的标准全局代理访问请求。

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**

这个信号恰好在请求发送给客户端之前发送。它传递名为 *response* 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**

这个信号在请求处理中抛出异常时发送。它在标准异常处理生效 *之前*，甚至是在没有异常处理的情况下发送。异常本身会通过 *exception* 传递到订阅函数。

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**

这个信号在请求销毁时发送。它总是被调用，即使发生异常。当前监听这个信号的函数会在常规销毁处理后被调用，但这不是你可以信赖的。

订阅示例:

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

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

从 Flask 0.9 ，如果有异常的话它会被传递一个 *exc* 关键字参数引用导致销毁的异常。

flask.**appcontext_tearing_down**

这个信号在应用上下文销毁时发送。它总是被调用，即使发生异常。当前监听这个信号的函数会在常规销毁处理后被调用，但这不是你可以信赖的。

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)
```

如果有异常它会被传递一个 *exc* 关键字参数引用导致销毁的异常。

flask.**appcontext_pushed**

这个信号在应用上下文压入栈时发送。发送者是应用对象。这通常在单元测试中为了暂时地钩住信息比较有用。例如这可以用来提前在 *g* 对象上设置一些资源。

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
```

测试代码:

```
def test_user_me(self):
```

```
    with user_set(app, 'john'):
        c = app.test_client()
        resp = c.get('/users/me')
        assert resp.data == 'username=john'
```

0.10 新版功能.

flask.**appcontext_popped**

这个信号在应用上下文弹出栈时发送。发送者是应用对象。这通常在 `appcontext_tearing_down` 信号发送后发送。

0.10 新版功能.

flask.**message_flashed**

这个信号在应用对象闪现一个消息时发送。消息作为 *message* 命名参数发送， 分类则是 *category* 参数。

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)
```

0.10 新版功能.

十、即插视图

0.7 新版功能.

Flask 0.7 引入了即插视图，灵感来自 Django 的基于类而不是函数的通用视图。 其主要目的是让你可以对已实现的部分进行替换，并且这个方式可以定制即插视图。

10.1 基本原则

想象你有一个从数据库载入一个对象列表并渲染到视图的函数:

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

这是简单而灵活的，但如果你想要用一种通用的，同样可以适应其它模型和模板的方式来提供这个视图，你会需要更大的灵活性。这就是基于类的即插视图所做的。 第一步，把它转换为基于类的视图，你要这样做:

```
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/', ShowUsers.as_view('show_users'))
```

如你所见，你需要做的是创建一个 `flask.views.View` 的子类， 并且实现 `dispatch_request()` 。然后我们需要用类方法 `as_view()` 把这个类转换到一个实际的视图函数。你传给这个函数的字符串是视图之后的最终名称。但是用它自己实现的方法不够有效，所以我们稍微重构一下代码:

```
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()
```

这当然不是那么有助于一个小例子，但是对于解释基本原则已经很有用了。当你有一个基于类的视图，那么问题来了， *self* 指向什么。它工作的方式是，无论何时请求被调度，会创建这个类的一个新实例，并且 `dispatch_request()` 方法会以 URL 规则为参数调用。 这个类本身会用传递到 `as_view()` 函数的参数来实例化。 比如，你可以像这样写一个类:

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

然后你可以这样注册它:: And then you can register it like this:

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

10.2 方法提示

即插视图可以像常规函数一样用 `route()` 或更好的 `add_url_rule()` 附加到应用中。然而当你附加它时，你必须提供 HTTP 方法的名称。为了将这个信息加入到类中，你可以提供 `methods` 属性来承载它:

```
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'))
```

10.3 基于调度的方法

对每个 HTTP 方法执行不同的函数，对 RESTful API 非常有用。你可以通过 `flask.views.MethodView` 容易地实现。每个 HTTP 方法映射到同名函数 （只有名称为小写的）：

```
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'))
```

如此，你可以不提供 `methods` 属性。它会自动的按照类中定义的方法来设置。

10.4 装饰视图

既然视图类自己不是加入到路由系统的视图函数，那么装饰视图类并没有多大意义。 相反的，你可以手动装饰 `as_view()` 的返回值:

```
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)
```

从 Flask 0.8 开始，你也有一种在类声明中设定一个装饰器列表的方法:

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

因为从调用者的视角来看 self 是不明确的，所以你不能在单独的视图方法上使用常规的视图装饰器，请记住这些。

10.5 用于 API 的方法视图

Web API 的工作通常与 HTTP 动词紧密相关，所以这使得实现这样一个基于 `MethodView` 类的 API 很有意义。也就是说，你会注意到大多数时候，API 需要不同的 URL 规则来访问相同的方法视图。譬如，想象一种情况，你在 web 上暴露一个用户对象:

URL	HTTP 方法	描述
`/users/`	`GET`	获得全部用户的列表
`/users/`	`POST`	创建一个新用户
`/users/<id>`	`GET`	显示某个用户
`/users/<id>`	`PUT`	更新某个用户
`/users/<id>`	`DELETE`	删除某个用户

那么，你会想用 `MethodView` 做什么？诀窍是利用你可以对相同的视图提供多个规则的事实。

让我们假设这时视图看起来是这个样子:

```
class UserAPI(MethodView):


    def get(self, user_id):
        if user_id is None:
            # return a list of users
            pass
        else:
            # expose a single user
            pass
```

```
    def post(self):
        # create a new user
        pass

    def delete(self, user_id):
        # delete a single user
        pass

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

如此，我们怎样把它挂载到路由系统中？添加两条规则，并且为每条规则显式地指出 HTTP 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'])
```

如果你有许多看起来类似的 API ，你可以重构上述的注册代码:

```
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')
```

十一、应用上下文

0.9 新版功能.

Flask 背后的设计理念之一就是，代码在执行时会处于两种不同的“状态”（states）。 当 `Flask` 对象被实例化后在模块层次上应用便开始隐式地处于应用配置状态。一直到第一个请求还是到达这种状态才隐式地结束。当应用处于这个状态的时候 ，你可以认为下面的假设是成立的：

- 程序员可以安全地修改应用对象
- 目前还没有处理任何请求
- 你必须得有一个指向应用对象的引用来修改它。不会有某个神奇的代理变量指向你刚创建的或者正在修改的应用对象的

相反，到了第二个状态，在处理请求时，有一些其它的规则：

- 当一个请求激活时，上下文的本地对象（ `flask.request` 和其它对象等） 指向当前的请求
- 你可以在任何时间里使用任何代码与这些对象通信

这里有一个第三种情况，有一点点差异。有时，你正在用类似请求处理时方式来与应用交互，即使并没有活动的请求。想象一下你用交互式 Python shell 与应用交互的情况，或是一个命令行应用的情况。

`current_app` 上下文本地变量就是应用上下文驱动的。

11.1 应用上下文的作用

应用上下问存在的主要原因是，在过去，请求上下文被附加了一堆函数，但是又没有什么好的解决方案。因为 Flask 设计的支柱之一是你可以在一个 Python 进程中拥有多个应用。

那么代码如何找到“正确的”应用？在过去，我们推荐显式地到处传递应用，但是这会让我们在使用不是以这种理念设计的库时遇到问题。

解决上述问题的常用方法是使用后面将会提到的 `current_app` 代理对象，它被绑定到当前请求的应用的引用。既然无论如何在没有请求时创建一个这样的请求上下文是一个没有必要的昂贵操作，应用上下文就被引入了。

11.2 创建应用上下文

有两种方式来创建应用上下文。第一种是隐式的：无论何时当一个请求上下文被压栈时， 如果有必要的话一个应用上下文会被一起创建。由于这个原因，你可以忽略应用上下文的存在，除非你需要它。

第二种是显式地调用 `app_context()` 方法:

```
from flask import Flask, current_app


app = Flask(__name__)
with app.app_context():
    # within this block, current_app points to app.
    print current_app.name
```

在配置了 `SERVER_NAME` 时，应用上下文也被用于 `url_for()` 函数。这允许你在没有请求时生成 URL 。

11.3 应用上下文局部变量

应用上下文会在必要时被创建和销毁。它不会在线程间移动，并且也不会在不同的请求之间共享。正因为如此，它是一个存储数据库连接信息或是别的东西的最佳位置。内部的栈对象叫做 `flask._app_ctx_stack` 。扩展可以在最顶层自由地存储额外信息，想象一下它们用一个充分独特的名字在那里存储信息，而不是在 `flask.g` 对象里， `flask.g` 是留给用户的代码用的。

更多详情见 *Flask 扩展开发* 。

11.4 上下文用法

上下文的一个典型应用场景就是用来缓存一些我们需要在发生请求之前或者要使用的资源。举个例子，比如数据库连接。当我们在应用上下文中来存储东西的时候你得选择一个唯一的名字，这是因为应用上下文为 Flask 应用和扩展所共享。

最常见的应用就是把资源的管理分成如下两个部分：

1. 一个缓存在上下文中的隐式资源
2. 当上下文被销毁时重新分配基础资源

通常来讲，这将会有一个 `get_X()` 函数来创建资源 `X`，如果它还不存在的话。 存在的话就直接返回它。另外还会有一个 `teardown_X()` 的回调函数用于销毁资源 `X`。

如下是我们刚刚提到的连接数据库的例子:

```
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()
```

当 `get_db()` 这个函数第一次被调用的时候数据库连接已经被建立了。 为了使得看起来更隐式一点我们可以使用 **`LocalProxy`** 这个类:

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

这样的话用户就可以直接通过访问 `db` 来获取数据句柄了， `db` 已经在内部完成了对 `get_db()` 的调用。

十二、请求上下文

这部分文档描述了在 Flask 0.7 中的行为，与旧的行为基本一致，但有细小微妙的差异。

这里推荐先阅读 *应用上下文* 章节。

12.1 深入上下文作用域

比如说你有一个应用函数返回用户应该跳转到的 URL 。想象它总是会跳转到 URL 的 `next` 参数，或 HTTP referrer ，或索引页:

```
from flask import request, url_for


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

如你所见，它访问了请求对象。当你试图在纯 Python shell 中运行这段代码时， 你会看见这样的异常:

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

这有很大意义，因为我们当前并没有可以访问的请求。所以我们需要制造一个请求并且绑定到当前的上下文。`test_request_context` 方法为我们创建一个 `RequestContext`:

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

可以通过两种方式利用这个上下文：使用 *with* 声明或是调用 `push()` 和 `pop()` 方法:

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

从这点开始，你可以使用请求对象:

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

直到你调用 *pop*:

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

因为请求上下文在内部作为一个栈来维护，所以你可以多次压栈出栈。这在实现内部重定向之类的东西时很方便。

更多如何从交互式 Python shell 中利用请求上下文的信息，请见 *与 Shell 共舞* 章节。

12.2 上下文如何工作

如果你研究 Flask WSGI 应用内部如何工作，你会找到和这非常相似的一段代码:

```
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，并结合 *with* 声明来绑定上下文。 从相同线程中被调用的一切，直到 *with* 声明结束前，都可以访问全局的请求变量（ `flask.request` 和其它）。

请求上下文内部工作如同一个栈。栈顶是当前活动的请求。`push()` 把上下文添加到栈顶， `pop()` 把它移出栈。在出栈时，应用的 `teardown_request()` 函数也会被执行。

另一件需要注意的事是，请求上下文被压入栈时，并且没有当前应用的应用上下文， 它会自动创建一个 *应用上下文* 。

12.3 回调和错误

在 Flask 中，请求处理时发生一个错误时会发生什么？这个特殊的行为在 0.7 中变更了，因为我们想要更简单地得知实际发生了什么。新的行为相当简单:

1. 在每个请求之前，执行 `before_request()` 上绑定的函数。 如果这些函数中的某个返回了一个响应，其它的函数将不再被调用。任何情况下，无论如何这个返回值都会替换视图的返回值。
2. 如果 `before_request()` 上绑定的函数没有返回一个响应， 常规的请求处理将会生效，匹配的视图函数有机会返回一个响应。
3. 视图的返回值之后会被转换成一个实际的响应对象，并交给 `after_request()` 上绑定的函数适当地替换或修改它。
4. 在请求的最后，会执行 `teardown_request()` 上绑定的函数。这总会发生，即使在一个未处理的异常抛出后或是没有请求前处理器执行过（例如在测试环境中你有时会想不执行请求前回调）。

现在错误时会发生什么？在生产模式中，如果一个异常没有被捕获，将调用 500 internal server 的处理。在生产模式中，即便异常没有被处理过，也会往上冒泡抛给给 WSGI 服务器。如此，像交互式调试器这样的东西可以提供有用的调试信息。

在 0.7 中做出的一个重大变更是内部服务器错误不再被请求后回调传递处理，而且请求后回调也不再保证会执行。这使得内部的调度代码更简洁，易于定制和理解。

新的绑定于销毁请求的函数被认为是用于代替那些请求的最后绝对需要发生的事。

12.4 销毁回调

销毁回调是是特殊的回调，因为它们在不同的点上执行。严格地说，它们不依赖实际的请求处理，因为它们限定在 `RequestContext` 对象的生命周期。当请求上下文出栈时， `teardown_request()` 上绑定的函数会被调用。

这对于了解请求上下文的寿命是否因为在 with 声明中使用测试客户端或在命令行中使用请求上下文时被延长很重要:

```
with app.test_client() as client:
    resp = client.get('/foo')
    # the teardown functions are still not called at that point
    # even though the response ended and you have the response
    # object in your hand

# only when the code reaches this point the teardown functions
# are called.  Alternatively the same thing happens if another
# request was triggered from the test client
```

从这些命令行操作中，很容易看出它的行为:

```
>>> 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
>>>
```

注意销毁回调总是会被执行，即使没有请求前回调执行过，或是异常发生。测试系统的特定部分也会临时地在不调用请求前处理器的情况下创建请求上下文。确保你写的请求销毁处理器不会报错。

12.5 留意代理

Flask 中提供的一些对象是其它对象的代理。背后的原因是，这些代理在线程间共享， 并且它们在必要的情景中被调度到限定在一个线程中的实际的对象。

大多数时间你不需要关心它，但是在一些例外情况中，知道一个对象实际上是代理是有益的:

- 代理对象不会伪造它们继承的类型，所以如果你想运行真正的实例检查，你需要在被代理的实例上这么做（见下面的 *_get_current_object* ）。
- 如果对象引用是重要的（例如发送 *信号* ）

如果你需要访问潜在的被代理的对象，你可以使用 `_get_current_object()` 方法:

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

12.6 错误时的上下文保护

无论错误出现与否，在请求的最后，请求上下文会出栈，并且相关的所有数据会被销毁。在开发中，当你想在异常发生时，长期地获取周围的信息，这会成为麻烦。 在 Flask 0.6 和更早版本中的调试模式，如果发生异常，请求上下文不会被弹出栈， 这样交互式调试器才能提供给你重要信息。

从 Flask 0.7 开始，我们设定 `PRESERVE_CONTEXT_ON_EXCEPTION` 配置变量来更好地控制该行为。这个值默认与 `DEBUG` 的设置相关。当应用工作在调试模式下时，上下文会被保护，而生产模式下相反。

不要在生产模式强制激活 `PRESERVE_CONTEXT_ON_EXCEPTION`，因为它会导致在异常时应用的内存泄露。不过，它在开发时获取开发模式下相同的错误行为来试图调试一个只有生产设置下才发生的错误时很有用。

十三、用蓝图实现模块化的应用

0.7 新版功能.

Flask 用 *蓝图（blueprints）* 的概念来在一个应用中或跨应用制作应用组件和支持通用的模式。蓝图很好地简化了大型应用工作的方式，并提供给 Flask 扩展在应用上注册操作的核心方法。一个 `Blueprint` 对象与 `Flask` 应用对象的工作方式很像，但它确实不是一个应用，而是一个描述如何构建或扩展应用的 *蓝图*。

13.1 为什么使用蓝图？

Flask 中的蓝图为这些情况设计:

- 把一个应用分解为一个蓝图的集合。这对大型应用是理想的。一个项目可以实例化一个应用对象，初始化几个扩展，并注册一集合的蓝图。
- 以 URL 前缀和/或子域名，在应用上注册一个蓝图。 URL 前缀/子域名中的参数即成为这个蓝图下的所有视图函数的共同的视图参数（默认情况下）。
- 在一个应用中用不同的 URL 规则多次注册一个蓝图。
- 通过蓝图提供模板过滤器、静态文件、模板和其它功能。一个蓝图不一定要实现应用或者视图函数。
- 初始化一个 Flask 扩展时，在这些情况中注册一个蓝图。

Flask 中的蓝图不是即插应用，因为它实际上并不是一个应用——它是可以注册，甚至可以多次注册到应用上的操作集合。为什么不使用多个应用对象？你可以做到那样（见 *应用调度*），但是你的应用的配置是分开的，并在 WSGI 层管理。

蓝图作为 Flask 层提供分割的替代，共享应用配置，并且在必要情况下可以更改所注册的应用对象。它的缺点是你不能在应用创建后撤销注册一个蓝图而不销毁整个应用对象。

13.2 蓝图的设想

蓝图的基本设想是当它们注册到应用上时，它们记录将会被执行的操作。当分派请求和生成从一个端点到另一个的 URL 时，Flask 会关联蓝图中的视图函数。

13.3 我的第一个蓝图

这看起来像是一个非常基本的蓝图。在这个案例中，我们想要实现一个简单渲染静态模板的蓝图:

```
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)
```

当我们使用 `@simple_page.route` 装饰器绑定函数时，在蓝图之后被注册时它会记录把 *show* 函数注册到应用上的意图。此外，它会给函数的端点加上由 **`Blueprint`** 的构造函数中给出的蓝图的名称作为前缀（在此例中是 `simple_page` ）。

13.4 注册蓝图

那么你如何注册蓝图？像这样:

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

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

如果你检查已经注册到应用的规则，你会发现这些:

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

第一个显然是来自应用自身，用于静态文件。其它的两个用于 `simple_page` 蓝图中的 *show* 函数。如你所见，它们的前缀是蓝图的名称，并且用一个点 （`.`）来分割。

不过，蓝图也可以在不同的位置挂载:

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

那么，这些果然是生成出的规则:

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

在此之上，你可以多次注册蓝图，虽然不是每个蓝图都会正确地响应这些。实际上， 蓝图能否被多次挂载，取决于蓝图是怎样实现的。

13.5 蓝图资源

蓝图也可以提供资源。有时候你会只为它提供的资源而引入一个蓝图。

13.5.1 蓝图资源文件夹

像常规的应用一样，蓝图被设想为包含在一个文件夹中。当多个蓝图源于同一个文件夹时，可以不必考虑上述情况，但也这通常不是推荐的做法。

这个文件夹会从 **`Blueprint`** 的第二个参数中推断出来，通常是 *__name__* 。这个参数决定对应蓝图的是哪个逻辑的 Python 模块或包。如果它指向一

个存在的 Python 包，这个包（通常是文件系统中的文件夹）就是资源文件夹。如果是一个模块， 模块所在的包就是资源文件夹。你可以访问 `Blueprint.root_path` 属性来查看资源文件夹是什么:

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

可以使用 `open_resource()` 函数来快速从这个文件夹打开源文件:

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

13.5.2 静态文件

一个蓝图可以通过 *static_folder* 关键字参数提供一个指向文件系统上文件夹的路径，来暴露一个带有静态文件的文件夹。这可以是一个绝对路径，也可以是相对于蓝图文件夹的路径:

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

默认情况下，路径最右边的部分就是它在 web 上所暴露的地址。因为这里这个文件夹叫做 `static` ，它会在蓝图 + `/static` 的位置上可用。也就是说，蓝图为 `/admin` 把静态文件夹注册到 `/admin/static` 。

最后是命名的 *blueprint_name.static* ， 这样你可以生成它的 URL ， 就像你对应用的静态文件夹所做的那样:

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

13.5.3 模板

如果你想要蓝图暴露模板，你可以提供 `Blueprint` 构造函数中的 *template_folder* 参数来实现:

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

像对待静态文件一样，路径可以是绝对的或是相对蓝图资源文件夹的。模板文件夹会被加入到模板的搜索路径中，但是比实际的应用模板文件夹优先级低。这样，你可以容易地在实际的应用中覆盖蓝图提供的模板。

那么当你有一个 `yourapplication/admin` 文件夹中的蓝图并且你想要渲染 `'admin/index.html'` 模板，且你已经提供了 `templates` 作为 *template_folder* ，你需要这样创建文件: `yourapplication/admin/templates/admin/index.html`

13.6 构造 URL

当你想要从一个页面链接到另一个页面，你可以像通常一个样使用 `url_for()` 函数，只是你要在 URL 的末端加上蓝图的名称和一个点（`.`）作为前缀:

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

此外，如果你在一个蓝图的视图函数或是模板中想要从链接到同一蓝图下另一个端点， 你可以通过对端点只加上一个点作为前缀来使用相对的重定向:

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

这个案例中，它实际上链接到 `admin.index`，假如请求被分派到任何其它的 admin 蓝图端点。

十四、Flask 扩展

Flask 扩展用多种不同的方式扩充 Flask 的功能。比如加入数据库支持和其它的常见任务。

14.1 寻找扩展

Flask Extension Registry 中列出了 Flask 扩展，并且可以通过 `easy_install` 或 `pip` 下载。如果你把一个 Flask 扩展添加到 `requirements.rst` 或 `setup.py` 文件的依赖关系中，它们通常可以用一个简单的命令或是在你应用安装时被安装。

14.2 使用扩展

扩展通常附带有文档，来展示如何使用它。扩展的行为没有一个可以预测的一般性规则，除了它们是从同一个位置导入的。如果你有一个名为 `Flask-Foo` 或是 `Foo-Flask` 的扩展，你可以从 `flask.ext.foo` 导入它:

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

14.3Flask 0.8 以前

如果你在使用 Flask 0.7 或更早的版本，包 `flask.ext` 并不存在，你不得不从 `flaskext.foo` 或 `flask_foo` 中导入，这取决与应用是如何分发的。如果你想要开发支持 Flask 0.7 或更早版本的应用，你仍然应该

从 `flask.ext` 中导入。我们提供了一个兼容性模块来在 Flask 的老版本中提供这个包。你可以从 github 上下载它：flaskext_compat.py

这里是使用它的方法:

```
import flaskext_compat
flaskext_compat.activate()

from flask.ext import foo
```

一旦激活了 `flaskext_compat` 模块，就会存在 `flask.ext`，并且你可以从那里开始导入。

十五、与 Shell 共舞

0.3 新版功能.

Python 拥有的交互式 Shell 是人人都喜欢它的一个重要原因。交互式 Shell 允许你实时的运行 Python 命令并且立即得到返回结果。Flask 本身并未内置一个交互式 Shell ，因为它并不需要任何前台的特殊设置，仅仅导入您的应用然后开始探索和使用即可。

然而这里有一些易于获得的助手，可以帮助您在 Shell 遨游时获得更为愉悦的体验。交互式控制台回话的一个重要问题是，您并不是像在浏览器当中那样激发一个请求，因此 `g` 和 `request` 以及其他的一些函数不能使用。然而您想要测试的代码也许依赖他们， 那么让我们瞧瞧该如何解决这个问题。

这就是该那些辅助函数登场的时候了。然而应当说明的是， 这些函数并非仅仅为在交互式 Shell 里使用而编写的，也可以用于单元测试或者其他需要一个虚假的请求上下文的情景。

一般来说，在阅读本章节之前还是建议大家先阅读 *请求上下文* 相关章节。

15.1 创建一个请求上下文

从 Shell 创建一个合适的上下文，最简单的方法是使用 `test_request_context` 方法，此方法会创建一个 `RequestContext` 对象:

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

一般来说，您可以使用 *with* 声明来激活这个请求对象， 但是在终端中，调用 `push()` 方法和 `pop()` 方法会更简单:

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

从这里往后，您就可以使用这个请求对象直到您调用 *pop* 方法为止:

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

15.2 激发请求发送前后的调用

仅仅创建一个请求上下文，您仍然不能运行请求发送前通常会运行的代码。如果您在将连接数据库的任务分配给发送请求前的函数调用，或者在当前用户并没有被储存在 `g` 对象里等等情况下，您可能无法访问到数据库。

您可以很容易的自己完成这件事，仅仅手动调用 `preprocess_request()` 函数即可:

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

请注意， `preprocess_request()` 函数可能会返回一个响应对象。这时，忽略它就好了。

要关闭一个请求，您需要在请求后的调用函数(由 `process_response()` 函数激发)运行之前要一些小小的把戏:

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

被注册为 `teardown_request()` 的函数将会在上下文环境出栈之后自动执行。所以这是用来销毁请求上下文(如数据库连接等)资源的最佳地点。

15.3 进一步提升 Shell 使用体验

如果您喜欢在 Shell 里实验您的新点子，您可以创建一个包含你想要导入交互式回话中的东西的的模块。在这里，您也可以定义更多的辅助方法用来完成一些常用的操作，例如初始化数据库、删除一个数据表等。

把他们放到一个模块里（比如 *shelltools* 然后在 Shell 中导入它）：

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

十六、Flask 代码模式

某些东西非常通用，以至于你有很大的机会在绝大部分 Web 应用中，都能找到他们的身影。例如相当多的应用在使用关系数据库而且包含用户注册和认证模块。 在这种情况下，请求开始之前，他们会打开数据库连接、获得当前已经登陆的用户信息。在请求结束的时候，数据库连接又会被关闭。

这章提供了一些由用户贡献的代码片段和模板来加速开发 Flask Snippet Archives.

16.1 大型应用

对于比较大型的应用，更好的做法是使用包管理代码，而不是模块来管理代码。 这非常简单，设想一个如下结构的应用:

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

16.1.1 简单的包

将一个项目改为一个更大的包，仅仅创建一个新的 *yourapplication* 文件夹在已存的文件夹下面，然后将所有的的文件都移动到它下面。之后将 *yourapplication.py* 重命名为 *__init__.py*（确保先删除了其中所有的 *.pyc* 文件，否则可能导致错误的结果）

您最后得到的东西应该像下面这样:

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

如何在此种方式下运行您的应用？原来的 `python yourapplication/__init__.py` 不能再工作了。这是由于 Python 不希望在包中的模块成为初始运行的文件。但这不是一个大问题，仅仅添加一个名叫 *runserver.py* 的新文件，把这个文件放在 *yourapplication* 文件夹里，并添加如下功能:

```
from yourapplication import app
```

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

然后，我们又能对应用做什么呢？现在我们可以重新构造我们的应用，将其改造为多个模块。你唯一需要记住的就是下面的速记备忘表:

1. *Flask* 程序对象的创建必须在 *__init__.py* 文件里完成， 这样我们就可以安全的导入每个模块，而 *__name__* 变量将会被分配给正确的包。
2. 所有（上面有 `route()` 装饰器的那些）视图函数必须导入到 *__init__.py* 文件。此时，请通过模块而不是对象本身作为路径导入这些视图函数。**必须在应用对象创建之后** 导入视图模块。

这里是 *__init__.py* 的一个例子:

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


import yourapplication.views
```

而 *views.py* 应该看起来像这样:

```
from yourapplication import app


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

您最终应该得到的程序结构应该是这样:

```
/yourapplication
```

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

循环导入

每个 Python 程序员都会讨厌他们，而我们反而还添加了几个进去：循环导入(在两个模块相互依赖对方的时候，就会发生循环导入)。在这里 *views.py* 依赖于 *__init__.py*。通常这被认为是个不好的主意，但是在这里实际上不会造成问题。之所以如此，是因为我们实际上没有在 *__init__.py* 里使用这些视图，而仅仅是保证模块被导入了。并且，我们是在文件的结尾这么做的。

这种做法仍然有些问题，但是如果您想要使用修饰器，那么没有其他更好的方法了。检查 *聚沙成塔* 这一章来寻找解决问题的些许灵感吧。

16.1.2 与蓝图一起工作

如果您有规模较大的应用，建议您将他们分拆成小的组，让每个组接口于蓝图提供的辅助功能。关于这一主题进一步的介绍请参考 *用蓝图实现模块化的应用* 这一章节的文档

16.2 应用程序的工厂函数

如果您已经开始使用包和蓝图(*用蓝图实现模块化的应用*)辅助您的应用开发了，那么这里还有一些非常好的办法可以进一步的提升开发体验。当蓝图被导入的时候， 一个通用的模板将会负责创建应用程序对象。但是如果你将这个对象的创建工作移交给一个函数来完成，那么你就可以在此后创建它的多个实例。

这么做的目的在于:

1. 测试。你可以使用多个应用程序的实例，为每个实例分配分配不同的配置， 从而测试每一种不同的情况。
2. 多个实例。想象以下情景：您需要同时运行同一个应用的不同版本，您当然可以在你的 Web 服务器中配置多个实例并分配不同的配置，但是如果你使用工厂函数， 你就可以在一个随手即得的进程中运行这一个应用的不同实例了！

那么该如何使用他们呢？

16.2.1 基础的工厂函数

您可以像下面展示的这样，从一个函数里启动这个应用:

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

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

    return app
```

有得必有失，在导入时，您无法在蓝图中使用这个应用程序对象。然而您可以在一个请求中使用他。如果获取当前配置下的对应的应用程序对象呢？请使用: `current_app` 函数:

```
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'])
```

在这里我们从配置中查找一个网页模板文件的名字。

16.2.2 使用应用程序

所以，要使用这样的一个应用，你必须先创建这个应用对象，这里是一个运行此类程序的 *run.py* 文件的例子:

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

16.2.3 工厂函数的改进

前文所提供的工厂函数并不是特别聪明好用，您可以改进它，如下的改变可以是直接且可行的:

1. 使得在单元测试中传入配置值成为可行，以使您不必在文件系统中创建多个配置文件。
2. 在程序初始时从蓝图中调用一个函数，这样您就有机会修改应用的参数属性了 (就像在在请求处理器前后的调用钩子等)
3. 如果必要的话，在应用正在被创建时添加 WSGI 中间件。

16.3 应用调度

应用调度指的是在 WSGI 层次合并运行多个 Flask 的应用的进程。您不能将 Flask 与更大的东西合并，但是可以和 WSGI 应用交叉。这甚至允许您将 Django 和 Flask 的应用运行在同一个解释器下。这么做的用处依赖于这个应用内部是如何运行的。

与 *模块方式* 的区别在于，此时您运行的不同 Flask 应用是相互之间完全独立的，他们运行在不同的配置，而且在 WSGI 层调度。

16.3.1 如何使用此文档

下面的所有技巧和例子都将最终得到一个 `application` 对象，这个对象可以在任何 WSGI 服务器上运行。在生产环境下，请参看 *部署选择* 相关章节。在开发时，Werkzeug 提供了一个提供了一个内置的开发服务器， 可以通过 `werkzeug.serving.run_simple()` 函数使用:

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

注意，`run_simple` 函数不是为生产用途设计的，发布应用时可以使用 *成熟的 WSGI 服务器* 。

为了能使用交互式调试器，调试必须在应用和简易开发服务器两边都被激活。 下面是一个带有调试功能的 “Hello World” 的例子:

```
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)
```

16.3.2 合并应用

如果您有一些完全独立的应用程序，而您希望他们使用同一个 Python 解释器， 背靠背地运行，您可以利用 `werkzeug.wsgi.DispatcherMiddleware` 这个类。 这里，每个 Flask 应用对象都是一个有效的 WSGI 应用对象，而且他们在调度中间层当中被合并进入一个规模更大的应用，并通过前缀来实现调度。

例如，您可以使您的主应用运行在 / 路径，而您的后台接口运行在 */backend* 路径:

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


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

16.3.3 通过子域名调度

有时，您希望使用对一个应用使用不同的配置，对每个配置运行一个实例，从而有多个实例存在。假设应用对象是在函数中生成的，您就可以调用这个函数并实例化一个实例，这相当容易实现。为了使您的应用支持在函数中创建新的对象，请先参考 *应用程序的工厂函数* 模式。

一个相当通用的例子，那就是为不同的子域名创建不同的应用对象。比如您将您的 Web 服务器设置为将所有的子域名都分发给您的引用，而您接下来使用这些子域名信息创建一个针对特定用户的实例。一旦您使得您的服务器侦听所有的子域名请求，那么您就可以使用一个非常简单的 WSGI 对象来进行动态的应用程序构造。

实现此功能最佳的抽象层就是 WSGI 层。您可以编写您自己的 WSGI 程序来检查访问请求，然后分发给您的 Flask 应用。如果您的应用尚未存在，那么就创建一个并且保存下来:

```
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)
```

调度器可以这样使用:

```
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:
        # if there is no user for that subdomain we still have
        # to return a WSGI application that handles that request.
        # We can then just return the NotFound() exception as
```

```
        # application which will render a default 404 page.
        # You might also redirect the user to the main page then
        return NotFound()

    # otherwise create the application for the specific user
    return create_app(user)

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

16.3.4 使用路径来调度

通过 URL 路径分发请求跟前面的方法很相似。只需要简单检查请求路径当中到第一个斜杠之前的部分，而不是检查用来确定子域名的 *HOST* 头信息就可以了:

```
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)
```

这种例子与之前子域名调度那里的区别是，这里如果创建应用对象的函数返回了 *None*，那么请求就被降级回推到另一个应用当中：

```
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)
```

16.4 使用 URL 处理器

0.7 新版功能.

Flask 0.7 版引入了 URL 处理器的概念。此概念的意义在于，对于一部分资源， 您并不是很清楚该如何设定其 URL 相同的部分。例如可能有一些 URL 包含了几个字母来指定的多国语言语种，但是你不想在每个函数里都手动识别到底是哪个语言。

搭配 Blueprint 使用时，URL 处理器尤其有用。这里我们将会就具体的应用例子介绍如何使用 URL 处理器和 Blueprint

16.4.1 国际化的应用程序 URL

试想如下一个网页应用:

```
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
    ...
```

这可能会产生一大片重复的代码，因为你必须在每个函数当中手动处理 `g` 对象。 当然，你可以使用装饰器来简化它，但想要从一个函数动态生成 URL 到另一个函数， 仍需详细地提供这段多国语言代号码，这将非常地恼人。

对于后者，这就是 `url_defaults()` 函数大展神威的地方了！ 这些函数可以自动地将值注入到 `url_for()` 的调用中去。下面的代码检查多语言代号码是否在包含各个 URL 值的字典里，以及末端调用的函数是否接受 `'lang_code'`

```
@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 映射的函数 `is_endpoint_expecting()` 可以被用来识别是否可以给末端的函数提供一个多国语言代号码。

相反的函数是 `url_value_preprocessor()` 。他们在请求成功匹配并且能够执行针对 URL 值的代码时立即执行。实际上，他们将信息从包含这些值的字典当中取出，然后将其放在某个其他的地方:

```
@app.url_value_preprocessor
```

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

这样，您再也不必在每个函数中都要将 *lang_code* 分配给 `g` 了。 您可以进一步的改进它，通过编写您自己的装饰器，并使用这些装饰器为包含多国语言代号码的 URL 添加前缀。但是使用蓝图相比起来会更优雅一些。一旦 `'lang_code'` 被从字典里弹出，他就不会在被传递到视图函数当中。这样，代码就可简化为如下形式:

```
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():
    ...
```

16.4.2 多国语言化的 Blueprint URL

因为 Blueprint 能够自动地为所有 URL 添加一个相同的字符串作为前缀，所以自动处理这些函数变得非常简单。 每个蓝图都可以有一个 URL 处理器，即从 `url_defaults()` 函数中移除一整套业务逻辑，因为它不再检查 URL 是否真正与 `'lang_code'` 相关:

```
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():
    ...
```

16.5 部署和分发

distribute 的前身是 `setuptools`，是一个通常用于分发 Python 库和扩展程序的外部库。它依赖于随 Python 预装的 `distutils` 库， 而后者则是一个基础的模块安装系统，这一安装系统也支持很多复杂的构造，使得大型应用更易于分发。

- **支持依赖关系管理**: 一个库可以声明自己依赖哪些软件包，从而在安装这个模块的时候，自动将依赖的软件包也安装到您的计算机。
- **注册软件包**: setuptools 将您的包注册到您的安装的 Python 环境中。这使得您可以使一个包中的代码查询另一个包所提供的信息。这一系统最知名的特性就是对接口机制的支持，也就是说一个包可以声明自己的一个接口，从而允许其他的包通过这个接口对自己进行扩展。
- **安装包管理器**: *easy_install* 默认随 Python 安装，它可以用于为您安装其他的库。您也可以使用 *pip* 这个可能早晚会代替 *easy_install* 的包管理器，它能够完成安装软件包之外更多的任务。

而对于 Flask 自己，则所有您可以在 cheessshop 上找到的软件包，都随着 distribute 分发管理器，或者更古老的 setuptools 和 distutils 分发。

在这里，我们假定您的应用名为 *yourapplication.py* ，而您没使用模块而是使用 *package* 的结构来组织代码。分发带有标准模块的代码不被 distribute 支持，所以我们不去管它。如果您还没有将您的应用转化为包的形式， 请参考前文 *大型应用* 的内容查找如何做到这件事。

利用 distribute 完成一个有效的部署进行更复杂和更自动化的部署方案的第一步， 如果您使程序完全自动化，可以阅读 *使用 Fabric 部署* 这一章。

16.5.1 基础的安装脚本

因为你已经让 Flask 运行起来了，所以不管怎么说您的系统上应该会有 setuptools 或者 d istribute，如果你没有这两样，不要害怕。这里帮你准备了一个脚本：`distribute_setup.py` 你只需要下载并用 Python 解释器运行它。

考虑这些操作可能会有风险，因此建议您参考 *你最好使用 virtualenv* 一文。

您的安装代码将总是保存在与您应用同目录下的 *setup.py* 文件中。为文件指定这一名称只是为了方便，不过一般来说每一个人自然而然的在程序目录下寻找这个文件，所以您最好别改变它。

同时，即使您在使用 *distribute* ，您也会导入一个名为 *setuptools* 的包。 *distribute* 完全向下兼容 *setuptools* ，所以我们也使用这个名字来导入它。

一个基本的 Flask 应用的 *setup.py* 文件看起来像如下这样:

```
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']
)
```

切记，您必须详细地列出子代码包，如果您想要 distribute 自动为您寻找这些包， 您可以使用 *find_packages* 函数:

```
from setuptools import setup, find_packages


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

大多数 *setup* 函数当中的参数的意义从字面意思就能看出来，然而 *include_package_data* 和 *zip_safe* 可能不在此列。 *include_package_data* 告诉 distribute 自动查找一个 *MANIFEST.in* 文件。 解析此文件获得有效的包类型的数据，并安装所有这些包。我们使用这个特性来分发 Python 模块自带的静态文件和模板(参考 *分发代码*)。而 *zip_safe* 标志可以被用来强制阻止 ZIP 安装包的建立。通常情况下，您不希望您的包以 ZIP 压缩包的形式被安装，因为一些工具不支持这种方式，而且这样也会让调试代码异常麻烦。

16.5.2 分发代码

如果您视图安装您刚刚创建的包，您会发现诸如 *static* 和 *templates* 这样的文件夹没有安装进去。这是因为 distribute 不知道该把哪些文件添加进去。您只要在 *setup.py* 相同的文件夹下创建一个 *MANIFEST.in* 文件，并在此文件中列出所有应该被添加进去的文件:

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

不要忘记，即使您已经将他们列在 *MANIFEST.in* 文件当中，也需要您将 *setup* 函数的 *include_package_data* 参数设置为 *True* ，否则他们仍然不会被安装。

16.5.3 声明依赖关系

您需要使用一个链表在 *install_requires* 参数中声明依赖关系。链表的每个元素是需要从 PyPI 下载并安装的包的名字，默认将总会下载安装最新的的版本。但是您也可以指定需要的最大和最小的版本区间。以下是一个例子:

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

前文曾经指出，这些依赖都从 PyPI 当中下载，如果您需要依赖一个不能在 PyPI 当中被下载的包，比如这个包是个内部的，您不想与别人分享。这时，您可以依然照原来那样将包列在列表里，但是同时提供一个包括所有可选下载地址的列表，以便于安装时从这些地点寻找分发的软件包:

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

请确认那个页面包含一个文件夹列表，且页面上的连接被指向实际需要下载的软件包。 distribute 通过扫描这个页面来寻找需要安装的文件，因此文件的名字必须是正确无误的。 如您有一个内部服务器包含有这些包，将 URL 指向这个服务器。

16.5.4 安装 / 开发

安装您的应用(到一个 virtualenv)，只需使用 *install* 指令运行 *setup.py* 即可。 这会将您的应用安装到一个 virtualenv 的 site-packages 文件夹下面，并且同时下载和安装所有的依赖包:

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

如果您在进行基于这个包的开发，并且希望安装开发所依赖的工具或软件包， 您可以使用 *develop* 命令代替 *install*

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

此时将不会把您的文件拷贝到 site-packages 文件夹，而仅仅是在那里创建指向这些文件的文件链接。您可以继续编辑和修改这些代码，而无需在每次修改之后运行 *install* 命令。

16.6 使用 Fabric 部署

Fabric 是一个 Python 下类似于 Makefiles 的工具，但是能够在远程服务器上执行命令。如果您有一个良好配置过的 Python 软件包 (*大型应用*) 且对“配置”概念的理解良好，那么在外部服务器上部署 Flask 应用将会非常容易。

开始之前，请先检查如下列表中的事项是否都已经满足了:

- 在本地已经安装了 Fabric 1.0 。即这个教程完成时， Fabric 的最新版本。

- 应用程序已经被封装为包的形式，而且有一个有效的 *setup.py* 文件(参考 *部署和分发*)。
- 在下文中的例子里，我们使用 *mod_wsgi* 作为远程服务器使用的服务端程序。 您当然也可以使用您喜欢的服务端程序，但是考虑到 Apache 和 *mod_wsgi* 的组合非常简单易用且容易安装配置，并且在无 root 权限的情况下，存在一个比较简单的方法来重启服务器。

16.6.1 创建第一个 Fabfile

Fabfile 用于指定 Fabric 执行的命令，它通常被命名为 *fabfile.py* 并使用 *fab* 命令运行。文件中所有的函数将被当做 *fab* 的子命令显示出来，他们可以在一个或多个主机上运行。这些主机要么在 fabfile 当中定义，要么在命令输入时指定。在本文中我们将他们定义在 fabfile 里。

这是第一个基础的例子，能够将现有源代码上传到指定服务器并将它们安装进如一个已经存在的虚拟环境中:

```
from fabric.api import *


# 远程服务器登陆使用的用户名
env.user = 'appuser'
# 需要进行操作的服务器地址
env.hosts = ['server1.example.com', 'server2.example.com']


def pack():
    # 以 tar 归档的方式创建一个新的代码分发
```

```
    local('python setup.py sdist --formats=gztar', capture=False)


def deploy():
    # 之处发布产品的名称和版本
    dist = local('python setup.py --fullname', capture=True).strip()
    # 将代码归档上传到服务器当中的临时文件夹内
    put('dist/%s.tar.gz' % dist, '/tmp/yourapplication.tar.gz')
    # 创建一个文件夹，进入这个文件夹，然后将我们的归档解压到那里
    run('mkdir /tmp/yourapplication')
    with cd('/tmp/yourapplication'):
        run('tar xzf /tmp/yourapplication.tar.gz')
        # 使用我们虚拟环境下的 Python 解释器安装我们的包
        run('/var/www/yourapplication/env/bin/python setup.py install')
    # 现在我们的代码已经部署成功了，可以删除这个文件夹了
    run('rm -rf /tmp/yourapplication /tmp/yourapplication.tar.gz')
    # 最终生成 .wsgi 文件，以便于 mod_wsgi 重新加载应用程序
    run('touch /var/www/yourapplication.wsgi')
```

上面的代码例子注释很清晰，应该很容易明白，下面是 fabric 常用命令的一个归纳:

- *run* - 在远程服务器上执行所有命令
- *local* - 在本地执行所有命令
- *put* - 将指定文件上传到指定服务器
- *cd* - 改变远程服务器当上的当前操作目录，此命令必须与 *with* 声明一起使用

16.6.2 运行 Fabfile

如何执行 fabfile 呢？您应该使用 *fab* 命令。若要发布当前版本的代码到远程服务器上，您只需执行如下命令:

```
$ fab pack deploy
```

然而这需要您的服务器已经创建过 `/var/www/yourapplication` 文件夹而且 `/var/www/yourapplication/env` 是一个可用的虚拟环境。而且， 我们还没有在服务器上创建配置文件或者 *.wsgi* 文件。因此，我们怎么样把一个新的服务器转换为可以使用基础设备呢。

这视我们想要配置的服务器数量的不同，实现起来有所差别。如果我们只有一个远程应用服务器(大部分应用都是都属于此类)，那么 fabfile 里添加一个专门负责此类的命令有些小题大做。但是显然我们可以这么做。在这里，您可以会运行命令 *setup* 或者 *bootstrap* 。然后将服务器的地址详细地在命令行当中指定:

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

初始化一个新的服务器，您大概需要执行如下几个步骤:

1. 在 `/var/www` 目录下创建目录结构:

```
2. $ mkdir /var/www/yourapplication
3. $ cd /var/www/yourapplication
4. $ virtualenv --distribute env
```

5. 上传一个新的 *application.wsgi* 文件以及为应用程序准备的配置文件(例如: *application.cfg*)等到服务器上
6. 为 *yourapplication* 创建一个新的 Apache 配置，并激活它。请确保激活了对 *.wsgi* 改变的监视功能，这样在我们创建或改变这个文件时 Apache 可以自动重新加载应用 (详细内容请参考 *mod_wsgi (Apache)*)

现在的问题是， *application.wsgi* 和 *application.cfg* 文件从何而来。

16.6.3WSGI 文件

WSGI 文件应导入这个应用并且设定一个环境变量，这个环境变量指定了应用程序应到哪里寻找配置文件。下面是一个完全完成上述功能的短例:

```
import os

os.environ['YOURAPPLICATION_CONFIG'] =
'/var/www/yourapplication/application.cfg'

from yourapplication import app
```

应用程序本身则应该向下面这样，通过查询环境变量来查找配置，以此初始化自己:

```
app = Flask(__name__)

app.config.from_object('yourapplication.default_config')

app.config.from_envvar('YOURAPPLICATION_CONFIG')
```

这种方法在本文档的 *配置处理* 这节中进行了详细介绍。

16.6.4 配置文件

正如上文所属，应用程序将会通过查

找 *YOURAPPLICATION_CONFIG* 环境变量以找到正确的配置文件。因此我们必须将配置文件放在应用程序可以找到的地方。 配置文件有在不同电脑上表现出不同效果的特质，所以您不应该以普通的方式对它进行版本控制。

一个流行的做法是将不同服务器的配置文件保存在不同的版本控制仓库里，然后在不同的服务器中分别抽取出来。然后建立到从配置应该在的地点（如: `/var/www/yourapplication`)到这个文件实际位置的符号链接。

我们预计只有一个或两个服务器需要部署，因此我们采用另一种方法，也就是提前手动将配置文件上传到需要的未知。

16.6.5 第一次部署

现在我们可以开始进行第一次部署了。我们已经初始化了服务器以使它拥有正确的虚拟环境和已经激活的 Apache 配置文件。现在我们可以把应用打包然后部署了:

```
$ fab pack deploy
```

Fabric 现在就会连接到所有服务器，然后运行在 fabfile 文件中所指定的命令。 最初他会执行打包工作，为我们创建代码归档，然后他部署和上

传代码到所有的服务器，并在那里安装他们。归功于 *setup.py* ，所有引用依赖的包和库都将自动被下载和安装到我们的虚拟环境中。

16.6.6 下一步操作

从现在开始，我们可以做的事情变得如此之多，以至于部署代码实际上可以看做一种乐趣:

- 创建一个 *bootstrap* 命令用于初始化新的服务器，它将初始化一个新的虚拟环境安装以及适当配置 Apache 等。
- 将配置文件放置到一个独立的版本控制仓库里，然后将活动的配置符号连接到它应该在的地方。
- 您应该将您的应用程序也放置到一个版本控制仓库中，然后在服务器中提取最新的版本并安装，您也可以很容易的回溯到以前的版本。
- 为测试提供函数接口，这样您就可以将测试代码部署到服务器上并在服务器端执行测试套件。

使用 Fabric 是相当有趣，键入 `fab deploy` 并看到您的应用自动部署到一个或多个服务器上，您会有“简直像是魔术”这样的感觉。

16.7 在 Flask 中使用 SQLite 3

在 Flask 中，在请求开始的时候用 `before_request()` 装饰器实现打开数据库连接的代码，然后在请求结束的时候用 `before_request()` 装饰器关闭数据库连接。在这个过程中需要配合 `g` 对象。

于是，在 Flask 里一个使用 SQLite 3 的简单例子就是下面这样:

```
import sqlite3
from flask import g

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

def connect_db():
    return sqlite3.connect(DATABASE)

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

@app.teardown_request
def teardown_request(exception):
    if hasattr(g, 'db'):
        g.db.close()
```

注解

请记住，teardown request 在请求结束时总会运行，即使 before-request 处理器运行失败或者从未运行过。我们需要确保数据库连接在关闭的时候在那里。

16.7.1 按需连接

上述方法的缺陷在于，它只能用于 Flask 会执行 before-request 处理器的场合下有效，如果您想要在一个脚本或者 Python 的交互式终端中访问数据库。那么您必须做一些类似下面的代码的事情:

```
with app.test_request_context():
    app.preprocess_request()
    # now you can use the g.db object
```

为了激发连接代码的执行，使用这种方式的话，您将不能离开对请求上下文的依赖。 但是您使用以下方法可以使应用程序在必要时才连接:

```
def get_connection():
    db = getattr(g, '_db', None)
    if db is None:
        db = g._db = connect_db()
    return db
```

缺点就是，您必须使用 `db = get_connection()` 而不是仅仅直接使用 `g.db` 来访问数据库连接。

16.7.2 简化查询

现在在每个请求处理函数里，您都可以访问 *g.db* 来获得当前打开的数据库连接。 此时，用一个辅助函数简化 SQLite 的使用是相当有用的:

```
def query_db(query, args=(), one=False):
    cur = g.db.execute(query, args)
```

```
    rv = [dict((cur.description[idx][0], value)
               for idx, value in enumerate(row)) for row in cur.fetchall()]
    return (rv[0] if rv else None) if one else rv
```

相比起直接使用原始的数据指针和连接对象。这个随手即得的小函数让操作数据库的操作更为轻松。 像下面这样使用它:

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

如果您只希望得到一个单独的结果:

```
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']
```

将变量传入 SQL 语句时，使用在语句之前使用一个问号，然后将参数以链表的形式穿进去。永远不要直接将他们添加到 SQL 语句中以字符串形式传入，这样做将会允许恶意用户以 SQL 注入 的方式攻击您的应用。

16.7.3 初始化数据库模型

关系数据库需要一个模型来定义储存数据的模式，所以应用程序通常携带一个 *schema.sql* 文件用于创建数据库。提供一个特定的函数来创建数据库是个不错的主意，以下的函数就能为您做到这件事:

```
from contextlib import closing


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

然后您就可以在 Python 的交互式终端中创建一个这样的数据库:

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

16.8 在 Flask 中使用 SQLAlchemy

很多人更倾向于使用 SQLAlchemy 进行数据库操作。在这种情况下，建议您使用包的而不是模块的方式组织您的应用代码，并将所有的模型放置到一个单独的模块中 (*大型应用*)。尽管这并非必要，但是这么做将会让程序的结构更加明晰。

使用 SQLAlchemy 有四种常用的方法，我们在下面列出了这几种方法的基本使用框架:

16.8.1Flask-SQLAlchemy 扩展

因为 SQLAlchemy 是一个常用的数据库抽象层和数据库关系映射包 (ORM)，并且需要一点点设置才可以使用，因此存在一个 Flask 扩展帮助您操作它。如果您想要快速开始使用，那么我们建议您使用这种方法。

您可以从 PyPI 下载到 Flask-SQLAlchemy

16.8.2 显式调用

SQLAlchemy 中的 declarative 扩展是最新的使用 SQLAlchemy 的方法。它允许您同时定义表和模型，就像 Django 一样工作。除了下文所介绍的内容外，我们建议您参考 declarative 扩展的官方文档。

这是一个 *database.py* 模块的例子:

```
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():
    # 在这里导入所有的可能与定义模型有关的模块，这样他们才会合适地
    # 在 metadata 中注册。否则，您将不得不在第一次执行 init_db() 时
    # 先导入他们。
    import yourapplication.models
    Base.metadata.create_all(bind=engine)
```

为了定义您的模型，仅仅构造一个上面代码编写的 *Base* 类的子类。如果您好奇为何我们在这里不用担心多线程的问题(就像我们在先前使用 `g` 对象操作 SQLite3 的例子一样):那是因为 SQLAlchemy 已经在 `scoped_session` 类当中为我们完成了这些任务。

在您的应用当中以一个显式调用 SQLAlchemy，您只需要将如下代码放置在您应用的模块中。Flask 将会在请求结束时自动移除数据库会话:

```
from yourapplication.database import db_session


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

这是一个模型的例子(将代码放入 *models.py* 或类似文件中):

```
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)
```

您可以使用 *init_db* 函数创建一个数据库:

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

按照如下方式将数据实体插入数据库:

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

查询代码也很简单:

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

16.8.3 手动实现 ORM

手动实现 ORM (对象关系映射) 相比前面的显式调用方法，既有一些优点，也有一些缺点。 主要差别在于这里的数据表和模型是分开定义的，然后再将其映射起来。这提供了更大的灵活性， 但是会增加了代码量。通常

来说它和上面显式调用的工作的方式很相似，所以请确保您的应用已经被合理分割到了包中的不同模块中。

这是一个 *database.py* 模块的例子:

```
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)
```

与显式调用相同，您需要在请求结束后关闭数据库会话。将下面的代码放到您的应用程序模块中:

```
from yourapplication.database import db_session

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

下面是一个数据表和模型的例子(将他们放到 *models.py* 当中):

```
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)
```

查询和插入操作和上面所给出的例子是一样的。

16.8.4SQL 抽象层

如果您仅用到数据库系统和 SQL 抽象层，那么您只需要引擎部分:

```
from sqlalchemy import create_engine, MetaData


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

```
metadata = MetaData(bind=engine)
```

然后您就可以像上文的例子一样声明数据表，或者像下面这样自动加载他们:

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

您可以使用 *insert* 方法插入数据，我们需要先获取一个数据库连接，这样我们就可以使用“事务”了:

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

SQLAlchemy 将会为我们自动提交对数据库的修改。

查询数据可以直接通过数据库引擎，也可以使用一个数据库连接:

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

返回的结果也是字典样式的元组:

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

您也可以将 SQL 语句的字符串传入到 `execute()` 函数中:

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

更多 SQLAlchemy 相关信息，请参考 其网站.

16.9 上传文件

哦，上传文件可是个经典的好问题了。文件上传的基本概念实际上非常简单， 他基本是这样工作的:

1. 一个 `<form>` 标签被标记有 `enctype=multipart/form-data` ，并且在里面包含一个 `<input type=file>` 标签。
2. 服务端应用通过请求对象上的 `files` 字典访问文件。
3. 使用文件的 `save()` 方法将文件永久地保存在文件系统上的某处。

16.9.1 一点点介绍

让我们建立一个非常基础的小应用，这个小应用可以上传文件到一个指定的文件夹里， 然后将这个文件显示给用户。让我们看看这个应用的基础代码:

```
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
```

首先我们导入一些东西，大多数内容都是直接而容易的。

`werkzeug.secure_filename()`将会在稍后进行解释。 *UPLOAD_FOLDER* 是我们储存上传的文件的地方，而 *ALLOWED_EXTENSIONS* 则是允许的文件类型的集合。然后我们手动为应用添加一个的 URL 规则。我们通常很少这样做，但是为什么这里要如此呢？原因是我们希望实际部署的服务器（或者我们的开发服务器）来为我们提供这些文件的访问服务，所以我们只需要一个规则用来生成指向这些文件的 URL 。

为什么我们限制上传文件的后缀呢？您可能不希望您的用户能够上传任何文件到服务器上，如果服务器直接将数据发送给客户端。以这种方式，您可以确保您的用户不能上传可能导致 XSS 问题(参考 *跨站脚本攻击（XSS）*)的 HTML 文件。也确保会阻止 *.php* 文件以防其会被运行。当然，谁还会在服务器上安装 PHP 啊，是不是？ :)

下一步，就是检查文件类型是否有效、上传通过检查的文件、以及将用户重定向到已经上传好的文件 URL 处的函数了:

```
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()** 函数具体做了那些事呢？现在的问题是，有一个信条叫做“永远别相信你用户的输入”，这句话对于上传文件的文件名也是同样有效的。所有提交的表单数据都可以伪造，而文件名本身也可能是危险的。在摄氏只需记住：在将文件保存在文件系统之前，要坚持使用这个函数来确保文件名是安全的。

关于文件名安全的更多信息

您对 **secure_filename()** 的具体工作和您没使用它会造成的后果感兴趣？试想一个人可以发送下列信息作为 *filename* 给您的应用：

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

假定 `../` 的数量是正确的，而您会将这串字符与 *UPLOAD_FOLDER* 所指定的路径相连接，那么这个用户就可能有能力修改服务器文件系统上的一个文件，而他不应该拥有这种权限。这么做需要一些关于此应用情况的技术知识，但是相信我， 骇客们都有足够的耐心 :)

现在我们来研究一下这个函数的功能:

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

现在还有最后一件事没有完成: 提供对已上传文件的访问服务。 在 Flask 0.5 以上的版本我们可以使用一个函数来实现此功能:

```
from flask import send_from_directory


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

或者，您也可以选择为 *uploaded_file* 注册 *build_only* 规则，然后使用 `SharedDataMiddleware` 类来实现下载服务。这种方法同时支持更老版本的 Flask:

```
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']
})
```

运行应用，不出意外的话，一切都应该像预期那样工作了。

16.9.2 改进上传功能

0.6 新版功能.

Flask 到底是如何处理上传的呢？如果服务器相对较小，那么他会先将文件储存在网页服务器的内存当中。否则就将其写入一个临时未知(如函数 `tempfile.gettempdir()` 返回的路径)。但是怎么指定一个文件大小的上限，当文件大于此限制，就放弃上传呢? 默认 Flask 会很欢乐地使用无限制的空间，但是您可以通过在配置中设定 `MAX_CONTENT_LENGTH` 键的值来限制它:

```
from flask import Flask, Request


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

上面的代码将会把上传文件限制为最大 16 MB 。如果请求传输一个更大的文件， Flask 会抛出一个 `RequestEntityTooLarge` 异常。

这个特性是在 Flask 0.6 中被加入的，但是更老的版本也可以通过构建请求对象的子类来实现。更多信息请查询 Werkzeug 文档中文件处理部分的内容。

16.9.3 上传进度条

以前，很多开发者实现进度条的方法是这样的：一边小块小块地读取传输来的文件， 一边将上传进度储存在数据库中，然后在通过客户端的 JavaScript 代码读取进度。 简单来说，客户端会每 5 秒钟询问服务器传输的进度。您感觉到这种讽刺了么？客户端询问一些他本应该已经知道的事情。

现在有了一些性能更好、运行更可靠的解决方案。WEB 已经有了不少变化，现在您可以使用 HTML5、Java、Silverlight 或者 Flash 来实现客户端更好的上传体验。看一看下面列出的库的连接，可以找到一些很好的样例。

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

16.9.4 更简单解决方案

因为存在一个处理上传文件的范式，这个范式在大多数应用中机会不会有太大改变， 所以 Flask 存在一个扩展名为 Flask-Uploads ，这个扩展实现了一整套成熟的文件上传架构。它提供了包括文件类型白名单、黑名单等多种功能。

16.10 缓存

如果您的应用运行很慢，那就尝试引入一些缓存吧。好吧，至少这是提高表现最简单的方法。缓存的工作是什么呢？比如说您有一个需要一段时间才能完成的函数，但是这个函数的返回结果可能在 5 分钟之内都是足够有效的，因此您可以将这个结果放到缓存中一段时间，而不用反复计算。

Flask 本身并不提供缓存功能，但是作为 Flask 基础的 Werkzeug 库，则提供了一些基础的缓存支持。Werkzeug 支持多种缓存后端，通常的选择是 Memcached 服务器。

16.10.1 配置缓存

类似于建立 `Flask` 的对象一样，您创建一个缓存对象，然后让他保持存在。如果您使用的是开发服务器，您可以创建一个 `SimpleCache` 对象，这个对象将元素缓存在 Python 解释器的控制的内存中:

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

如果您希望使用 Memcached 进行缓存，请确保您已经安装了 Memcache 模块支持（您可以通过 *PyPi<http://pypi.python.org/* 获取），并且有一个可用的 Memcached 服务器正在运行。然后您可以像下面这样连接到缓存服务器:

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

如果您在使用 App Engine ，您可以轻易地通过下面的代码连接到 App Engine 的缓存服务器:

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

16.10.2 使用缓存

有两个非常重要的函数可以用来使用缓存。那就是 `get()` 函数和 `set()` 函数。他们的使用方法如下:

从缓存中读取项目，请使用 `get()` 函数， 如果现在缓存中存在对应项目，它将会返回。否则函数将会返回 *None*

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

在缓存中添加项目，使用 `set()` 函数。 第一个参数是想要设定的键，第二个参数是想要缓存的值。您可以设定一个超时时间， 当时间超过时，缓存系统将会自动清除这个项目。

以下是一个通常情况下实现功能完整例子:

```
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
```

16.11 视图装饰器

Python 拥有一件非常有趣的特性，那就是函数装饰器。这个特性允许您使用一些非常简介的语法编辑 Web 应用。因为 Flask 中的每个视图都是一个函数装饰器， 这些装饰器被用来将附加的功能注入到一个或者多个函数中。 `route()` 装饰器您可能已经使用过了。但是在一些情况下您需要实现自己的装饰器。例如， 您有一个仅供登陆后的用户访问的视图，如果未登录的用户试图访问，则把用户转接到登陆界面。这个例子很好地说明了装饰器的用武之地。

16.11.1 过滤未登录用户的装饰器

现在让我们实现一个这样的装饰器。装饰器是指返回函数的函数，它其实非常简单。 您仅需要记住，当实现一个类似的东西，其实是更新 *__name__* 、 *__module__* 以及函数的其他一些属性，这件事情经常被遗忘。但是您不必亲自动手，这里有一个专门用于处理这些的以装饰器形式调用的函数(`functools.wraps()`)。

这个例子家丁登陆页面的名字是 `'login'` 并且当前用户被保存在 *g.user* 当中， 如果么有用户登陆， *g.user* 会是 *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
```

所以您怎么使用这些装饰器呢？将它加为视图函数外最里层的装饰器。当添加更多装饰器的话，一定要记住 `route()` 考试最外面的:

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

16.11.2 缓存装饰器

试想你有一个运算量很大的函数，而且您希望能够将生成的结果在一段时间内缓存起来，一个装饰器将会非常适合用于干这种事。我们假定您已经参考 *缓存* 中提到的内容配置好了缓存功能。

这里有一个用作例子的缓存函数，它从一个指定的前缀(通常是一个格式化字符串) 和当前请求的路径生成一个缓存键。请注意我们创建了一个这样的函数：它先创建一个装饰器，然后用这个装饰器包装目标函数。听起来很复杂？不幸的是，这的确有些难，但是代码看起来会非常直接明了。

被装饰器包装的函数将能做到如下几点:

1. 以当前请求和路径为基础生成缓存时使用的键。

2. 从缓存中取出对应键的值，如果缓存返回的不是空，我们就将它返回回去。
3. 如果缓存中没有这个键，那么最初的函数将会被执行，并且返回的值在指定时间 (默认 5 分钟内)被缓存起来。

代码如下:

```
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，这段代码假定一个示例用的 *cache* 对象时可用的。请参考 *缓存* 以获取更多信息。

16.11.3 模板装饰器

TurboGears 的家伙们前一段时间发明了一种新的常用范式，那就是模板装饰器。 这个装饰器的关键在于，您将想要传递给模板的值组织成字典的形式，然后从视图函数中返回，这个模板将会被自动渲染。这样，下面的三个例子就是等价的了:

```
@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)
```

正如您所看到的，如果没有模板名被指定，那么他会使用 URL 映射的最后一部分，然后将点转换为反斜杠，最后添加上 `'.html'` 作为模板的名字。当装饰器包装的函数返回，返回的字典就会被传递给模板渲染函数。如果 *None* 被返回了，那么相当于一个空的字典。如果非字典类型的对象被

返回，函数将照原样将那个对象再次返回。这样您就可以继续使用重定向函数或者返回简单的字符串了。

这是那个装饰器的源代码:

```
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
```

16.11.4 终端装饰器

如果您希望使用 werkzeug 路由系统来获得更多的灵活性。您需要将终点 (Endpoint) 像 `Rule` 中定义的那样映射起来。通过一个装饰器是可以做到的，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"
```

16.12 使用 WTForms 进行表单验证

如果您不得不跟浏览器提交的表单数据打交道，视图函数里的代码将会很快变得难以阅读。有不少的代码库被开发用来简化这个过程的操作。其中一个就是 WTForms， 这也是我们今天主要讨论的。如果您发现您自己陷入处理很多表单的境地，那您也许应该尝试一下他。

要使用 WTForms ，您需要先将您的表单定义为类。我建议您将应用分割为多个模块 (*大型应用*) ，这样的话您仅需为表单添加一个独立的模块。

挖掘 WTForms 的最大潜力

Flask-WTF 扩展在这个模式的基础上扩展并添加了一些随手即得的精巧的帮助函数，这些函数将会使在 Flask 里使用表单更加有趣，您可以通过 PyPI 获取它。

16.12.1 表单

以下是一个典型的注册页面的例子:

```
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()])
```

16.12.2 在视图里

在视图函数中，表单的使用是像下面这个样子的:

```
@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)
```

注意到我们视图中使用了 SQLAlchemy (参考 *在 Flask 中使用 SQLAlchemy*)。但是这并非必要的，请按照您的需要修正代码。

备忘表:

1. 如果数据是以 *POST* 方式提交的，那么基于请求的 `form` 属性的值创建表单。反过来，如果是使用 *GET* 提交的，就从 `args` 属性创建。
2. 验证表单数据，调用 `validate()` 方法。如果数据验证通过，此方法将会返回 *True* ，否则返回 *False* 。
3. 访问表单的单个值，使用 *form.<NAME>.data* 。

16.12.3 在模板中使用表单

在模板这边，如果您将表单传递给模板，您可以很容易地渲染他们。参看如下代码， 您就会发现这有多么简单了。WTForms 已经为我们完成了一半的表单生成工作。更棒的是，我们可以编写一个宏来渲染表单的字段，让这个字段包含一个标签，如果存在验证错误，则列出列表来。

以下是一个使用这种宏的 *_formhelpers.html* 模板的例子:

```
{% 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 %}
```

这些宏接受一对键值对，WTForms 的字段函数接收这个宏然后为我们渲染他们。 键值对参数将会被转化为 HTML 属性，所以在这个例子里，您可以调用 `render_field(form.username,class="username")` 来将一个类添加到这个输入框元素中。请注意 WTForms 返回标准 Python unicode 字符串，所以我们使用 *|safe* 告诉 Jinjan2 这些数据已经是经过 HTML 过滤处理的了。

以下是 *register.html* 模板，它对应于上面我们使用过的函数，同时也利用了 *_formhelpers.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>
```

关于 WTForms 的更多信息，请访问 WTForms 网站 。

16.13 模板继承

Jinja 最为强大的地方在于他的模板继承功能，模板继承允许你创建一个基础的骨架模板， 这个模板包含您网站的通用元素，并且定义子模板可以重载的 **blocks** 。

听起来虽然复杂，但是其实非常初级。理解概念的最好方法就是通过例子。

16.13.1 基础模板

在这个叫做 `layout.html` 的模板中定义了一个简单的 HTML 文档骨架，你可以将这个骨架用作一个简单的双栏页面。而子模板负责填充空白的 block:

```
<!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>
```

在这个例子中，使用 `{% block %}` 标签定义了四个子模板可以重载的块。*block* 标签所做的的所有事情就是告诉模板引擎：一个子模板可能会重写父模板的这个部分。

16.13.2 子模板

子模板看起来像这个样子：

```
{% 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 %}
```

`{% extends %}` 是这个例子的关键，它会告诉模板引擎这个模板继承自另一个模板的， 模板引擎分析这个模板时首先会定位其父父模板。extends 标签必须是模板的首个标签。 想要渲染父模板中的模板需要使用 `{{ super() }}`。

16.14 消息闪现

好的应用和用户界面的重点是回馈。如果用户没有得到足够的反馈，他们可能最终会对您的应用产生不好的评价。Flask 提供了一个非常简单的方法来使用闪现系统向用户反馈信息。闪现系统使得在一个请求结束的时候记录一个信息，然后在且仅仅在下一个请求中访问这个数据。这通常配合一个布局模板实现。

16.14.1 简单的闪现

这里是一个完成的例子:

```
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()
```

这里的 `layout.html` 模板完成了所有的魔术:

```
<!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 %}
```

这里是 *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 %}
```

这里是登陆模板:

```
{% 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 %}
```

16.14.2 分类闪现

0.3 新版功能.

当闪现一个消息时，是可以提供一个分类的。未指定分类时默认的分类为 `'message'` 。 可以使用分类来提供给用户更好的反馈，例如，错误信息应该被显示为红色北京。

要使用一个自定义的分类，只要使用 `flash()` 函数的第二个参数:

```
flash(u'Invalid password provided', 'error')
```

在模板中，您接下来可以调用 `get_flashed_messages()` 函数来返回这个分类，在下面的情景中，循环看起来将会有一点点不一样:

```
{% 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 %}
```

这仅仅是一个渲染闪现信息的例子，您可也可以使用分类来加入一个诸如 `<strong>Error:</strong>` 的前缀给信息。

16.14.3 过滤闪现消息

0.9 新版功能.

可选地，您可以将一个分类的列表传入到 `get_flashed_messages()` 中， 以过滤函数返回的结果。如果您希望将每个分类渲染到独立的块中，这会非常有用。

```
{% 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 %}
```

16.15 用 jQuery 实现 Ajax

jQuery 是一个小型的 JavaScript 库，它通常被用来简化 DOM 和 JavaScript 操作。通过在服务器和客户端之间交换 JSON 数据是使得 Web 应用动态化的完美方式。

JSON 本身是一个很清量级的数据传输格式，非常近似于 Python 的原始数据类型 (数字、字符串、字典和链表等)，这一数据格式被广泛支持，而且非常容易解析。它几年前开始流行，然后迅速取代了 XML 在 Web 应用常用数据传输格式中的地位。

如果您使用 Python 2.6 以上版本，JSON 的解析库是开箱即用的。在 Python 2.5 中您则必须从 PyPI 安装 simplejson 库。

16.15.1 加载 jQuery

为了使用 jQuery 您需要先下载它，然后将其放置在您应用的静态文件夹中，并确认他被加载了。理想的情况下是，您有一个用于所有页面的布局模板。要加载 jQuery 您只需要在这个布局模板中 <*body*> 标签的最下方添加一个 script 标签。

```
<script type=text/javascript src="{{
  url_for('static', filename='jquery.js') }}"></script>
```

另一个加载 jQuery 的技巧是使用 Google 的 AJAX Libraries API :

```
<script
src="//ajax.googleapis.com/ajax/libs/jquery/1.6.1/jquery.js"></script>
<script>window.jQuery || document.write('<script src="{{
  url_for('static', filename='jquery.js') }}">\x3C/script>')</script>
```

在以上配置的情况下，您需要将 jQuery 放置到静态文件夹当中作为一个备份。浏览器将会首先尝试直接从 Google 加载 jQuery。如果您的用户至少一次访问过使用 Google 提供的的 jQuery 版本的话，浏览器就会缓存这个代码，这样您的网站就可以从中获得加载更快的好处了。

16.15.2 我的站点在哪?

您知道您的应用在哪里运行么？如果您在开发过程当中，那么答案非常简单: 它运行在本地端口，而且就在这个 URL 的根路径位置。但是如果您后来决定将您的应哟 ing 移动到一个不同的未知怎么办？比如 `http://example.com/myapp`？ 在服务器这边，这从来不是一个问题，原因是我们使用的 `url_for()` 函数可以帮我们回答这个问题。但是如果我们在使用 jQuery 我们不应该将指向应用的路径硬编码到程序中，而是将它动态化。该如何做到这点呢？

一个简单的技巧可能是为我们的页面添加一个 script 标签，然后设定一个全局变量作为一个应用根路径的前缀。如下所示:

```
<script type=text/javascript>
  $SCRIPT_ROOT = {{ request.script_root|tojson|safe }};
</script>
```

这里的 `|safe` 是必要的。这样 Jinja 才不会将 JSON 编码的字符串以 HTML 的规则过滤处理掉。通常这种过滤是必要的，但是在 *script* 标签块当中有着不同于原先的过滤规则。

可能有用的信息

在 HTML 中， *script* 标签被声明为 *CDATA* 。这意味着 HTML 转义实体将不会被解析。在 `</script>` 出现之前的所有内容都被当做脚本处理。这也意味着在 script 标签的内容之中不应该出现 `</` 字样。`|tojson` 足以在这里完成正确的事情，他将会为您过滤掉斜杠 (`{{ "</script>"|tojson|safe }}` 将会被渲染成 `"<\/script>"`)。

16.15.3JSON 视图函数

现在让我们创建一个服务端函数，这个服务端函数接收两个数字形式的 URL 参数， 然后将这两个数字相加并以 JSON 对象的形式返回给应用。这是一个相当可笑的例子， 您通常会在服务端直接实现这个功能。但是这是一个方便展示如何配合使用 jQuery 和 Flask 最简单的例子了:

```
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')
```

正如您所见，我们在这里添加了一个 *index* 函数，这个函数用于渲染一个模板。 这个模板将会按照上面的提供的方法加载 jQuery ，并且包含一个小表单用于提供加法运算的两个数，同时表单还提供了用于激发服务器端函数的一个链接。

注意，这里我们使用不会抛出错误的 `get()` 方法。 如果对应的键不存在，一个默认值(这里是 `0`)将 hi 被返回。更进一步，我们还可以将值转换为一个特定类型(就像我们这里的 *int* 类型)。这对于由脚本(APIs,JavaScript 等)激发的代码来说是个非常顺手的工具，因为在这种情况下您不需要特别的错误报告。

16.15.4HTML 部分

您的 index.html 要么继承一个已经加载了 jQuery 且设定了 *$SCRIPT_ROOT* 环境变量的 *layout.html* 模板，要么自己在上方完成了这些事。以下是我们的小应用 (*index.html*) 所需的 HTML 代码。请注意这里我们也将脚本直接写入了 HTML。通常来讲，将脚本代码放置到一个独立的脚本文件里是一个更好的点子。

```
<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>
```

我们不会过多介绍 jQuery 使用的细节，仅仅对以上代买做一个快速的解释:

1. `$(function() { ... })` 将会在浏览器加载完页面的基础内容之后立即执行。
2. `$('selector')` 选择一个用于操作的元素。

3. `element.bind('event', func)` 指定元素被单击时运行的函数，如果这个函数返回 *false* ，那么单击操作的默认行为将被取消。在本例中，点击操作的默认行为是导航到 # 链接标签。
4. `$.getJSON(url, data, func)` 发送一个 *GET* 请求给 *url* ，其中 *data* 对象的内容将以查询参数的形式发送。一旦数据抵达，它将以返回值作为参数执行给定的函数。请注意，我们在这里可以使用我们先前设定的 *$SCRIPT_ROOT* 变量。

如果您还没有完全了解这个例子，可以从 github 上下载 本例源码 。

16.16 自定义错误页面

Flask 自带了很顺手的 `abort()` 函数用于以一个 HTTP 失败代码中断一个请求，他也会提供一个非常简单的错误页面，用于提供一些基础的描述。这个页面太朴素了以至于缺乏一点灵气。

依赖于错误代码的不同，用户看到某个错误的可能性大小也不同。

16.16.1 通常的错误代码

下面列出了一些用户经常遇到的错误代码，即使在这个应用准确无误的情况下也可能发生:

404 Not Found

经典的“哎呦，您输入的 URL 当中有错误”消息。这个消息太常见了，即使是 互联网的新手也知道 404 代号的意义：该死，我寻找

的东西不在那儿。确保 404 页面上有一些有用的信息是一个好主意，至少应该提供一个返回主页的链接。

403 Forbidden

如果您的网站包含一些类型的访问控制，您必须向非法的请求返回 403 错误代号。 所以请确保用户不会在试图访问了一个禁止访问的资源后不知所措。

410 Gone

您知道 404 Not Found 代号还有一个兄弟名为 410 Gone 么？很少有人真正实现 它，您可以考虑将其返回给对以前曾经存在、但是现在已经删除的资源的请求，而 不是直接返回 404 。 如果您还没有从数据库里永久删除这个文档，仅仅是将他们 标记为删除。那么可以为用户展示一个消息，说明他们寻找的东西已经永远删除了。

500 Internal Server Error

通常在出现编程错误或者服务器过载的时候会返回这个错误代号。在这里放一个 漂亮的页面是一个非常好的主意。因为您的应用 *总有一天* 会出现错误(请参考 *记录应用错误*)

16.16.2 错误处理器

一个错误处理器是一个类似于视图函数的函数，但是它在错误发生时被执行，并且错误被当成一个参数传递进来。一般来说错误可能是 `HTTPException` ， 但是在有些情况下会是其他错误：内部服务器的错误的处理器在被执行时，将会同时得到被捕捉到的实际代码错误作为参数。

错误处理器和要捕捉的错误代码使用 `errorhandler()` 装饰器注册。 请记住 Flask *不会* 替您设置错误代码，所以请确保在返回 response 对象时，提供了对应的 HTTP 状态代码。

如下实现了一个“404 Page Not Found”错误处理的例子:

```
from flask import render_template


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

一个示例模板可能会如下所示:

```
{% 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 %}
```

16.17 延迟加载视图

Flask 通常配合装饰器使用，装饰器使用非常简单，而且使您可以将 URL 和处理它的函数放在一起。然而这种方法也有一种不足: 这就意味着您使用装饰器的代码必须在前面导入， 否则 Flask 将无法找到您的函数。

这对于需要很快导入的应用程序来说是一个问题，这种情况可能出现在类似谷歌的 App Engine 这样的系统上。所以如果您突然发现您的引用超出了这种方法可以处理的能力，您可以降级到中央 URL 映射的方法。

用于激活中央 URL 映射的函数是 `add_url_rule()` 方法。 您需要提供一个设置应用程序所有 URL 的文件，而不是使用装饰器。

16.17.1 转换到中央 URL 映射

假象现在的应用的样子如下所示:

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


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


@app.route('/user/<username>')
def user(username):
    pass
```

而中央 URL 映射的方法下，您需要一个不包含任何装饰器的文件 (*views.py*)， 如下所示:

```
def index():
    pass
```

```
def user(username):
    pass
```

然后使用一个文件初始化应用并将函数映射到 URLs:

```
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)
```

16.17.2 延迟加载

目前我们仅仅将视图和路径配置分开了，但是模块仍然是在前面导入的。下面的技巧使得视图函数可以按需加载。可以使用一个辅助类来实现，这个辅助类以函数的方式作用，但是当第一次使用某个函数时，它才在内部导入这个函数:

```
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)
```

在使用这种方法时，将 *__module__* 和 *__name__* 变量设定为合适的值是很重要的。 在你没有手动指定一个 URL 规则时，这两个变量被 Flask 用于在内部确定如何命名 URL 规则。

现在您就可以定义您将视图整合到的位置，如下所示:

```
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'))
```

您可以进一步改进它，以便于节省键盘敲击次数。通过编写一个在内部调用 `add_url_rule()` 方法的函数，自动将一个包含项目名称以及点符号的字符串添加为前缀，并按需将 *view_func* 封装进 *LazyView*

```
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')
```

需要记住的是，请求前后激发的回调处理器必须在一个文件里，并在前面导入， 使之在第一个请求到来之间能够合适地工作。对于其他所有的装饰器来说也是一样的。

16.18 在 Flask 中使用 MongoKit

近些日子，使用基于文档的数据库而不是基于表的关系数据库变得越来越流行。 这一方案展示了如何使用文档映射库 MongoKit ，来与 MongoDB 交互。

这一方案的使用需要一个可用的 MongoDB 服务器，并且安装有 MongoKit 库。

使用 MongoKit 有两种常用的方法，我们将会逐一介绍:

16.18.1 显式调用

MongoKit 的默认行为是这种显式调用的方法。这种方法跟 Django 或者 SQLAlchemy 扩展显示调用扩展大体精神是相同的。

下面是一个 *app.py* 模块的例子:

```
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'])
```

要定义您的模型，只需编写一个从 MongoKit 导入的 *Document* 类的子类。如果您已经看过了 SQLAlchemy 的方案，您可能会奇怪为什么这里没有一个会话，甚至没有定义 *init_db* 函数。一方面， MongoKit 并没有类似会话这种东西。这有时会增加代码量，但是同时也使得数据库操作非常高效。另一方面， MongoDB 是没有模式的。 这意味着您在相同的插入查询，可以使用不同的数据结构。 MongoKit 本身也是没有模式的。但是实现了一些用来确保数据完整的验证。

以下是一个文档的例子 (您可以将这个也放进 *app.py* 文件里):

```
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)


# register the User document with our current connection
connection.register([User])
```

这个例子向您展示了怎么定义您自己的结构(名为 structure)、一个最大字符长度的验证器以及使用 Monkit 的一项名为 *use_dot_notation* 的特性。某人情况下 MongoKit 按照字典的方式行为，但是将 *use_dot_notation* 为真之后，您可以像您在几乎所有的 ORM 当中那样，使用点运算符来分割属性的方式访问您的文档。

向数据库里添加数据的方法如下所示:

```
>>> 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 在列的类型方面有些严格，您必须使用一个通常的 *unicode* 来作为 *name* 和 *email* 的类型，而不是普通的 *str* 类型。

查询也很简单:

```
>>> list(collection.User.find())
[<User u'admin'>]
>>> collection.User.find_one({'name': u'admin'})
<User u'admin'>
```

16.18.2PyMongo 兼容层

如果您想直接使用 PyMongo 。 您也可以利用 MongoKit 实现。如果您希望应用程序实现最佳的表现，您也许希望使用这种方法。注意，例子并没有展示配合 Flask 使用的具体方法。请参考上面 MongoKit 的例子代码:

```
from MongoKit import Connection


connection = Connection()
```

插入数据可以使用 *insert* 方法。我们必须先获得一个连接。这跟在 SQL 的世界使用表有些类似。

```
>>> collection = connection['test'].users
>>> user = {'name': u'admin', 'email': u'admin@localhost'}
>>> collection.insert(user)
```

print list(collection.find()) print collection.find_one({‘name’: u’admin’})

MongoKit 将会为我们自动提交修改。

查询数据库，您要直接使用数据库连接:

```
>>> 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'}
```

返回的结果也同样是类字典的对象:

```
>>> r = collection.find_one({'name': u'admin'})
>>> r['email']
u'admin@localhost'
```

关于 MongoKit 的更多信息，请访问 website.

16.19 在 Flask 中使用 MongoKit

近些日子，使用基于文档的数据库而不是基于表的关系数据库变得越来越流行。 这一方案展示了如何使用文档映射库 MongoKit ，来与 MongoDB 交互。

这一方案的使用需要一个可用的 MongoDB 服务器，并且安装有 MongoKit 库。

使用 MongoKit 有两种常用的方法，我们将会逐一介绍:

16.19.1 显式调用

MongoKit 的默认行为是这种显式调用的方法。这种方法跟 Django 或者 SQLAlchemy 扩展显示调用扩展大体精神是相同的。

下面是一个 *app.py* 模块的例子:

```
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'])
```

要定义您的模型，只需编写一个从 MongoKit 导入的 *Document* 类的子类。如果您已经看过了 SQLAlchemy 的方案，您可能会奇怪为什么这里没有一个会话，甚至没有定义 *init_db* 函数。一方面， MongoKit 并没有类似会话这种东西。这有时会增加代码量，但是同时也使得数据库操作非常高效。另一方面， MongoDB 是没有模式的。 这意味着您在相同的插入查询，可以使用不同的数据结构。 MongoKit 本身也是没有模式的。但是实现了一些用来确保数据完整的验证。

以下是一个文档的例子 (您可以将这个也放进 *app.py* 文件里):

```
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)

# register the User document with our current connection
connection.register([User])
```

这个例子向您展示了怎么定义您自己的结构(名为 structure)、一个最大字符长度的验证器以及使用 Monkit 的一项名为 *use_dot_notation* 的特性。某人情况下 MongoKit 按照字典的方式行为，但是将 *use_dot_notation* 为真之后，您可以像您在几乎所有的 ORM 当中那样，使用点运算符来分割属性的方式访问您的文档。

向数据库里添加数据的方法如下所示:

```
>>> 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 在列的类型方面有些严格，您必须使用一个通常的 *unicode* 来作为 *name* 和 *email* 的类型，而不是普通的 *str* 类型。

查询也很简单:

```
>>> list(collection.User.find())
[<User u'admin'>]
>>> collection.User.find_one({'name': u'admin'})
<User u'admin'>
```

16.19.2PyMongo 兼容层

如果您想直接使用 PyMongo 。 您也可以利用 MongoKit 实现。如果您希望应用程序实现最佳的表现，您也许希望使用这种方法。注意，例子并没有展示配合 Flask 使用的具体方法。请参考上面 MongoKit 的例子代码:

```
from MongoKit import Connection


connection = Connection()
```

插入数据可以使用 *insert* 方法。我们必须先获得一个连接。这跟在 SQL 的世界使用表有些类似。

```
>>> collection = connection['test'].users
>>> user = {'name': u'admin', 'email': u'admin@localhost'}
>>> collection.insert(user)
```

print list(collection.find()) print collection.find_one({‘name’: u’admin’})

MongoKit 将会为我们自动提交修改。

查询数据库，您要直接使用数据库连接:

```
>>> 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'}
```

返回的结果也同样是类字典的对象:

```
>>> r = collection.find_one({'name': u'admin'})

>>> r['email']

u'admin@localhost'
```

关于 MongoKit 的更多信息，请访问 website.

16.20 添加 Favicon

“Favicon” 是指您的网页浏览器显示在标签页或者历史记录里的图标。 这个图标能帮助用户将您的网站与其他网站区分开，因此请使用一个独特的标志

一个普遍的问题是如何将一个 Favicon 添加到您的 Flask 应用中。首先，您当然得先有一个可用的图标，此图标应该是 16 x 16 像素的，且格式为

ICO 。这些虽然不是必需的规则，但是是被所有浏览器所支持的事实标准。将这个图标放置到您的静态文件目录下，文件名为 `favicon.ico` 。

现在，为了让浏览器找到您的图标，正确的方法是添加一个 Link 标签到 HTML 当中例如:

```
<link rel="shortcut icon" href="{{ url_for('static',
filename='favicon.ico') }}">
```

对于大多数浏览器来说，这就足够了。然后一些非常老的浏览器不支持这个标准。 原来的标准是在网站的根路径下，查找 favicon 文件，并使用它。如果应用程序不是挂在在域名的根路径，您要么需要配置 Web 服务器来在根路径提供这一图标， 要么您就很不幸地无法实现这一功能了。然而，如果您饿应用是在根路径，您就可以简单的配置一条重定向的路由:

```
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，您也可以使用 `send_from_directory()` 函数写一个视图函数:

```
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')
```

我们可以不详细指定 mimetype ，浏览器将会自行猜测文件的类型。但是我们也可以指定它以便于避免额外的猜测，因为这个 mimetype 总是固定的。

以上的代码将会通过您的应用程序来提供图标文件的访问。然而，如果可能的话配置您的网页服务器来提供访问服务会更好。请参考对应网页服务器的文档。

16.20.1 参考

- Wikipedia 上有关 Favicon 的文章

16.21 数据流

有时，您希望发送非常巨量的数据到客户端，远远超过您可以保存在内存中的量。 在您实时地产生这些数据时，如何才能直接把他发送给客户端，而不需要在文件系统中中转呢?

答案是生成器和 Direct Response。

16.21.1 基本使用

下面是一个简单的视图函数，这一视图函数实时生成大量的 CSV 数据，这一技巧使用了一个内部函数，这一函数使用生成器来生成数据，并且稍后激发这个生成器函数时，把返回值传递给一个 response 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')
```

每一个 `yield` 表达式直接被发送给浏览器。现在，仍然有一些 WSGI 中间件可能打断数据流，所以在这里请注意那些在带缓存快照的调试环境，以及其他一些您可能激活了的东西。

16.21.2 在模板中生成流

Jinja2 模板引擎同样支持分块逐个渲染模板。Flask 没有直接暴露这一功能到模板中，因为它很少被用到，但是您可以很轻易的自己实现:

```
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))
```

这一技巧是从应用程序上的 Jinja2 的环境中得到那个模板对象，然后调用 `stream()` 函数而不是 `render()` 函数。前者返回的是一个流对象，而不是后者的字符串。因为我们绕过了 Flask 的模板渲染函数，而是直接使用了模板对象，所以我们手动必须调用 `update_template_context()` 函数来确保更新了模板的渲染上下文。 这一模板随后以流的方式迭代直到结束。因为每一次您使用使用一个 yield 。服务器都会将所有的已经产生的内容塞给给客户端，因可能希望在模板中缓冲一部分元素之后再发送，而不是每次都直接发送。您可以使用 `rv.enable_buffering(size)` 来实现，size 的较为合理的默认值是 `5` 。

16.22 延迟请求回调

Flask 的设计原则中有一条是响应对象被创建并在一条可能的回调链中传递，而在这条回调链但中的任意一个回调，您都可以修改或者替换掉他们。当请求开始被处理时，还没有响应对象，响应对象将在这一过程中，被某个视图函数或者系统的其他组件按照实际需要来闯将。

但是，如果您想在响应过程的结尾修改响应对象，但是这是对象还不存在，那么会发生什么呢？一个常见的例子是您可能需要在 before-request 函数当中在响应对象上设定 Cookie 。

解决这一情况的一个常用方法是改变代码的逻辑，将这一部分代码迁移到 after-request 回调中。然而有些时候这种迁移并不是一个非常容易的敬礼而且可能使代码看起来非常糟糕。

一个可能的替代方法是将一些回调函数绑定到 `g` 对象中。然后在请求结束的时候调用他们。使用这种方法，您可以从应用里的任何一个地方来指定代码延迟执行。

16.22.1 装饰器

下面的装饰器就是关键，它将一个函数注册到 `g` 对象上的一个函数列表中:

```
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
```

16.22.2 调用延迟函数

现在您可以使用 *after_this_request* 装饰器来将一个函数标记为在请求结束之后执行，但是我们仍然需要手动调用他们。为此，如下函数将被注册为 `after_request()` 回调:

```
@app.after_request
def call_after_request_callbacks(response):
```

```
    for callback in getattr(g, 'after_request_callbacks', ()):
        response = callback(response)
    return response
```

16.22.3 一个实际应用的例子

现在我们可以在任何时间点将一个函数注册为在某个特定请求结束后执行，例如您可以在 before-request 中将用户当前语言的信息保存在 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
```

16.23 添加 HTTP Method Overrides

某些 HTTP 代理不支持任意的 HTTP 方法或更新的 HTTP 方法（比如 PATCH）。 这种情况下，通过另一种完全违背协议的 HTTP 方法来“代理” HTTP 方法是可行的。

这个方法使客户端发出 HTTP POST 请求并设置 `X-HTTP-Method-Override` 标头的值为想要的 HTTP 方法（比如 `PATCH` ）。

这很容易通过一个 HTTP 中间件来完成:

```
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)
```

在 Flask 中使用它的必要步骤见下:

```
from flask import Flask


app = Flask(__name__)
app.wsgi_app = HTTPMethodOverrideMiddleware(app.wsgi_app)
```

16.24 请求内容校验码

许多代码可以消耗请求数据并对其进行预处理。例如最终出现在已读取的请求对象上的 JSON 数据、通过另外的代码路径出现的表单数据。当你想要校验收到的请求数据时，这似乎带来不便。而有时这对某些 API 是必要的。

幸运的是，无论如何可以包装输入流来简单地改变这种状况。

下面的例子计算收到数据的 SHA1 校验码，它从 WSGI 环境中读取数据并把校验码存放到其中:

```
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
```

要使用这段代码，所有你需要做的就是在请求消耗数据之前调用计算流。（例如： 小心访问 `request.form` 或其它此类的东西。例如，应注意避免 `before_request_handlers` 访问它）。

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
```

16.25 基于 Celery 的后台任务

Celery 是一个 Python 的任务队列，包含线程/进程池。曾经有一个 Flask 的集成， 但在 Celery 3 重构了内部细节后变得不必要了。本指导补充了如何妥善在 Flask 中使用 Celery 的空白，但假设你已经读过了 Celery 官方文档中的教程 使用 Celery 的首要步骤

16.25.1 安装 Celery

Celery 提交到了 Python Package Index (PyPI)，所以可以通过标准 Python 工具 `pip` 或 `easy_install` 安装:

```
$ pip install celery
```

16.25.2 配置 Celery

你需要的第一个东西是一个 Celery 实例，称为 Celery 应用。仅就 Celery 而言其与 Flask 中的 `Flask` 对象有异曲同工之妙。因为这个实例用于你在 Celery 中做任何事——诸如创建任务和管理职程（Worker）——的入口点，它必须可以在其它模块中导入。

例如，你可以把它放置到 `tasks` 模块中。虽然你可以在不重新配置 Flask 的情况下使用 Celery，但继承任务、添加对 Flask 应用上下文的支持以及关联 Flask 配置会让情况变得更好。

这就是把 Celery 集成到 Flask 的全部必要步骤:

```
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
```

该函数创建一个新的 Celery 对象，并用应用配置来配置中间人（Broker），用 Flask 配置更新其余的 Celery 配置，之后在应用上下文中创建一个封装任务执行的任务子类。

16.25.3 最简示例

通过上面的步骤，下面即是在 Flask 中使用 Celery 的最简示例:

```
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
```

这项任务可以在后台调用:

```
>>> result = add_together.delay(23, 42)
>>> result.wait()
65
```

16.25.4 运行 Celery 职程

现在如果你行动迅速，已经执行过了上述的代码，你会失望地得知 `.wait()` 永远不会实际地返回。这是因为你也需要运行 Celery。你可以这样把 Celery 以职程运行:

```
$ celery -A your_application worker
```

`your_application` 字符串需要指向创建 *celery* 对象的应用所在包或模块。

十七、部署选择

取决于你现有的，有多种途径来运行 Flask 应用。你可以在开发过程中使用内置的服务器，但是你应该为用于生产的应用选择使用完整的部署。（不要在生产环境中使用内置的开发服务器）。这里给出几个可选择的方法并且给出了文档。

如果你有一个不同的 WSGI 服务器，查阅文档中关于如何用它运行一个 WSGI 应用度部分。请记住你的 `Flask` 应用对象就是实际的 WSGI 应用。

选择托管服务来快速配置并运行，参阅快速上手中的 *部署到 Web 服务器* 部分。

- mod_wsgi (Apache)
  - 安装 *mod_wsgi*
  - 创建一个 *.wsgi* 文件
  - 配置 Apache
  - 故障排除
  - 自动重加载支持
  - 使用虚拟环境
- 独立 WSGI 容器
  - Gunicorn
  - Tornado
  - Gevent
  - 代理设置
- uWSGI

    - 用 uwsgi 启动你的应用
    - 配置 nginx
- FastCGI
    - 创建一个 *.fcgi* 文件
    - 配置 lighttpd
    - 配置 nginx
    - 运行 FastCGI 进程
    - 调试
- CGI
    - 创建一个 *.cgi* 文件
    - 服务器配置

17.1mod_wsgi (Apache)

如果你使用 Apache web 服务器，请考虑使用 mod_wsgi 。

注意

请确保在任何 `app.run()` 调用之前，你应该把应用文件放在一个 `if __name__ ==`__main__`:` 块中或移动到独立的文件。只确保它没被调用是因为这总是会启动一个本地的 WSGI 服务器，而当我们使用 mod_wsgi 部署应用时并不想让它出现。

17.1.1 安装 *mod_wsgi*

如果你还没有安装过 *mod_wsgi*，你需要使用包管理器来安装或手动编译它。 mod_wsgi 的 安装指引 涵盖了 UNIX 系统中的源码安装。

如果你使用 Ubuntu/Debian 你可以按照下面的命令使用 apt-get 获取并激活它:

```
# apt-get install libapache2-mod-wsgi
```

在 FreeBSD 上，通过编译 *www/mode_wsgi* port 或使用 pkg_add 来安装:

```
# pkg_add -r mod_wsgi
```

如果你在使用 pkgsrc 你可以编译 *www/ap2-wsgi* 包来安装 *mod_wsgi* 。

如果你在 apache 第一次重加载后遇到子进程段错误，你可以安全地忽略它们。 只需要重启服务器。

17.1.2 创建一个 *.wsgi* 文件

你需要一个 *yourapplication.wsgi* 文件来运行你的应用。这个文件包含 *mod_wsgi* 启动时执行的获取应用对象的代码。这个对象在该文件中名为 *application* ，并在之后作为应用。

对于大多数应用，下面度文件就可以胜任:

```
from yourapplication import app as application
```

如果你没有一个用于创建应用的工厂函数而是单例的应用，你可以直接导入它为 *application* 。

把这个文件放在你可以找到的地方（比如 */var/www/yourapplication* ）并确保 *yourapplication* 和所有使用的库在 python 载入的路径。如果你不想在系统全局安装它，请考虑使用 virtual python 实例。记住你也会需要在 virtualenv 中安装应用。可选地，你可以在 *.wsgi* 文件中在导入前修补路径:

```
import sys
sys.path.insert(0, '/path/to/the/application')
```

17.1.3 配置 Apache

你需要做的最后一件事情就是为你的应用创建一个 Apache 配置文件。在本例中，考虑安全因素，我们让 *mod_wsgi* 来在不同度用户下执行应用:

```
<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>
```

更多信息请翻阅 mod_wsgi wiki 。

17.1.4 故障排除

如果你的应用不能运行，按照下面的指导来排除故障:

问题: 应用不能运行，错误日志显示, SystemExit 忽略

你的应用文件中有一个不在 `if __name__ == '__main__':` 声明保护下的 `app.run()` 调用。把 `run()` 从文件中移除，或者把 它移到一个独立的 *run.py* 文件，再或者把它放到这样一个 if 块中。

问题: 应用报出权限错误

可能是因为使用了错误的用户运行应用。确保需要访问的应用有合适的权限设置，并且使用正确的用户来运行（*WSGIDaemonProcess* 指令的 `user` 和 `group` 参数）

问题: 应用崩溃时打印一条错误

记住 mod_wsgi 禁止对 `sys.stdout` 和 `sys.stderr` 做操作。 你可以通过设定配置中的 *WSGIRestrictStdout* 为 `off` 来禁用这个保护。

```
WSGIRestrictStdout Off
```

或者，你可以在 .wsgi 文件中用不同的流来替换标准输出:

```
import sys
```

```
sys.stdout = sys.stderr
```

问题: 访问资源时报出 IO 错误

你的应用可能是一个你符号链接到 site-packages 文件夹的单个 .py 文件。 请注意这不会正常工作，除非把这个文件放进 pythonpath 包含的文件夹中， 或是把应用转换成一个包。

这个问题同样适用于非安装的包，模块文件名用于定位资源，而符号链接会获取错误的文件名。

17.1.5 自动重加载支持

你可以激活自动重载入支持来协助部署工具。无论何时，当 *.wsgi* 文件， *mod_wsgi* 会为我们自动重新加载所有的守护进程。

为此，只需要直接在你的 *Directory* 节中添加如下内容:

```
WSGIScriptReloading On
```

17.1.6 使用虚拟环境

虚拟环境的优势是它们永远不在系统全局安装所需的依赖关系，这样你可以更好地控制使用什么。如果你想要同 mod_wsgi 使用虚拟环境，你需要稍微修改一下 *.wsgi* 文件。

把下面的几行添加到你 *.wsgi* 文件的顶部:

```
activate_this = '/path/to/env/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))
```

这根据虚拟环境的设置设定了加载路径。记住这个路径一经是绝对的。

17.2 独立 WSGI 容器

有用 Python 编写的流行服务器来容纳 WSGI 应用并提供 HTTP 服务。这些服务器在运行时是独立的：你可以从你的 web 服务器设置到它的代理。如果你遇见问题，请注意 *代理设置* 一节的内容。

17.2.1Gunicorn

Gunicorn 'Green Unicorn' 是一个给 UNIX 用的 WSGI HTTP 服务器。这是一个从 Ruby 的 Unicorn 项目移植的 pre-fork worker 模式。它既支持 eventlet，也支持 greenlet。在这个服务器上运行 Flask 应用是相当简单的:

```
gunicorn myproject:app
```

Gunicorn 提供了许多命令行选项 —— 见 `gunicorn -h`。 例如，用四个 worker 进程（`gunicorn -h`）来运行一个 Flask 应用，绑定到 localhost 的 4000 端口（`-b127.0.0.1:4000`）:

```
gunicorn -w 4 -b 127.0.0.1:4000 myproject:app
```

17.2.2Tornado

Tornado 是一个开源的可伸缩的、非阻塞式的 web 服务器和工具集，它驱动了 FriendFeed。因为它使用了 epoll 模型且是非阻塞的，它可以处

理数以千计的并发固定连接，这意味着它对实时 web 服务是理想的。把 Flask 集成这个服务是直截了当的:

```
from tornado.wsgi import WSGIContainer
from tornado.httpserver import HTTPServer
from tornado.ioloop import IOLoop
from yourapplication import app

http_server = HTTPServer(WSGIContainer(app))
http_server.listen(5000)
IOLoop.instance().start()
```

17.2.3Gevent

Gevent 是一个基于协同程序的 Python 网络库，使用 greenlet 来在 libevent 的事件循环上提供高层的同步 API

```
from gevent.wsgi import WSGIServer
from yourapplication import app

http_server = WSGIServer(('', 5000), app)
http_server.serve_forever()
```

17.2.4 代理设置

如果你在一个 HTTP 代理后把你的应用部署到这些服务器中的之一，你需要重写一些标头来让应用正常工作。在 WSGI 环境中两个有问题的值通常是 *REMOTE_ADDR* 和 *HTTP_HOST* 。你可以配置你的 httpd 来传

递这些标头，或者在中间件中手动修正。 Werkzeug 带有一个修正工具来解决常见的配置，但是你可能想要为特定的安装自己写 WSGI 中间件。

这是一个简单的 nginx 配置，它监听 localhost 的 8000 端口，并提供到一个应用的代理，设置了合适的标头:

```
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 不提供这些标头，最常见的配置引用从 *X-Forwarded-Host* 设置的主机名和从 *X-Forwarded-For* 设置的远程地址:

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

信任标头

请记住在一个非代理配置中使用这样一个中间件会是一个安全问题，因为它盲目地信任一个可能由恶意客户端伪造的标头。

如果你想从另一个标头重写标头，你可能会使用这样的一个修正程序:

```
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 = CustomProxyFix(app.wsgi_app)
```

17.3uWSGI

uWSGI 是在像 nginx 、lighttpd 以及 cherokee 服务器上的一个部署的选择。更多选择见 *FastCGI* 和 *独立 WSGI 容器*。 你会首先需要一个 uWSGI 服务器来用 uWSGI 协议来使用你的 WSGI 应用。 uWSGI 是一个协议，同样也是一个应用服务器，可以提供 uWSGI 、FastCGI 和 HTTP 协议。

最流行的 uWSGI 服务器是 uwsgi，我们会在本指导中使用。确保你已经安装好它来跟随下面的说明。

注意

请提前确保你在应用文件中的任何 `app.run()` 调用在 `if __name__ == '__main__':` 块中或是移到一个独立的文件。这是因为它总会启动一个本地的 WSGI 服务器，并且我们在部署应用到 uWSGI 时不需要它。

17.3.1 用 uwsgi 启动你的应用

uwsgi 被设计为操作在 python 模块中找到的 WSGI 可调用量。

已知在 myapp.py 中有一个 flask 应用，使用下面的命令:

```
$ uwsgi -s /tmp/uwsgi.sock --module myapp --callable app
```

或者，你喜欢这样:

```
$ uwsgi -s /tmp/uwsgi.sock -w myapp:app
```

17.3.2 配置 nginx

一个基本的 flaks uWSGI 的给 nginx 的配置看起来是这样:

```
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;
}
```

这个配置绑定应用到 */yourapplication* 。如果你想要绑定到 URL 根会更简单，因你不许要告诉它 WSGI *SCRIPT_NAME* 或设置 uwsgi modifier 来使用它:

```
location / { try_files $uri @yourapplication; }
location @yourapplication {
    include uwsgi_params;
    uwsgi_pass unix:/tmp/uwsgi.sock;
}
```

17.4FastCGI

FastCGI 是在像 nginx 、 lighttpd 和 cherokee 服务器上的一个部署选择。其它选择见 *uWSGI* 和 *独立 WSGI 容器* 章节。在它们上的任何一个运行

你的 WSGI 应用首先需要一个 FastCGI 服务器。最流行的一个是 flup，我们会在本指导中使用它。确保你已经安装好它来跟随下面的说明。

注意

请提前确保你在应用文件中的任何 `app.run()` 调用在 `if __name__ == '__main__':` 块中或是移到一个独立的文件。这是因为它总会启动一个本地的 WSGI 服务器，并且我们在部署应用到 uWSGI 时不需要它。

17.4.1 创建一个 *.fcgi* 文件

首先你需要创建一个 FastCGI 服务器文件。让我们把它叫做 *yourapplication.fcgi*:

```
#!/usr/bin/python
from flup.server.fcgi import WSGIServer
from yourapplication import app


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

这已经可以为 Apache 工作，而 nginx 和老版本的 lighttpd 需要传递一个显式的 socket 来与 FastCGI 通信。为此，你需要传递 socket 的路径到 `WSGIServer`:

```
WSGIServer(application, bindAddress='/path/to/fcgi.sock').run()
```

这个路径一定与你在服务器配置中定义的路径相同。

把 *yourapplication.fcgi* 文件保存到你能找到的地方。保存在 */var/www/yourapplication* 或类似的地方是有道理的。

确保这个文件有执行权限，这样服务器才能执行它:

```
# chmod +x /var/www/yourapplication/yourapplication.fcgi
```

17.4.2 配置 lighttpd

一个给 lighttpd 的基本的 FastCGI 配置看起来是这样:

```
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"
```

记得启用 FastCGI ，别名和重写模块。这份配置把应用绑定到 */yourapplication* 。如果想要应用运行在 URL 根路径，你需要用 `LighttpdCGIRootFix` 中间件来处理一个 lighttpd 的 bug 。

确保只在应用挂载到 URL 根路径时才应用它。同样，更多信息请翻阅 Lighty 的文档关于 FastCGI and Python 的部分（注意显示传递一个 socket 到 run() 不再是必须的）。

17.4.3 配置 nginx

在 nginx 上安装 FastCGI 应用有一点不同，因为默认没有 FastCGI 参数被转发。

一个给 nginx 的基本的 FastCGI 配置看起来是这样:

```
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;
}
```

这份配置把应用绑定到 */yourapplication* 。如果你想要绑定到 URL 跟了路径会更简单，因为你不需要指出如何获

取 *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;
}
```

17.4.4 运行 FastCGI 进程

既然 Nginx 和其它服务器并不加载 FastCGI 应用，你需要手动这么做。 Supervisor 可以管理 FastCGI 进程。 你可以寻找其它 FastCGI 进程管理器或写一个启动时运行 *.fcgi* 文件的脚本， 例如使用一个 SysV `init.d` 脚本。对于临时的解决方案，你总是可以在 GNU screen 中运行 `.fcgi` 。更多细节见 `man screen` ，注意这是一个手动的解决方案，并且不会在系统重启后保留:

```
$ screen
$ /var/www/yourapplication/yourapplication.fcgi
```

17.4.5 调试

FastCGI 在大多数 web 服务器上的部署，对于调试趋于复杂。服务器日志最经常告诉发生的事就是成行的“未预期的标头结尾”。为了调试应用，唯一可以让你了解什么东西破碎的方案就是切换到正确的用户并手动执行应用。

这个例子假设你的应用叫做 *application.fcgi* 并且你的 web 服务器用户是 *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
```

在这种情况下，错误看起来是“yourapplication”不在 python 路径下。常见的问题是:

- 使用了相对路径。不要依赖于当前工作目录
- 代码依赖于不是从 web 服务器设置的环境变量
- 使用了不同的 python 解释器

17.5CGI

如果所有其它的部署方式都不能奏效，那么 CGI 毫无疑问会奏效。 CGI 被所有主流服务器支持，但通常性能欠佳。

这也是你在 Google 的 App Engine 上使用 Flask 应用的方式，其执行方式恰好是一个 CGI-like 的环境。

> 注意
>
> 请提前确保你在应用文件中的任何 `app.run()` 调用在 `if __name__ == '__main__':` 块中或是移到一个独立的文件。这是因为它总会启动一个本地的 WSGI 服务器，并且我们在部署应用到 uWSGI 时不需要它。

17.5.1 创建一个 *.cgi* 文件

首先你需要创建一个 CGI 应用程序文件。我们把它叫做 *yourapplication.cgi*:

```
#!/usr/bin/python
from wsgiref.handlers import CGIHandler
from yourapplication import app

CGIHandler().run(app)
```

17.5.2 服务器配置

通常有两种方式来配置服务器。直接把 *.cgi* 复制到 *cgi-bin* （并且使用 *mod_rewrite* 或其它类似的东西来重写 URL ） 或让服务器直接指向这个文件。

例如，在 Apache 中你可以在配置中写入这样的语句:

```
ScriptAlias /app /path/to/the/application.cgi
```

更多信息请查阅你的 web 服务器的文档。

十八、聚沙成塔

这里是增长你的代码库或是扩大应用规模的一些选择。

18.1 阅读源码

Flask 的创建一定程度上是为了展示如何在现有的常用工具 Werkzeug（WSGI） 和 Jinja（模板）之上构建你自己的框架，并且当它开发出来之后，它对广大受众很有用。当你增长你的代码库时，不要仅仅使用 Flask——去理解它。阅读源码。 Flask 的代码是为了阅读而写；它是发布的文档，所以你可以使用它的内部 API。 Flask 坚持为上游库里的 API 写文档，并且为内部工具写文档，这样你们可以找到你的项目需要的钩子注册点。

18.2 钩子，继承

API 文档里面全都是可用的覆盖、钩子注册点和 *信号* 。 你可以提供诸如请求和响应对象的自定义类。深入你所用的 API，并且在 Flask 中探寻框架外可用的定制。去寻找把你的项目重构为实用工具集合和 Flask 扩展的方法，探索社区中的大量的 扩展， 如果你没找到你需要的工具，就去寻找可以用于构建你自己扩展的 模式。

18.3 继承

`Flask` 类有许多为继承设计的方法。你可以继承 `Flask` 快速添加或自定义行为（见链接的方法文档），并且无论你在哪里实例化一个应用类都会使用那个子类。这与 *应用程序的工厂函数* 工作良好。

18.4 用中间件包装

应用调度 章节描述了如何应用中间件的细节。你可以引入 WSGI 中间件来包装你的 Flask 实例并在 Flask 应用和 HTTP 服务器之间的中间层引入修正和变更。Werkzeug 包含了一些 中间件 。

18.5 分支

如果上述选择都不奏效，分支(fork) Flask。Flask 的大部分代码都限定在 Werkzeug 和 Jinja2 中。这些库做了大部分工作。Flask 只作为胶水把它们粘合在一起。 对每个项目，都有一个底层框架带来阻碍的点（归咎于原始开发者的假设）。 这很正常，因为如果不是这样，框架本身会是一个非常复杂的系统，导致学习曲线陡峭，给用户带来许多挫折。

不仅仅是对 Flask，许多人用打了补丁的或修改过的框架来弥补短处。这个思路也体现在 Flask 的许可证上。如果你决定修改这个框架，你不需要回馈任何的修改。

分支的消极面当然就是 Flask 扩展会更容易不可用，因为新的框架有一个不同的导入名称。此外，集成上游的修改可能是一个复杂的过程，取决于修改的数目。 为此，分支应该作为最后手段。

18.6 像专家一样扩大规模

对许多 web 应用，代码的复杂程度比起为预期的用户或数据条目而扩大规模就不是问题了。 Flask 自己扩大规模的限制只在于你的应用代码、你想用的数据存储和 Python 解释器以及你运行的 web 服务器。

良好的规模扩张意味着，如果你把服务器的数量加倍，你会得到大约两倍于原来的性能。 而糟糕的则意味着，当你添加了一台新的服务器，应用不会有任何性能提升或根本不支持第二台服务器。

在 Flask 中关于应用的扩张只有一个制约因素，那就是上下文局部代理。它们依赖于在 Flask 中上下文是被定义为是线程、还是进程或 greenlet。如果你的服务器使用不是基于线程或 greenlet 的并行计算， Flask 不再能支持这些全局代理。然而大多数服务器使用线程、 greenlet 或独立进程来实现并发，而这些方法在底层的 Werkzeug 库中有着良好的支持。

18.7 与社区对话

Flask 开发者维护框架对大小代码库用户的可理解性，所以一旦你遇到了由 Flask 引起的麻烦，不要犹豫，用邮件列表或 IRC 频道联系开发者。

Flask 和 Flask 扩展开发者为更大型应用改进的最佳途径就是从用户那里获取反馈。