

Flask 1.1.1 中文文档



Flask是一个使用Python编写的轻量级Web应用框架。基于Werkzeug WSGI工具箱和Jinja2 模板引擎。Flask使用BSD授权。 Flask被称为“microframework”，因为它使用简单的核心，用extension增加其他功能。Flask没有默认使...



下载手机APP
畅享精彩阅读

目 录

致谢

介绍

前言

针对高级程序员的前言

安装

快速上手

教程

项目布局

应用设置

定义和操作数据库

蓝图和视图

模板

静态文件

博客蓝图

项目可安装化

测试覆盖

部署产品

继续开发！

模板

测试 Flask 应用

应用错误处理

日志

配置管理

信号

可插拨视图

应用情境

请求情境

使用蓝图的模块化应用

扩展

命令行接口

开发服务器

在 Shell 中使用 Flask

Flask 方案

大型应用

应用工厂

应用调度

[实现 API 异常](#)

[URL 处理器](#)

[使用 Setuptools 部署](#)

[使用 Fabric 部署](#)

[使用 SQLite 3](#)

[使用 SQLAlchemy](#)

[上传文件](#)

[缓存](#)

[视图装饰器](#)

[使用 WTForms 进行表单验证](#)

[模板继承](#)

[消息闪现](#)

[通过 jQuery 使用 AJAX](#)

[自定义出错页面](#)

[惰性载入视图](#)

[通过 MongoEngine 使用 MongoDB](#)

[添加一个页面图标](#)

[流内容](#)

[延迟的请求回调](#)

[添加 HTTP 方法重载](#)

[请求内容校验](#)

[基于 Celery 的后台任务](#)

[继承 Flask](#)

[单页应用](#)

[部署方式](#)

[大型应用](#)

[API](#)

[Flask 的设计思路](#)

[HTML/XHTML 常见问答](#)

[安全注意事项](#)

[Flask 中的 Unicode](#)

[Flask 扩展开发](#)

[Pocoo 风格指南](#)

[Upgrading to Newer Releases](#)

[更新日志](#)

[License](#)

[如何为 Flask 做出贡献](#)

致谢

当前文档《Flask 1.1.1 中文文档》由 进击的皇虫 使用 书栈网(BookStack.CN) 进行构建，生成于 2020-01-02。

书栈网仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到书栈网，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到书栈网获取最新的文档，以跟上知识更新换代的步伐。

内容来源：[dormousehole](https://dormousehole.readthedocs.io/en/latest/) <https://dormousehole.readthedocs.io/en/latest/>

文档地址：<http://www.bookstack.cn/books/Flask-1.1.1-zh>

书栈官网：<https://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

欢迎来到 Flask 的世界



欢迎阅读 Flask 的文档。推荐您先阅读《[安装](#)》，然后阅读《[快速上手](#)》。《[教程](#)》比快速上手文档更详细一点，该文档介绍了如何创建一个完整（尽管很小）的 Flask 应用。《[Flask 方案](#)》中介绍了一些常用的解决方案。其余的文档详细介绍了 Flask 的每一个组件。《[API](#)》提供了最详细的参考。

Flask 依赖 [Jinja](#) 模板引擎和 [Werkzeug](#) WSGI 套件。这两个库的文档请移步：

- [Jinja 文档](#)
- [Werkzeug 文档](#)

用户指南

这部分文档是比较松散的，首先介绍了 Flask 的一些背景材料，然后专注于一步一步地说明如何使用 Flask 进行 Web 开发。

- [前言](#)
 - [“微”的含义](#)
 - [配置和惯例](#)
 - [可持续发展](#)
- [针对高级程序员的前言](#)
 - [Flask 中的本地线程对象](#)
 - [做网络开发时要谨慎](#)
- [安装](#)
 - [Python 版本](#)
 - [依赖](#)
 - [虚拟环境](#)
 - [安装 Flask](#)
 - [安装 virtualenv](#)
- [快速上手](#)
 - [一个最小的应用](#)
 - [如果服务器不能启动怎么办](#)
 - [调试模式](#)
 - [路由](#)
 - [静态文件](#)

- 渲染模板
- 操作请求数据
- 重定向和错误
- 关于响应
- 会话
- 消息闪现
- 日志
- 集成 WSGI 中间件
- 使用 Flask 扩展
- 部署到网络服务器
- 教程
 - 项目布局
 - 应用设置
 - 定义和操作数据库
 - 蓝图和视图
 - 模板
 - 静态文件
 - 博客蓝图
 - 项目可安装化
 - 测试覆盖
 - 部署产品
 - 继续开发！
- 模板
 - Jinja 设置
 - 标准环境
 - 标准过滤器
 - 控制自动转义
 - 注册过滤器
 - 环境处理器
- 测试 Flask 应用
 - 应用
 - 测试骨架
 - 第一个测试
 - 登录和注销
 - 测试添加消息
 - 其他测试技巧
 - 伪造资源和环境
 - 保持环境
 - 访问和修改会话
 - 测试 JSON API
 - 测试 CLI 命令
- 应用错误处理
 - 错误日志工具
 - 错误处理
 - 日志
- 排除应用错误

- 有疑问时，请手动运行
- 使用调试器
- 日志
 - 基本配置
 - 把出错信息通过电子邮件发送给管理者
 - 注入请求信息
 - 其他库
- 配置管理
 - 配置入门
 - 环境和调试特征
 - 内置配置变量
 - 使用配置文件
 - 使用环境变量来配置
 - 配置的最佳实践
 - 开发/生产
 - 实例文件夹
- 信号
 - 订阅信号
 - 创建信号
 - 发送信号
 - 信号与 Flask 的请求环境
 - 信号订阅装饰器
 - 核心信号
- 可插拔视图
 - 基本原理
 - 方法提示
 - 基于方法调度
 - 装饰视图
 - 用于 API 的方法视图
- 应用情境
 - 情境的目的
 - 情境的生命周期
 - 手动推送情境
 - 存储数据
 - 事件和信号
- 请求情境
 - 情境的用途
 - 情境的生命周期
 - 手动推送情境
 - 情境如何工作
 - 回调和错误
 - 出错情境保存
 - 关于代理的说明
- 使用蓝图的模块化应用
 - 为什么使用蓝图？
 - 蓝图的概念

- [第一个蓝图](#)
- [注册蓝图](#)
- [蓝图资源](#)
- [创建 URL](#)
- [错误处理器](#)
- [扩展](#)
 - [寻找扩展](#)
 - [使用扩展](#)
 - [创建扩展](#)
- [命令行接口](#)
 - [探索应用](#)
 - [运行开发服务器](#)
 - [打开一个 Shell](#)
 - [环境](#)
 - [调试模式](#)
 - [通过 dotenv 设置环境变量](#)
 - [通过 virtualenv 设置环境变量](#)
 - [自定义命令](#)
 - [插件](#)
 - [自定义脚本](#)
 - [PyCharm 集成](#)
- [开发服务器](#)
 - [通过命令行使用开发服务器](#)
 - [通过代码使用开发服务器](#)
- [在 Shell 中使用 Flask](#)
 - [命令行接口](#)
 - [创建一个请求情境](#)
 - [发送请求前/后动作](#)
 - [在 Shell 中玩得更爽](#)
- [Flask 方案](#)
 - [大型应用](#)
 - [应用工厂](#)
 - [应用调度](#)
 - [实现 API 异常](#)
 - [URL 处理器](#)
 - [使用 Setuptools 部署](#)
 - [使用 Fabric 部署](#)
 - [使用 SQLite 3](#)
 - [使用 SQLAlchemy](#)
 - [上传文件](#)
 - [缓存](#)
 - [视图装饰器](#)
 - [使用 WTForms 进行表单验证](#)
 - [模板继承](#)
 - [消息闪现](#)
 - [通过 jQuery 使用 AJAX](#)

- [自定义出错页面](#)
- [惰性载入视图](#)
- [通过 MongoEngine 使用 MongoDB](#)
- [添加一个页面图标](#)
- [流内容](#)
- [延迟的请求回调](#)
- [添加 HTTP 方法重载](#)
- [请求内容校验](#)
- [基于 Celery 的后台任务](#)
- [继承 Flask](#)
- [单页应用](#)
- [部署方式](#)
 - [托管选项](#)
 - [自主部署选项](#)
- [大型应用](#)
 - [阅读源代码](#)
 - [挂接, 扩展](#)
 - [继承](#)
 - [用中间件包装](#)
 - [派生](#)
 - [专家级的伸缩性](#)
 - [与社区沟通](#)

API 参考

这部分文档详细说明某个函数、类或方法。

- [API](#)
 - [Application Object](#)
 - [Blueprint Objects](#)
 - [Incoming Request Data](#)
 - [Response Objects](#)
 - [Sessions](#)
 - [Session Interface](#)
 - [Test Client](#)
 - [Test CLI Runner](#)
 - [Application Globals](#)
 - [Useful Functions and Classes](#)
 - [Message Flashing](#)
 - [JSON Support](#)
 - [Template Rendering](#)
 - [Configuration](#)
 - [Stream Helpers](#)
 - [Useful Internals](#)
 - [Signals](#)
 - [Class-Based Views](#)

- [URL Route Registrations](#)
- [View Function Options](#)
- [Command Line Interface](#)

其他材料

这部分文档包括：设计要点、法律信息和变动记录。

- [Flask 的设计思路](#)
 - [显式的应用对象](#)
 - [路由系统](#)
 - [唯一模板引擎](#)
 - [我依赖所以我微](#)
 - [线程本地对象](#)
 - [Flask 是什么，不是什么](#)
- [HTML/XHTML 常见问答](#)
 - [XHTML 的历史](#)
 - [HTML5 的历史](#)
 - [HTML 对比 XHTML](#)
 - [“严格”意味着什么？](#)
 - [HTML5 中的新技术](#)
 - [应该使用什么？](#)
- [安全注意事项](#)
 - [跨站脚本攻击（XSS）](#)
 - [跨站请求伪造（CSRF）](#)
 - [JSON 安全](#)
 - [安全头部](#)
- [Flask 中的 Unicode](#)
 - [自动转换](#)
 - [金科玉律](#)
 - [自助编码和解码](#)
 - [配置编辑器](#)
- [Flask 扩展开发](#)
 - [剖析一个扩展](#)
 - [“Hello Flaskext!”](#)
 - [初始化扩展](#)
 - [扩展的代码](#)
 - [使用 _app_ctx_stack](#)
 - [学习借鉴](#)
 - [已审核的扩展](#)
- [Pocoo 风格指南](#)
 - [总体布局](#)
 - [表达式和语句](#)
 - [命名约定](#)
 - [文档字符串](#)
 - [注释](#)

- [Upgrading to Newer Releases](#)

- [Version 0.12](#)
- [Version 0.11](#)
- [Version 0.10](#)
- [Version 0.9](#)
- [Version 0.8](#)
- [Version 0.7](#)
- [Version 0.6](#)
- [Version 0.5](#)
- [Version 0.4](#)
- [Version 0.3](#)

- [更新日志](#)

- [Version 1.1.1](#)
- [Version 1.1.0](#)
- [Version 1.0.4](#)
- [Version 1.0.3](#)
- [Version 1.0.2](#)
- [Version 1.0.1](#)
- [Version 1.0](#)
- [Version 0.12.4](#)
- [Version 0.12.3](#)
- [Version 0.12.2](#)
- [Version 0.12.1](#)
- [Version 0.12](#)
- [Version 0.11.1](#)
- [Version 0.11](#)
- [Version 0.10.1](#)
- [Version 0.10](#)
- [Version 0.9](#)
- [Version 0.8.1](#)
- [Version 0.8](#)
- [Version 0.7.2](#)
- [Version 0.7.1](#)
- [Version 0.7](#)
- [Version 0.6.1](#)
- [Version 0.6](#)
- [Version 0.5.2](#)
- [Version 0.5.1](#)
- [Version 0.5](#)
- [Version 0.4](#)
- [Version 0.3.1](#)
- [Version 0.3](#)
- [Version 0.2](#)
- [Version 0.1](#)

- [License](#)

- [Source License](#)

- [Artwork License](#)
- [如何为 Flask 做出贡献](#)
 - [问答支持](#)
 - [报告问题](#)
 - [提交补丁](#)
 - [注意：零填充文件模式](#)

前言

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

“微”的含义

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

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

配置和惯例

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

可持续发展

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

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

Flask 包含许多可以自定义其行为的钩子。考虑到你的定制需求，Flask 的类专为继承而打造。如果对这一点感兴趣，请阅读 [大型应用](#) 一节。如果对 Flask 的设计原则感兴趣，请移步 [Flask 的设计思路](#)。

接下来请阅读 [安装](#)、[快速上手](#) 或者 [针对高级程序员的前言](#)。

针对高级程序员的前言

Flask 中的本地线程对象

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

做网络开发时要谨慎

做网络应用开发时，安全要永记在心。

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

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

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

Flask 与其他框架相同，你在开发时必须小心谨慎。

安装

Python 版本

我们推荐使用最新版本的 Python 3 。 Flask 支持 Python 3.4 及更高版本的Python 3 、 Python 2.7 和 PyPy 。

依赖

当安装 Flask 时，以下配套软件会被自动安装。

- [Werkzeug](#) 用于实现 WSGI ，应用和服务之间的标准 Python 接口。
- [Jinja](#) 用于渲染页面的模板语言。
- [MarkupSafe](#) 与 Jinja 共用，在渲染页面时用于避免不可信的输入，防止注入攻击。
- [ItsDangerous](#) 保证数据完整性的安全标志数据，用于保护 Flask 的 session cookie 。
- [Click](#) 是一个命令行应用的框架。用于提供 `flask` 命令，并允许添加自定义管理命令。

可选依赖

以下配套软件不会被自动安装。如果安装了，那么 Flask 会检测到这些软件。

- [Blinker](#) 为 [信号](#) 提供支持。
- [SimpleJSON](#) 是一个快速的 JSON 实现，兼容 Python's `json` 模块。如果安装了这个软件，那么会优先使用这个软件来进行 JSON 操作。
- [python-dotenv](#) 当运行 `flask` 命令时为 [通过 dotenv 设置环境变量](#) 提供支持。
- [Watchdog](#) 为开发服务器提供快速高效的重载。

虚拟环境

建议在开发环境和生产环境下都使用虚拟环境来管理项目的依赖。

为什么要使用虚拟环境？随着你的 Python 项目越来越多，你会发现不同的项目会需要不同的版本的 Python 库。同一个 Python 库的不同版本可能不兼容。

虚拟环境可以为每一个项目安装独立的 Python 库，这样就可以隔离不同项目之间的Python 库，也可以隔离项目与操作系统之间的 Python 库。

Python 3 内置了用于创建虚拟环境的 `venv` 模块。如果你使用的是较新的Python 版本，那么请接着阅读本文下面的内容。

如果你使用 Python 2 ，请首先参阅 [安装 virtualenv](#) 。

创建一个虚拟环境

创建一个项目文件夹，然后创建一个虚拟环境。创建完成后项目文件夹中会有一个 `venv` 文件夹：

```
1. $ mkdir myproject
2. $ cd myproject
3. $ python3 -m venv venv
```

在 Windows 下：

```
1. $ py -3 -m venv venv
```

在老版本的 Python 中要使用下面的命令创建虚拟环境：

```
1. $ python2 -m virtualenv venv
```

在 Windows 下：

```
1. > \Python27\Scripts\virtualenv.exe venv
```

激活虚拟环境

在开始工作前，先要激活相应的虚拟环境：

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

在 Windows 下：

```
1. > venv\Scripts\activate
```

激活后，你的终端提示符会显示虚拟环境的名称。

安装 Flask

在已激活的虚拟环境中可以使用如下命令安装 Flask：

```
1. $ pip install Flask
```

Flask 现在已经安装完毕。请阅读 [快速上手](#) 或者[文档目录](#) 。

与时俱进

如果想要在正式发行之前使用最新的 Flask 开发版本，可以使用如下命令从主分支安装或者更新代码：

```
1. $ pip install -U https://github.com/pallets/flask/archive/master.tar.gz
```

安装 virtualenv

如果你使用的是 Python 2，那么 venv 模块无法使用。相应的，必须安装 [virtualenv](#)。

在 Linux 下，virtualenv 通过操作系统的包管理器安装：

```
1. # Debian, Ubuntu
2. $ sudo apt-get install python-virtualenv
3.
4. # CentOS, Fedora
5. $ sudo yum install python-virtualenv
6.
7. # Arch
8. $ sudo pacman -S python-virtualenv
```

如果是 Mac OS X 或者 Windows，下载 [get-pip.py](#)，然后：

```
1. $ sudo python2 Downloads/get-pip.py
2. $ sudo python2 -m pip install virtualenv
```

在 Windows 下，需要要 administrator 权限：

```
1. > \Python27\python.exe Downloads\get-pip.py
2. > \Python27\python.exe -m pip install virtualenv
```

现在可以返回上面，[创建一个虚拟环境](#)。

快速上手

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

一个最小的应用

一个最小的 Flask 应用如下：

```
1. from flask import Flask
2. app = Flask(__name__)
3.
4. @app.route('/')
5. def hello_world():
6.     return 'Hello, World!'
```

那么，这些代码是什么意思呢？

- 首先我们导入了 `Flask` 类。该类的实例将会成为我们的 WSGI 应用。
- 接着我们创建一个该类的实例。第一个参数是应用模块或者包的名称。如果你使用一个单一模块（就像本例），那么应当使用 `__name__` ，因为名称会根据这个模块是按应用方式使用还是作为一个模块导入而发生变化（可能是 `'main'` ，也可能是实际导入的名称）。这个参数是必需的，这样 Flask 才能知道在哪里可以找到模板和静态文件等东西。更多内容详见 `Flask` 文档。
- 然后我们使用 `route()` 装饰器来告诉 Flask 触发函数的 URL 。
- 函数名称被用于生成相关联的 URL 。函数最后返回需要在用户浏览器中显示的信息。

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

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

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

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

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

在 PowerShell 下：

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

还可以使用 `python -m flask`：

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

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

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

外部可见的服务器

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

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

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

这行代码告诉你的操作系统监听所有公开的 IP 。

如果服务器不能启动怎么办

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

老版本的 Flask

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

非法导入名称

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

最常见的错误是因为拼写错误而没有真正创建一个 `app` 对象。

调试模式

（只需要记录出错信息和追踪堆栈？参见 [应用错误处理](#)）

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

出错时还会提供一个有用的调试器。

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

```
1. $ export FLASK_ENV=development
2. $ flask run
```

（在 Windows 下需要使用 `set` 来代替 `export`。）

这样可以实现以下功能：

- 激活调试器。
- 激活自动重载。
- 打开 Flask 应用的调试模式。

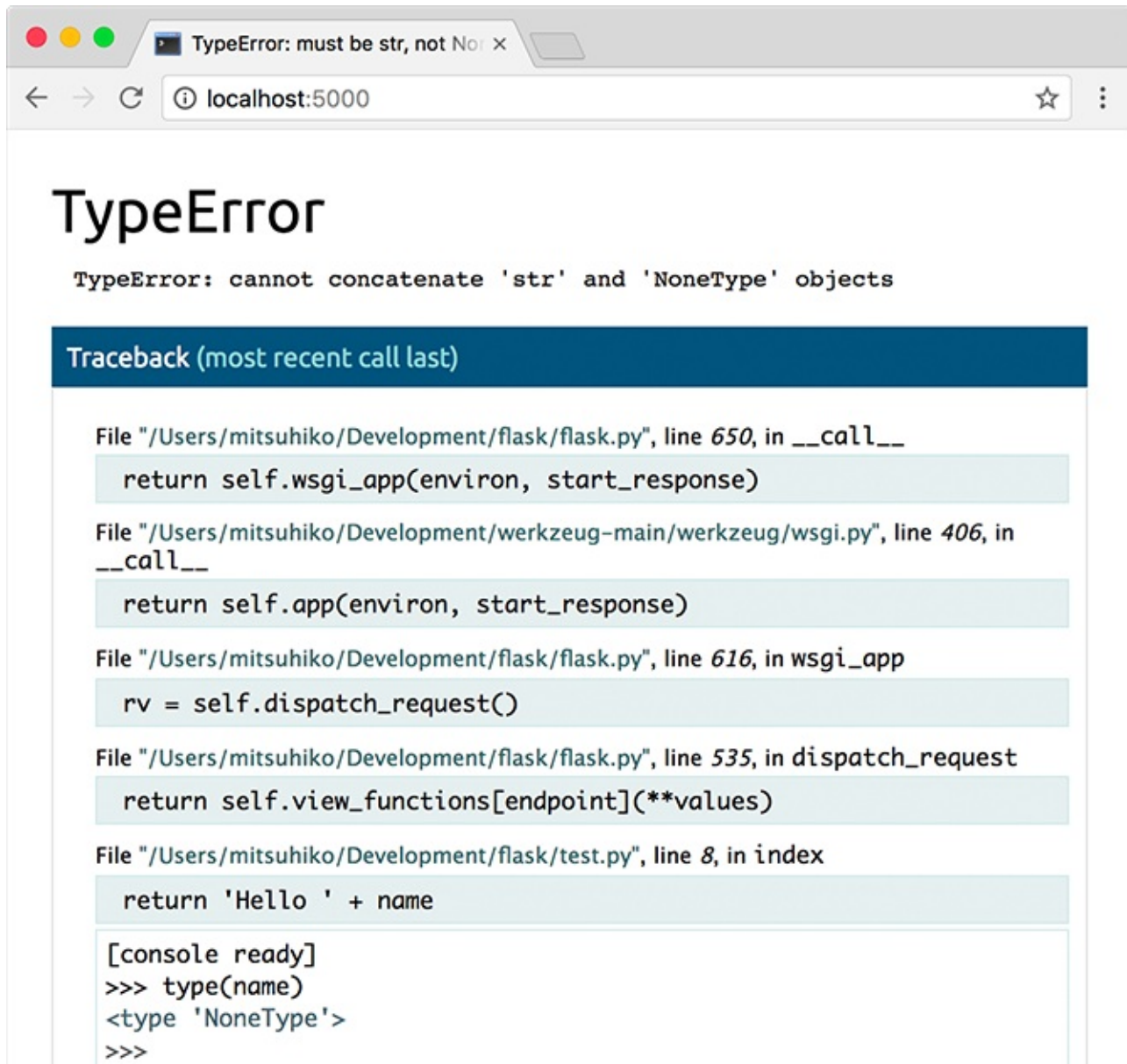
还可以通过导出 `FLASK_DEBUG=1` 来单独控制调试模式的开关。

[开发服务器](#) 文档中还有更多的参数说明。

Attention

虽然交互调试器不能在分布环境下工作（这使得它基本不可能用于生产环境），但是它允许执行任意代码，这样会成为一重大安全隐患。因此，绝对不能在生产环境中使用调试器。

运行中的调试器截图：


[更多关于](#)

调试器的信息参见 [Werkzeug 文档](#)。

想使用其他调试器？请参阅 [使用调试器](#)。

路由

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

使用 `route()` 装饰器来把函数绑定到 URL：

```
1. @app.route('/')
2. def index():
3.     return 'Index Page'
4.
5. @app.route('/hello')
6. def hello():
7.     return 'Hello, World'
```

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

变量规则

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

```
1. @app.route('/user/<username>')
2. def show_user_profile(username):
3.     # show the user profile for that user
4.     return 'User %s' % escape(username)
5.
6. @app.route('/post/<int:post_id>')
7. def show_post(post_id):
8.     # show the post with the given id, the id is an integer
9.     return 'Post %d' % post_id
10.
11. @app.route('/path/<path:subpath>')
12. def show_subpath(subpath):
13.     # show the subpath after /path/
14.     return 'Subpath %s' % escape(subpath)
```

转换器类型：

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

唯一的 URL / 重定向行为

以下两条规则的不同之处在于是否使用尾部的斜杠。：

```
1. @app.route('/projects/')
2. def projects():
3.     return 'The project page'
4.
5. @app.route('/about')
6. def about():
7.     return 'The about page'
```

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

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

URL 构建

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

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

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

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

```
1. from flask import Flask, escape, url_for
2.
3. app = Flask(__name__)
4.
5. @app.route('/')
6. def index():
7.     return 'index'
8.
9. @app.route('/login')
10. def login():
11.     return 'login'
12.
13. @app.route('/user/<username>')
14. def profile(username):
15.     return '{}\s profile'.format(escape(username))
16.
17. with app.test_request_context():
18.     print(url_for('index'))
19.     print(url_for('login'))
20.     print(url_for('login', next='/'))
21.     print(url_for('profile', username='John Doe'))
```

```
1. /
2. /login
3. /login?next=/
4. /user/John%20Doe
```

HTTP 方法

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

```
1. from flask import request
2.
3. @app.route('/login', methods=['GET', 'POST'])
4. def login():
5.     if request.method == 'POST':
6.         return do_the_login()
7.     else:
8.         return show_the_login_form()
```

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

静态文件

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

使用特定的 `'static'` 端点就可以生成相应的 URL

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

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

渲染模板

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

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

```
1. from flask import render_template
2.
3. @app.route('/hello/')
4. @app.route('/hello/<name>')
5. def hello(name=None):
6.     return render_template('hello.html', name=name)
```

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

情形 1：一个模块：


```

1. /application.py
2. /templates
3.     /hello.html

```

情形 2 ： 一个包：

```

1. /application
2.     /__init__.py
3.     /templates
4.         /hello.html

```

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

模板示例：

```

1. <!doctype html>
2. <title>Hello from Flask</title>
3. {% if name %}
4.     <h1>Hello {{ name }}!</h1>
5. {% else %}
6.     <h1>Hello, World!</h1>
7. {% endif %}

```

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

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

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

下面 `Markup` 类的基本使用方法：

```

1. >>> from flask import Markup
2. >>> Markup('<strong>Hello %s!</strong>') % '<blink>hacker</blink>'
3. Markup(u'<strong>Hello &lt;blink&gt;hacker&lt;/blink&gt;!</strong>')
4. >>> Markup.escape('<blink>hacker</blink>')
5. Markup(u'&lt;blink&gt;hacker&lt;/blink&gt;')
6. >>> Markup('<em>Marked up</em> &raquo; HTML').striptags()
7. u'Marked up \xbbb HTML'

```

Changelog

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

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

操作请求数据

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

本地环境

内部信息

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

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

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

这对你有什么用？基本上你可以完全不必理会。这个只有在做单元测试时才有用。在测试时会遇到由于没有请求对象而导致依赖于请求的代码会突然崩溃的情况。对策是自己创建一个请求对象并绑定到环境。最简单的单元测试解决方案是使用 `test_request_context()` 环境管理器。通过使用 `with` 语句可以绑定一个测试请求，以便于交互。例如：

```
1. from flask import request
2.
3. with app.test_request_context('/hello', method='POST'):
4.     # now you can do something with the request until the
5.     # end of the with block, such as basic assertions:
6.     assert request.path == '/hello'
7.     assert request.method == 'POST'
```

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

```
1. from flask import request
2.
3. with app.request_context(environ):
4.     assert request.method == 'POST'
```

请求对象

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

```
1. from flask import request
```

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

```
1. @app.route('/login', methods=['POST', 'GET'])
2. def login():
3.     error = None
4.     if request.method == 'POST':
5.         if valid_login(request.form['username'],
6.                         request.form['password']):
7.             return log_the_user_in(request.form['username'])
8.         else:
9.             error = 'Invalid username/password'
10.    # the code below is executed if the request method
11.    # was GET or the credentials were invalid
12.    return render_template('login.html', error=error)
```

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

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

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

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

完整的请求对象方法和属性参见 `Request` 文档。

文件上传

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

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

```
1. from flask import request
2.
3. @app.route('/upload', methods=['GET', 'POST'])
4. def upload_file():
5.     if request.method == 'POST':
6.         f = request.files['the_file']
7.         f.save('/var/www/uploads/uploaded_file.txt')
8.     ...
```

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

```
1. from flask import request
2. from werkzeug.utils import secure_filename
3.
4. @app.route('/upload', methods=['GET', 'POST'])
5. def upload_file():
6.     if request.method == 'POST':
7.         f = request.files['the_file']
8.         f.save('/var/www/uploads/' + secure_filename(f.filename))
9.     ...
```

更好的例子参见 [上传文件](#) 方案。

Cookies

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

读取 cookies：

```
1. from flask import request
2.
3. @app.route('/')
4. def index():
5.     username = request.cookies.get('username')
6.     # use cookies.get(key) instead of cookies[key] to not get a
7.     # KeyError if the cookie is missing.
```

储存 cookies：

```
1. from flask import make_response
2.
3. @app.route('/')
4. def index():
5.     resp = make_response(render_template(...))
6.     resp.set_cookie('username', 'the username')
7.     return resp
```

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

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

同时可以参见 [关于响应](#)。

重定向和错误

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

```
1. from flask import abort, redirect, url_for
2.
3. @app.route('/')
4. def index():
5.     return redirect(url_for('login'))
6.
7. @app.route('/login')
8. def login():
9.     abort(401)
10.     this_is_never_executed()
```

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

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

```
1. from flask import render_template
2.
3. @app.errorhandler(404)
4. def page_not_found(error):
5.     return render_template('page_not_found.html'), 404
```

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

详见 [错误处理](#)。

关于响应

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

- 如果视图返回的是一个响应对象，那么就直接返回它。
- 如果返回的是一个字符串，那么根据这个字符串和缺省参数生成一个用于返回的响应对象。
- 如果返回的是一个字典，那么调用 `jsonify` 创建一个响应对象。
- 如果返回的是一个元组，那么元组中的项目可以提供额外的信息。元组中必须至少包含一个项目，且项目应当由 `(response, status)`、`(response, headers)` 或者 `(response, status, headers)` 组成。`status` 的值会重载状态代码，`headers` 是一个由额外头部值组成的列表或字典。
- 如果以上都不是，那么 Flask 会假定返回值是一个有效的 WSGI 应用并把它转换为一个响应对象。

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

设想有如下视图：

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

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

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

JSON 格式的 API

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

```
1. @app.route("/me")
2. def me_api():
3.     user = get_current_user()
4.     return {
5.         "username": user.username,
6.         "theme": user.theme,
7.         "image": url_for("user_image", filename=user.image),
8.     }
```

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

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

会话

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

使用会话之前你必须设置一个密钥。举例说明：

```
1. from flask import Flask, session, redirect, url_for, escape, request
2.
```

```

3. app = Flask(__name__)
4.
5. # Set the secret key to some random bytes. Keep this really secret!
6. app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'
7.
8. @app.route('/')
9. def index():
10.     if 'username' in session:
11.         return 'Logged in as %s' % escape(session['username'])
12.     return 'You are not logged in'
13.
14. @app.route('/login', methods=['GET', 'POST'])
15. def login():
16.     if request.method == 'POST':
17.         session['username'] = request.form['username']
18.         return redirect(url_for('index'))
19.     return '''
20.         <form method="post">
21.             <p><input type="text" name="username">
22.             <p><input type="submit" value="Login">
23.         </form>
24.     '''
25.
26. @app.route('/logout')
27. def logout():
28.     # remove the username from the session if it's there
29.     session.pop('username', None)
30.     return redirect(url_for('index'))

```

这里用到的 `escape()` 是用来转义的。如果不使用模板引擎就可以像上例一样使用这个函数来转义。

如何生成一个好的密钥

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

```

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

```

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

除了缺省的客户端会话之外，还有许多 Flask 扩展支持服务端会话。

消息闪现

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

一个布局模板来展现闪现的消息。

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

日志

Changelog

New in version 0.3.

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

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

以下是一些日志调用示例：

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

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

更多内容请参阅 [应用错误处理](#)。

集成 WSGI 中间件

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

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

使用 Flask 扩展

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

更多关于 `Flask` 扩展的内容请参阅 [扩展](#)。

部署到网络服务器

已经准备好部署你的新 `Flask` 应用了？请移步 [部署方式](#)。

教程

Contents:

- [项目布局](#)
- [应用设置](#)
- [定义和操作数据库](#)
- [蓝图和视图](#)
- [模板](#)
- [静态文件](#)
- [博客蓝图](#)
- [项目可安装化](#)
- [测试覆盖](#)
- [部署产品](#)
- [继续开发！](#)

本教程中我们将会创建一个名为 Flaskr 的具备基本功能的博客应用。应用用户可以注册、登录、发贴和编辑或者删除自己的帖子。可以打包这个应用并且安装到其他电脑上。



本文假设你已经熟悉 Python 。不熟悉？那么建议先从学习或者复习 Python 文档的 [官方教程](#) 入手。

本教程的目的是提供一个良好的起点，因此不会涵盖 Flask 的所有内容。如果了解 Flask 能够做什么，可以通过 [快速上手](#) 作一个大概的了解，想深入了解的话那就只有仔细阅读所有文档了。本教程只会涉及 Flask 和 Python 。在实际项目中可以通过使用 [扩展](#) 或者其他的库，达到事半功倍的效果。

A screenshot of a web application interface for logging in. At the top, there is a header bar with the word "Flaskr" on the left and two links, "Register" and "Log In", on the right. Below the header, the main content area has a title "Log In" followed by a horizontal line. Underneath, there are two labels: "Username" and "Password", each followed by a text input field. At the bottom of the form is a button labeled "Log In".

Flaskr [Register](#) [Log In](#)

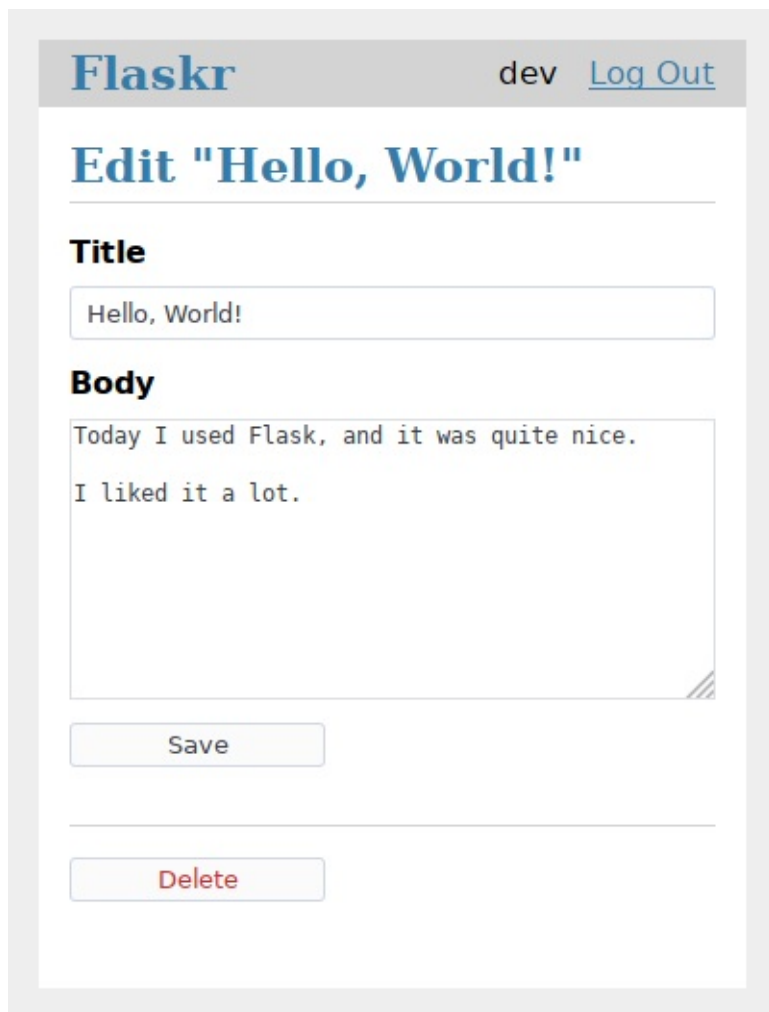
Log In

Username

Password

Log In

Flask 是非常灵活的，不需要使用任何特定的项目或者代码布局。但是对于初学者，使用结构化的方法是有益无害的，亦即本教程会有一点样板的意思。本教程可以让初学者避免一些常见的陷阱，并且完成后的应用可以方便的扩展。一旦熟悉了 Flask 之后就可以跳出这个结构，充分享受 Flask 的灵活性。

A screenshot of a web application interface for editing a post. At the top, there is a header bar with the word "Flaskr" on the left, the word "dev" in the middle, and a link "Log Out" on the right. Below the header, the main content area has a title "Edit 'Hello, World!'" followed by a horizontal line. Underneath, there are two labels: "Title" and "Body". The "Title" label is followed by a text input field containing the text "Hello, World!". The "Body" label is followed by a larger text area containing the text "Today I used Flask, and it was quite nice. I liked it a lot." At the bottom of the form are two buttons: "Save" and "Delete".

Flaskr dev [Log Out](#)

Edit "Hello, World!"

Title

Body

Today I used Flask, and it was quite nice.
I liked it a lot.

Save

Delete

如果在学习教程过程中需要比较项目代码与最终结果的差异，那么可以在[Flask 官方资源库的示例](#) 中找到完成的教程项目代码。

下面请阅读 [项目布局](#) 。

项目布局

创建并进入项目文件夹：

```
1. $ mkdir flask-tutorial
2. $ cd flask-tutorial
```

接下来按照 [安装简介](#) 设置一个 Python 虚拟环境，然后为项目安装 Flask 。

本教程假定项目文件夹名称为 `flask-tutorial`，本教程中代码块的顶端的文件名是基于该文件夹的相对名称。

一个最简单的 Flask 应用可以是单个文件。

`hello.py`

```
1. from flask import Flask
2.
3. app = Flask(__name__)
4.
5.
6. @app.route('/')
7. def hello():
8.     return 'Hello, World!'
```

然而，当项目越来越大的时候，把所有代码放在单个文件中就有点不堪重负了。Python 项目使用 包 来管理代码，把代码分为不同的模块，然后在需要的地方导入模块。本教程也会按这一方式管理代码。

教程项目包含如下内容：

- `flaskr/`，一个包含应用代码和文件的 Python 包。
- `tests/`，一个包含测试模块的文件夹。
- `venv/`，一个 Python 虚拟环境，用于安装 Flask 和其他依赖的包。
- 告诉 Python 如何安装项目的安装文件。
- 版本控制配置，如 [git](#)。不管项目大小，应当养成使用版本控制的习惯。
- 项目需要的其他文件。

最后，项目布局如下：

```
1. /home/user/Projects/flask-tutorial
2. └─ flaskr/
3.   └─ __init__.py
4.   └─ db.py
5.   └─ schema.sql
6.   └─ auth.py
```

```
7. | | └─ blog.py
8. | | └─ templates/
9. | |   └─ base.html
10. | |   └─ auth/
11. | |     └─ login.html
12. | |     └─ register.html
13. | |     └─ blog/
14. | |       └─ create.html
15. | |       └─ index.html
16. | |       └─ update.html
17. | └─ static/
18. |   └─ style.css
19. └─ tests/
20. | └─ conftest.py
21. | └─ data.sql
22. | └─ test_factory.py
23. | └─ test_db.py
24. | └─ test_auth.py
25. | └─ test_blog.py
26. └─ venv/
27. └─ setup.py
28. └─ MANIFEST.in
```

如果使用了版本控制，那么应当忽略运行项目时产生的临时文件以及编辑代码时编辑器产生的临时文件。忽略文件的基本原则是：不是你自己写的文件就可以忽略。举例来说，假设使用 `git` 来进行版本控制，那么使用 `.gitignore` 来设置应当忽略的文件，`.gitignore` 文件应当与下面类似：

```
.gitignore
```

```
1. venv/
2.
3. *.pyc
4. __pycache__/
5.
6. instance/
7.
8. .pytest_cache/
9. .coverage
10. htmlcov/
11.
12. dist/
13. build/
14. *.egg-info/
```

下面请阅读 [应用设置](#)。

应用设置

一个 Flask 应用是一个 `Flask` 类的实例。应用的所有东西（例如配置和 URL）都会和这个实例一起注册。

创建一个 Flask 应用最粗暴直接的方法是在代码的最开始创建一个全局 `Flask` 实例。前面的“Hello, World!”示例就是这样做的。有的情况下这样做是简单和有效的，但是当项目越来越大的时候就会有些力不从心了。

可以在一个函数内部创建 `Flask` 实例来代替创建全局实例。这个函数被称为 应用工厂。所有应用相关的配置、注册和其他设置都会在函数内部完成，然后返回这个应用。

应用工厂

写代码的时候到了！创建 `flaskr` 文件夹并且文件夹内添加 `init.py` 文件。`init.py` 有两个作用：一是包含应用工厂；二是告诉 Python `flaskr` 文件夹应当视作为一个包。

```
1. $ mkdir flaskr
```

`flaskr/init.py`

```
1. import os
2.
3. from flask import Flask
4.
5.
6. def create_app(test_config=None):
7.     # create and configure the app
8.     app = Flask(__name__, instance_relative_config=True)
9.     app.config.from_mapping(
10.         SECRET_KEY='dev',
11.         DATABASE=os.path.join(app.instance_path, 'flaskr.sqlite'),
12.     )
13.
14.     if test_config is None:
15.         # load the instance config, if it exists, when not testing
16.         app.config.from_pyfile('config.py', silent=True)
17.     else:
18.         # load the test config if passed in
19.         app.config.from_mapping(test_config)
20.
21.     # ensure the instance folder exists
22.     try:
23.         os.makedirs(app.instance_path)
24.     except OSError:
25.         pass
26.
27.     # a simple page that says hello
28.     @app.route('/hello')
29.     def hello():
```

```

30.         return 'Hello, World!'
31.
32.     return app

```

`create_app` 是一个应用工厂函数，后面的教程中会用到。这个看似简单的函数其实已经做了许多事情。

- `app = Flask(name, instance_relative_config=True)` 创建 `Flask` 实例。
 - `name` 是当前 Python 模块的名称。应用需要知道在哪里设置路径，使用 `name` 是一个方便的方法。
 - `instance_relative_config=True` 告诉应用配置文件是相对于 `instance folder` 的相对路径。实例文件夹在 `flaskr` 包的外面，用于存放本地数据（例如配置密钥和数据库），不应当提交到版本控制系统。
- `app.config.from_mapping()` 设置一个应用的缺省配置：
 - `SECRET_KEY` 是被 `Flask` 和扩展用于保证数据安全的。在开发过程中，为了方便可以设置为 `'dev'`，但是在发布的时候应当使用一个随机值来重载它。
 - `DATABASE` SQLite 数据库文件存放在路径。它位于 `Flask` 用于存放实例的 `app.instance_path` 之内。下一节会更详细地学习数据库的东西。
- `app.config.from_pyfile()` 使用 `config.py` 中的值来重载缺省配置，如果 `config.py` 存在的话。例如，当正式部署的时候，用于设置一个正式的 `SECRET_KEY`。
 - `test_config` 也会被传递给工厂，并且会替代实例配置。这样可以实现测试和开发的配置分离，相互独立。
- `os.makedirs()` 可以确保 `app.instance_path` 存在。`Flask` 不会自动创建实例文件夹，但是必须确保创建这个文件夹，因为 SQLite 数据库文件会被保存在里面。
- `@app.route()` 创建一个简单的路由，这样在继续教程下面的内容前你可以先看看应用如何运行的。它创建了 URL `/hello` 和一个函数之间的关联。这个函数会返回一个响应，即一个 `'Hello, World!'` 字符串。

运行应用

现在可以通过使用 `flask` 命令来运行应用。在终端中告诉 `Flask` 你的应用在哪里，然后在开发模式下运行应用。请记住，现在还是应当在最顶层的 `flask-tutorial` 目录下，不是在 `flaskr` 包里面。

开发模式下，当页面出错的时候会显示一个可以互动的调试器；当你修改代码保存的时候会重启服务器。在学习本教程的过程中，你可以一直让它保持运行，只需要刷新页面就可以了。

在 Linux and Mac 下：

```

1. $ export FLASK_APP=flaskr
2. $ export FLASK_ENV=development
3. $ flask run

```

在 Windows 下，使用 `set` 代替 `export` ：

```
1. > set FLASK_APP=flaskr
2. > set FLASK_ENV=development
3. > flask run
```

可以看到类似如下输出内容：

```
1. * Serving Flask app "flaskr"
2. * Environment: development
3. * Debug mode: on
4. * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
5. * Restarting with stat
6. * Debugger is active!
7. * Debugger PIN: 855-212-761
```

在浏览器中访问 <http://127.0.0.1:5000/hello> ，就可以看到 “Hello, World!” 信息。恭喜你， Flask 网络应用成功运行了！

下面请阅读 [定义和操作数据库](#) 。

定义和操作数据库

应用使用一个 `SQLite` 数据库来储存用户和博客内容。Python 内置了 `SQLite` 数据库支持，相应的模块为 `sqlite3`。

使用 `SQLite` 的便利性在于不需要单独配置一个数据库服务器，并且 Python 提供了内置支持。但是当并发请求同时要写入时，会比较慢一点，因为每个写操作是按顺序进行的。小应用没有问题，但是大应用可能就需要考虑换成别的数据库了。

本教程不会详细讨论 `SQL`。如果你不是很熟悉 `SQL`，请先阅读 `SQLite` 文档中的[相关内容](#)。

连接数据库

当使用 `SQLite` 数据库（包括其他多数数据库的 Python 库）时，第一件事就是创建一个数据库的连接。所有查询和操作都要通过该连接来执行，完事后该连接关闭。

在网络应用中连接往往与请求绑定。在处理请求的某个时刻，连接被创建。在发送响应之前连接被关闭。

`flaskr/db.py`

```
1. import sqlite3
2.
3. import click
4. from flask import current_app, g
5. from flask.cli import with_appcontext
6.
7.
8. def get_db():
9.     if 'db' not in g:
10.         g.db = sqlite3.connect(
11.             current_app.config['DATABASE'],
12.             detect_types=sqlite3.PARSE_DECLTYPES
13.         )
14.         g.db.row_factory = sqlite3.Row
15.
16.     return g.db
17.
18.
19. def close_db(e=None):
20.     db = g.pop('db', None)
21.
22.     if db is not None:
23.         db.close()
```

`g` 是一个特殊对象，独立于每一个请求。在处理请求过程中，它可以用于储存可能多个函数都会用到的数据。把连接储存于其中，可以多次使用，而不用在同一个请求中每次调用 `get_db` 时都创建一个新的连接。

`current_app` 是另一个特殊对象，该对象指向处理请求的 `Flask` 应用。这里使用了应用工厂，那么在其余的代码

中就不会出现应用对象。当应用创建后，在处理一个请求时，`get_db` 会被调用。这样就需要使用 `current_app`。

`sqlite3.connect()` 建立一个数据库连接，该连接指向配置中的 `DATABASE` 指定的文件。这个文件现在还没有建立，后面会在初始化数据库的时候建立该文件。

`sqlite3.Row` 告诉连接返回类似于字典的行，这样可以通过列名称来操作数据。

通过检查 `g.db` 来确定连接是否已经建立。如果连接已建立，那么就关闭连接。以后会在应用工厂中告诉应用 `close_db` 函数，这样每次请求后就会调用它。

创建表

在 SQLite 中，数据储存在 表 和 列 中。在储存和调取数据之前需要先创建它们。Flaskr 会把用户数据储存在 `user` 表中，把博客内容储存在 `post` 表中。下面创建一个文件储存用于创建空表的 SQL 命令：

`flaskr/schema.sql`

```
1. DROP TABLE IF EXISTS user;
2. DROP TABLE IF EXISTS post;
3.
4. CREATE TABLE user (
5.     id INTEGER PRIMARY KEY AUTOINCREMENT,
6.     username TEXT UNIQUE NOT NULL,
7.     password TEXT NOT NULL
8. );
9.
10. CREATE TABLE post (
11.     id INTEGER PRIMARY KEY AUTOINCREMENT,
12.     author_id INTEGER NOT NULL,
13.     created TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
14.     title TEXT NOT NULL,
15.     body TEXT NOT NULL,
16.     FOREIGN KEY (author_id) REFERENCES user (id)
17. );
```

在 `db.py` 文件中添加 Python 函数，用于运行这个 SQL 命令：

`flaskr/db.py`

```
1. def init_db():
2.     db = get_db()
3.
4.     with current_app.open_resource('schema.sql') as f:
5.         db.executescript(f.read().decode('utf8'))
6.
7.
8. @click.command('init-db')
9. @with_appcontext
10. def init_db_command():
11.     """Clear the existing data and create new tables."""
```

```
12.     init_db()
13.     click.echo('Initialized the database.')
```

`open_resource()` 打开一个文件，该文件名是相对于 `flaskr` 包的。这样就不需要考虑以后应用具体部署在哪个位置。`get_db` 返回一个数据库连接，用于执行文件中的命令。

`click.command()` 定义一个名为 `init-db` 命令行，它调用 `initdb` 函数，并为用户显示一个成功的消息。更多关于如何写命令行的内容请参阅 `ref:_cli`。

在应用中注册

`close_db` 和 `init_db_command` 函数需要在应用实例中注册，否则无法使用。然而，既然我们使用了工厂函数，那么在写函数的时候应用实例还无法使用。代替地，我们写一个函数，把应用作为参数，在函数中进行注册。

`flaskr/db.py`

```
1. def init_app(app):
2.     app.teardown_appcontext(close_db)
3.     app.cli.add_command(init_db_command)
```

`app.teardown_appcontext()` 告诉 Flask 在返回响应后进行清理的时候调用此函数。

`app.cli.add_command()` 添加一个新的可以与 `flask` 一起工作的命令。

在工厂中导入并调用这个函数。在工厂函数中把新的代码放到函数的尾部，返回应用代码的前面。

`flaskr/init.py`

```
1. def create_app():
2.     app = ...
3.     # existing code omitted
4.
5.     from . import db
6.     db.init_app(app)
7.
8.     return app
```

初始化数据库文件

现在 `init-db` 已经在应用中注册好了，可以与 `flask` 命令一起使用了。使用的方式与前一页的 `run` 命令类似。

Note

如果你还在运行着前一页的服务器，那么现在要么停止该服务器，要么在新的终端中运行这个命令。如果是新的终端请记住在进行项目文件夹并激活环境，参见 [激活虚拟环境](#)。同时还要像前一页所述设置 `FLASK_APP` 和

`FLASK_ENV`。

运行 `init-db` 命令：

```
1. $ flask init-db
2. Initialized the database.
```

现在会有一个 `flaskr.sqlite` 文件出现在项目所在文件夹的 `instance` 文件夹中。

下面请阅读 [蓝图和视图](#) 。

蓝图和视图

视图是一个应用对请求进行响应的函数。 Flask 通过模型把进来的请求 URL 匹配到对应的处理视图。视图返回数据， Flask 把数据变成出去的反应。 Flask 也可以反过来，根据视图的名称和参数生成 URL 。

创建蓝图

`Blueprint` 是一种组织一组相关视图及其他代码的方式。与把视图及其他代码直接注册到应用的方式不同，蓝图方式是把它们注册到蓝图，然后在工厂函数中把蓝图注册到应用。

Flaskr 有两个蓝图，一个用于认证功能，另一个用于博客帖子管理。每个蓝图的代码都在一个单独的模块中。使用博客首先需要认证，因此我们先写认证蓝图。

flaskr/auth.py

```
1. import functools
2.
3. from flask import (
4.     Blueprint, flash, g, redirect, render_template, request, session, url_for
5. )
6. from werkzeug.security import check_password_hash, generate_password_hash
7.
8. from flaskr.db import get_db
9.
10. bp = Blueprint('auth', __name__, url_prefix='/auth')
```

这里创建了一个名称为 `'auth'` 的 `Blueprint` 。和应用对象一样，蓝图需要知道是在哪里定义的，因此把 `name` 作为函数的第二个参数。 `url_prefix` 会添加到所有与该蓝图关联的 URL 前面。

使用 `app.register_blueprint()` 导入并注册蓝图。新的代码放在工厂函数的尾部返回应用之前。

flaskr/init.py

```
1. def create_app():
2.     app = ...
3.     # existing code omitted
4.
5.     from . import auth
6.     app.register_blueprint(auth.bp)
7.
8.     return app
```

认证蓝图将包括注册新用户、登录和注销视图。

第一个视图：注册

当用访问 `/auth/register` URL 时， `register` 视图会返回用于填写注册内容的表单的 HTML 。当用户提交表

单时，视图会验证表单内容，然后要么再次显示表单并显示一个出错信息，要么创建新用户并显示登录页面。

这里是视图代码，下一页会写生成 HTML 表单的模板。

flaskr/auth.py

```

1. @bp.route('/register', methods=('GET', 'POST'))
2. def register():
3.     if request.method == 'POST':
4.         username = request.form['username']
5.         password = request.form['password']
6.         db = get_db()
7.         error = None
8.
9.         if not username:
10.            error = 'Username is required.'
11.         elif not password:
12.            error = 'Password is required.'
13.         elif db.execute(
14.             'SELECT id FROM user WHERE username = ?', (username,)
15.         ).fetchone() is not None:
16.            error = 'User {} is already registered.'.format(username)
17.
18.         if error is None:
19.            db.execute(
20.                'INSERT INTO user (username, password) VALUES (?, ?)',
21.                (username, generate_password_hash(password))
22.            )
23.            db.commit()
24.            return redirect(url_for('auth.login'))
25.
26.         flash(error)
27.
28.     return render_template('auth/register.html')

```

这个 `register` 视图做了以下工作：

- `@bp.route` 关联了 URL `/register` 和 `register` 视图函数。当 Flask 收到一个指向 `/auth/register` 的请求时就会调用 `register` 视图并把其返回值作为响应。
- 如果用户提交了表单，那么 `request.method` 将会是 `'POST'`。这咱情况下会开始验证用户的输入内容。
- `request.form` 是一个特殊类型的 `dict`，其映射了提交表单的键和值。表单中，用户将会输入其 `username` 和 `password`。
- 验证 `username` 和 `password` 不为空。
- 通过查询数据库，检查是否有查询结果返回来验证 `username` 是否已被注册。`db.execute` 使用了带有 `?` 占位符的 SQL 查询语句。占位符可以代替后面的元组参数中相应的值。使用占位符的好处是会自动帮你转义输入值，以抵御 SQL 注入攻击。

`fetchone()` 根据查询返回一个记录行。如果查询没有结果，则返回 `None`。后面还用到了 `fetchall()`，它返

回包括所有结果的列表。

- 如果验证成功，那么在数据库中插入新用户数据。为了安全原因，不能把密码明文储存在数据库中。相代替的，使用 `generate_password_hash()` 生成安全的哈希值并储存到数据库中。查询修改了数据库是的数据后使用 `meth:db.commit()` 保存修改。
- 用户数据保存后将转到登录页面。`url_for()` 根据登录视图的名称生成相应的 URL。与写固定的 URL 相比，这样做的好处是如果以后需要修改该视图相应的 URL，那么不用修改所有涉及到URL 的代码。
`redirect()` 为生成的 URL 生成一个重定向响应。
- 如果验证失败，那么会向用户显示一个出错信息。`flash()` 用于储存在渲染模块时可以调用的信息。
- 当用户最初访问 `auth/register` 时，或者注册出错时，应用显示一个注册表单。`render_template()` 会渲染一个包含 HTML 的模板。你会在教程的下一节学习如何写这个模板。

登录

这个视图和上述 `register` 视图原理相同。

`flaskr/auth.py`

```
1. @bp.route('/login', methods=('GET', 'POST'))
2. def login():
3.     if request.method == 'POST':
4.         username = request.form['username']
5.         password = request.form['password']
6.         db = get_db()
7.         error = None
8.         user = db.execute(
9.             'SELECT * FROM user WHERE username = ?', (username,)
10.        ).fetchone()
11.
12.        if user is None:
13.            error = 'Incorrect username.'
14.        elif not check_password_hash(user['password'], password):
15.            error = 'Incorrect password.'
16.
17.        if error is None:
18.            session.clear()
19.            session['user_id'] = user['id']
20.            return redirect(url_for('index'))
21.
22.        flash(error)
23.
24.    return render_template('auth/login.html')
```

与 `register` 有以下不同之处：

- 首先需要查询用户并存放在变量中，以备后用。
- `check_password_hash()` 以相同的方式哈希提交的密码并安全的比较哈希值。如果匹配成功，那么密码就是正

确的。

- `session` 是一个 `dict`，它用于储存横跨请求的值。当验证成功后，用户的 `id` 被储存于一个新的会话中。会话数据被储存到一个向浏览器发送的 `cookie` 中，在后继请求中，浏览器会返回它。Flask 会安全对数据进行 签名 以防数据被篡改。

现在用户的 `id` 已被储存在 `session` 中，可以被后续的请求使用。请每个请求的开头，如果用户已登录，那么其用户信息应当被载入，以使其可用于其他视图。

flaskr/auth.py

```
1. @bp.before_app_request
2. def load_logged_in_user():
3.     user_id = session.get('user_id')
4.
5.     if user_id is None:
6.         g.user = None
7.     else:
8.         g.user = get_db().execute(
9.             'SELECT * FROM user WHERE id = ?', (user_id,)
10.        ).fetchone()
```

`bp.before_app_request()` 注册一个在视图函数之前运行的函数，不论其 URL 是什么。`load_logged_in_user` 检查用户 `id` 是否已经储存在 `session` 中，并从数据库中获取用户数据，然后储存在 `g.user` 中。`g.user` 的持续时间比请求要长。如果没有用户 `id`，或者 `id` 不存在，那么 `g.user` 将会是 `None`。

注销

注销的时候需要把用户 `id` 从 `session` 中移除。然后 `load_logged_in_user` 就不会在后继请求中载入用户了。

flaskr/auth.py

```
1. @bp.route('/logout')
2. def logout():
3.     session.clear()
4.     return redirect(url_for('index'))
```

在其他视图中验证

用户登录以后才能创建、编辑和删除博客帖子。在每个视图中可以使用装饰器 来完成这个工作。

flaskr/auth.py

```
1. def login_required(view):
2.     @functools.wraps(view)
3.     def wrapped_view(**kwargs):
4.         if g.user is None:
5.             return redirect(url_for('auth.login'))
6.
```



```
7.         return view(**kwargs)
8.
9.         return wrapped_view
```

装饰器返回一个新的视图，该视图包含了传递给装饰器的原视图。新的函数检查用户是否已载入。如果已载入，那么就继续正常执行原视图，否则就重定向到登录页面。我们会在博客视图中使用这个装饰器。

端点和 URL

`url_for()` 函数根据视图名称和发生 URL 。视图相关联的名称亦称为端点，缺省情况下，端点名称与视图函数名称相同。

例如，前文被加入应用工厂的 `hello()` 视图端点为 `'hello'`，可以使用 `url_for('hello')` 来连接。如果视图有参数，后文会看到，那么可使用 `url_for('hello', who='World')` 连接。

当使用蓝图的时候，蓝图的名称会添加到函数名称的前面。上面的 `login` 函数的端点为 `'auth.login'`，因为它已被加入 `'auth'` 蓝图中。

下面请阅读 [模板](#)。

模板

应用已经写好验证视图，但是如果现在运行服务器的话，无论访问哪个 URL，都会看到一个 `TemplateNotFound` 错误。这是因为视图调用了 `render_template()`，但是模板还没有写。模板文件会储存在 `flaskr` 包内的 `templates` 文件夹内。

模板是包含静态数据和动态数据占位符的文件。模板使用指定的数据生成最终的文档。Flask 使用 `Jinja` 模板库来渲染模板。

在教程的应用中会使用模板来渲染显示在用户浏览器中的 `HTML`。在 Flask 中，Jinja 被配置为自动转义 HTML 模板中的任何数据。即渲染用户的输入是安全的。任何用户输入的可能出现歧意的字符，如 `<` 和 `>`，会被转义，替换为安全 的值。这些值在浏览器中看起来一样，但是没有副作用。

Jinja 看上去并且运行地很像 Python。Jinja 语句与模板中的静态数据通过特定的分界符分隔。任何位于 `{{` 和 `}}` 这间的东西是一个会输出到最终文档的静态式。`{%` 和 `%}` 之间的东西表示流程控制语句，如 `if` 和 `for`。与 Python 不同，代码块使用分界符分隔，而不是使用缩进分隔。因为代码块内的静态文本可以会改变缩进。

基础布局

应用中的每一个页面主体不同，但是基本布局是相同的。每个模板会扩展 同一个基础模板并重载相应的小节，而不是重写整个 HTML 结构。

`flaskr/templates/base.html`

```
1. <!doctype html>
2. <title>{% block title %}{% endblock %} - Flaskr</title>
3. <link rel="stylesheet" href="{% url_for('static', filename='style.css') %}">
4. <nav>
5.   <h1>Flaskr</h1>
6.   <ul>
7.     {% if g.user %}
8.       <li><span>{{ g.user['username'] }}</span>
9.       <li><a href="{% url_for('auth.logout') %}">Log Out</a>
10.    {% else %}
11.      <li><a href="{% url_for('auth.register') %}">Register</a>
12.      <li><a href="{% url_for('auth.login') %}">Log In</a>
13.    {% endif %}
14.  </ul>
15. </nav>
16. <section class="content">
17.   <header>
18.     {% block header %}{% endblock %}
19.   </header>
20.   {% for message in get_flashed_messages() %}
21.     <div class="flash">{{ message }}</div>
22.   {% endfor %}
23.   {% block content %}{% endblock %}
```

```
24. </section>
```

`g` 在模板中自动可用。根据 `g.user` 是否被设置（在 `load_logged_in_user` 中进行），要么显示用户名和注销连接，要么显示注册和登录连接。`url_for()` 也是自动可用的，可用于生成视图的 URL，而不用手动来指定。

在标题下面，正文内容前面，模板会循环显示 `get_flashed_messages()` 返回的每个消息。在视图中使用 `flash()` 来处理出错信息，在模板中就可以这样显示出来。

模板中定义三个块，这些块会被其他模板重载。

- `{% block title %}` 会改变显示在浏览器标签和窗口中的标题。
- `{% block header %}` 类似于 `title`，但是会改变页面的标题。
- `{% block content %}` 是每个页面的具体内容，如登录表单或者博客帖子。

其他模板直接放在 `templates` 文件夹内。为了更好地管理文件，属于某个蓝图的模板会被放在与蓝图同名的文件夹内。

注册

```
flaskr/templates/auth/register.html
```

```
1. {% extends 'base.html' %}
2.
3. {% block header %}
4.     <h1>{% block title %}Register{% endblock %}</h1>
5. {% endblock %}
6.
7. {% block content %}
8.     <form method="post">
9.         <label for="username">Username</label>
10.        <input name="username" id="username" required>
11.        <label for="password">Password</label>
12.        <input type="password" name="password" id="password" required>
13.        <input type="submit" value="Register">
14.    </form>
15. {% endblock %}
```

`{% extends 'base.html' %}` 告诉 Jinja 这个模板基于基础模板，并且需要替换相应的块。所有替换的内容必须位于 `{% block %}` 标签之内。

一个实用的模式是把 `{% block title %}` 放在 `{% block header %}` 内部。这里不但可以设置 `title` 块，还可以把其值作为 `header` 块的内容，一举两得。

`input` 标记使用了 `required` 属性。这是告诉浏览器这些字段是必填的。如果用户使用不支持这个属性的旧版浏览器或者不是浏览器的东西创建的请求，那么你还是在视图中验证输入数据。总是在服务端中完全验证数据，即使客户端已经做了一些验证，这一点非常重要。

登录

本模板除了标题和提交按钮外与注册模板相同。

`flaskr/templates/auth/login.html`

```
1. {% extends 'base.html' %}
2.
3. {% block header %}
4.     <h1>{% block title %}Log In{% endblock %}</h1>
5. {% endblock %}
6.
7. {% block content %}
8.     <form method="post">
9.         <label for="username">Username</label>
10.        <input name="username" id="username" required>
11.        <label for="password">Password</label>
12.        <input type="password" name="password" id="password" required>
13.        <input type="submit" value="Log In">
14.    </form>
15. {% endblock %}
```

注册一个用户

现在验证模板已写好，你可以注册一个用户了。请确定服务器还在运行（如果没有请使用 `flask run` ），然后访问 <http://127.0.0.1:5000/auth/register> 。

在不填写表单的情况，尝试点击 “Register” 按钮，浏览器会显示出错信息。尝试在 `register.html` 中删除 `required` 属性后再次点击 “Register” 按钮。页面会重载并显示来自于视图中的 `flash()` 的出错信息，而不是浏览器显示出错信息。

填写用户名和密码后会重定向到登录页面。尝试输入错误的用户名，或者输入正常的用户名和错误的密码。如果登录成功，那么会看到一个出错信息，因为还没有写登录后要转向的 `index` 视图。

下面请阅读 [静态文件](#) 。

静态文件

验证视图和模板已经可用了，但是看上去很朴素。可以使用一些 [CSS](#) 给 HTML 添加点样式。样式不会改变，所以应当使用 静态文件 ，而不是模板。

Flask 自动添加一个 `static` 视图，视图使用相对于 `flaskr/static` 的相对路径。`base.html` 模板已经使用了一个 `style.css` 文件连接：

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

除了 CSS ，其他类型的静态文件可以是 JavaScript 函数文件或者 logo 图片。它们都放置于 `flaskr/static` 文件夹中，并使用 `url_for('static', filename='...')` 引用。

本教程不专注于如何写 CSS ，所以你只要复制以下内容到 `flaskr/static/style.css` 文件：

`flaskr/static/style.css`

```
1. html { font-family: sans-serif; background: #eee; padding: 1rem; }
2. body { max-width: 960px; margin: 0 auto; background: white; }
3. h1 { font-family: serif; color: #377ba8; margin: 1rem 0; }
4. a { color: #377ba8; }
5. hr { border: none; border-top: 1px solid lightgray; }
6. nav { background: lightgray; display: flex; align-items: center; padding: 0 0.5rem; }
7. nav h1 { flex: auto; margin: 0; }
8. nav h1 a { text-decoration: none; padding: 0.25rem 0.5rem; }
9. nav ul { display: flex; list-style: none; margin: 0; padding: 0; }
10. nav ul li a, nav ul li span, header .action { display: block; padding: 0.5rem; }
11. .content { padding: 0 1rem 1rem; }
12. .content > header { border-bottom: 1px solid lightgray; display: flex; align-items: flex-end; }
13. .content > header h1 { flex: auto; margin: 1rem 0 0.25rem 0; }
14. .flash { margin: 1em 0; padding: 1em; background: #cae6f6; border: 1px solid #377ba8; }
15. .post > header { display: flex; align-items: flex-end; font-size: 0.85em; }
16. .post > header > div:first-of-type { flex: auto; }
17. .post > header h1 { font-size: 1.5em; margin-bottom: 0; }
18. .post .about { color: slategray; font-style: italic; }
19. .post .body { white-space: pre-line; }
20. .content:last-child { margin-bottom: 0; }
21. .content form { margin: 1em 0; display: flex; flex-direction: column; }
22. .content label { font-weight: bold; margin-bottom: 0.5em; }
23. .content input, .content textarea { margin-bottom: 1em; }
24. .content textarea { min-height: 12em; resize: vertical; }
25. input.danger { color: #cc2f2e; }
26. input[type=submit] { align-self: start; min-width: 10em; }
```

你可以在[示例代码](#) 找到一个排版不紧凑的 `style.css` 。

访问 <http://127.0.0.1:5000/auth/login> ，页面如下所示。



The image shows a web interface for a Flask application. At the top, there's a header with the word 'Flaskr' in a large, bold, blue font. To its right are two links: 'Register' and 'Log In', both in blue and underlined. Below the header, there's a section titled 'Log In' in a large, bold, blue font. Underneath this title, there are two labels: 'Username' and 'Password', both in bold black font. Each label is followed by a text input field. At the bottom of the form, there is a button labeled 'Log In' in a light gray box. The entire form is enclosed in a light gray border.

关于 CSS 的更多内容参见 [Mozilla 的文档](#) 。改
动静态文件后需要刷新页面。如果刷新没有作用，请清除浏览器的缓存。

下面请阅读 [博客蓝图](#) 。

博客蓝图

博客蓝图与验证蓝图所使用的技术一样。博客页面应当列出所有的帖子，允许已登录用户创建帖子，并允许帖子作者修改和删除帖子。

当你完成每个视图时，请保持开发服务器运行。当你保存修改后，请尝试在浏览器中访问 URL ，并进行测试。

蓝图

定义蓝图并注册到应用工厂。

flaskr/blog.py

```
1. from flask import (
2.     Blueprint, flash, g, redirect, render_template, request, url_for
3. )
4. from werkzeug.exceptions import abort
5.
6. from flaskr.auth import login_required
7. from flaskr.db import get_db
8.
9. bp = Blueprint('blog', __name__)
```

使用 `app.register_blueprint()` 在工厂中导入和注册蓝图。将新代码放在工厂函数的尾部，返回应用之前。

flaskr/init.py

```
1. def create_app():
2.     app = ...
3.     # existing code omitted
4.
5.     from . import blog
6.     app.register_blueprint(blog.bp)
7.     app.add_url_rule('/', endpoint='index')
8.
9.     return app
```

与验证蓝图不同，博客蓝图没有 `url_prefix` 。因此 `index` 视图会用于 `/` ， `create` 会用于 `/create` ，以此类推。博客是 Flaskr 的主要功能，因此把博客索引作为主索引是合理的。

但是，下文的 `index` 视图的端点会被定义为 `blog.index` 。一些验证视图会指定向普通的 `index` 端点。我们使用 `app.add_url_rule()` 关联端点名称 `'index'` 和 `/` URL ，这样 `url_for('index')` 或 `url_for('blog.index')` 都会有效，会生成同样的 `/` URL 。

在其他应用中，可能会在工厂中给博客蓝图一个 `url_prefix` 并定义一个独立的 `index` 视图，类似前文中的 `hello` 视图。在这种情况下 `index` 和 `blog.index` 的端点和 URL 会有所不同。

索引

索引会显示所有帖子，最新的会排在最前面。为了在结果中包含 `user` 表中的作者信息，使用了一个 `JOIN`。

`flaskr/blog.py`

```
1. @bp.route('/')
2. def index():
3.     db = get_db()
4.     posts = db.execute(
5.         'SELECT p.id, title, body, created, author_id, username'
6.         ' FROM post p JOIN user u ON p.author_id = u.id'
7.         ' ORDER BY created DESC'
8.     ).fetchall()
9.     return render_template('blog/index.html', posts=posts)
```

`flaskr/templates/blog/index.html`

```
1. {% extends 'base.html' %}
2.
3. {% block header %}
4.     <h1>{% block title %}Posts{% endblock %}</h1>
5.     {% if g.user %}
6.         <a class="action" href="{{ url_for('blog.create') }}">New</a>
7.     {% endif %}
8. {% endblock %}
9.
10. {% block content %}
11.     {% for post in posts %}
12.         <article class="post">
13.             <header>
14.                 <div>
15.                     <h1>{{ post['title'] }}</h1>
16.                     <div class="about">by {{ post['username'] }} on {{ post['created'].strftime('%Y-%m-%d') }}</div>
17.                 </div>
18.                 {% if g.user['id'] == post['author_id'] %}
19.                     <a class="action" href="{{ url_for('blog.update', id=post['id']) }}">Edit</a>
20.                 {% endif %}
21.             </header>
22.             <p class="body">{{ post['body'] }}</p>
23.         </article>
24.         {% if not loop.last %}
25.             <hr>
26.         {% endif %}
27.     {% endfor %}
28. {% endblock %}
```

当用户登录后，`header` 块添加了一个指向 `create` 视图的连接。当用户是博客作者时，可以看到一个“ Edit ”连接，指向 `update` 视图。`loop.last` 是一个 Jinja for 循环 内部可用的特殊变量，它用于在每个博客帖子后面显示一条线来分隔帖子，最后一个帖子除外。

创建

`create` 视图与 `register` 视图原理相同。要么显示表单，要么发送内容已通过验证且内容已加入数据库，或者显示一个出错信息。

先前写的 `login_required` 装饰器用在了博客视图中，这样用户必须登录以后才能访问这些视图，否则会被重定向到登录页面。

`flaskr/blog.py`

```

1. @bp.route('/create', methods=('GET', 'POST'))
2. @login_required
3. def create():
4.     if request.method == 'POST':
5.         title = request.form['title']
6.         body = request.form['body']
7.         error = None
8.
9.         if not title:
10.            error = 'Title is required.'
11.
12.        if error is not None:
13.            flash(error)
14.        else:
15.            db = get_db()
16.            db.execute(
17.                'INSERT INTO post (title, body, author_id)'
18.                ' VALUES (?, ?, ?)',
19.                (title, body, g.user['id'])
20.            )
21.            db.commit()
22.            return redirect(url_for('blog.index'))
23.
24.    return render_template('blog/create.html')
```

`flaskr/templates/blog/create.html`

```

1. {% extends 'base.html' %}
2.
3. {% block header %}
4.     <h1>{% block title %}New Post{% endblock %}</h1>
5. {% endblock %}
6.
7. {% block content %}
8.     <form method="post">
9.         <label for="title">Title</label>
10.        <input name="title" id="title" value="{{ request.form['title'] }}" required>
11.        <label for="body">Body</label>
12.        <textarea name="body" id="body">{{ request.form['body'] }}</textarea>
13.        <input type="submit" value="Save">
14.    </form>
```

```
15. {% endblock %}
```

更新

`update` 和 `delete` 视图都需要通过 `id` 来获取一个 `post`，并且检查作者与登录用户是否一致。为避免重复代码，可以写一个函数来获取 `post`，并在每个视图中调用它。

```
flaskr/blog.py
```

```
1. def get_post(id, check_author=True):
2.     post = get_db().execute(
3.         'SELECT p.id, title, body, created, author_id, username'
4.         ' FROM post p JOIN user u ON p.author_id = u.id'
5.         ' WHERE p.id = ?',
6.         (id,)
7.     ).fetchone()
8.
9.     if post is None:
10.         abort(404, "Post id {0} doesn't exist.".format(id))
11.
12.     if check_author and post['author_id'] != g.user['id']:
13.         abort(403)
14.
15.     return post
```

`abort()` 会引发一个特殊的异常，返回一个 HTTP 状态码。它有一个可选参数，用于显示出错信息，若不使用该参数则返回缺省出错信息。`404` 表示“未找到”，`403` 代表“禁止访问”。（`401` 表示“未授权”，但是我们重定向到登录页面来代替返回这个状态码）

`check_author` 参数的作用是函数可以用于在不检查作者的情况下获取一个 `post`。这主要用于显示一个独立的帖子页面的情况，因为这时用户是谁没有关系，用户不会修改帖子。

```
flaskr/blog.py
```

```
1. @bp.route('/<int:id>/update', methods=('GET', 'POST'))
2. @login_required
3. def update(id):
4.     post = get_post(id)
5.
6.     if request.method == 'POST':
7.         title = request.form['title']
8.         body = request.form['body']
9.         error = None
10.
11.         if not title:
12.             error = 'Title is required.'
13.
14.         if error is not None:
15.             flash(error)
16.         else:
```

```

17.         db = get_db()
18.         db.execute(
19.             'UPDATE post SET title = ?, body = ?'
20.             ' WHERE id = ?',
21.             (title, body, id)
22.         )
23.         db.commit()
24.         return redirect(url_for('blog.index'))
25.
26.     return render_template('blog/update.html', post=post)

```

和所有以前的视图不同，`update` 函数有一个 `id` 参数。该参数对应路由中的 `<int:id>`。一个真正的 URL 类似 `/1/update`。Flask 会捕捉到 URL 中的 `1`，确保其为一个 `int`，并将其作为 `id` 参数传递给视图。如果没有指定 `int:` 而是仅仅写了 `<id>`，那么将会传递一个字符串。要生成一个指向更新页面的 URL，需要传递 `id` 参数给 `url_for()`：`url_for('blog.update', id=post['id'])`。前文的 `index.html` 文件中同样如此。

`create` 和 `update` 视图看上去是相似的。主要的不同之处在于 `update` 视图使用了一个 `post` 对象和一个 `UPDATE` 查询代替了一个 `INSERT` 查询。作为一个明智的重构者，可以使用一个视图和一个模板来同时完成这两项工作。但是作为一个初学者，把它们分别处理要清晰一些。

`flaskr/templates/blog/update.html`

```

1. {% extends 'base.html' %}
2.
3. {% block header %}
4.     <h1>{% block title %}Edit "{{ post['title'] }}" {% endblock %}</h1>
5. {% endblock %}
6.
7. {% block content %}
8.     <form method="post">
9.         <label for="title">Title</label>
10.        <input name="title" id="title"
11.            value="{{ request.form['title'] or post['title'] }}" required>
12.        <label for="body">Body</label>
13.        <textarea name="body" id="body">{{ request.form['body'] or post['body'] }}</textarea>
14.        <input type="submit" value="Save">
15.    </form>
16.    <hr>
17.    <form action="{{ url_for('blog.delete', id=post['id']) }}" method="post">
18.        <input class="danger" type="submit" value="Delete" onclick="return confirm('Are you sure?');">
19.    </form>
20. {% endblock %}

```

这个模板有两个表单。第一个提交已编辑过的数据给当前页面（`/<id>/update`）。另一个表单只包含一个按钮。它指定一个 `action` 属性，指向删除视图。这个按钮使用了一些 JavaScript 用以在提交前显示一个确认对话框。

参数 `{{ request.form['title'] or post['title'] }}` 用于选择在表单显示什么数据。当表单还未提交时，显示原 `post` 数据。但是，如果提交了非法数据，然后需要显示这些非法数据以便于用户修改时，就显示 `request.form` 中的数据。`request` 是又一个自动在模板中可用的变量。

删除

删除视图没有自己的模板。删除按钮已包含于 `update.html` 之中，该按钮指向 `/<id>/delete` URL。既然没有模板，该视图只处理 `POST` 方法并重定向到 `index` 视图。

`flaskr/blog.py`

```
1. @bp.route('/<int:id>/delete', methods=('POST',))
2. @login_required
3. def delete(id):
4.     get_post(id)
5.     db = get_db()
6.     db.execute('DELETE FROM post WHERE id = ?', (id,))
7.     db.commit()
8.     return redirect(url_for('blog.index'))
```

恭喜，应用写完了！花点时间在浏览器中试试这个应用吧。然而，构建一个完整的应用还有一些工作要做。

下面请阅读 [项目可安装化](#)。

项目可安装化

项目可安装化是指创建一个项目 发行 文件，以使用项目可以安装到其他环境，就像在你的项目中安装 Flask 一样。这样可以使你的项目如同其他库一样进行部署，可以使用标准的 Python 工具来管理项目。

可安装化还可以带来如下好处，这些好处在教程中可以不太明显或者初学者可能没注意到：

- 现在，Python 和 Flask 能够理解如何 `flaskr` 包，是因为你是在项目文件夹中运行的。可安装化后，可以从任何地方导入项目并运行。
- 可以和其他包一样管理项目的依赖，即使用 `pip install yourproject.whl` 来安装项目并安装相关依赖。
- 测试工具可以分离测试环境和开发环境。

Note

这些内容会在随后的教程中说明，但是在以后的项目中应当以此为项目的起点。

描述项目

`setup.py` 文件描述项目及其从属的文件。

`setup.py`

```
1. from setuptools import find_packages, setup
2.
3. setup(
4.     name='flaskr',
5.     version='1.0.0',
6.     packages=find_packages(),
7.     include_package_data=True,
8.     zip_safe=False,
9.     install_requires=[
10.         'flask',
11.     ],
12. )
```

`packages` 告诉 Python 包所包括的文件夹（及其所包含的 Python 文件）。`find_packages()` 自动找到这些文件夹，这样就不用手动写出来。为了包含其他文件夹，如静态文件和模板文件所在的文件夹，需要设置 `include_package_data` 。 Python 还需要一个名为 `MANIFEST.in` 文件来说明这些文件有哪些。

`MANIFEST.in`

```
1. include flaskr/schema.sql
2. graft flaskr/static
3. graft flaskr/templates
4. global-exclude *.pyc
```

这告诉 Python 复制所有 `static` 和 `templates` 文件夹中的文件，`schema.sql` 文件，但是排除所有字节文件。

更多内容和参数参见 [官方打包指南](#)。

安装项目

使用 `pip` 在虚拟环境中安装项目。

```
1. $ pip install -e .
```

这个命令告诉 `pip` 在当前文件夹中寻找 `setup.py` 并在 编辑 或 开发模式下安装。编辑模式是指当改变本地代码后，只需要重新项目。比如改变了项目依赖之类的元数据的情况下。

可以通过 `pip list` 来查看项目的安装情况。

```
1. $ pip list
2.
3. Package           Version  Location
4. -----
5. click              6.7
6. Flask              1.0
7. flaskr             1.0.0    /home/user/Projects/flask-tutorial
8. itsdangerous       0.24
9. Jinja2             2.10
10. MarkupSafe         1.0
11. pip                9.0.3
12. setuptools        39.0.1
13. Werkzeug          0.14.1
14. wheel              0.30.0
```

至此，没有改变项目运行的方式，`FLASK_APP` 还是被设置为 `flaskr`，还是使用 `flask run` 运行应用。不同的是可以在任何地方运行应用，而不仅仅是在 `flask-tutorial` 目录下。

下面请阅读 [测试覆盖](#)。

测试覆盖

为应用写单元测试可以检查代码是否按预期执行。Flask 提供了测试客户端，可以模拟向应用发送请求并返回响应数据。

应当尽可能多地进行测试。函数中的代码只有在函数被调用的情况下才会运行。分支中的代码，如 `if` 块中的代码，只有在符合条件的情况下才会运行。测试应当覆盖每个函数和每个分支。

越接近 100% 的测试覆盖，越能够保证修改代码后不会出现意外。但是 100% 测试覆盖不能保证应用没有错误。通常，测试不会覆盖用户如何在浏览器中与应用进行交互。尽管如此，在开发过程中，测试覆盖仍然是非常重要的。

Note

这部分内容在教程中是放在后面介绍的，但是在以后的项目中，应当在开发的时候进行测试。

我们使用 `pytest` 和 `coverage` 来进行测试和衡量代码。先安装它们：

```
1. $ pip install pytest coverage
```

配置和固件

测试代码位于 `tests` 文件夹中，该文件夹位于 `flaskr` 包的 旁边 ，而不是里面。`tests/conftest.py` 文件包含名为 `fixtures`（固件）的配置函数。每个测试都会用到这个函数。测试位于 Python 模块中，以 `test` 开头，并且模块中的每个测试函数也以 `test` 开头。

每个测试会创建一个新的临时数据库文件，并产生一些用于测试的数据。写一个SQL 文件来插入数据。

```
tests/data.sql
```

```
1. INSERT INTO user (username, password)
2. VALUES
3.     ('test', 'pbkdf2:sha256:50000$TCI4GzcX$0de171a4f4dac32e3364c7ddc7c14f3e2fa61f2d17574483f7ffbb431b4acb2f'),
4.     ('other', 'pbkdf2:sha256:50000$kJKsz6N$d2d4784f1b030a9761f5ccaeeca413f27f2ecb76d6168407af962ddce849f79');
5.
6. INSERT INTO post (title, body, author_id, created)
7. VALUES
8.     ('test title', 'test' || x'0a' || 'body', 1, '2018-01-01 00:00:00');
```

`app` 固件会调用工厂并为测试传递 `test_config` 来配置应用和数据库，而不使用本地的开发配置。

```
tests/conftest.py
```

```
1. import os
2. import tempfile
3.
4. import pytest
5. from flaskr import create_app
6. from flaskr.db import get_db, init_db
```

```

7.
8. with open(os.path.join(os.path.dirname(__file__), 'data.sql'), 'rb') as f:
9.     _data_sql = f.read().decode('utf8')
10.
11.
12. @pytest.fixture
13. def app():
14.     db_fd, db_path = tempfile.mkstemp()
15.
16.     app = create_app({
17.         'TESTING': True,
18.         'DATABASE': db_path,
19.     })
20.
21.     with app.app_context():
22.         init_db()
23.         get_db().executescript(_data_sql)
24.
25.     yield app
26.
27.     os.close(db_fd)
28.     os.unlink(db_path)
29.
30.
31. @pytest.fixture
32. def client(app):
33.     return app.test_client()
34.
35.
36. @pytest.fixture
37. def runner(app):
38.     return app.test_cli_runner()

```

`tempfile.mkstemp()` 创建并打开一个临时文件，返回该文件对象和路径。`DATABASE` 路径被重载，这样它会指向临时路径，而不是实例文件夹。设置好路径之后，数据库表被创建，然后插入数据。测试结束后，临时文件会被关闭并删除。

`TESTING` 告诉 Flask 应用处在测试模式下。Flask 会改变一些内部行为以方便测试。其他的扩展也可以使用这个标志方便测试。

`client` 固件调用 `app.test_client()` 由 `app` 固件创建的应用对象。测试会使用客户端来向应用发送请求，而不用启动服务器。

`runner` 固件类似于 `client`。`app.test_cli_runner()` 创建一个运行器，可以调用应用注册的 Click 命令。

Pytest 通过匹配固件函数名称和测试函数的参数名称来使用固件。例如下面要写 `test_hello` 函数有一个 `client` 参数。Pytest 会匹配 `client` 固件函数，调用该函数，把返回值传递给测试函数。

工厂

工厂本身没有什么好测试的，其大部分代码会被每个测试用到。因此如果工厂代码有问题，那么在进行其他测试时会被发现。

唯一可以改变的行为是传递测试配置。如果没传递配置，那么会有一些缺省配置可用，否则配置会被重载。

tests/test_factory.py

```
1. from flaskr import create_app
2.
3.
4. def test_config():
5.     assert not create_app().testing
6.     assert create_app({'TESTING': True}).testing
7.
8.
9. def test_hello(client):
10.     response = client.get('/hello')
11.     assert response.data == b'Hello, World!'
```

在本教程开头的部分添加了一个 `hello` 路由作为示例。它返回“Hello, World!”，因此测试响应数据是否匹配。

数据库

在一个应用环境中，每次调用 `get_db` 都应当返回相同的连接。退出环境后，连接应当已关闭。

tests/test_db.py

```
1. import sqlite3
2.
3. import pytest
4. from flaskr.db import get_db
5.
6.
7. def test_get_close_db(app):
8.     with app.app_context():
9.         db = get_db()
10.         assert db is get_db()
11.
12.     with pytest.raises(sqlite3.ProgrammingError) as e:
13.         db.execute('SELECT 1')
14.
15.     assert 'closed' in str(e.value)
```

`init-db` 命令应当调用 `init_db` 函数并输出一个信息。

tests/test_db.py

```
1. def test_init_db_command(runner, monkeypatch):
2.     class Recorder(object):
3.         called = False
```

```

4.
5.     def fake_init_db():
6.         Recorder.called = True
7.
8.     monkeypatch.setattr('flaskr.db.init_db', fake_init_db)
9.     result = runner.invoke(args=['init-db'])
10.    assert 'Initialized' in result.output
11.    assert Recorder.called

```

这个测试使用 Pytest 的 `monkeypatch` 固件来替换 `init_db` 函数。前文写的 `runner` 固件用于通过名称调用 `init-db` 命令。

验证

对于大多数视图，用户需要登录。在测试中最方便的方法是使用客户端制作一个 `POST` 请求发送给 `login` 视图。与其每次都写一遍，不如写一个类，用类的方法来做这件事，并使用一个固件把它传递给每个测试的客户端。

tests/conftest.py

```

1. class AuthActions(object):
2.     def __init__(self, client):
3.         self._client = client
4.
5.     def login(self, username='test', password='test'):
6.         return self._client.post(
7.             '/auth/login',
8.             data={'username': username, 'password': password}
9.         )
10.
11.    def logout(self):
12.        return self._client.get('/auth/logout')
13.
14.
15. @pytest.fixture
16. def auth(client):
17.     return AuthActions(client)

```

通过 `auth` 固件，可以在调试中调用 `auth.login()` 登录为 `test` 用户。这个用户的数据已经在 `app` 固件中写入了数据。

`register` 视图应当在 `GET` 请求时渲染成功。在 `POST` 请求中，表单数据合法时，该视图应当重定向到登录 URL，并且用户的数据已在数据库中保存好。数据非法时，应当显示出错信息。

tests/test_auth.py

```

1. import pytest
2. from flask import g, session
3. from flaskr.db import get_db
4.
5.

```

```

6. def test_register(client, app):
7.     assert client.get('/auth/register').status_code == 200
8.     response = client.post(
9.         '/auth/register', data={'username': 'a', 'password': 'a'}
10.    )
11.    assert 'http://localhost/auth/login' == response.headers['Location']
12.
13.    with app.app_context():
14.        assert get_db().execute(
15.            "select * from user where username = 'a'",
16.        ).fetchone() is not None
17.
18.
19. @pytest.mark.parametrize(('username', 'password', 'message'), (
20.     ('', '', b'Username is required.'),
21.     ('a', '', b'Password is required.'),
22.     ('test', 'test', b'already registered'),
23. ))
24. def test_register_validate_input(client, username, password, message):
25.     response = client.post(
26.         '/auth/register',
27.         data={'username': username, 'password': password}
28.     )
29.     assert message in response.data

```

`client.get()` 制作一个 `GET` 请求并由 Flask 返回 `Response` 对象。类似的 `client.post()` 制作一个 `POST` 请求，转换 `data` 字典为表单数据。

为了测试页面是否渲染成功，制作一个简单的请求，并检查是否返回一个 `200 OK` `status_code`。如果渲染失败，Flask 会返回一个 `500 Internal Server Error` 代码。

当注册视图重定向到登录视图时，`headers` 会有一个包含登录URL 的 `Location` 头部。

`data` 以字节方式包含响应的身体。如果想要检测渲染页面中的某个值，请 `data` 中检测。字节值只能与字节值作比较，如果想比较 Unicode文本，请使用 `get_data(as_text=True)`

`pytest.mark.parametrize` 告诉 Pytest 以不同的参数运行同一个测试。这里用于测试不同的非法输入和出错信息，避免重复写三次相同的代码。

`login` 视图的测试与 `register` 的非常相似。后者是测试数据库中的数据，前者是测试登录之后 `session` 应当包含 `user_id`。

`tests/test_auth.py`

```

1. def test_login(client, auth):
2.     assert client.get('/auth/login').status_code == 200
3.     response = auth.login()
4.     assert response.headers['Location'] == 'http://localhost/'
5.
6.     with client:
7.         client.get('/')
8.         assert session['user_id'] == 1

```

```

9.         assert g.user['username'] == 'test'
10.
11.
12. @pytest.mark.parametrize(('username', 'password', 'message'), (
13.     ('a', 'test', b'Incorrect username.'),
14.     ('test', 'a', b'Incorrect password.'),
15. ))
16. def test_login_validate_input(auth, username, password, message):
17.     response = auth.login(username, password)
18.     assert message in response.data

```

在 `with` 块中使用 `client`，可以在响应返回之后操作环境变量，比如 `session`。通常，在请求之外操作 `session` 会引发一个异常。

`logout` 测试与 `login` 相反。注销之后，`session` 应当不包含 `user_id`。

`tests/test_auth.py`

```

1. def test_logout(client, auth):
2.     auth.login()
3.
4.     with client:
5.         auth.logout()
6.         assert 'user_id' not in session

```

博客

所有博客视图使用之前所写的 `auth` 固件。调用 `auth.login()`，并且客户端的后继请求会登录为 `test` 用户。

`index` 索引视图应当显示已添加的测试帖子数据。作为作者登录之后，应当有编辑博客的连接。

当测试 `index` 视图时，还可以测试更多验证行为。当没有登录时，每个页面显示登录或注册连接。当登录之后，应当有一个注销连接。

`tests/test_blog.py`

```

1. import pytest
2. from flaskr.db import get_db
3.
4.
5. def test_index(client, auth):
6.     response = client.get('/')
7.     assert b"Log In" in response.data
8.     assert b"Register" in response.data
9.
10.    auth.login()
11.    response = client.get('/')
12.    assert b'Log Out' in response.data
13.    assert b'test title' in response.data
14.    assert b'by test on 2018-01-01' in response.data

```

```

15.     assert b'test\nbody' in response.data
16.     assert b'href="/1/update"' in response.data

```

用户必须登录后才能访问 `create` 、 `update` 和 `delete` 视图。帖子作者才能访问 `update` 和 `delete` 。否则返回一个 `403 Forbidden` 状态码。如果要访问 `post` 的 `id` 不存在,那么 `update` 和 `delete` 应当返回 `404 Not Found` 。

`tests/test_blog.py`

```

1. @pytest.mark.parametrize('path', (
2.     '/create',
3.     '/1/update',
4.     '/1/delete',
5. ))
6. def test_login_required(client, path):
7.     response = client.post(path)
8.     assert response.headers['Location'] == 'http://localhost/auth/login'
9.
10.
11. def test_author_required(app, client, auth):
12.     # change the post author to another user
13.     with app.app_context():
14.         db = get_db()
15.         db.execute('UPDATE post SET author_id = 2 WHERE id = 1')
16.         db.commit()
17.
18.     auth.login()
19.     # current user can't modify other user's post
20.     assert client.post('/1/update').status_code == 403
21.     assert client.post('/1/delete').status_code == 403
22.     # current user doesn't see edit link
23.     assert b'href="/1/update"' not in client.get('/').data
24.
25.
26. @pytest.mark.parametrize('path', (
27.     '/2/update',
28.     '/2/delete',
29. ))
30. def test_exists_required(client, auth, path):
31.     auth.login()
32.     assert client.post(path).status_code == 404

```

对于 `GET` 请求, `create` 和 `update` 视图应当渲染和返回一个 `200 OK` 状态码。当 `POST` 请求发送了合法数据后, `create` 应当在数据库中插入新的帖子数据, `update` 应当修改数据库中现存的数据。当数据非法时,两者都应当显示一个出错信息。

`tests/test_blog.py`

```

1. def test_create(client, auth, app):
2.     auth.login()
3.     assert client.get('/create').status_code == 200
4.     client.post('/create', data={'title': 'created', 'body': ''})

```

```

5.
6.     with app.app_context():
7.         db = get_db()
8.         count = db.execute('SELECT COUNT(id) FROM post').fetchone()[0]
9.         assert count == 2
10.
11.
12. def test_update(client, auth, app):
13.     auth.login()
14.     assert client.get('/1/update').status_code == 200
15.     client.post('/1/update', data={'title': 'updated', 'body': ''})
16.
17.     with app.app_context():
18.         db = get_db()
19.         post = db.execute('SELECT * FROM post WHERE id = 1').fetchone()
20.         assert post['title'] == 'updated'
21.
22.
23. @pytest.mark.parametrize('path', (
24.     '/create',
25.     '/1/update',
26. ))
27. def test_create_update_validate(client, auth, path):
28.     auth.login()
29.     response = client.post(path, data={'title': '', 'body': ''})
30.     assert b'Title is required.' in response.data

```

`delete` 视图应当重定向到索引 URL，并且帖子应当从数据库中删除。

`tests/test_blog.py`

```

1. def test_delete(client, auth, app):
2.     auth.login()
3.     response = client.post('/1/delete')
4.     assert response.headers['Location'] == 'http://localhost/'
5.
6.     with app.app_context():
7.         db = get_db()
8.         post = db.execute('SELECT * FROM post WHERE id = 1').fetchone()
9.         assert post is None

```

运行测试

额外的配置可以添加到项目的 `setup.cfg` 文件。这些配置不是必需的，但是可以使用测试更简洁明了。

`setup.cfg`

```

1. [tool:pytest]
2. testpaths = tests
3.
4. [coverage:run]

```

```

5. branch = True
6. source =
7.     flaskr

```

使用 `pytest` 来运行测试。该命令会找到并且运行所有测试。

```

1. $ pytest
2.
3. ===== test session starts =====
4. platform linux -- Python 3.6.4, pytest-3.5.0, py-1.5.3, pluggy-0.6.0
5. rootdir: /home/user/Projects/flask-tutorial, inifile: setup.cfg
6. collected 23 items
7.
8. tests/test_auth.py ..... [ 34%]
9. tests/test_blog.py ..... [ 86%]
10. tests/test_db.py .. [ 95%]
11. tests/test_factory.py .. [100%]
12.
13. ===== 24 passed in 0.64 seconds =====

```

如果有测试失败，`pytest` 会显示引发的错误。可以使用 `pytest -v` 得到每个测试的列表，而不是一串点。

可以使用 `coverage` 命令代替直接使用 `pytest` 来运行测试，这样可以衡量测试覆盖率。

```

1. $ coverage run -m pytest

```

在终端中，可以看到一个简单的覆盖率报告：

```

1. $ coverage report
2.
3. Name                               Stmts   Miss Branch BrPart  Cover
4. -----
5. flaskr/__init__.py                 21      0      2      0   100%
6. flaskr/auth.py                     54      0     22      0   100%
7. flaskr/blog.py                     54      0     16      0   100%
8. flaskr/db.py                       24      0      4      0   100%
9. -----
10. TOTAL                             153      0     44      0   100%

```

还可以生成 HTML 报告，可以看到每个文件中测试覆盖了哪些行：

```

1. $ coverage html

```

这个命令在 `htmlcov` 文件夹中生成测试报告，然后在浏览器中打开 `htmlcov/index.html` 查看。

下面请阅读 [部署产品](#)。

部署产品

本文假设你要把应用部署到一个服务器上。本文只是给出如何创建发行文件并进行安装的概览，但是不会具体讨论使用哪种服务器或者软件。你可以在用于开发的电脑中设置一个新的虚拟环境，以便于尝试下面的内容。但是建议不要用于部署一个真正的公开应用。以多种不同方式部署应用的列表参见 [部署方式](#)。

构建和安装

当需要把应用部署到其他地方时，需要构建一个发行文件。当前 Python 的标准发行文件是 *wheel* 格式的，扩展名为 `.whl`。先确保已经安装好 *wheel* 库：

```
1. $ pip install wheel
```

用 Python 运行 `setup.py` 会得到一个命令行工具，以使用构建相关命令。`bdist_wheel` 命令会构建一个 *wheel* 发行文件。

```
1. $ python setup.py bdist_wheel
```

构建的文件为 `dist/flaskr-1.0.0-py3-none-any.whl`。文件名由项目名称、版本号和一些关于项目安装要求的标记组成。

复制这个文件到另一台机器，[创建一个新的虚拟环境](#)，然后用 `pip` 安装这个文件。

```
1. $ pip install flaskr-1.0.0-py3-none-any.whl
```

Pip 会安装项目和相关依赖。

既然这是一个不同的机器，那么需要再次运行 `init-db` 命令，在实例文件夹中创建数据库。

```
1. $ export FLASK_APP=flaskr
2. $ flask init-db
```

当 Flask 探测到它已被安装（不在编辑模式下），它会与前文不同，使用 `venv/var/flaskr-instance` 作为实例文件夹。

配置密钥

在教程开始的时候给了 `SECRET_KEY` 一个缺省值。在产品中我们应当设置一些随机内容。否则网络攻击者就可以使用公开的 `'dev'` 键来修改会话cookie，或者其他任何使用密钥的东西。

可以使用下面的命令输出一个随机密钥：

```
1. $ python -c 'import os; print(os.urandom(16))'
2.
```



```
3. b'_5#y2L"F4Q8z\n\xec]/'
```

在实例文件夹创建一个 `config.py` 文件。工厂会读取这个文件，如果该文件存在的话。提制生成的值到该文件中。

```
venv/var/flaskr-instance/config.py
```

```
1. SECRET_KEY = b'_5#y2L"F4Q8z\n\xec]/'
```

其他必须的配置也可以写入该文件中。 Flaskr 只需要 `SECRET_KEY` 即可。

运行产品服务器

当运行公开服务器而不是进行开发的时候，应当不使用内建的开发服务器（ `flask run` ）。开发服务器由 Werkzeug 提供，目的是为了更方便开发，但是不够高效、稳定和安全。

替代地，应当选用一个产品级的 WSGI 服务器。例如，使用 `Waitress` 。首先在虚拟环境中安装它：

```
1. $ pip install waitress
```

需要把应用告知 Waitree ，但是方式与 `flask run` 那样使用 `FLASK_APP` 不同。需要告知 Waitree 导入并调用应用工厂来得到一个应用对象。

```
1. $ waitress-serve --call 'flaskr:create_app'
2.
3. Serving on http://0.0.0.0:8080
```

以多种不同方式部署应用的列表参见 [部署方式](#) 。使用 Waitress 只是一个示例，选择它是因为它同时支持 Windows 和 Linux 。还有其他许多 WSGI服务器和部署选项可供选择。

下面请阅读 [继续开发！](#) 。

继续开发！

通过教程你已经学到了许多 Flask 和 Python 的概念。复习一下教程，并比较每一步的代码有何变化。比较你的项目与 [示例项目](#)，可能会发现有较大的区别，蹒跚学步，很自然。

Flask 远不止教程所涉及的这些内容，然而你已经可以开始网络应用开发的。请阅读[快速上手](#)，对 Flask 的功能有个大致了解，然后深入文档进行学习。Flask 在幕后使用了 [Jinja](#)、[Click](#)、[Werkzeug](#) 和 [ItsDangerous](#)，它们也有各自在文档。Flask 还有许多功能强大的 [扩展](#)，比如数据库扩展或者表单验证扩展等等，你一定会感兴趣的。

如果要继续开发 Flaskr 项目，建议尝试以下内容：

- 点击帖子标题，显示一个帖子详细页面。
- 喜欢或者不喜欢一个帖子。
- 评论。
- 标记。点击标记显示所有带有该标记的帖子。
- 一个可以过滤标题的搜索框。
- 分布显示索引。每页显示只显示五个帖子。
- 帖子可以上传图片。
- 帖子支持用 Markdown 撰写。
- 一个新帖子的 RSS 源。

祝你开心并写出令人惊叹的应用！

模板

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

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

Jinja 设置

在 Flask 中， Jinja2 默认配置如下：

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

标准环境

缺省情况下，以下全局变量可以在 Jinja2 模板中使用：

- `config`
- 当前配置对象 (`flask.config`)

Changelog

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

New in version 0.6.

- `request`
- 当前请求对象 (`flask.request`)。在没有活动请求环境情况下渲染模板时，这个变量不可用。
- `session`
- 当前会话对象 (`flask.session`)。在没有活动请求环境情况下渲染模板时，这个变量不可用。
- `g`
- 请求绑定的全局变量 (`flask.g`)。在没有活动请求环境情况下渲染模板时，这个变量不可用。
- `url_for` ()
- `flask.url_for()` 函数。
- `get_flashed_messages` ()

- `flask.get_flashed_messages()` 函数。

Jinja 环境行为

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

那么意义何在？假设你需要导入一个宏，这个宏需要访问请求对象，那么你有两个选择：

- 显式地把请求或都该请求有用的属性作为参数传递给宏。
- 导入“with context”宏。

导入方式如下：

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

标准过滤器

在 Flask 中的模板中添加了以下 Jinja2 本身没有的过滤器：

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

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

在一个单引号 HTML 属性中使用 `|tojson` 的输出也是安全的：

```
1. <button onclick='doSomethingWith({{ user.username|tojson }})''>
2.     Click me
3. </button>
```

请注意，在 0.10 版本之前的 Flask 中，如果在 `<script>` 里面使用 `|tojson` 的输出，请确保使用 `|safe` 禁用转义。在 Flask 0.10 及更高版本中，这会自动发生。

控制自动转义

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

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

有三种方法可以控制自动转义：

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

在模板中关闭自动转义系统可以使用 `{% autoescape %}` 块：

```
1. {% autoescape false %}
2.     <p>autoescaping is disabled here
3.     <p>{{ will_not_be_escaped }}
4. {% endautoescape %}
```

在这样做的时候，要非常小心块中的变量的安全性。

注册过滤器

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

下面两个例子功能相同，都是倒序一个对象：

```
1. @app.template_filter('reverse')
2. def reverse_filter(s):
3.     return s[::-1]
4.
5. def reverse_filter(s):
6.     return s[::-1]
7. app.jinja_env.filters['reverse'] = reverse_filter
```

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

```
1. {% for x in mylist | reverse %}
2. {% endfor %}
```

环境处理器

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

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

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

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

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

上例中的环境处理器把 `format_price` 函数传递给了所有模板：

```
1. {{ format_price(0.33) }}
```

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

测试 Flask 应用

未经测试的小猫，肯定不是一只好猫。

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

Flask 提供的测试渠道是使用 Werkzeug 的 `Client` 类，并为你处理本地环境。你可以结合这个渠道使用你喜欢的测试工具。

本文使用 `pytest` 包作为测试的基础框架。你可以像这样使用 `pip` 来安装它：

```
1. $ pip install pytest
```

应用

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

测试骨架

首先我们在应用的根文件夹中添加一个测试文件夹。然后创建一个 Python 文件来储存测试内容（`testflaskr.py`）。名称类似 `test` `*.py` 的文件会被 `pytest` 自动发现。

接着，我们创建一个名为 `client()` 的 `pytest` 固件，用来配置调试应用并初始化一个新的数据库：

```
1. import os
2. import tempfile
3.
4. import pytest
5.
6. from flaskr import flaskr
7.
8.
9. @pytest.fixture
10. def client():
11.     db_fd, flaskr.app.config['DATABASE'] = tempfile.mkstemp()
12.     flaskr.app.config['TESTING'] = True
13.
14.     with flaskr.app.test_client() as client:
15.         with flaskr.app.app_context():
16.             flaskr.init_db()
17.         yield client
18.
19.     os.close(db_fd)
20.     os.unlink(flaskr.app.config['DATABASE'])
```

这个客户端固件会被每个独立的测试调用。它提供了一个简单的应用接口，用于向应用发送请求，还可以为我们追踪 cookie 。

在配置中，`TESTING` 配置标志是被激活的。这样在处理请求过程中，错误捕捉被关闭，以利于在测试过程得到更好的错误报告。

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

为了在测试后删除数据库，固件关闭并删除了文件。

如果现在进行测试，那么会输出以下内容：

```
1. $ pytest
2.
3. ===== test session starts =====
4. rootdir: ./flask/examples/flaskr, inifile: setup.cfg
5. collected 0 items
6.
7. ===== no tests ran in 0.07 seconds =====
```

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

第一个测试

现在开始测试应用的功能。当我们访问应用的根 URL（ / ）时应该显示“ No entries here so far ”。在

`test_flaskr.py` 文件中新增一个测试函数来测试这个功能：

```
1. def test_empty_db(client):
2.     """Start with a blank database."""
3.
4.     rv = client.get('/')
5.     assert b'No entries here so far' in rv.data
```

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

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

再次运行测试，会看到通过了一个测试：

```
1. $ pytest -v
2.
3. ===== test session starts =====
4. rootdir: ./flask/examples/flaskr, inifile: setup.cfg
```



```

5. collected 1 items
6.
7. tests/test_flaskr.py::test_empty_db PASSED
8.
9. ===== 1 passed in 0.10 seconds =====

```

登录和注销

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

在 `test_flaskr.py` 文件中添加以下两个函数：

```

1. def login(client, username, password):
2.     return client.post('/login', data=dict(
3.         username=username,
4.         password=password
5.     ), follow_redirects=True)
6.
7.
8. def logout(client):
9.     return client.get('/logout', follow_redirects=True)

```

现在可以方便地测试登录成功、登录失败和注销功能了。下面为新增的测试函数：

```

1. def test_login_logout(client):
2.     """Make sure login and logout works."""
3.
4.     rv = login(client, flaskr.app.config['USERNAME'], flaskr.app.config['PASSWORD'])
5.     assert b'You were logged in' in rv.data
6.
7.     rv = logout(client)
8.     assert b'You were logged out' in rv.data
9.
10.    rv = login(client, flaskr.app.config['USERNAME'] + 'x', flaskr.app.config['PASSWORD'])
11.    assert b'Invalid username' in rv.data
12.
13.    rv = login(client, flaskr.app.config['USERNAME'], flaskr.app.config['PASSWORD'] + 'x')
14.    assert b'Invalid password' in rv.data

```

测试添加消息

我们还要测试添加消息功能。添加如下测试函数：

```

1. def test_messages(client):
2.     """Test that messages work."""
3.

```

```

4.     login(client, flaskr.app.config['USERNAME'], flaskr.app.config['PASSWORD'])
5.     rv = client.post('/add', data=dict(
6.         title='<Hello>',
7.         text='<strong>HTML</strong> allowed here'
8.     ), follow_redirects=True)
9.     assert b'No entries here so far' not in rv.data
10.    assert b'&lt;Hello&gt;' in rv.data
11.    assert b'<strong>HTML</strong> allowed here' in rv.data

```

这里我们验证了 HTML 出现在文本中，但是不出现在标题中，符合我们的预期。

运行测试，应当显示通过了三个测试：

```

1. $ pytest -v
2.
3. ===== test session starts =====
4. rootdir: ./flask/examples/flaskr, inifile: setup.cfg
5. collected 3 items
6.
7. tests/test_flaskr.py::test_empty_db PASSED
8. tests/test_flaskr.py::test_login_logout PASSED
9. tests/test_flaskr.py::test_messages PASSED
10.
11. ===== 3 passed in 0.23 seconds =====

```

其他测试技巧

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

```

1. import flask
2.
3. app = flask.Flask(__name__)
4.
5. with app.test_request_context('/?name=Peter'):
6.     assert flask.request.path == '/'
7.     assert flask.request.args['name'] == 'Peter'

```

所有其他与环境绑定的对象也可以这样使用。

如果要使用不同的配置来测试应用，而且没有什么好的测试方法，那么可以考虑使用应用工厂（参见 [应用工厂](#)）。

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

```

1. app = flask.Flask(__name__)
2.
3. with app.test_request_context('/?name=Peter'):

```

```
4.     app.preprocess_request()
5.     ...
```

在这函数中可以打开数据库连接或者根据应用需要打开其他类似东西。

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

```
1. app = flask.Flask(__name__)
2.
3. with app.test_request_context('/?name=Peter'):
4.     resp = Response('...')
5.     resp = app.process_response(resp)
6.     ...
```

这个例子中的情况基本没有用处，因为在这种情况下可以直接开始使用测试客户端。

伪造资源和环境

Changelog

New in version 0.10.

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

```
1. def get_user():
2.     user = getattr(g, 'user', None)
3.     if user is None:
4.         user = fetch_current_user_from_database()
5.         g.user = user
6.     return user
```

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

```
1. from contextlib import contextmanager
2. from flask import appcontext_pushed, g
3.
4. @contextmanager
5. def user_set(app, user):
6.     def handler(sender, **kwargs):
7.         g.user = user
8.     with appcontext_pushed.connected_to(handler, app):
9.         yield
```

然后使用它：

```
1. from flask import json, jsonify
2.
3. @app.route('/users/me')
```

```

4. def users_me():
5.     return jsonify(username=g.user.username)
6.
7. with user_set(app, my_user):
8.     with app.test_client() as c:
9.         resp = c.get('/users/me')
10.        data = json.loads(resp.data)
11.        self.assertEqual(data['username'], my_user.username)

```

保持环境

Changelog

New in version 0.4.

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

```

1. app = flask.Flask(__name__)
2.
3. with app.test_client() as c:
4.     rv = c.get('/?tequila=42')
5.     assert request.args['tequila'] == '42'

```

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

访问和修改会话

Changelog

New in version 0.8.

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

```

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

```

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

```

1. with app.test_client() as c:
2.     with c.session_transaction() as sess:
3.         sess['a_key'] = 'a value'
4.
5.     # once this is reached the session was stored and ready to be used by the client

```

```
6.     c.get(...)
```

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

测试 JSON API

Changelog

New in version 1.0.

Flask 对 JSON 的支持非常好，并且是一个创建 JSON API 的流行选择。使用 JSON 生成请求和在响应中检查 JSON 数据非常方便：

```
1. from flask import request, jsonify
2.
3. @app.route('/api/auth')
4. def auth():
5.     json_data = request.get_json()
6.     email = json_data['email']
7.     password = json_data['password']
8.     return jsonify(token=generate_token(email, password))
9.
10. with app.test_client() as c:
11.     rv = c.post('/api/auth', json={
12.         'email': 'flask@example.com', 'password': 'secret'
13.     })
14.     json_data = rv.get_json()
15.     assert verify_token(email, json_data['token'])
```

在测试客户端方法中传递 `json` 参数，设置请求数据为 JSON 序列化对象，并设置内容类型为 `application/json`。可以使用 `get_json` 从请求或者响应中获取 JSON 数据。

测试 CLI 命令

Click 来自于 [测试工具](#)，可用于测试 CLI 命令。一个 `CliRunner` 独立运行命令并通过 `Result` 对象捕获输出。

Flask 提供 `test_cli_runner()` 来创建一个 `FlaskCliRunner`，以自动传递 Flask 应用给 CLI。用它的 `invoke()` 方法调用命令，与在命令行中调用一样：

```
1. import click
2.
3. @app.cli.command('hello')
4. @click.option('--name', default='world')
5. def hello_command(name):
6.     click.echo(f'Hello, {name}!')
7.
8. def test_hello():
9.     runner = app.test_cli_runner()
```

```

10.
11.     # invoke the command directly
12.     result = runner.invoke(hello_command, ['--name', 'Flask'])
13.     assert 'Hello, Flask' in result.output
14.
15.     # or by name
16.     result = runner.invoke(args=['hello'])
17.     assert 'World' in result.output

```

在上面的例子中，通过名称引用命令的好处是可以验证命令是否在应用中已正确注册过。

如果要在不运行命令的情况下测试运行参数解析，可以使用其 `make_context()` 方法。这样有助于测试复杂验证规则和自定义类型：

```

1. def upper(ctx, param, value):
2.     if value is not None:
3.         return value.upper()
4.
5. @app.cli.command('hello')
6. @click.option('--name', default='World', callback=upper)
7. def hello_command(name):
8.     click.echo(f'Hello, {name}!')
9.
10. def test_hello_params():
11.     context = hello_command.make_context('hello', ['--name', 'flask'])
12.     assert context.params['name'] == 'FLASK'

```

应用错误处理

Changelog

New in version 0.3.

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

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

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

但可做的还不只这些，下面介绍一些更好的出错处理方法。

错误日志工具

当足够多的用户触发了错误时，发送关于出错信息的邮件，即使仅包含严重错误的邮件也会是一场空难。更不用提从来不会去看的日志文件了。因此，推荐使用 [Sentry](#) 来处理应用错误。它可以在一个开源项目 [on GitHub](#) 中获得，也可以在 [hosted version](#) 中免费试用。Sentry 统计重复错误，捕获堆栈数据和本地变量用于排错，并在发生新的或者指定频度的错误时发送电子邮件。

要使用 Sentry 需要安装带有 `flask` 依赖的 `sentry-sdk` 客户端：

```
1. $ pip install sentry-sdk[flask]
```

把下面内容加入 Flask 应用：

```
1. import sentry_sdk
2. from sentry_sdk.integrations.flask import FlaskIntegration
3.
4. sentry_sdk.init('YOUR_DSN_HERE', integrations=[FlaskIntegration()])
```

`YOUR_DSN_HERE` 需要被替换为在 Sentry 安装时获得的 DSN 值。

安装好以后，内部服务出错信息会自动向 Sentry 报告，你会接收到出错通知。

后续阅读：

- Sentry 也支持从队列（ RQ 、 Celery ）中捕获错误。详见[Python SDK 文档](#)。
- [Sentry 入门](#)
- [Flask-相关文档](#)

错误处理

当错误发生时，你可能想要向用户显示自定义的出错页面。注册出错处理器或以做到这点。

一个出错处理器是一个返回响应的普通视图函数。但是不同之处在于它不是用于路由的，而是用于一个异常或者当尝试处理请求时抛出 HTTP 状态码。

注册

通过使用 `errorhandler()` 装饰函数来注册或者稍后使用 `register_error_handler()` 来注册。记得当返回响应的时候设置出错代码：

```
1. @app.errorhandler(werkzeug.exceptions.BadRequest)
2. def handle_bad_request(e):
3.     return 'bad request!', 400
4.
5. # or, without the decorator
6. app.register_error_handler(400, handle_bad_request)
```

当注册时， `werkzeug.exceptions.HTTPException` 的子类，如 `BadRequest` ，和它们的 HTTP 代码是可替换的。（ `BadRequest.code == 400` ）

因为 Werkzeug 无法识别非标准 HTTP 代码，因此它们不能被注册。替代地，使用适当的代码定义一个 `HTTPException` 子类，注册并抛出异常类：

```
1. class InsufficientStorage(werkzeug.exceptions.HTTPException):
2.     code = 507
3.     description = 'Not enough storage space.'
4.
5. app.register_error_handler(InsufficientStorage, handle_507)
6.
7. raise InsufficientStorage()
```

出错处理器可被用于任何异常类的注册，除了 `HTTPException` 子类或者 HTTP 状态码。出错处理器可被用于特定类的注册，也可用于一个父类的所有子类的注册。

处理

在处理请求时，当 Flask 捕捉到一个异常时，它首先根据代码检索。如果该代码没有注册处理器，它会根据类的继承来查找，确定最合适的注册处理器。如果找不到已注册的处理器，那么 `HTTPException` 子类会显示一个关于代码的通用消息。没有代码的异常会被转化为一个通用的 500 内部服务器错误。

例如，如果一个 `ConnectionRefusedError` 的实例被抛出，并且一个出错处理器注册到 `ConnectionError` 和 `ConnectionRefusedError`，那么会使用更合适的 `ConnectionRefusedError` 来处理异常实例，生成响应。

当一个蓝图在处理抛出异常的请求时，在蓝图中注册的出错处理器优先于在应用中全局注册的出错处理器。但是，蓝图无法处理 404 路由错误，因为 404 发生的路由级别还不能检测到蓝图。

通用异常处理器

可以为非常通用的基类注册异常处理器，例如 `HTTPException` 基类或者甚至 `Exception` 基类。但是，请注意，这样会捕捉到超出你预期的异常。

基于 `HTTPException` 的异常处理器对于把缺省的 HTML 出错页面转换为 JSON 非常有用，但是这个处理器会触发不由你直接产生的东西，如路由过程中产生的 404 和 405 错误。请仔细制作你的处理器，确保不会丢失关于 HTTP 错误的信息。

```
1. from flask import json
2. from werkzeug.exceptions import HTTPException
3.
4. @app.errorhandler(HTTPException)
5. def handle_exception(e):
6.     """Return JSON instead of HTML for HTTP errors."""
7.     # start with the correct headers and status code from the error
8.     response = e.get_response()
9.     # replace the body with JSON
10.    response.data = json.dumps({
11.        "code": e.code,
12.        "name": e.name,
13.        "description": e.description,
14.    })
15.    response.content_type = "application/json"
16.    return response
```

基于 `Exception` 的异常处理器有助于改变所有异常处理的表现形式，甚至包含未处理的异常。但是，与在 Python 使用 `except Exception:` 类似，这样会捕获 所有 未处理的异常，包括所有 HTTP 状态码。因此，在大多数情况下，设定只针对特定异常的处理器比较安全。因为 `HTTPException` 实例是一个合法的 WSGI 响应，你可以直接传递该实例。

```
1. from werkzeug.exceptions import HTTPException
2.
3. @app.errorhandler(Exception)
4. def handle_exception(e):
5.     # pass through HTTP errors
6.     if isinstance(e, HTTPException):
7.         return e
8.
9.     # now you're handling non-HTTP exceptions only
```

```
10.     return render_template("500_generic.html", e=e), 500
```

异常处理器仍然遵循异常烦类的继承层次。如果同时基于 `HTTPException` 和 `Exception` 注册了异常处理器，`Exception` 处理器不会处理 `HTTPException` 子类，因为 `HTTPException` 更有针对性。

未处理的异常

当一个异常发生时，如果没有对应的异常处理器，那么就会返回一个 500 内部服务错误。关于此行为的更多内容参见 `flask.Flask.handle_exception()` 。

如果针对 `InternalServerError` 注册了异常处理器，那么出现内部服务错误时就会调用这个处理器。自 Flask 1.1.0 开始，总是会传递一个 `InternalServerError` 实例给这个异常处理器，而不是以前的未处理异常。原始的异常可以通过 `e.original_error` 访问。在 Werkzeug 1.0.0 以前，这个属性只有未处理异常有。建议使用 `getattr` 访问这个属性，以保证兼容性。

```
1. @app.errorhandler(InternalServerError)
2. def handle_500(e):
3.     original = getattr(e, "original_exception", None)
4.
5.     if original is None:
6.         # direct 500 error, such as abort(500)
7.         return render_template("500.html"), 500
8.
9.     # wrapped unhandled error
10.    return render_template("500_unhandled.html", e=original), 500
```

日志

如何记录异常，比如向管理者发送邮件，参见 [日志](#) 。

排除应用错误

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

有疑问时，请手动运行

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

使用调试器

为了更深入的挖掘错误，追踪代码的执行，Flask 提供一个开箱即用的调试器（参见 [调试模式](#)）。如果你需要使

用其他 Python 调试器，请注意调试器之间的干扰问题。在使用你自己的调试器前要做一些参数调整：

- `debug` - 是否开启调试模式并捕捉异常
- `use_debugger` - 是否使用 Flask 内建的调试器
- `use_reloader` - 模块变化后是否重载并派生进程

`debug` 必须设置为 `True`（即必须捕获异常），另两个随便。

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

一个有用的配置模式如下（当然要根据你的应用调整缩进）：

```
1. FLASK:  
2.     DEBUG: True  
3.     DEBUG_WITH_APTANA: True
```

然后，在应用入口（`main.py`），修改如下：

```
1. if __name__ == "__main__":  
2.     # To allow aptana to receive errors, set use_debugger=False  
3.     app = create_app(config="config.yaml")  
4.  
5.     use_debugger = app.debug and not(app.config.get('DEBUG_WITH_APTANA'))  
6.     app.run(use_debugger=use_debugger, debug=app.debug,  
7.             use_reloader=use_debugger, host='0.0.0.0')
```

日志

Flask 使用标准 Python `logging` 。所有与 Flask 相关的消息都用 `app.logger` 来记录，其名称与 `app.name` 相同。这个日志记录器也可用于你自己的日志记录。

```
1. @app.route('/login', methods=['POST'])
2. def login():
3.     user = get_user(request.form['username'])
4.     if user.check_password(request.form['password']):
5.         login_user(user)
6.         app.logger.info('%s logged in successfully', user.username)
7.         return redirect(url_for('index'))
8.     else:
9.         app.logger.info('%s failed to log in', user.username)
10.        abort(401)
```

基本配置

当想要为项目配置日志时，应当在程序启动时尽早进行配置。如果完了，那么 `app.logger` 就会成为缺省记录器。如果有可能的话，应当在创建应用对象之前配置日志。

这个例子使用 `dictConfig()` 来创建一个类似于 Flask 缺省配置的日志记录配置：

```
1. from logging.config import dictConfig
2.
3. dictConfig({
4.     'version': 1,
5.     'formatters': {'default': {
6.         'format': '%(asctime)s %(levelname)s in %(module)s: %(message)s',
7.     }},
8.     'handlers': {'wsgi': {
9.         'class': 'logging.StreamHandler',
10.        'stream': 'ext://flask.logging.wsgi_errors_stream',
11.        'formatter': 'default'
12.    }},
13.    'root': {
14.        'level': 'INFO',
15.        'handlers': ['wsgi']
16.    }
17. })
18.
19. app = Flask(__name__)
```

缺省配置

如果没有自己配置日志， Flask 会自动添加一个 `StreamHandler` 到 `app.logger` 。在请求过程中，它会写到由

WSGI 服务器指定的，保存在 `environ['wsgi.errors']` 变量中的日志流（通常是 `sys.stderr`）中。在请求之外，则会记录到 `sys.stderr`。

移除缺省配置

如果在操作 `app.logger` 之后配置日志，并且需要移除缺省的日志记录器，可以导入并移除它：

```
1. from flask.logging import default_handler
2.
3. app.logger.removeHandler(default_handler)
```

把出错信息通过电子邮件发送给管理者

当产品运行在一个远程服务器上时，可能不会经常查看日志信息。WSGI 服务器可能会在一个文件中记录日志消息，而你只会当用户告诉你出错的时候才会查看日志文件。

为了主动发现并修复错误，可以配置一个 `logging.handlers.SMTPHandler`，用于在一般错误或者更高级别错误发生时发送一封电子邮件：

```
1. import logging
2. from logging.handlers import SMTPHandler
3.
4. mail_handler = SMTPHandler(
5.     mailhost='127.0.0.1',
6.     fromaddr='server-error@example.com',
7.     toaddrs=['admin@example.com'],
8.     subject='Application Error'
9. )
10. mail_handler.setLevel(logging.ERROR)
11. mail_handler.setFormatter(logging.Formatter(
12.     '%(asctime)s %(levelname)s in %(module)s: %(message)s'
13. ))
14.
15. if not app.debug:
16.     app.logger.addHandler(mail_handler)
```

这需要在同一台服务器上拥有一个 SMTP 服务器。关于配置日志的更多内容请参阅Python 文档。

注入请求信息

看到更多请求信息，如 IP 地址，有助调试某些错误。可以继承 `logging.Formatter` 来注入自己的内容，以显示在日志消息中。然后，可以修改 Flask 缺省的日志记录器、上文所述的电子邮件日志记录器或者其他日志记录器的格式器。：

```
1. from flask import has_request_context, request
2. from flask.logging import default_handler
3.
```

```

4. class RequestFormatter(logging.Formatter):
5.     def format(self, record):
6.         if has_request_context():
7.             record.url = request.url
8.             record.remote_addr = request.remote_addr
9.         else:
10.            record.url = None
11.            record.remote_addr = None
12.
13.         return super().format(record)
14.
15. formatter = RequestFormatter(
16.     '%(asctime)s] %(remote_addr)s requested %(url)s\n'
17.     '%(levelname)s in %(module)s: %(message)s'
18. )
19. default_handler.setFormatter(formatter)
20. mail_handler.setFormatter(formatter)

```

其他库

其他库可能也会产生大量日志，而你也正好需要查看这些日志。最简单的方法是向根记录器中添加记录器。：

```

1. from flask.logging import default_handler
2.
3. root = logging.getLogger()
4. root.addHandler(default_handler)
5. root.addHandler(mail_handler)

```

单独配置每个记录器更好还是只配置一个根记录器更好，取决于你的项目。：

```

1. for logger in (
2.     app.logger,
3.     logging.getLogger('sqlalchemy'),
4.     logging.getLogger('other_package'),
5. ):
6.     logger.addHandler(default_handler)
7.     logger.addHandler(mail_handler)

```

Werkzeug

Werkzeug 记录基本的请求/响应信息到 `'werkzeug'` 日志记录器。如果根记录器没有配置，那么 Werkzeug 会向记录器添加一个 `StreamHandler`。

Flask 扩展

根据情况不同，一个扩展可能会选择记录到 `app.logger` 或者其自己的日志记录器。具体请查阅扩展的文档。

配置管理

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

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

不管你使用何种方式载入配置，都可以使用 `Flask` 对象的 `config` 属性来操作配置的值。Flask 本身就使用这个对象来保存一些配置，扩展也可以使用这个对象保存配置。同时这也是你保存配置的地方。

配置入门

`config` 实质上是一个字典的子类，可以像字典一样操作：

```
1. app = Flask(__name__)
2. app.config['TESTING'] = True
```

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

```
1. app.testing = True
```

一次更新多个配置值可以使用 `dict.update()` 方法：

```
1. app.config.update(
2.     TESTING=True,
3.     SECRET_KEY=b'_5#y2L"F4Q8z\n\xec]/'
4. )
```

环境和调试特征

`ENV` 和 `DEBUG` 配置值是特殊的，因为它们如果在应用设置完成之后改变，那么可以会有不同的行为表现。为了重可靠的设置环境和调试，Flask 使用环境变量。

环境用于为 Flask、扩展和其他程序（如 Sentry）指明 Flask 运行的情境是什么。环境由 `FLASK_ENV` 环境变量控制，缺省值为 `production`。

把 `FLASK_ENV` 设置为 `development` 可以打开调试模式。在调试模式下，`flask run` 会缺省使用交互调试器和重载器。如果需要脱离环境，单独控制调试模式，请使用 `FLASK_DEBUG` 标示。

Changelog

Changed in version 1.0: Added `FLASK_ENV` to control the environment separately from debug mode. The development environment enables debug mode.

为把 Flask 转换到开发环境并开启调试模式，设置 `FLASK_ENV`：

```
1. $ export FLASK_ENV=development
2. $ flask run
```

（在 Windows 下，使用 `set` 代替 `export`。）

推荐使用如上文的方式设置环境变量。虽然可以在配置或者代码中设置环境变量无法及时地被 `flask` 命令读取，一个系统或者扩展就可能会使用自己已定义的环境变量。

内置配置变量

以下配置变量由 Flask 内部使用：

- `ENV`
- 应用运行于什么环境。Flask 和 扩展可以根据环境不同而行为不同，如打开或关闭调试模式。`env` 属性映射了这个配置键。本变量由 `FLASK_ENV` 环境变量设置。如果本变量是在代码中设置的话，可能出现意外。

在生产环境中不要使用 `development`。

缺省值： `'production'`

Changelog

New in version 1.0.

- `DEBUG`
- 是否开启调试模式。使用 `flask run` 启动开发服务器时，遇到未能处理的异常时会显示一个交互调试器，并且当代码变动后服务器会重启。`debug` 属性映射了这个配置键。当 `ENV` 是 `'development'` 时，本变量会启用，并且会被 `FLASK_DEBUG` 环境变量重载。如果本变量是在代码中设置的话，可能会出现意外。

在生产环境中不要开启调试模式。

缺省值：当 `ENV` 是 `'development'` 时，为 `True`；否则为 `False`。

- `TESTING`
- 开启测试模式。异常会被广播而不是被应用的错误处理器处理。扩展可能也会为了测试方便而改变它们的行为。你应当在自己的调试中开启本变量。

缺省值： `False`

- `PROPAGATE_EXCEPTIONS`
- 异常会重新引发而不是被应用的错误处理器处理。在没有设置本变量的情况下，当 `TESTING` 或 `DEBUG` 开启时，本变量隐式地为真。

缺省值： `None`

- `PRESERVE_CONTEXT_ON_EXCEPTION`
- 当异常发生时，不要弹出请求情境。在没有设置该变量的情况下，如果 `DEBUG` 为真，则本变量为真。这样允许调试器错误请求数据。本变量通常不需要直接设置。

缺省值: `None`

- `TRAP_HTTP_EXCEPTIONS`
- 如果没有处理 `HTTPException` 类型异常的处理器，重新引发该异常用于被交互调试器处理，而不是作为一个简单的错误响应来返回。

缺省值: `False`

- `TRAP_BAD_REQUEST_ERRORS`
- 尝试操作一个请求字典中不存在的键，如 `args` 和 `form`，会返回一个 400 Bad Request error 页面。开启本变量，可以把这种错误作为一个未处理的异常处理，这样就可以使用交互调试器了。本变量是一个特殊版本的 `TRAP_HTTP_EXCEPTIONS`。如果没有设置，本变量会在调试模式下开启。

缺省值: `None`

- `SECRET_KEY`
- 密钥用于会话 cookie 的安全签名，并可用于应用或者扩展的其他安全需求。本变量应当是一个字节型长随机字符串，虽然 unicode 也是可以接受的。例如，复制如下输出到你的配置中：

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

当发贴提问或者提交代码时，不要泄露密钥。

缺省值: `None`

- `SESSION_COOKIE_NAME`
- 会话 cookie 的名称。假如已存在同名 cookie，本变量可改变。

缺省值: `'session'`

- `SESSION_COOKIE_DOMAIN`
- 认可会话 cookie 的域的匹配规则。如果本变量没有设置，那么 cookie 会被 `SERVER_NAME` 的所有子域认可。如果本变量设置为 `False`，那么 cookie 域不会被设置。

缺省值: `None`

- `SESSION_COOKIE_PATH`
- 认可会话 cookie 的路径。如果没有设置本变量，那么路径为 `APPLICATION_ROOT`，如果 `APPLICATION_ROOT` 也没有设置，那么会是 `/`。

缺省值: `None`

- `SESSION_COOKIE_HTTPONLY`
- 为了安全，浏览器不会允许 JavaScript 操作标记为“HTTP only”的 cookie。

缺省值: `True`

- `SESSION_COOKIE_SECURE`
- 如果 cookie 标记为“secure”，那么浏览器只会使用基于 HTTPS 的请求发送 cookie。应用必须使用 HTTPS 服务来启用本变量。

缺省值: `False`

- `SESSION_COOKIE_SAMESITE`
- 限制来自外部站点的请求如何发送 cookie 。可以被设置为 `'Lax'` （推荐）或者 `'Strict'` 。参见 [Set-Cookie 选项](#)。

缺省值: `None`

Changelog

New in version 1.0.

- `PERMANENT_SESSION_LIFETIME`
- 如果 `session.permanent` 为真， cookie 的有效期为本变量设置的数字，单位为秒。本变量可能是一个 `datetime.timedelta` 或者一个 `int` 。

Flask 的缺省 cookie 机制会验证电子签章不基于这个变量的值。

缺省值: `timedelta(days=31)` （ `2678400` 秒）

- `SESSION_REFRESH_EACH_REQUEST`
- 当 `session.permanent` 为真时，控制是否每个响应都发送 cookie 。每次都发送 cookie （缺省情况）可以有效地防止会话过期，但是会使用更多的带宽。会持续会话不受影响。

缺省值: `True`

- `USE_X_SENDFILE`
- 当使用 Flask 提供文件服务时，设置 `X-Sendfile` 头部。有些网络服务器，如 Apache ，识别这种头部，以利于更有效地提供数据服务。本变量只有使用这种服务器时才有效。

缺省值: `False`

- `SEND_FILE_MAX_AGE_DEFAULT`
- 当提供文件服务时，设置缓存，控制最长存活期，以秒为单位。可以是一个 `datetime.timedelta` 或者一个 `int` 。在一个应用或者蓝图上使用 `get_send_file_max_age()` 可以基于单个文件重载本变量。

缺省值: `timedelta(hours=12)` （ `43200` 秒）

- `SERVER_NAME`
- 通知应用其所绑定的主机和端口。子域路由匹配需要本变量。

如果配置了本变量， `SESSION_COOKIE_DOMAIN` 没有配置，那么本变量会被用于会话 cookie 的域。现代网络浏览器不会允许为没有点的域设置 cookie 。为了使用一个本地域，可以在你的 `host` 文件中为应用路由添加任意名称。：

```
1. 127.0.0.1 localhost.dev
```

如果这样配置了， `url_for` 可以为应用生成一个单独的外部 URL ，而不是一个请求情境。

缺省值: `None`

- `APPLICATION_ROOT`
- 通知应用应用的根路径是什么。这个变量用于生成请求环境之外的 URL （请求内的会根据 `SCRIPT_NAME` 生成；参见 [应用调度](#)）。

如果 `SESSION_COOKIE_PATH` 没有配置，那么本变量会用于会话 cookie 路径。

缺省值： `'/'`

- `PREFERRED_URL_SCHEME`
- 当不在请求情境内时使用些预案生成外部 URL 。

缺省值： `'http'`

- `MAX_CONTENT_LENGTH`
- 在进来的请求数据中读取的最大字节数。如果本变量没有配置，并且请求没有指定 `CONTENT_LENGTH` ，那么为了安全原因，不会读任何数据。

缺省值： `None`

- `JSON_AS_ASCII`
- 把对象序列化为 ASCII-encoded JSON 。如果禁用，那么 JSON 会被返回为一个Unicode 字符串或者被 `jsonify` 编码为 `UTF-8` 格式。本变量应当保持启用，因为在模块内把 JSON 渲染到 JavaScript 时会安全一点。

缺省值： `True`

- `JSON_SORT_KEYS`
- 按字母排序 JSON 对象的键。这对于缓存是有用的，因为不管 Python 的哈希种子是什么都能够保证数据以相同的方式序列化。为了以缓存为代价的性能提高可以禁用它，虽然不推荐这样做。

缺省值： `True`

- `JSONIFY_PRETTYPRINT_REGULAR`
- `jsonify` 响应会输出新行、空格和缩进以便于阅读。在调试模式下总是启用的。

缺省值： `False`

- `JSONIFY_MIMETYPE`
- `jsonify` 响应的媒体类型。

缺省值： `'application/json'`

- `TEMPLATES_AUTO_RELOAD`
- 当模板改变时重载它们。如果没有配置，在调试模式下会启用。

缺省值： `None`

- `EXPLAIN_TEMPLATE_LOADING`
- 记录模板文件如何载入的调试信息。使用本变量有助于查找为什么模板没有载入或者载入了错误的模板的原因。

缺省值： `False`

- `MAX_COOKIE_SIZE`
- 当 cookie 头部大于本变量配置的字节数时发出警告。缺省值为 `4093`。更大的 cookie 会被浏览器悄悄地忽略。本变量设置为 `0` 时关闭警告。

Changelog

Changed in version 1.0: `LOGGER_NAME` 和 `LOGGER_HANDLER_POLICY` 被删除。关于配置的更多内容参见 [日志](#)。

添加 `ENV` 来映射 `FLASK_ENV` 环境变量。

添加 `SESSION_COOKIE_SAMESITE` 来控制会话 cookie 的 `SameSite` 选项。

添加 `MAX_COOKIE_SIZE` 来控制来自于 Werkzeug 警告。

New in version 0.11: `SESSION_REFRESH_EACH_REQUEST` , `TEMPLATES_AUTO_RELOAD` , `LOGGER_HANDLER_POLICY` , `EXPLAIN_TEMPLATE_LOADING`

New in version 0.10: `JSON_AS_ASCII` , `JSON_SORT_KEYS` , `JSONIFY_PRETTYPRINT_REGULAR`

New in version 0.9: `PREFERRED_URL_SCHEME`

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

New in version 0.7: `PROPAGATE_EXCEPTIONS` , `PRESERVE_CONTEXT_ON_EXCEPTION`

New in version 0.6: `MAX_CONTENT_LENGTH`

New in version 0.5: `SERVER_NAME`

New in version 0.4: `LOGGER_NAME`

使用配置文件

如果把配置放在一个单独的文件中会更有用。理想情况下配置文件应当放在应用包之外。这样可以使用不同的工具进行打包与分发（[使用 Setuptools 部署](#)），而后修改配置文件也没有影响。

因此，常见用法如下：

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

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

```
1. $ export YOURAPPLICATION_SETTINGS=/path/to/settings.cfg
2. $ python run-app.py
```

```
3.  * Running on http://127.0.0.1:5000/
4.  * Restarting with reloader...
```

在 Windows 系统中使用内置的 `set` 来代替：

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

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

一个配置文件的例子：

```
1. # Example configuration
2. DEBUG = False
3. SECRET_KEY = b'_5#y2L"F4Q8z\xec]/'
```

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

使用环境变量来配置

除了使用环境变量指向配置文件之外，你可能会发现直接从环境中控制配置值很有用（或必要）。

启动服务器之前，可以在 Linux 或 OS X 上使用 shell 中的 `export` 命令设置环境变量：

```
1. $ export SECRET_KEY='5f352379324c22463451387a0aec5d2f'
2. $ export MAIL_ENABLED=false
3. $ python run-app.py
4.  * Running on http://127.0.0.1:5000/
```

在 Windows 系统中使用内置的 `set` 来代替：

```
1. > set SECRET_KEY='5f352379324c22463451387a0aec5d2f'
```

尽管这种方法很简单易用，但重要的是要记住环境变量是字符串，它们不会自动反序列化为 Python 类型。

以下是使用环境变量的配置文件示例：

```
1. import os
2.
3. _mail_enabled = os.environ.get("MAIL_ENABLED", default="true")
4. MAIL_ENABLED = _mail_enabled.lower() in {"1", "t", "true"}
5.
6. SECRET_KEY = os.environ.get("SECRET_KEY")
7.
8. if not SECRET_KEY:
9.     raise ValueError("No SECRET_KEY set for Flask application")
```

请注意，除了空字符串之外的任何值都将被解释为 Python 中的布尔值 `True`，如果环境显式设置值为 `False`，则需要注意。

确保尽早加载配置，以便扩展能够在启动时访问配置。除了从文件加载，配置对象还有其它方法可以加载。完整的参考参见 `Config` 类文档。

配置的最佳实践

前面提到的方法的缺点是它使测试更加困难。一般来说，这个问题没有一个 100% 完美的解决方案，但你可以牢记几件事以改善这种体验：

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

开发/生产

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

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

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

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

一个有趣的方案是使用类和类的继承来配置：

```
1. class Config(object):
2.     DEBUG = False
3.     TESTING = False
4.     DATABASE_URI = 'sqlite:///memory:'
5.
6. class ProductionConfig(Config):
7.     DATABASE_URI = 'mysql://user@localhost/foo'
8.
9. class DevelopmentConfig(Config):
10.     DEBUG = True
11.
12. class TestingConfig(Config):
```

```
13. TESTING = True
```

如果要使用这样的方案，那么必须使用 `from_object()`：

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

注意 `from_object()` 不会实例化类对象。如果要操作已经实例化的类，比如读取一个属性，那么在调用 `from_object()` 之前应当先实例化这个类：

```
1. from configmodule import ProductionConfig
2. app.config.from_object(ProductionConfig())
3.
4. # Alternatively, import via string:
5. from werkzeug.utils import import_string
6. cfg = import_string('configmodule.ProductionConfig')()
7. app.config.from_object(cfg)
```

在你的配置类中，实例化配置对象时允许使用 `@property`

```
1. class Config(object):
2.     """Base config, uses staging database server."""
3.     DEBUG = False
4.     TESTING = False
5.     DB_SERVER = '192.168.1.56'
6.
7.     @property
8.     def DATABASE_URI(self):          # Note: all caps
9.         return 'mysql://user@{}/foo'.format(self.DB_SERVER)
10.
11. class ProductionConfig(Config):
12.     """Uses production database server."""
13.     DB_SERVER = '192.168.19.32'
14.
15. class DevelopmentConfig(Config):
16.     DB_SERVER = 'localhost'
17.     DEBUG = True
18.
19. class TestingConfig(Config):
20.     DB_SERVER = 'localhost'
21.     DEBUG = True
22.     DATABASE_URI = 'sqlite:///memory:'
```

配置的方法多种多样，由你定度。以下是一些好的建议：

- 在版本控制中保存一个缺省配置。要么在应用中使用这些缺省配置，要么先导入缺省配置然后用你自己的配置文件来重载缺省配置。
- 使用一个环境变量来切换不同的配置。这样就可以在 Python 解释器外进行切换，而根本不用改动代码，使开发和部署更方便，更快捷。如果你经常在不同的项目间切换，那么你甚至可以创建代码来激活 `virtualenv` 并导出开发配置。

- 在生产应用中使用 `fabric` 之类的工具，向服务器分别传送代码和配置。更多细节参见 [使用 Fabric 部署方案](#)。

实例文件夹

Changelog

New in version 0.8.

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

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

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

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

请记住，这里提供的路径 必须 是绝对路径。

如果 `instance_path` 参数没有提供，那么会使用以下缺省位置：

- 未安装的模块：

```
1. /myapp.py
2. /instance
```

- 未安装的包：

```
1. /myapp
2. /__init__.py
3. /instance
```

- 已安装的模块或包：

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

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

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


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

以下是一个完整的配置 Flask 的例子，从一个模块预先载入配置，然后从实例文件夹中的一个配置文件（如果这个文件存在的话）载入要重载的配置：

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

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

`Flask.open_instance_resource()` 。

举例说明：

```
1. filename = os.path.join(app.instance_path, 'application.cfg')
2. with open(filename) as f:
3.     config = f.read()
4.
5. # or via open_instance_resource:
6. with app.open_instance_resource('application.cfg') as f:
7.     config = f.read()
```

信号

Changelog

New in version 0.6.

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

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

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

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

订阅信号

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

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

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

```
1. from flask import template_rendered
2. from contextlib import contextmanager
3.
4. @contextmanager
5. def captured_templates(app):
6.     recorded = []
7.     def record(sender, template, context, **extra):
8.         recorded.append((template, context))
9.     template_rendered.connect(record, app)
10.    try:
11.        yield recorded
12.    finally:
13.        template_rendered.disconnect(record, app)
```

上例可以在测试客户端中轻松使用：

```
1. with captured_templates(app) as templates:
```

```

2.     rv = app.test_client().get('/')
3.     assert rv.status_code == 200
4.     assert len(templates) == 1
5.     template, context = templates[0]
6.     assert template.name == 'index.html'
7.     assert len(context['items']) == 10

```

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

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

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

```

1. from flask import template_rendered
2.
3. def captured_templates(app, recorded, **extra):
4.     def record(sender, template, context):
5.         recorded.append((template, context))
6.     return template_rendered.connected_to(record, app)

```

上例可以这样使用：

```

1. templates = []
2. with captured_templates(app, templates, **extra):
3.     ...
4.     template, context = templates[0]

```

Blinker API 变化

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

创建信号

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

```

1. from blinker import Namespace
2. my_signals = Namespace()

```

现在可以像这样创建新的信号：

```

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

```

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

扩展开发者注意

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

发送信号

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

```
1. class Model(object):
2.     ...
3.
4.     def save(self):
5.         model_saved.send(self)
```

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

传递代理作为发送者

不要把 `current_app` 作为发送者传递给信号。请使用 `current_app._get_current_object()`。因为 `current_app` 是一个代理，不是实际的应用对象。

信号与 Flask 的请求环境

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

信号订阅装饰器

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

```
1. from flask import template_rendered
2.
3. @template_rendered.connect_via(app)
4. def when_template_rendered(sender, template, context, **extra):
5.     print 'Template %s is rendered with %s' % (template.name, context)
```

核心信号

所有内置信号请参阅 [Signals](#)。

可插拨视图

Changelog

New in version 0.7.

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

基本原理

假设有一个函数用于从数据库中载入一个对象列表并在模板中渲染：

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

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

```
1. from flask.views import View
2.
3. class ShowUsers(View):
4.
5.     def dispatch_request(self):
6.         users = User.query.all()
7.         return render_template('users.html', objects=users)
8.
9. app.add_url_rule('/users/', view_func=ShowUsers.as_view('show_users'))
```

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

```
1. from flask.views import View
2.
3. class ListView(View):
4.
5.     def get_template_name(self):
6.         raise NotImplementedError()
7.
8.     def render_template(self, context):
9.         return render_template(self.get_template_name(), **context)
10.
11.     def dispatch_request(self):
12.         context = {'objects': self.get_objects()}
```

```

13.         return self.render_template(context)
14.
15. class UserView(ListView):
16.
17.     def get_template_name(self):
18.         return 'users.html'
19.
20.     def get_objects(self):
21.         return User.query.all()

```

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

```

1. class RenderTemplateView(View):
2.     def __init__(self, template_name):
3.         self.template_name = template_name
4.     def dispatch_request(self):
5.         return render_template(self.template_name)

```

然后可以这样注册：

```

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

```

方法提示

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

```

1. class MyView(View):
2.     methods = ['GET', 'POST']
3.
4.     def dispatch_request(self):
5.         if request.method == 'POST':
6.             ...
7.         ...
8.
9. app.add_url_rule('/myview', view_func=MyView.as_view('myview'))

```

基于方法调度

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

```

1. from flask.views import MethodView
2.
3. class UserAPI(MethodView):
4.
5.     def get(self):
6.         users = User.query.all()
7.         ...
8.
9.     def post(self):
10.        user = User.from_form_data(request.form)
11.        ...
12.
13. app.add_url_rule('/users/', view_func=UserAPI.as_view('users'))

```

使用这种方式，不必提供 `methods` 属性，它会自动使用相应的类方法。

装饰视图

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

```

1. def user_required(f):
2.     """Checks whether user is logged in or raises error 401."""
3.     def decorator(*args, **kwargs):
4.         if not g.user:
5.             abort(401)
6.         return f(*args, **kwargs)
7.     return decorator
8.
9. view = user_required(UserAPI.as_view('users'))
10. app.add_url_rule('/users/', view_func=view)

```

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

```

1. class UserAPI(MethodView):
2.     decorators = [user_required]

```

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

用于 API 的方法视图

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

URL	方法	说明
<code>/users/</code>	<code>GET</code>	给出一个包含所有用户的列表
<code>/users/</code>	<code>POST</code>	创建一个新用户

<code>/users/<id></code>	GET	显示一个用户
<code>/users/<id></code>	PUT	更新一个用户
<code>/users/<id></code>	DELETE	删除一个用户

那么如何使用 `MethodView` 来实现呢？方法是使用多个规则对应到同一个视图。

假设视图是这样的：

```

1. class UserAPI(MethodView):
2.
3.     def get(self, user_id):
4.         if user_id is None:
5.             # 返回一个包含所有用户的列表
6.             pass
7.         else:
8.             # 显示一个用户
9.             pass
10.
11.    def post(self):
12.        # 创建一个新用户
13.        pass
14.
15.    def delete(self, user_id):
16.        # 删除一个用户
17.        pass
18.
19.    def put(self, user_id):
20.        # update a single user
21.        pass

```

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

```

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

```

如果你有许多类似的 API，那么可以代码如下：

```

1. def register_api(view, endpoint, url, pk='id', pk_type='int'):
2.     view_func = view.as_view(endpoint)
3.     app.add_url_rule(url, defaults={pk: None},
4.                      view_func=view_func, methods=['GET',])
5.     app.add_url_rule(url, view_func=view_func, methods=['POST',])
6.     app.add_url_rule('%s<%s:%s>' % (url, pk_type, pk), view_func=view_func,
7.                      methods=['GET', 'PUT', 'DELETE'])
8.
9. register_api(UserAPI, 'user_api', '/users/', pk='user_id')

```


应用情境

应用情境在请求，CLI 命令或其他活动期间跟踪应用级数据。不是将应用程序传递给每个函数，而是代之以访问 `current_app` 和 `g` 代理。

这与 [请求情境](#) 类似，它在请求期间跟踪请求级数据。推送请求情境时会推送相应的应用情境。

情境的目的

`Flask` 应用对象具有诸如 `config` 之类的属性，这些属性对于在视图和 `CLI commands` 中访问很有用。但是，在项目中的模块内导入 `app` 实例容易导致循环导入问题。当使用[应用程序工厂方案](#)或编写可重用的 `blueprints` 或 `extensions` 时，根本不会有应用程序实例导入。

- `Flask` 通过 应用情境 解决了这个问题。不是直接引用一个 `app`，而是使用
- `current_app` 代理，该代理指向处理当前活动的应用。

处理请求时，`Flask` 自动 推送 应用情境。在请求期间运行的视图函数、错误处理器和其他函数将有权访问 `current_app`。

运行使用 `@app.cli.command()` 注册到 `Flask.cli` 的 CLI 命令时，`Flask` 还会自动推送应用情境。

情境的生命周期

应用情境根据需要创建和销毁。当 `Flask` 应用开始处理请求时，它会推送应用情境和 [请求情境](#)。当请求结束时，它会在请求情境中弹出，然后在应用情境中弹出。通常，应用情境将具有与请求相同的生命周期。

请参阅 [请求情境](#) 以获取有关情境如何工作以及请求的完整生命周期的更多信息。

手动推送情境

如果您尝试在应用情境之外访问 `current_app`，或其他任何使用它的东西，则会看到以下错误消息：

```
1. RuntimeError: Working outside of application context.
2.
3. 这通常意味着您试图使用功能需要以某种方式与当前的应用程序对象进行交互。
4. 要解决这个问题，请使用 app.app_context() 设置应用情境。
```

如果在配置应用时发现错误（例如初始化扩展时），那么可以手动推送上下文。因为你可以直接访问 `app`。在 `with` 块中使用 `app_context()`，块中运行的所有内容都可以访问 `current_app`。：

```
1. def create_app():
2.     app = Flask(__name__)
3.
4.     with app.app_context():
5.         init_db()
```

```
6.
7.     return app
```

如果您在代码中的其他地方看到与配置应用无关的错误，则很可能表明应该将该代码移到视图函数或 CLI 命令中。

存储数据

应用情境是在请求或 CLI 命令期间存储公共数据的好地方。Flask 为此提供了 `g` 对象。它是一个简单的命名空间对象，与应用情境具有相同的生命周期。

Note

`g` 表示“全局”的意思，但是指的是数据在 情境 之中是全局的。`g` 中的数据在情境结束后丢失，因此它不是在请求之间存储数据的恰当位置。使用 `session` 或数据库跨请求存储数据。

`g` 的常见用法是在请求期间管理资源。

- `get_X()` 创建资源 `x`（如果它不存在），将其缓存为 `g.x`。
- `teardown_X()` 关闭或以其他方式解除分配资源（如果存在）。它被注册为 `teardown_appcontext()` 处理器。

例如，您可以使用以下方案管理数据库连接：

```
1. from flask import g
2.
3. def get_db():
4.     if 'db' not in g:
5.         g.db = connect_to_database()
6.
7.     return g.db
8.
9. @app.teardown_appcontext
10. def teardown_db():
11.     db = g.pop('db', None)
12.
13.     if db is not None:
14.         db.close()
```

在一个请求中，每次调用 `get_db()` 会返回同一个连接，并且会在请求结束时自动关闭连接。

你可以使用 `LocalProxy` 基于 `get_db()` 生成一个新的本地情境：

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

访问 `db` 就会内部调用 `get_db`，与 `current_app` 的工作方式相同。

如果你正在编写扩展，`g` 应该保留给用户。你可以将内部数据存储在情境本身中，但一定要使用足够唯一的名称。当前上下文使用 `_app_ctx_stack.top` 访问。欲了解更多信息，请参阅[Flask 扩展开发](#)。

事件和信号

当应用情境被弹出时，应用将调用使用 `teardown_appcontext()` 注册的函数。

如果 `signals_available` 为真，则发送以下信号：`appcontext_pushed` 、`appcontext_tearing_down` 和 `appcontext_popped` 。

请求情境

请求情境在请求期间跟踪请求级数据。不是将请求对象传递给请求期间运行的每个函数，而是访问 `request` 和 `session` 代理。

这类似于 [应用情境](#)，它跟踪独立于请求的应用级数据。推送请求情境时会推送相应的应用情境。

情境的用途

当 `Flask` 应用处理请求时，它会根据从 WSGI 服务器收到的环境创建一个 `Request` 对象。因为工作者（取决于服务器的线程，进程或协程）一次只能处理一个请求，所以在该请求期间请求数据可被认为是该工作者的全局数据。`Flask` 对此使用术语 [本地情境](#)。

处理请求时，`Flask` 自动推送请求情境。在请求期间运行的视图函数，错误处理器和其他函数将有权访问 `request` 代理，该请求代理指向当前请求的请求对象。

情境的生命周期

当 `Flask` 应用开始处理请求时，它会推送请求情境，这也会推送[应用情境](#)。当请求结束时，它会弹出请求情境，然后弹出应用程序情境。

情境对于每个线程（或其他工作者类型）是唯一的。`request` 不能传递给另一个线程，另一个线程将拥有不同的情境堆栈，并且不会知道父线程指向的请求。

本地情境在 `Werkzeug` 中实现。有关内部如何工作的更多信息，请参阅[Context Locals](#)。

手动推送情境

如果尝试在请求情境之外访问 `request` 或任何使用它的东西，那么会收到这个错误消息：

```
1. RuntimeError: Working outside of request context.
2.
3. 这通常表示您试图使用功能需要一个活动的 HTTP 请求。
4. 有关如何避免此问题的信息，请参阅测试文档
```

通常只有在测试代码期望活动请求时才会发生这种情况。一种选择是使用 `测试客户端` 来模拟完整的请求。或者，可以在 `with` 块中使用 `test_request_context()`，块中运行的所有内容都可以访问请求，并填充测试数据。：

```
1. def generate_report(year):
2.     format = request.args.get('format')
3.     ...
4.
5. with app.test_request_context(
6.     '/make_report/2017', data={'format': 'short'}):
7.     generate_report()
```

如果在你的代码中的其他地方看到与测试无关的错误，则说明可能应该将该代码移到视图函数中。

有关如何从交互式 Python shell 使用请求情境的信息，请参阅 [在 Shell 中使用 Flask](#)。

情境如何工作

处理每个请求时都会调用 `Flask.wsgi_app()` 方法。它在请求期间管理情境。在内部，请求和应用程序情境实质是 `_request_ctx_stack` 和 `_app_ctx_stack` 堆栈。当情境被压入堆栈时，依赖它们的代理可用并指向堆栈顶部情境中的信息。

当请求开始时，将创建并推送 `RequestContext`，如果该应用程序的情境尚不是顶级情境，则该请求会首先创建并推送 `AppContext`。在推送这些情境时，`current_app`、`g`、`request` 和 `session` 代理可用于处理请求的原始线程。

由于情境是堆栈，因此在请求期间可能会压入其他情境导致代理变更。虽然这不是一种常见模式，但它可以在高级应用使用。比如，执行内部重定向或将不同应用程序链接在一起。

在分派请求并生成和发送响应之后，会弹出请求情境，然后弹出应用情境。在紧临弹出之前，会执行 `teardown_request()` 和 `teardown_appcontext()` 函数。即使在调度期间发生未处理的异常，也会执行这些函数。

回调和错误

Flask 会在多个阶段调度请求，这会影响请求，响应以及如何处理错误。情境在所有这些阶段都处于活动状态。

`Blueprint` 可以为该蓝图的事件添加处理器，处理器会在蓝图与请求路由匹配的情况下运行。

- 在每次请求之前，`before_request()` 函数都会被调用。如果其中一个函数返回了一个值，则其他函数将被跳过。返回值被视为响应，并且视图函数不会被调用。
- 如果 `before_request()` 函数没有返回响应，则调用匹配路由的视图函数并返回响应。
- 视图的返回值被转换为实际的响应对象并传递给 `after_request()` 函数。每个函数都返回一个修改过的或新的响应对象。
- 返回响应后，将弹出情境，该情境调用 `teardown_request()` 和 `teardown_appcontext()` 函数。即使上面任何一处引发了未处理的异常，也会调用这些函数。

如果在拆卸函数之前引发了异常，Flask 会尝试将它与 `errorhandler()` 函数进行匹配，以处理异常并返回响应。如果找不到错误处理器，或者处理器本身引发异常，Flask 将返回一个通用的 `500 Internal Server Error` 响应。拆卸函数仍然被调用，并传递异常对象。

如果开启了调试模式，则未处理的异常不会转换为 `500` 响应，而是会传播到WSGI 服务器。这允许开发服务器向交互式调试器提供回溯。

拆解回调

拆除回调与请求派发无关，而在情境弹出时由情境调用。即使在调度过程中出现未处理的异常，以及手动推送的情

境，也会调用这些函数。这意味着不能保证请求调度的任何其他部分都先运行。 一定要以不依赖其他回调的方式编写这些函数，并且不会失败。

在测试期间，推迟请求结束后弹出情境会很有用，这样可以在测试函数中访问它们的数据。在 `with` 块中使用 `test_client()` 来保存情境，直到 `with` 块结束。

```
1. from flask import Flask, request
2.
3. app = Flask(__name__)
4.
5. @app.route('/')
6. def hello():
7.     print('during view')
8.     return 'Hello, World!'
9.
10. @app.teardown_request
11. def show_teardown(exception):
12.     print('after with block')
13.
14. with app.test_request_context():
15.     print('during with block')
16.
17. # teardown functions are called after the context with block exits
18.
19. with app.test_client() as client:
20.     client.get('/')
21.     # the contexts are not popped even though the request ended
22.     print(request.path)
23.
24. # the contexts are popped and teardown functions are called after
25. # the client with block exists
```

信号

如果 `signals_available` 为真，那么会发送以下信号：

- `request_started` 发送于 `before_request()` 函数被调用之前。
- `request_finished` 发送于 `after_request()` 函数被调用之后。
- `got_request_exception` 发送于异常开始处理的时候但早于 an `errorhandler()` 被找到或者调用的时候。
- `request_tearing_down` 发送于 `teardown_request()` 函数被调用之后。

出错情境保存

在请求结束时，会弹出请求情境，并且与其关联的所有数据都将被销毁。如果在开发过程中发生错误，延迟销毁数据以进行调试是有用的。

当开发服务器以开发模式运行时（ `FLASK_ENV` 环境变量设置为 `'development'` ），错误和数据将被保留并显示在

交互式调试器中。

该行为可以通过 `PRESERVE_CONTEXT_ON_EXCEPTION` 配置进行控制。如前文所述，它在开发环境中默认为 `True`。

不要在生产环境中启用 `PRESERVE_CONTEXT_ON_EXCEPTION`，因为它会导致应用在发生异常时泄漏内存。

关于代理的说明

Flask 提供的一些对象是其他对象的代理。每个工作线程都能以相同的方式访问代理，但是在后台每个工作线程绑定了唯一对象。

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

- 代理对象不能将它们的类型伪装为实际的对象类型。如果要执行实例检查，则必须检查被代理的原始对象。
- 对象引用非常重要的情况，例如发送 [信号](#) 或将数据传递给后台线程。

如果您需要访问被代理的源对象，请使用 `_get_current_object()` 方法：

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


使用蓝图的模块化应用

Changelog

New in version 0.7.

为了在一个或多个应用中，使应用模块化并且支持常用方案， Flask 引入了 蓝图概念。蓝图可以极大地简化大型应用并为扩展提供集中的注册入口。 `Blueprint` 对象与 `Flask` 应用对象的工作方式类似，但不是一个真正的应用。它更像一个用于构建和扩展应用的 蓝图 。

为什么使用蓝图？

Flask 中蓝图有以下用途：

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

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

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

蓝图的概念

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

第一个蓝图

以下是一个最基本的蓝图示例。在这里，我们将使用蓝图来简单地渲染静态模板：

```
1. from flask import Blueprint, render_template, abort
2. from jinja2 import TemplateNotFound
3.
4. simple_page = Blueprint('simple_page', __name__,
```

```

5.             template_folder='templates')
6.
7. @simple_page.route('/', defaults={'page': 'index'})
8. @simple_page.route('/<page>')
9. def show(page):
10.     try:
11.         return render_template('pages/%s.html' % page)
12.     except TemplateNotFound:
13.         abort(404)

```

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

注册蓝图

可以这样注册蓝图：

```

1. from flask import Flask
2. from yourapplication.simple_page import simple_page
3.
4. app = Flask(__name__)
5. app.register_blueprint(simple_page)

```

以下是注册蓝图后形成的规则：

```

1. >>> app.url_map
2. Map([<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
3. <Rule '/<page>' (HEAD, OPTIONS, GET) -> simple_page.show>,
4. <Rule '/' (HEAD, OPTIONS, GET) -> simple_page.show>])

```

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

蓝图还可以挂接到不同的位置：

```

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

```

这样就会形成如下规则：

```

1. >>> app.url_map
2. Map([<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
3. <Rule '/pages/<page>' (HEAD, OPTIONS, GET) -> simple_page.show>,
4. <Rule '/pages/' (HEAD, OPTIONS, GET) -> simple_page.show>])

```

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

蓝图资源

蓝图还可以用于提供资源。有时候，我们仅仅是为了使用一些资源而使用蓝图。

蓝图资源文件夹

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

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

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

可以使用 `open_resource()` 函数快速打开这个文件夹中的资源：

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

静态文件

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

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

缺省情况下，路径最右端的部分是在 URL 中暴露的部分。这可以通过 `static_url_path` 来改变。因为上例中的文件夹为名称是 `static`，那么URL 应该是蓝图的 `url_prefix` 加上 `/static`。如果蓝图注册前缀为 `/admin`，那么静态文件 URL 就是 `/admin/static`。

端点的名称是 `blueprint_name.static`。你可以像对待应用中的文件夹一样使用 `url_for()` 来生成其 URL：

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

但是，如果蓝图没有 `url_prefix`，那么不可能访问蓝图的静态文件夹。这是因为在这种情况下，URL应该是 `/static`，而应用程序的 `/static` 路线优先。与模板文件夹不同，如果文件不存在于应用静态文件夹中，那么不会搜索蓝图静态文件夹。

模板

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

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

对于静态文件，路径可以是绝对的或相对于蓝图的资源文件夹。

模板文件夹被添加到模板的搜索路径，但优先级低于实际应用的模板文件夹。这样就可以轻松地重载在实际应用中蓝图提供的模板。这也意味着如果你不希望蓝图模板出现意外重写，那么就要确保没有其他蓝图或实际的应用模板具有相同的相对路径。多个蓝图提供相同的相对路径时，第一个注册的优先。

假设你的蓝图便于 `yourapplication/admin` 中，要渲染的模板是 `'admin/index.html'`，`template_folder` 参数值为 `templates`，那么真正的模板文件为：`yourapplication/admin/templates/admin/index.html`。多出一个 `admin` 文件夹是为了避免模板被实际应用模板文件夹中的 `index.html` 重载。

更详细一点说：如果你有一个名为 `admin` 的蓝图，该蓝图指定的模版文件是 `index.html`，那么最好按照如下结构存放模版文件：

```
1. yourpackage/
2.     blueprints/
3.         admin/
4.             templates/
5.                 admin/
6.                     index.html
7.             __init__.py
```

这样，当你需要渲染模板的时候就可以使用 `admin/index.html` 来找到模板。如果没有载入正确的模板，那么应该启用 `EXPLAIN_TEMPLATE_LOADING` 配置变量。启用这个变量以后，每次调用 `render_template` 时，Flask 会打印出定位模板的步骤，方便调试。

创建 URL

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

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

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

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

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

错误处理器

蓝图像 `Flask` 应用对象一样支持 `errorhandler` 装饰器，所以很容易使用蓝图特定的自定义错误页面。

下面是“404 Page Not Found”异常的例子：

```
1. @simple_page.errorhandler(404)
2. def page_not_found(e):
```

```
3.     return render_template('pages/404.html')
```

大多数错误处理器会按预期工作。然而，有一个涉及 404 和 405 例外处理器的警示。这些错误处理器只会由一个适当的 `raise` 语句引发或者调用在另一个蓝图视图中调用 `abort` 引发。它们不会引发于无效的 URL 访问。这是因为蓝图不“拥有”特定的 URL 空间，在发生无效 URL 访问时，应用实例无法知道应该运行哪个蓝图错误处理器。如果你想基于 URL 前缀执行不同的错误处理策略，那么可以在应用层使用 `request` 代理对象定义它们：

```
1. @app.errorhandler(404)
2. @app.errorhandler(405)
3. def _handle_api_error(ex):
4.     if request.path.startswith('/api/'):
5.         return jsonify_error(ex)
6.     else:
7.         return ex
```

更多关于错误处理信息参见 [自定义出错页面](#)。

扩展

扩展是指为 Flask 应用增加功能的包，比如增加发送电子邮件或者连接数据库中的功能。有些扩展还有助于为应用添加全新的框架，如 REST API 。

寻找扩展

Flask 的扩展通常命名为“ Flask-Foo ”或者“ Foo-Flask ”。许多扩展已经列入了 [扩展仓库](#)，由其开发者维护。还可以在 PyPI 搜索标记为 `Framework :: Flask` 扩展包。

使用扩展

请参阅每个扩展的文档以了解其安装、配置和使用说明。一般来说，扩展从 `app.config` 获取其自身的配置并在初始化时传递给应用实例。例如，一个名为“ Flask-Foo ”的扩展使用如下：

```
1. from flask_foo import Foo
2.
3. foo = Foo()
4.
5. app = Flask(__name__)
6. app.config.update(
7.     FOO_BAR='baz',
8.     FOO_SPAM='eggs',
9. )
10.
11. foo.init_app(app)
```

创建扩展

虽然 [扩展仓库](#) 已经包含许多 Flask 扩展，但是如果找不到合适的，那么可以创建自己的扩展。如何创建扩展请参阅 [Flask 扩展开发](#)。

命令行接口

在虚拟环境中安装 Flask 时会同时安装 `flask` 脚本，这是一个 Click 命令行接口。在终端中执行该脚本可以操作内建的、扩展的和应用定义的命令。关于命令的更多信息和选择可以通过使用 `-help` 参数查看。

探索应用

`flask` 命令由 Flask 安装，而不是你的应用。为了可以使用，它必须被告知可以在哪里找到你的应用。

`FLASK_APP` 环境变量用于定义如何载入应用。

Unix Bash (Linux 、 Mac 及其他)：

```
1. $ export FLASK_APP=hello
2. $ flask run
```

Windows CMD：

```
1. > set FLASK_APP=hello
2. > flask run
```

Windows PowerShell：

```
1. > $env:FLASK_APP = "hello"
2. > flask run
```

虽然 `FLASK_APP` 支持多种选项来定义应用，但多数情况下应该很简单。以下是典型值：

- (空)
- `wsgi.py` 文件被导入，自动探测应用 (`app`)。这提供了一种简单的方法来从工厂创建具有额外参数的应用。
- `FLASK_APP=hello`
- 名称被导入，自动探测一个应用 (`app`) 或者工厂 (`create_app`)。

`FLASK_APP` 分三个部分：一是一个可选路径，用于设置当前工作文件夹；二是一个 Python 文件或者带点的导入路径；三是一个可选的实例或工厂的变量名称。如果名称是工厂，则可以选择在后面的括号中加上参数。以下演示说明：

- `FLASK_APP=src/hello`
- 设置当前工作文件夹为 `src` 然后导入 `hello` 。
- `FLASK_APP=hello.web`

- 导入路径 `hello.web` 。
- `FLASK_APP=hello:app2`
- 使用 `hello` 中的 `app2` Flask 实例。
- `FLASK_APP="hello:create_app('dev')"`
- 调用 `hello` 中的 `create_app` 工厂，把 `'dev'` 作为参数。

如果没有设置 `FLASK_APP`，命令会查找 `wsgi.py` 文件或者 `app.py` 文件并尝试探测一个应用实例或者工厂。

根据给定的导入，命令会寻找一个名为 `app` 或者 `application` 的应用实例。如果找不到会继续寻找任意应用实例。如果找不到任何实例，会接着寻找名为 `create_app` 或者 `make_app` 的函数，使用该函数返回的实例。

当调用一个应用工厂时，如果工厂接收一个名为 `info` 的参数，那么 `class:~cli.ScriptInfo` 实例会被作为一个关键字参数传递。如果括号紧随着工厂名称，那么其中的内容会被视作为 Python 语言内容，并用作函数的参数。这意味着字符串必须使用双引号包围。

运行开发服务器

`run` 命令可以启动开发服务器，它在大多数情况下替代 `Flask.run()` 方法。：

```
1. $ flask run
2. * Serving Flask app "hello"
3. * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Warning

不要在生产中使用此命令运行应用，只能在开发过程中使用开发服务器。开发服务器只是为了提供方便，但是不够安全、稳定和高效。有关如何在生产中运行服务器，请参阅 [部署方式](#)。

打开一个 Shell

为了探索应用中的数据，可以 `shell` 命令开启一个交互 Python shell。这样，一个应用情境被激活，应用实例会被导入。：

```
1. $ flask shell
2. Python 3.6.2 (default, Jul 20 2017, 03:52:27)
3. [GCC 7.1.1 20170630] on linux
4. App: example
5. Instance: /home/user/Projects/hello/instance
6. >>>
```

使用 `shell_context_processor()` 添加其他自动导入。

环境

Changelog

New in version 1.0.

Flask 应用所运行的环境由 `FLASK_ENV` 环境变更指定。如果配置该变量，那么缺省为 `production`。另一个可用的环境值是 `development`。Flask 和扩展可能基于环境不同而改变行为。

如果环境是 `development`，`flask` 命令会开启调试模式并且 `flask run` 会开启交互调试器和重启器。

```
1. $ FLASK_ENV=development flask run
2.  * Serving Flask app "hello"
3.  * Environment: development
4.  * Debug mode: on
5.  * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
6.  * Restarting with inotify reloader
7.  * Debugger is active!
8.  * Debugger PIN: 223-456-919
```

调试模式

如前文所述，当 `FLASK_ENV` 是 `development` 时会开启调试模式。如果想要单独控制调试模式，要使用 `FLASK_DEBUG`。值为 `1` 表示开启，`0` 表示关闭。

通过 dotenv 设置环境变量

与其每次打开新的终端都要设置 `FLASK_APP`，不如使用 Flask 的 `dotenv` 支持功能自动设置环境变量。

如果 `python-dotenv` 已安装，那么运行 `flask` 会根据 `.env` 和 `.flaskenv` 中配置来设置环境变量。这样可以在每次打开终端后，避免手动设置 `FLASK_APP` 和其他类似使用环境变量进行配置的服务部署工作。

命令行设置的变量会重载 `.env` 中的变量，`.env` 中的变量会重载 `.flaskenv` 中的变量。`.flaskenv` 应当用于公共变量，如 `FLASK_APP` 而 `.env` 则应用于私有变量，并且不提交到储存库。

为了找到定位文件，将会从运行 `flask` 的文件夹向上扫描文件夹。当前工作目录将被设置为文件的位置，假定这是最高级别的项目文件夹。

这些文件只能由 `flask` 命令或调用 `run()` 加载。如果想在生产运行时加载这些文件，你应该手动调用 `load_dotenv()`。

设置命令参数

Click 被配置为根据环境变量为命令选项载入缺省值。变量使用 `FLASK_COMMAND_OPTION` 模式。例如，要为运行命令设置端口，不使用 `flask run --port 8000`，而是使用：

```
1. export FLASK_RUN_PORT=8000
2. flask run
3.  * Running on http://127.0.0.1:8000/
```

这些可以添加到 `.flaskenv` 文件，就像 `FLASK_APP` 来控制缺省命令选项。

禁用 dotenv

如果检测到 `dotenv` 文件，但是没有安装 `python-dotenv`，那么 `flask` 命令会显示一个消息。

```
1. flask run
2. * Tip: There are .env files present. Do "pip install python-dotenv" to use them.
```

通过设置 `FLASK_SKIP_DOTENV` 可以告诉 Flask 不要载入 `dotenv` 文件。在 `python-dotenv` 没有安装到情况下这个设置也是有效的。这个设置主要用于以下情形：当你想要手动载入它们的时候，或者当你已经使用了一个项目运行器载入了它们。请牢记，环境变量必须在项目载入之前设置，否则出问题。

```
1. export FLASK_SKIP_DOTENV=1
2. flask run
```

通过 virtualenv 设置环境变量

如果不想安装 `dotenv` 支持，可以通过把它们添加到 `virtualenv` 的 `activate` 文件末尾来设置环境变量。激活 `virtualenv` 时会设置环境变量。

Unix Bash，`venv/bin/activate`：

```
1. export FLASK_APP=hello
```

Windows CMD，`venv\Scripts\activate.bat`：

```
1. set FLASK_APP=hello
```

建议使用 `dotenv` 支持来做，因为 `.flaskenv` 可以被提交到储存库，当提取项目代码后就可以自动发挥作用。

自定义命令

`flask` 命令使用 `Click` 来实现。如何编写命令的完整信息参见该项目的文档。

以下示例添加了 `create-user` 命令，带有 `name` 参数。

```
1. import click
2. from flask import Flask
3.
4. app = Flask(__name__)
5.
6. @app.cli.command("create-user")
7. @click.argument("name")
8. def create_user(name):
9.     ...
```

```
1. $ flask create-user admin
```

以下示例也添加了同样功能的命令，但是以命令组的方式添加的，名为 `user create` 。这样做有助于组织一组相关的命令。

```
1. import click
2. from flask import Flask
3. from flask.cli import AppGroup
4.
5. app = Flask(__name__)
6. user_cli = AppGroup('user')
7.
8. @user_cli.command('create')
9. @click.argument('name')
10. def create_user(name):
11.     ...
12.
13. app.cli.add_command(user_cli)
```

```
1. flask user create demo
```

关于如何测试自定义命令的概览，参见 [测试 CLI 命令](#) 。

以蓝图注册命令

如果你的应用使用蓝图，那么可以把 CLI 命令 直接注册到蓝图上。当蓝图注册到应用上的时候，相关的命令就可以应用于 `flask` 命令了。缺省情况下，那些命令会嵌套于一个与蓝图相关匹配的组。

```
1. from flask import Blueprint
2.
3. bp = Blueprint('students', __name__)
4.
5. @bp.cli.command('create')
6. @click.argument('name')
7. def create(name):
8.     ...
9.
10. app.register_blueprint(bp)
```

```
1. $ flask students create alice
```

组名称可以在创建 `Blueprint` 对象时通过 `cli_group` 参数定义，也可以创建之后使用 `app.register_blueprint(bp, cli_group='...')` 来变更。下面两条命令功能是相同的：

```
1. bp = Blueprint('students', __name__, cli_group='other')
2. # or
```

```
3. app.register_blueprint(bp, cli_group='other')
```

```
1. $ flask other create alice
```

指定 `cli_group=None` 会删除嵌套并把命令直接合并到应用级别：

```
1. bp = Blueprint('students', __name__, cli_group=None)
2. # or
3. app.register_blueprint(bp, cli_group=None)
```

```
1. $ flask create alice
```

应用情境

使用 Flask 应用的 `cli` `command()` 装饰器添加的命令会在执行时压入应用情境，这样命令和扩展就可以访问应用和应用的配置。如果使用 Click 的 `command()` 装饰器创建命令，而不是 Flask 的装饰器，那么可以使用 `with_appcontext()`，达到同样的效果。

```
1. import click
2. from flask.cli import with_appcontext
3.
4. @click.command
5. @with_appcontext
6. def do_work():
7.     ...
8.
9. app.cli.add_command(do_work)
```

如果确定命令不需要情境，那么可以禁用它：

```
1. @app.cli.command(with_appcontext=False)
2. def do_work():
3.     ...
```

插件

Flask 会自动载入在 `flask.commands` `entry point` 定义的命令。这样有助于扩展在安装时添加命令。入口点在 `setup.py` 中定义：

```
1. from setuptools import setup
2.
3. setup(
4.     name='flask-my-extension',
5.     ...,
6.     entry_points={
7.         'flask.commands': [
```

```

8.         'my-command=flask_my_extension.commands:cli'
9.     ],
10. },
11. )

```

在 `flask_my_extension/commands.py` 内可以导出一个 Click 对象：

```

1. import click
2.
3. @click.command()
4. def cli():
5.     ...

```

一旦该软件包与 Flask 项目安装在相同的 `virtualenv` 中，你可以运行 `flask my-command` 来调用该命令。

自定义脚本

当使用应用工厂方案时，自定义 Click 脚本会更方便。这样可以创建自己的 Click 对象并导出它作为一个 `console script` 入口点，而不是使用 `FLASK_APP` 并让 Flask 载入应用。

创建一个 `FlaskGroup` 的实例并传递给工厂：

```

1. import click
2. from flask import Flask
3. from flask.cli import FlaskGroup
4.
5. def create_app():
6.     app = Flask('wiki')
7.     # other setup
8.     return app
9.
10. @click.group(cls=FlaskGroup, create_app=create_app)
11. def cli():
12.     """Management script for the Wiki application."""

```

在 `setup.py` 中定义入口点：

```

1. from setuptools import setup
2.
3. setup(
4.     name='flask-my-extension',
5.     ...,
6.     entry_points={
7.         'console_scripts': [
8.             'wiki=wiki:cli'
9.         ],
10.     },
11. )

```

在 `virtualenv` 中以可编辑模式安装应用，自定义脚本可用。注意，不需要设置 `FLASK_APP` 。

```
1. $ pip install -e .
2. $ wiki run
```

自定义脚本错误

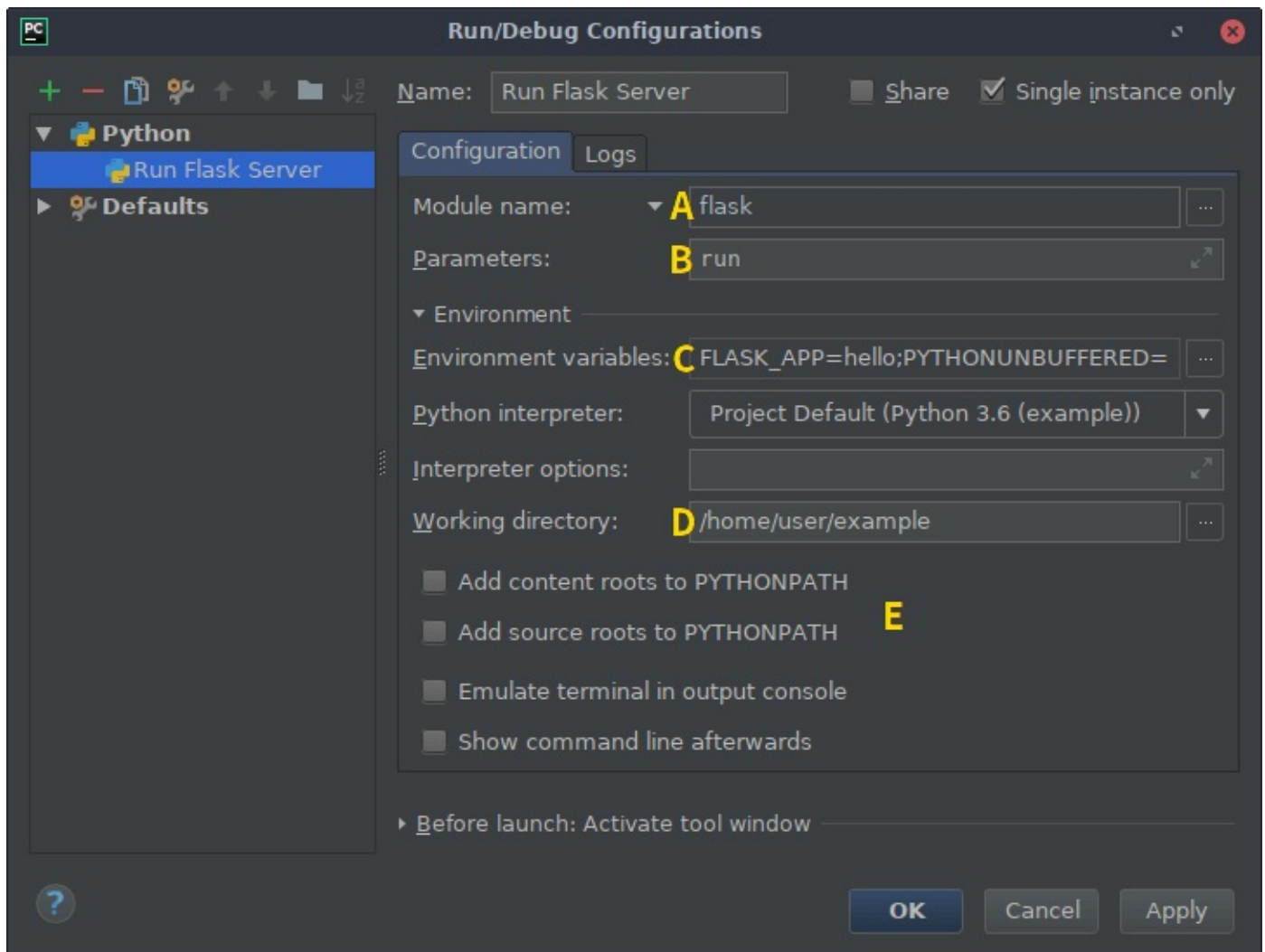
当使用自定义脚本时，如果模块级别代码出错，重载器会失效，因为它无法再载入口点。

一般建议使用 `flask` 命令，因为该命令与你的代码是分离的，不会出现这种问题。

PyCharm 集成

在 PyCharm 2018.1 版本之前，Flask CLI 功能尚未能完全整合进 PyCharm，因此我们必需做些调整才能顺利使用。这些说明同样也适用于其他 IDE。

在 PyCharm 中，打开你的项目，在菜单中点击 `Run` 后点击 `Edit Configurations`。你会看到类似如下窗口：



有许多选项要改变，但一旦做好了一条命令，其他命令只要复制整个配置调整一下就行了。包括你想自定义的其他命令也同样如此。

点击 `+` (`Add New Configuration`) 按钮选择 `Python`。为配置取一个具有良好描述性的名称，例如“ `Run Flask Server` ”(运行 Flask 服务器)。为 `flask run` 命令选择“ `Single instance only` ”，因为不能

同时运行多个服务器。

从 (**A**) 下拉框中选择 *Module name* , 然后输入 `flask` 。

Parameters 字段 (**B**) 用于设置 CLI 命令 (可以带参数)。本例中我们使用 `run` , 它可用于运行开发服务器。

如果使用 [通过 dotenv 设置环境变量](#) 可以跳过这一步。为了标识我们的应用, 需要添加一个环境变量 (**C**)。点击展开按钮, 并在左边添加一个 `FLASK_APP` 条目, Python 要导入的库或者文件放在右边 (本例使用 `hello`)。

下一步我们需要把应用所在的文件夹设置为工作文件夹 (**D**)。

如果你是在 `virtualenv` 中把项目作为一个包安装的, 那么可以取消选择 `PYTHONPATH` 选项 (**E**), 这样会与以后部署应用更匹配。

点击 *Apply* 保存配置, 或者点击 *OK* 保存并关闭窗口。在 PyCharm 主窗口中选择该配置并点击旁边的运行按钮开始运行服务器。

现在我们已经在 PyCharm 中有了一个运行 `flask run` 的配置, 复制这个配置并改变 *Script* 参数, 就可以运行一个不同的 CLI 命令, 比如 `flask shell` 。

开发服务器

自 Flask 0.11 开始有多种内建方法可以运行开发服务器。最好的方法是使用 **flask** 命令行工具。当然，继续使用 `Flask.run()` 亦可。

通过命令行使用开发服务器

强烈推荐开发时使用 **flask** 命令行脚本（[命令行接口](#)），因为有强大的重载功能，提供了超好的重载体验。基本用法如下：

```
1. $ export FLASK_APP=my_application
2. $ export FLASK_ENV=development
3. $ flask run
```

这样做开始了开发环境（包括交互调试器和重载器），并在 <http://localhost:5000/> 提供服务。

通过使用不同 `run` 参数可以控制服务器的单独功能。例如禁用重载器：

```
1. $ flask run --no-reload
```

Note

在 Flask 1.0 版之前，`FLASK_ENV` 环境不可用。开启调试模式需要使用 `FLASK_DEBUG=1`。这样做还是有用的，但是建议如前文所述使用设置开发环境变量来实现。

通过代码使用开发服务器

另一种方法是通过 `Flask.run()` 方法启动应用，这样立即运行一个本地服务器，与使用 **flask** 脚本效果相同。

示例：

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

通常情况下这样做不错，但是对于开发就不行了。正是基于这个原因自 Flask 0.11版开始推荐使用 **flask** 方法。这是因为重载的工作机制有一些奇怪的副作用（如执行某些代码两次，有时会在没有消息的情况下崩溃，或者在某个语法或导入错误发生时宕机）。

然而，它仍然是一个调用非自动重装应用程序的非常有效的方法。

在 Shell 中使用 Flask

Changelog

New in version 0.3.

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

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

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

在读下去之前最好你已经读过 [请求情境](#) 一节。

命令行接口

自 Flask 0.11 版开始，推荐在 shell 中使用 `flask shell` 命令，它可以为你做许多自动化工作。比如在 shell 中自动初始化应用情境。

更多信息参见 [命令行接口](#)。

创建一个请求情境

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

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

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

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

从这里开始，直到调用 `pop` 之前，你可以使用请求对象：

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

发送请求前/后动作

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

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

```
1. >>> ctx = app.test_request_context()
2. >>> ctx.push()
3. >>> app.preprocess_request()
```

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

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

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

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

在 Shell 中玩得更爽

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

```
1. >>> from shelltools import *
```

Flask 方案

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

更多用户贡献的代码片断和方案参见 [Flask代码片断归档](#)。

- [大型应用](#)
 - [简单的包](#)
 - [使用蓝图](#)
- [应用工厂](#)
 - [基础工厂](#)
 - [工厂与扩展](#)
 - [使用应用](#)
 - [改进工厂](#)
- [应用调度](#)
 - [说明](#)
 - [组合应用](#)
 - [根据子域调度](#)
 - [根据路径调度](#)
- [实现 API 异常](#)
 - [简单异常类](#)
 - [注册一个错误处理器](#)
 - [在视图中的用法](#)
- [URL 处理器](#)
 - [国际化应用的 URL](#)
 - [国际化的蓝图 URL](#)
- [使用 Setuptools 部署](#)
 - [基础设置脚本](#)
 - [标记构建版本](#)
 - [分发资源](#)
 - [声明依赖](#)
 - [安装 / 开发](#)
- [使用 Fabric 部署](#)
 - [创建第一个 Fabfile](#)
 - [运行 Fabfile](#)
 - [WSGI 文件](#)
 - [配置文件](#)
 - [第一次部署](#)
 - [下一步](#)
- [使用 SQLite 3](#)
 - [按需连接](#)
 - [简化查询](#)
 - [初始化模式](#)
- [使用 SQLAlchemy](#)

- [Flask-SQLAlchemy 扩展](#)
- [声明](#)
- [人工对象关系映射](#)
- [SQL 抽象层](#)
- [上传文件](#)
 - [简介](#)
 - [改进上传](#)
 - [上传进度条](#)
 - [一个更简便的方案](#)
- [缓存](#)
- [视图装饰器](#)
 - [检查登录装饰器](#)
 - [缓存装饰器](#)
 - [模板装饰器](#)
 - [端点装饰器](#)
- [使用 WTForms 进行表单验证](#)
 - [表单](#)
 - [视图](#)
 - [模板中的表单](#)
- [模板继承](#)
 - [基础模板](#)
 - [子模板](#)
- [消息闪现](#)
 - [简单的例子](#)
 - [闪现消息的类别](#)
 - [过滤闪现消息](#)
- [通过 jQuery 使用 AJAX](#)
 - [载入 jQuery](#)
 - [我的网站在哪里？](#)
 - [JSON 视图函数](#)
 - [HTML](#)
- [自定义出错页面](#)
 - [常见出错代码](#)
 - [出错处理器](#)
 - [以 JSON 格式返回 API 错误](#)
- [惰性载入视图](#)
 - [转换为集中 URL 映射](#)
 - [延迟载入](#)
- [通过 MongoEngine 使用 MongoDB](#)
 - [配置](#)
 - [映射文档](#)
 - [创建数据](#)
 - [查询](#)
 - [相关文档](#)
- [添加一个页面图标](#)
 - [另见](#)

- [流内容](#)
 - [基本用法](#)
 - [模板中的流内容](#)
 - [情境中的流内容](#)
- [延迟的请求回调](#)
- [添加 HTTP 方法重载](#)
- [请求内容校验](#)
- [基于 Celery 的后台任务](#)
 - [安装](#)
 - [配置](#)
 - [一个示例任务](#)
 - [运行 Celery 工人](#)
- [继承 Flask](#)
- [单页应用](#)

大型应用

假设有一个简单的应用结构如下：

```
1. /yourapplication
2.     yourapplication.py
3.     /static
4.         style.css
5.     /templates
6.         layout.html
7.         index.html
8.         login.html
9.         ...
```

这个结构对于小应用来说没有问题，但是对于大应用来说就不好了，应当用包来代替模块。[教程项目](#) 就使用了包方案，参见[示例代码](#)。

简单的包

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

修改完后应该如下例：

```
1. /yourapplication
2.     /yourapplication
3.         __init__.py
4.         /static
5.             style.css
6.         /templates
7.             layout.html
8.             index.html
9.             login.html
10.            ...
```

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

```
1. from setuptools import setup
2.
3. setup(
4.     name='yourapplication',
5.     packages=['yourapplication'],
6.     include_package_data=True,
7.     install_requires=[
```

```

8.         'flask',
9.     ],
10. )

```

为了运行应用，需要导出一个环境变量，告诉 Flask 应用实例的位置：

```
1. $ export FLASK_APP=yourapplication
```

如果位于项目文件夹之外，请确保提供绝对路径。同样可以这样打开开发功能：

```
1. $ export FLASK_ENV=development
```

为了安装并运行应用，需要执行以下命令：

```

1. $ pip install -e .
2. $ flask run

```

我们从中学到了什么？现在我们来重构一下应用以适应多模块。只要记住以下几点：

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

`init.py` 示例：

```

1. from flask import Flask
2. app = Flask(__name__)
3.
4. import yourapplication.views

```

`views.py` 内容如下：

```

1. from yourapplication import app
2.
3. @app.route('/')
4. def index():
5.     return 'Hello World!'

```

最终全部内容如下：

```

1. /yourapplication
2.     setup.py
3. /yourapplication
4.     __init__.py
5.     views.py
6.     /static

```

```
7.         style.css
8.     /templates
9.         layout.html
10.        index.html
11.        login.html
12.        ...
```

回环导入

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

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

使用蓝图

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

应用工厂

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

那么这样做有什么用呢？

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

那么如何做呢？

基础工厂

方法是在一个函数中设置应用，具体如下：

```
1. def create_app(config_filename):
2.     app = Flask(__name__)
3.     app.config.from_pyfile(config_filename)
4.
5.     from yourapplication.model import db
6.     db.init_app(app)
7.
8.     from yourapplication.views.admin import admin
9.     from yourapplication.views.frontend import frontend
10.    app.register_blueprint(admin)
11.    app.register_blueprint(frontend)
12.
13.    return app
```

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

```
1. from flask import current_app, Blueprint, render_template
2. admin = Blueprint('admin', __name__, url_prefix='/admin')
3.
4. @admin.route('/')
5. def index():
6.     return render_template(current_app.config['INDEX_TEMPLATE'])
```

这里我们在配置中查找模板的名称。

工厂与扩展

最好分别创建扩展和应用工厂，这样扩展对象就不会过早绑定到应用。

以使用 `Flask-SQLAlchemy` 为例，不应当这样：

```
1. def create_app(config_filename):
2.     app = Flask(__name__)
3.     app.config.from_pyfile(config_filename)
4.
5.     db = SQLAlchemy(app)
```

而是在 `model.py`（或其他等价文件）中：

```
1. db = SQLAlchemy()
```

在 `application.py`（或其他等价文件）中：

```
1. def create_app(config_filename):
2.     app = Flask(__name__)
3.     app.config.from_pyfile(config_filename)
4.
5.     from yourapplication.model import db
6.     db.init_app(app)
```

使用这个设计方案，不会有应用特定状态储存在扩展对象上，因此扩展对象就可以被多个应用使用。更多关于扩展设计的信息参见 [Flask 扩展开发](#)。

使用应用

使用 `flask` 命令运行工厂应用：

```
1. $ export FLASK_APP=myapp
2. $ flask run
```

`Flask` 会自动在 `myapp` 中探测工厂（`create_app` 或者 `make_app`）。还可这样向工厂传递参数：

```
1. $ export FLASK_APP="myapp:create_app('dev')"
2. $ flask run
```

这样，`myapp` 中的 `create_app` 工厂就会使用 `'dev'` 作为参数。更多细节参见 [命令行接口](#)。

改进工厂

上面的工厂函数还不是足够好，可以改进的地方主要有以下几点：

- 为了单元测试，要想办法传入配置，这样就不必在文件系统中创建配置文件。

- 当设置应用时从蓝图调用一个函数，这样就可以有机会修改属性（如挂接请求前/后处理器等）。
- 如果有必要的话，当创建一个应用时增加一个 WSGI 中间件。

应用调度

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

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

说明

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

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

注意 `run_simple` 不能用于生产环境，生产环境服务器参见 [成熟的 WSGI 服务器](#)。

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

```
1. from flask import Flask
2. from werkzeug.serving import run_simple
3.
4. app = Flask(__name__)
5. app.debug = True
6.
7. @app.route('/')
8. def hello_world():
9.     return 'Hello World!'
10.
11. if __name__ == '__main__':
12.     run_simple('localhost', 5000, app,
13.               use_reloader=True, use_debugger=True, use_evalex=True)
```

组合应用

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

假设你的主应用运行于 `/`，后台接口位于 `/backend`：

```

1. from werkzeug.wsgi import DispatcherMiddleware
2. from frontend_app import application as frontend
3. from backend_app import application as backend
4.
5. application = DispatcherMiddleware(frontend, {
6.     '/backend': backend
7. })

```

根据子域调度

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

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

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

```

1. from threading import Lock
2.
3. class SubdomainDispatcher(object):
4.
5.     def __init__(self, domain, create_app):
6.         self.domain = domain
7.         self.create_app = create_app
8.         self.lock = Lock()
9.         self.instances = {}
10.
11.     def get_application(self, host):
12.         host = host.split(':')[0]
13.         assert host.endswith(self.domain), 'Configuration error'
14.         subdomain = host[:-len(self.domain)].rstrip('.')
15.         with self.lock:
16.             app = self.instances.get(subdomain)
17.             if app is None:
18.                 app = self.create_app(subdomain)
19.                 self.instances[subdomain] = app
20.             return app
21.
22.     def __call__(self, environ, start_response):
23.         app = self.get_application(environ['HTTP_HOST'])
24.         return app(environ, start_response)

```

调度器示例：

```

1. from myapplication import create_app, get_user_for_subdomain
2. from werkzeug.exceptions import NotFound
3.

```

```

4. def make_app(subdomain):
5.     user = get_user_for_subdomain(subdomain)
6.     if user is None:
7.         # 如果子域没有对应的用户，那么还是得返回一个 WSGI 应用
8.         # 用于处理请求。这里我们把 NotFound() 异常作为应用返回，
9.         # 它会被渲染为一个缺省的 404 页面。然后，可能还需要把
10.        # 用户重定向到主页。
11.        return NotFound()
12.
13.    # 否则为特定用户创建应用
14.    return create_app(user)
15.
16. application = SubdomainDispatcher('example.com', make_app)

```

根据路径调度

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

```

1. from threading import Lock
2. from werkzeug.wsgi import pop_path_info, peek_path_info
3.
4. class PathDispatcher(object):
5.
6.     def __init__(self, default_app, create_app):
7.         self.default_app = default_app
8.         self.create_app = create_app
9.         self.lock = Lock()
10.        self.instances = {}
11.
12.    def get_application(self, prefix):
13.        with self.lock:
14.            app = self.instances.get(prefix)
15.            if app is None:
16.                app = self.create_app(prefix)
17.                if app is not None:
18.                    self.instances[prefix] = app
19.            return app
20.
21.    def __call__(self, environ, start_response):
22.        app = self.get_application(peek_path_info(environ))
23.        if app is not None:
24.            pop_path_info(environ)
25.        else:
26.            app = self.default_app
27.        return app(environ, start_response)

```

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

```

1. from myapplication import create_app, default_app, get_user_for_prefix

```

```
2.  
3. def make_app(prefix):  
4.     user = get_user_for_prefix(prefix)  
5.     if user is not None:  
6.         return create_app(user)  
7.  
8. application = PathDispatcher(default_app, make_app)
```

实现 API 异常

在 Flask 上实现 RESTful API 很普通。开发人员遇到最头疼的一件事是意识到内建的异常对 API 没有足够的表达能力，并且它们触发的 `text/html` 的内容类型对于 API 消费者来说如同鸡肋。

对于非法使用 API 来说，比仅仅对信号错误使用 `abort` 更好的是，实现你自己的异常类型并为之安装一个错误处理器，这样更符合用户的预期。

简单异常类

基本的思路是引入一个新异常，提供具有高可读性的信息、错误状态码和一些为错误提供更多情境的负载。

下面是一个简单的例子：

```
1. from flask import jsonify
2.
3. class InvalidUsage(Exception):
4.     status_code = 400
5.
6.     def __init__(self, message, status_code=None, payload=None):
7.         Exception.__init__(self)
8.         self.message = message
9.         if status_code is not None:
10.             self.status_code = status_code
11.         self.payload = payload
12.
13.     def to_dict(self):
14.         rv = dict(self.payload or ())
15.         rv['message'] = self.message
16.         return rv
```

现在，一个视图可以引发带有消息的异常。通过 `payload` 参数，可以以字典方式提供一些额外的负载。

注册一个错误处理器

此时，视图可以引发异常，但会立即导致内部服务器错误。原因是没有为这个错误类注册处理器。注册示例如下：

```
1. @app.errorhandler(InvalidUsage)
2. def handle_invalid_usage(error):
3.     response = jsonify(error.to_dict())
4.     response.status_code = error.status_code
5.     return response
```

在视图中的用法

以下是视图使用示例：

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

URL 处理器

Changelog

New in version 0.7.

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

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

国际化应用的 URL

假设有应用如下：

```
1. from flask import Flask, g
2.
3. app = Flask(__name__)
4.
5. @app.route('/<lang_code>/')
6. def index(lang_code):
7.     g.lang_code = lang_code
8.     ...
9.
10. @app.route('/<lang_code>/about')
11. def about(lang_code):
12.     g.lang_code = lang_code
13.     ...
```

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

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

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

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

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

```

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

```

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

```

1. from flask import Flask, g
2.
3. app = Flask(__name__)
4.
5. @app.url_defaults
6. def add_language_code(endpoint, values):
7.     if 'lang_code' in values or not g.lang_code:
8.         return
9.     if app.url_map.is_endpoint_expecting(endpoint, 'lang_code'):
10.         values['lang_code'] = g.lang_code
11.
12. @app.url_value_preprocessor
13. def pull_lang_code(endpoint, values):
14.     g.lang_code = values.pop('lang_code', None)
15.
16. @app.route('/<lang_code>/')
17. def index():
18.     ...
19.
20. @app.route('/<lang_code>/about')
21. def about():
22.     ...

```

国际化的蓝图 URL

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

```

1. from flask import Blueprint, g
2.
3. bp = Blueprint('frontend', __name__, url_prefix='<lang_code>')
4.
5. @bp.url_defaults
6. def add_language_code(endpoint, values):
7.     values.setdefault('lang_code', g.lang_code)
8.
9. @bp.url_value_preprocessor
10. def pull_lang_code(endpoint, values):
11.     g.lang_code = values.pop('lang_code')
12.

```

```
13. @bp.route('/')
14. def index():
15.     ...
16.
17. @bp.route('/about')
18. def about():
19.     ...
```

使用 Setuptools 部署

Setuptools 是一个扩展库，通常用于分发 Python 库和扩展。它扩展了 Python 自带的一个基础模块安装系统 **distutils**，支持多种更复杂的结构，方便了大型应用的分发部署。它的主要特色：

- 支持依赖：一个库或者应用可以声明其所依赖的其他库的列表。依赖库将被自动安装。
- 包注册：可以在安装过程中注册包，这样就可以通过一个包查询其他包的信息。这套系统最有名的功能是“切入点”，即一个包可以定义一个入口，以便于其他包挂接，用以扩展包。
- 安装管理：**pip** 可以为你安装其他库。

如果你从 Python.org 安装了 Python 2 ($\geq 2.7.9$) 或者 Python 3 (≥ 3.4)，那么已经安装好了 **pip** 和 **setuptools**。否则需要自行安装它们。

Flask 本身，以及其他所有在 PyPI 中可以找到的库要么是用 **setuptools** 分发的，要么是用 **distutils** 分发的。

在这里我们假设你的应用名称是 `yourapplication.py`，且没有使用模块，而是一个包。关于如何把模块转换为包的信息参见 [大型应用](#) 方案。

可用的 **setuptools** 部署是进行复杂开发的第一步，它将使发布工作更加自动化。如果你想要完全自动化处理，请同时阅读 [使用 Fabric 部署](#) 一节。

基础设置脚本

因为 Flask 依赖 **setuptools**，所以安装好了 Flask，就表示 **setuptools** 也已经安装好了。

标准声明：[最好使用 virtualenv](#)。

你的设置代码应用放在 `setup.py` 文件中，这个文件应当位于应用旁边。这个文件名只是一个约定，但是最好不要改变，因为大家都会去找这个文件。

Flask 应用的基础 `setup.py` 文件示例如下：

```
1. from setuptools import setup
2.
3. setup(
4.     name='Your Application',
5.     version='1.0',
6.     long_description=__doc__,
7.     packages=['yourapplication'],
8.     include_package_data=True,
9.     zip_safe=False,
10.    install_requires=['Flask']
11. )
```

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

```

1. from setuptools import setup, find_packages
2.
3. setup(
4.     ...
5.     packages=find_packages()
6. )

```

大多数 `setup` 的参数可以望文生义，但是 `include_package_data` 和 `zip_safe` 可能不容易理解。

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

标记构建版本

区分发行版本和开发版本是有益的。添加一个 `setup.cfg` 文件来配置这些选项：

```

1. [egg_info]
2. tag_build = .dev
3. tag_date = 1
4.
5. [aliases]
6. release = egg_info -Db ''

```

运行 `python setup.py sdist` 会创建一个带有 `.dev` 的开发包，并且当前的数据会添加到 `flaskr-1.0.dev20160314.tar.gz` 中。运行 `python setup.py release sdist` 会创建一个发行包 `flaskr-1.0.tar.gz`。只有一个版本。

分发资源

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

```

1. recursive-include yourapplication/templates *
2. recursive-include yourapplication/static *

```

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

声明依赖

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

```
1. install_requires=[
2.     'Flask>=0.2',
3.     'SQLAlchemy>=0.6',
4.     'BrokenPackage>=0.7,<=1.0'
5. ]
```

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

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

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

安装 / 开发

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

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

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

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

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

使用 Fabric 部署

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

在下文开始之前，有几点需要明确：

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

创建第一个 Fabfile

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

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

```
1. from fabric.api import *
2.
3. # 使用远程命令的用户名
4. env.user = 'appuser'
5. # 执行命令的服务器
6. env.hosts = ['server1.example.com', 'server2.example.com']
7.
8. def pack():
9.     # build the package
10.    local('python setup.py sdist --formats=gztar', capture=False)
11.
12. def deploy():
13.     # figure out the package name and version
14.    dist = local('python setup.py --fullname', capture=True).strip()
15.    filename = '%s.tar.gz' % dist
16.
17.    # upload the package to the temporary folder on the server
18.    put('dist/%s' % filename, '/tmp/%s' % filename)
19.
20.    # install the package in the application's virtualenv with pip
21.    run('/var/www/yourapplication/env/bin/pip install /tmp/%s' % filename)
22.
23.    # remove the uploaded package
24.    run('rm -r /tmp/%s' % filename)
25.
26.    # touch the .wsgi file to trigger a reload in mod_wsgi
27.    run('touch /var/www/yourapplication.wsgi')
```


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

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

运行 Fabfile

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

```
1. $ fab pack deploy
```

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

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

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

设置一个新服务器大致有以下几个步骤：

- 在 `:file: /var/www` 创建目录结构：

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

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

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

WSGI 文件

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

```
1. import os
```

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

应用本身必须像下面这样初始化自己才会根据环境变量搜索配置：

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

这个方法在 [配置管理](#) 一节已作了详细的介绍。

配置文件

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

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

不管如何，我们这里只有一到两台服务器，因此我们可以预先手动上传配置文件。

第一次部署

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

```
1. $ fab pack deploy
```

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

下一步

在前文的基础上，还有更多的方法可以全部署工作更加轻松：

- 创建一个初始化新服务器的 `bootstrap` 命令。它可以初始化一个新的 `virtual` 环境、正确设置 `apache` 等等。
- 把配置文件放入一个独立的版本库中，把活动配置的符号链接放在适当的地方。
- 还可以把应用代码放在一个版本库中，在服务器上检出最新版本后安装。这样你可以方便的回滚到老版本。
- 挂接测试功能，方便部署到外部服务器进行测试。

使用 Fabric 是一件有趣的事情。你会发现在电脑上打出 `fab deploy` 是非常神奇的。你可以看到你的应用被部

署到一个又一个服务器上。

使用 SQLite 3

在 Flask 中可以方便地按需打开数据库连接，并在情境结束时（通常是请求结束时）关闭。

下面是一个如何在 Flask 中使用 SQLite 3 的例子：

```
1. import sqlite3
2. from flask import g
3.
4. DATABASE = '/path/to/database.db'
5.
6. def get_db():
7.     db = getattr(g, '_database', None)
8.     if db is None:
9.         db = g._database = sqlite3.connect(DATABASE)
10.    return db
11.
12. @app.teardown_appcontext
13. def close_connection(exception):
14.     db = getattr(g, '_database', None)
15.     if db is not None:
16.         db.close()
```

现在，要使用数据库，应用必须要么有一个活动的应用情境（在存在请求的情况下，总会有一个），要么创建一个应用情境。在这种情况下， `get_db` 函数可以用于获得当前数据库连接。一旦情境灭失，数据库连接就会中断。

注意：如果使用 Flask 0.9 版或者更早版本，需要使用 `flask._app_ctx_stack.top` 代替 `g`，因为 `flask.g` 对象绑定到请求而不是应用情境。

示例：

```
1. @app.route('/')
2. def index():
3.     cur = get_db().cursor()
4.     ...
```

Note

请牢记，拆卸请求（ `teardown request` ）和应用情境（ `appcontext` ）函数总是会执行，即使一个请求前处理器（ `before-request handler` ）失败或者没有执行也是如此。因此，我们在关闭数据库前应当确认数据库已经存在。

按需连接

在第一次使用时连接的好处是只会在真正需要的时候打开连接。如果需要在 一个请求情境之外使用这个代码，可以在 Python shell 中手动打开应用情境后使用：

```

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

```

简化查询

现在每个请求处理函数中可以通过 `get_db()` 来得到当前打开的数据库连接。一个行工厂 (`row factory`) 可以简化 SQLite 的使用，它会在每个结果返回的时候对返回结果进行加工。例如，为了得到字典型而不是元组型的结果，以下内容可以插入到前文的 `get_db` 函数中：

```

1. def make_dicts(cursor, row):
2.     return dict((cursor.description[idx][0], value)
3.                 for idx, value in enumerate(row))
4.
5. db.row_factory = make_dicts

```

这样，`sqlite3` 模块就会返回方便处理的字典类型的结果了。更进一步，我们可以把以下内容放到 `get_db` 中：

```

1. db.row_factory = sqlite3.Row

```

这样查询会返回 `Row` 对象，而不是字典。`Row` 对象是 `namedtuple`，因此既可以通过索引访问也可以通过键访问。例如，假设我们有一个 `sqlite3.Row` 名为 `r`，记录包含 `id`、`FirstName`、`LastName` 和 `MiddleInitial` 字段：

```

1. >>> # 基于键的名称取值
2. >>> r['FirstName']
3. John
4. >>> # 或者基于索引取值
5. >>> r[1]
6. John
7. # Row 对象是可迭代的：
8. >>> for value in r:
9.     ...     print(value)
10. 1
11. John
12. Doe
13. M

```

另外，提供一个函数，用于获得游标、执行查询和获取结果是一个好主意：

```

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

```

这个方便称手的小函数与行工厂联合使用比使用原始的数据库游标和连接对象要方便多了。

使用该函数示例：

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

只需要得到单一结果的用法：

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

如果要给 SQL 语句传递参数，请在语句中使用问号来代替参数，并把参数放在一个列表中一起传递。不要用字符串格式化的方式直接把参数加入 SQL 语句中，这样会给应用带来 [SQL 注入](#) 的风险。

初始化模式

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

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

接下来可以在 Python shell 中创建数据库：

```
1. >>> from yourapplication import init_db
2. >>> init_db()
```

使用 SQLAlchemy

许多人喜欢使用 [SQLAlchemy](#) 来访问数据库。建议在你的 Flask 应用中使用包来代替模块，并把模型放入一个独立的模块中（参见 [大型应用](#)）。虽然这不是必须的，但是很有用。

有四种 SQLAlchemy 的常用方法，下面一一道来：

Flask-SQLAlchemy 扩展

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

你可以从 [PyPI](#) 下载 [Flask-SQLAlchemy](#)。

声明

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

以下是示例 `database.py` 模块：

```
1. from sqlalchemy import create_engine
2. from sqlalchemy.orm import scoped_session, sessionmaker
3. from sqlalchemy.ext.declarative import declarative_base
4.
5. engine = create_engine('sqlite:///tmp/test.db', convert_unicode=True)
6. db_session = scoped_session(sessionmaker(autocommit=False,
7.                                           autoflush=False,
8.                                           bind=engine))
9. Base = declarative_base()
10. Base.query = db_session.query_property()
11.
12. def init_db():
13.     # 在这里导入定义模型所需要的所有模块，这样它们就会正确的注册在
14.     # 元数据上。否则你就必须在调用 init_db() 之前导入它们。
15.     import yourapplication.models
16.     Base.metadata.create_all(bind=engine)
```

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

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

```
1. from yourapplication.database import db_session
```

```

2.
3. @app.teardown_appcontext
4. def shutdown_session(exception=None):
5.     db_session.remove()

```

以下是一个示例模型（放入 `models.py` 中）：

```

1. from sqlalchemy import Column, Integer, String
2. from yourapplication.database import Base
3.
4. class User(Base):
5.     __tablename__ = 'users'
6.     id = Column(Integer, primary_key=True)
7.     name = Column(String(50), unique=True)
8.     email = Column(String(120), unique=True)
9.
10.    def __init__(self, name=None, email=None):
11.        self.name = name
12.        self.email = email
13.
14.    def __repr__(self):
15.        return '<User %r>' % (self.name)

```

可以使用 `init_db` 函数来创建数据库：

```

1. >>> from yourapplication.database import init_db
2. >>> init_db()

```

在数据库中插入条目示例：

```

1. >>> from yourapplication.database import db_session
2. >>> from yourapplication.models import User
3. >>> u = User('admin', 'admin@localhost')
4. >>> db_session.add(u)
5. >>> db_session.commit()

```

查询很简单：

```

1. >>> User.query.all()
2. [<User u'admin'>]
3. >>> User.query.filter(User.name == 'admin').first()
4. <User u'admin'>

```

人工对象关系映射

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

示例 `database.py` 模块：

```
1. from sqlalchemy import create_engine, MetaData
2. from sqlalchemy.orm import scoped_session, sessionmaker
3.
4. engine = create_engine('sqlite:///tmp/test.db', convert_unicode=True)
5. metadata = MetaData()
6. db_session = scoped_session(sessionmaker(autocommit=False,
7.                                           autoflush=False,
8.                                           bind=engine))
9. def init_db():
10.     metadata.create_all(bind=engine)
```

就像声明方法一样，你需要在每个请求结束后或者应用情境关闭后关闭会话。把以下代码放入你的应用模块：

```
1. from yourapplication.database import db_session
2.
3. @app.teardown_appcontext
4. def shutdown_session(exception=None):
5.     db_session.remove()
```

以下是一个示例表和模型（放入 `models.py` 中）：

```
1. from sqlalchemy import Table, Column, Integer, String
2. from sqlalchemy.orm import mapper
3. from yourapplication.database import metadata, db_session
4.
5. class User(object):
6.     query = db_session.query_property()
7.
8.     def __init__(self, name=None, email=None):
9.         self.name = name
10.        self.email = email
11.
12.    def __repr__(self):
13.        return '<User %r>' % (self.name)
14.
15. users = Table('users', metadata,
16.               Column('id', Integer, primary_key=True),
17.               Column('name', String(50), unique=True),
18.               Column('email', String(120), unique=True)
19.               )
20. mapper(User, users)
```

查询和插入与声明方式的一样。

SQL 抽象层

如果你只需要使用数据库系统（和 SQL）抽象层，那么基本上只要使用引擎：

```

1. from sqlalchemy import create_engine, MetaData, Table
2.
3. engine = create_engine('sqlite:///tmp/test.db', convert_unicode=True)
4. metadata = MetaData(bind=engine)

```

然后你要么像前文中一样在代码中声明表，要么自动载入它们：

```

1. from sqlalchemy import Table
2.
3. users = Table('users', metadata, autoload=True)

```

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

```

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

```

SQLAlchemy 会自动提交。

可以直接使用引擎或连接来查询数据库：

```

1. >>> users.select(users.c.id == 1).execute().first()
2. (1, u'admin', u'admin@localhost')

```

查询结果也是类字典元组：

```

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

```

你也可以把 SQL 语句作为字符串传递给 `execute()` 方法：

```

1. >>> engine.execute('select * from users where id = :1', [1]).first()
2. (1, u'admin', u'admin@localhost')

```

关于 SQLAlchemy 的更多信息请移步其[官方网站](#)。

上传文件

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

- 一个带有 `enctype=multipart/form-data` 的 `<form>` 标记，标记中含有一个 `<input type=file>` 。
- 应用通过请求对象的 `files` 字典来访问文件。
- 使用文件的 `save()` 方法把文件永久地保存在文件系统中。

简介

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

```
1. import os
2. from flask import Flask, flash, request, redirect, url_for
3. from werkzeug.utils import secure_filename
4.
5. UPLOAD_FOLDER = '/path/to/the/uploads'
6. ALLOWED_EXTENSIONS = {'txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif'}
7.
8. app = Flask(__name__)
9. app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
```

首先我们导入了一堆东西，大多数是浅显易懂的。`werkzeug.secure_filename()` 会在稍后解释。`UPLOAD_FOLDER` 是上传文件要储存的目录，`ALLOWED_EXTENSIONS` 是允许上传的文件扩展名的集合。

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

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

```
1. def allowed_file(filename):
2.     return '.' in filename and \
3.         filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
4.
5. @app.route('/', methods=['GET', 'POST'])
6. def upload_file():
7.     if request.method == 'POST':
8.         # check if the post request has the file part
9.         if 'file' not in request.files:
10.             flash('No file part')
11.             return redirect(request.url)
12.         file = request.files['file']
13.         # if user does not select file, browser also
14.         # submit an empty part without filename
15.         if file.filename == '':
```

```

16.         flash('No selected file')
17.         return redirect(request.url)
18.     if file and allowed_file(file.filename):
19.         filename = secure_filename(file.filename)
20.         file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
21.         return redirect(url_for('uploaded_file',
22.                                 filename=filename))
23.     return ''
24.     <!doctype html>
25.     <title>Upload new File</title>
26.     <h1>Upload new File</h1>
27.     <form method=post enctype=multipart/form-data>
28.         <input type=file name=file>
29.         <input type=submit value=Upload>
30.     </form>
31.     '''

```

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

进一步说明

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

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

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

现在来看看函数是如何工作的：

```

1. >>> secure_filename('../../../home/username/.bashrc')
2. 'home_username_.bashrc'

```

现在还剩下一件事：为已上传的文件提供服务。在 `upload_file()` 中，我们把用户重定向到 `url_for('uploaded_file', filename=filename)`，即 `/uploads/filename`。因此我们写一个 `uploaded_file()` 来返回该文件名称。 Flask 0.5 版本开始我们可以使用一个函数来完成这个任务：

```

1. from flask import send_from_directory
2.
3. @app.route('/uploads/<filename>')
4. def uploaded_file(filename):
5.     return send_from_directory(app.config['UPLOAD_FOLDER'],
6.                               filename)

```

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

```
1. from werkzeug import SharedDataMiddleware
2. app.add_url_rule('/uploads/<filename>', 'uploaded_file',
3.                 build_only=True)
4. app.wsgi_app = SharedDataMiddleware(app.wsgi_app, {
5.     '/uploads': app.config['UPLOAD_FOLDER']
6. })
```

如果你现在运行应用，那么应该一切都应该按预期正常工作。

改进上传

Changelog

New in version 0.6.

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

```
1. from flask import Flask, Request
2.
3. app = Flask(__name__)
4. app.config['MAX_CONTENT_LENGTH'] = 16 * 1024 * 1024
```

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

连接重置问题

当使用本地开发服务器时，可能会得到一个连接重置，而不是一个 413 响应。在生产 WSGI 服务器上运行应用时会得到正确的响应。

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

上传进度条

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

一个更简便的方案

现在有了更好的解决方案，更快且更可靠。像 `jQuery` 之类的 JavaScript 库包含成的轻松构建进度条的插件。

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

缓存

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

Flask 本身不提供缓存，但是 [Flask-Caching](#) 扩展可以。Flask-Caching 支持多种后端，甚至可以支持你自己开发的后端。

视图装饰器

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

检查登录装饰器

让我们来实现这个装饰器。装饰器是一个包装并替换另一个函数的函数。既然源函数已经被替代，就需要记住：要复制源函数的信息到新函数中。可以用 `functools.wraps()` 处理这个事情。

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

```
1. from functools import wraps
2. from flask import g, request, redirect, url_for
3.
4. def login_required(f):
5.     @wraps(f)
6.     def decorated_function(*args, **kwargs):
7.         if g.user is None:
8.             return redirect(url_for('login', next=request.url))
9.         return f(*args, **kwargs)
10.    return decorated_function
```

为了使用这个装饰器呢，需要把这个装饰器放在最靠近函数的地方。当使用更进一步的装饰器时，请记住要把

`route()` 装饰器放在最外面：

```
1. @app.route('/secret_page')
2. @login_required
3. def secret_page():
4.     pass
```

Note

登录页面在一个 `GET` 请求之后 `next` 值会存在于 `request.args` 之中。当从登录表单发送 `POST` 请求时必须一起传递它。可以使用一个隐藏标记来做到这点，然后当用户登录时，从 `request.form` 获取它。

```
1. <input type="hidden" value="{ request.args.get('next', '') }"/>
```

缓存装饰器

假设有一个视图函数需要消耗昂贵的计算成本，因此你需要在一定时间内缓存这个视图的计算结果。这种情况下装饰

器是一个好的选择。我们假设你像[缓存](#) 方案中一样设置了缓存。

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

被装饰代码按如下步骤工作

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

代码：

```
1. from functools import wraps
2. from flask import request
3.
4. def cached(timeout=5 * 60, key='view/%s'):
5.     def decorator(f):
6.         @wraps(f)
7.         def decorated_function(*args, **kwargs):
8.             cache_key = key % request.path
9.             rv = cache.get(cache_key)
10.            if rv is not None:
11.                return rv
12.            rv = f(*args, **kwargs)
13.            cache.set(cache_key, rv, timeout=timeout)
14.            return rv
15.        return decorated_function
16.    return decorator
```

注意，以上代码假设存在一个可用的实例化的 `cache` 对象，更多信息参见[缓存](#) 方案。

模板装饰器

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

```
1. @app.route('/')
2. def index():
3.     return render_template('index.html', value=42)
4.
5. @app.route('/')
6. @templated('index.html')
7. def index():
8.     return dict(value=42)
9.
10. @app.route('/')
```



```

11. @templated()
12. def index():
13.     return dict(value=42)

```

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

以下是装饰器的代码：

```

1. from functools import wraps
2. from flask import request, render_template
3.
4. def templated(template=None):
5.     def decorator(f):
6.         @wraps(f)
7.         def decorated_function(*args, **kwargs):
8.             template_name = template
9.             if template_name is None:
10.                 template_name = request.endpoint \
11.                     .replace('.', '/') + '.html'
12.             ctx = f(*args, **kwargs)
13.             if ctx is None:
14.                 ctx = {}
15.             elif not isinstance(ctx, dict):
16.                 return ctx
17.             return render_template(template_name, **ctx)
18.         return decorated_function
19.     return decorator

```

端点装饰器

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

```

1. from flask import Flask
2. from werkzeug.routing import Rule
3.
4. app = Flask(__name__)
5. app.url_map.add(Rule('/', endpoint='index'))
6.
7. @app.endpoint('index')
8. def my_index():
9.     return "Hello world"

```

使用 WTForms 进行表单验证

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

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

使用一个扩展获得大部分 WTForms 的功能

[Flask-WTF](#) 扩展可以实现本方案的所有功能，并且还提供一些辅助小工具。使用这个扩展可以更好的使用表单和 Flask。你可以从 [PyPI](#) 获得这个扩展。

表单

下面是一个典型的注册页面的示例：

```
1. from wtforms import Form, BooleanField, StringField, PasswordField, validators
2.
3. class RegistrationForm(Form):
4.     username = StringField('Username', [validators.Length(min=4, max=25)])
5.     email = StringField('Email Address', [validators.Length(min=6, max=35)])
6.     password = PasswordField('New Password', [
7.         validators.DataRequired(),
8.         validators.EqualTo('confirm', message='Passwords must match')
9.     ])
10.    confirm = PasswordField('Repeat Password')
11.    accept_tos = BooleanField('I accept the TOS', [validators.DataRequired()])
```

视图

在视图函数中，表单用法示例如下：

```
1. @app.route('/register', methods=['GET', 'POST'])
2. def register():
3.     form = RegistrationForm(request.form)
4.     if request.method == 'POST' and form.validate():
5.         user = User(form.username.data, form.email.data,
6.                     form.password.data)
7.         db_session.add(user)
8.         flash('Thanks for registering')
9.         return redirect(url_for('login'))
10.    return render_template('register.html', form=form)
```

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

请记住以下几点：

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

模板中的表单

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

以下是一个使用宏的示例 `_formhelpers.html` 模板：

```
1. {% macro render_field(field) %}
2.     <dt>{{ field.label }}
3.     <dd>{{ field(**kwargs)|safe }}
4.     {% if field.errors %}
5.         <ul class=errors>
6.             {% for error in field.errors %}
7.                 <li>{{ error }}</li>
8.             {% endfor %}
9.         </ul>
10.    {% endif %}
11.    </dd>
12. {% endmacro %}
```

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

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

```
1. {% from "_formhelpers.html" import render_field %}
2. <form method=post>
3.     <dl>
4.         {{ render_field(form.username) }}
5.         {{ render_field(form.email) }}
6.         {{ render_field(form.password) }}
7.         {{ render_field(form.confirm) }}
8.         {{ render_field(form.accept_tos) }}
9.     </dl>
10.    <p><input type=submit value=Register>
11. </form>
```

更多关于 WTForms 的信息请移步 [WTForms 官方网站](#)。

模板继承

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

听起来很复杂其实做起来简单，看看下面的例子就容易理解了。

基础模板

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

```
1. <!doctype html>
2. <html>
3.   <head>
4.     {% block head %}
5.     <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
6.     <title>{% block title %}{% endblock %} - My Webpage</title>
7.     {% endblock %}
8.   </head>
9.   <body>
10.    <div id="content">{% block content %}{% endblock %}</div>
11.    <div id="footer">
12.      {% block footer %}
13.      &copy; Copyright 2010 by <a href="http://domain.invalid/">you</a>.
14.      {% endblock %}
15.    </div>
16.  </body>
17. </html>
```

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

子模板

子模板示例：

```
1. {% extends "layout.html" %}
2. {% block title %}Index{% endblock %}
3. {% block head %}
4.   {{ super() }}
5.   <style type="text/css">
6.     .important { color: #336699; }
7.   </style>
8. {% endblock %}
9. {% block content %}
10.   <h1>Index</h1>
```

```
11. <p class="important">
12.     Welcome on my awesome homepage.
13. {% endblock %}
```

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

消息闪现

一个好的应用和用户界面都需要良好的反馈。如果用户得不到足够的反馈，那么应用最终会被用户唾弃。 Flask 的闪现系统提供了一个良好的反馈方式。闪现系统的基本工作方式是：在且只在下一个请求中访问上一个请求结束时记录的消息。一般我们结合布局模板来使用闪现系统。注意，浏览器会限制 cookie 的大小，有时候网络服务器也会。这样如果消息比会话 cookie 大的话，那么会导致消息闪现静默失败。

简单的例子

以下是一个完整的示例：

```
1. from flask import Flask, flash, redirect, render_template, \
2.     request, url_for
3.
4. app = Flask(__name__)
5. app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'
6.
7. @app.route('/')
8. def index():
9.     return render_template('index.html')
10.
11. @app.route('/login', methods=['GET', 'POST'])
12. def login():
13.     error = None
14.     if request.method == 'POST':
15.         if request.form['username'] != 'admin' or \
16.            request.form['password'] != 'secret':
17.             error = 'Invalid credentials'
18.         else:
19.             flash('You were successfully logged in')
20.             return redirect(url_for('index'))
21.     return render_template('login.html', error=error)
```

以下是实现闪现的 `layout.html` 模板：

```
1. <!doctype html>
2. <title>My Application</title>
3. {% with messages = get_flashed_messages() %}
4.     {% if messages %}
5.         <ul class=flashes>
6.             {% for message in messages %}
7.                 <li>{{ message }}</li>
8.             {% endfor %}
9.         </ul>
10.    {% endif %}
11. {% endwith %}
12. {% block body %}{% endblock %}
```

以下是继承自 `layout.html` 的 `index.html` 模板：

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

以下是同样继承自 `layout.html` 的 `login.html` 模板：

```
1. {% extends "layout.html" %}
2. {% block body %}
3.     <h1>Login</h1>
4.     {% if error %}
5.         <p class=error><strong>Error:</strong> {{ error }}
6.     {% endif %}
7.     <form method=post>
8.         <dl>
9.             <dt>Username:
10.            <dd><input type=text name=username value="{{
11.                request.form.username }}">
12.            <dt>Password:
13.            <dd><input type=password name=password>
14.        </dl>
15.        <p><input type=submit value=Login>
16.    </form>
17. {% endblock %}
```

闪现消息的类别

Changelog

New in version 0.3.

闪现消息还可以指定类别，如果没有指定，那么缺省的类别为 `'message'`。不同的类别可以给用户提供更好的反馈。例如错误消息可以使用红色背景。

使用 `flash()` 函数可以指定消息的类别：

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

模板中的 `get_flashed_messages()` 函数也应当返回类别，显示消息的循环也要略作改变：

```
1. {% with messages = get_flashed_messages(with_categories=true) %}
2.     {% if messages %}
3.         <ul class=flashes>
4.             {% for category, message in messages %}
5.                 <li class="{{ category }}">{{ message }}</li>
6.             {% endfor %}
```

```
7.     </ul>
8.     {% endif %}
9. {% endwith %}
```

上例展示如何根据类别渲染消息，还可以给消息加上前缀，如 `Error:` 。

过滤闪现消息

Changelog

New in version 0.9.

你可以视情况通过传递一个类别列表来过滤 `get_flashed_messages()` 的结果。这个功能有助于在不同位置显示不同类别的消息。

```
1. {% with errors = get_flashed_messages(category_filter=["error"]) %}
2. {% if errors %}
3. <div class="alert-message block-message error">
4.   <a class="close" href="#">x</a>
5.   <ul>
6.     {%- for msg in errors %}
7.     <li>{{ msg }}</li>
8.     {% endfor -%}
9.   </ul>
10. </div>
11. {% endif %}
12. {% endwith %}
```


通过 jQuery 使用 AJAX

jQuery 是一个小型的 JavaScript 库，通常用于简化 DOM 和 JavaScript 的使用。它是一个非常好的工具，可以通过在服务端和客户端交换 JSON 来使网络应用更具有动态性。

JSON 是一种非常轻巧的传输格式，非常类似于 Python 原语（数字、字符串、字典和列表）。JSON 被广泛支持，易于解析。JSON 在几年之前开始流行，在网络应用中迅速取代了 XML。

载入 jQuery

为了使用 jQuery，你必须先把它下载下来，放在应用的静态目录中，并确保它被载入。理想情况下你有一个用于所有页面的布局模板。在这个模板的 `<body>` 的底部添加一个 `script` 语句来载入 jQuery：

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

另一个方法是使用 Google 的 [AJAX 库 API](#) 来载入 jQuery：

```
1. <script src="//ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>
2. <script>window.jQuery || document.write('<script src="{{
3.   url_for('static', filename='jquery.js') }}">\x3C/script>')</script>
```

在这种方式中，应用会先尝试从 Google 下载 jQuery，如果失败则会调用静态目录中的备用 jQuery。这样做的好处是如果用户已经去过使用与 Google 相同版本的 jQuery 的网站后，访问你的网站时，页面可能会更快地载入，因为浏览器已经缓存了 jQuery。

我的网站在哪里？

我的网站在哪里？如果你的应用还在开发中，那么答案很简单：它在本机的某个端口上，且在服务器的根路径下。但是如果以后要把应用移到其他位置（例如 `http://example.com/myapp`）上呢？在服务端，这个问题不成为问题，可以使用 `url_for()` 函数来得到答案。但是如果使用 jQuery，那么就不能硬码应用的路径，只能使用动态路径。怎么办？

一个简单的方法是在页面中添加一个 `script` 标记，设置一个全局变量来表示应用的根路径。示例：

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

在 Flask 0.10 版本以前版本中，使用 `|safe` 是有必要的，是为了使 Jinja 不要转义 JSON 编码的字符串。通常这样做不是必须的，但是在 `script` 内部我们需要这么做。

进一步说明

在 HTML 中，`script` 标记是用于声明 `CDATA` 的，也就是说声明的内容不会被解析。`<script>` 与 `</script>` 之间的内容都会被作为脚本处理。这也意味着在 `script` 标记之间不会存在任何 `</`。在这里 `|tojson` 会正确处理问题，并为你转义斜杠（`{{ "</script>"|tojson|safe }}` 会被渲染为 `"<\script>"`）。

在 Flask 0.10 版本中更进了一步，把所有 HTML 标记都用 unicode 转义了，这样使 Flask 自动把 HTML 转换为安全标记。

JSON 视图函数

现在让我们来创建一个服务端函数，这个函数接收两个 URL 参数（两个需要相加的数字），然后向应用返回一个 JSON 对象。下面这个例子是非常不实用的，因为一般会在客户端完成类似工作，但这个例子可以简单明了地展示如何使用 jQuery 和 Flask：

```
1. from flask import Flask, jsonify, render_template, request
2. app = Flask(__name__)
3.
4. @app.route('/_add_numbers')
5. def add_numbers():
6.     a = request.args.get('a', 0, type=int)
7.     b = request.args.get('b', 0, type=int)
8.     return jsonify(result=a + b)
9.
10. @app.route('/')
11. def index():
12.     return render_template('index.html')
```

正如你所见，我还添加了一个 `index` 方法来渲染模板。这个模板会按前文所述载入 jQuery。模板中有一个用于两个数字相加的表单和一个触发服务端函数的链接。

注意，这里我们使用了 `get()` 方法。它不会调用失败。如果字典的键不存在，就会返回一个缺省值（这里是 `0`）。更进一步它还会把值转换为指定的格式（这里是 `int`）。在脚本（API、JavaScript 等）触发的代码中使用它特别方便，因为在这种情况下不需要特殊的错误报告。

HTML

你的 `index.html` 模板要么继承一个已经载入 jQuery 和设置好 `$SCRIPT_ROOT` 变量的 `layout.html` 模板，要么在模板开头就做好那两件事。下面就是应用的 HTML 示例（`index.html`）。注意，我们把脚本直接放入了 HTML 中。通常更好的方式是放在独立的脚本文件中：

```
1. <script type=text/javascript>
2. $(function() {
3.     $('#calculate').bind('click', function() {
4.         $.getJSON($SCRIPT_ROOT + '/_add_numbers', {
5.             a: $('#input[name="a"]').val(),
6.             b: $('#input[name="b"]').val()
7.         }, function(data) {
8.             $('#result').text(data.result);
```

```

9.         });
10.        return false;
11.    });
12. });
13. </script>
14. <h1>jQuery Example</h1>
15. <p><input type=text size=5 name=a> +
16.    <input type=text size=5 name=b> =
17.    <span id=result>?</span>
18. <p><a href=# id=calculate>calculate server side</a>

```

这里不讲述 jQuery 运行详细情况，仅对上例作一个简单说明：

- `$(function() { ... })` 定义浏览器在页面的基本部分载入完成后立即执行的代码。
- `$('#selector')` 选择一个元素供你操作。
- `element.bind('event', func)` 定义一个用户点击元素时运行的函数。如果函数返回 `false`，那么缺省行为就不会起作用（本例为转向 `# URL`）。
- `$.getJSON(url, data, func)` 向 `url` 发送一个 `GET` 请求，并把 `data` 对象的内容作为查询参数。一旦有数据返回，它将调用指定的函数，并把返回值作为函数的参数。注意，我们可以在这里使用先前定义的 `$SCRIPT_ROOT` 变量。

本页的完整代码可以在 [示例源代码](#) 下载。使用 `XMLHttpRequest` 和 `fetch` 同样。

自定义出错页面

Flask 有一个方便的 `abort()` 函数，它可以通过一个 HTTP 出错代码退出一个请求。它还提供一个包含基本说明的出错页面，页面显示黑白的文本，很朴素。

用户可以根据错误代码或多或少知道发生了什么错误。

常见出错代码

以下出错代码是用户常见的，即使应用正常也会出现这些出错代码：

- *404 Not Found*
- 这是一个古老的“朋友，你使用了一个错误的 URL ”信息。这个信息出现得如此频繁，以至于连刚上网的新手都知道 404 代表：该死的，我要看的东西不见了。一个好的做法是确保 404 页面上有一些真正有用的东西，至少要有个返回首页的链接。
- *403 Forbidden*
- 如果你的网站上有某种权限控制，那么当用户访问未获授权内容时应当发送403 代码。因此请确保当用户尝试访问未获授权内容时得到正确的反馈。
- *410 Gone*
- 你知道 “404 Not Found” 有一个名叫 “410 Gone” 的兄弟吗？很少有人使用这个代码。如果资源以前曾经存在过，但是现在已经被删除了，那么就应该使用410 代码，而不是 404 。如果你不是在数据库中把文档永久地删除，而只是给文档打了一个删除标记，那么请为用户考虑，应当使用 410 代码，并显示信息告知用户要找的东西已经删除。
- *500 Internal Server Error*
- 这个代码通常表示程序出错或服务器过载。强烈建议把这个页面弄得友好一点，因为你的应用 迟早 会出现故障的（参见 [应用错误处理](#) ）。

出错处理器

一个出错处理器是一个返回响应的函数，就像一个视图函数类似。不同之处在于出错处理器在某类错误引发时作出响应，而视图在发生 URL 匹配的请求时作出响应。出错处理器传递错误实例，大多数情况是一个 `HTTPException` 。但是处理“ 500 Internal Server Error ”的处理器除了 500 错误外还会传递未捕获的异常。

出错处理器使用 `errorhandler()` 装饰器或者 `register_error_handler()` 方法注册。出错处理器可以为一个状态码（如 404 ）注册，也可以为一个异常类注册。

响应的代码不会被设置为处理器的代码。因此请确保在从处理器返回响应时，提供恰当的 HTTP 状态代码。

“500 Internal Server Error” 的出错处理器在调试模式下不会启用，而会显示交互调试器。

以下是一个处理 “404 Page Not Found” 异常的示例：

```
1. from flask import render_template
2.
3. @app.errorhandler(404)
4. def page_not_found(e):
5.     # note that we set the 404 status explicitly
6.     return render_template('404.html'), 404
```

使用 [工厂模式](#) 的话：

```
1. from flask import Flask, render_template
2.
3. def page_not_found(e):
4.     return render_template('404.html'), 404
5.
6. def create_app(config_filename):
7.     app = Flask(__name__)
8.     app.register_error_handler(404, page_not_found)
9.     return app
```

示例模板：

```
1. {% extends "layout.html" %}
2. {% block title %}Page Not Found{% endblock %}
3. {% block body %}
4.     <h1>Page Not Found</h1>
5.     <p>What you were looking for is just not there.
6.     <p><a href="{{ url_for('index') }}">go somewhere nice</a>
7. {% endblock %}
```

以 JSON 格式返回 API 错误

当把 Flask 用于 网络 API 的时候，可以使用上述同样的技术，以 JSON 格式返回错误。调用 `abort()` 时使用 `description` 参数，`errorhandler()` 会把它作为 JSON 格式的错误信息，并设置状态码为 404 。

```
1. from flask import abort, jsonify
2.
3. @app.errorhandler(404)
4. def resource_not_found(e):
5.     return jsonify(error=str(e)), 404
6.
7. @app.route("/cheese")
8. def get_one_cheese():
9.     resource = get_resource()
10.
11.     if resource is None:
12.         abort(404, description="Resource not found")
13.
```

```
14.     return jsonify(resource)
```

惰性载入视图

Flask 通常使用装饰器。装饰器简单易用，只要把 URL 放在相应的函数的前面就可以了。但是这种方式有一个缺点：使用装饰器的代码必须预先导入，否则 Flask 就无法真正找到你的函数。

当你必须快速导入应用时，这就会成为一个问题。在 Google App Engine 或其他系统中，必须快速导入应用。因此，如果你的应用存在这个问题，那么必须使用集中URL 映射。

`add_url_rule()` 函数用于集中 URL 映射，与使用装饰器不同的是你需要一个设置应用所有 URL 的专门文件。

转换为集中 URL 映射

假设有如下应用：

```
1. from flask import Flask
2. app = Flask(__name__)
3.
4. @app.route('/')
5. def index():
6.     pass
7.
8. @app.route('/user/<username>')
9. def user(username):
10.    pass
```

为了集中映射，我们创建一个不使用装饰器的文件（ `views.py` ）：

```
1. def index():
2.     pass
3.
4. def user(username):
5.     pass
```

在另一个文件中集中映射函数与 URL：

```
1. from flask import Flask
2. from yourapplication import views
3. app = Flask(__name__)
4. app.add_url_rule('/', view_func=views.index)
5. app.add_url_rule('/user/<username>', view_func=views.user)
```

延迟载入

至此，我们只是把视图与路由分离，但是模块还是预先载入了。理想的方式是按需载入视图。下面我们使用一个类似函数的辅助类来实现按需载入：

```

1. from werkzeug import import_string, cached_property
2.
3. class LazyView(object):
4.
5.     def __init__(self, import_name):
6.         self.__module__, self.__name__ = import_name.rsplit('.', 1)
7.         self.import_name = import_name
8.
9.     @cached_property
10.    def view(self):
11.        return import_string(self.import_name)
12.
13.    def __call__(self, *args, **kwargs):
14.        return self.view(*args, **kwargs)

```

上例中最重要的是正确设置 *module* 和 *name*，它被用于在不提供 URL 规则的情况下正确命名 URL 规则。

然后可以这样集中定义 URL 规则：

```

1. from flask import Flask
2. from yourapplication.helpers import LazyView
3. app = Flask(__name__)
4. app.add_url_rule('/',
5.                 view_func=LazyView('yourapplication.views.index'))
6. app.add_url_rule('/user/<username>',
7.                 view_func=LazyView('yourapplication.views.user'))

```

还可以进一步优化代码：写一个函数调用 `add_url_rule()`，加上应用前缀和点符号。：

```

1. def url(import_name, url_rules=[], **options):
2.     view = LazyView('yourapplication.' + import_name)
3.     for url_rule in url_rules:
4.         app.add_url_rule(url_rule, view_func=view, **options)
5.
6. # add a single route to the index view
7. url('views.index', ['/'])
8.
9. # add two routes to a single function endpoint
10. url_rules = ['/user/', '/user/<username>']
11. url('views.user', url_rules)

```

有一件事情要牢记：请求前和请求后处理器必须在第一个请求前导入。

其余的装饰器可以同样用上述方法改写。

通过 MongoEngine 使用 MongoDB

使用一个 MongoDB 之类的文档型数据库来代替关系 SQL 数据是很常见的。本方案演示如何使用文档映射库 [MongoEngine](#) 来集成 MongoDB 。

先准备好一个运行中的 MongoDB 服务和 [Flask-MongoEngine](#)

```
1. pip install flask-mongoengine
```

配置

基本的配置是在 `app.config` 中定义 `MONGODB_SETTINGS` 并创建一个 `MongoEngine` 实例：

```
1. from flask import Flask
2. from flask_mongoengine import MongoEngine
3.
4. app = Flask(__name__)
5. app.config['MONGODB_SETTINGS'] = {
6.     "db": "myapp",
7. }
8. db = MongoEngine(app)
```

映射文档

声明用于一个 Mongo 文档的模型的方法是创建一个 `Document` 的子类，然后声明每个字段：

```
1. import mongoengine as me
2.
3. class Movie(me.Document):
4.     title = me.StringField(required=True)
5.     year = me.IntField()
6.     rated = me.StringField()
7.     director = me.StringField()
8.     actors = me.ListField()
```

如果文档包含嵌套的字段，那么使用 `EmbeddedDocument` 来定义嵌套的文档，并在父文档中使用 `EmbeddedDocumentField` 声明相应的字段：

```
1. class Imdb(me.EmbeddedDocument):
2.     imdb_id = me.StringField()
3.     rating = me.DecimalField()
4.     votes = me.IntField()
5.
6. class Movie(me.Document):
7.     ...
```

```
8.         imdb = me.EmbeddedDocumentField(Imdb)
```

创建数据

使用字段的关键字参数实例化文档类。还可以在实例化后为字段属性指定值。然后调用

```
doc.save ()
```

```
1. bttf = Movie(title="Back To The Future", year=1985)
2. bttf.actors = [
3.     "Michael J. Fox",
4.     "Christopher Lloyd"
5. ]
6. bttf.imdb = Imdb(imdb_id="tt0088763", rating=8.5)
7. bttf.save()
```

查询

使用类的 `objects` 属性来执行查询。关键字参数用于字段的等值查询：

```
1. bttf = Movies.objects(title="Back To The Future").get_or_404()
```

字段名称后加双下划线可以连接查询操作符。 `objects` 及其返回的查询是可迭代的：

```
1. some_theron_movie = Movie.objects(actors__in=["Charlize Theron"]).first()
2.
3. for recents in Movie.objects(year__gte=2017):
4.     print(recents.title)
```

相关文档

有许多关于使用 MongoEngine 定义和查询文档数据的方法，更多信息请参阅其[官方文档](#)。

Flask-MongoEngine 为 MongoEngine 添加了有用的工具，请参阅其[文档说明](#)。

添加一个页面图标

一个“页面图标”是浏览器在标签或书签中使用的图标，它可以给你的网站加上一个唯一的标示，方便区别于其他网站。

那么如何给一个 Flask 应用添加一个页面图标呢？首先，显而易见的，需要一个图标。图标应当是 16 X 16 像素的 ICO 格式文件。这不是规定的，但却是一个所有浏览器都支持的事实上的标准。把 ICO 文件命名为

`favicon.ico` 并放入静态文件目录中。

现在我们要让浏览器能够找到你的图标，正确的做法是在你的 HTML 中添加一个链接。示例：

```
1. <link rel="shortcut icon" href="{{ url_for('static', filename='favicon.ico') }}">
```

对于大多数浏览器来说，这样就完成任务了，但是一些老古董不支持这个标准。老的标准是把名为“favicon.ico”的图标放在服务器的根目录下。如果你的应用不是挂接在域的根目录下，那么你需要定义网页服务器在根目录下提供这个图标，否则就无计可施了。如果你的应用位于根目录下，那么你可以简单地进行重定向：

```
1. app.add_url_rule('/favicon.ico',  
2.                  redirect_to=url_for('static', filename='favicon.ico'))
```

如果想要保存额外的重定向请求，那么还可以使用 `send_from_directory()` 函数来写一个视图：

```
1. import os  
2. from flask import send_from_directory  
3.  
4. @app.route('/favicon.ico')  
5. def favicon():  
6.     return send_from_directory(os.path.join(app.root_path, 'static'),  
7.                               'favicon.ico', mimetype='image/vnd.microsoft.icon')
```

上例中的 MIME 类型可以省略，浏览器会自动猜测类型。但是我们在例子中明确定义了，省去了额外的猜测，反正这个类型是不变的。

上例会通过你的应用来提供图标，如果可能的话，最好配置你的专用服务器来提供图标，配置方法参见网页服务器的文档。

另见

- Wikipedia 上的 [页面图标](#) 词条

流内容

有时候你会需要把大量数据传送到客户端，不在内存中保存这些数据。当你想把运行中产生的数据不经过文件系统，而是直接发送给客户端时，应当怎么做呢？

答案是使用生成器和直接响应。

基本用法

下面是一个在运行中产生大量 CSV 数据的基本视图函数。其技巧是调用一个内联函数生成数据，把这个函数传递给一个响应对象：

```
1. from flask import Response
2.
3. @app.route('/large.csv')
4. def generate_large_csv():
5.     def generate():
6.         for row in iter_all_rows():
7.             yield ','.join(row) + '\n'
8.     return Response(generate(), mimetype='text/csv')
```

每个 `yield` 表达式被直接传送给浏览器。注意，有一些 WSGI 中间件可能会打断流内容，因此在使用分析器或者其他工具的调试环境中要小心一些。

模板中的流内容

Jinja2 模板引擎也支持分片渲染模板。这个功能不是直接被 Flask 支持的，因为它太特殊了，但是你可以方便地自己来做：

```
1. from flask import Response
2.
3. def stream_template(template_name, **context):
4.     app.update_template_context(context)
5.     t = app.jinja_env.get_template(template_name)
6.     rv = t.stream(context)
7.     rv.enable_buffering(5)
8.     return rv
9.
10. @app.route('/my-large-page.html')
11. def render_large_template():
12.     rows = iter_all_rows()
13.     return Response(stream_template('the_template.html', rows=rows))
```

上例的技巧是从 Jinja2 环境中获得应用的模板对象，并调用 `stream()` 来代替 `render()`，返回一个流对象来代替一个字符串。由于我们绕过了 Flask 的模板渲染函数使用了模板对象本身，因此我们需要调用 `update_template_context()`，以确保更新被渲染的内容。这样，模板遍历流内容。由于每次产生内容后，服务器

都会把内容发送给客户端，因此可能需要缓存来保存内容。我们使用了 `rv.enable_buffering(size)` 来进行缓存。
5 是一个比较明智的缺省值。

情境中的流内容

Changelog

New in version 0.9.

注意，当你生成流内容时，请求情境已经在函数执行时消失了。 Flask 0.9 为你提供了一点帮助，让你可以在生成器运行期间保持请求情境：

```
1. from flask import stream_with_context, request, Response
2.
3. @app.route('/stream')
4. def streamed_response():
5.     def generate():
6.         yield 'Hello '
7.         yield request.args['name']
8.         yield '!'
9.     return Response(stream_with_context(generate()))
```

如果没有使用 `stream_with_context()` 函数，那么就会引发 `RuntimeError` 错误。

延迟的请求回调

Flask 的设计思路之一是：响应对象创建后被传递给一串回调函数，这些回调函数可以修改或替换响应对象。当请求处理开始的时候，响应对象还没有被创建。响应对象是由一个视图函数或者系统中的其他组件按需创建的。

但是当响应对象还没有创建时，我们如何修改响应对象呢？比如在一个 `before_request()` 回调函数中，我们需要根据响应对象设置一个 `cookie` 。

通常我们选择避开这种情形。例如可以尝试把应用逻辑移动到 `after_request()` 回调函数中。但是，有时候这个方法让人不爽，或者让代码变得很丑陋。

变通的方法使用 `after_this_request()` 回调函数，该函数只在当前请求后执行。这样你就可以在应用的任意地方延迟回调函数的执行。

在请求中的任何时候，可以注册在请求结束时将被调用的函数。例如，下例在 `before_request()` 回调函数中在 `cookie` 中记住了当前用户的语言：

```
1. from flask import request, after_this_request
2.
3. @app.before_request
4. def detect_user_language():
5.     language = request.cookies.get('user_lang')
6.
7.     if language is None:
8.         language = guess_language_from_request()
9.
10.     # when the response exists, set a cookie with the language
11.     @after_this_request
12.     def remember_language(response):
13.         response.set_cookie('user_lang', language)
14.         return response
15.
16.     g.language = language
```

添加 HTTP 方法重载

一些 HTTP 代理不支持所有 HTTP 方法或者不支持一些较新的 HTTP 方法（例如 PATCH）。在这种情况下，可以通过使用完全相反的协议，用一种 HTTP 方法来“代理”另一种 HTTP 方法。

实现的思路是让客户端发送一个 HTTP POST 请求，并设置 `X-HTTP-Method-Override` 头部。然后 HTTP 方法就会在传递给 Flask 前被头部的值代替。

通过 HTTP 中间件可以实现：

```
1. class HTTPMethodOverrideMiddleware(object):
2.     allowed_methods = frozenset([
3.         'GET',
4.         'HEAD',
5.         'POST',
6.         'DELETE',
7.         'PUT',
8.         'PATCH',
9.         'OPTIONS'
10.    ])
11.     bodyless_methods = frozenset(['GET', 'HEAD', 'OPTIONS', 'DELETE'])
12.
13.     def __init__(self, app):
14.         self.app = app
15.
16.     def __call__(self, environ, start_response):
17.         method = environ.get('HTTP_X_HTTP_METHOD_OVERRIDE', '').upper()
18.         if method in self.allowed_methods:
19.             environ['REQUEST_METHOD'] = method
20.         if method in self.bodyless_methods:
21.             environ['CONTENT_LENGTH'] = '0'
22.         return self.app(environ, start_response)
```

用中间件包裹 app 对象就可以与 Flask 一同工作了：

```
1. from flask import Flask
2.
3. app = Flask(__name__)
4. app.wsgi_app = HTTPMethodOverrideMiddleware(app.wsgi_app)
```

请求内容校验

请求数据会由不同的代码来处理或者预处理。例如 JSON 数据和表单数据都来源于已经读取并处理的请求对象，但是它们的处理代码是不同的。这样，当需要校验进来的请求数据时就会遇到麻烦。因此，有时候就有必要使用一些 API 。

幸运的是可以通过包装输入流来方便地改变这种情况。

下面的例子演示在 WSGI 环境下读取和储存输入数据，得到数据的 SHA1 校验：

```
1. import hashlib
2.
3. class ChecksumCalcStream(object):
4.
5.     def __init__(self, stream):
6.         self._stream = stream
7.         self._hash = hashlib.sha1()
8.
9.     def read(self, bytes):
10.         rv = self._stream.read(bytes)
11.         self._hash.update(rv)
12.         return rv
13.
14.     def readline(self, size_hint):
15.         rv = self._stream.readline(size_hint)
16.         self._hash.update(rv)
17.         return rv
18.
19. def generate_checksum(request):
20.     env = request.environ
21.     stream = ChecksumCalcStream(env['wsgi.input'])
22.     env['wsgi.input'] = stream
23.     return stream._hash
```

要使用上面的类，你只要在请求开始消耗数据之前钩接要计算的流就可以了。（按：小心操作 `request.form` 或类似东西。例如 `before_request_handlers` 就应当小心不要操作。）

用法示例：

```
1. @app.route('/special-api', methods=['POST'])
2. def special_api():
3.     hash = generate_checksum(request)
4.     # Accessing this parses the input stream
5.     files = request.files
6.     # At this point the hash is fully constructed.
7.     checksum = hash.hexdigest()
8.     return 'Hash was: %s' % checksum
```


基于 Celery 的后台任务

如果应用有一个长时间运行的任务，如处理上传数据或者发送电子邮件，而你不想在请求中等待任务结束，那么可以使用任务队列发送必须的数据给另一个进程。这样就可以在后台运行任务，立即返回请求。

Celery 是强大的任务队列库，它可以用于简单的后台任务，也可用于复杂的多阶段应用的计划。本文主要说明如何在 Flask 中配置使用 Celery。本文假设你已经阅读过了其官方文档中的 [Celery 入门](#)。

安装

Celery 是一个独立的 Python 包。使用 pip 从 PyPI 安装：

```
1. $ pip install celery
```

配置

你首先需要有一个 Celery 实例，这个实例称为 celery 应用。其地位就相当于 Flask 中 `Flask` 一样。这个实例被用作所有 Celery 相关事务的入口，如创建任务和管理工人，因此它必须可以被其他模块导入。

例如，你可以把它放在一个 `tasks` 模块中。这样不需要重新配置，你就可以使用 tasks 的子类，增加 Flask 应用情境的支持，并钩接 Flask 的配置。

只要如下这样就可以在 Flask 中使用 Celery 了：

```
1. from celery import Celery
2.
3. def make_celery(app):
4.     celery = Celery(
5.         app.import_name,
6.         backend=app.config['CELERY_RESULT_BACKEND'],
7.         broker=app.config['CELERY_BROKER_URL']
8.     )
9.     celery.conf.update(app.config)
10.
11.     class ContextTask(celery.Task):
12.         def __call__(self, *args, **kwargs):
13.             with app.app_context():
14.                 return self.run(*args, **kwargs)
15.
16.     celery.Task = ContextTask
17.     return celery
```

这个函数创建了一个新的 Celery 对象，使用了应用配置中的 broker，并从 Flask 配置中更新了 Celery 的其余配置。然后创建了一个任务子类，在一个应用情境中包装了任务执行。

一个示例任务

让我们来写一个任务，该任务把两个数字相加并返回结果。我们配置 Celery 的 broker，后端使用 Redis。使用上文的工厂创建一个 `celery` 应用，并用它定义任务。：

```
1. from flask import Flask
2.
3. flask_app = Flask(__name__)
4. flask_app.config.update(
5.     CELERY_BROKER_URL='redis://localhost:6379',
6.     CELERY_RESULT_BACKEND='redis://localhost:6379'
7. )
8. celery = make_celery(flask_app)
9.
10. @celery.task()
11. def add_together(a, b):
12.     return a + b
```

这个任务现在可以在后台调用了：

```
1. result = add_together.delay(23, 42)
2. result.wait() # 65
```

运行 Celery 工人

至此，如果你已经按上文一步一步执行，你会失望地发现你的 `.wait()` 不会真正返回。这是因为还需要运行一个 Celery 工人来接收和执行任务。：

```
1. $ celery -A your_application.celery worker
```

把 `yourapplication` 字符串替换为你创建 `_celery` 对像的应用包或模块。

现在工人已经在运行中，一旦任务结束，`wait` 就会返回结果。

继承 Flask

`Flask` 类可以被继承。

例如，这样可以通过继承重载请求参数如何保留其顺序：

```
1. from flask import Flask, Request
2. from werkzeug.datastructures import ImmutableOrderedMultiDict
3. class MyRequest(Request):
4.     """Request subclass to override request parameter storage"""
5.     parameter_storage_class = ImmutableOrderedMultiDict
6. class MyFlask(Flask):
7.     """Flask subclass using the custom request class"""
8.     request_class = MyRequest
```

推荐以这种方式重载或者增强 `Flask` 的内部功能。

单页应用

Flask 可以用为单页应用（SPA）提供服务，实现方式是把前端框架生成的静态文件放在项目的子文件夹中。你还需要创建一个全包端点把所有请求指向你的 SPA。

下面的演示如何用 一个 API 为 SPA 提供服务：

```
1. from flask import Flask, jsonify
2.
3. app = Flask(__name__, static_folder='app')
4.
5.
6. @app.route("/heartbeat")
7. def heartbeat():
8.     return jsonify({"status": "healthy"})
9.
10.
11. @app.route('/', defaults={'path': ''})
12. @app.route('/<path:path>')
13. def catch_all(path):
14.     return app.send_static_file("index.html")
```

部署方式

虽然轻便且易于使用，但是 **Flask** 的内建服务器不适用于生产，它也不能很好的扩展。本文主要说明在生产环境下正确使用 Flask 的一些方法。

如果想要把 Flask 应用部署到这里没有列出的 WSGI 服务器，请查询其文档中关于如何使用 WSGI 的部分，只要记住：`Flask` 应用对象实质上是一个 WSGI应用。

托管选项

托管于：

- [Heroku](#)
- [Google App Engine](#)
- [AWS Elastic Beanstalk](#)
- [Azure \(IIS\)](#)
- [PythonAnywhere](#)

自主部署选项

- [独立 WSGI 容器](#)
 - [Gunicorn](#)
 - [uWSGI](#)
 - [Gevent](#)
 - [Twisted Web](#)
 - [代理设置](#)
- [uWSGI](#)
 - [使用 uwsgi 启动你的应用](#)
 - [配置 nginx](#)
- [mod_wsgi \(Apache\)](#)
 - [安装 mod_wsgi](#)
 - [创建一个 .wsgi 文件](#)
 - [配置 Apache](#)
 - [故障排除](#)
 - [支持自动重载](#)
 - [使用虚拟环境](#)
- [FastCGI](#)
 - [创建一个 .fcgi 文件](#)
 - [配置 Apache](#)
 - [配置 lighttpd](#)
 - [配置 nginx](#)

部署方式

- [运行 FastCGI 进程](#)
- [调试](#)
- [CGI](#)
 - [创建一个 `.cgi` 文件](#)
 - [服务器设置](#)

大型应用

以下是一些建议，当你的代码库日益壮大或者应用需要规划时可以参考。

阅读源代码

Werkzeug（WSGI）和 Jinja（模板）是两个被广泛使用的工具，而 Flask 起源就是用于展示如何基于这两个工具创建你自己的框架。随着不断地开发，Flask 被越来越多的人认可了。当你的代码库日益壮大时，不应当仅仅是使用 Flask，而更应当理解它。所以，请阅读 Flask 的源代码吧。Flask 的源代码阅读方便，文档公开，有利于你直接使用内部的 API。Flask 坚持把上游库的 API 文档化，并文档化自己内部的工具，因此通过阅读源代码，可以为你的项目找到更好的切入点。

挂接，扩展

API 文档随处可见可用重载、挂接点和 [信号](#)。你可以定制类似请求或响应对象的自定义类。请深入研究你所使用的 API，并在 Flask 发行版中有哪些可以立即使用的可定制部分。请研究你的哪些项目可以重构为工具集或 Flask 扩展。你可以在社区中发现很多 [扩展](#)。如果找不到满意的，那就写一个你自己的吧。

继承

`Flask` 类有许多方法专门为继承而设计。你可通过继承 `Flask`（参见链接的方法文档）快速的添加或者定制行为，并把子类实例化为一个应用类。这种方法同样适用于 [应用工厂](#)。示例参见[继承 Flask](#)。

用中间件包装

[应用调度](#) 一文中详细阐述了如何使用中间件。你可以引入中间件来包装你的 Flask 实例，在你的应用和 HTTP 服务器之间的层有所作为。Werkzeug 包含许多[中间件](#)。

派生

如果以上建议都没有用，那么直接派生 Flask 吧。Flask 的主要代码都在 Werkzeug 和 Jinja2 这两个库内。这两个库起了主要作用。Flask 只是把它们粘合在一起而已。对于一个项目来讲，底层框架的切入点很重要。因为如果不重视这一点，那么框架会变得非常复杂，势必带来陡峭的学习曲线，从而吓退用户。

Flask 并不推崇唯一版本。许多人为了避免缺陷，都使用打过补丁或修改过的版本。这个理念在 Flask 的许可中也有所体现：你不必返回你对框架所做的修改。

分支的缺点是大多数扩展都会失效，因为新的框架会使用不同的导入名称。更进一步：整合上游的变动将会变得十分复杂，上游变动越多，则整合越复杂。因此，创建分支一般是不得不为之的最后一招。

专家级的伸缩性

对于大多数网络应用来说，最复杂的莫过于对于用户量和数据量提供良好的伸缩性。Flask 本身具有良好的伸缩性，其伸缩性受限于你的应用代码、所使用的数据储存方式、Python 实现和应用所运行的服务器。

如果服务器数量增加一倍，你的应用性能就增加一倍，那么就代表伸缩性好。如果伸缩性不好，那么即使增加服务器的数量，也不会得到更好的性能。伸缩性更差的甚至不支持增加第二台服务器。

Flask 中唯一影响伸缩性的因素是环境本地代理。Flask 中的环境本地代理可以被定义为线程、进程或 greenlet。如果你的服务器不支持这些，那么 Flask 就不能支持全局代理。但是，当今主流的服务器都支持线程、进程或 greenlet，以提高并发性。Flask 的基础库 Werkzeug 对于线程、进程或 greenlet 都能够提供良好的支持。

与社区沟通

不管你的代码库是否强大，Flask 开发者总是保持框架的可操作性。如果发现 Flask 有什么问题，请立即通过邮件列表或 IRC 与社区进行沟通。对于 Flask 及其扩展的开发都来说，提升其在大型应用中的功能的最佳途径是倾听用户的心声。

API

This part of the documentation covers all the interfaces of Flask. For parts where Flask depends on external libraries, we document the most important right here and provide links to the canonical documentation.

Application Object

- `class flask.Flask(import_name, static_url_path=None, static_folder='static', static_host=None, host_matching=False, subdomain_matching=False, template_folder='templates', instance_path=None, instance_relative_config=False, root_path=None)`
- The flask object implements a WSGI application and acts as the central object. It is passed the name of the module or package of the application. Once it is created it will act as a central registry for the view functions, the URL rules, template configuration and much more.

The name of the package is used to resolve resources from inside the package or the folder the module is contained in depending on if the package parameter resolves to an actual python package (a folder with an `init.py` file inside) or a standard module (just a `.py` file).

For more information about resource loading, see `open_resource()`.

Usually you create a `Flask` instance in your main module or in the `init.py` file of your package like this:

```
1. from flask import Flask
2. app = Flask(__name__)
```

About the First Parameter

The idea of the first parameter is to give Flask an idea of what belongs to your application. This name is used to find resources on the filesystem, can be used by extensions to improve debugging information and a lot more.

So it's important what you provide there. If you are using a single module, **name** is always the correct value. If you however are using a package, it's usually recommended to hardcode the name of your package there.

For example if your application is defined in `yourapplication/app.py` you should create it with one of the two versions below:

```
1. app = Flask('yourapplication')
2. app = Flask(__name__.split('.')[0])
```

Why is that? The application will work even with **name**, thanks to how resources are looked up. However it will make debugging more painful. Certain extensions can make assumptions based on the import name of your application. For example the Flask-SQLAlchemy extension will look for the code in your application that triggered an SQL query in debug mode. If the import name is not properly setup, that debugging information is lost. (For example it would only pick up SQL queries in `yourapplication.app` and not `yourapplication.views.frontend`)

Changelog

New in version 1.0: The `host_matching` and `static_host` parameters were added.

New in version 1.0: The `subdomain_matching` parameter was added. Subdomain matching needs to be enabled manually now. Setting `SERVER_NAME` does not implicitly enable it.

New in version 0.11: The `root_path` parameter was added.

New in version 0.8: The `instance_path` and `instance_relative_config` parameters were added.

New in version 0.7: The `static_url_path`, `static_folder`, and `_template_folder` parameters were added.

- Parameters
- - **import_name** – the name of the application package
 - **static_url_path** – can be used to specify a different path for the static files on the web. Defaults to the name of the `static_folder` folder.
 - **static_folder** – the folder with static files that should be served at `static_url_path`. Defaults to the `'static'` folder in the root path of the application.
 - **static_host** – the host to use when adding the static route. Defaults to None. Required when using `host_matching=True` with a `static_folder` configured.
 - **host_matching** – set `url_map.host_matching` attribute. Defaults to False.
 - **subdomain_matching** – consider the subdomain relative to `SERVER_NAME` when matching routes. Defaults to False.
 - **template_folder** – the folder that contains the templates that should be used by the application. Defaults to `'templates'` folder in the root path of the application.
 - **instance_path** – An alternative instance path for the application. By default the folder `'instance'` next to the package or module is assumed to be the instance path.

- **instance_relative_config** - if set to `True` relative filenames for loading the config are assumed to be relative to the instance path instead of the application root.
- **root_path** - Flask by default will automatically calculate the path to the root of the application. In certain situations this cannot be achieved (for instance if the package is a Python 3 namespace package) and needs to be manually defined.
- `add_template_filter` (`_f, name=None`)
- Register a custom template filter. Works exactly like the `template_filter()` decorator.
 - Parameters
 - **name** - the optional name of the filter, otherwise the function name will be used.
- `add_template_global` (`_f, name=None`)
- Register a custom template global function. Works exactly like the `template_global()` decorator.

Changelog

New in version 0.10.

1. - Parameters
2. -

name - the optional name of the global function, otherwise the function name will be used.

- `add_template_test` (`_f, name=None`)
- Register a custom template test. Works exactly like the `template_test()` decorator.

Changelog

New in version 0.10.

1. - Parameters
2. -

name - the optional name of the test, otherwise the function name will be used.

- `add_url_rule` (`_rule, endpoint=None, view_func=None, provide_automatic_options=None, **options`)
- Connects a URL rule. Works exactly like the `route()` decorator. If a `view_func` is provided it will be registered with the endpoint.

Basically this example:

```
1. @app.route('/')
2. def index():
3.     pass
```

Is equivalent to the following:

```
1. def index():
2.     pass
3. app.add_url_rule('/', 'index', index)
```

If the `view_func` is not provided you will need to connect the endpoint to a view function like so:

```
1. app.view_functions['index'] = index
```

Internally `route()` invokes `add_url_rule()` so if you want to customize the behavior via subclassing you only need to change this method.

For more information refer to [URL Route Registrations](#).

Changelog

Changed in version 0.6: `OPTIONS` is added automatically as method.

Changed in version 0.2: `view_func` parameter added.

```
1. - Parameters
2. -
3. -
```

rule – the URL rule as string

```
1. -
```

endpoint – the endpoint for the registered URL rule. Flask itself assumes the name of the view function as endpoint

```
1. -
```

view_func – the function to call when serving a request to the provided endpoint

```
1. -
```

provide_automatic_options – controls whether the `OPTIONS` method should be added automatically. This can also be controlled by setting the `view_func.provide_automatic_options =`

`False` before adding the rule.

```
1. -
```

options – the options to be forwarded to the underlying `Rule` object. A change to Werkzeug is handling of method options. `methods` is a list of methods this rule should be limited to (`GET` , `POST` etc.). By default a rule just listens for `GET` (and implicitly `HEAD`). Starting with Flask 0.6, `OPTIONS` is implicitly added and handled by the standard request handling.

- `afterrequest` (`_f`)
- Register a function to be run after each request.

Your function must take one parameter, an instance of `response_class` and return a new response object or the same (see `process_response()`).

As of Flask 0.7 this function might not be executed at the end of the request in case an unhandled exception occurred.

- `afterrequest_funcs` = `None`
- A dictionary with lists of functions that should be called after each request. The key of the dictionary is the name of the blueprint this function is active for, `None` for all requests. This can for example be used to close database connections. To register a function here, use the `after_request()` decorator.
- `app_context` (`()`)
- Create an `AppContext` . Use as a `with` block to push the context, which will make `current_app` point at this application.

An application context is automatically pushed by `RequestContext.push()` when handling a request, and when running a CLI command. Use this to manually create a context outside of these situations.

```
1. with app.app_context():
2.     init_db()
```

See [应用情境](#).

Changelog

New in version 0.9.

- `app_ctx_globals_class`
- alias of `flask.ctx._AppCtxGlobals`
- `auto_find_instance_path` (`()`)

- Tries to locate the instance path if it was not provided to the constructor of the application class. It will basically calculate the path to a folder named `instance` next to your main file or the package.

Changelog

New in version 0.8.

- `beforefirst_request` (`_f`)
- Registers a function to be run before the first request to this instance of the application.

The function will be called without any arguments and its return value is ignored.

Changelog

New in version 0.8.

- `beforefirst_request_funcs` = `None`
- A list of functions that will be called at the beginning of the first request to this instance. To register a function, use the `before_first_request()` decorator.

Changelog

New in version 0.8.

- `beforerequest` (`_f`)
- Registers a function to run before each request.

For example, this can be used to open a database connection, or to load the logged in user from the session.

The function will be called without any arguments. If it returns a non-None value, the value is handled as if it was the return value from the view, and further request handling is stopped.

- `beforerequest_funcs` = `None`
- A dictionary with lists of functions that will be called at the beginning of each request. The key of the dictionary is the name of the blueprint this function is active for, or `None` for all requests. To register a function, use the `before_request()` decorator.
- `blueprints` = `None`
- all the attached blueprints in a dictionary by name. Blueprints can be attached multiple times so this dictionary does not tell you how often they got attached.

Changelog

New in version 0.7.

- `config` = `None`
- The configuration dictionary as `Config`. This behaves exactly like a regular dictionary but supports additional methods to load a config from files.
- `config_class`
- alias of `flask.config.Config`
- `contextprocessor` (`_f`)
- Registers a template context processor function.
- `create_global_jinja_loader` ()
- Creates the loader for the Jinja2 environment. Can be used to override just the loader and keeping the rest unchanged. It's discouraged to override this function. Instead one should override the `jinja_loader()` function instead.

The global loader dispatches between the loaders of the application and the individual blueprints.

Changelog

New in version 0.7.

- `create_jinja_environment` ()
- Create the Jinja environment based on `jinja_options` and the various Jinja-related methods of the app. Changing `jinja_options` after this will have no effect. Also adds Flask-related globals and filters to the environment.

Changelog

Changed in version 0.11: `Environment.auto_reload` set in accordance with `TEMPLATES_AUTO_RELOAD` configuration option.

New in version 0.5.

- `createurl_adapter` (`_request`)
- Creates a URL adapter for the given request. The URL adapter is created at a point where the request context is not yet setup so the request is passed explicitly.

Changelog

Changed in version 1.0: `SERVER_NAME` no longer implicitly enables subdomain matching. Use `subdomain_matching` instead.

Changed in version 0.9: This can now also be called without a request object when the URL adapter is created for the application context.

New in version 0.6.

- `property` `debug`
- Whether debug mode is enabled. When using `flask run` to start the development server, an interactive debugger will be shown for unhandled exceptions, and the server will be reloaded when code changes. This maps to the `DEBUG` config key. This is enabled when `env` is `'development'` and is overridden by the `FLASK_DEBUG` environment variable. It may not behave as expected if set in code.

Do not enable debug mode when deploying in production.

Default: `True` if `env` is `'development'`, or `False` otherwise.

- `defaultconfig` = {'APPLICATION_ROOT': '/', 'DEBUG': None, 'ENV': None, 'EXPLAIN_TEMPLATE_LOADING': False, 'JSONIFY_MIMETYPE': 'application/json', 'JSONIFY_PRETTYPRINT_REGULAR': False, 'JSON_AS_ASCII': True, 'JSON_SORT_KEYS': True, 'MAX_CONTENT_LENGTH': None, 'MAX_COOKIE_SIZE': 4093, 'PERMANENT_SESSION_LIFETIME': datetime.timedelta(days=31), 'PREFERRED_URL_SCHEME': 'http', 'PRESERVE_CONTEXT_ON_EXCEPTION': None, 'PROPAGATE_EXCEPTIONS': None, 'SECRET_KEY': None, 'SEND_FILE_MAX_AGE_DEFAULT': datetime.timedelta(seconds=43200), 'SERVER_NAME': None, 'SESSION_COOKIE_DOMAIN': None, 'SESSION_COOKIE_HTTPONLY': True, 'SESSION_COOKIE_NAME': 'session', 'SESSION_COOKIE_PATH': None, 'SESSION_COOKIE_SAMESITE': None, 'SESSION_COOKIE_SECURE': False, 'SESSION_REFRESH_EACH_REQUEST': True, 'TEMPLATES_AUTO_RELOAD': None, 'TESTING': False, 'TRAP_BAD_REQUEST_ERRORS': None, 'TRAP_HTTP_EXCEPTIONS': False, 'USE_X_SENDFILE': False}
- Default configuration parameters.
- `dispatch_request` ()
- Does the request dispatching. Matches the URL and returns the return value of the view or error handler. This does not have to be a response object. In order to convert the return value to a proper response object, call `make_response()`.

Changelog

Changed in version 0.7: This no longer does the exception handling, this code was moved to the new `full_dispatch_request()`.

- `teardown_appcontext` (_exc=

Blueprint Objects

- `class flask.Blueprint` (name, import_name, static_folder=None, static_url_path=None, template_folder=None, url_prefix=None, subdomain=None, url_defaults=None, root_path=None, cli_group=

Incoming Request Data

- `class flask.Request (environ, populate_request=True, shallow=False)`
- The request object used by default in Flask. Remembers the matched endpoint and view arguments.

It is what ends up as `request`. If you want to replace the request object used you can subclass this and set `request_class` to your subclass.

The request object is a `Request` subclass and provides all of the attributes Werkzeug defines plus a few Flask specific ones.

- `environ`
- The underlying WSGI environment.
- `path`
- `full_path`
- `script_root`
- `url`
- `base_url`
- `url_root`
- Provides different ways to look at the current [IRI](#). Imagine your application is listening on the following application root:

```
http://www.example.com/myapplication
```

And a user requests the following URI:

```
http://www.example.com/myapplication/%CF%80/page.html?x=y
```

In this case the values of the above mentioned attributes would be the following:

`path`

```
u'/π/page.html'
```

`full_path`

```
u'/π/page.html?x=y'
```

`script_root`

```
u'/myapplication'
```

`base_url`

```
u'http://www.example.com/myapplication/π/page.html&#39;
```

`url`

```
u'http://www.example.com/myapplication/π/page.html?x=y&#39;
```

`url_root`

```
u'http://www.example.com/myapplication/&#39;
```

- `property` `accept_charsets`
- List of charsets this client supports as `CharsetAccept` object.
- `property` `accept_encodings`
- List of encodings this client accepts. Encodings in a HTTP term are compression encodings such as gzip. For charsets have a look at `accept_charset` .
- `property` `accept_languages`
- List of languages this client accepts as `LanguageAccept` object.
- `property` `accept_mimetypes`
- List of mimetypes this client supports as `MIMEAccept` object.
- `property` `access_route`
- If a forwarded header exists this is a list of all ip addresses from the client ip to the last proxy server.
- `classmethod` `application` (*f*)
- Decorate a function as responder that accepts the request as the last argument. This works like the `responder()` decorator but the function is passed the request object as the last argument and the request object will be closed automatically:

```
@Request.application
def my_wsgi_app(request):
    return Response('Hello World!')
```

As of Werkzeug 0.14 HTTP exceptions are automatically caught and converted to responses instead of failing.

```
- Parameters
-
```

f – the WSGI callable to decorate

```
- Returns
-
```

a new WSGI callable

- property `args`
- The parsed URL parameters (the part in the URL after the questionmark).

By default an `ImmutableMultiDict` is returned from this function. This can be changed by setting `parameter_storage_class` to a different type. This might be necessary if the order of the form data is important.

- property `authorization`
- The Authorization object in parsed form.
- property `base_url`
- Like `url` but without the querystring see also: `trusted_hosts` .
- property `blueprint`
- The name of the current blueprint
- property `cache_control`
- A `RequestCacheControl` object for the incoming cache control headers.
- `close` ()
- Closes associated resources of this request object. This closes all file handles explicitly. You can also use the request object in a `with` statement which will automatically close it.

Changelog

New in version 0.9.

- `content_encoding`
- The Content-Encoding entity-header field is used as a modifier to the media-type. When present, its value indicates what additional content codings have been applied to the entity-body, and thus what decoding mechanisms must be applied in order to obtain the media-type referenced by the Content-Type header field.

Changelog

New in version 0.9.

- property `content_length`
- The Content-Length entity-header field indicates the size of the entity-body in bytes or, in the case of the HEAD method, the size of the entity-body that would have been sent had the request been a GET.
- `content_md5`
- The Content-MD5 entity-header field, as defined in RFC 1864, is an MD5

digest of the entity-body for the purpose of providing an end-to-end message integrity check (MIC) of the entity-body. (Note: a MIC is good for detecting accidental modification of the entity-body in transit, but is not proof against malicious attacks.)

Changelog

New in version 0.9.

- `content_type`
- The Content-Type entity-header field indicates the media type of the entity-body sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had the request been a GET.
- property `cookies`
- A `dict` with the contents of all cookies transmitted with the request.
- property `data`
- Contains the incoming request data as string in case it came with a mime type Werkzeug does not handle.
- `date`
- The Date general-header field represents the date and time at which the message was originated, having the same semantics as orig-date in RFC 822.
- `dict_storage_class`
- alias of `werkzeug.datastructures.ImmutableTypeConversionDict`
- property `endpoint`
- The endpoint that matched the request. This in combination with `view_args` can be used to reconstruct the same or a modified URL. If an exception happened when matching, this will be `None`.
- property `files`
- `MultiDict` object containing all uploaded files. Each key in `files` is the name from the `<input type="file" name="">`. Each value in `files` is a Werkzeug `FileStorage` object.

It basically behaves like a standard file object you know from Python, with the difference that it also has a `save()` function that can store the file on the filesystem.

Note that `files` will only contain data if the request method was POST, PUT or PATCH and the `<form>` that posted to the request had `enctype="multipart/form-data"`. It will be empty otherwise.

See the `MultiDict` / `FileStorage` documentation formore details about the used data structure.

- property `form`
- The form parameters. By default an `ImmutableMultiDict` is returned from this function. This can be changed by setting `parameter_storage_class` to a different type. This mightbe necessary if the order of the form data is important.

Please keep in mind that file uploads will not end up here, but insteadin the `files` attribute.

Changelog

Changed in version 0.9: Previous to Werkzeug 0.9 this would only contain form data for POSTand PUT requests.

- `form_data_parser_class`
- alias of `werkzeug.formparser.FormDataParser`
- classmethod `fromvalues` (args, *kwargs_)
- Create a new request object based on the values provided. Ifenviron is given missing values are filled from there. This method isuseful for small scripts when you need to simulate a request from an URL.Do not use this method for unittesting, there is a full featured clientobject (`Client`) that allows to create multipart requests,support for cookies etc.

This accepts the same options as the `EnvironBuilder` .

Changelog

Changed in version 0.5: This method now accepts the same arguments as `EnvironBuilder` . Because of this theenviron parameter is now called `environ_overrides`.

```
- Returns
-
```

request object

- property `full_path`
- Requested path as unicode, including the query string.
- `getdata` (_cache=True, as_text=False, parse_form_data=False)
- This reads the buffered incoming data from the client into onebytestring. By default this is cached but that behavior can bechanged by setting cache to False.

Usually it's a bad idea to call this method without checking the content length first as a client could send dozens of megabytes or more to cause memory problems on the server.

Note that if the form data was already parsed this method will not return anything as form data parsing does not cache the data like this method does. To implicitly invoke form data parsing function set `parse_form_data` to `True`. When this is done the return value of this method will be an empty string if the form parser handles the data. This generally is not necessary as if the whole data is cached (which is the default) the form parser will use the cached data to parse the form data. Please be generally aware of checking the content length first in any case before calling this method to avoid exhausting server memory.

If `as_text` is set to `True` the return value will be a decoded unicode string.

Changelog

New in version 0.9.

- `getjson` (`_force=False`, `silent=False`, `cache=True`)
- Parse `data` as JSON.

If the `mimetype` does not indicate JSON (`application/json`, see `is_json()`), this returns `None`.

If parsing fails, `on_json_loading_failed()` is called and its return value is used as the return value.

```
- Parameters
-
-
```

force – Ignore the `mimetype` and always try to parse JSON.

```
-
```

silent – Silence parsing errors and return `None` instead.

```
-
```

cache – Store the parsed JSON to return for subsequent calls.

- property `headers`
- The headers from the WSGI environ as immutable `EnvironHeaders`.
- property `host`
- Just the host including the port if available. See also: `trusted_hosts`.

- `property` `host_url`
- Just the host with scheme as IRI. See also: `trusted_hosts` .
- `property` `if_match`
- An object containing all the etags in the If-Match header.
 - Return type
 - `ETags`
- `property` `if_modified_since`
- The parsed If-Modified-Since header as datetime object.
- `property` `if_none_match`
- An object containing all the etags in the If-None-Match header.
 - Return type
 - `ETags`
- `property` `if_range`
- The parsed If-Range header.

Changelog

New in version 0.7.

```
- Return type
-
```

`IfRange`

- `property` `if_unmodified_since`
- The parsed If-Unmodified-Since header as datetime object.
- `property` `is_json`
- Check if the mimetype indicates JSON data, either `application/json` or `application/*+json`.
- `is_multiprocess`
- boolean that is True if the application is served by aWSGI server that spawns multiple processes.
- `is_multithread`
- boolean that is True if the application is served by a multithreaded WSGI server.

- `is_run_once`
- boolean that is True if the application will be executed only once in a process lifetime. This is the case for CGI for example, but it's not guaranteed that the execution only happens one time.
- property `is_secure`
- True if the request is secure.
- property `is_xhr`
- True if the request was triggered via a JavaScript XMLHttpRequest. This only works with libraries that support the `X-Requested-With` header and set it to "XMLHttpRequest". Libraries that do that are prototype, jQuery and Mochikit and probably some more.

Deprecated since version 0.13: `X-Requested-With` is not standard and is unreliable. You may be able to use `AcceptMixin.accept_mimetypes` instead.

- property `json`
- The parsed JSON data if `mimetype` indicates JSON(application/json, see `is_json()`).

Calls `get_json()` with default arguments.

- `jsonmodule` = _
-
- `list_storage_class`
- alias of `werkzeug.datastructures.ImmutableList`
- `make_form_data_parser` ()
- Creates the form data parser. Instantiates the `form_data_parser_class` with some parameters.

Changelog

New in version 0.8.

- property `max_content_length`
- Read-only view of the `MAX_CONTENT_LENGTH` config key.
- `max_forwards`
- The Max-Forwards request-header field provides a mechanism with the TRACE and OPTIONS methods to limit the number of proxies or gateways that can forward the request to the next inbound server.
- `method`
- The request method. (For example `'GET'` or `'POST'`).

- `property` `mimetype`
- Like `content_type`, but without parameters (eg, without charset, type etc.) and always lowercase. For example if the contenttype is `text/HTML; charset=utf-8` the mimetype would be `'text/html'`.
- `property` `mimetype_params`
- The mimetype parameters as dict. For example if the contenttype is `text/html; charset=utf-8` the params would be `{'charset': 'utf-8'}`.
- `onjson_loading_failed` (`_e`)
- Called if `get_json()` parsing fails and isn't silenced. If this method returns a value, it is used as the return value for `get_json()`. The default implementation raises `BadRequest`.
- `parameter_storage_class`
- alias of `werkzeug.datastructures.ImmutableMultiDict`
- `property` `path`
- Requested path as unicode. This works a bit like the regular pathinfo in the WSGI environment but will always include a leading slash, even if the URL root is accessed.
- `property` `pragma`
- The Pragma general-header field is used to include implementation-specific directives that might apply to any recipient along the request/response chain. All pragma directives specify optional behavior from the viewpoint of the protocol; however, some systems MAY require that behavior be consistent with the directives.
- `query_string`
- The URL parameters as raw bytestring.
- `property` `range`
- The parsed Range header.

Changelog

New in version 0.7.

- Return type
-

Range

- `referrer`

- The `Referer[sic]` request-header field allows the client to specify, for the server's benefit, the address (URI) of the resource from which the Request-URI was obtained (the "referrer", although the header field is misspelled).
- property `remote_addr`
- The remote address of the client.
- `remote_user`
- If the server supports user authentication, and the script is protected, this attribute contains the username the user has authenticated as.
- `routing_exception` = None_
- If matching the URL failed, this is the exception that will be raised / was raised as part of the request handling. This is usually a `NotFound` exception or something similar.
- `scheme`
- URL scheme (http or https).

Changelog

New in version 0.7.

- property `script_root`
- The root path of the script without the trailing slash.
- property `stream`
- If the incoming form data was not encoded with a known mime type the data is stored unmodified in this stream for consumption. Most of the time it is a better idea to use `data` which will give you that data as a string. The stream only returns the data once.

Unlike `input_stream` this stream is properly guarded that you can't accidentally read past the length of the input. Werkzeug will internally always refer to this stream to read data which makes it possible to wrap this object with a stream that does filtering.

Changelog

Changed in version 0.9: This stream is now always available but might be consumed by the form parser later on. Previously the stream was only set if no parsing happened.

- property `url`
- The reconstructed current URL as IRI. See also: `trusted_hosts`.

- property `url_charset`

▪ The charset that is assumed for URLs. Defaults to the value of `charset`.
Changelog

New in version 0.6.

- property `url_root`

▪ The full URL root (with hostname), this is the application root as IRI. See also: `trusted_hosts`.

- `url_rule` = None_

▪ The internal URL rule that matched the request. This can be useful to inspect which methods are allowed for the URL from before/after handler (`request.url_rule.methods`) etc. Though if the request's method was invalid for the URL rule, the valid list is available in `routing_exception.valid_methods` instead (an attribute of the Werkzeug exception `MethodNotAllowed`) because the request was never internally bound.

Changelog

New in version 0.6.

- property `user_agent`

▪ The current user agent.

- property `values`

▪ A `werkzeug.datastructures.CombinedMultiDict` that combines `args` and `form`.

- `viewargs` = None_

▪ A dict of view arguments that matched the request. If an exception happened when matching, this will be `None`.

- property `want_form_data_parsed`

▪ Returns True if the request method carries content. As of Werkzeug 0.9 this will be the case if a content type is transmitted.

Changelog

New in version 0.8.

- `flask.request`

▪ To access incoming request data, you can use the global `_request_object`. Flask parses incoming request data for you and gives you access to it through that global object. Internally Flask makes sure that you always get the correct data for the active thread if you are in a multithreaded environment.

This is a proxy. See [关于代理的说明](#) for more information.

The request object is an instance of a `Request` subclass and provides all of the attributes Werkzeug defines. This just shows a quick overview of the most important ones.

Response Objects

- class `flask.Response` (`response=None`, `status=None`, `headers=None`, `mimetype=None`, `content_type=None`, `direct_passthrough=False`)
- The response object that is used by default in Flask. Works like the `response` object from Werkzeug but is set to have an HTML mimetype by default. Quite often you don't have to create this object yourself because `make_response()` will take care of that for you.

If you want to replace the response object used you can subclass this and set `response_class` to your subclass.

Changelog

Changed in version 1.0: JSON support is added to the response, like the request. This is useful when testing to get the test client response data as JSON.

Changed in version 1.0: Added `max_cookie_size`.

- `headers`
- A `Headers` object representing the response headers.
- `status`
- A string with a response status.
- `status_code`
- The response status as integer.
- property `data`
- A descriptor that calls `get_data()` and `set_data()`.
- `get_json` (`_force=False`, `silent=False`, `cache=True`)
- Parse `data` as JSON.

If the mimetype does not indicate JSON (`application/json`, see `is_json()`), this returns `None`.

If parsing fails, `on_json_loading_failed()` is called and its return value is used as the return value.

```
- Parameters
```

```
-
```

force – Ignore the *mimetype* and always try to parse JSON.

```
-
```

silent – Silence parsing errors and return `None` instead.

```
-
```

cache – Store the parsed JSON to return for subsequent calls.

- property `is_json`
- Check if the *mimetype* indicates JSON data, either *application/json* or *application/*+json*.
- property `max_cookie_size`
- Read-only view of the `MAX_COOKIE_SIZE` config key.

See `max_cookie_size` in Werkzeug's docs.

- property `mimetype`
- The *mimetype* (content type without charset etc.)
- `setcookie` (`_key`, `value=''`, `max_age=None`, `expires=None`, `path='/'`, `domain=None`, `secure=False`, `httponly=False`, `samesite=None`)
- Sets a cookie. The parameters are the same as in the cookie `_Morsel` object in the Python standard library but it accepts unicode data, too.

A warning is raised if the size of the cookie header exceeds `max_cookie_size`, but the header will still be set.

```
- Parameters
```

```
-
```

key – the key (name) of the cookie to be set.

```
-
```

value – the value of the cookie.

```
-
```

max_age – should be a number of seconds, or None (default) if the cookie should last only as long as the client's browser session.

-

expires – should be a datetime object or UNIX timestamp.

-

path – limits the cookie to a given path, per default it will span the whole domain.

-

domain – if you want to set a cross-domain cookie. For example, `domain=".example.com"` will set a cookie that is readable by the domain `www.example.com`, `foo.example.com` etc. Otherwise, a cookie will only be readable by the domain that set it.

-

secure – If True, the cookie will only be available via HTTPS

-

httponly – disallow JavaScript to access the cookie. This is an extension to the cookie standard and probably not supported by all browsers.

-

samesite – Limits the scope of the cookie such that it will only be attached to requests if those requests are "same-site".

Sessions

If you have set `Flask.secret_key` (or configured it from `SECRET_KEY`) you can use sessions in Flask applications. A session makes it possible to remember information from one request to another. The way Flask does this is by using a signed cookie. The user can look at the session contents, but can't modify it unless they know the secret key, so make sure to set that to something complex and unguessable.

To access the current session you can use the `session` object:

- class `flask.session`
- The session object works pretty much like an ordinary dict, with

the difference that it keeps track on modifications. This is a proxy. See [关于代理的说明](#) for more information.

The following attributes are interesting:

- `new`
- `True` if the session is new, `False` otherwise.
- `modified`
- `True` if the session object detected a modification. Be advised that modifications on mutable structures are not picked up automatically, in that situation you have to explicitly set the attribute to `True` yourself. Here an example:

```
# this change is not picked up because a mutable object (here
# a list) is changed.
session['objects'].append(42)
# so mark it as modified yourself
session.modified = True
```

- `permanent`
- If set to `True` the session lives for `permanent_session_lifetime` seconds. The default is 31 days. If set to `False` (which is the default) the session will be deleted when the user closes the browser.

Session Interface

Changelog

New in version 0.8.

The session interface provides a simple way to replace the session implementation that Flask is using.

- class `flask.sessions.SessionInterface`
- The basic interface you have to implement in order to replace the default session interface which uses Werkzeug's `SecureCookieImplementation`. The only methods you have to implement are `open_session()` and `save_session()`, the others have useful defaults which you don't need to change.

The session object returned by the `open_session()` method has to provide a dictionary like interface plus the properties and methods from the `SessionMixin`. We recommend just subclassing a dict and adding that mixin:

```
class Session(dict, SessionMixin):
    pass
```

If `open_session()` returns `None` Flask will call into `make_null_session()` to create a session that acts as replacement if the session support cannot work

because some requirement is not fulfilled. The default `NullSession` class that is created will complain that the secret key was not set.

To replace the session interface on an application all you have to do is to assign `flask.Flask.session_interface` :

```
app = Flask(__name__)
app.session_interface = MySessionInterface()
```

Changelog

New in version 0.8.

- `getcookie_domain` (`_app`)
- Returns the domain that should be set for the session cookie.

Uses `SESSION_COOKIE_DOMAIN` if it is configured, otherwise falls back to detecting the domain based on `SERVER_NAME` .

Once detected (or if not set at all), `SESSION_COOKIE_DOMAIN` is updated to avoid re-running the logic.

- `getcookie_httponly` (`_app`)
- Returns True if the session cookie should be httponly. This currently just returns the value of the `SESSION_COOKIE_HTTPONLY` config var.
- `getcookie_path` (`_app`)
- Returns the path for which the cookie should be valid. The default implementation uses the value from the `SESSION_COOKIE_PATH` config var if it's set, and falls back to `APPLICATION_ROOT` or uses `/` if it's `None` .
- `getcookie_samesite` (`_app`)
- Return `'Strict'` or `'Lax'` if the cookie should use the `SameSite` attribute. This currently just returns the value of the `SESSION_COOKIE_SAMESITE` setting.
- `getcookie_secure` (`_app`)
- Returns True if the cookie should be secure. This currently just returns the value of the `SESSION_COOKIE_SECURE` setting.
- `getexpiration_time` (`_app`, `session`)
- A helper method that returns an expiration date for the session or `None` if the session is linked to the browser session. The default implementation returns now + the permanent session lifetime configured on the application.
- `isnull_session` (`_obj`)

- Checks if a given object is a null session. Null sessions are not asked to be saved.

This checks if the object is an instance of `null_session_class` by default.

- `make_null_session(_app)`
- Creates a null session which acts as a replacement object if the real session support could not be loaded due to a configuration error. This mainly aids the user experience because the job of the null session is to still support lookup without complaining but modifications are answered with a helpful error message of what failed.

This creates an instance of `null_session_class` by default.

- `null_session_class`
- `make_null_session()` will look here for the class that should be created when a null session is requested. Likewise the `is_null_session()` method will perform a typecheck against this type.

alias of `NullSession`

- `open_session(_app, request)`
- This method has to be implemented and must either return `None` in case the loading failed because of a configuration error or an instance of a session object which implements a dictionary like interface + the methods and attributes on `SessionMixin`.
- `picklebased = False`
- A flag that indicates if the session interface is pickle based. This can be used by Flask extensions to make a decision in regard to how to deal with the session object.

Changelog

New in version 0.10.

- `save_session(_app, session, response)`
- This is called for actual sessions returned by `open_session()` at the end of the request. This is still called during a request context so if you absolutely need access to the request you can do that.
- `should_set_cookie(_app, session)`
- Used by session backends to determine if a `Set-Cookie` header should be set for this session cookie for this response. If the session has been modified, the cookie is set. If the session is permanent and the `SESSION_REFRESH_EACH_REQUEST` config is true, the cookie is always set.

This check is usually skipped if the session was deleted.

Changelog

New in version 0.11.

- class `flask.sessions. SecureCookieSessionInterface`
 - The default session interface that stores sessions in signed cookies through the `itsdangerous` module.
 - static `digest_method ()`
 - the hash function to use for the signature. The default is `sha1`
 - `keyderivation` = `'hmac' _`
 - the name of the `itsdangerous` supported key derivation. The default is `hmac`.
 - `opensession (_app, request)`
 - This method has to be implemented and must either return `None` in case the loading failed because of a configuration error or an instance of a session object which implements a dictionary like interface + the methods and attributes on `SessionMixin`.
 - `salt` = `'cookie-session'`
 - the salt that should be applied on top of the secret key for the signing of cookie based sessions.
 - `savesession (_app, session, response)`
 - This is called for actual sessions returned by `open_session()` at the end of the request. This is still called during a request context so if you absolutely need access to the request you can do that.
 - `serializer` =
 - A python serializer for the payload. The default is a compactJSON derived serializer with support for some extra Python types such as datetime objects or tuples.
 - `session_class`
 - alias of `SecureCookieSession`
 - class `flask.sessions. SecureCookieSession (initial=None)`
 - Base class for sessions based on signed cookies.
- This session backend will set the `modified` and `accessed` attributes. It cannot reliably track whether a session is new (vs. empty), so `new` remains hard coded to `False`.
- `accessed` = `False`

- `header`, which allows caching proxies to cache different pages for different users.
- `get` (key, default=None)
- Return the value for key if key is in the dictionary, else default.
- `modified` = False
- When data is changed, this is set to `True`. Only the session dictionary itself is tracked; if the session contains mutable data (for example a nested dict) then this must be set to `True` manually when modifying that data. The session cookie will only be written to the response if this is `True`.
- `setdefault` (key, default=None)
- Insert key with a value of default if key is not in the dictionary. Return the value for key if key is in the dictionary, else default.
- class `flask.sessions.NullSession` (initial=None)
- Class used to generate nicer error messages if sessions are not available. Will still allow read-only access to the empty session but fail on setting.
- class `flask.sessions.SessionMixin`
- Expands a basic dictionary with session attributes.
 - `accessed` = True
 - Some implementations can detect when session data is read or written and set this when that happens. The mixin default is hardcoded to `True`.
 - `modified` = True
 - Some implementations can detect changes to the session and set this when that happens. The mixin default is hard coded to `True`.
 - property `permanent`
 - This reflects the `'_permanent'` key in the dict.

Notice

The `PERMANENT_SESSION_LIFETIME` config key can also be an integer starting with Flask 0.8. Either catch this down yourself or use the `permanent_session_lifetime` attribute on the app which converts the result to an integer automatically.

Test Client

- `class flask.testing.FlaskClient(*args, **kwargs)`
- Works like a regular Werkzeug test client but has some knowledge about how Flask works to defer the cleanup of the request context stack to the end of a `with` body when used in a `with` statement. For general information about how to use this class refer to `werkzeug.test.Client`.

Changelog

Changed in version 0.12: `app.test_client()` includes preset default environment, which can be set after instantiation of the `app.test_client()` object in `client.environ_base`.

Basic usage is outlined in the [测试 Flask 应用](#) chapter.

- `open(*args, **kwargs)`
- Takes the same arguments as the `EnvironBuilder` class with some additions: You can provide a `EnvironBuilder` or a WSGI environment as only argument instead of the `EnvironBuilder` arguments and two optional keyword arguments (`as_tuple`, `buffered`) that change the type of the return value or the way the application is executed.

Changelog

Changed in version 0.5: If a dict is provided as file in the dict for the data parameter the content type has to be called `content_type` now instead of `mimetype`. This change was made for consistency with `werkzeug.FileWrapper`.

The `follow_redirects` parameter was added to `open()`.

Additional parameters:

```
- Parameters
-
-
```

as_tuple – Returns a tuple in the form `(environ, result)`

```
-
```

buffered – Set this to `True` to buffer the application run. This will automatically close the application for you as well.

```
-
```

follow_redirects – Set this to `True` if the Client should follow HTTP redirects.

- `session_transaction(args, *kwargs_)`
- When used in combination with a `with` statement this opens a session transaction. This can be used to modify the session that the test client

uses. Once the `with` block is left the session is stored back.

```
with client.session_transaction() as session:
    session['value'] = 42
```

Internally this is implemented by going through a temporary `testrequest` context and since session handling could depend on request variables this function accepts the same arguments as `test_request_context()` which are directly passed through.

Test CLI Runner

- class `flask.testing.FlaskCliRunner` (`app`, `**kwargs`)
- A `CliRunner` for testing a Flask app's CLI commands. Typically created using `test_cli_runner()`. See [测试 CLI 命令](#).
 - `invoke` (`cli=None`, `args=None`, `**kwargs`)
 - Invokes a CLI command in an isolated environment. See `CliRunner.invoke` for full method documentation. See [测试 CLI 命令](#) for examples.

If the `obj` argument is not given, passes an instance of `ScriptInfo` that knows how to load the Flask app being tested.

```
- Parameters
-
-
```

cli – Command object to invoke. Default is the app's `cli` group.

```
-
```

args – List of strings to invoke the command with.

```
- Returns
-
```

a `Result` object.

Application Globals

To share data that is valid for one request only from one function to another, a global variable is not good enough because it would break in threaded environments. Flask provides you with a special object that ensures it is only valid for the active request and that will return different values for each request. In a nutshell: it does the right thing, like it does for `request` and `session`.

- `flask.g`

- A namespace object that can store data during an *application context*. This is an instance of `Flask.app_ctx_globals_class`, which defaults to `ctx._AppCtxGlobals`.

This is a good place to store resources during a request. During testing, you can use the *伪造资源和环境* pattern to pre-configure such resources.

This is a proxy. See [关于代理的说明](#) for more information.

Changelog

Changed in version 0.10: Bound to the application context instead of the request context.

- class `flask.ctx._AppCtxGlobals`
- A plain object. Used as a namespace for storing data during an application context.

Creating an app context automatically creates this object, which is made available as the `g` proxy.

- `'key' in g`
- Check whether an attribute is present.

Changelog

New in version 0.10.

- `iter(g)`
- Return an iterator over the attribute names.

Changelog

New in version 0.10.

- `get(name, default=None)`
- Get an attribute by name, or a default value. Like `dict.get()`.
 - Parameters
 - - **name** – Name of attribute to get.
 - **default** – Value to return if the attribute is not present.

Changelog

New in version 0.10.

- `pop(name, default=`

Useful Functions and Classes

- `flask.current_app`
- A proxy to the application handling the current request. This is useful to access the application without needing to import it, or if it can't be imported, such as when using the application factory pattern or in blueprints and extensions.

This is only available when an [application context](#) is pushed. This happens automatically during requests and CLI commands. It can be controlled manually with `app_context()`.

This is a proxy. See [关于代理的说明](#) for more information.

- `flask.has_request_context()`
- If you have code that wants to test if a request context is there or not this function can be used. For instance, you may want to take advantage of request information if the request object is available, but fail silently if it is unavailable.

```
class User(db.Model):

    def __init__(self, username, remote_addr=None):
        self.username = username
        if remote_addr is None and has_request_context():
            remote_addr = request.remote_addr
        self.remote_addr = remote_addr
```

Alternatively you can also just test any of the context bound objects (such as `request` or `g`) for truthness:

```
class User(db.Model):

    def __init__(self, username, remote_addr=None):
        self.username = username
        if remote_addr is None and request:
            remote_addr = request.remote_addr
        self.remote_addr = remote_addr
```

Changelog

New in version 0.7.

- `flask.copycurrent_request_context(_f)`
- A helper function that decorates a function to retain the current request context. This is useful when working with greenlets. The moment the function is decorated a copy of the request context is created and then pushed when the function is called. The current session is also included in the copied request context.

Example:

```
import gevent
from flask import copy_current_request_context

@app.route('/')
def index():
    @copy_current_request_context
    def do_some_work():
        # do some work here, it can access flask.request or
        # flask.session like you would otherwise in the view function.
        ...
    gevent.spawn(do_some_work)
    return 'Regular response'
```

Changelog

New in version 0.10.

- `flask.has_app_context()`
- Works like `has_request_context()` but for the application context. You can also just do a boolean check on the `current_app` object instead.

Changelog

New in version 0.9.

- `flask.url_for(_endpoint, **values)`
- Generates a URL to the given endpoint with the method provided.

Variable arguments that are unknown to the target endpoint are appended to the generated URL as query arguments. If the value of a query argument is `None`, the whole pair is skipped. In case blueprints are active you can shortcut references to the same blueprint by prefixing the local endpoint with a dot (`.`).

This will reference the index function local to the current blueprint:

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

For more information, head over to the [Quickstart](#).

Configuration values `APPLICATION_ROOT` and `SERVER_NAME` are only used when generating URLs outside of a request context.

To integrate applications, `Flask` has a hook to intercept URL build errors through `Flask.url_build_error_handlers`. The `url_for` function results in a `BuildError` when the current app does not have a URL for the given endpoint and values. When it does, the `current_app` calls its `url_build_error_handlers` if it is not `None`, which can return a string to use as the result of `url_for` (instead of `url_for`'s default to raise the `BuildError` exception) or re-raise the exception. An example:


```
def external_url_handler(error, endpoint, values):
    "Looks up an external URL when `url_for` cannot build a URL."
    # This is an example of hooking the build_error_handler.
    # Here, lookup_url is some utility function you've built
    # which looks up the endpoint in some external URL registry.
    url = lookup_url(endpoint, **values)
    if url is None:
        # External lookup did not have a URL.
        # Re-raise the BuildError, in context of original traceback.
        exc_type, exc_value, tb = sys.exc_info()
        if exc_value is error:
            raise exc_type, exc_value, tb
        else:
            raise error
    # url_for will use this result, instead of raising BuildError.
    return url

app.url_build_error_handlers.append(external_url_handler)
```

Here, `error` is the instance of `BuildError`, and `endpoint` and `values` are the arguments passed into `url_for`. Note that this is for building URLs outside the current application, and not for handling 404 Not Found errors.

Changelog

New in version 0.10: The `_scheme` parameter was added.

New in version 0.9: The **`anchor_`** and **`method_`** parameters were added.

New in version 0.9: Calls `Flask.handle_build_error()` on `BuildError`.

Parameters

▪

- **`endpoint`** – the endpoint of the URL (name of the function)
- **`values`** – the variable arguments of the URL rule
- **`_external`** – if set to `True`, an absolute URL is generated. Server address can be changed via `SERVERNAME` configuration variable which falls back to the `_Host` header, then to the IP and port of the request.
- **`_scheme`** – a string specifying the desired URL scheme. The `_external` parameter must be set to `True` or a `ValueError` is raised. The default behavior uses the same scheme as the current request, or `PREFERRED_URL_SCHEME` from the [app configuration](#) if no request context is available. As of Werkzeug 0.10, this also can be set to an empty string to build protocol-relative URLs.
- **`_anchor`** – if provided this is added as anchor to the URL.

- **`_method`** - if provided this explicitly specifies an HTTP method.

- `flask.abort(status, *args, **kwargs)`
- Raises an `HTTPException` for the given status code or `WSGIApplication`:

```
abort(404) # 404 Not Found
abort(Response('Hello World'))
```

Can be passed a WSGI application or a status code. If a status code is given it's looked up in the list of exceptions and will raise that exception, if passed a WSGI application it will wrap it in a proxy `WSGIException` and raise that:

```
abort(404)
abort(Response('Hello World'))
```

- `flask.redirect(location, code=302, Response=None)`
- Returns a response object (a WSGI application) that, if called, redirects the client to the target location. Supported codes are 301, 302, 303, 305, 307, and 308. 300 is not supported because it's not a real redirect and 304 because it's the answer for a request with a request with defined If-Modified-Since headers.

Changelog

New in version 0.10: The class used for the Response object can now be passed in.

New in version 0.6: The location can now be a unicode string that is encoded using the `iri_to_uri()` function.

- Parameters

-

- **`location`** - the location the response should redirect to.
- **`code`** - the redirect status code. defaults to 302.
- **`Response` (class)** - a Response class to use when instantiating a response. The default is `werkzeug.wrappers.Response` if unspecified.

- `flask.make_response(*args_)`
- Sometimes it is necessary to set additional headers in a view. Because views do not have to return response objects but can return a value that is converted into a response object by Flask itself, it becomes tricky to add headers to it. This function can be called instead of using a return and you will get a response object which you can use to attach headers.

If view looked like this and you want to add a new header:

```
def index():
    return render_template('index.html', foo=42)
```

You can now do something like this:

```
def index():
    response = make_response(render_template('index.html', foo=42))
    response.headers['X-Parachutes'] = 'parachutes are cool'
    return response
```

This function accepts the very same arguments you can return from a view function. This for example creates a response with a 404 errorcode:

```
response = make_response(render_template('not_found.html'), 404)
```

The other use case of this function is to force the return value of a view function into a response which is helpful with viewdecorators:

```
response = make_response(view_function())
response.headers['X-Parachutes'] = 'parachutes are cool'
```

Internally this function does the following things:

- if no arguments are passed, it creates a new response argument
- if one argument is passed, `flask.Flask.make_response()` is invoked with it.
- if more than one argument is passed, the arguments are passed to the `flask.Flask.make_response()` function as tuple.

Changelog

New in version 0.6.

- `flask.after_this_request` (`_f`)
- Executes a function after this request. This is useful to modify response objects. The function is passed the response object and has to return the same or a new one.

Example:

```
@app.route('/')
def index():
    @after_this_request
    def add_header(response):
        response.headers['X-Foo'] = 'Parachute'
        return response
    return 'Hello World!'
```

This is more useful if a function other than the view function wants to modify a response. For instance think of a decorator that wants to

adds some headers without converting the return value into a response object.

Changelog

New in version 0.9.

- `flask.sendfile` (`_filename_or_fp`, `mimetype=None`, `as_attachment=False`, `attachment_filename=None`, `add_etags=True`, `cache_timeout=None`, `conditional=False`, `last_modified=None`)
- Sends the contents of a file to the client. This will use the most efficient method available and configured. By default it will try to use the WSGI server's `file_wrapper` support. Alternatively you can set the application's `use_x_sendfile` attribute to `True` to directly emit an `X-Sendfile` header. This however requires support of the underlying webserver for `X-Sendfile`.

By default it will try to guess the `mimetype` for you, but you can also explicitly provide one. For extra security you probably want to send certain files as attachment (HTML for instance). The `mimetype` guessing requires a `filename` or an `attachment_filename` to be provided.

ETags will also be attached automatically if a `filename` is provided. You can turn this off by setting `add_etags=False`.

If `conditional=True` and `filename` is provided, this method will try to upgrade the response stream to support range requests. This will allow the request to be answered with partial content response.

Please never pass filenames to this function from user sources; you should use `send_from_directory()` instead.

Changelog

Changed in version 1.0: UTF-8 filenames, as specified in [RFC 2231](#), are supported.

Changed in version 0.12: The `filename` is no longer automatically inferred from file objects. If you want to use automatic `mimetype` and `etag` support, pass a `filepath` via `filename_or_fp` or `attachment_filename`.

Changed in version 0.12: The `attachment_filename` is preferred over `filename` for MIME-type detection.

Changed in version 0.9: `cache_timeout` pulls its default from application config, when `None`.

Changed in version 0.7: `mimetype` guessing and `etag` support for file objects was deprecated because it was unreliable. Pass a `filename` if you are able to, otherwise attach an `etag` yourself. This functionality will be removed in Flask 1.0

New in version 0.5: The `add_etags`, `cache_timeout` and `conditional` parameters were added. The default behavior is now to attach etags.

New in version 0.2.

Changed in version 1.1: Filename may be a `PathLike` object.

New in version 1.1: Partial content supports `BytesIO`.

Changelog

Changed in version 1.0.3: Filenames are encoded with ASCII instead of Latin-1 for broader compatibility with WSGI servers.

Parameters

▪

- **filename_or_fp** – the filename of the file to send. This is relative to the `root_path` if a relative path is specified. Alternatively a file object might be provided in which case `X-Sendfile` might not work and fallback to the traditional method. Make sure that the file pointer is positioned at the start of data to send before calling `send_file()`.
- **mimetype** – the mimetype of the file if provided. If a file path is given, auto detection happens as fallback, otherwise an error will be raised.
- **as_attachment** – set to `True` if you want to send this file with a `Content-Disposition: attachment` header.
- **attachment_filename** – the filename for the attachment if it differs from the file's filename.
- **add_etags** – set to `False` to disable attaching of etags.
- **conditional** – set to `True` to enable conditional responses.
- **cache_timeout** – the timeout in seconds for the headers. When `None` (default), this value is set by `get_send_file_max_age()` of `current_app`.
- **last_modified** – set the `Last-Modified` header to this value, a `datetime` or timestamp. If a file was passed, this overrides its `mtime`.
- `flask.send_from_directory` (`_directory`, `filename`, `**options`)
- Send a file from a given directory with `send_file()`. This is a secure way to quickly expose static files from an upload folder or something similar.

Example usage:

```
@app.route('/uploads/<path:filename>')
def download_file(filename):
    return send_from_directory(app.config['UPLOAD_FOLDER'],
                               filename, as_attachment=True)
```

Sending files and Performance

It is strongly recommended to activate either `X-Sendfile` support in your webserver or (if no authentication happens) to tell the webserver to serve files for the given path on its own without calling into the web application for improved performance.

Changelog

New in version 0.5.

- **Parameters**
- - **directory** – the directory where all the files are stored.
 - **filename** – the filename relative to that directory to download.
 - **options** – optional keyword arguments that are directly forwarded to `send_file()`.
- `flask. safe_join` (`_directory`, `*pathnames`)
- Safely join directory and zero or more untrusted `_pathnames_components`.

Example usage:

```
@app.route('/wiki/<path:filename>')
def wiki_page(filename):
    filename = safe_join(app.config['WIKI_FOLDER'], filename)
    with open(filename, 'rb') as fd:
        content = fd.read() # Read and process the file content...
```

- **Parameters**
- - **directory** – the trusted base directory.
 - **pathnames** – the untrusted pathnames relative to that directory.
- **Raises**
- `NotFound` if one or more passed paths fall out of its boundaries.
- `flask. escape` (`s`) → markup
- Convert the characters `&`, `<`, `>`, `'`, and `"` in string `s` to HTML-

`safe` sequences. Use this if you need to display text that might contain such characters in HTML. Marks return value as markup string.

- `class flask. Markup`
- A string that is ready to be safely inserted into an HTML or XML document, either because it was escaped or because it was marked safe.

Passing an object to the constructor converts it to text and wraps it to mark it safe without escaping. To escape the text, use the `escape()` class method instead.

```
>>> Markup('Hello, <em>World</em>!')
Markup('Hello, <em>World</em>!')
>>> Markup(42)
Markup('42')
>>> Markup.escape('Hello, <em>World</em>!')
Markup('Hello &lt;em&gt;World&lt;/em&gt;!')
```

This implements the `html()` interface that some frameworks use. Passing an object that implements `html()` will wrap the output of that method, marking it safe.

```
>>> class Foo:
...     def __html__(self):
...         return '<a href="/foo">foo</a>'
...
>>> Markup(Foo())
Markup('<a href="/foo">foo</a>')
```

This is a subclass of the text type (`str` in Python 3, `unicode` in Python 2). It has the same methods as that type, but all methods escape their arguments and return a `Markup` instance.

```
>>> Markup('<em>%s</em>') % 'foo & bar'
Markup('<em>foo & bar</em>')
>>> Markup('<em>Hello</em> ') + '<foo>'
Markup('<em>Hello</em> &lt;foo&gt;')
```

- classmethod `escape(s)`
- Escape a string. Calls `escape()` and ensures that for subclasses the correct type is returned.
- `striptags()`
- `unescape()` the markup, remove tags, and normalize whitespace to single spaces.

```
>>> Markup('Main &raquo; <em>About</em>').striptags()
'Main » About'
```

- `unescape()`

- Convert escaped markup back into a text string. This replaces HTML entities with the characters they represent.

```
>>> Markup('Main &raquo; <em>About</em>').unescape()
'Main » <em>About</em>'
```

Message Flashing

- `flask.flash` (`message`, `category='message'`)
- Flashes a message to the next request. In order to remove the flashed message from the session and to display it to the user, the template has to call `get_flashed_messages()`.

Changelog

Changed in version 0.3: `category` parameter added.

- Parameters
- - **message** – the message to be flashed.
 - **category** – the category for the message. The following values are recommended: `'message'` for any kind of message, `'error'` for errors, `'info'` for information messages and `'warning'` for warnings. However any kind of string can be used as category.
- `flask.get_flashed_messages` (`_with_categories=False`, `category_filter=()`)
- Pulls all flashed messages from the session and returns them. Further calls in the same request to the function will return the same messages. By default just the messages are returned, but when `with_categories` is set to `True`, the return value will be a list of tuples in the form `(category, message)` instead.

Filter the flashed messages to one or more categories by providing those categories in `category_filter`. This allows rendering categories in separate HTML blocks. The `with_categories` and `_category_filter_arguments` are distinct:

- `with_categories` controls whether categories are returned with message text (`True` gives a tuple, where `False` gives just the message text).
- `category_filter` filters the messages down to only those matching the provided categories.

See [消息闪现](#) for examples.

Changelog

Changed in version 0.9: `category_filter` parameter added.

Changed in version 0.3: `with_categories` parameter added.

- **Parameters**
-
- **`with_categories`** – set to `True` to also receive categories.
- **`category_filter`** – whitelist of categories to limit return values

JSON Support

Flask uses `simplejson` for the JSON implementation. Since `simplejson` is provided by both the standard library as well as extension, Flask will try `simplejson` first and then fall back to the `stdlib json` module. On top of that it will delegate access to the current application's JSON encoders and decoders for easier customization.

So for starters instead of doing:

```
try:
    import simplejson as json
except ImportError:
    import json
```

You can instead just do this:

```
from flask import json
```

For usage examples, read the `json` documentation in the standard library. The following extensions are by default applied to the `stdlib's JSON` module:

- `datetime` objects are serialized as **RFC 822** strings.
- Any object with an `html` method (like `Markup`) will have that method called and then the return value is serialized as a string.

The `htmlsafe_dumps()` function of this `json` module is also available as a filter called `|tojson` in Jinja2. Note that inside `script` tags no escaping must take place, so make sure to disable escaping with `|safe` if you intend to use it inside `script` tags unless you are using Flask 0.10 which implies that:

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

Auto-Sort JSON Keys

The configuration variable `JSON_SORT_KEYS` (配置管理) can be set to `false` to stop Flask from auto-sorting keys. By default sorting is enabled and outside of the app context sorting is turned on.

Notice that disabling key sorting can cause issues when using content-based HTTP caches and Python's hash randomization feature.

- `flask.json.jsonify(*args, **kwargs)`
- This function wraps `dump()` to add a few enhancements that make life easier. It turns the JSON output into a `Response` object with the `application/json` mimetype. For convenience, it also converts multiple arguments into an array or multiple keyword arguments into a dict. This means that both `jsonify(1,2,3)` and `jsonify([1,2,3])` serialize to `[1,2,3]`.

For clarity, the JSON serialization behavior has the following differences from `dump()`:

- Single argument: Passed straight through to `dump()`.
- Multiple arguments: Converted to an array before being passed to `dump()`.
- Multiple keyword arguments: Converted to a dict before being passed to `dump()`.
- Both args and kwargs: Behavior undefined and will throw an exception.

Example usage:

```
from flask import jsonify

@app.route('/_get_current_user')
def get_current_user():
    return jsonify(username=g.user.username,
                  email=g.user.email,
                  id=g.user.id)
```

This will send a JSON response like this to the browser:

```
{
  "username": "admin",
  "email": "admin@localhost",
  "id": 42
}
```

Changelog

Changed in version 0.11: Added support for serializing top-level arrays. This introduces a security risk in ancient browsers. See [JSON 安全](#) for

details.

This function's response will be pretty printed if the `JSONIFY_PRETTYPRINT_REGULAR` config parameter is set to `True` or the Flask app is running in debug mode. Compressed (not pretty) formatting currently means no indents and no spaces after separators.

Changelog

New in version 0.2.

- `flask.json.dumps` (`obj`, `app=None`, `**kwargs`)
- Serialize `obj` to a JSON-formatted string. If there is an app context pushed, use the current app's configured encoder (`json_encoder`), or fall back to the default `JSONEncoder`.

Takes the same arguments as the built-in `json.dumps()`, and does some extra configuration based on the application. If the `simplejson` package is installed, it is preferred.

Parameters

-

- **`obj`** - Object to serialize to JSON.
- **`app`** - App instance to use to configure the JSON encoder. Uses `current_app` if not given, and falls back to the default encoder when not in an app context.
- **`kwargs`** - Extra arguments passed to `json.dumps()`.

Changelog

Changed in version 1.0.3: `app` can be passed directly, rather than requiring an app context for configuration.

- `flask.json.dump` (`obj`, `fp`, `app=None`, `**kwargs`)
- Like `dumps()` but writes into a file object.
- `flask.json.loads` (`s`, `app=None`, `**kwargs`)
- Deserialize an object from a JSON-formatted string `s`. If there is an app context pushed, use the current app's configured decoder (`json_decoder`), or fall back to the default `JSONDecoder`.

Takes the same arguments as the built-in `json.loads()`, and does some extra configuration based on the application. If the `simplejson` package is installed, it is preferred.

Parameters

-

- **`s`** - JSON string to deserialize.

- **app** – App instance to use to configure the JSON decoder. Uses `current_app` if not given, and falls back to the default encoder when not in an app context.
- **kwargs** – Extra arguments passed to `json.dumps()`.

Changelog

Changed in version 1.0.3: `app` can be passed directly, rather than requiring an app context for configuration.

- `flask.json.load(fp, app=None, **kwargs)`
- Like `loads()` but reads from a file object.
- class `flask.json.JSONEncoder` (*, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None)
- The default Flask JSON encoder. This one extends the default encoder by also supporting `datetime`, `UUID`, `dataclasses`, and `Markup` objects. `datetime` objects are serialized as RFC 822 datetime strings. This is the same as the HTTP date format.

In order to support more data types, override the `default()` method.

- `default(o)`
- Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement `default` like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    return JSONEncoder.default(self, o)
```

- class `flask.json.JSONDecoder` (*, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, strict=True, object_pairs_hook=None)
- The default JSON decoder. This one does not change the behavior from the default simplejson decoder. Consult the `json` documentation for more information. This decoder is not only used for the load functions of this module but also `Request`.

Tagged JSON

A compact representation for lossless serialization of non-standard JSON

types. `SecureCookieSessionInterface` uses this to serialize the session data, but it may be useful in other places. It can be extended to support other types.

- class `flask.json.tag.TaggedJSONSerializer`
- Serializer that uses a tag system to compactly represent objects that are not JSON types. Passed as the intermediate serializer to `itsdangerous.Serializer`.

The following extra types are supported:

- `dict`
- `tuple`
- `bytes`
- `Markup`
- `UUID`
- `datetime`
- `default_tags` = `[, , , , , , ,]_`
- Tag classes to bind when creating the serializer. Other tags can be added later using `register()`.
- `dumps` (value)
- Tag the value and dump it to a compact JSON string.
- `loads` (value)
- Load data from a JSON string and deserialize any tagged objects.
- `register` (tag_class, force=False, index=None)
- Register a new tag with this serializer.
 - Parameters
 - - **tag_class** - tag class to register. Will be instantiated with this serializer instance.
 - **force** - overwrite an existing tag. If false (default), a `KeyError` is raised.
 - **index** - index to insert the new tag in the tag order. Useful when the new tag is a special case of an existing tag. If `None` (default), the tag is appended to the end of the order.

- *Raises*
 - **KeyError** – if the tag key is already registered and `force` is not true.
- `tag` (value)
 - Convert a value to a tagged representation if necessary.
- `untag` (value)
 - Convert a tagged representation back to the original type.
- `class flask.json.tag. JSONTag (serializer)`
- Base class for defining type tags for `TaggedJSONSerializer` .
 - `check` (value)
 - Check if the given value should be tagged by this tag.
 - `key` = None
 - The tag to mark the serialized object with. If `None` , this tag is only used as an intermediate step during tagging.
 - `tag` (value)
 - Convert the value to a valid JSON type and add the tag structure around it.
 - `tojson` (_value)
 - Convert the Python object to an object that is a valid JSON type. The tag will be added later.
 - `topython` (_value)
 - Convert the JSON representation back to the correct type. The tag will already be removed.

Let's see an example that adds support for `OrderedDict` . Dicts don't have an order in Python or JSON, so to handle this we will dump the items as a list of `[key, value]` pairs. Subclass `JSONTag` and give it the new key `'od'` to identify the type. The session serializer processes dicts first, so insert the new tag at the front of the order since `OrderedDict` must be processed before `dict` .

```

from flask.json.tag import JSONTag

class TagOrderedDict(JSONTag):
    __slots__ = ('serializer',)
    key = 'od'

    def check(self, value):
        return isinstance(value, OrderedDict)

    def to_json(self, value):
        return [[k, self.serializer.tag(v)] for k, v in iteritems(value)]

    def to_python(self, value):
        return OrderedDict(value)

app.session_interface.serializer.register(TagOrderedDict, index=0)

```

Template Rendering

- `flask. rendertemplate` (`_template_name_or_list`, `**context`)
 - Renders a template from the template folder with the given context.
 - Parameters
 - `template_name_or_list` - the name of the template to be rendered, or an iterable with template names the first one existing will be rendered
 - `context` - the variables that should be available in the context of the template.
- `flask. rendertemplate_string` (`_source`, `**context`)
 - Renders a template from the given template source string with the given context. Template variables will be autoescaped.
 - Parameters
 - `source` - the source code of the template to be rendered
 - `context` - the variables that should be available in the context of the template.
 - `flask. gettemplate_attribute` (`_template_name`, `attribute`)
 - Loads a macro (or variable) a template exports. This can be used to invoke a macro from within Python code. If you for example have a template named `_cider.html` with the following contents:

```
{% macro hello(name) %}Hello {{ name }}!{% endmacro %}
```

You can access this from Python code like this:

```
hello = get_template_attribute('_cider.html', 'hello')
return hello('World')
```

Changelog

New in version 0.2.

- Parameters
-
- **template_name** - the name of the template
- **attribute** - the name of the variable of macro to access

Configuration

- class `flask.Config` (root_path, defaults=None)
- Works exactly like a dict but provides ways to fill it from files or special dictionaries. There are two common patterns to populate the config.

Either you can fill the config from a config file:

```
app.config.from_pyfile('yourconfig.cfg')
```

Or alternatively you can define the configuration options in the module that calls `from_object()` or provide an import path to a module that should be loaded. It is also possible to tell it to use the same module and with that provide the configuration values just before the call:

```
DEBUG = True
SECRET_KEY = 'development key'
app.config.from_object(__name__)
```

In both cases (loading from any Python file or loading from modules), only uppercase keys are added to the config. This makes it possible to use lowercase values in the config file for temporary values that are not added to the config or to define the config keys in the same file that implements the application.

Probably the most interesting way to load configurations is from an environment variable pointing to a file:

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

In this case before launching the application you have to set this environment variable to the file you want to use. On Linux and OS

Use the export statement:

```
export YOURAPPLICATION_SETTINGS='/path/to/config/file'
```

On windows use set instead.

- Parameters

-

- **root_path** - path to which files are read relative from. When the config object is created by the application, this is the application's `root_path`.

- **defaults** - an optional dictionary of default values

- `fromenvvar` (`_variable_name`, `silent=False`)

- Loads a configuration from an environment variable pointing to a configuration file. This is basically just a shortcut with nicer error messages for this line of code:

```
app.config.from_pyfile(os.environ['YOURAPPLICATION_SETTINGS'])
```

```
- Parameters
```

```
-
```

```
-
```

variable_name - name of the environment variable

```
-
```

silent - set to `True` if you want silent failure for missing files.

```
- Returns
```

```
-
```

bool. `True` if able to load config, `False` otherwise.

- `fromjson` (`_filename`, `silent=False`)

- Updates the values in the config from a JSON file. This function behaves as if the JSON object was a dictionary and passed to the `from_mapping()` function.

- Parameters

-

- **filename** - the filename of the JSON file. This can either be an absolute filename or a filename relative to the root path.

- **silent** – set to `True` if you want silent failure for missingfiles.

Changelog

New in version 0.11.

- `frommapping` (mapping, *kwargs_)
- Updates the config like `update()` ignoring items with non-upperkeys.

Changelog

New in version 0.11.

- `fromobject` (_obj)
- Updates the values from the given object. An object can be of one of the following two types:
 - a string: in this case the object with that name will be imported
 - an actual object reference: that object is used directly

Objects are usually either modules or classes. `from_object()` loads only the uppercase attributes of the module/class. A `dict` object will not work with `from_object()` because the keys of a `dict` are not attributes of the `dict` class.

Example of module-based configuration:

```
app.config.from_object('yourapplication.default_config')
from yourapplication import default_config
app.config.from_object(default_config)
```

Nothing is done to the object before loading. If the object is a class and has `@property` attributes, it needs to be instantiated before being passed to this method.

You should not use this function to load the actual configuration but rather configuration defaults. The actual config should be loaded with `from_pyfile()` and ideally from a location not within the package because the package might be installed system wide.

See [开发/生产](#) for an example of class-based configuration using

```
from_object()
```

```
- Parameters
-
```

obj – an import name or object

- `frompyfile` (`_filename`, `silent=False`)
- Updates the values in the config from a Python file. This function behaves as if the file was imported as module with the `from_object()` function.
 - Parameters
 - - **filename** – the filename of the config. This can either be an absolute filename or a filename relative to the root path.
 - **silent** – set to `True` if you want silent failure for missing files.

Changelog

New in version 0.7: `silent` parameter.

- `getnamespace` (`_namespace`, `lowercase=True`, `trim_namespace=True`)
- Returns a dictionary containing a subset of configuration options that match the specified namespace/prefix. Example usage:

```
app.config['IMAGE_STORE_TYPE'] = 'fs'
app.config['IMAGE_STORE_PATH'] = '/var/app/images'
app.config['IMAGE_STORE_BASE_URL'] = 'http://img.website.com'
image_store_config = app.config.get_namespace('IMAGE_STORE_')
```

The resulting dictionary `image_store_config` would look like:

```
{
  'type': 'fs',
  'path': '/var/app/images',
  'base_url': 'http://img.website.com'
}
```

This is often useful when configuration options map directly to keyword arguments in functions or class constructors.

```
- Parameters
-
-
```

namespace – a configuration namespace

```
-
```

lowercase – a flag indicating if the keys of the resulting dictionary should be lowercase

```
-
```

`trim_namespace` – a flag indicating if the keys of the resulting dictionary should not include the namespace

Changelog

New in version 0.11.

Stream Helpers

- `flask.stream_with_context` (`_generator_or_function`)
- Request contexts disappear when the response is started on the server. This is done for efficiency reasons and to make it less likely to encounter memory leaks with badly written WSGI middlewares. The downside is that if you are using streamed responses, the generator cannot access request bound information any more.

This function however can help you keep the context around for longer:

```
from flask import stream_with_context, request, Response

@app.route('/stream')
def streamed_response():
    @stream_with_context
    def generate():
        yield 'Hello '
        yield request.args['name']
        yield '!'
    return Response(generate())
```

Alternatively it can also be used around a specific generator:

```
from flask import stream_with_context, request, Response

@app.route('/stream')
def streamed_response():
    def generate():
        yield 'Hello '
        yield request.args['name']
        yield '!'
    return Response(stream_with_context(generate()))
```

Changelog

New in version 0.9.

Useful Internals

- class `flask.ctx.RequestContext` (`app, environ, request=None, session=None`)
- The request context contains all request relevant information. It

is created at the beginning of the request and pushed to the `request_ctx_stack` and removed at the end of it. It will create the URL adapter and request object for the WSGI environment provided.

Do not attempt to use this class directly, instead

use `test_request_context()` and `request_context()` to create this object.

When the request context is popped, it will evaluate all the functions registered on the application for teardown execution(`teardown_request()`).

The request context is automatically popped at the end of the request for you. In debug mode the request context is kept around if exceptions happen so that interactive debuggers have a chance to introspect the data. With 0.4 this can also be forced for requests that did not fail and outside of `DEBUG` mode. By setting `'flask._preserve_context'` to `True` on the WSGI environment the context will not pop itself at the end of the request. This is used by the `test_client()` for example to implement the deferred cleanup functionality.

You might find this helpful for unittests where you need the information from the context local around for a little longer. Make sure to properly `pop()` the stack yourself in that situation, otherwise your unittests will leak memory.

- `copy ()`
- Creates a copy of this request context with the same request object. This can be used to move a request context to a different greenlet. Because the actual request object is the same this cannot be used to move a request context to a different thread unless access to the request object is locked.

Changed in version 1.1: The current session object is used instead of reloading the original data. This prevents flask.session pointing to an out-of-date object.

Changelog

New in version 0.10.

- `match_request ()`
- Can be overridden by a subclass to hook into the matching of the request.
- `pop (exc=`
 - `flask._request_ctx_stack`
 - The internal `LocalStack` that holds `RequestContext` instances. Typically, the `request` and `session` proxies should be accessed instead of the stack. It may be useful to access the

stack inextension code.

The following attributes are always present on each layer of thestack:

- *app*
- *the active Flask application.*
- *url_adapter*
- *the URL adapter that was used to match the request.*
- *request*
- *the current request object.*
- *session*
- *the active session object.*
- *g*
- *an object with all the attributes of the flask.g object.*
- *flashes*
- *an internal cache for the flashed messages.*

Example usage:

```
from flask import _request_ctx_stack

def get_session():
    ctx = _request_ctx_stack.top
    if ctx is not None:
        return ctx.session
```

- *class flask.ctx. AppContext (app)*
- *The application context binds an application object implicitlyto the current thread or greenlet, similar to how the RequestContext binds request information. The applicationcontext is also implicitly created if a request context is createdbut the application is not on top of the individual applicationcontext.*
- *pop (exc=*
 - *flask. _app_ctx_stack*
 - *The internal LocalStack that holds AppContext instances. Typically, the current_app and g proxies should be accessed insteadof the stack. Extensions can access the contexts on the stack as anamespace to store*

data. Changelog

New in version 0.9.

- class `flask.blueprints.BlueprintSetupState` (`blueprint`, `app`, `options`, `first_registration`)
- Temporary holder object for registering a blueprint with the application. An instance of this class is created by the `make_setup_state()` method and later passed to all register callback functions.
 - `addurl_rule` (`_rule`, `endpoint=None`, `view_func=None`, `**options`)
 - A helper method to register a rule (and optionally a view function) to the application. The endpoint is automatically prefixed with the blueprint's name.
 - `app` = None
 - a reference to the current application
 - `blueprint` = None
 - a reference to the blueprint that created this setup state.
 - `first_registration` = None_
 - as blueprints can be registered multiple times with the application and not everything wants to be registered multiple times on it, this attribute can be used to figure out if the blueprint was registered in the past already.
 - `options` = None
 - a dictionary with all options that were passed to the `register_blueprint()` method.
 - `subdomain` = None
 - The subdomain that the blueprint should be active for, `None` otherwise.
 - `url_defaults` = None_
 - A dictionary with URL defaults that is added to

each and every URL that was defined with the blueprint.

- `urlprefix` = None_
- The prefix that should be used for all URLs defined on the blueprint.

Signals

Changelog

New in version 0.6.

- `signals.signals_available`
- `True` if the signaling system is available. This is the case when `blinker` is installed.

The following signals exist in Flask:

- `flask.template_rendered`
- This signal is sent when a template was successfully rendered. The signal is invoked with the instance of the template as `template` and the context as dictionary (named `_context`).

Example subscriber:

```
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.before_render_template`
- This signal is sent before template rendering process. The signal is invoked with the instance of the template as `template` and the context as dictionary (named `_context`).

Example subscriber:

```
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 before_render_template
before_render_template.connect(log_template_renders, app)
```

- `flask.request_started`
- This signal is sent when the request context is set

up, before any request processing happens. Because the request context is already bound, the subscriber can access the request with the standard global proxies such as `request`.

Example subscriber:

```
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`
- This signal is sent right before the response is sent to the client. It is passed the response to be sent named response.

Example subscriber:

```
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`
- This signal is sent when an exception happens during request processing. It is sent before the standard exception handling kicks in and even in debug mode, where no exception handling happens. The exception itself is passed to the subscriber as exception.

Example subscriber:

```
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`
- This signal is sent when the request is tearing down. This is always called, even if an exception is caused. Currently functions listening to this signal are called after the regular teardown handlers, but this is not something you can rely on.

Example subscriber:

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

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

As of Flask 0.9, this will also be passed an `exc` keyword argument that has a reference to the exception that caused the teardown if there was one.

- `flask.appcontext_tearing_down`
- This signal is sent when the app context is tearing down. This is always called, even if an exception is caused. Currently functions listening to this signal are called after the regular teardown handlers, but this is not something you can rely on.

Example subscriber:

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

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

This will also be passed an `exc` keyword argument that has a reference to the exception that caused the teardown if there was one.

- `flask.appcontext_pushed`
- This signal is sent when an application context is pushed. The sender is the application. This is usually useful for unit tests in order to temporarily hook in information. For instance it can be used to set a resource early onto the `g` object.

Example usage:

```
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
```

And in the test code:

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

Changelog

New in version 0.10.

- `flask. appcontext_popped`
- This signal is sent when an application context is popped. The sender is the application. This usually falls in line with the `appcontext_tearing_down` signal.

Changelog

New in version 0.10.

- `flask. message_flashed`
- This signal is sent when the application is flashing a message. The message is sent as message keyword argument and the category as category.

Example subscriber:

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

from flask import message_flashed
message_flashed.connect(record, app)
```

Changelog

New in version 0.10.

- class `signals. Namespace`
- An alias for `blinker.base.Namespace` if blinker is available, otherwise a dummy class that creates fake signals. This class is available for Flask extensions that want to provide the same fallback system as Flask itself.
 - `signal` (name, doc=None)
 - Creates a new signal for this namespace if blinker is available, otherwise returns a fake signal that has a send method that will do nothing but will fail with a `RuntimeError` for all other operations, including connecting.

Class-Based Views

Changelog

New in version 0.7.

- `class flask.views.View`
- Alternative way to use view functions. A subclass has to implement `dispatch_request()` which is called with the view arguments from the URL routing system. If `methods` is provided the methods do not have to be passed to the `add_url_rule()` method explicitly:

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

    def dispatch_request(self, name):
        return 'Hello %s!' % name

app.add_url_rule('/hello/<name>', view_func=MyView.as_view('myview'))
```

When you want to decorate a pluggable view you will have to either do that when the view function is created (by wrapping the return value of `as_view()`) or you can use the `decorators` attribute:

```
class SecretView(View):
    methods = ['GET']
    decorators = [superuser_required]

    def dispatch_request(self):
        ...
```

The decorators stored in the `decorators` list are applied one after another when the view function is created. Note that you can not use the class-based decorators since those would decorate the view class and not the generated view function!

- classmethod `asview` (`_name`, `*class_args`, `**class_kwargs`)
- Converts the class into an actual view function that can be used with the routing system. Internally this generates a function on the fly which will instantiate the `View` on each request and call the `dispatch_request()` method on it.

The arguments passed to `as_view()` are forwarded to the constructor of the class.

- `decorators` = ()
- The canonical way to decorate class-based views is to

decorate the return value of `as_view()`. However since this moves parts of the logic from the class declaration to the place where it's hooked into the routing system.

You can place one or more decorators in this list and whenever the view function is created the result is automatically decorated.

Changelog

New in version 0.8.

- `dispatch_request()`
- Subclasses have to override this method to implement the actual view function code. This method is called with all the arguments from the URL rule.
- `methods = None`
- A list of methods this view can handle.
- `provide_automatic_options = None`
- Setting this disables or force-enables the automatic options handling.
- class `flask.views.MethodView`
- A class-based view that dispatches request methods to the corresponding class methods. For example, if you implement a `get` method, it will be used to handle `GET` requests.

```
class CounterAPI(MethodView):
    def get(self):
        return session.get('counter', 0)

    def post(self):
        session['counter'] = session.get('counter', 0) + 1
        return 'OK'

app.add_url_rule('/counter', view_func=CounterAPI.as_view('counter'))
```

- `dispatch_request(args, *kwargs_)`
- Subclasses have to override this method to implement the actual view function code. This method is called with all the arguments from the URL rule.

URL Route Registrations

Generally there are three ways to define rules for the routing system:

- You can use the `flask.Flask.route()` decorator.
- You can use the `flask.Flask.add_url_rule()` function.
- You can directly access the underlying Werkzeug routing system which is exposed as `flask.Flask.url_map`.

Variable parts in the route can be specified with angular brackets(`/user/<username>`). By default a variable part in the URL accepts any string without a slash however a different converter can be specified as well by using `<converter:name>` .

Variable parts are passed to the view function as keyword arguments.

The following converters are available:

<code>string</code>	accepts any text without a slash (the default)
<code>int</code>	accepts integers
<code>float</code>	like <code>int</code> but for floating point values
<code>path</code>	like the default but also accepts slashes
<code>any</code>	matches one of the items provided
<code>uuid</code>	accepts UUID strings

Custom converters can be defined using `flask.Flask.url_map` .

Here are some examples:

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

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

@app.route('/post/<int:post_id>')
def show_post(post_id):
    pass
```

An important detail to keep in mind is how Flask deals with trailing slashes. The idea is to keep each URL unique so the following rules apply:

- If a rule ends with a slash and is requested without a slash by the user, the user is automatically redirected to the same page with a trailing slash attached.

- If a rule does not end with a trailing slash and the user requests the page with a trailing slash, a 404 not found is raised.

This is consistent with how web servers deal with static files. This also makes it possible to use relative link targets safely.

You can also define multiple rules for the same function. They have to be unique however. Defaults can also be specified. Here for example is a definition for a URL that accepts an optional page:

```
@app.route('/users/', defaults={'page': 1})
@app.route('/users/page/<int:page>')
def show_users(page):
    pass
```

This specifies that `/users/` will be the URL for page one and `/users/page/N` will be the URL for page `N`.

If a URL contains a default value, it will be redirected to its simpler form with a 301 redirect. In the above example, `/users/page/1` will be redirected to `/users/`. If your route handles `GET` and `POST` requests, make sure the default route only handles `GET`, as redirects can't preserve form data.

```
@app.route('/region/', defaults={'id': 1})
@app.route('/region/<id>', methods=['GET', 'POST'])
def region(id):
    pass
```

Here are the parameters that `route()` and `add_url_rule()` accept. The only difference is that with the `route` parameter the view function is defined with the decorator instead of the `view_func` parameter.

<code>rule</code>	the URL rule as string
<code>endpoint</code>	the endpoint for the registered URL rule. Flask itself assumes that the name of the view function is the name of the endpoint if not explicitly stated.
<code>view_func</code>	the function to call when serving a request to the provided endpoint. If this is not provided one can specify the function later by storing it in the <code>view_functions</code> dictionary with the endpoint as key.
<code>defaults</code>	A dictionary with defaults for this rule. See the example above for how defaults

	work.
<code>subdomain</code>	specifies the rule for the subdomain in case <code>subdomainmatching</code> is in use. If not specified the default <code>subdomain</code> is assumed.
<code>**options</code>	the options to be forwarded to the underlying <code>Rule</code> object. A change to Werkzeug is handling of method options. <code>methods</code> is a list of methods this rule should be limited to (<code>GET</code> , <code>POST</code> etc.). By default a rule just listens for <code>GET</code> (and implicitly <code>HEAD</code>). Starting with Flask 0.6, <code>OPTIONS</code> is implicitly added and handled by the standard request handling. They have to be specified as keyword arguments.

View Function Options

For internal usage the view functions can have some attributes attached to customize behavior the view function would normally not have control over. The following attributes can be provided optionally to either override some defaults to `add_url_rule()` or general behavior:

- **name:** The name of a function is by default used as endpoint. If endpoint is provided explicitly this value is used. Additionally this will be prefixed with the name of the blueprint by default which cannot be customized from the function itself.
- **methods:** If methods are not provided when the URL rule is added, Flask will look on the view function object itself if a `_methods_attribute` exists. If it does, it will pull the information for the methods from there.
- **provide_automatic_options:** if this attribute is set Flask will either force enable or disable the automatic implementation of the HTTP `OPTIONS` response. This can be useful when working with decorators that want to customize the `OPTIONS` response on a per-view basis.
- **required_methods:** if this attribute is set, Flask will always add these methods when registering a URL rule even if the methods were explicitly overridden in the `route()` call.

Full example:


```
def index():
    if request.method == 'OPTIONS':
        # custom options handling here
        ...
    return 'Hello World!'
index.provide_automatic_options = False
index.methods = ['GET', 'OPTIONS']

app.add_url_rule('/', index)
```

Changelog

New in version 0.8: The `provide_automatic_options` functionality was added.

Command Line Interface

- class `flask.cli.FlaskGroup` (`add_default_commands=True`, `create_app=None`, `add_version_option=True`, `load_dotenv=True`, `set_debug_flag=True`, `**extra`)
- Special subclass of the `AppGroup` group that supports loading more commands from the configured Flask app. Normally a developer does not have to interface with this class but there are some very advanced use cases for which it makes sense to create an instance of this.

For information as of why this is useful see [自定义脚本](#).

Parameters

-

- **`add_default_commands`** – if this is `True` then the default run and shell commands will be added.
- **`add_version_option`** – adds the `-version` option.
- **`create_app`** – an optional callback that is passed the script info and returns the loaded app.
- **`load_dotenv`** – Load the nearest `.env` and `.flaskenv` files to set environment variables. Will also change the working directory to the directory containing the first file found.
- **`set_debug_flag`** – Set the app's debug flag based on the active environment

Changelog

Changed in version 1.0: If installed, `python-dotenv` will

be used to load environment variables from `.env` and `.flaskenv` files.

- `getcommand` (`_ctx`, `name`)
- Given a context and a command name, this returns a `Command` object if it exists or returns `None`.
- `listcommands` (`_ctx`)
- Returns a list of subcommand names in the order they should appear.
- `main` (`*args`, `**kwargs`)
- This is the way to invoke a script with all the bells and whistles as a command line application. This will always terminate the application after a call. If this is not wanted, `SystemExit` needs to be caught.

This method is also available by directly calling the instance of a `Command`.

New in version 3.0: Added the `standalone_mode` flag to control the standalone mode.

```
- Parameters
-
-
```

args – the arguments that should be used for parsing. If not provided, `sys.argv[1:]` is used.

```
-
```

prog_name – the program name that should be used. By default the program name is constructed by taking the filename from `sys.argv[0]`.

```
-
```

complete_var – the environment variable that controls the bash completion support. The default is `"_<prog_name>_COMPLETE"` with `prog_name` in uppercase.

```
-
```

standalone_mode – the default behavior is to invoke the script in standalone mode. Click will then handle exceptions and convert them into error messages and the function will

never return but shut down the interpreter. If this is set to `False` they will be propagated to the caller and the return value of this function is the return value of

```
invoke() .
```

```
-
```

extra – extra keyword arguments are forwarded to the context constructor. See `Context` for more information.

- class `flask.cli. AppGroup` (`name=None`, `commands=None`, `**attrs`)
- This works similar to a regular `click.Group` but it changes the behavior of the `command()` decorator so that it automatically wraps the functions in `with_appcontext()` .

Not to be confused with `FlaskGroup` .

- `command` (`*args`, `**kwargs`)
- This works exactly like the method of the same name on a regular `click.Group` but it wraps callbacks in `with_appcontext()` unless it's disabled by passing `with_appcontext=False` .
- `group` (`*args`, `**kwargs`)
- This works exactly like the method of the same name on a regular `click.Group` but it defaults the group class to `AppGroup` .
- class `flask.cli. ScriptInfo` (`app_import_path=None`, `create_app=None`, `set_debug_flag=True`)
- Helper object to deal with Flask applications. This is usually not necessary to interface with as it's used internally in the dispatching to `click`. In future versions of Flask this object will most likely play a bigger role. Typically it's created automatically by the `FlaskGroup` but you can also manually create it and pass it onwards as `click` object.
 - `appimport_path` = `None`
 - Optionally the import path for the Flask application.
 - `createapp` = `None`
 - Optionally a function that is passed the script

info to create the instance of the application.

- `data` = None
- A dictionary with arbitrary data that can be associated with this script info.
- `load_app` ()
 - Loads the Flask app (if not yet loaded) and returns it. Calling this multiple times will just result in the already loaded app to be returned.
- `flask.cli. load_dotenv` (_path=None)
- Load “dotenv” files in order of precedence to set environment variables.

If an env var is already set it is not overwritten, so earlier files in the list are preferred over later files.

Changes the current working directory to the location of the first file found, with the assumption that it is in the top level project directory and will be where the Python path should import local packages from.

This is a no-op if `python-dotenv` is not installed.

- Parameters
 - **path** – Load the file at this location instead of searching.
- Returns
 - `True` if a file was loaded.

Changed in version 1.1.0: Returns `False` when `python-dotenv` is not installed, or when the given path isn’t a file.

Changelog

New in version 1.0.

- `flask.cli. with_appcontext` (_f)
- Wraps a callback so that it’s guaranteed to be executed with the script’s application context. If callbacks are registered directly to the `app.cli` object then they are wrapped with this function by default unless it’s disabled.
- `flask.cli. pass_script_info` (_f)
- Marks a function so that an instance of `ScriptInfo` is passed as first argument to the click callback.

- `flask.cli.runcommand` = _

- Run a local development server.

This server is for development purposes only. It does not provide the stability, security, or performance of production WSGI servers.

The reloader and debugger are enabled by default if `FLASK_ENV=development` or `FLASK_DEBUG=1`.

- `flask.cli.shellcommand` = _

- Run an interactive Python shell in the context of a given Flask application. The application will populate the default namespace of this shell according to its configuration.

This is useful for executing small snippets of management code without having to manually configure the application.

Flask 的设计思路

为什么 Flask 要这样做，而不是那样做？如果你对这点好奇，那么本节可以满足你的好奇心。当与其他框架直接进行比较时，Flask 的设计思路乍看可能显得武断并且令人吃惊，下面我们就来看看为什么在设计的时候进行这样决策。

显式的应用对象

一个基于 WSGI 的 Python web 应用必须有一个实现实际的应用的中心调用对象。在 Flask 中，中心调用对象是一个 `Flask` 类的实例。每个 Flask 应用必须创建一个该类的实例，并且把模块的名称传递给该实例。但是为什么 Flask 不自动把这些事都做好呢？

下面的代码：

```
1. from flask import Flask
2. app = Flask(__name__)
3.
4. @app.route('/')
5. def index():
6.     return 'Hello World!'
```

如果没有一个显式的应用对象，那么会是这样的：

```
1. from hypothetical_flask import route
2.
3. @route('/')
4. def index():
5.     return 'Hello World!'
```

使用对象的主要有三个原因。最重要的一个原因是显式对象可以保证实例的唯一性。有很多方法可以用单个应用对象来冒充多应用，比如维护一个应用堆栈，但是这样将会导致一些问题，这里我就不展开了。现在的问题是：一个微框架何时会需要多应用？最好的回答是当进行单元测试的时候。在进行测试时，创建一个最小应用用于测试特定的功能，是非常有用的。当这个最小应用的应用对象被删除时，将会释放其占用的所有资源。

另外当使用显式对象时，你可以继承基类（`Flask`），以便于修改特定的功能。如果不使用显式对象，那么就无从下手了。

第二个原因也很重要，那就是 Flask 需要包的名称。当你创建一个 Flask 实例时，通常会传递 `name` 作为包的名称。Flask 根据包的名称来载入也模块相关的正确资源。通过 Python 杰出的反射功能，就可以找到模板和静态文件（参见 `open_resource()`）。很显然，有其他的框架不需要任何配置就可以载入与模块相关的模板。但其前提是必须使用当前工作目录，这是一个不可靠的实现方式。当前工作目录是进程级的，如果有多个应用使用同一个进程（web 服务器可能在你不知情的情况下这样做），那么当前工作目录就不可用了。还有更糟糕的情况：许多 web 服务器把文档根目录作为当前工作目录，如果你的应用所在的目录不是文档根目录，那么就会出错。

第三个原因是“显式比隐式更好”。这个对象就是你的 WSGI 应用，你不必再记住其他东西。如果你要实现一个 WSGI 中间件，那么只要封装它就可以了（还有更好的方式，可以不丢失应用对象的引用，参见：`wsgi_app()`）。

再者，只有这样设计才能使用工厂函数来创建应用，方便单元测试和类似的工作（参见：[应用工厂](#)）。

路由系统

Flask 使用 Werkzeug 路由系统，该系统是自动根据复杂度来为路由排序的。也就是说你可以以任意顺序来声明路由，路由系统仍然能够正常工作。为什么要实现这个功能？因为当应用被切分成多个模块时，基于路由的装饰器会以乱序触发，所以这个功能是必须的。

另一点是 Werkzeug 路由系统必须确保 URL 是唯一的，并且会把模糊路由重定向到标准的 URL 。

唯一模板引擎

Flask 原生只使用 Jinja2 模板引擎。为什么不设计一个可插拔的模板引擎接口？当然，你可以在 Flask 中使用其他模板引擎，但是当前 Flask 原生只会支持 Jinja2 。将来也许 Flask 会使用其他引擎，但是永远只会绑定一个模板引擎。

模板引擎与编程语言类似，每个引擎都有自己的一套工作方式。表面上它们都看上去差不多：你把一套变量丢给引擎，然后得到字符串形式的模板。

但是相似之处也仅限于此。例如 Jinja2 有丰富的过滤系统、有一定的模板继承能力、支持从模板内或者 Python 代码内复用块（宏）、所有操作都使用 Unicode 、支持迭代模板渲染以及可配置语法等等。而比如 Genshi 基于 XML 流赋值，其模板继承基于 XPath 的能力。再如 Mako 使用类似 Python 模块的方式来处理模板。

当一个应用或者框架与一个模板引擎结合在一起的时候，事情就不只是渲染模板这么简单了。例如， Flask 使用了 Jinja2 的强大的自动转义功能。同时 Flask 也为 Jinja2 提供了在模板中操作宏的途径。

一个不失模板引擎独特性的模板抽象层本身就是一门学问，因此这不是一个 Flask 之类的微框架应该考虑的事情。

此外，只使用一个模板语言可以方便扩展。你可以使用你自己的模板语言，但扩展仍然使用 Jinja 。

我依赖所以我微

为什么 Flask 依赖两个库（ Werkzeug 和 Jinja2 ），但还是自称是微框架？为什么不可以呢？如果我们看一看 Web 开发的另一大阵营 Ruby ，那么可以发现一个与 WSGI 十分相似的协议。这个协议被称为 Rack ，除了名称不同外，基本可以视作 Ruby 版的 WSGI 。但是几乎所有 Ruby 应用都不直接使用 Rack 协议，而是使用一个相同名字的库。在 Python 中，与 Rack 库等价的有 WebOb （前身是 Paste ）和 Werkzeug 两个库。 Paste 任然可用，但是个人认为正逐步被 WebOb 取代。WebOb 和 Werkzeug 的开发初衷都是：做一个 WSGI 协议的出色实现，让其他应用受益。

正因为 Werkzeug 出色地实现了 WSGI 协议（有时候这是一个复杂的任务），使得依赖于 Werkzeug 的 Flask 受益良多。同时要感谢 Python 包管理的近期开发，包依赖问题已经解决，几乎没有理由不使用包依赖的方式。

线程本地对象

Flask 使用线程本地对象（实际是上下文本地对象，它们也支持 greenlet 上下文）来支持请求、会话和一个可以放置你自己的东西的额外对象（ `g` ）。为什么要这样做？这不是一个坏主意吗？

是的，通常情况下使用线程本地对象不是一个明智的选择，这会在不是基于线程理念的服务器上造成麻烦，并且加大大型应用的维护难度。但是 Flask 不仅是为大型应用或异步服务器设计的，Flask 还想简化和加速传统 web 应用的开发。

一些关于基于 Flask 大型应用的灵感，见文档的 [大型应用](#) 一节。

Flask 是什么，不是什么

Flask 永远不会包含数据库层，也不会有表单库或是这个方面的其它东西。 Flask 本身只是 Werkzeug 和 Jinja2 的之间的桥梁，前者实现一个合适的 WSGI 应用，后者处理模板。 当然， Flask 也绑定了一些通用的标准库包，比如 logging 。 除此之外其它所有一切都交给扩展来实现。

为什么呢？因为人们有不同的偏好和需求， Flask 不可能把所有的需求都囊括在核心里。大多数 web 应用会需要一个模板引擎。然而不是每个应用都需要一个 SQL 数据库的。

Flask 的理念是为所有应用建立一个良好的基础，其余的一切都取决于你自己或者扩展。

HTML/XHTML 常见问题

Flask 的文档和示例应用使用 HTML5 。你可能会注意到，在许多情况下，当结束标记是可选的时候，并不使用它们，这样 HTML 会更简洁且加载更迅速。因为在开发者中，有许多关于 HTML 和 XHTML 的混淆，本文档尝试回答一些主要的疑问。

XHTML 的历史

有一段时间，XHTML 横空出世，大有取代 HTML 之势。然而时至今日，鲜有真正使用 XHTML（根据 XML 规则处理的 HTML）的网站。出现这种情况的原因很多。其一是 Internet Explorer 对 XHTML 支持不完善。根据规范要求 XHTML 必须使用 `application/xhtml+xml` MIME 类型，但是 Internet Explorer 却拒绝读取这个 MIME 类型的文件。

虽然通过配置 Web 服务器来为 XHTML 提供正确的服务相对简单，但是却很少有人这么做。这可能是因为正确地使用 XHTML 是一件很痛苦的事情。

痛中之通是 XML 苛刻的（严厉且无情）错误处理。当 XML 处理中遭遇错误时，浏览器会把一个丑陋的错误消息显示给用户，而不是尝试从错误中恢复并显示出能显示的部分。web 上大多数的 (X)HTML 是基于非 XML 的模板引擎（比如 Flask 所使用的 Jinja）生成的。而这些模板引擎并不会阻止你偶然创建无效的 XHTML。也有基于 XML 的模板引擎，诸如 Kid 和流行的 Genshi，但是它们通常具有更大的运行时开销，并且用起来很不爽，因为它们必须遵守 XML 规则。

大多数用户，不管怎样，假设他们正确地使用了 XHTML。他们在文档的顶部写下一个 XHTML doctype，并且闭合了所有必要闭合的标签（在 XHTML 中 `
` 要写作 `
` 或 `
</br>`）。即使文档可以正确地通过 XHTML 验证，然而真正决定浏览器中 XHTML/HTML 处理的是前面提到的，经常不被正确设置的 MIME 类型。一旦类型错误，有效的 XHTML 会被视作无效的 HTML 处理。

XHTML 也改变了使用 JavaScript 的方式。要在 XHTML 下正确地工作，程序员不得不使用带有 XHTML 名称空间的 DOM 接口来查询 HTML 元素。

HTML5 的历史

HTML5 规范是由网络超文本应用技术工作组（WHATWG）于 2004 年开始制定的，最初的名称是“Web 应用 1.0”。WHATWG 由主要的浏览器供应商苹果、Mozilla 和 Opera 组成。HTML5 规范的目标是编写一个新的更好的 HTML 规范，该规范是基于现有浏览器的行为的，而不是不切实际的，不向后兼容的。

例如，在 HTML4 中 `<title>Hello/` 与 `<title>Hello</title>` 理论上完全相同。然而，由于人们沿用了 `<link />` 之类的 XHTML-like 标签，浏览器就会识别为 XHTML 而不是 HTML。

2007 年，W3C 以这个规范为基础，制定了一个新的 HTML 规范，也就是 HTML5。现在，随着 XHTML 2 工作组的解散，而且 HTML5 正在被所有主流浏览器供应商实现，XHTML 逐渐失去了吸引力。

HTML 对比 XHTML

下面的表格展示 HTML 4.01、XHTML 1.1 和 HTML5 简要功能比价。（不包括 XHTML 1.0，因为它已经被

XHTML 1.1 和几乎不使用的 XHTML5 代替。)

	HTML4.01	XHTML1.1	HTML5
<code><tag/value/ == <tag>value</tag></code>	✓ ₁	✗	✗
支持 <code>
</code>	✗	✓	✓ ₂
支持 <code><script/></code>	✗	✓	✗
应该解析为 <code>text/html</code>	✓	✗ ₃	✓
应该解析为 <code>application/xhtml+xml</code>	✗	✓	✗
严格的错误处理	✗	✓	✗
内联 SVG	✗	✓	✓
内联 MathML	✗	✓	✓
<code><video></code> 标记	✗	✗	✓
<code><audio></code> 标记	✗	✗	✓
新的语义标记，比如 <code><article></code>	✗	✗	✓

- [1](#)
 - 这是一个从 SGML 继承过来的隐晦的功能。由于上述的原因，它通常不被浏览器支持。
- [2](#)
 - 这用于兼容根据 XHTML 规范为 `
` 之类的标记生成的服务代码。它不应该在新代码中出现。
- [3](#)
 - XHTML 1.0 是考虑向后兼容，允许呈现为 `text/html` 的最后一个 XHTML标准。

“严格”意味着什么？

HTML5 严格地定义了解析规则，但是同时也明确地规定了浏览器如何处理解析错误。而不是像 XHTML 一样，只是简单的终止解析。有的人对有显而易见的语法错误的标记任然能够得到预想的结果感到疑惑不解（例如结尾标记缺失或者属性值未用引号包裹）。

之所以能够得到预想的结果，有的是因为大多数浏览器会宽容处理错误标记，有的是因为错误已经被指定了解决方式。以下结构在 HTML5 标准中是可选的，但一定被浏览器支持：

- 用 `<html>` 标签包裹文档。

- 把页首元素包裹在 `<head>` 里或把主体元素包裹在 `<body>` 里。
- 闭合 `<p>` 、 `` 、 `<dt>` 、 `<dd>` 、 `<tr>` 、 `<td>` 、 `<th>` 、 `<tbody>` 、 `<thead>` 或 `<tfoot>` 标签。
- 用引号包裹属性值，只要它们不含有空白字符或其特殊字符（比如 `<` 、 `>` 、 `'` 或 `"` ）。
- 布尔属性必须赋值。

这意味着下面的页面在 HTML5 中是完全有效的：

```
1. <!doctype html>
2. <title>Hello HTML5</title>
3. <div class=header>
4.   <h1>Hello HTML5</h1>
5.   <p class=tagline>HTML5 is awesome
6. </div>
7. <ul class=nav>
8.   <li><a href=/index>Index</a>
9.   <li><a href=/downloads>Downloads</a>
10.  <li><a href=/about>About</a>
11. </ul>
12. <div class=body>
13.   <h2>HTML5 is probably the future</h2>
14.   <p>
15.     There might be some other things around but in terms of
16.     browser vendor support, HTML5 is hard to beat.
17.   <dl>
18.     <dt>Key 1
19.     <dd>Value 1
20.     <dt>Key 2
21.     <dd>Value 2
22.   </dl>
23. </div>
```

HTML5 中的新技术

HTML5 增加了许多新功能，使得网络应用易写易用。

- `<audio>` 和 `<video>` 标记提供了不使用 QuickTime 或 Flash 之类的复杂附件的嵌入音频和视频的方式。
- 像 `<article>` 、 `<header>` 、 `<nav>` 和 `<time>` 之类的语义化元素，使得内容易于理解。
- `<canvas>` 标记支持强大的绘图 API ，减少了图形化展示数据时在服务器端生成图像的需求。
- 新的表单控件类型，比如 `<input type="data">` 方便用户代理输入和验证数据。
- 高级 JavaScript API ，诸如 Web Storage 、 Web Workers 、 Web Sockets 、地理位置以及离线应用。

除了上述功能之外，HTML5 还添加了许多其它的特性。Mark Pilgrim 即将出版的[Dive Into HTML5](#) 一书是 HTML5 新特性的优秀指导书。目前，并不是所有的新特性都已被浏览器支持，无论如何，请谨慎使用。

应该使用什么？

当前情况下，答案是 HTML5 。考虑到 web 浏览器最新的开发，几乎没有理由再使用XHTML 。综上所述：

- Internet Explorer （令人悲伤的是目前市场份额处于领先）对 XHTML 支持不佳。
- 许多 JavaScript 库也不支持 XHTML ，原因是它需要复杂的命名空间 API 。
- HTML5 添加了数个新特性，包括语义标记和期待已久的 `<audio>` 和 `<video>` 标记。
- 它背后获得了大多数浏览器供应商的支持。
- 它易于编写，而且更简洁。

对于大多数应用，毫无疑问使用 HTML5 比 XHTML 更好。

安全注意事项

Web 应用常常会面对各种各样的安全问题，因此要把所有问题都解决是很难的。Flask 尝试为你解决许多安全问题，但是更多的还是只能靠你自己。

跨站脚本攻击（XSS）

跨站脚本攻击是指在一个网站的环境中注入恶任意的 HTML（包括附带的JavaScript）。要防防御这种攻击，开发者需要正确地转义文本，使其不能包含恶意的 HTML 标记。更多的相关信息请参维基百科上在文章：[跨站脚本](#)。

在 Flask 中，除非显式指明不转义， Jinja2 会自动转义所有值。这样可以排除所有模板导致的 XSS 问题，但是其它地方仍需小心：

- 不使用 Jinja2 生成 HTML 。
- 在用户提交的数据上调用了 `Markup` 。
- 发送上传的 HTML ，永远不要这么做，使用 `Content-Disposition: attachment` 头部来避免这个问题。
- 发送上传的文本文件。一些浏览器基于文件开头几个字节来猜测文件的content-type ，用户可以利用这个漏洞来欺骗浏览器，通过伪装文本文件来执行 HTML 。

另一件非常重要的漏洞是不用引号包裹的属性值。虽然 Jinja2 可以通过转义 HTML来保护你免受 XSS 问题，但是仍无法避免一种情况：属性注入的 XSS 。为了免受这种攻击，必须确保在属性中使用 Jinja 表达式时，始终用单引号或双引号包裹：

```
1. <input value="{{ value }}">
```

为什么必须这么做？因为如果不这么做，攻击者可以轻易地注入自制的 JavaScript处理器。例如一个攻击者可以注入以下 HTML+JavaScript 代码：

```
1. onmouseover=alert(document.cookie)
```

当用户鼠标停放在这个输入框上时，会在警告窗口里显示 cookie 信息。一个精明的攻击者可能还会执行其它的 JavaScript 代码，而不是把 cookie 显示给用户。结合CSS 注入，攻击者甚至可以把元素填满整个页面，这样用户把鼠标停放在页面上的任何地方都会触发攻击。

有一类 XSS 问题 Jinja 的转义无法阻止。 `a` 标记的 `href` 属性可以包含一个 `javascript: URI` 。如果没有正确保护，那么当点击它时浏览器将执行其代码。

```
1. <a href="{{ value }}">click here</a>
2. <a href="javascript:alert('unsafe');">click here</a>
```

为了防止发生这种问题，需要设置 [Content Security Policy \(CSP\)](#) 响应头部。

跨站请求伪造（ CSRF ）

另一个大问题是 CSRF 。这个问题非常复杂，因此我不会在此详细展开，只是介绍 CSRF 是什么以及在理论上如何避免这个问题。

如果你的验证信息存储在 cookie 中，那么你就使用了隐式的状态管理。“已登入”这个状态由一个 cookie 控制，并且这个 cookie 在页面的每个请求中都会发送。不幸的是，在第三方站点发送的请求中也会发送这个 cookie 。如果你不注意这点，一些人可能会通过社交引擎来欺骗应用的用户在不知情的状态下做一些蠢事。

假设你有一个特定的 URL ，当你发送 `POST` 请求时会删除一个用户的资料（例如 `http://example.com/user/delete` ）。如果一个攻击者现在创建一个页面并通过页面中的 JavaScript 发送这个 post 请求，只要诱骗用户加载该页面，那么用户的资料就会被删除。

设想在有数百万的并发用户的 Facebook 上，某人放出一些小猫图片的链接。当用户访问那个页面欣赏毛茸茸的小猫图片时，他们的资料就被删除了。

那么如何预防这个问题呢？基本思路是：对于每个要求修改服务器内容的请求，应该使用一次性令牌，并存储在 cookie 里， 并且 在发送表单数据的同时附上它。在服务器再次接收数据之后，需要比较两个令牌，并确保它们相等。

为什么 Flask 没有替你做这件事？因为这应该是表单验证框架做的事，而 Flask 不包括表单验证。

JSON 安全

Flask 0.10 版和更低版本中， `jsonify()` 没序列化顶层数组为 JSON 。这是因为 ECMAScript 4 存在安全漏洞。

ECMAScript 5 关闭了这个漏洞，所以只有非常老的浏览器仍然脆弱，而且还有其他更严重的漏洞 。因此，这个行为被改变了，并且 `jsonify()` 现在支持了序列化数据。

安全头部

为了控件安全性，浏览器识别多种头部。我们推荐检查应用所使用的以下每种头部。[Flask-Talisman](#) 扩展可用于管理 HTTPS 和安全头部。

HTTP Strict Transport Security (HSTS)

告诉浏览器把所有 HTTP 请求转化为 HTTPS ，以防止 man-in-the-middle (MITM) 攻击。

```
1. response.headers['Strict-Transport-Security'] = 'max-age=31536000; includeSubDomains'
```

- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security>

Content Security Policy (CSP)

告诉浏览器哪里可以加载各种资源。这个头部应当尽可能使用，但是需要为网站定义正确的政策。一个非常严格的政策是：

```
1. response.headers['Content-Security-Policy'] = "default-src 'self'"
```

- <https://csp.withgoogle.com/docs/index.html>
- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy>

X-Content-Type-Options

强制浏览器遵守内容类型而不是尝试检测它，这可以会被滥用，以生成一个跨站脚本（XSS）攻击。

```
1. response.headers['X-Content-Type-Options'] = 'nosniff'
```

- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Content-Type-Options>

X-Frame-Options

防止外部网站把你的站点嵌入到 `iframe` 中。这样可以防止外部框架点击转化针对你的页面元素的隐藏点击，也称为“点击支持”。

```
1. response.headers['X-Frame-Options'] = 'SAMEORIGIN'
```

- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options>

X-XSS-Protection

如果请求包含类似于 JavaScript 的东西且响应的内容包含相同的数据时，浏览器将尝试通过不加载页面来防止反射的 XSS 攻击。：

```
1. response.headers['X-XSS-Protection'] = '1; mode=block'
```

- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection>

Set-Cookie 选项

这些选项可以被添加到一个 `Set-Cookie` 头部以增强其安全性。Flask 具有将其配置于会话 cookie 上的配置选项。它们也可以配置在其他 cookie 上。

- `Secure` 限制 cookies 仅用于 HTTPS 流量。
- `HttpOnly` 保护 cookies 内容不被 JavaScript 读取。
- `SameSite` 限制如何从外部网站通过请求发送 cookie。可以设置为 `'Lax'`（推荐）或者 `'Strict'`。
 - `Lax` 防止从外部网站通过有 CSRF倾向请求（比如一个表单）发送 cookie。
 - `Strict` 防止通过所

有外部请求发送 cookie ，包括常规连接。

```
1. app.config.update(
2.     SESSION_COOKIE_SECURE=True,
3.     SESSION_COOKIE_HTTPONLY=True,
4.     SESSION_COOKIE_SAMESITE='Lax',
5. )
6.
7. response.set_cookie('username', 'flask', secure=True, httponly=True, samesite='Lax')
```

指定 `Expires` 或者 `Max-Age` 选项后，将会分别在给定时间后或者当前时间加上所定义存活期后删除 cookie 。如果两个参数都没有指定，则会在关闭浏览器时删除。

```
1. # cookie expires after 10 minutes
2. response.set_cookie('snakes', '3', max_age=600)
```

对于会话 cookie 来说，如果 `session.permanent` 被设置了，那么 `PERMANENT_SESSION_LIFETIME` 会被用于设置有效期。Flask 的缺省 cookie 实现会验证加密签名不会超过这个值。降低这个值有助于缓解重播攻击，可以在稍后发送被拦截的 cookie 。

```
1. app.config.update(
2.     PERMANENT_SESSION_LIFETIME=600
3. )
4.
5. @app.route('/login', methods=['POST'])
6. def login():
7.     ...
8.     session.clear()
9.     session['user_id'] = user.id
10.    session.permanent = True
11.    ...
```

Use `itsdangerous.TimedSerializer` to sign and validate other cookie values (or any values that need secure signatures).

- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies>
- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie>

HTTP Public Key Pinning (HPKP)

告诉浏览器只使用指定的证书密钥进行服务器验证，以防止 MITM 攻击。

Warning

启用后请小心，如果密钥设置或者升级不正确则难以撤消。

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Public_Key_Pinning

Flask 中的 Unicode

Flask 与 Jinja2 、 Werkzeug 一样，文本方面完全基于 Unicode ，大多数与 web 相关的 Python 库都是这样处理文本的。如果你还不了解 Unicode ，最好先阅读[软件开发人员 Unicode 和 字符集应知应会](#) 。本文档尝试介绍一些基本的知识，以便于能够愉快地处理与 Unicode 相关的问题。

自动转换

为了提供基本的、无痛的 Unicode 支持， Flask 做了以下假设：

- 你网站上文本编码是 UTF-8 。
- 你在内部对文本始终只使用 Unicode ，除非是只有 ASCII 字符的文字字符串。
- 只要通过协议传送字节，都离不开编码和解码过程。

所以，这对你来说有什么意义？

HTTP 是基于字节的，不仅是协议，用于定位服务器文档的系统也是这样（即 URI 或 URL ）。然而，通常在 HTTP 上传送的 HTML 支持很多种字符集，并且需要在 HTTP header 中注明。为了避免不必要的复杂性， Flask 假设你发送的都是 UTF-8 编码的 Unicode， Flask 会为你完成编码工作，并设置适当的 header。

如果你使用 SQLAlchemy 或类似的 ORM 系统操作数据库，道理也是同样的。一些数据库已经使用传输 Unicode 的协议，即使没有， SQLAlchemy 或其它 ORM 也会自动处理好这个问题。

金科玉律

经验法则：如果不是处理二进制数据，一律使用 Unicode 。在 Python 2.x 中，如何使用 Unicode ？

- 只使用 ASCII 码点（基本是数字、非变音或非奇特的拉丁字母）时，可以使用常规的字符串常量（ `'Hello World'` ）。
- 如果你的字符串里有 ASCII 之外的东西，需要把这个字符串标记为 Unicode 字符串，方法是加上一个小写 `u` 作为前缀（比如 `u'Hänsel und Gretel'` ）
- 如果在 Python 文件中使用了非 Unicode 字符，那么需要告诉 Python 使用了何种编码。这里，我再次建议使用 UTF-8 。你可以在 Python 源文件的第一行或第二行写入 `# -- coding: utf-8 --` 来告知解释器你的编码类型。
- Jinja 被配置为以 UTF-8 解码模板文件，所以请同时确保你的编辑器使用 UTF-8 编码保存文件。

自助编码和解码

如果你打交道的文件系统或环境不是真正基于 Unicode 编码的话，那么使用 Unicode 接口时需要妥善地解码。比如，当从文件系统中加载一个文件，并嵌入到 Jinja2 模板时，需要按照文件的编码来解码。这里有一个老问题就是文本文件不指定其本身的编码。所以帮你自己一个忙，限定在文本文件中使用 UTF-8 。

无论如何，转入一个 Unicode 文件，可以使用内置的 `str.decode()` 方法：

```
1. def read_file(filename, charset='utf-8'):
2.     with open(filename, 'r') as f:
3.         return f.read().decode(charset)
```

把 Unicode 转换成指定的字符集（UTF-8），可以使用 `unicode.encode()` 方法：

```
1. def write_file(filename, contents, charset='utf-8'):
2.     with open(filename, 'w') as f:
3.         f.write(contents.encode(charset))
```

配置编辑器

现在的大多数编辑器默认存储为 UTF-8，但是如果你的编辑器不是，你必须重新配置。下面是设置你编辑器存储为 UTF-8 的常用做法：

- Vim：在你的 `.vimrc` 文件中加入 `set enc=utf-8`
- Emacs：要么使用 encoding cookie，要么把这段文字加入到你的 `.emacs` 文件：

```
1. (prefer-coding-system 'utf-8)
2. (setq default-buffer-file-coding-system 'utf-8)
```

- Notepad++：
 - 打开 设置 -> 首选项 ...
 - 选择“新建/缺省路径”选项卡
 - 选择“ UTF-8 无 BOM ”作为编码

同样也建议使用 Unix 的换行格式，可以在相同的面板中选择，但不是必须的。

Flask 扩展开发

Flask 作为一个微框架，不可避免地会使用第三方库。使用第三方库时，经常需要做一些重复工作。为了避免重复劳动，[PyPI](#) 提供了许多扩展。

如果你需要创建自己的扩展，那么本文可以帮助你让扩展立马上线运行。

剖析一个扩展

扩展都放在一个名如 `flasksomething` 的包中。其中的“*something*”就是扩展所要连接的库的名称。例如假设你要为 *Flask* 添加 `_simplexml` 库的支持，那么扩展的包名称就应该是 `flask_simplexml`。

但是，真正扩展的名称（可读名称）应当形如“*Flask-SimpleXML*”。请确保名称中包含“*Flask*”，并且注意大小写。这样用户就可以在他们的 `setup.py` 文件中注册依赖。

但是扩展具体是怎么样的呢？一个扩展必须保证可以同时多个 *Flask* 应用中工作。这是必要条件，因为许多人为了进行单元测试，会使用类似 [应用工厂](#) 模式来创建应用并且需要支持多套配置。因此，你的应用支持这种行为非常重要。

最重要的是，扩展必须与一个 `setup.py` 文件一起分发，并且在 *PyPI* 上注册。同时，用于开发的检出链接也应该能工作，以便于在 *virtualenv* 中安装开发版本，而不是手动下载库。

Flask 扩展必须使用 *BSD* 或 *MIT* 或更自由的许可证来许可，这样才能被添加进 *Flask* 扩展注册表。请记住，*Flask* 扩展注册表是比较稳健的，并且扩展在发布前会进行预审是否符合要求。

“ Hello Flaskext! ”

好吧，让我们开展创建一个 *Flask* 扩展。这个扩展的用途是提供最基本的 *SQLite3*支持。

首先创建如下结构的文件夹和文件：

```
1. flask-sqlite3/  
2.     flask_sqlite3.py  
3.     LICENSE  
4.     README
```

以下是最重要的文件及其内容：

setup.py

接下来 `setup.py` 是必需的，该文件用于安装你的 *Flask* 扩展。文件内容如下：

```
1. """  
2. Flask-SQLite3  
3. -----  
4.
```

```

5. This is the description for that library
6. """
7. from setuptools import setup
8.
9.
10. setup(
11.     name='Flask-SQLite3',
12.     version='1.0',
13.     url='http://example.com/flask-sqlite3/',
14.     license='BSD',
15.     author='Your Name',
16.     author_email='your-email@example.com',
17.     description='Very short description',
18.     long_description=__doc__,
19.     py_modules=['flask_sqlite3'],
20.     # if you would be using a package instead use packages instead
21.     # of py_modules:
22.     # packages=['flask_sqlite3'],
23.     zip_safe=False,
24.     include_package_data=True,
25.     platforms='any',
26.     install_requires=[
27.         'Flask'
28.     ],
29.     classifiers=[
30.         'Environment :: Web Environment',
31.         'Intended Audience :: Developers',
32.         'License :: OSI Approved :: BSD License',
33.         'Operating System :: OS Independent',
34.         'Programming Language :: Python',
35.         'Topic :: Internet :: WWW/HTTP :: Dynamic Content',
36.         'Topic :: Software Development :: Libraries :: Python Modules'
37.     ]
38. )

```

代码相当多，但是你可以从现有的扩展中直接复制/粘贴，并修改相应内容。

flask_sqlite3.py

这个文件是你的扩展的具体实现。但是一个扩展到底是怎样的？最佳实践是什么？继续阅读吧。

初始化扩展

许多扩展会需要某种类型的初始化步骤。例如，假设一个应用像文档中建议的一样（[使用 SQLite 3](#)）正在连接到 SQLite。那么，扩展如何获知应用对象的名称？

相当简单：你把名称传递给扩展。

推荐两种初始化扩展的方式：

初始化函数：

如果你的扩展名为 `helloworld`，那么你可能有一个名为 `init_helloworld(app[, extra_args])` 的函数。该函数用来为应用初始化扩展，它可以在处理器之前或之后。

初始化类：

初始化类与初始化函数的工作方式大致相同，区别是类在以后可以进一步改动。例如，查看一下 [OAuth 扩展](#) 的工作方式：有一个 `OAuth` 对象提供一些辅助函数（比如 `OAuth.remote_app`）来创建使用 `OAuth` 的远程应用的引用。

使用哪种方式取决于你。对于 `SQLite 3` 扩展，我们会使用基于类的方式，因为这样可以提供给用户一个用于打开和关闭数据库连接的对象。

当设计类时，重要的一点是使用它们在模块层易于复用。也就是说，对象本身在任何情况下不应存储任何应用的特定状态，而必须可以在不同的应用之间共享。

扩展的代码

以下是 `flask_sqlite3.py` 的内容，可以复制/粘贴：

```
1. import sqlite3
2. from flask import current_app, _app_ctx_stack
3.
4.
5. class SQLite3(object):
6.     def __init__(self, app=None):
7.         self.app = app
8.         if app is not None:
9.             self.init_app(app)
10.
11.     def init_app(self, app):
12.         app.config.setdefault('SQLITE3_DATABASE', ':memory:')
13.         app.teardown_appcontext(self.teardown)
14.
15.     def connect(self):
16.         return sqlite3.connect(current_app.config['SQLITE3_DATABASE'])
17.
18.     def teardown(self, exception):
19.         ctx = _app_ctx_stack.top
20.         if hasattr(ctx, 'sqlite3_db'):
21.             ctx.sqlite3_db.close()
22.
23.     @property
24.     def connection(self):
25.         ctx = _app_ctx_stack.top
26.         if ctx is not None:
27.             if not hasattr(ctx, 'sqlite3_db'):
28.                 ctx.sqlite3_db = self.connect()
29.             return ctx.sqlite3_db
```

那么这是这些代码的含义是什么：

- `init` 方法接收应用对象，该对象是可选的。如果提供了该对象，那么就调用 `init_app` 。
- `init_app` 方法使得 `SQLite3` 对象不需要应用对象就可以实例化。这个方法支持工厂模式来创建应用。`init_app` 会配置数据库。如果不提供配置，默认配置为内存数据库。此外，`init_app` 方法附加了 `teardown` 处理器。
- 接下来，我们定义了 `connect` 方法来打开一个数据库连接。
- 最后，我们添加一个 `connection` 属性，首次访问时打开数据库连接，并把它存储在环境中。这也是处理资源的推荐方式：在资源第一次使用时获取资源，即惰性获取。

注意这里，我们把数据库连接通过 `_app_ctx_stack.top` 附加到应用环境的栈顶。扩展应该使用上下文的栈顶来存储它们自己的信息，并使用足够复杂的名称。

那么为什么我们决定在此使用基于类的方法？因为我们的扩展是这样使用的：

```
1. from flask import Flask
2. from flask_sqlite3 import SQLite3
3.
4. app = Flask(__name__)
5. app.config.from_pyfile('the-config.cfg')
6. db = SQLite3(app)
```

你可以在视图中这样使用数据库：

```
1. @app.route('/')
2. def show_all():
3.     cur = db.connection.cursor()
4.     cur.execute(...)
```

同样，如果在请求之外，可以通过压入应用情境的方法使用数据库：

```
1. with app.app_context():
2.     cur = db.connection.cursor()
3.     cur.execute(...)
```

在 `with` 块的末尾，拆卸处理器会自动执行。

另外，`init_app` 方法用于在创建应用时支持工厂模式：

```
1. db = SQLite3()
2. # Then later on.
3. app = create_app('the-config.cfg')
4. db.init_app(app)
```

记住已审核的 Flask 扩展必须支持用工厂模式来创建应用（下面会解释）。

`init_app` 的注意事项

如你所见，`init_app` 不分配 `app` 到 `self` 。这是故意的！基于类的 Flask 扩展必须只在应用传递到构造函数时才在对象上存储应用。这告诉扩展：我对使用多个应用没有兴趣。

当扩展需要找到当前应用，且没有一个指向当前应用的引用时，必须使用 `current_app` 环境局部变量或用一种你可以显式传递应用的方法更改 API 。

使用 `_app_ctx_stack`

在上面的例子中，在每个请求之前，一个 `sqlite3db` 变量被分配到 `_app_ctx_stack.top` 。在一个视图函数中，这个变量可以使用 `SQLite3` 的属性 `connection` 来访问。在请求解散时，`sqlite3_db` 连接被关闭。通过使用这个模式，在请求持续的期间，可以访问 相同_ 的 sqlite3 数据库连接。

学习借鉴

本文只是谈了一些扩展开发的皮毛。如果想要深入，那么查看 [Flask 扩展注册表](#) 上已有的扩展是明智的。如果你感到迷失，还可以通过 [邮件列表](#) 和 [IRC 频道](#) 学习到优秀的 APIs 。尤其当你要开发一个全新的扩展时，建议先多看多问多听，这样不仅可以知道别人的需求，同时也避免闭门造车。

谨记：设计优秀的 API 是艰难的。因此请先在邮件列表里介绍你的项目，让其他开发者在 API 设计上助你一臂之力。

最好的 Flask 扩展是那些共享 API 智慧的扩展，因此越早共享越有效。

已审核的扩展

Flask 有已审核的扩展的概念。已审核的扩展会被视作 Flask 的一部分来测试，以保证扩展在新版本发布时不会出问题。这些已审核的扩展会在 [Flask 扩展注册表](#) 中列出，并有相应的标记。如果你想要自己的扩展通过审核，请遵循以下的指导方针：

- 一个已审核的 Flask 扩展需要一个维护者。如果一个扩展作者想要放弃项目，那么项目应该寻找一个新的维护者，包括移交完整的源码托管和 PyPI 访问。如果找不到新的维护者，请赋予 Flask 核心团队访问权限。
- 一个已审核的 Flask 扩展必须提供一个名如 `flask_extensionname` 的包或模块。
- 它必须带有测试套件，套件可以使用 `make test` 或者 `python setup.py test` 来调用。如果是使用 `make test` 调用的测试套件，那么必须保证所有的依赖可以自动安装。如果是使用 `python setup.pytest` 调用的测试套件，那么测试的依赖可以在 `setup.py` 文件中定义。测试套件分发的必要组成部分。
- 已审核的扩展的 API 可以通过下面特性的检查：

- 一个已审核的扩展必须支持在同一个 Python 进程中运行的多个应用。
- 它必须支持使用工厂模式创建应用

- 许可协议必须是 BSD/MIT/WTFPL 协议。

- 官方扩展的命名模式是 *Flask-ExtensionName* 或 *ExtensionName-Flask* 。
- 已审核的扩展必须在 `setup.py` 文件里定义所有的依赖关系，除非在PyPI 上不可用。
- 扩展的文档必须使用来自 [官方 Pallets 主题](#) 的 `flask` 主题。
- `setup.py` 描述（即 PyPI 描述）必须链接到文档、网站（如果有的话），并且必须有自动安装开发版本的链接（`PackageName==dev`）。
- 安装脚本中的 `zip_safe` 标志必须被设置为 `False`，即使扩展对于压缩是安全的。
- 现行扩展必须支持 Python 2.7 和 Python 3.4（及以后版本）。

Pocoo 风格指南

所有 Pocoo 项目都遵循 Pocoo 风格指南， Flask 项目也不例外。 Flask 补丁必须遵循这个指南，同时也推荐 Flask 扩展遵循这个指南。

一般而言， Pocoo 风格指南遵循 [PEP 8](#)，有一些小差异和扩充。

总体布局

- 缩进：
- 4个空格。不使用制表符，没有例外。
- 最大行长：
- 软限制为 79 个字符，不超过 84 个字符。尝试合理放置 *break*、*continue*和 *return* 声明来避免代码过度嵌套。
- 续行：
- 可以使用反斜杠来续行，续行应对齐最后一个点号或等于号，或者缩进四个空格：

```
1. this_is_a_very_long(function_call, 'with many parameters') \
2.     .that_returns_an_object_with_an_attribute
3.
4. MyModel.query.filter(MyModel.scalar > 120) \
5.     .order_by(MyModel.name.desc()) \
6.     .limit(10)
```

如果你在括号内的换行，那么续行应对齐括号：

```
1. this_is_a_very_long(function_call, 'with many parameters',
2.                     23, 42, 'and even more')
```

对于有许多元素的元组或列表，在起始括号后立即换行：

```
1. items = [
2.     'this is the first', 'set of items', 'with more items',
3.     'to come in this line', 'like this'
4. ]
```

- 空行：
- 顶层函数和类由两个空行分隔，其它一个空行。不要使用过多空行来分隔代码逻辑段。例如：

```
1. def hello(name):
2.     print 'Hello %s!' % name
3.
```

```

4.
5. def goodbye(name):
6.     print 'See you %s.' % name
7.
8.
9. class MyClass(object):
10.     """This is a simple docstring"""
11.
12.     def __init__(self, name):
13.         self.name = name
14.
15.     def get_annoying_name(self):
16.         return self.name.upper() + '!!!!111'

```

表达式和语句

- 常规空格规则：
 -
 - 不是单词的一元运算符不使用空格（例如： `-` 、 `~` 等等），在圆括号也是这样。
 - 用空格包围二元运算符。

对：

```

1. exp = -1.05
2. value = (item_value / item_count) * offset / exp
3. value = my_list[index]
4. value = my_dict['key']

```

错：

```

1. exp = - 1.05
2. value = ( item_value / item_count ) * offset / exp
3. value = (item_value/item_count)*offset/exp
4. value=( item_value/item_count ) * offset/exp
5. value = my_list[ index ]
6. value = my_dict ['key']

```

- 禁止 Yoda 语句：
- 永远不要用变量来比较常量，而是用常量来比较变量：

对：

```

1. if method == 'md5':
2.     pass

```

错：

```
1. if 'md5' == method:
2.     pass
```

- 比较：
 - 针对任意类型使用 `==` 和 `!=`
 - 针对单一类型使用 `is` 和 `is not`（例如：`foo is not None`）
 - 永远不要与 `True` 或 `False` 作比较（例如永远不要写 `foo == False`，而应当写 `not foo`）
- 排除检验：
- 使用 `foo not in bar` 而不是 `not foo in bar`
- 实例检验：
- 使用 `isinstance(a, C)` 而不是 `type(A) is C`，但是通常应当避免检验实例，而应当检验特性。

命名约定

- 类名：`CamelCase`，缩写词大写（`HTTPWriter` 而不是 `HttpWriter`）
- 变量名：`lowercase_with_underscores`
- 方法和函数名：`lowercase_with_underscores`
- 常量：`UPPERCASE_WITH_UNDERSCORES`
- 预编译正则表达式：`name_re`

被保护的成员以单个下划线作为前缀，混合类则使用双下划线。

如果使用关键字作为类的名称，那么在名称末尾添加下划线。与内置构件冲突是允许的，请一定不要用在变量名后添加下划线的方式解决冲突。如果函数需要访问一个隐蔽的内置构件，请重新绑定内置构件到一个不同的名字。

- 函数和方法参数：
 - 类方法：`cls` 作为第一个参数
 - 实例方法：`self` 作为第一个参数
 - 用于属性的 `lambda` 表达式应该把第一个参数替换为 `x`，像 `display_name = property(lambda x: x.real_name or x.username)` 中一样

文档字符串

- 文档字符串约定：
- 所有的文档字符串为 Sphinx 可理解的 reStructuredText 格式。它们的形态因行数不同而不同。如果只有一行，三引号闭合在同一行，否则开头的三引号与文本在同一行，结尾的三引号独立一行：

```

1. def foo():
2.     """This is a simple docstring"""
3.
4.
5. def bar():
6.     """This is a longer docstring with so much information in there
7.     that it spans three lines. In this case the closing triple quote
8.     is on its own line.
9.     """

```

- 模块头：
- 模块头包含一个 utf-8 编码声明（即使没有使用非 ASCII 字符，也始终推荐这么做）和一个标准的文档字符串：

```

1. # -*- coding: utf-8 -*-
2. """
3.     package.module
4.     ~~~~~
5.
6.     A brief description goes here.
7.
8.     :copyright: (c) YEAR by AUTHOR.
9.     :license: LICENSE_NAME, see LICENSE_FILE for more details.
10. """

```

谨记使用合适的版权和许可证文件以利于通过 Flask 扩展审核。

注释

注释的规则与文档字符串类似。两者都使用 reStructuredText 格式。如果一个注释被用于一个说明类属性，在起始的井号（`#`）后加一个冒号：

```

1. class User(object):
2.     #: the name of the user as unicode string
3.     name = Column(String)
4.     #: the sha1 hash of the password + inline salt
5.     pw_hash = Column(String)

```

Upgrading to Newer Releases

Flask itself is changing like any software is changing over time. Most of the changes are the nice kind, the kind where you don't have to change anything in your code to profit from a new release.

However every once in a while there are changes that do require some changes in your code or there are changes that make it possible for you to improve your own code quality by taking advantage of new features in Flask.

This section of the documentation enumerates all the changes in Flask from release to release and how you can change your code to have a painless updating experience.

Use the **pip** command to upgrade your existing Flask installation by providing the `--upgrade` parameter:

```
1. $ pip install --upgrade Flask
```

Version 0.12

Changes to `send_file`

The `filename` is no longer automatically inferred from file-like objects. This means that the following code will no longer automatically have `X-Sendfile` support, etag generation or MIME-type guessing:

```
1. response = send_file(open('/path/to/file.txt'))
```

Any of the following is functionally equivalent:

```
1. fname = '/path/to/file.txt'
2.
3. # Just pass the filepath directly
4. response = send_file(fname)
5.
6. # Set the MIME-type and ETag explicitly
7. response = send_file(open(fname), mimetype='text/plain')
8. response.set_etag(...)
9.
10. # Set `attachment_filename` for MIME-type guessing
11. # ETag still needs to be manually set
12. response = send_file(open(fname), attachment_filename=fname)
13. response.set_etag(...)
```

The reason for this is that some file-like objects have an invalid or even misleading

`name` attribute. Silently swallowing errors in such cases was not a satisfying solution.

Additionally the default of falling back to `application/octet-stream` has been restricted. If Flask can't guess one or the user didn't provide one, the function fails if no filename information was provided.

Version 0.11

0.11 is an odd release in the Flask release cycle because it was supposed to be the 1.0 release. However because there was such a long lead time up to the release we decided to push out a 0.11 release first with some changes removed to make the transition easier. If you have been tracking the master branch which was 1.0 you might see some unexpected changes.

In case you did track the master branch you will notice that **flask -app** is removed now. You need to use the environment variable to specify an application.

Debugging

Flask 0.11 removed the `debug_log_format` attribute from Flask applications. Instead the new `LOGGER_HANDLER_POLICY` configuration can be used to disable the default log handlers and custom log handlers can be set up.

Error handling

The behavior of error handlers was changed. The precedence of handlers used to be based on the decoration/call order of `errorhandler()` and `register_error_handler()`, respectively. Now the inheritance hierarchy takes precedence and handlers for more specific exception classes are executed instead of more general ones. See [错误处理](#) for specifics.

Trying to register a handler on an instance now raises `ValueError`.

Note

There used to be a logic error allowing you to register handlers only for exception *instances*. This was unintended and plain wrong, and therefore was replaced with the intended behavior of registering handlers only using exception classes and HTTP error codes.

Templating

The `render_template_string()` function has changed to autoescape template variables by default. This better matches the behavior of `render_template()`.

Extension imports

Extension imports of the form `flask.ext.foo` are deprecated, you should use `flask_foo`.

The old form still works, but Flask will issue a `flask.exthook.ExtDeprecationWarning` for each extension you import the old way. We also provide a migration utility called `flask-ext-migrate` that is supposed to automatically rewrite your imports for this.

Version 0.10

The biggest change going from 0.9 to 0.10 is that the cookie serialization format changed from pickle to a specialized JSON format. This change has been done in order to avoid the damage an attacker can do if the secret key is leaked. When you upgrade you will notice two major changes: all sessions that were issued before the upgrade are invalidated and you can only store a limited amount of types in the session. The new sessions are by design much more restricted to only allow JSON with a few small extensions for tuples and strings with HTML markup.

In order to not break people's sessions it is possible to continue using the old session system by using the `Flask-OldSessions` extension.

Flask also started storing the `flask.g` object on the application context instead of the request context. This change should be transparent for you but it means that you now can store things on the `g` object when there is no request context yet but an application context. The old `flask.Flask.request_globals_class` attribute was renamed to `flask.Flask.app_ctx_globals_class`.

Version 0.9

The behavior of returning tuples from a function was simplified. If you return a tuple it no longer defines the arguments for the response object you're creating, it's now always a tuple in the form `(response, status, headers)` where at least one item has to be provided. If you depend on the old behavior, you can add it easily by subclassing Flask:

```
1. class TraditionalFlask(Flask):
2.     def make_response(self, rv):
3.         if isinstance(rv, tuple):
4.             return self.response_class(*rv)
5.         return Flask.make_response(self, rv)
```

If you maintain an extension that was using `_request_ctx_stack` before, please consider changing to `_app_ctx_stack` if it makes sense for your extension. For instance, the app context stack makes sense for extensions which connect to databases. Using the app context stack instead of the request context stack will make extensions more readily handle use cases outside of requests.

Version 0.8

Flask introduced a new session interface system. We also noticed that there was a naming collision between `flask.session` the module that implements sessions and `flask.session` which is the global session object. With that introduction we moved the implementation details for the session system into a new module called `flask.sessions`. If you used the previously undocumented session support we urge you to upgrade.

If invalid JSON data was submitted Flask will now raise a `BadRequest` exception instead of letting the default `ValueError` bubble up. This has the advantage that you no longer have to handle that error to avoid an internal server error showing up for the user. If you were catching this down explicitly in the past as `ValueError` you will need to change this.

Due to a bug in the test client Flask 0.7 did not trigger teardown handlers when the test client was used in a with statement. This was since fixed but might require some changes in your test suites if you relied on this behavior.

Version 0.7

In Flask 0.7 we cleaned up the code base internally a lot and did some backwards incompatible changes that make it easier to implement larger applications with Flask. Because we want to make upgrading as easy as possible we tried to counter the problems arising from these changes by providing a script that can ease the transition.

The script scans your whole application and generates a unified diff with changes it assumes are safe to apply. However as this is an automated tool it won't be able to find all use cases and it might miss some. We internally spread a lot of deprecation warnings all over the place to make it easy to find pieces of code that it was unable to upgrade.

We strongly recommend that you hand review the generated patchfile and only apply the chunks that look good.

If you are using git as version control system for your project we recommend applying the patch with `path -p1 < patchfile.diff` and then using the interactive commit feature to only apply the chunks that look good.

To apply the upgrade script do the following:

- Download the script: [flask-07-upgrade.py](#)
- Run it in the directory of your application:

```
1. $ python flask-07-upgrade.py > patchfile.diff
```

- Review the generated patchfile.
- Apply the patch:


```
1. $ patch -p1 < patchfile.diff
```

- If you were using per-module template folders you need to move some templates around. Previously if you had a folder named `templates` next to a blueprint named `admin` the implicit template path automatically was `admin/index.html` for a template file called `templates/index.html`. This no longer is the case. Now you need to name the template `templates/admin/index.html`. The tool will not detect this so you will have to do that on your own.

Please note that deprecation warnings are disabled by default starting with Python 2.7. In order to see the deprecation warnings that might be emitted you have to enable them with the `warnings` module.

If you are working with windows and you lack the `patch` command line utility you can get it as part of various Unix runtime environments for windows including cygwin, msysgit or ming32. Also source control systems like svn, hg or git have builtin support for applying unified diffs as generated by the tool. Check the manual of your version control system for more information.

Bug in Request Locals

Due to a bug in earlier implementations the request local proxies now raise a `RuntimeError` instead of an `AttributeError` when they are unbound. If you caught these exceptions with `AttributeError` before, you should catch them with `RuntimeError` now.

Additionally the `send_file()` function is now issuing deprecation warnings if you depend on functionality that will be removed in Flask 0.11. Previously it was possible to use etags and mimetypes when file objects were passed. This was unreliable and caused issues for a few setups. If you get a deprecation warning, make sure to update your application to work with either filenames there or disable etag attaching and attach them yourself.

Old code:

```
1. return send_file(my_file_object)
2. return send_file(my_file_object)
```

New code:

```
1. return send_file(my_file_object, add_etags=False)
```

Upgrading to new Teardown Handling

We streamlined the behavior of the callbacks for request handling. For things that modify the response the `after_request()` decorators continue to work as expected, but for things that absolutely must happen at the end of request we introduced the

new `teardown_request()` decorator. Unfortunately that change also made after-request work differently under error conditions. It's not consistently skipped if exceptions happen whereas previously it might have been called twice to ensure it is executed at the end of the request.

If you have database connection code that looks like this:

```
1. @app.after_request
2. def after_request(response):
3.     g.db.close()
4.     return response
```

You are now encouraged to use this instead:

```
1. @app.teardown_request
2. def after_request(exception):
3.     if hasattr(g, 'db'):
4.         g.db.close()
```

On the upside this change greatly improves the internal code flow and makes it easier to customize the dispatching and error handling. This makes it now a lot easier to write unit tests as you can prevent closing down of database connections for a while. You can take advantage of the fact that the teardown callbacks are called when the response context is removed from the stack so a test can query the database after request handling:

```
1. with app.test_client() as client:
2.     resp = client.get('/')
3.     # g.db is still bound if there is such a thing
4.
5. # and here it's gone
```

Manual Error Handler Attaching

While it is still possible to attach error handlers to `Flask.error_handlers` it's discouraged to do so and in fact deprecated. In general we no longer recommend custom error handler attaching via assignments to the underlying dictionary due to the more complex internal handling to support arbitrary exception classes and blueprints. See `Flask.errorhandler()` for more information.

The proper upgrade is to change this:

```
1. app.error_handlers[403] = handle_error
```

Into this:

```
1. app.register_error_handler(403, handle_error)
```

Alternatively you should just attach the function with a decorator:

```
1. @app.errorhandler(403)
2. def handle_error(e):
3.     ...
```

(Note that `register_error_handler()` is new in Flask 0.7)

Blueprint Support

Blueprints replace the previous concept of “Modules” in Flask. They provide better semantics for various features and work better with large applications. The update script provided should be able to upgrade your applications automatically, but there might be some cases where it fails to upgrade. What changed?

- Blueprints need explicit names. Modules had an automatic name guessing scheme where the shortname for the module was taken from the last part of the import module. The upgrade script tries to guess that name but it might fail as this information could change at runtime.
- Blueprints have an inverse behavior for `url_for()`. Previously `.foo` told `url_for()` that it should look for the endpoint `foo` on the application. Now it means “relative to current module”. The script will inverse all calls to `url_for()` automatically for you. It will do this in a very eager way so you might end up with some unnecessary leading dots in your code if you’re not using modules.
- Blueprints do not automatically provide static folders. They will also no longer automatically export templates from a folder called `templates` next to their location however but it can be enabled from the constructor. Same with static files: if you want to continue serving static files you need to tell the constructor explicitly the path to the static folder (which can be relative to the blueprint’s module path).
- Rendering templates was simplified. Now the blueprints can provide template folders which are added to a general template search path. This means that you need to add another subfolder with the blueprint’s name into that folder if you want `blueprintname/template.html` as the template name.

If you continue to use the `Module` object which is deprecated, Flask will restore the previous behavior as good as possible. However we strongly recommend upgrading to the new blueprints as they provide a lot of useful improvement such as the ability to attach a blueprint multiple times, blueprint specific error handlers and a lot more.

Version 0.6

Flask 0.6 comes with a backwards incompatible change which affects the order of after-

request handlers. Previously they were called in the order of the registration, now they are called in reverse order. This change was made so that Flask behaves more like people expected it to work and how other systems handle request pre- and post-processing. If you depend on the order of execution of post-request functions, be sure to change the order.

Another change that breaks backwards compatibility is that context processors will no longer override values passed directly to the template rendering function. If for example `request` is a variable passed directly to the template, the default context processor will not override it with the current request object. This makes it easier to extend context processors later to inject additional variables without breaking existing template not expecting them.

Version 0.5

Flask 0.5 is the first release that comes as a Python package instead of a single module. There were a couple of internal refactorings so if you depend on undocumented internal details you probably have to adapt the imports.

The following changes may be relevant to your application:

- autoescaping no longer happens for all templates. Instead it is configured to only happen on files ending with `.html`, `.htm`, `.xml` and `.xhtml`. If you have templates with different extensions you should override the `select_jinja_autoescape()` method.
- Flask no longer supports zipped applications in this release. This functionality might come back in future releases if there is demand for this feature. Removing support for this makes the Flask internal code easier to understand and fixes a couple of small issues that made debugging harder than necessary.
- The `create_jinja_loader` function is gone. If you want to customize the Jinja loader now, use the `create_jinja_environment()` method instead.

Version 0.4

For application developers there are no changes that require changes in your code. In case you are developing on a Flask extension however, and that extension has a unittest-mode you might want to link the activation of that mode to the new `TESTING` flag.

Version 0.3

Flask 0.3 introduces configuration support and logging as well as categories for flashing messages. All these are features that are 100% backwards compatible but you might want to take advantage of them.

Configuration Support

The configuration support makes it easier to write any kind of application that requires some sort of configuration. (Which most likely is the case for any application out there).

If you previously had code like this:

```
1. app.debug = DEBUG
2. app.secret_key = SECRET_KEY
```

You no longer have to do that, instead you can just load a configuration into the config object. How this works is outlined in [配置管理](#).

Logging Integration

Flask now configures a logger for you with some basic and useful defaults. If you run your application in production and want to profit from automatic error logging, you might be interested in attaching a proper log handler. Also you can start logging warnings and errors into the logger when appropriately. For more information on that, read [应用错误处理](#).

Categories for Flash Messages

Flash messages can now have categories attached. This makes it possible to render errors, warnings or regular messages differently for example. This is an opt-in feature because it requires some rethinking in the code.

Read all about that in the [消息闪现](#) pattern.

更新日志

Version 1.1.1

Released 2019-07-08

- The `flask.json_available` flag was added back for compatibility with some extensions. It will raise a deprecation warning when used, and will be removed in version 2.0.0. [#3288](#)

Version 1.1.0

Released 2019-07-04

- Bump minimum Werkzeug version to `>= 0.15`.
- Drop support for Python 3.4.
- Error handlers for `InternalServerError` or `500` will always be passed an instance of `InternalServerError`. If they are invoked due to an unhandled exception, that original exception is now available as `e.original_exception` rather than being passed directly to the handler. The same is true if the handler is for the base `HTTPException`. This makes error handler behavior more consistent. [#3266](#)
 - `Flask.finalize_request()` is called for all unhandled exceptions even if there is no `500` error handler.
- `Flask.logger` takes the same name as `Flask.name` (the value passed as `Flask(import_name)`). This reverts 1.0's behavior of always logging to `"flask.app"`, in order to support multiple apps in the same process. A warning will be shown if old configuration is detected that needs to be moved. [#2866](#)
- `flask.RequestContext.copy()` includes the current session object in the request context copy. This prevents `session` pointing to an out-of-date object. [#2935](#)
- Using built-in `RequestContext`, unprintable Unicode characters in `Host` header will result in a HTTP 400 response and not HTTP 500 as previously. [#2994](#)
- `send_file()` supports `PathLike` objects as described in PEP 0519, to support `pathlib` in Python 3. [#3059](#)
- `send_file()` supports `BytesIO` partial content. [#2957](#)
- `open_resource()` accepts the `"rt"` file mode. This still does the same thing as `"r"`. [#3163](#)
- The `MethodView.methods` attribute set in a base class is used by subclasses. [#3138](#)

- `Flask.jinja_options` is a `dict` instead of an `ImmutableDict` to allow easier configuration. Changes must still be made before creating the environment. [#3190](#)
- Flask's `JSONMixin` for the request and response wrappers was moved into Werkzeug. Use Werkzeug's version with Flask-specific support. This bumps the Werkzeug dependency to `>= 0.15`. [#3125](#)
- The `flask` command entry point is simplified to take advantage of Werkzeug 0.15's better reloader support. This bumps the Werkzeug dependency to `>= 0.15`. [#3022](#)
- Support `static_url_path` that ends with a forward slash. [#3134](#)
- Support empty `static_folder` without requiring setting an empty `static_url_path` as well. [#3124](#)
- `jsonify()` supports `dataclasses.dataclass` objects. [#3195](#)
- Allow customizing the `Flask.url_map_class` used for routing. [#3069](#)
- The development server port can be set to 0, which tells the OS to pick an available port. [#2926](#)
- The return value from `cli.load_dotenv()` is more consistent with the documentation. It will return `False` if python-dotenv is not installed, or if the given path isn't a file. [#2937](#)
- Signaling support has a stub for the `connect_via` method when the Blinker library is not installed. [#3208](#)
- Add an `-extra-files` option to the `flask run` CLI command to specify extra files that will trigger the reloader on change. [#2897](#)
- Allow returning a dictionary from a view function. Similar to how returning a string will produce a `text/html` response, returning a dict will call `jsonify` to produce a `application/json` response. [#3111](#)
- Blueprints have a `cli` Click group like `app.cli`. CLI commands registered with a blueprint will be available as a group under the `flask` command. [#1357](#).
- When using the test client as a context manager (`with client:`), all preserved request contexts are popped when the block exits, ensuring nested contexts are cleaned up correctly. [#3157](#)
- Show a better error message when the view return type is not supported. [#3214](#)
- `flask.testing.make_test_environ_builder()` has been deprecated in favour of a new class `flask.testing.EnvironBuilder`. [#3232](#)
- The `flask run` command no longer fails if Python is not built with SSL support. Using the `-cert` option will show an appropriate error message. [#3211](#)

- URL matching now occurs after the request context is pushed, rather than when it's created. This allows custom URL converters to access the app and request contexts, such as to query a database for an id. [#3088](#)

Version 1.0.4

Released 2019-07-04

- The key information for `BadRequestKeyError` is no longer cleared outside debug mode, so error handlers can still access it. This requires upgrading to Werkzeug 0.15.5. [#3249](#)
- `send_file` url quotes the ":" and "/" characters for more compatible UTF-8 filename support in some browsers. [#3074](#)
- Fixes for PEP451 import loaders and pytest 5.x. [#3275](#)
- Show message about dotenv on stderr instead of stdout. [#3285](#)

Version 1.0.3

Released 2019-05-17

- `send_file()` encodes filenames as ASCII instead of Latin-1 (ISO-8859-1). This fixes compatibility with Gunicorn, which is stricter about header encodings than PEP 3333. [#2766](#)
- Allow custom CLIs using `FlaskGroup` to set the debug flag without it always being overwritten based on environment variables. [#2765](#)
- `flask --version` outputs Werkzeug's version and simplifies the Python version. [#2825](#)
- `send_file()` handles an `attachment_filename` that is a native Python 2 string (bytes) with UTF-8 coded bytes. [#2933](#)
- A catch-all error handler registered for `HTTPException` will not handle `RoutingException`, which is used internally during routing. This fixes the unexpected behavior that had been introduced in 1.0. [#2986](#)
- Passing the `json` argument to `app.test_client` does not push/pop an extra app context. [#2900](#)

Version 1.0.2

Released 2018-05-02

- Fix more backwards compatibility issues with merging slashes between a blueprint prefix and route. [#2748](#)

- Fix error with `flask routes` command when there are no routes. [#2751](#)

Version 1.0.1

Released 2018-04-29

- Fix registering partials (with no `name`) as view functions. [#2730](#)
- Don't treat lists returned from view functions the same as tuples. Only tuples are interpreted as response data. [#2736](#)
- Extra slashes between a blueprint's `url_prefix` and a route URL are merged. This fixes some backwards compatibility issues with the change in 1.0. [#2731](#), [#2742](#)
- Only trap `BadRequestKeyError` errors in debug mode, not all `BadRequest` errors. This allows `abort(400)` to continue working as expected. [#2735](#)
- The `FLASK_SKIP_DOTENV` environment variable can be set to `1` to skip automatically loading dotenv files. [#2722](#)

Version 1.0

Released 2018-04-26

- Python 2.6 and 3.3 are no longer supported.
- Bump minimum dependency versions to the latest stable versions: Werkzeug >= 0.14, Jinja >= 2.10, itsdangerous >= 0.24, Click >= 5.1. [#2586](#)
- Skip `app.run` when a Flask application is run from the command line. This avoids some behavior that was confusing to debug.
- Change the default for `JSONIFY_PRETTYPRINT_REGULAR` to `False`. `jsonify()` returns a compact format by default, and an indented format in debug mode. [#2193](#)
- `Flask.init` accepts the `host_matching` argument and sets it on `url_map`. [#1559](#)
- `Flask.init` accepts the `static_host` argument and passes it as the `host` argument when defining the static route. [#1559](#)
- `send_file()` supports Unicode in `attachment_filename`. [#2223](#)
- Pass `_scheme` argument from `url_for()` to `handle_url_build_error()`. [#2017](#)
- `add_url_rule()` accepts the `provide_automatic_options` argument to disable adding the `OPTIONS` method. [#1489](#)
- `MethodView` subclasses inherit method handlers from base classes. [#1936](#)
- Errors caused while opening the session at the beginning of the request are

handled by the app's error handlers. [#2254](#)

- Blueprints gained `json_encoder` and `json_decoder` attributes to override the app's encoder and decoder. [#1898](#)
- `Flask.make_response()` raises `TypeError` instead of `ValueError` for bad response types. The error messages have been improved to describe why the type is invalid. [#2256](#)
- Add `routes` CLI command to output routes registered on the application. [#2259](#)
- Show warning when session cookie domain is a bare hostname or an IP address, as these may not behave properly in some browsers, such as Chrome. [#2282](#)
- Allow IP address as exact session cookie domain. [#2282](#)
- `SESSION_COOKIE_DOMAIN` is set if it is detected through `SERVER_NAME`. [#2282](#)
- Auto-detect zero-argument app factory called `create_app` or `make_app` from `FLASK_APP`. [#2297](#)
- Factory functions are not required to take a `script_info` parameter to work with the `flask` command. If they take a single parameter or a parameter named `script_info`, the `ScriptInfo` object will be passed. [#2319](#)
- `FLASK_APP` can be set to an app factory, with arguments if needed, for example `FLASK_APP=myproject.app:create_app('dev')`. [#2326](#)
- `FLASK_APP` can point to local packages that are not installed in editable mode, although `pip install -e` is still preferred. [#2414](#)
- The `View` class attribute `provide_automatic_options` is set in `as_view()`, to be detected by `add_url_rule()`. [#2316](#)
- Error handling will try handlers registered for `blueprint, code`, `app, code`, `blueprint, exception`, `app, exception`. [#2314](#)
- `Cookie` is added to the response's `Vary` header if the session is accessed at all during the request (and not deleted). [#2288](#)
- `test_request_context()` accepts `subdomain` and `url_scheme` arguments for use when building the base URL. [#1621](#)
- Set `APPLICATION_ROOT` to `'/'` by default. This was already the implicit default when it was set to `None`.
- `TRAP_BAD_REQUEST_ERRORS` is enabled by default in debug mode. `BadRequestKeyError` has a message with the bad key in debug mode instead of the generic bad request message. [#2348](#)
- Allow registering new tags with `TaggedJSONSerializer` to support storing other types in the session cookie. [#2352](#)

- Only open the session if the request has not been pushed onto the context stack yet. This allows `stream_with_context()` generators to access the same session that the containing view uses. [#2354](#)
- Add `json` keyword argument for the test client request methods. This will dump the given object as JSON and set the appropriate content type. [#2358](#)
- Extract JSON handling to a mixin applied to both the `Request` and `Response` classes. This adds the `is_json()` and `get_json()` methods to the response to make testing JSON response much easier. [#2358](#)
- Removed error handler caching because it caused unexpected results for some exception inheritance hierarchies. Register handlers explicitly for each exception if you want to avoid traversing the MRO. [#2362](#)
- Fix incorrect JSON encoding of aware, non-UTC datetimes. [#2374](#)
- Template auto reloading will honor debug mode even if `jinja_env` was already accessed. [#2373](#)
- The following old deprecated code was removed. [#2385](#)
 - `flask.ext` - import extensions directly by their name instead of through the `flask.ext` namespace. For example, `import flask.ext.sqlalchemy` becomes `import flask_sqlalchemy`.
 - `Flask.init_jinja_globals` - extend `Flask.create_jinja_environment()` instead.
 - `Flask.error_handlers` - tracked by `Flask.error_handler_spec`, use `Flask.errorhandler()` to register handlers.
 - `Flask.request_globals_class` - use `Flask.app_ctx_globals_class` instead.
 - `Flask.static_path` - use `Flask.static_url_path` instead.
 - `Request.module` - use `Request.blueprint` instead.
- The `Request.json` property is no longer deprecated. [#1421](#)
- Support passing a `EnvironBuilder` or `dict` to `test_client.open`. [#2412](#)
- The `flask` command and `Flask.run()` will load environment variables from `.env` and `.flaskenv` files if python-dotenv is installed. [#2416](#)
- When passing a full URL to the test client, the scheme in the URL is used instead of `PREFERRED_URL_SCHEME`. [#2430](#)
- `Flask.logger` has been simplified. `LOGGER_NAME` and `LOGGER_HANDLER_POLICY` config was removed. The logger is always named `flask.app`. The level is only set on first access, it doesn't check `Flask.debug` each time. Only one format is used, not different ones depending on `Flask.debug`. No handlers are removed, and a handler is

only added if no handlers are already configured. [#2436](#)

- Blueprint view function names may not contain dots. [#2450](#)
- Fix a `ValueError` caused by invalid `Range` requests in some cases. [#2526](#)
- The development server uses threads by default. [#2529](#)
- Loading config files with `silent=True` will ignore `ENOTDIR` errors. [#2581](#)
- Pass `-cert` and `-key` options to `flask run` to run the development server over HTTPS. [#2606](#)
- Added `SESSION_COOKIE_SAMESITE` to control the `SameSite` attribute on the session cookie. [#2607](#)
- Added `test_cli_runner()` to create a Click runner that can invoke Flask CLI commands for testing. [#2636](#)
- Subdomain matching is disabled by default and setting `SERVER_NAME` does not implicitly enable it. It can be enabled by passing `subdomain_matching=True` to the `Flask` constructor. [#2635](#)
- A single trailing slash is stripped from the blueprint `url_prefix` when it is registered with the app. [#2629](#)
- `Request.get_json()` doesn't cache the result if parsing fails when `silent` is true. [#2651](#)
- `Request.get_json()` no longer accepts arbitrary encodings. Incoming JSON should be encoded using UTF-8 per [RFC 8259](#), but Flask will autodetect UTF-8, -16, or -32. [#2691](#)
- Added `MAX_COOKIE_SIZE` and `Response.max_cookie_size` to control when Werkzeug warns about large cookies that browsers may ignore. [#2693](#)
- Updated documentation theme to make docs look better in small windows. [#2709](#)
- Rewrote the tutorial docs and example project to take a more structured approach to help new users avoid common pitfalls. [#2676](#)

Version 0.12.4

Released 2018-04-29

- Repackage 0.12.3 to fix package layout issue. [#2728](#)

Version 0.12.3

Released 2018-04-26

- `Request.get_json()` no longer accepts arbitrary encodings. Incoming JSON should be encoded using UTF-8 per [RFC 8259](#), but Flask will autodetect UTF-8, -16, or -32. [#2692](#)
- Fix a Python warning about imports when using `python -m flask`. [#2666](#)
- Fix a `ValueError` caused by invalid `Range` requests in some cases.

Version 0.12.2

Released 2017-05-16

- Fix a bug in `safe_join` on Windows.

Version 0.12.1

Released 2017-03-31

- Prevent `flask run` from showing a `NoAppException` when an `ImportError` occurs within the imported application module.
- Fix encoding behavior of `app.config.from_pyfile` for Python 3. [#2118](#)
- Use the `SERVER_NAME` config if it is present as default values for `app.run`. [#2109](#), [#2152](#)
- Call `ctx.auto_pop` with the exception object instead of `None`, in the event that a `BaseException` such as `KeyboardInterrupt` is raised in a request handler.

Version 0.12

Released 2016-12-21, codename Punsch

- The cli command now responds to `-version`.
- Mimetype guessing and ETag generation for file-like objects in `send_file` has been removed. [#104](#), [:pr 1849](#)
- Mimetype guessing in `send_file` now fails loudly and doesn't fallback to `application/octet-stream`. [#1988](#)
- Make `flask.safe_join` able to join multiple paths like `os.path.join`. [#1730](#)
- Revert a behavior change that made the dev server crash instead of returning an Internal Server Error. [#2006](#)
- Correctly invoke response handlers for both regular request dispatching as well as error handlers.

- Disable logger propagation by default for the app logger.
- Add support for range requests in `send_file` .
- `app.test_client` includes preset default environment, which cannow be directly set, instead of per `client.get` .
- Fix crash when running under PyPy3. [#1814](#)

Version 0.11.1

Released 2016-06-07

- Fixed a bug that prevented `FLASKAPP=foobar/_init.py` from working. [#1872](#)

Version 0.11

Released 2016-05-29, codename Absinthe

- Added support to serializing top-level arrays to `flask jsonify()` . This introduces a security risk in ancientbrowsers. See [JSON 安全](#) for details.
- Added `before_render_template` signal.
- Added `**kwargs` to `flask.Test.test_client()` to support passing additional keyword arguments to the constructor of `flask.Flask.test_client_class` .
- Added `SESSION_REFRESH_EACH_REQUEST` config key that controls the set-cookie behavior. If set to `True` a permanent session will be refreshed each request and get their lifetime extended, if set to `False` it will only be modified if the session actually modifies. Non permanent sessions are not affected by this and will always expire if the browser window closes.
- Made Flask support custom JSON mimetypes for incoming data.
- Added support for returning tuples in the form `(response, headers)` from a view function.
- Added `flask.Config.from_json()` .
- Added `flask.Flask.config_class` .
- Added `flask.Config.get_namespace()` .
- Templates are no longer automatically reloaded outside of debug mode. This can be configured with the new `TEMPLATES_AUTO_RELOAD` config key.
- Added a workaround for a limitation in Python 3.3's namespace loader.
- Added support for explicit root paths when using Python 3.3's namespace packages.

- Added **flask** and the `flask.cli` module to start the local debug server through the click CLI system. This is recommended over the old `flask.run()` method as it works faster and more reliable due to a different design and also replaces `Flask-Script`.
- Error handlers that match specific classes are now checked first, thereby allowing catching exceptions that are subclasses of `HTTPExceptions` (in `werkzeug.exceptions`). This makes it possible for an extension author to create exceptions that will by default result in the HTTP error of their choosing, but may be caught with a custom error handler if desired.
- Added `flask.Config.from_mapping()`.
- Flask will now log by default even if debug is disabled. The log format is now hardcoded but the default log handling can be disabled through the `LOGGER_HANDLER_POLICY` configuration key.
- Removed deprecated module functionality.
- Added the `EXPLAIN_TEMPLATE_LOADING` config flag which when enabled will instruct Flask to explain how it locates templates. This should help users debug when the wrong templates are loaded.
- Enforce blueprint handling in the order they were registered for template loading.
- Ported test suite to `py.test`.
- Deprecated `request.json` in favour of `request.get_json()`.
- Add “pretty” and “compressed” separators definitions in `jsonify()` method. Reduces JSON response size when `JSONIFY_PRETTYPRINT_REGULAR=False` by removing unnecessary whitespace included by default after separators.
- JSON responses are now terminated with a newline character, because it is a convention that UNIX text files end with a newline and some clients don't deal well when this newline is missing. This came up originally as a part of <https://github.com/postmanlabs/httpbin/issues/168>. #1262
- The automatically provided `OPTIONS` method is now correctly disabled if the user registered an overriding rule with the lowercase-version `options`. #1288
- `flask.json.jsonify` now supports the `datetime.date` type. #1326
- Don't leak exception info of already caught exceptions to context teardown handlers. #1393
- Allow custom Jinja environment subclasses. #1422
- Updated extension dev guidelines.
- `flask.g` now has `pop()` and `setdefault` methods.

- Turn on autoescape for `flask.templating.render_template_string` by default. [#1515](#)
- `flask.ext` is now deprecated. [#1484](#)
- `send_from_directory` now raises `BadRequest` if the filename is invalid on the server OS. [#1763](#)
- Added the `JSONIFY_MIMETYPE` configuration variable. [#1728](#)
- Exceptions during teardown handling will no longer leave `badapplication` contexts lingering around.
- Fixed broken `test_appcontext_signals()` test case.
- Raise an `AttributeError` in `flask.helpers.find_package()` with a useful message explaining why it is raised when a PEP 302 import hook is used without an `is_package()` method.
- Fixed an issue causing exceptions raised before entering a requestor app context to be passed to teardown handlers.
- Fixed an issue with query parameters getting removed from requests in the test client when absolute URLs were requested.
- Made `@before_first_request` into a decorator as intended.
- Fixed an etags bug when sending a file streams with a name.
- Fixed `send_from_directory` not expanding to the application rootpath correctly.
- Changed logic of before first request handlers to flip the flag after invoking. This will allow some uses that are potentially dangerous but should probably be permitted.
- Fixed Python 3 bug when a handler from `app.url_build_error_handlers` reraises the `BuildError`.

Version 0.10.1

Released 2013-06-14

- Fixed an issue where `|tojson` was not quoting single quotes which made the filter not work properly in HTML attributes. Now it's possible to use that filter in single quoted attributes. This should make using that filter with angular.js easier.
- Added support for byte strings back to the session system. This broke compatibility with the common case of people putting binary data for token verification into the session.
- Fixed an issue where registering the same method twice for the same endpoint would

trigger an exception incorrectly.

Version 0.10

Released 2013-06-13, codename Limoncello

- Changed default cookie serialization format from pickle to JSON to limit the impact an attacker can do if the secret key leaks. See [Version 0.10](#) for more information.
- Added `template_test` methods in addition to the already existing `template_filter` method family.
- Added `template_global` methods in addition to the already existing `template_filter` method family.
- Set the content-length header for x-sendfile.
- `tojson` filter now does not escape script blocks in HTML5 parsers.
- `tojson` used in templates is now safe by default due. This was allowed due to the different escaping behavior.
- Flask will now raise an error if you attempt to register a new function on an already used endpoint.
- Added wrapper module around simplejson and added default serialization of datetime objects. This allows much easier customization of how JSON is handled by Flask or any Flask extension.
- Removed deprecated internal `flask.session` module alias. Use `flask.sessions` instead to get the session module. This is not to be confused with `flask.session` the session proxy.
- Templates can now be rendered without request context. The behavior is slightly different as the `request`, `session` and `g` objects will not be available and blueprint's context processors are not called.
- The config object is now available to the template as a real global and not through a context processor which makes it available even in imported templates by default.
- Added an option to generate non-ascii encoded JSON which should result in less bytes being transmitted over the network. It's disabled by default to not cause confusion with existing libraries that might expect `flask.json.dumps` to return bytestrings by default.
- `flask.g` is now stored on the app context instead of the request context.
- `flask.g` now gained a `get()` method for not erroring out on non existing items.

- `flask.g` now can be used with the `in` operator to see what's defined and it now is iterable and will yield all attributes stored.
- `flask.Flask.request_globals_class` got renamed to `flask.Flask.app_ctx_globals_class` which is a better name to what it does since 0.10.
- `request`, `session` and `g` are now also added as proxies to the template context which makes them available in imported templates. One has to be very careful with those though because usage outside of macros might cause caching.
- Flask will no longer invoke the wrong error handlers if a proxy exception is passed through.
- Added a workaround for chrome's cookies in localhost not working as intended with domain names.
- Changed logic for picking defaults for cookie values from session to work better with Google Chrome.
- Added `message_flashed` signal that simplifies flashing testing.
- Added support for copying of request contexts for better working with greenlets.
- Removed custom JSON HTTP exception subclasses. If you were relying on them you can reintroduce them again yourself trivially. Using them however is strongly discouraged as the interface was flawed.
- Python requirements changed: requiring Python 2.6 or 2.7 now to prepare for Python 3.3 port.
- Changed how the teardown system is informed about exceptions. This is now more reliable in case something handles an exception halfway through the error handling process.
- Request context preservation in debug mode now keeps the exception information around which means that teardown handlers are able to distinguish error from success cases.
- Added the `JSONIFY_PRETTYPRINT_REGULAR` configuration variable.
- Flask now orders JSON keys by default to not trash HTTP caches due to different hash seeds between different workers.
- Added `appcontext_pushed` and `appcontext_popped` signals.
- The builtin run method now takes the `SERVER_NAME` into account when picking the default port to run on.
- Added `flask.request.get_json()` as a replacement for the old `flask.request.json` property.

Version 0.9

Released 2012-07-01, codename Campari

- The `flask.Request.on_json_loading_failed()` now returns a JSONformatted response by default.
- The `flask.url_for()` function now can generate anchors to the generated links.
- The `flask.url_for()` function now can also explicitly generate URL rules specific to a given HTTP method.
- Logger now only returns the debug log setting if it was not set explicitly.
- Unregister a circular dependency between the WSGI environment and the request object when shutting down the request. This means that `environ['werkzeug.request']` will be `None` after the response was returned to the WSGI server but has the advantage that the garbage collector is not needed on CPython to tear down the request unless the user created circular dependencies themselves.
- Session is now stored after callbacks so that if the session payload is stored in the session you can still modify it in an after request callback.
- The `flask.Flask` class will avoid importing the provided import name if it can (the required first parameter), to benefit tools which build Flask instances programmatically. The Flask class will fall back to using import on systems with custom module hooks, e.g. Google App Engine, or when the import name is inside a zip archive (usually a .egg) prior to Python 2.7.
- Blueprints now have a decorator to add custom template filters application wide, `flask.Blueprint.app_template_filter()`.
- The Flask and Blueprint classes now have a non-decorator method for adding custom template filters application wide, `flask.Flask.add_template_filter()` and `flask.Blueprint.add_app_template_filter()`.
- The `flask.get_flashed_messages()` function now allows rendering flashed message categories in separate blocks, through a `category_filter` argument.
- The `flask.Flask.run()` method now accepts `None` for `host` and `port` arguments, using default values when `None`. This allows for calling run using configuration values, e.g. `app.run(app.config.get('MYHOST'), app.config.get('MYPORT'))`, with proper behavior whether or not a config file is provided.
- The `flask.render_template()` method now accepts either an iterable of template names or a single template name. Previously, it only accepted a single template name. On an iterable, the first template found is rendered.
- Added `flask.Flask.app_context()` which works very similar to the request context but only provides access to the current application. This also adds support for URL

generation without an active request context.

- View functions can now return a tuple with the first instance being an instance of `flask.Response`. This allows for returning `jsonify(error="error msg"), 400` from a view function.
- `Flask` and `Blueprint` now provide a `get_send_file_max_age()` hook for subclasses to override behavior of serving static files from Flask when using `flask.Flask.send_static_file()` (used for the default static file handler) and `send_file()`. This hook is provided a filename, which for example allows changing cache controls by file extension. The default max-age for `send_file` and static files can be configured through a new `SEND_FILE_MAX_AGE_DEFAULT` configuration variable, which is used in the default `get_send_file_max_age` implementation.
- Fixed an assumption in sessions implementation which could break message flashing on sessions implementations which use external storage.
- Changed the behavior of tuple return values from functions. They are no longer arguments to the response object, they now have a defined meaning.
- Added `flask.Flask.request_globals_class` to allow a specific class to be used on creation of the `g` instance of each request.
- Added `required_methods` attribute to view functions to force-add methods on registration.
- Added `flask.after_this_request()`.
- Added `flask.stream_with_context()` and the ability to push contexts multiple times without producing unexpected behavior.

Version 0.8.1

Released 2012-07-01

- Fixed an issue with the undocumented `flask.session` module to not work properly on Python 2.5. It should not be used but did cause some problems for package managers.

Version 0.8

Released 2011-09-29, codename Rakija

- Refactored session support into a session interface so that the implementation of the sessions can be changed without having to override the Flask class.
- Empty session cookies are now deleted properly automatically.
- View functions can now opt out of getting the automatic OPTIONS implementation.

- HTTP exceptions and Bad Request errors can now be trapped so that they show up normally in the traceback.
- Flask in debug mode is now detecting some common problems and tries to warn you about them.
- Flask in debug mode will now complain with an assertion error if a view was attached after the first request was handled. This gives earlier feedback when users forget to import view code ahead of time.
- Added the ability to register callbacks that are only triggered once at the beginning of the first request. (`Flask.before_first_request()`)
- Malformed JSON data will now trigger a bad request HTTP exception instead of a value error which usually would result in a 500 internal server error if not handled. This is a backwards incompatible change.
- Applications now not only have a root path where the resources and modules are located but also an instance path which is the designated place to drop files that are modified at runtime (upload etc.). Also this is conceptually only instance depending and outside version control so it's the perfect place to put configuration files etc. For more information see [实例文件夹](#).
- Added the `APPLICATION_ROOT` configuration variable.
- Implemented `session_transaction()` to easily modify sessions from the test environment.
- Refactored test client internally. The `APPLICATION_ROOT` configuration variable as well as `SERVER_NAME` are now properly used by the test client as defaults.
- Added `flask.views.View.decorators` to support simpler decorating of pluggable (class-based) views.
- Fixed an issue where the test client if used with the "with" statement did not trigger the execution of the teardown handlers.
- Added finer control over the session cookie parameters.
- HEAD requests to a method view now automatically dispatch to the `get` method if no handler was implemented.
- Implemented the virtual `flask.ext` package to import extensions from.
- The context preservation on exceptions is now an integral component of Flask itself and no longer of the test client. This cleaned up some internal logic and lowers the odds of runaway request contexts in unit tests.
- Fixed the Jinja2 environment's `list_templates` method not returning the correct names when blueprints or modules were involved.

Version 0.7.2

Released 2011-07-06

- Fixed an issue with URL processors not properly working on blueprints.

Version 0.7.1

Released 2011-06-29

- Added missing future import that broke 2.5 compatibility.
- Fixed an infinite redirect issue with blueprints.

Version 0.7

Released 2011-06-28, codename Grappa

- Added `make_default_options_response()` which can be used by subclasses to alter the default behavior for `OPTIONS` responses.
- Unbound locals now raise a proper `RuntimeError` instead of an `AttributeError`.
- Mimetype guessing and etag support based on file objects is now deprecated for `flask.send_file()` because it was unreliable. Pass filenames instead or attach your own etags and provide a proper mimetype by hand.
- Static file handling for modules now requires the name of the static folder to be supplied explicitly. The previous autodetection was not reliable and caused issues on Google's App Engine. Until 1.0 the old behavior will continue to work but issue dependency warnings.
- Fixed a problem for Flask to run on jython.
- Added a `PROPAGATE_EXCEPTIONS` configuration variable that can be used to flip the setting of exception propagation which previously was linked to `DEBUG` alone and is now linked to either `DEBUG` or `TESTING`.
- Flask no longer internally depends on rules being added through the `add_url_rule` function and can now also accept regular werkzeug rules added to the url map.
- Added an `endpoint` method to the flask application object which allows one to register a callback to an arbitrary endpoint with a decorator.
- Use Last-Modified for static file sending instead of Date which was incorrectly introduced in 0.6.
- Added `create_jinja_loader` to override the loader creation process.

- Implemented a silent flag for `config.from_pyfile` .
- Added `teardown_request` decorator, for functions that should run at the end of a request regardless of whether an exception occurred. Also the behavior for `after_request` was changed. It's now no longer executed when an exception is raised. See [Upgrading to new Teardown Handling](#)
- Implemented `flask.has_request_context()`
- Deprecated `init_jinja_globals` . Override the `create_jinja_environment()` method instead to achieve the same functionality.
- Added `flask.safe_join()`
- The automatic JSON request data unpacking now looks at the `charset_mimetype` parameter.
- Don't modify the session on `flask.get_flashed_messages()` if there are no messages in the session.
- `before_request` handlers are now able to abort requests with errors.
- It is not possible to define user exception handlers. That way you can provide custom error messages from a central hub for certain errors that might occur during request processing (for instance database connection errors, timeouts from remote resources etc.).
- Blueprints can provide blueprint specific error handlers.
- Implemented generic [可插拔视图](#) (class-based views).

Version 0.6.1

Released 2010-12-31

- Fixed an issue where the default `OPTIONS` response was not exposing all valid methods in the `Allow` header.
- Jinja2 template loading syntax now allows `./` in front of a template load path. Previously this caused issues with module setups.
- Fixed an issue where the subdomain setting for modules was ignored for the static folder.
- Fixed a security problem that allowed clients to download arbitrary files if the host server was a windows based operating system and the client uses backslashes to escape the directory the files were exposed from.

Version 0.6

Released 2010-07-27, codename Whisky

- After request functions are now called in reverse order of registration.
- OPTIONS is now automatically implemented by Flask unless the application explicitly adds 'OPTIONS' as method to the URL rule. In this case no automatic OPTIONS handling kicks in.
- Static rules are now even in place if there is no static folder for the module. This was implemented to aid GAE which will remove the static folder if it's part of a mapping in the .yaml file.
- The `config` is now available in the templates as `config`.
- Context processors will no longer override values passed directly to the render function.
- Added the ability to limit the incoming request data with the new `MAX_CONTENT_LENGTH` configuration value.
- The endpoint for the `flask.Module.add_url_rule()` method is now optional to be consistent with the function of the same name on the application object.
- Added a `flask.make_response()` function that simplifies creating response object instances in views.
- Added signalling support based on blinker. This feature is currently optional and supposed to be used by extensions and applications. If you want to use it, make sure to have [blinker](#) installed.
- Refactored the way URL adapters are created. This process is now fully customizable with the `create_url_adapter()` method.
- Modules can now register for a subdomain instead of just an URL prefix. This makes it possible to bind a whole module to a configurable subdomain.

Version 0.5.2

Released 2010-07-15

- Fixed another issue with loading templates from directories when modules were used.

Version 0.5.1

Released 2010-07-06

- Fixes an issue with template loading from directories when modules were used.

Version 0.5

Released 2010-07-06, codename Calvados

- Fixed a bug with subdomains that was caused by the inability to specify the server name. The server name can now be set with the `SERVER_NAME` config key. This key is now also used to set the session cookie cross-subdomain wide.
- Autoescaping is no longer active for all templates. Instead it is only active for `.html`, `.htm`, `.xml` and `.xhtml`. Inside templates this behavior can be changed with the `autoescape` tag.
- Refactored Flask internally. It now consists of more than a single file.
- `flask.send_file()` now emits etags and has the ability to do conditional responses builtin.
- (temporarily) dropped support for zipped applications. This was a rarely used feature and led to some confusing behavior.
- Added support for per-package template and static-file directories.
- Removed support for `create_jinja_loader` which is no longer used in 0.5 due to the improved module support.
- Added a helper function to expose files from any directory.

Version 0.4

Released 2010-06-18, codename Rakia

- Added the ability to register application wide error handlers from modules.
- `after_request()` handlers are now also invoked if the request dies with an exception and an error handling page kicks in.
- Test client has not the ability to preserve the request context for a little longer. This can also be used to trigger custom requests that do not pop the request stack for testing.
- Because the Python standard library caches loggers, the name of the logger is configurable now to better support unittests.
- Added `TESTING` switch that can activate unittesting helpers.
- The logger switches to `DEBUG` mode now if debug is enabled.

Version 0.3.1

Released 2010-05-28

- Fixed a error reporting bug with `flask.Config.from_envvar()`
- Removed some unused code from flask
- Release does no longer include development leftover files (.gitfolder for themes, built documentation in zip and pdf file and some.pyc files)

Version 0.3

Released 2010-05-28, codename Schnaps

- Added support for categories for flashed messages.
- The application now configures a `logging.Handler` and willlog request handling exceptions to that logger when not in debugmode. This makes it possible to receive mails on server errors forexample.
- Added support for context binding that does not require the use ofthe with statement for playing in the console.
- The request context is now available within the with statementmaking it possible to further push the request context or pop it.
- Added support for configurations.

Version 0.2

Released 2010-05-12, codename J?germeister

- Various bugfixes
- Integrated JSON support
- Added `get_template_attribute()` helper function.
- `add_url_rule()` can now also register a viewfunction.
- Refactored internal request dispatching.
- Server listens on 127.0.0.1 by default now to fix issues withchrome.
- Added external URL support.
- Added support for `send_file()`
- Module support and internal request handling refactoring to bettersupport pluggable applications.

- Sessions can be set to be permanent now on a per-session basis.
- Better error reporting on missing secret keys.
- Added support for Google Appengine.

Version 0.1

Released 2010-04-16

- First public preview release.

License

Source License

This license applies to all files in the Flask repository and sourcedistribution. This includes Flask's source code, the examples, andtests, as well as the documentation.

Copyright 2010 Pallets

Redistribution and use in source and binary forms, with or withoutmodification, are permitted provided that the following conditions aremet:

- Redistributions of source code must retain the above copyrightnotice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyrightnotice, this list of conditions and the following disclaimer in thedocumentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of itscontributors may be used to endorse or promote products derived fromthis software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOTLIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR APARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHTHOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITEDTO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, ORPROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OFLIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDINGNEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THISSOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Artwork License

This license applies to Flask's logo.

Copyright 2010 Pallets

This logo or a modified version may be used by anyone to refer to theFlask project, but does not indicate endorsement by the project.

Redistribution and use in source (SVG) and binary (renders in PNG, etc.)forms, with or without modification, are permitted provided that thefollowing conditions are met:

- Redistributions of source code must retain the above copyrightnotice and this

list of conditions.

- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

We would appreciate that you make the image a link

to <https://palletsprojects.com/p/flask/> if you use it in a medium that supports links.

如何为 Flask 做出贡献

感谢您为 Flask 做出贡献！

问答支持

请不要使用问题跟踪器来提问。有关你自己代码的问题请使用下列途径之一提问：

- 在 FreeNode 上的 IRC `#pocoo` 频道。
- 更普通的问题请使用 FreeNode 上的 IRC `#python` 频道。
- Discord chat 上的 `#get-help` 频道：<https://discordapp.com/invite/t6rrQZH>
- FreeNode 上的 IRC `#pocoo` 频道与 Discord 关联，但是请优先使用 Discord。
- 邮件列表 flask@python.org 用于长期或者大型问题讨论。
- 在 [Stack Overflow](#) 上提问。首先使用以下方法在 Google 上搜索：`site:stackoverflow.com flask {search term, exception message, etc.}`

报告问题

- 描述你希望发生的事情。
- 如果可能，提供一个 [最小的可重现的示例](#) 以帮助我们找到问题。这也有助于鉴别问题是否也你自己的代码有关。
- 描述实际发生了什么。如果有异常，则应当包含完整的回溯。
- 列出你的 Python 、 Flask 和 Werkzeug 版本。如果可能，检查是否这个问题已在存储库中修复。

提交补丁

- 使用 [Black](#) 自动格式化你的代码。当你运行 `pip install -e .[dev]` 时，这个功能将被配置为一个 git [pre-commit](#) 钩子。你还可以希望使用 Black 的 [编辑器集成](#)。
- 如果补丁是用于解决错误的，那么应当包含一个测试，并明确说明错误发生于何种情况之下。确保如果没有补丁，测试就会失败。
- 在你的提交信息中包含一个类似 “Fixes #123” 字符串（其中的 123 是指你修正的 issue 编号）。参见 [Closing issues using keywords](#)。

首次设置

- 下载并安装 [最新版的 git](#)。

- 配置使用 git 的 `username` 和 `email`:

```
1. git config --global user.name 'your name'
2. git config --global user.email 'your email'
```

- 确保你有一个 [GitHub 账号](#).
- 点击 [Fork](#) 按钮将 Flask fork 到你的 GitHub 账户。
- 把你的 GitHub fork [Clone](#) 到本地:

```
1. git clone https://github.com/{username}/flask
2. cd flask
```

- 添加一个主存储库作为远程库，稍后更新:

```
1. git remote add pallets https://github.com/pallets/flask
2. git fetch pallets
```

- 创建一个 `virtualenv`:

```
1. python3 -m venv env
2. . env/bin/activate
3. # or "env\Scripts\activate" on Windows
```

- 在带有开发依赖的编辑模式下安装 Flask:

```
1. pip install -e ".[dev]"
```

- 安装 `pre-commit` 钩子:

```
1. pre-commit install --install-hooks
```

开始写代码

- 创建一个分支来鉴别你想要处理的问题。如果要提交一个缺陷或者文档修正，请从最后的“`.x`”分支来创建分支:

```
1. git checkout -b your-branch-name origin/1.0.x
```

如果要提交一个功能增加或者更改，请从“`master`”分支来创建分支:

```
1. git checkout -b your-branch-name origin/master
```

- 使用你最喜欢的编辑器，修改代码，[随时提交](#)。

- 应当包含覆盖你所做的全部修改的测试。确保没有补丁则测试失败。[运行测试](#)。
- 将你的提交推送到 GitHub 并 [创建一个 pull request](#)

```
1. git push --set-upstream origin your-branch-name
```

- 庆祝成功

运行测试

用以下命令运行基础测试：

```
1. pytest
```

这只在当前环境下运行测试。这是否相关取决于你在处理 Flask 的哪个部分。当你提交 pull request 时，Travis-CI 会运行全部测试。

完整的测试套件运行时间会很长，因为它会在多种 Python 及其依赖的环境下运行。在所有环境下运行测试需要有 Python 2.7 、 3.4 、 3.5 、 3.6 和 PyPy 2.7 。然后运行：

```
1. tox
```

运行测试覆盖

生成一个哪些代码未被测试覆盖的报告可以指明从哪里开始贡献。使用 `coverage` 运行 `pytest` 并在终端生成一个报告和一份交互 HTML 文档：

```
1. coverage run -m pytest
2. coverage report
3. coverage html
4. # then open htmlcov/index.html
```

请阅读更多关于 [coverage](#) 的文档。

用 `tox` 运行完整测试套件会组合所有运行测试的覆盖报告。

构建文档

使用 Sphinx 构建 `docs` 文件夹中的文档：

```
1. cd docs
2. make html
```

在浏览器中打开 `_build/html/index.html` 以查看文档。

请阅读更多关于 [Sphinx](#) 的内容。

注意：零填充文件模式

本存储库包含多个零填充文件模式，当提交存储库到 GitHub 之外的 git 主机时可能会引发问题。修复这个问题会破坏提交历史记录，因此我们建议忽略这个问题。如果推送失败并且你使用的是如 GitLab 这样的自托管 git 服务，那么在管理面板中关闭存储库检查。

这些文件还会在克隆时引发问题。如果你在 git 配置文件中有以下设置：

```
1. [fetch]
2. fsckobjects = true
```

或者

```
1. [receive]
2. fsckObjects = true
```

那么克隆时会失败。唯一的解决方法是在克隆时把上面的设置项目设置为 `false`，并在克隆完成后恢复。