

# Table of Contents

[欢迎使用 Flask](#Top_of_index_html)

[前言](#Top_of_foreword_html)

[前言](#Top_of_foreword_html)

[“微” 是什么意思？](#_Wei___Shi_Shi_Yao_Yi_Si____Wei)

[配置与惯例](#Pei_Zhi_Yu_Guan_Li__Flask_Shu_Mu)

[与 Flask 共成长](#Yu__Flask_Gong_Cheng_Chang__Dang)

[给有经验程序员的前言](#Top_of_advanced_foreword_html)

[给有经验程序员的前言](#Top_of_advanced_foreword_html)

[Flask 中的线程局部变量](#Flask_Zhong_De_Xian_Cheng_Ju_Bu)

[Web 开发是危险的](#Web_Kai_Fa_Shi_Wei_Xian_De__Qing)

[Python 3 的状态](#Python_3_De_Zhuang_Tai__Python_S)

[安装](#Top_of_installation_html)

[安装](#Top_of_installation_html)

[virtualenv](#virtualenv_Ni_Zai_Kai_Fa_Zhong_H)

[全局安装](#Quan_Ju_An_Zhuang__Zhe_Yang_Ye_S)

[活在边缘](#Huo_Zai_Bian_Yuan__Ru_Guo_Ni_Xu)

[Windows 下的 pip 和 distribute](#Windows_Xia_De__pip_He__distribu)

[快速入门](#Top_of_quickstart_html)

[快速入门](#Top_of_quickstart_html)

[一个最小的应用](#Yi_Ge_Zui_Xiao_De_Ying_Yong__Yi)

[调试模式](#Diao_Shi_Mo_Shi__Sui_Ran__run)

[路由](#Lu_You__Xian_Dai__web_Ying_Yong)

[路由](#Lu_You__Xian_Dai__web_Ying_Yong)

[变量规则](#Bian_Liang_Gui_Ze__Yao_Gei__URL)

[构建 URL](#Gou_Jian__URL_Ru_Guo_Ta_Neng_Pi)

[HTTP 方法](#HTTP_Fang_Fa__HTTP__web_Ying_Yon)

[静态文件](#Jing_Tai_Wen_Jian__Dong_Tai__web)

[模板渲染](#Mo_Ban_Xuan_Ran__Zai__Python_Li)

[访问请求数据](#Fang_Wen_Qing_Qiu_Shu_Ju__Dui_Yu)

[访问请求数据](#Fang_Wen_Qing_Qiu_Shu_Ju__Dui_Yu)

[局部上下文](#Ju_Bu_Shang_Xia_Wen___Nei_Mu__Ru)

[请求对象](#Qing_Qiu_Dui_Xiang__Qing_Qiu_Dui)

[文件上传](#Wen_Jian_Shang_Chuan__Ni_Ke_Yi_H)

[Cookies](#Cookies_Ni_Ke_Yi_Tong_Guo__cooki)

[重定向和错误](#Zhong_Ding_Xiang_He_Cuo_Wu__Zhon)

[关于响应](#Guan_Yu_Xiang_Ying__Shi_Tu_Han_S)

[会话](#Hui_Hua__Chu_Qing_Qiu_Dui_Xiang)

[消息闪现](#Xiao_Xi_Shan_Xian__Liang_Hao_De)

[日志记录](#Ri_Zhi_Ji_Lu___0_3_Xin_Ban_Gong)

[整合 WSGI 中间件](#Zheng_He__WSGI_Zhong_Jian_Jian)

[部署到 Web 服务器](#Bu_Shu_Dao__Web_Fu_Wu_Qi__Zhun_B)

[教程](#Top_of_index_html_1)

[教程](#Top_of_index_html_1)

[介绍 Flaskr](#Top_of_introduction_html)

[步骤 0: 创建文件夹](#Top_of_folders_html)

[步骤 1: 数据库模式](#Top_of_schema_html)

[步骤 2: 应用设置代码](#Top_of_setup_html)

[步骤 3: 创建数据库](#Top_of_dbinit_html)

[步骤 4: 请求数据库连接](#Top_of_dbcon_html)

[步骤 5: 视图函数](#Top_of_views_html)

[步骤 5: 视图函数](#Top_of_views_html)

[显示条目](#Xian_Shi_Tiao_Mu__Zhe_Ge_Shi_Tu)

[添加条目](#Tian_Jia_Tiao_Mu__Zhe_Ge_Shi_Tu)

[登入和登出](#Deng_Ru_He_Deng_Chu__Zhe_Xie_Han)

[步骤 6: 模板](#Top_of_templates_html)

[步骤 6: 模板](#Top_of_templates_html)

[layout.html](#layout_html_Zhe_Ge_Mo_Ban_Bao_Ha)

[show\_entries.html](#show_entries_html_Zhe_Ge_Mo_Ban)

[login.html](#login_html_Zui_Hou_Shi_Deng_Ru_M)

[步骤 7: 添加样式](#Top_of_css_html)

[福利: 应用测试](#Top_of_testing_html)

[模板](#Top_of_templating_html)

[模板](#Top_of_templating_html)

[Jinja 配置](#Jinja_Pei_Zhi__Jinja_2_Mo_Ren_Pe)

[标准上下文](#Biao_Zhun_Shang_Xia_Wen__Xia_Mia)

[标准过滤器](#Biao_Zhun_Guo_Lu_Qi__Zhe_Xie_Guo)

[控制自动转义](#Kong_Zhi_Zi_Dong_Zhuan_Yi__Zi_Do)

[注册过滤器](#Zhu_Ce_Guo_Lu_Qi__Ru_Guo_Ni_Yao)

[上下文处理器](#Shang_Xia_Wen_Chu_Li_Qi__Flask_S)

[测试 Flask 应用](#Top_of_testing_html_1)

[测试 Flask 应用](#Top_of_testing_html_1)

[应用程序](#Ying_Yong_Cheng_Xu__Shou_Xian__W)

[测试的大框架](#Ce_Shi_De_Da_Kuang_Jia__Wei_Le_C)

[第一个测试](#Di_Yi_Ge_Ce_Shi__Shi_Jin_Xing_Di)

[登陆和登出](#Deng_Lu_He_Deng_Chu__Wo_Men_Ying)

[测试消息的添加](#Ce_Shi_Xiao_Xi_De_Tian_Jia__Wo_M)

[其他测试技巧](#Qi_Ta_Ce_Shi_Ji_Qiao__Chu_Le_Ru)

[伪造资源和上下文](#Wei_Zao_Zi_Yuan_He_Shang_Xia_Wen)

[保存上下文](#Bao_Cun_Shang_Xia_Wen___0_4_Xin)

[访问和修改 Sessions](#Fang_Wen_He_Xiu_Gai__Sessions__0)

[记录应用错误](#Top_of_errorhandling_html)

[记录应用错误](#Top_of_errorhandling_html)

[错误邮件](#Cuo_Wu_You_Jian__Ru_Guo_Ni_De_Yi)

[记录到文件](#Ji_Lu_Dao_Wen_Jian__Ji_Bian_Ni_S)

[控制日志格式](#Kong_Zhi_Ri_Zhi_Ge_Shi__Mo_Ren_Q)

[控制日志格式](#Kong_Zhi_Ri_Zhi_Ge_Shi__Mo_Ren_Q)

[邮件](#You_Jian__from_logging_import_Fo)

[日志文件](#Ri_Zhi_Wen_Jian__from_logging_im)

[复杂日志格式](#Fu_Za_Ri_Zhi_Ge_Shi__Zhe_Li_Gei)

[其它的库](#Qi_Ta_De_Ku__Zhi_Ci__Wo_Men_Zhi)

[调试应用错误](#Diao_Shi_Ying_Yong_Cuo_Wu__Dui_Y)

[调试应用错误](#Diao_Shi_Ying_Yong_Cuo_Wu__Dui_Y)

[有疑问时，手动运行](#You_Yi_Wen_Shi__Shou_Dong_Yun_Xi)

[调试器操作](#Diao_Shi_Qi_Cao_Zuo__Wei_Le_Shen)

[配置处理](#Top_of_config_html)

[配置处理](#Top_of_config_html)

[配置基础](#Pei_Zhi_Ji_Chu__config_Shi_Ji_Sh)

[内置的配置值](#Nei_Zhi_De_Pei_Zhi_Zhi__Xia_Lie)

[从文件配置](#Cong_Wen_Jian_Pei_Zhi__Ru_Guo_Ni)

[配置的最佳实践](#Pei_Zhi_De_Zui_Jia_Shi_Jian__Zhi)

[开发 / 生产](#Kai_Fa____Sheng_Chan__Da_Duo_Shu)

[实例文件夹](#Shi_Li_Wen_Jian_Jia___0_8_Xin_Ba)

[信号](#Top_of_signals_html)

[信号](#Top_of_signals_html)

[订阅信号](#Ding_Yue_Xin_Hao__Ni_Ke_Yi_Shi_Y)

[创建信号](#Chuang_Jian_Xin_Hao__Ru_Guo_Ni_X)

[发送信号](#Fa_Song_Xin_Hao__Ru_Guo_Ni_Xiang)

[信号与 Flask 的请求上下文](#Xin_Hao_Yu__Flask_De_Qing_Qiu_Sh)

[基于装饰器的信号订阅](#Ji_Yu_Zhuang_Shi_Qi_De_Xin_Hao_D)

[核心信号](#He_Xin_Xin_Hao__Xia_Lie_Shi__Fla)

[即插视图](#Top_of_views_html_1)

[即插视图](#Top_of_views_html_1)

[基本原则](#Ji_Ben_Yuan_Ze__Xiang_Xiang_Ni_Y)

[方法提示](#Fang_Fa_Ti_Shi__Ji_Cha_Shi_Tu_Ke)

[基于调度的方法](#Ji_Yu_Diao_Du_De_Fang_Fa__Dui_Me)

[装饰视图](#Zhuang_Shi_Shi_Tu__Ji_Ran_Shi_Tu)

[用于 API 的方法视图](#Yong_Yu__API_De_Fang_Fa_Shi_Tu)

[应用上下文](#Top_of_appcontext_html)

[应用上下文](#Top_of_appcontext_html)

[应用上下文的作用](#Ying_Yong_Shang_Xia_Wen_De_Zuo_Y)

[创建应用上下文](#Chuang_Jian_Ying_Yong_Shang_Xia)

[应用上下文局部变量](#Ying_Yong_Shang_Xia_Wen_Ju_Bu_Bi)

[上下文用法](#Shang_Xia_Wen_Yong_Fa__Shang_Xia)

[请求上下文](#Top_of_reqcontext_html)

[请求上下文](#Top_of_reqcontext_html)

[深入上下文作用域](#Shen_Ru_Shang_Xia_Wen_Zuo_Yong_Y)

[上下文如何工作](#Shang_Xia_Wen_Ru_He_Gong_Zuo__Ru)

[回调和错误](#Hui_Diao_He_Cuo_Wu__Zai__Flask_Z)

[销毁回调](#Xiao_Hui_Hui_Diao__Xiao_Hui_Hui)

[留意代理](#Liu_Yi_Dai_Li__Flask_Zhong_Ti_Go)

[错误时的上下文保护](#Cuo_Wu_Shi_De_Shang_Xia_Wen_Bao)

[用蓝图实现模块化的应用](#Top_of_blueprints_html)

[用蓝图实现模块化的应用](#Top_of_blueprints_html)

[为什么使用蓝图？](#Wei_Shi_Yao_Shi_Yong_Lan_Tu___Fl)

[蓝图的设想](#Lan_Tu_De_She_Xiang__Lan_Tu_De_J)

[我的第一个蓝图](#Wo_De_Di_Yi_Ge_Lan_Tu__Zhe_Kan_Q)

[注册蓝图](#Zhu_Ce_Lan_Tu__Na_Yao_Ni_Ru_He_Z)

[蓝图资源](#Lan_Tu_Zi_Yuan__Lan_Tu_Ye_Ke_Yi)

[蓝图资源](#Lan_Tu_Zi_Yuan__Lan_Tu_Ye_Ke_Yi)

[蓝图资源文件夹](#Lan_Tu_Zi_Yuan_Wen_Jian_Jia__Xia)

[静态文件](#Jing_Tai_Wen_Jian__Yi_Ge_Lan_Tu)

[模板](#Mo_Ban__Ru_Guo_Ni_Xiang_Yao_Lan)

[构造 URL](#Gou_Zao__URL_Dang_Ni_Xiang_Yao_C)

[Flask 扩展](#Top_of_extensions_html)

[Flask 扩展](#Top_of_extensions_html)

[寻找扩展](#Xun_Zhao_Kuo_Zhan__Flask_Extensi)

[使用扩展](#Shi_Yong_Kuo_Zhan__Kuo_Zhan_Tong)

[Flask 0.8 以前](#Flask_0_8_Yi_Qian__Ru_Guo_Ni_Zai)

[与 Shell 共舞](#Top_of_shell_html)

[与 Shell 共舞](#Top_of_shell_html)

[创建一个请求上下文](#Chuang_Jian_Yi_Ge_Qing_Qiu_Shang)

[激发请求发送前后的调用](#Ji_Fa_Qing_Qiu_Fa_Song_Qian_Hou)

[进一步提升 Shell 使用体验](#Jin_Yi_Bu_Ti_Sheng__Shell_Shi_Yo)

[Flask 代码模式](#Top_of_index_html_2)

[Flask 代码模式](#Top_of_index_html_2)

[大型应用](#Top_of_packages_html)

[大型应用](#Top_of_packages_html)

[简单的包](#Jian_Dan_De_Bao__Jiang_Yi_Ge_Xia)

[与蓝图一起工作](#Yu_Lan_Tu_Yi_Qi_Gong_Zuo__Ru_Guo)

[应用程序的工厂函数](#Top_of_appfactories_html)

[应用程序的工厂函数](#Top_of_appfactories_html)

[基础的工厂函数](#Ji_Chu_De_Gong_Chang_Han_Shu__Ni)

[使用应用程序](#Shi_Yong_Ying_Yong_Cheng_Xu__Suo)

[工厂函数的改进](#Gong_Chang_Han_Shu_De_Gai_Jin__Q)

[应用调度](#Top_of_appdispatch_html)

[应用调度](#Top_of_appdispatch_html)

[如何使用此文档](#Ru_He_Shi_Yong_Ci_Wen_Dang__Xia)

[合并应用](#He_Bing_Ying_Yong__Ru_Guo_Nin_Yo)

[通过子域名调度](#Tong_Guo_Zi_Yu_Ming_Diao_Du__You)

[使用路径来调度](#Shi_Yong_Lu_Jing_Lai_Diao_Du__To)

[使用 URL 处理器](#Top_of_urlprocessors_html)

[使用 URL 处理器](#Top_of_urlprocessors_html)

[国际化的应用程序 URL](#Guo_Ji_Hua_De_Ying_Yong_Cheng_Xu)

[多国语言化的 Blueprint URL](#Duo_Guo_Yu_Yan_Hua_De__Blueprint)

[部署和分发](#Top_of_distribute_html)

[部署和分发](#Top_of_distribute_html)

[基础的安装脚本](#Ji_Chu_De_An_Zhuang_Jiao_Ben__Yi)

[分发代码](#Fen_Fa_Dai_Ma__Ru_Guo_Nin_Shi_Tu)

[声明依赖关系](#Sheng_Ming_Yi_Lai_Guan_Xi__Nin_X)

[安装 / 开发](#An_Zhuang____Kai_Fa__An_Zhuang_N)

[使用 Fabric 部署](#Top_of_fabric_html)

[使用 Fabric 部署](#Top_of_fabric_html)

[创建第一个 Fabfile](#Chuang_Jian_Di_Yi_Ge__Fabfile_Fa)

[运行 Fabfile](#Yun_Xing__Fabfile_Ru_He_Zhi_Xing)

[WSGI 文件](#WSGI_Wen_Jian__WSGI_Wen_Jian_Yin)

[配置文件](#Pei_Zhi_Wen_Jian__Zheng_Ru_Shang)

[第一次部署](#Di_Yi_Ci_Bu_Shu__Xian_Zai_Wo_Men)

[下一步操作](#Xia_Yi_Bu_Cao_Zuo__Cong_Xian_Zai)

[在 Flask 中使用 SQLite 3](#Top_of_sqlite3_html)

[在 Flask 中使用 SQLite 3](#Top_of_sqlite3_html)

[按需连接](#An_Xu_Lian_Jie__Shang_Shu_Fang_F)

[简化查询](#Jian_Hua_Cha_Xun__Xian_Zai_Zai_M)

[初始化数据库模型](#Chu_Shi_Hua_Shu_Ju_Ku_Mo_Xing__G)

[在 Flask 中使用 SQLAlchemy](#Top_of_sqlalchemy_html)

[在 Flask 中使用 SQLAlchemy](#Top_of_sqlalchemy_html)

[Flask-SQLAlchemy 扩展](#Flask_SQLAlchemy_Kuo_Zhan__Yin_W)

[显式调用](#Xian_Shi_Diao_Yong__SQLAlchemy_Z)

[手动实现 ORM](#Shou_Dong_Shi_Xian__ORM_Shou_Don)

[SQL 抽象层](#SQL_Chou_Xiang_Ceng__Ru_Guo_Nin)

[上传文件](#Top_of_fileuploads_html)

[上传文件](#Top_of_fileuploads_html)

[一点点介绍](#Yi_Dian_Dian_Jie_Shao__Rang_Wo_M)

[改进上传功能](#Gai_Jin_Shang_Chuan_Gong_Neng)

[上传进度条](#Shang_Chuan_Jin_Du_Tiao__Yi_Qian)

[更简单解决方案](#Geng_Jian_Dan_Jie_Jue_Fang_An__Y)

[缓存](#Top_of_caching_html)

[缓存](#Top_of_caching_html)

[配置缓存](#Pei_Zhi_Huan_Cun__Lei_Si_Yu_Jian)

[使用缓存](#Shi_Yong_Huan_Cun__You_Liang_Ge)

[视图装饰器](#Top_of_viewdecorators_html)

[视图装饰器](#Top_of_viewdecorators_html)

[过滤未登录用户的装饰器](#Guo_Lu_Wei_Deng_Lu_Yong_Hu_De_Zh)

[缓存装饰器](#Huan_Cun_Zhuang_Shi_Qi__Shi_Xian)

[模板装饰器](#Mo_Ban_Zhuang_Shi_Qi__TurboGears)

[终端装饰器](#Zhong_Duan_Zhuang_Shi_Qi__Ru_Guo)

[使用 WTForms 进行表单验证](#Top_of_wtforms_html)

[使用 WTForms 进行表单验证](#Top_of_wtforms_html)

[表单](#Biao_Dan__Yi_Xia_Shi_Yi_Ge_Dian)

[在视图里](#Zai_Shi_Tu_Li__Zai_Shi_Tu_Han_Sh)

[在模板中使用表单](#Zai_Mo_Ban_Zhong_Shi_Yong_Biao_D)

[模板继承](#Top_of_templateinheritance_html)

[模板继承](#Top_of_templateinheritance_html)

[基础模板](#Ji_Chu_Mo_Ban__Zai_Zhe_Ge_Jiao_Z)

[子模板](#Zi_Mo_Ban__Zi_Mo_Ban_Kan_Qi_Lai)

[消息闪现](#Top_of_flashing_html)

[消息闪现](#Top_of_flashing_html)

[简单的闪现](#Jian_Dan_De_Shan_Xian__Zhe_Li_Sh)

[分类闪现](#Fen_Lei_Shan_Xian___0_3_Xin_Ban)

[过滤闪现消息](#Guo_Lu_Shan_Xian_Xiao_Xi___0_9_X)

[用 jQuery 实现 Ajax](#Top_of_jquery_html)

[用 jQuery 实现 Ajax](#Top_of_jquery_html)

[加载 jQuery](#Jia_Zai__jQuery_Wei_Le_Shi_Yong)

[我的站点在哪?](#Wo_De_Zhan_Dian_Zai_Na___Nin_Zhi)

[JSON 视图函数](#JSON_Shi_Tu_Han_Shu__Xian_Zai_Ra)

[HTML 部分](#HTML_Bu_Fen__Nin_De__index_html)

[自定义错误页面](#Top_of_errorpages_html)

[自定义错误页面](#Top_of_errorpages_html)

[通常的错误代码](#Tong_Chang_De_Cuo_Wu_Dai_Ma__Xia)

[错误处理器](#Cuo_Wu_Chu_Li_Qi__Yi_Ge_Cuo_Wu_C)

[延迟加载视图](#Top_of_lazyloading_html)

[延迟加载视图](#Top_of_lazyloading_html)

[转换到中央 URL 映射](#Zhuan_Huan_Dao_Zhong_Yang__URL_Y)

[延迟加载](#Yan_Chi_Jia_Zai__Mu_Qian_Wo_Men)

[在 Flask 中使用 MongoKit](#Top_of_mongokit_html)

[在 Flask 中使用 MongoKit](#Top_of_mongokit_html)

[显式调用](#Xian_Shi_Diao_Yong__MongoKit_De)

[PyMongo 兼容层](#PyMongo_Jian_Rong_Ceng__Ru_Guo_N)

[添加 Favicon](#Top_of_favicon_html)

[添加 Favicon](#Top_of_favicon_html)

[参考](#Can_Kao___Wikipedia_Shang_You_Gu)

[数据流](#Top_of_streaming_html)

[数据流](#Top_of_streaming_html)

[基本使用](#Ji_Ben_Shi_Yong__Xia_Mian_Shi_Yi)

[在模板中生成流](#Zai_Mo_Ban_Zhong_Sheng_Cheng_Liu)

[延迟请求回调](#Top_of_deferredcallbacks_html)

[延迟请求回调](#Top_of_deferredcallbacks_html)

[装饰器](#Zhuang_Shi_Qi__Xia_Mian_De_Zhuan)

[调用延迟函数](#Diao_Yong_Yan_Chi_Han_Shu__Xian)

[一个实际应用的例子](#Yi_Ge_Shi_Ji_Ying_Yong_De_Li_Zi)

[添加 HTTP Method Overrides](#Top_of_methodoverrides_html)

[请求内容校验码](#Top_of_requestchecksum_html)

[基于 Celery 的后台任务](#Top_of_celery_html)

[基于 Celery 的后台任务](#Top_of_celery_html)

[安装 Celery](#An_Zhuang__Celery_Celery_Ti_Jiao)

[配置 Celery](#Pei_Zhi__Celery_Ni_Xu_Yao_De_Di)

[最简示例](#Zui_Jian_Shi_Li__Tong_Guo_Shang)

[运行 Celery 职程](#Yun_Xing__Celery_Zhi_Cheng__Xian)

[部署选择](#Top_of_index_html_3)

[部署选择](#Top_of_index_html_3)

[mod\_wsgi (Apache)](#Top_of_mod_wsgi_html)

[mod\_wsgi (Apache)](#Top_of_mod_wsgi_html)

[安装 mod\_wsgi](#An_Zhuang__mod_wsgi_Ru_Guo_Ni_Hu)

[创建一个 .wsgi 文件](#Chuang_Jian_Yi_Ge___wsgi_Wen_Jia)

[配置 Apache](#Pei_Zhi__Apache_Ni_Xu_Yao_Zuo_De)

[故障排除](#Gu_Zhang_Pai_Chu__Ru_Guo_Ni_De_Y)

[自动重加载支持](#Zi_Dong_Zhong_Jia_Zai_Zhi_Chi__N)

[使用虚拟环境](#Shi_Yong_Xu_Ni_Huan_Jing__Xu_Ni)

[独立 WSGI 容器](#Top_of_wsgi_standalone_html)

[独立 WSGI 容器](#Top_of_wsgi_standalone_html)

[Gunicorn](#Gunicorn_Gunicorn__http___gunico)

[Tornado](#Tornado_Tornado__http___www_torn)

[Gevent](#Gevent_Gevent__http___www_gevent)

[代理设置](#Dai_Li_She_Zhi__Ru_Guo_Ni_Zai_Yi)

[uWSGI](#Top_of_uwsgi_html)

[uWSGI](#Top_of_uwsgi_html)

[用 uwsgi 启动你的应用](#Yong__uwsgi_Qi_Dong_Ni_De_Ying_Y)

[配置 nginx](#Pei_Zhi__nginx_Yi_Ge_Ji_Ben_De)

[FastCGI](#Top_of_fastcgi_html)

[FastCGI](#Top_of_fastcgi_html)

[创建一个 .fcgi 文件](#Chuang_Jian_Yi_Ge___fcgi_Wen_Jia)

[配置 lighttpd](#Pei_Zhi__lighttpd_Yi_Ge_Gei__lig)

[配置 nginx](#Pei_Zhi__nginx_Zai__nginx_Shang)

[运行 FastCGI 进程](#Yun_Xing__FastCGI_Jin_Cheng__Ji)

[调试](#Diao_Shi__FastCGI_Zai_Da_Duo_Shu)

[CGI](#Top_of_cgi_html)

[CGI](#Top_of_cgi_html)

[创建一个 .cgi 文件](#Chuang_Jian_Yi_Ge___cgi_Wen_Jian)

[服务器配置](#Fu_Wu_Qi_Pei_Zhi__Tong_Chang_You)

[聚沙成塔](#Top_of_becomingbig_html)

[聚沙成塔](#Top_of_becomingbig_html)

[阅读源码](#Yue_Du_Yuan_Ma__Flask_De_Chuang)

[钩子，继承](#Gou_Zi__Ji_Cheng__API_Wen_Dang_L)

[继承](#Ji_Cheng__Flask_Lei_You_Xu_Duo_W)

[用中间件包装](#Yong_Zhong_Jian_Jian_Bao_Zhuang)

[分支](#Fen_Zhi__Ru_Guo_Shang_Shu_Xuan_Z)

[像专家一样扩大规模](#Xiang_Zhuan_Jia_Yi_Yang_Kuo_Da_G)

[与社区对话](#Yu_She_Qu_Dui_Hua__Flask_Kai_Fa)

[API](#Top_of_api_html)

[API](#Top_of_api_html)

[应用对象](#Ying_Yong_Dui_Xiang____class_fla)

[蓝图对象](#Lan_Tu_Dui_Xiang____class_flask)

[进入的请求对象](#Jin_Ru_De_Qing_Qiu_Dui_Xiang)

[响应对象](#Xiang_Ying_Dui_Xiang____class_fl)

[会话](#Hui_Hua__Ru_Guo_Ni_She_Zhi_Le__F)

[会话接口](#Hui_Hua_Jie_Kou___0_8_Xin_Ban_Go)

[测试客户端](#Ce_Shi_Ke_Hu_Duan____class_flask)

[应用全局变量](#Ying_Yong_Quan_Ju_Bian_Liang__Zh)

[有用的函数和类](#You_Yong_De_Han_Shu_He_Lei____fl)

[消息闪现](#Xiao_Xi_Shan_Xian____flask_flash)

[JSON 支持](#JSON_Zhi_Chi__Flask_Shi_Yong__si)

[模板渲染](#Mo_Ban_Xuan_Ran____flask_render)

[配置](#Pei_Zhi____class_flask_Config_ro)

[扩展](#Kuo_Zhan____flask_ext_Zhe_Ge_Mo)

[流的辅助函数](#Liu_De_Fu_Zhu_Han_Shu____flask_s)

[有用的内构件](#You_Yong_De_Nei_Gou_Jian____clas)

[信号](#Xin_Hao___0_6_Xin_Ban_Gong_Neng_1)

[基于类的视图](#Ji_Yu_Lei_De_Shi_Tu___0_7_Xin_Ba)

[URL 路由注册](#URL_Lu_You_Zhu_Ce__Zai_Lu_You_Xi)

[视图函数选项](#Shi_Tu_Han_Shu_Xuan_Xiang__Dui_N)

[Flask 中的设计决策](#Top_of_design_html)

[Flask 中的设计决策](#Top_of_design_html)

[显式的应用对象](#Xian_Shi_De_Ying_Yong_Dui_Xiang)

[路由系统](#Lu_You_Xi_Tong__Flask_Shi_Yong)

[某个模板引擎](#Mou_Ge_Mo_Ban_Yin_Qing__Flask_Za)

[微与依赖](#Wei_Yu_Yi_Lai__Wei_Shi_Yao__Flas)

[线程局域变量](#Xian_Cheng_Ju_Yu_Bian_Liang__Fla)

[Flask 是什么，不是什么？](#Flask_Shi_Shi_Yao__Bu_Shi_Shi_Ya)

[HTML/XHTML 常见问题](#Top_of_htmlfaq_html)

[HTML/XHTML 常见问题](#Top_of_htmlfaq_html)

[XHTML 的历史](#XHTML_De_Li_Shi__Yi_Duan_Shi_Jia)

[HTML5 的历史](#HTML5_De_Li_Shi__HTML5_Gui_Fan_D)

[HTML vs. XHTML](#HTML_vs__XHTML_Xia_Mian_De_Biao)

[“严格”意味着什么？](#_Yan_Ge__Yi_Wei_Zhao_Shi_Yao___H)

[HTML5 中的新技术](#HTML5_Zhong_De_Xin_Ji_Zhu__HTML5)

[应该使用什么？](#Ying_Gai_Shi_Yong_Shi_Yao___Yi_B)

[安全注意事项](#Top_of_security_html)

[安全注意事项](#Top_of_security_html)

[跨站脚本攻击（XSS）](#Kua_Zhan_Jiao_Ben_Gong_Ji__XSS)

[跨站请求伪造（CSRF）](#Kua_Zhan_Qing_Qiu_Wei_Zao__CSRF)

[JSON 安全](#JSON_An_Quan___ECMAScript_5_De_B)

[Flask 中的 Unicode](#Top_of_unicode_html)

[Flask 中的 Unicode](#Top_of_unicode_html)

[自动转换](#Zi_Dong_Zhuan_Huan__Wei_Le_Ti_Go)

[金科玉律](#Jin_Ke_Yu_Lu__Jing_Yan_Fa_Ze__Ru)

[自行编解码](#Zi_Xing_Bian_Jie_Ma__Ru_Guo_Ni_D)

[配置编辑器](#Pei_Zhi_Bian_Ji_Qi__Xian_Zai_De)

[Flask 扩展开发](#Top_of_extensiondev_html)

[Flask 扩展开发](#Top_of_extensiondev_html)

[剖析扩展](#Pou_Xi_Kuo_Zhan__Suo_You_De_Kuo)

[“Hello Flaskext!”](#_Hello_Flaskext___Na_Yao_Rang_Wo)

[“Hello Flaskext!”](#_Hello_Flaskext___Na_Yao_Rang_Wo)

[setup.py](#setup_py_Xia_Yi_Ge_Jue_Dui_Xu_Ya)

[flask\_sqlite3.py](#flask_sqlite3_py_Xian_Zai_Zhe_Ge)

[初始化扩展](#Chu_Shi_Hua_Kuo_Zhan__Xu_Duo_Kuo)

[扩展的代码](#Kuo_Zhan_De_Dai_Ma__Xia_Mian_Shi)

[使用 \_app\_ctx\_stack](#Shi_Yong___app_ctx_stack_Zai_Sha)

[销毁行为](#Xiao_Hui_Xing_Wei__Zhe_Zhi_Zai_N)

[他山之石，可以攻玉](#Ta_Shan_Zhi_Shi__Ke_Yi_Gong_Yu)

[已审核的扩展](#Yi_Shen_He_De_Kuo_Zhan__Flask_Ye)

[扩展导入的过渡](#Kuo_Zhan_Dao_Ru_De_Guo_Du__Yi_Du)

[Pocoo 风格指引](#Top_of_styleguide_html)

[Pocoo 风格指引](#Top_of_styleguide_html)

[总体布局](#Zong_Ti_Bu_Ju___Suo_Jin___4Ge_Ko)

[表达式和语句](#Biao_Da_Shi_He_Yu_Ju___Chang_Gui)

[命名约定](#Ming_Ming_Yue_Ding___Lei_Ming)

[文档注释](#Wen_Dang_Zhu_Shi___Wen_Dang_Zi_F)

[注释](#Zhu_Shi__Zhu_Shi_De_Gui_Ze_He_We)

[Python 3 支持](#Top_of_python3_html)

[Python 3 支持](#Top_of_python3_html)

[需求](#Xu_Qiu__Ru_Guo_Ni_Xiang_Yao_Zai)

[API 稳定性](#API_Wen_Ding_Xing__Python_3_Zhon)

[少数派](#Shao_Shu_Pai__Gen_Ju__PyPI_Xia_Z)

[小生态系统](#Xiao_Sheng_Tai_Xi_Tong__Da_Duo_S)

[建议](#Jian_Yi__Chu_Fei_Ni_Yi_Jing_Shu)

[升级到最新版本](#Top_of_upgrading_html)

[升级到最新版本](#Top_of_upgrading_html)

[Version 0.10](#Version_0_10_Ban_Ben__0_9_Dao__0)

[Version 0.9](#Version_0_9_Cong_Han_Shu_Zhong_F)

[Version 0.8](#Version_0_8_Flask_introduced_a_n)

[Version 0.7](#Version_0_7_In_Flask_0_7_we_clea)

[Version 0.7](#Version_0_7_In_Flask_0_7_we_clea)

[Bug in Request Locals](#Bug_in_Request_Locals_Due_to_a_b)

[Upgrading to new Teardown Handling](#Upgrading_to_new_Teardown_Handli)

[Manual Error Handler Attaching](#Manual_Error_Handler_Attaching_W)

[Blueprint Support](#Blueprint_Support_Blueprints_rep)

[Version 0.6](#Version_0_6_Flask_0_6_comes_with)

[Version 0.5](#Version_0_5_Flask_0_5_is_the_fir)

[Version 0.4](#Version_0_4_For_application_deve)

[Version 0.3](#Version_0_3_Flask_0_3_introduces)

[Version 0.3](#Version_0_3_Flask_0_3_introduces)

[Configuration Support](#Configuration_Support_The_config)

[Logging Integration](#Logging_Integration_Flask_now_co)

[Categories for Flash Messages](#Categories_for_Flash_Messages_Fl)

[Flask Changelog](#Top_of_changelog_html)

[Flask Changelog](#Top_of_changelog_html)

[Version 1.0](#Version_1_0__release_date_to_be)

[Version 0.10.2](#Version_0_10_2__bugfix_release)

[Version 0.10.1](#Version_0_10_1__bugfix_release)

[Version 0.10](#Version_0_10_Released_on_June_13)

[Version 0.9](#Version_0_9_Released_on_July_1st)

[Version 0.8.1](#Version_0_8_1_Bugfix_release__re)

[Version 0.8](#Version_0_8_Released_on_Septembe)

[Version 0.7.3](#Version_0_7_3_Bugfix_release__re)

[Version 0.7.2](#Version_0_7_2_Bugfix_release__re)

[Version 0.7.1](#Version_0_7_1_Bugfix_release__re)

[Version 0.7](#Version_0_7_Released_on_June_28t)

[Version 0.6.1](#Version_0_6_1_Bugfix_release__re)

[Version 0.6](#Version_0_6_Released_on_July_27t)

[Version 0.5.2](#Version_0_5_2_Bugfix_Release__re)

[Version 0.5.1](#Version_0_5_1_Bugfix_Release__re)

[Version 0.5](#Version_0_5_Released_on_July_6th)

[Version 0.4](#Version_0_4_Released_on_June_18t)

[Version 0.3.1](#Version_0_3_1_Bugfix_release__re)

[Version 0.3](#Version_0_3_Released_on_May_28th)

[Version 0.2](#Version_0_2_Released_on_May_12th)

[Version 0.1](#Version_0_1_First_public_preview)

[许可证](#Top_of_license_html)

[许可证](#Top_of_license_html)

[作者](#Zuo_Zhe__Flask_is_written_and_ma)

[作者](#Zuo_Zhe__Flask_is_written_and_ma)

[Development Lead](#Development_Lead__Armin_Ronacher)

[Patches and Suggestions](#Patches_and_Suggestions__Adam_Za)

[通用许可证定义](#Tong_Yong_Xu_Ke_Zheng_Ding_Yi__X)

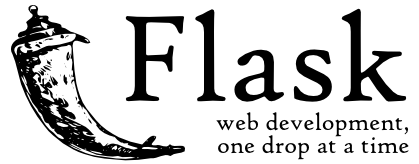
[Flask License](#Flask_License_Copyright__c__2012)

[Flask Artwork License](#Flask_Artwork_License_Copyright)

[术语表](#Top_of_glossary_html)

[索引](#Top_of_genindex_html) [下一页](#Top_of_foreword_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 欢迎使用 Flask



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

Flask 依赖两个外部库： [Jinja2](http://jinja.pocoo.org/2/) [http://jinja.pocoo.org/2/] 模板引擎和 [Werkzeug](http://werkzeug.pocoo.org/) [http://werkzeug.pocoo.org/] WSGI 工具 集。此文档不包含这两个库的文档。要细读它们的文档，请点击下面的链接：

* [Jinja2 文档](http://jinja.pocoo.org/2/documentation/) [http://jinja.pocoo.org/2/documentation/]
* [Werkzeug 文档](http://werkzeug.pocoo.org/documentation/) [http://werkzeug.pocoo.org/documentation/]

## 用户指南

这部分文档很枯燥，以介绍 Flask 的背景开始，然后注重说明 Flask 的 Web 开发的各个步骤。

* [前言](#Top_of_foreword_html)
  + [“微” 是什么意思？](#_Wei___Shi_Shi_Yao_Yi_Si____Wei)
  + [配置与惯例](#Pei_Zhi_Yu_Guan_Li__Flask_Shu_Mu)
  + [与 Flask 共成长](#Yu__Flask_Gong_Cheng_Chang__Dang)
* [给有经验程序员的前言](#Top_of_advanced_foreword_html)
  + [Flask 中的线程局部变量](#Flask_Zhong_De_Xian_Cheng_Ju_Bu)
  + [Web 开发是危险的](#Web_Kai_Fa_Shi_Wei_Xian_De__Qing)
  + [Python 3 的状态](#Python_3_De_Zhuang_Tai__Python_S)
* [安装](#Top_of_installation_html)
  + [virtualenv](#virtualenv_Ni_Zai_Kai_Fa_Zhong_H)
  + [全局安装](#Quan_Ju_An_Zhuang__Zhe_Yang_Ye_S)
  + [活在边缘](#Huo_Zai_Bian_Yuan__Ru_Guo_Ni_Xu)
  + [Windows 下的 pip 和 distribute](#Windows_Xia_De__pip_He__distribu)
* [快速入门](#Top_of_quickstart_html)
  + [一个最小的应用](#Yi_Ge_Zui_Xiao_De_Ying_Yong__Yi)
  + [调试模式](#Diao_Shi_Mo_Shi__Sui_Ran__run)
  + [路由](#Lu_You__Xian_Dai__web_Ying_Yong)
  + [静态文件](#Jing_Tai_Wen_Jian__Dong_Tai__web)
  + [模板渲染](#Mo_Ban_Xuan_Ran__Zai__Python_Li)
  + [访问请求数据](#Fang_Wen_Qing_Qiu_Shu_Ju__Dui_Yu)
  + [重定向和错误](#Zhong_Ding_Xiang_He_Cuo_Wu__Zhon)
  + [关于响应](#Guan_Yu_Xiang_Ying__Shi_Tu_Han_S)
  + [会话](#Hui_Hua__Chu_Qing_Qiu_Dui_Xiang)
  + [消息闪现](#Xiao_Xi_Shan_Xian__Liang_Hao_De)
  + [日志记录](#Ri_Zhi_Ji_Lu___0_3_Xin_Ban_Gong)
  + [整合 WSGI 中间件](#Zheng_He__WSGI_Zhong_Jian_Jian)
  + [部署到 Web 服务器](#Bu_Shu_Dao__Web_Fu_Wu_Qi__Zhun_B)
* [教程](#Top_of_index_html_1)
  + [介绍 Flaskr](#Top_of_introduction_html)
  + [步骤 0: 创建文件夹](#Top_of_folders_html)
  + [步骤 1: 数据库模式](#Top_of_schema_html)
  + [步骤 2: 应用设置代码](#Top_of_setup_html)
  + [步骤 3: 创建数据库](#Top_of_dbinit_html)
  + [步骤 4: 请求数据库连接](#Top_of_dbcon_html)
  + [步骤 5: 视图函数](#Top_of_views_html)
  + [步骤 6: 模板](#Top_of_templates_html)
  + [步骤 7: 添加样式](#Top_of_css_html)
  + [福利: 应用测试](#Top_of_testing_html)
* [模板](#Top_of_templating_html)
  + [Jinja 配置](#Jinja_Pei_Zhi__Jinja_2_Mo_Ren_Pe)
  + [标准上下文](#Biao_Zhun_Shang_Xia_Wen__Xia_Mia)
  + [标准过滤器](#Biao_Zhun_Guo_Lu_Qi__Zhe_Xie_Guo)
  + [控制自动转义](#Kong_Zhi_Zi_Dong_Zhuan_Yi__Zi_Do)
  + [注册过滤器](#Zhu_Ce_Guo_Lu_Qi__Ru_Guo_Ni_Yao)
  + [上下文处理器](#Shang_Xia_Wen_Chu_Li_Qi__Flask_S)
* [测试 Flask 应用](#Top_of_testing_html_1)
  + [应用程序](#Ying_Yong_Cheng_Xu__Shou_Xian__W)
  + [测试的大框架](#Ce_Shi_De_Da_Kuang_Jia__Wei_Le_C)
  + [第一个测试](#Di_Yi_Ge_Ce_Shi__Shi_Jin_Xing_Di)
  + [登陆和登出](#Deng_Lu_He_Deng_Chu__Wo_Men_Ying)
  + [测试消息的添加](#Ce_Shi_Xiao_Xi_De_Tian_Jia__Wo_M)
  + [其他测试技巧](#Qi_Ta_Ce_Shi_Ji_Qiao__Chu_Le_Ru)
  + [伪造资源和上下文](#Wei_Zao_Zi_Yuan_He_Shang_Xia_Wen)
  + [保存上下文](#Bao_Cun_Shang_Xia_Wen___0_4_Xin)
  + [访问和修改 Sessions](#Fang_Wen_He_Xiu_Gai__Sessions__0)
* [记录应用错误](#Top_of_errorhandling_html)
  + [错误邮件](#Cuo_Wu_You_Jian__Ru_Guo_Ni_De_Yi)
  + [记录到文件](#Ji_Lu_Dao_Wen_Jian__Ji_Bian_Ni_S)
  + [控制日志格式](#Kong_Zhi_Ri_Zhi_Ge_Shi__Mo_Ren_Q)
  + [其它的库](#Qi_Ta_De_Ku__Zhi_Ci__Wo_Men_Zhi)
* [调试应用错误](#Diao_Shi_Ying_Yong_Cuo_Wu__Dui_Y)
  + [有疑问时，手动运行](#You_Yi_Wen_Shi__Shou_Dong_Yun_Xi)
  + [调试器操作](#Diao_Shi_Qi_Cao_Zuo__Wei_Le_Shen)
* [配置处理](#Top_of_config_html)
  + [配置基础](#Pei_Zhi_Ji_Chu__config_Shi_Ji_Sh)
  + [内置的配置值](#Nei_Zhi_De_Pei_Zhi_Zhi__Xia_Lie)
  + [从文件配置](#Cong_Wen_Jian_Pei_Zhi__Ru_Guo_Ni)
  + [配置的最佳实践](#Pei_Zhi_De_Zui_Jia_Shi_Jian__Zhi)
  + [开发 / 生产](#Kai_Fa____Sheng_Chan__Da_Duo_Shu)
  + [实例文件夹](#Shi_Li_Wen_Jian_Jia___0_8_Xin_Ba)
* [信号](#Top_of_signals_html)
  + [订阅信号](#Ding_Yue_Xin_Hao__Ni_Ke_Yi_Shi_Y)
  + [创建信号](#Chuang_Jian_Xin_Hao__Ru_Guo_Ni_X)
  + [发送信号](#Fa_Song_Xin_Hao__Ru_Guo_Ni_Xiang)
  + [信号与 Flask 的请求上下文](#Xin_Hao_Yu__Flask_De_Qing_Qiu_Sh)
  + [基于装饰器的信号订阅](#Ji_Yu_Zhuang_Shi_Qi_De_Xin_Hao_D)
  + [核心信号](#He_Xin_Xin_Hao__Xia_Lie_Shi__Fla)
* [即插视图](#Top_of_views_html_1)
  + [基本原则](#Ji_Ben_Yuan_Ze__Xiang_Xiang_Ni_Y)
  + [方法提示](#Fang_Fa_Ti_Shi__Ji_Cha_Shi_Tu_Ke)
  + [基于调度的方法](#Ji_Yu_Diao_Du_De_Fang_Fa__Dui_Me)
  + [装饰视图](#Zhuang_Shi_Shi_Tu__Ji_Ran_Shi_Tu)
  + [用于 API 的方法视图](#Yong_Yu__API_De_Fang_Fa_Shi_Tu)
* [应用上下文](#Top_of_appcontext_html)
  + [应用上下文的作用](#Ying_Yong_Shang_Xia_Wen_De_Zuo_Y)
  + [创建应用上下文](#Chuang_Jian_Ying_Yong_Shang_Xia)
  + [应用上下文局部变量](#Ying_Yong_Shang_Xia_Wen_Ju_Bu_Bi)
  + [上下文用法](#Shang_Xia_Wen_Yong_Fa__Shang_Xia)
* [请求上下文](#Top_of_reqcontext_html)
  + [深入上下文作用域](#Shen_Ru_Shang_Xia_Wen_Zuo_Yong_Y)
  + [上下文如何工作](#Shang_Xia_Wen_Ru_He_Gong_Zuo__Ru)
  + [回调和错误](#Hui_Diao_He_Cuo_Wu__Zai__Flask_Z)
  + [销毁回调](#Xiao_Hui_Hui_Diao__Xiao_Hui_Hui)
  + [留意代理](#Liu_Yi_Dai_Li__Flask_Zhong_Ti_Go)
  + [错误时的上下文保护](#Cuo_Wu_Shi_De_Shang_Xia_Wen_Bao)
* [用蓝图实现模块化的应用](#Top_of_blueprints_html)
  + [为什么使用蓝图？](#Wei_Shi_Yao_Shi_Yong_Lan_Tu___Fl)
  + [蓝图的设想](#Lan_Tu_De_She_Xiang__Lan_Tu_De_J)
  + [我的第一个蓝图](#Wo_De_Di_Yi_Ge_Lan_Tu__Zhe_Kan_Q)
  + [注册蓝图](#Zhu_Ce_Lan_Tu__Na_Yao_Ni_Ru_He_Z)
  + [蓝图资源](#Lan_Tu_Zi_Yuan__Lan_Tu_Ye_Ke_Yi)
  + [构造 URL](#Gou_Zao__URL_Dang_Ni_Xiang_Yao_C)
* [Flask 扩展](#Top_of_extensions_html)
  + [寻找扩展](#Xun_Zhao_Kuo_Zhan__Flask_Extensi)
  + [使用扩展](#Shi_Yong_Kuo_Zhan__Kuo_Zhan_Tong)
  + [Flask 0.8 以前](#Flask_0_8_Yi_Qian__Ru_Guo_Ni_Zai)
* [与 Shell 共舞](#Top_of_shell_html)
  + [创建一个请求上下文](#Chuang_Jian_Yi_Ge_Qing_Qiu_Shang)
  + [激发请求发送前后的调用](#Ji_Fa_Qing_Qiu_Fa_Song_Qian_Hou)
  + [进一步提升 Shell 使用体验](#Jin_Yi_Bu_Ti_Sheng__Shell_Shi_Yo)
* [Flask 代码模式](#Top_of_index_html_2)
  + [大型应用](#Top_of_packages_html)
  + [应用程序的工厂函数](#Top_of_appfactories_html)
  + [应用调度](#Top_of_appdispatch_html)
  + [使用 URL 处理器](#Top_of_urlprocessors_html)
  + [部署和分发](#Top_of_distribute_html)
  + [使用 Fabric 部署](#Top_of_fabric_html)
  + [在 Flask 中使用 SQLite 3](#Top_of_sqlite3_html)
  + [在 Flask 中使用 SQLAlchemy](#Top_of_sqlalchemy_html)
  + [上传文件](#Top_of_fileuploads_html)
  + [缓存](#Top_of_caching_html)
  + [视图装饰器](#Top_of_viewdecorators_html)
  + [使用 WTForms 进行表单验证](#Top_of_wtforms_html)
  + [模板继承](#Top_of_templateinheritance_html)
  + [消息闪现](#Top_of_flashing_html)
  + [用 jQuery 实现 Ajax](#Top_of_jquery_html)
  + [自定义错误页面](#Top_of_errorpages_html)
  + [延迟加载视图](#Top_of_lazyloading_html)
  + [在 Flask 中使用 MongoKit](#Top_of_mongokit_html)
  + [添加 Favicon](#Top_of_favicon_html)
  + [数据流](#Top_of_streaming_html)
  + [延迟请求回调](#Top_of_deferredcallbacks_html)
  + [添加 HTTP Method Overrides](#Top_of_methodoverrides_html)
  + [请求内容校验码](#Top_of_requestchecksum_html)
  + [基于 Celery 的后台任务](#Top_of_celery_html)
* [部署选择](#Top_of_index_html_3)
  + [mod\_wsgi (Apache)](#Top_of_mod_wsgi_html)
  + [独立 WSGI 容器](#Top_of_wsgi_standalone_html)
  + [uWSGI](#Top_of_uwsgi_html)
  + [FastCGI](#Top_of_fastcgi_html)
  + [CGI](#Top_of_cgi_html)
* [聚沙成塔](#Top_of_becomingbig_html)
  + [阅读源码](#Yue_Du_Yuan_Ma__Flask_De_Chuang)
  + [钩子，继承](#Gou_Zi__Ji_Cheng__API_Wen_Dang_L)
  + [继承](#Ji_Cheng__Flask_Lei_You_Xu_Duo_W)
  + [用中间件包装](#Yong_Zhong_Jian_Jian_Bao_Zhuang)
  + [分支](#Fen_Zhi__Ru_Guo_Shang_Shu_Xuan_Z)
  + [像专家一样扩大规模](#Xiang_Zhuan_Jia_Yi_Yang_Kuo_Da_G)
  + [与社区对话](#Yu_She_Qu_Dui_Hua__Flask_Kai_Fa)

## API 参考

如果你在寻找一个特定函数、类或方法的信息，那么这部分文档就是给你准备的。

* [API](#Top_of_api_html)
  + [应用对象](#Ying_Yong_Dui_Xiang____class_fla)
  + [蓝图对象](#Lan_Tu_Dui_Xiang____class_flask)
  + [进入的请求对象](#Jin_Ru_De_Qing_Qiu_Dui_Xiang)
  + [响应对象](#Xiang_Ying_Dui_Xiang____class_fl)
  + [会话](#Hui_Hua__Ru_Guo_Ni_She_Zhi_Le__F)
  + [会话接口](#Hui_Hua_Jie_Kou___0_8_Xin_Ban_Go)
  + [测试客户端](#Ce_Shi_Ke_Hu_Duan____class_flask)
  + [应用全局变量](#Ying_Yong_Quan_Ju_Bian_Liang__Zh)
  + [有用的函数和类](#You_Yong_De_Han_Shu_He_Lei____fl)
  + [消息闪现](#Xiao_Xi_Shan_Xian____flask_flash)
  + [JSON 支持](#JSON_Zhi_Chi__Flask_Shi_Yong__si)
  + [模板渲染](#Mo_Ban_Xuan_Ran____flask_render)
  + [配置](#Pei_Zhi____class_flask_Config_ro)
  + [扩展](#Kuo_Zhan____flask_ext_Zhe_Ge_Mo)
  + [流的辅助函数](#Liu_De_Fu_Zhu_Han_Shu____flask_s)
  + [有用的内构件](#You_Yong_De_Nei_Gou_Jian____clas)
  + [信号](#Xin_Hao___0_6_Xin_Ban_Gong_Neng_1)
  + [基于类的视图](#Ji_Yu_Lei_De_Shi_Tu___0_7_Xin_Ba)
  + [URL 路由注册](#URL_Lu_You_Zhu_Ce__Zai_Lu_You_Xi)
  + [视图函数选项](#Shi_Tu_Han_Shu_Xuan_Xiang__Dui_N)

## 额外说明

这部分的设计说明，法律信息和变更记录为有兴趣的人准备。

* [Flask 中的设计决策](#Top_of_design_html)
  + [显式的应用对象](#Xian_Shi_De_Ying_Yong_Dui_Xiang)
  + [路由系统](#Lu_You_Xi_Tong__Flask_Shi_Yong)
  + [某个模板引擎](#Mou_Ge_Mo_Ban_Yin_Qing__Flask_Za)
  + [微与依赖](#Wei_Yu_Yi_Lai__Wei_Shi_Yao__Flas)
  + [线程局域变量](#Xian_Cheng_Ju_Yu_Bian_Liang__Fla)
  + [Flask 是什么，不是什么？](#Flask_Shi_Shi_Yao__Bu_Shi_Shi_Ya)
* [HTML/XHTML 常见问题](#Top_of_htmlfaq_html)
  + [XHTML 的历史](#XHTML_De_Li_Shi__Yi_Duan_Shi_Jia)
  + [HTML5 的历史](#HTML5_De_Li_Shi__HTML5_Gui_Fan_D)
  + [HTML vs. XHTML](#HTML_vs__XHTML_Xia_Mian_De_Biao)
  + [“严格”意味着什么？](#_Yan_Ge__Yi_Wei_Zhao_Shi_Yao___H)
  + [HTML5 中的新技术](#HTML5_Zhong_De_Xin_Ji_Zhu__HTML5)
  + [应该使用什么？](#Ying_Gai_Shi_Yong_Shi_Yao___Yi_B)
* [安全注意事项](#Top_of_security_html)
  + [跨站脚本攻击（XSS）](#Kua_Zhan_Jiao_Ben_Gong_Ji__XSS)
  + [跨站请求伪造（CSRF）](#Kua_Zhan_Qing_Qiu_Wei_Zao__CSRF)
  + [JSON 安全](#JSON_An_Quan___ECMAScript_5_De_B)
* [Flask 中的 Unicode](#Top_of_unicode_html)
  + [自动转换](#Zi_Dong_Zhuan_Huan__Wei_Le_Ti_Go)
  + [金科玉律](#Jin_Ke_Yu_Lu__Jing_Yan_Fa_Ze__Ru)
  + [自行编解码](#Zi_Xing_Bian_Jie_Ma__Ru_Guo_Ni_D)
  + [配置编辑器](#Pei_Zhi_Bian_Ji_Qi__Xian_Zai_De)
* [Flask 扩展开发](#Top_of_extensiondev_html)
  + [剖析扩展](#Pou_Xi_Kuo_Zhan__Suo_You_De_Kuo)
  + [“Hello Flaskext!”](#_Hello_Flaskext___Na_Yao_Rang_Wo)
  + [初始化扩展](#Chu_Shi_Hua_Kuo_Zhan__Xu_Duo_Kuo)
  + [扩展的代码](#Kuo_Zhan_De_Dai_Ma__Xia_Mian_Shi)
  + [使用 \_app\_ctx\_stack](#Shi_Yong___app_ctx_stack_Zai_Sha)
  + [销毁行为](#Xiao_Hui_Xing_Wei__Zhe_Zhi_Zai_N)
  + [他山之石，可以攻玉](#Ta_Shan_Zhi_Shi__Ke_Yi_Gong_Yu)
  + [已审核的扩展](#Yi_Shen_He_De_Kuo_Zhan__Flask_Ye)
  + [扩展导入的过渡](#Kuo_Zhan_Dao_Ru_De_Guo_Du__Yi_Du)
* [Pocoo 风格指引](#Top_of_styleguide_html)
  + [总体布局](#Zong_Ti_Bu_Ju___Suo_Jin___4Ge_Ko)
  + [表达式和语句](#Biao_Da_Shi_He_Yu_Ju___Chang_Gui)
  + [命名约定](#Ming_Ming_Yue_Ding___Lei_Ming)
  + [文档注释](#Wen_Dang_Zhu_Shi___Wen_Dang_Zi_F)
  + [注释](#Zhu_Shi__Zhu_Shi_De_Gui_Ze_He_We)
* [Python 3 支持](#Top_of_python3_html)
  + [需求](#Xu_Qiu__Ru_Guo_Ni_Xiang_Yao_Zai)
  + [API 稳定性](#API_Wen_Ding_Xing__Python_3_Zhon)
  + [少数派](#Shao_Shu_Pai__Gen_Ju__PyPI_Xia_Z)
  + [小生态系统](#Xiao_Sheng_Tai_Xi_Tong__Da_Duo_S)
  + [建议](#Jian_Yi__Chu_Fei_Ni_Yi_Jing_Shu)
* [升级到最新版本](#Top_of_upgrading_html)
  + [Version 0.10](#Version_0_10_Ban_Ben__0_9_Dao__0)
  + [Version 0.9](#Version_0_9_Cong_Han_Shu_Zhong_F)
  + [Version 0.8](#Version_0_8_Flask_introduced_a_n)
  + [Version 0.7](#Version_0_7_In_Flask_0_7_we_clea)
  + [Version 0.6](#Version_0_6_Flask_0_6_comes_with)
  + [Version 0.5](#Version_0_5_Flask_0_5_is_the_fir)
  + [Version 0.4](#Version_0_4_For_application_deve)
  + [Version 0.3](#Version_0_3_Flask_0_3_introduces)
* [Flask Changelog](#Top_of_changelog_html)
  + [Version 1.0](#Version_1_0__release_date_to_be)
  + [Version 0.10.2](#Version_0_10_2__bugfix_release)
  + [Version 0.10.1](#Version_0_10_1__bugfix_release)
  + [Version 0.10](#Version_0_10_Released_on_June_13)
  + [Version 0.9](#Version_0_9_Released_on_July_1st)
  + [Version 0.8.1](#Version_0_8_1_Bugfix_release__re)
  + [Version 0.8](#Version_0_8_Released_on_Septembe)
  + [Version 0.7.3](#Version_0_7_3_Bugfix_release__re)
  + [Version 0.7.2](#Version_0_7_2_Bugfix_release__re)
  + [Version 0.7.1](#Version_0_7_1_Bugfix_release__re)
  + [Version 0.7](#Version_0_7_Released_on_June_28t)
  + [Version 0.6.1](#Version_0_6_1_Bugfix_release__re)
  + [Version 0.6](#Version_0_6_Released_on_July_27t)
  + [Version 0.5.2](#Version_0_5_2_Bugfix_Release__re)
  + [Version 0.5.1](#Version_0_5_1_Bugfix_Release__re)
  + [Version 0.5](#Version_0_5_Released_on_July_6th)
  + [Version 0.4](#Version_0_4_Released_on_June_18t)
  + [Version 0.3.1](#Version_0_3_1_Bugfix_release__re)
  + [Version 0.3](#Version_0_3_Released_on_May_28th)
  + [Version 0.2](#Version_0_2_Released_on_May_12th)
  + [Version 0.1](#Version_0_1_First_public_preview)
* [许可证](#Top_of_license_html)
  + [作者](#Zuo_Zhe__Flask_is_written_and_ma)
  + [通用许可证定义](#Tong_Yong_Xu_Ke_Zheng_Ding_Yi__X)
  + [Flask License](#Flask_License_Copyright__c__2012)
  + [Flask Artwork License](#Flask_Artwork_License_Copyright)
* [术语表](#Top_of_glossary_html)

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_advanced_foreword_html) | [上一页](#Top_of_index_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 前言

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

## “微” 是什么意思？

“微”(micro) 并不表示你需要把整个 Web 应用塞进单个 Python 文件（虽然确实可以 ），也不意味着 Flask 在功能上有所欠缺。微框架中的“微”意味着 Flask 旨在保持核心 简单且易扩展。Flask 不会替你作出过多选择，比如使用何种数据库。并且，Flask 已经选择好的，比如使用何种模板引擎，是易于修改的。除此之外的一切都取决于你， 因此 Flask 可以满足你的一切需求。

默认情况下，Flask 不包含数据库抽象层、表单验证或是任何其它现有库可以胜任的东 西。作为替代的是，Flask 支持扩展来给应用添加这些功能，如同是在 Flask 自身 中实现。众多的扩展提供了数据库集成、表单验证、上传处理、多种开放认证技术等功 能。Flask 可能是“微小”的，但它已经能在需求繁杂的生产环境中投入使用。

## 配置与惯例

Flask 数目众多的配置选项在初始状况下都有一个明智的默认值，并遵循一些惯例。 例如，按照惯例，模板和静态文件存储在应用的 Python 源代码树下的子目录中，名称 分别为 templates 和 static 。虽然可以更改这个配置，但你通常不必这么做， 尤其是在刚接触 Flask 的时候。

## 与 Flask 共成长

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

随着你的代码库逐渐庞大，你仍会在把握项目设计决策上拥有自由。Flask 会继续尽可 能提供 Python 应该提供的一个非常简单的胶水层。你可以在 SQLAlchemy 或其它数据 库工具中实现更高级的模式，酌情引入非关系型数据持久化，也可以从框架无关的 WSGI （Python 的 Web 接口） 工具中获益。

Flask 里有许多可以定制其行为的钩子。如若你需要更深层次的定制，可以继承 Flask 类。 如果你对此有兴趣，请阅读 [聚沙成塔](#Ju_Sha_Cheng_Ta__Zhe_Li_Shi_Zeng) 章节。如果你好奇 Flask 的设计原 则，请查阅 [Flask 中的设计决策](#design) 章节。

继续阅读 [安装](#An_Zhuang__Flask_Yi_Lai_Yu_Liang) 、 [快速入门](#Kuai_Su_Ru_Men__Po_Bu_Ji_Dai_Yao) 、或 [给有经验程序员的前言](#Gei_You_Jing_Yan_Cheng_Xu_Yuan_D).

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_installation_html) | [上一页](#Top_of_foreword_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 给有经验程序员的前言

本章节是给有其它框架工作经验的程序员，以及有具体或深刻涉及那些的典型用户，

## Flask 中的线程局部变量

Flask 的设计抉择之一就是，简单的任务应该保持简单；它们的实现不应是大量代码的 堆叠，并不应该限制到你。为此，我们选择了一些可能让某些人觉得惊讶或异端的设 计。例如，Flask 内部使用线程局部的对象，这样你不必在请求内的函数间传递对象来 保证线程安全。这个方法很方便，但为依赖注入，或尝试重用使用了与请求挂钩的值的 代码，需要一个有效的请求上下文。

## Web 开发是危险的

请在构建 Web 应用时牢记安全。

如果你编写了一个 Web 应用，你很可能允许用户在你的服务器上注册并留下数据。即 使你是这唯一的用户，也会在应用中留下数据。用户们把数据托付给你，你当然更希望 这些数据被妥善安全地保存。

不幸的是，有许多方式可以让 web 应用的安全措施形同虚设。 Flask 保护你免受现代 Web 应用最常见的一个安全问题的困扰：跨站脚本攻击（XSS）。除非你蓄意把不安全 的 HTML 标记为安全，Flask 和底层的 Jinja2 模板引擎已经为你严防死守。但许多安 全问题依然存在。

本文档会在 web 开发中那些需要注意安全的方面警示你。这些安全考虑中的某些远比 人们想象的复杂，我们有时候低估漏洞被利用的可能性——直到一个聪明的攻击者找出利 用我们程序的方法。并且，不要想着你的应用没有重要到可以吸引攻击者。取决于攻 击的类型，有时候是自动化的僵尸机器搜寻在你数据库中填充垃圾、恶意程序链接或 之类东西的方法。

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

## Python 3 的状态

Python 社区目前处于改善库对 Python 编程语言中迭代支持的进程中。而当前大力改 进中的处境仍有一些问题，使得我们难以迁移到 Python 3 。导致这些问题的原因一部 分是语言中的变更长时间没有复查，一部分也是因为我们没有找出低层 API 应该如何 做出修改来适应 Python 3 中 Unicode 的差异。

一旦应对变更的解决方案出现，Werkzeug 和 Flask 就会立刻迁移到 Python 3 ， 并且我们会提供升级现有应用到 Python 3 的提示。在那之前，我们强烈建议 在开发时使用 Python 2.6 和 2.7 ，并激活 Python 3 警告。如果你计划在近期升级 到 Python 3 ，我们强烈推荐你阅读 [如何编写向后兼容的 Python 代码](http://lucumr.pocoo.org/2011/1/22/forwards-compatible-python/) [http://lucumr.pocoo.org/2011/1/22/forwards-compatible-python/] 。

继续阅读 [安装](#An_Zhuang__Flask_Yi_Lai_Yu_Liang) 或 [快速入门](#Kuai_Su_Ru_Men__Po_Bu_Ji_Dai_Yao) 。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_quickstart_html) | [上一页](#Top_of_advanced_foreword_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 安装

Flask 依赖于两个外部库：[Werkzeug](http://werkzeug.pocoo.org/) [http://werkzeug.pocoo.org/] 和 [Jinja2](http://jinja.pocoo.org/2/) [http://jinja.pocoo.org/2/] 。 Werkzeug 是一个 WSGI （在 web 应用和多种服务器之间开发和部署的标 准 Python 接口) 的工具集Jinja2 负责渲染模板。

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

你首先需要 Python 2.6 或更高的版本，所以请确认有一个最新的 Python 2.x 安装。 在 Python 3 中使用 Flask 请参考： [Python 3 支持](#python3_support) 。

## virtualenv

你在开发中很可能想要使用 virtualenv，如果你拥有生产环境的 shell 权限， 同样会乐于在生产环境中使用它。

virtualenv 解决了什么问题？如果你像我一样喜欢 Python，你可能还要在基于 Flask 的 web 应用以外的项目中使用它。你拥有的项目越多，同时使用不同版本 Python 工作 的可能性越大，或者至少需要不同版本的 Python 库。我们需要面对的是：常常有库会破坏自身的向后兼容性， 然而正常应用零依赖的可能性也不大。当你的项目中的两个或更多出现依赖性冲突时，你会怎么做？

virtualenv 来拯救世界！virtualenv 允许多个版本的 Python 同时存在，对应不同的项目。 它实际上并没有安装独立的 Python 副本，但是它确实提供了一种巧妙的方式来让各项 目环境保持独立。让我们来看看 virtualenv 是怎么工作的。

如果你在 Mac OS X 或 Linux下，下面两条命令可能会适用:

$ sudo easy\_install virtualenv

或更好的:

$ sudo pip install virtualenv

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

$ sudo apt-get install python-virtualenv

如果你所使用的 Windows 上并没有 easy\_install 命令，你必须先安装它。查阅 [Windows 下的 pip 和 distribute](#windows_easy_install) 章节来了解如何安装。之后，运行上述的命令，但是要 去掉 sudo 前缀。

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

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

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

$ . venv/bin/activate

下面的操作适用 Windows:

$ venv\scripts\activate

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

现在你只需要键入以下的命令来激活 virtualenv 中的 Flask:

$ pip install Flask

几秒钟后，一切都搞定了。

## 全局安装

这样也是可以的，虽然我不推荐。只需要以 root 权限运行 pip:

$ sudo pip install Flask

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

## 活在边缘

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

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

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

这会拉取依赖关系并激活 git head 作为 virtualenv 中的当前版本。然后你只需要执 行 git pull origin 来升级到最新版本。

没有 git 时，获取开发版本的替代操作:

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

## Windows 下的 pip 和 distribute

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

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

;C:\Python27\Scripts

于是，你就搞定了！检查它是否正常工作，打开命令提示符并执行 easy\_install 。如果你开启了 Windows Vista 或 Windows 7 中的用户账户控 制，它应该会提示你使用管理员权限。

现在你有了 easy\_install ，你可以用它来安装 pip:

> easy\_install pip

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_index_html_1) | [上一页](#Top_of_installation_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 快速入门

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

## 一个最小的应用

一个最小的 Flask 应用看起来是这样:

from flask import Flask app = Flask(\_\_name\_\_) @app.route('/') def hello\_world(): return 'Hello World!' if \_\_name\_\_ == '\_\_main\_\_': app.run()

把它保存为 hello.py （或是类似的），然后用 Python 解释器来运行。 但是确保你的应用没有命名为 flask.py ，因为这将与 Flask 本身冲突。

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

现在访问 <http://127.0.0.1:5000/> ，你会 看见 hello world 问候。

那么，这段代码做了什么？

1. 首先，我们导入了 [Flask](#class_flask_Flask_import_name__s) 类。这个类的实例将会是我 们的 WSGI 应用程序。
2. 接下来，我们创建一个该类的实例，第一个参数是应用模块或者包的名称。 如果你使用单一的模块（如本例），你应该使用 \_\_name\_\_ ，因为取决于 作为单独应用启动或者模块导入，它的名称将会不同（ '\_\_main\_\_' 相对实际的导入名称）。这是必须的，这样Flask 才会知道到哪里去寻找模板、 静态文件等等。详情参见 [Flask](#class_flask_Flask_import_name__s) 的文档。
3. 然后，我们使用 [route()](#route_rule____options) 装饰器告诉 Flask 什么样的 URL 应该触发我们的函数。
4. 这个函数的名字也用作给特定的函数生成 URL，并且返回我们想要显 示在用户浏览器中的信息。
5. 最后我们用 [run()](#run_host_None__port_None__debug) 函数来让应用运行在本地服务器上。 其中 if \_\_name\_\_ == '\_\_main\_\_': 确保服务器只会在该脚本被 Python 解释器直接执行的时候才会运行，而不是作为模块导入的时候。

要停止服务器，按 Ctrl+C。

可外部访问的服务器

如果你运行服务器，你会注意到它只能从你自己的计算机上访问，网络中其 它任何的地方都不能访问。这是在默认情况，因为在调试模式，用户可以在你的计算机上执行任意 Python 代码。

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

app.run(host='0.0.0.0')

这会让操作系统监听所有公开的IP。

## 调试模式

虽然 [run()](#run_host_None__port_None__debug) 方法适用于本地开发服务器的启动，但是 你每次修改代码后都要手动重启它。这样并不是很好，然而 Flask 可以做得更 好。如果你启用了调试 支持，服务器会在代码变更时自动重新载入，并且如果 发生错误，它会提供一个有用的调试器。

有两种途径来启用调试模式。一种是在应用对象上设置:

app.debug = True app.run()

另一种是作为 run 方法的一个参数传入:

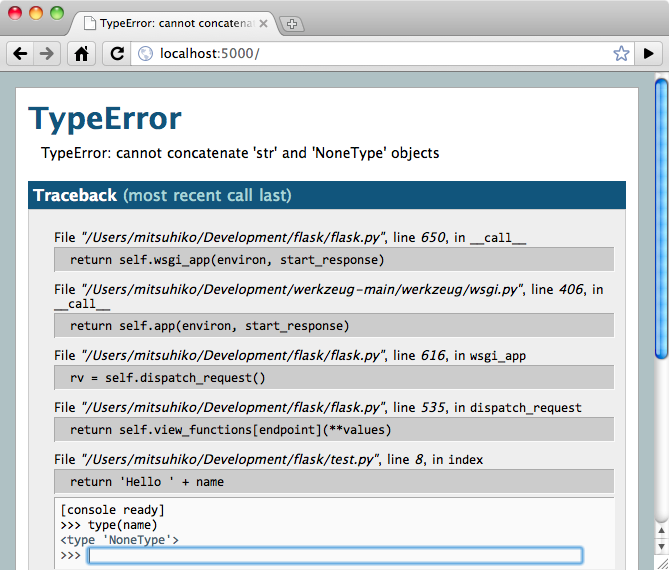
app.run(debug=True)

两种方法的效果完全相同。

注意

尽管交互式调试器不能在 forking 环境（即在生产服务器上使用几乎是不可 能的），它依然允许执行任意代码。这使它成为一个巨大的安全隐患，因此 它 绝对不能用于生产环境 。

运行中的调试器截图：



想用另一个调试器？ 参见 [调试器操作](#Diao_Shi_Qi_Cao_Zuo__Wei_Le_Shen) 。

## 路由

现代 web 应用使用优雅的 URL，这易于人们记住 URL ，这点在面向使用慢网络连 接的移动设备的应用上特别有用。如果可以不访问索引页而直接访问想要的页面，他们多半会喜欢这个页面而再度光顾。

如上所见， [route()](#route_rule____options) 装饰器把一个函数绑定到对应的 URL 上。 这里是一些基本的例子:

@app.route('/') def index(): return 'Index Page' @app.route('/hello') def hello(): return 'Hello World'

但是，不仅如此！你可以构造特定部分动态的 URL，也可以在一个函数上附加多个规则。

### 变量规则

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

@app.route('/user/<username>') def show\_user\_profile(username): # show the user profile for that user return 'User %s' % username @app.route('/post/<int:post\_id>') def show\_post(post\_id): # show the post with the given id, the id is an integer return 'Post %d' % post\_id

现有的转换器如下：

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

唯一的网址 / 重定向行为

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

以这两个规则为例:

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

虽然它们看起来确实相似，但它们结尾斜线的使用在 URL 定义 中不同。 第一种情况中，规范的 URL 指向 projects 尾端有一个斜线。这种感觉 很像在文件系统中的文件夹。访问一个结尾不带斜线的 URL 会被 Flask 重定向到带斜线的规范 URL 去。

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

当用户访问页面时忘记结尾斜线时，这个行为允许关联的 URL 继续工作，并 且与 Apache 和其它的服务器的行为一致。另外，URL 会保持唯一，有助于 避免搜索引擎索引同一个页面两次。

### 构建 URL

如果它能匹配 URL ，那么 Flask 可以生成它们吗？当然可以。你可以使用 [url\_for()](#flask_url_for_endpoint____values) 来给一个特定函数构造 URL。它接受一个函数名作 为第一个参数和一些关键字参数，每个对应 URL 规则的变量部分。未知变量部 分会添加到 URL 末尾作为查询参数。这里是一些例子:

>>> from flask import Flask, url\_for >>> app = Flask(\_\_name\_\_) >>> @app.route('/') ... def index(): pass ... >>> @app.route('/login') ... def login(): pass ... >>> @app.route('/user/<username>') ... def profile(username): pass ... >>> with app.test\_request\_context(): ... print url\_for('index') ... print url\_for('login') ... print url\_for('login', next='/') ... print url\_for('profile', username='John Doe') ... / /login /login?next=/ /user/John%20Doe

（这里也用到了 [test\_request\_context()](#test_request_context__args____kw) 方法，下面会解 释。即使我们正在通过 Python 的 shell 进行交互，它依然会告诉 Flask 像对待请求一样处理。 请看下面的解释。 [局部上下文](#Ju_Bu_Shang_Xia_Wen___Nei_Mu__Ru) ）

为什么你会想要构建 URL 而非在模板中硬编码？这里有三个好理由：

1. 反向构建通常比硬编码更具备描述性。更重要的是，它允许你一次性修改 URL， 而不是到处找 URL 改。
2. URL 构建会显式地处理特殊字符和 Unicode 数据的转义，所以你不需要亲自处理。
3. 如果你的应用不位于 URL 的根路径（比如，在 /myapplication 而不是 / ）， [url\_for()](#flask_url_for_endpoint____values) 会为你妥善地处理这些。

### HTTP 方法

HTTP （web 应用会话的协议）知道访问 URL 的不同方法。默认情况下，路由只回应 GET 请求，但是通过给 [route()](#route_rule____options) 装饰器提供 methods 参数 可以更改这个行为。这里有一些例子:

@app.route('/login', methods=['GET', 'POST']) def login(): if request.method == 'POST': do\_the\_login() else: show\_the\_login\_form()

如果当前是 GET 请求， 它也会自动的为你添加`HEAD`，无需你操心。它会确保 HEAD 请求按照 [HTTP RFC](http://www.ietf.org/rfc/rfc2068.txt) [http://www.ietf.org/rfc/rfc2068.txt] （描述 HTTP 协议的文档）来处理，所以你可以完全忽略这部 分的 HTTP 规范。同样，自从 Flask 0.6 起， OPTIONS 也实现了自动处理。

你不知道一个 HTTP 方法是什么？不必担心，这里快速介绍 HTTP 方法和它们为什么重要：

HTTP 方法（也经常被叫做“谓词”）告诉服务器客户端想对请求的页面 做 什么。 以下都是非常常见的方法：

GET

浏览器告诉服务器，只 获取 页面上的信息并发给我。这是最常用的方法。

HEAD

浏览器告诉服务器获取信息，但是只对 消息头 感兴趣。应用期望像 GET 请求 一样处理它，但是不传递实际内容。在 Flask 中你完全不用处理它，底层的 Werkzeug 库已经替你处理好了。

POST

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

PUT

类似 POST 但是服务器可能触发了存储过程多次，多次覆盖掉旧值。你可能会问这 有什么用，当然这是有原因的。考虑到传输中连接可能会丢失，在这种情况下浏览器 和服务器之间的系统可能安全地第二次接收请求，而不破坏其它东西。使用 POST 不能实现，因为它只会被触发一次。

DELETE

删除给定位置的信息。

OPTIONS

给客户端提供一个快速的途径来弄清这个 URL 支持哪些 HTTP 方法。从 Flask 0.6 开 始，自动实现了它。

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

## 静态文件

动态 web 应用也会需要静态文件，通常是 CSS 和 JavaScript 文件的存放位置。理想情况下， 你已经配置 web 服务器来提供它们，但是在开发中， Flask 也可以做到。只要在你的包中 或模块旁边创建一个名为 static 的文件夹，在应用中使用 /static 即可访问。

给静态文件生成 URL ，使用特殊的 'static' 端点名:

url\_for('static', filename='style.css')

这个文件应该存储在文件系统上的 static/style.css 。

## 模板渲染

在 Python 里生成 HTML 十分无趣，而且相当繁琐，因为你需要自行对 HTML 做转 义来保证应用安全。由于这个原因， Flask 自动配置了 [Jinja2](http://jinja.pocoo.org/2/) [http://jinja.pocoo.org/2/] 模板引擎。

你可以使用 [render\_template()](#flask_render_template_template_n) 方法来渲染模板。你需要做的所有事就是将 模板名和你想作为关键字的参数传入模板的变量。这里有一个描述如何渲染模板的简例:

from flask import render\_template @app.route('/hello/') @app.route('/hello/<name>') def hello(name=None): return render\_template('hello.html', name=name)

Flask 会在 templates 文件夹里寻找模板。所以，如果你的应用是个模块，这个文件 夹在模块的旁边；如果它是一个包，那么这个文件夹在你的包里面:

情况 1: 模块:

/application.py /templates /hello.html

情况 2: 包:

/application /\_\_init\_\_.py /templates /hello.html

对于模板，你可以使用 Jinja2 模板的全部能力。更多信息请见官方的 [Jinja2 模板文档](http://jinja.pocoo.org/2/documentation/templates) [http://jinja.pocoo.org/2/documentation/templates] 。

这里是一个模板实例：

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

在模板里，你也可以访问 [request](#class_flask_request) 、 [session](#class_flask_session) 和 [g](#flask_g) [[1]](#Top_of_quickstart_html) 对象，以及 [get\_flashed\_messages()](#flask_get_flashed_messages_with) 函数。

使用继承，模板会相当有用。如果你想知道继承如何工作，请跳转到 [模板继承](#Mo_Ban_Ji_Cheng__Jinja_Zui_Wei_Q) 模式文档。基本的模板继承使得特定元素（比如页眉、导航 栏和页脚）可以出现在所有的页面。

自动转义默认是开启的，所以如果 name 包含 HTML ，它将会被自动转义。如果你能信任一个 变量，并且你知道它是安全的（例如一个模块把 wiki 标记转换到 HTML ），你可以用 Markup 类或 |safe 过滤器在模板中标记它是安全的。在 Jinja 2 文档中，你会见到更多例子。

这里是一个 Markup 类如何工作的基本介绍:

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

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

|  |  |
| --- | --- |
| [[1]](#_1) | 不确定 [g](#flask_g) 对象是什么？它是你可以按需存储信息的东西， 查看（ [g](#flask_g) ）对象的文档和 [在 Flask 中使用 SQLite 3](#sqlite3) 的文档以获取更多信息。 |

## 访问请求数据

对于 web 应用，对客户端发送给服务器的数据做出反应至关重要。在 Flask 中由全局 的 [request](#class_flask_request) 对象来提供这些信息。如果你有一定的 Python 经验，你 会好奇这个对象怎么可能是全局的，并且 Flask 是怎么还能保证线程安全。答案是上下 文作用域:

### 局部上下文

内幕

如果你想理解它是如何工作以及如何实现测试，请阅读此节，否则可跳过。

Flask 中的某些对象是全局对象，但不是通常的类型。这些对象实际上是给定上下文 的局部对象的代理。虽然很拗口，但实际上很容易理解。

想象一下处理线程的上下文。一个请求传入，web 服务器决定生成一个新线程（或者别 的什么东西，只要这个基础对象可以胜任并发系统，而不仅仅是线程）。当 Flask 开始它 内部请求处理时，它认定当前线程是活动的上下文并绑定当前的应用和 WSGI 环境到那 个上下文（线程）。它以一种智能的方法来实现，以保证一个应用调用另一个应用时 不会中断。

所以这对你来说意味着什么？除非你要做类似单元测试的东西，基本上可以完全忽略 这种情况。你会发现依赖于一个请求对象的代码会突然中断，因为不会有请求对象。解 决方案是自己创建一个请求对象并且把它绑定到上下文。单元测试的最早的解决方案是 使用 [test\_request\_context()](#test_request_context__args____kw) 上下文管理器。结合 with 声 明，它将绑定一个测试请求来进行交互。这里是一个例子:

from flask import request with app.test\_request\_context('/hello', method='POST'): # now you can do something with the request until the # end of the with block, such as basic assertions: assert request.path == '/hello' assert request.method == 'POST'

另一种可能是传递整个 WSGI 环境给 [request\_context()](#request_context_environ) 方法:

from flask import request with app.request\_context(environ): assert request.method == 'POST'

### 请求对象

请求对象在 API 章节有详细的描述（参见 [request](#class_flask_request) ），这里不会赘 述。这里宽泛介绍一些最常用的操作。首先你需要从 flask 模块里导入它:

from flask import request

当前的请求方式通过 method 属性来访问。通过 form 属性来访问表单数据（ POST 或 PUT 请求提交的数 据）。这里有一个上面提到的两个属性的完整实例:

@app.route('/login', methods=['POST', 'GET']) def login(): error = None if request.method == 'POST': if valid\_login(request.form['username'], request.form['password']): return log\_the\_user\_in(request.form['username']) else: error = 'Invalid username/password' # the code below is executed if the request method # was GET or the credentials were invalid return render\_template('login.html', error=error)

当 form 属性中的键值不存在会发生什么？在这种情况，一个特殊的 [KeyError](http://docs.python.org/dev/library/exceptions.html#KeyError) [http://docs.python.org/dev/library/exceptions.html#KeyError] 异常会抛出。你可以像捕获标准的 [KeyError](http://docs.python.org/dev/library/exceptions.html#KeyError) [http://docs.python.org/dev/library/exceptions.html#KeyError] 来捕获它。 如果你不这么做，它会显示一个 HTTP 400 Bad Request 错误页面。所以，很多情况下你并不需 要处理这个问题。

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

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

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

想获取请求对象的完整方法和属性清单，请参阅 [request](#class_flask_request) 的文档。

### 文件上传

你可以很容易的用 Flask 处理文件上传。只需要确保没忘记在你的 HTML 表单中设置 enctype="multipart/form-data" 属性，否则你的浏览器将根本不提交文件。

已上传的文件存储在内存或是文件系统上的临时位置。你可以通过请求对象的 files 属性访问那些文件。每个上传的文件都会存储在那个 字典里。它表现得如同一个标准的 Python file 对象，但它还有一个 [save()](http://werkzeug.pocoo.org/docs/datastructures/#werkzeug.datastructures.FileStorage.save) [http://werkzeug.pocoo.org/docs/datastructures/#werkzeug.datastructures.FileStorage.save] 方法来允许你在服务器的文件 系统上保存它。这里是一个它如何工作的例子:

from flask import request @app.route('/upload', methods=['GET', 'POST']) def upload\_file(): if request.method == 'POST': f = request.files['the\_file'] f.save('/var/www/uploads/uploaded\_file.txt') ...

如果你想知道上传前文件在客户端的文件名，你可以访问 [filename](http://werkzeug.pocoo.org/docs/datastructures/#werkzeug.datastructures.FileStorage.filename) [http://werkzeug.pocoo.org/docs/datastructures/#werkzeug.datastructures.FileStorage.filename] 属性。但请记住永远不 要信任这个值，因为这个值可以伪造。如果你想要使用客户端的文件名来在服务器上 存储文件，把它传递给 Werkzeug 提供的 [secure\_filename()](http://werkzeug.pocoo.org/docs/utils/#werkzeug.utils.secure_filename) [http://werkzeug.pocoo.org/docs/utils/#werkzeug.utils.secure\_filename] 函数:

from flask import request from werkzeug import secure\_filename @app.route('/upload', methods=['GET', 'POST']) def upload\_file(): if request.method == 'POST': f = request.files['the\_file'] f.save('/var/www/uploads/' + secure\_filename(f.filename)) ...

一些更好的例子，查看 [上传文件](#Shang_Chuan_Wen_Jian__E__Shang_C) 模式。

### Cookies

你可以通过 [cookies](#cookies) 属性来访问 cookies 。设置 cookies 通过响应对象的 [set\_cookie](#set_cookie_key__value_____max_ag) 方法。请求对象 的 [cookies](#cookies) 属性是一个客户端提交的所有 cookies 的 字典。如果你想使用会话，请不要直接使用 cookies，请参考 [会话](#Hui_Hua__Chu_Qing_Qiu_Dui_Xiang) 一节。在 Flask 中，已经在 cookies 上增加了一些安全细节。

读取 cookies:

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

存储 cookies:

from flask import make\_response @app.route('/') def index(): resp = make\_response(render\_template(...)) resp.set\_cookie('username', 'the username') return resp

注意 cookies 是设置在响应对象上。由于通常只是从视图函数返回字符串， Flask 会将其转换为响应对象。如果你显式地想要这么做，你可以使用 [make\_response()](#flask_make_response__args) 函数然后修改它。

有时候你会想要在响应对象不存在的时候设置 cookie ，这在利用 [延迟请求回调](#Yan_Chi_Qing_Qiu_Hui_Diao__Flask) 模式时是可行的。

为此也可以参阅 [关于响应](#Guan_Yu_Xiang_Ying__Shi_Tu_Han_S) 。

## 重定向和错误

重定向用户到其它地方你可以使用 [redirect()](#flask_redirect_location__code_30) 函数。放弃请求并 返回错误代码可以使用 [abort()](#flask_abort_code) 函数。这里是一个它们如何工作的 例子:

from flask import abort, redirect, url\_for @app.route('/') def index(): return redirect(url\_for('login')) @app.route('/login') def login(): abort(401) this\_is\_never\_executed()

这是一个相当无意义的例子因为用户会从主页重定向到一个不能访问的页面（401意 味着禁止访问），但是它说明了重定向如何工作。

默认情况下，每个错误代码会显示一个黑白错误页面。如果你想定制错误页面，可 以使用 [errorhandler()](#errorhandler__args____kwargs) 装饰器:

from flask import render\_template @app.errorhandler(404) def page\_not\_found(error): return render\_template('page\_not\_found.html'), 404

注意 [render\_template()](#flask_render_template_template_n) 调用之后的 404 。这告诉 Flask 该 页的错误代码应是 404 ，即没有找到。默认的 200 被假定为：一切正常。

## 关于响应

视图函数的返回值会被自动转换为一个响应对象。如果返回值是一个字符串， 它被转换为响应主体为该字符串、状态码为 200 OK 、 MIME 类型为 text/html 的响应对象。Flask 把返回值转换为响应对象的逻辑如下：

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

如果你想在视图里掌控上述步骤结果的响应对象，你可以使用 [make\_response()](#flask_make_response__args) 函数。

想象你有这样一个视图:

@app.errorhandler(404) def not\_found(error): return render\_template('error.html'), 404

你只需要用 [make\_response()](#flask_make_response__args) 封装返回表达式，获取结果对象并修 改，然后返回它:

@app.errorhandler(404) def not\_found(error): resp = make\_response(render\_template('error.html'), 404) resp.headers['X-Something'] = 'A value' return resp

## 会话

除请求对象之外，还有 [session](#class_flask_session) 对象允许你在不同请求间存储特 定用户的信息。这是在 cookies 的基础上实现的，并且在 cookies 中使用加密的 签名。这意味着用户可以查看你 cookie 的内容，但是不能修改它，除非它知道签 名的密钥。

要使用会话，你需要设置一个密钥。这里介绍会话如何工作:

from flask import Flask, session, redirect, url\_for, escape, request app = Flask(\_\_name\_\_) @app.route('/') def index(): if 'username' in session: return 'Logged in as %s' % escape(session['username']) return 'You are not logged in' @app.route('/login', methods=['GET', 'POST']) def login(): if request.method == 'POST': session['username'] = request.form['username'] return redirect(url\_for('index')) return ''' <form action="" method="post"> <p><input type=text name=username> <p><input type=submit value=Login> </form> ''' @app.route('/logout') def logout(): # remove the username from the session if it's there session.pop('username', None) return redirect(url\_for('index')) # set the secret key. keep this really secret: app.secret\_key = 'A0Zr98j/3yX R~XHH!jmN]LWX/,?RT'

这里提到的 [escape()](#flask_escape_s____markup) 可以在你不使用模板引擎的时候做转义（如同 本例）。

如何生成一个强壮的密钥

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

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

把这个值复制粘贴到你的代码，你就搞定了密钥。

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

## 消息闪现

良好的应用和用户界面全部涉及反馈。如果用户得不到足够的反馈，他们很可能开始 厌恶这个应用。 Flask 提供一种消息闪现系统给用户反馈的简单方法。 消息闪现系统通常会在请求结束时记录信息，并在下一个（且仅在下一个）请求中 访问。通常结合模板布局来展现消息。

使用 [flash()](#flask_flash_message__category__m) 方法可以闪现一条消息。要掌控消息本身，使用 [get\_flashed\_messages()](#flask_get_flashed_messages_with) 函数，并且在模板中也可以使用。完整的例 子请查阅 [消息闪现](#Xiao_Xi_Shan_Xian__Hao_De_Ying_Y) 部分。

## 日志记录

0.3 新版功能.

有时候你处于一种境地，你处理的数据本应该是正确的，但实际上不是。比如你有一些 客户端代码向服务器发送请求，但请求显然是畸形的。这可能是用户篡改了数据，或 是客户端代码的失败。大多数情况下，正常地返回 400 Bad Request 就可以了， 但是有时不这么做，并且代码要继续运行。

你可能依然想要记录发生了什么不对劲。这时日志记录就派上了用场。Flask 0.3 开始已经预置了日志系统。

这里有一些日志调用的例子:

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

附带的 [logger](#logger) 是一个标准日志类 [Logger](http://docs.python.org/dev/library/logging.html#logging.Logger) [http://docs.python.org/dev/library/logging.html#logging.Logger] ，所以更多信息请见 [logging 文档](http://docs.python.org/library/logging.html) [http://docs.python.org/library/logging.html] 。

## 整合 WSGI 中间件

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

from werkzeug.contrib.fixers import LighttpdCGIRootFix app.wsgi\_app = LighttpdCGIRootFix(app.wsgi\_app)

## 部署到 Web 服务器

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

* [在 Heroku 上部署 Flask](http://devcenter.heroku.com/articles/python) [http://devcenter.heroku.com/articles/python]
* [在 dotCloud 上部署 Flask](http://docs.dotcloud.com/services/python/) [http://docs.dotcloud.com/services/python/] 附 [Flask 的具体说明](http://flask.pocoo.org/snippets/48/) [http://flask.pocoo.org/snippets/48/]

托管 Flask 应用的其它选择:

* [在 Webfaction 上部署 Flask](http://flask.pocoo.org/snippets/65/) [http://flask.pocoo.org/snippets/65/]
* [在 Google App Engine 上部署 Flask](https://github.com/kamalgill/flask-appengine-template) [https://github.com/kamalgill/flask-appengine-template]
* [用 Localtunnel 共享你的本地服务器](http://flask.pocoo.org/snippets/89/) [http://flask.pocoo.org/snippets/89/]

如果你管理自己的主机并且想要亲自运行，参见 [部署选择](#Bu_Shu_Xuan_Ze__Qu_Jue_Yu_Ni_Xia) 章节。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_introduction_html) | [上一页](#Top_of_quickstart_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 教程

你想要用 Python 和 Flask 开发一个应用？这里你将有机会通过实例来学习。 在本教程中，我们会创建一个简单的微博客应用。它只支持单用户和纯文本的 条目，并且没有推送或是评论，但是它仍然有你开始时需要的一切。我们使 用 Flask ，采用在 Python 方框中产生的 SQLite 数据库 ，所以你不会需 要其它的东西。

如果你想提前获得完整源码或是用于对照，请检查 [示例源码](http://github.com/mitsuhiko/flask/tree/master/examples/flaskr/) [http://github.com/mitsuhiko/flask/tree/master/examples/flaskr/]

* [介绍 Flaskr](#Top_of_introduction_html)
* [步骤 0: 创建文件夹](#Top_of_folders_html)
* [步骤 1: 数据库模式](#Top_of_schema_html)
* [步骤 2: 应用设置代码](#Top_of_setup_html)
* [步骤 3: 创建数据库](#Top_of_dbinit_html)
* [步骤 4: 请求数据库连接](#Top_of_dbcon_html)
* [步骤 5: 视图函数](#Top_of_views_html)
  + [显示条目](#Xian_Shi_Tiao_Mu__Zhe_Ge_Shi_Tu)
  + [添加条目](#Tian_Jia_Tiao_Mu__Zhe_Ge_Shi_Tu)
  + [登入和登出](#Deng_Ru_He_Deng_Chu__Zhe_Xie_Han)
* [步骤 6: 模板](#Top_of_templates_html)
  + [layout.html](#layout_html_Zhe_Ge_Mo_Ban_Bao_Ha)
  + [show\_entries.html](#show_entries_html_Zhe_Ge_Mo_Ban)
  + [login.html](#login_html_Zui_Hou_Shi_Deng_Ru_M)
* [步骤 7: 添加样式](#Top_of_css_html)
* [福利: 应用测试](#Top_of_testing_html)

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_folders_html) | [上一页](#Top_of_index_html_1) | [Flask 0.10.1 文档](#Top_of_index_html) » [教程](#Top_of_index_html_1) »

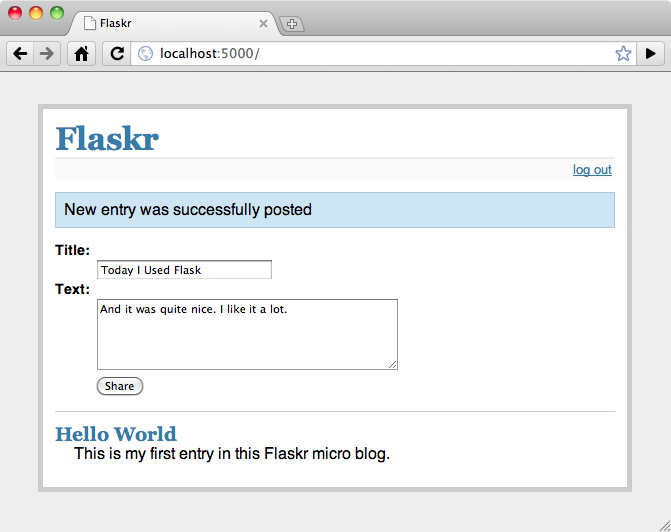
# 介绍 Flaskr

这里，我们把我们的博客应用称为 flaskr ，也可以选一个不那么 web 2.0 的名字 ;) 。基本上，我们希望它能做如下的事情：

1. 根据配置文件里的凭证允许用户登入登出。只支持一个用户。
2. 当用户登入后，可以向页面添加条目。条目标题是纯文本，正文可以是一些 HTML 。因假定信任这里的用户，这部分 HTML 不做审查。
3. 页面倒序显示所有条目（后来居上），并且用户登入后可以在此添加新条目。

我们将会在应用中直接使用 SQLite3 ，因为它对这种规模的应用足够适用。对于更大型的应用，就有必要使用 [SQLAlchemy](http://www.sqlalchemy.org/) [http://www.sqlalchemy.org/] 更加智能地处理数据库连接、允许你一次连接不同的关系数据库等等。你也可以考虑流行的 NoSQL 数据库，如果你的数据更适合它们。

这里是一个应用最终效果的截图:



继续 [步骤 0: 创建文件夹](#Bu_Zou__0__Chuang_Jian_Wen_Jian) 。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_schema_html) | [上一页](#Top_of_introduction_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [教程](#Top_of_index_html_1) »

# 步骤 0: 创建文件夹

在我们开始之前，让我们创建这个应用需要的文件夹:

/flaskr /static /templates

flaskr 文件夹不是一个 python 包，只是我们放置文件的地方。在接下来的步骤中，我们会直接把数据库模式和主模块放在这个目录中。 可以应用用户通过 HTTP 访问`static` 文件夹中的文件，这也是 css 和 javascript 文件存放的地方。在 templates 文件夹里， Flask 会寻找 [Jinja2](http://jinja.pocoo.org/2/) [http://jinja.pocoo.org/2/] 模板，你之后教程中创建的模板会放在这一文件夹里。

继续 [步骤 1: 数据库模式](#Bu_Zou__1__Shu_Ju_Ku_Mo_Shi__Sho).

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_setup_html) | [上一页](#Top_of_folders_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [教程](#Top_of_index_html_1) »

# 步骤 1: 数据库模式

首先我们要创建数据库模式。对于这个应用只有一张表就足够了，并且我们只需要支持 SQLite ，所以很简单。只需要把下面的内容放进一个名为 schema.sql 的文件，放在刚才创建的 flaskr 文件夹中:

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

这个模式由一个名为 entries 的表组成，表中每列包含一个 id 、 一个 title 和 一个 text 。 id 是一个自增的整数，也是主键；其余的两个是字符串，且为非空。

继续 [步骤 2: 应用设置代码](#Bu_Zou__2__Ying_Yong_She_Zhi_Dai).

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_dbinit_html) | [上一页](#Top_of_schema_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [教程](#Top_of_index_html_1) »

# 步骤 2: 应用设置代码

现在我们已经有了数据库模式，我们可以创建应用的模块了。让我们叫它 flaskr.py ， 并放置在 flaskr 目录下。为面向初学者，我们会添加所有需要的导入像配置的章节中 一样。对于小应用，直接把配置放在主模块里，正如我们现在要做的一样，是可行的。但 是，一个更简洁的方案是创建独立的 .ini 或 .py 文件，并载入或导入里面的值。

flaskr.py 中

# all the imports import sqlite3 from flask import Flask, request, session, g, redirect, url\_for, \ abort, render\_template, flash # configuration DATABASE = '/tmp/flaskr.db' DEBUG = True SECRET\_KEY = 'development key' USERNAME = 'admin' PASSWORD = 'default'

接下来我们要创建真正的应用，并且在同一个文件 flaskr.py 中配置并初始化:

# create our little application :) app = Flask(\_\_name\_\_) app.config.from\_object(\_\_name\_\_)

[from\_object()](#from_object_obj) 会遍历给定的对象（如果它是一个字符串，则 会导入它），搜寻里面定义的全部大写的变量。这种情况，配置文件就是我们上面写 的几行代码。你也可以将他们分开存储到多个文件。

从一个配置文件导入配置通常是个好主意。 [from\_envvar()](#from_envvar_variable_name__silen) 也能做到，用它替换上面的 [from\_object()](#from_object_obj) 一行:

app.config.from\_envvar('FLASKR\_SETTINGS', silent=True)

这种方法我们可以设置一个名为 FLASKR\_SETTINGS 环境变量来设定一个配置 文件载入后是否覆盖默认值。静默开关告诉 Flask 不去关心这个环境变量键值是否存在。

我们需要 secret\_key 来保证客户端会话的安全。一个尽可能难猜测，尽可能复杂的密 钥是正确的选择。调试标志关系交互式调试器的开启。 永远不要在生产系统中激活调试 模式 ，因为它将允许用户在服务器上执行代码。

我们还添加了一个快速连接到指定数据库的方法，这个方法用于在请求时打开一个连接， 并且在交互式 Python shell 和脚本中也能使用。这对以后很方便。

def connect\_db(): return sqlite3.connect(app.config['DATABASE'])

最后，如果我们想要把那个文件当做独立应用来运行，我们只需在服务器启动文件的末 尾添加这一行:

if \_\_name\_\_ == '\_\_main\_\_': app.run()

如此我们便可以顺利开始运行这个应用，使用如下命令:

python flaskr.py

你将会看见有消息提示你可以访问服务器的地址。 You will see a message telling you that server has started along with the address at which you can access it.

当你在浏览器中访问服务器获得一个 404 page not found 错误时，是因为我们还没有 任何视图。我们之后再来关注这些。首先我们应该让数据库工作起来。

外部可见的服务器

想要你的服务器公开可见？ [外部可见的服务器](#Ke_Wai_Bu_Fang_Wen_De_Fu_Wu_Qi) 一节有更多信息。

继续 [步骤 3: 创建数据库](#Bu_Zou__3__Chuang_Jian_Shu_Ju_Ku) 。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_dbcon_html) | [上一页](#Top_of_setup_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [教程](#Top_of_index_html_1) »

# 步骤 3: 创建数据库

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

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

sqlite3 /tmp/flaskr.db < schema.sql

这种方法的缺点是需要安装 sqlite3 命令，而并不是每个系统都有安装。而且你必 须提供数据库的路径，否则将报错。添加一个函数来对初始化数据库是个不错的想法。

如果你想这么做，你首先要从 contextlib 包中导入 [contextlib.closing()](http://docs.python.org/dev/library/contextlib.html#contextlib.closing) [http://docs.python.org/dev/library/contextlib.html#contextlib.closing] 函数。如果你想使用 Python 2.5 ，那么必须先启用 with 声明（ \_\_future\_\_ 导入必须先于其它的导入）:

from \_\_future\_\_ import with\_statement from contextlib import closing

接下来，我们可以创建一个名为 init\_db 的函数来初始化数据库。为此，我们可以 使用之前定义的 connect\_db 函数。只需要在 connect\_db 函数后面添加这个函 数:

def init\_db(): with closing(connect\_db()) as db: with app.open\_resource('schema.sql') as f: db.cursor().executescript(f.read()) db.commit()

[closing()](http://docs.python.org/dev/library/contextlib.html#contextlib.closing) [http://docs.python.org/dev/library/contextlib.html#contextlib.closing] 助手函数允许我们在 with 块中保持数据库连接可 用。应用对象的 [open\_resource()](#open_resource_resource__mode__rb) 方法在其方框外也支持这个 功能，因此可以在 with 块中直接使用。这个函数从资源位置（你的 flaskr 文 件夹）中打开一个文件，并且允许你读取它。我们在这里用它在数据库连接上执行一 个脚本。

当我们连接到数据库时会得到一个数据库连接对象（这里命名它为 db ），这个对 象提供给我们一个数据库指针。指针上有一个可以执行完整脚本的方法。最后我们不 显式地提交更改， SQLite 3 或者其它事务数据库不会这么做。

现在可以在 Python shell 里创建数据库，导入并调用刚才的函数:

>>> from flaskr import init\_db >>> init\_db()

故障排除

如果你获得了一个表无法找到的异常，请检查你确实调用了 init\_db 函数并且 表的名称是正确的（如单数复数混淆）。

继续 [步骤 4: 请求数据库连接](#Bu_Zou__4__Qing_Qiu_Shu_Ju_Ku_Li)

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_views_html) | [上一页](#Top_of_dbinit_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [教程](#Top_of_index_html_1) »

# 步骤 4: 请求数据库连接

现在我们知道如何在建立数据库连接并且如何执行脚本，但是我们如何能优雅的在请求 中这么做？所有的函数都需要数据库连接，所以在请求之前初始化，在请求结束后自动 关闭就很有意义。

Flask 允许我们用 [before\_request()](#before_request__args____kwargs) 、 [after\_request()](#after_request__args____kwargs) 和 [teardown\_request()](#teardown_request__args____kwargs) 装饰器来实现这个功能:

@app.before\_request def before\_request(): g.db = connect\_db() @app.teardown\_request def teardown\_request(exception): g.db.close()

用 [before\_request()](#before_request__args____kwargs) 装饰的函数会在请求前调用，它没有参 数。用 [after\_request()](#after_request__args____kwargs) 装饰的函数在请求结束后调用，需要 传入响应。它们必须返回那个响应对象或是不同的响应对象。但当异常抛出时，它们 不一定会被执行，这时可以使用 [teardown\_request()](#teardown_request__args____kwargs) 装饰器， 它装饰的函数将在响应构造后执行，并不允许修改请求，返回的值会被忽略。如果在 请求已经被处理的时候抛出异常，它会被传递到每个函数，否则会传入一个 None 。

我们把当前的数据库连接保存在 Flask 提供的 [g](#flask_g) 特殊对象中。这个 对象只能保存一次请求的信息，并且在每个函数里都可用。不要用其它对象来保存信 息，因为在多线程环境下将不可行。特殊的对象 [g](#flask_g) 在后台有一些神 奇的机制来保证它在做正确的事情。

继续 [步骤 5: 视图函数](#Bu_Zou__5__Shi_Tu_Han_Shu__Xian) 。

提示

我该把代码放在哪？

如果你一直遵循教程，你应该会问从这步到以后的步骤，代码放在什么地方。逻辑上 应该按照模块来组织函数，即把你新的 before\_request 和 teardown\_request 装饰的函数放在之前的 init\_db 函数下面（逐行遵照教程）。

如果你需要一个时刻来找到你的方位，看一下 [示例源码](http://github.com/mitsuhiko/flask/tree/master/examples/flaskr/) [http://github.com/mitsuhiko/flask/tree/master/examples/flaskr/] 是怎么组织的。在 Flask 中，你可以把你应用所有的代码放在一个 Python 模块里。但你无需这么做， 并且在你的应用 [规模扩大](#Da_Xing_Ying_Yong__Dui_Yu_Bi_Jia) 的时候，这显然不妥。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_templates_html) | [上一页](#Top_of_dbcon_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [教程](#Top_of_index_html_1) »

# 步骤 5: 视图函数

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

## 显示条目

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

视图函数会将条目作为字典传递给 show\_entries.html 模板，并返回渲染结果:

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

## 添加条目

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

@app.route('/add', methods=['POST']) def add\_entry(): if not session.get('logged\_in'): abort(401) g.db.execute('insert into entries (title, text) values (?, ?)', [request.form['title'], request.form['text']]) g.db.commit() flash('New entry was successfully posted') return redirect(url\_for('show\_entries'))

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

安全提示

确保像上面例子中一样，使用问号标记来构建 SQL 语句。否则，当你使用格式化 字符串构建 SQL 语句时，你的应用容易遭受 SQL 注入。 更多请见 [在 Flask 中使用 SQLite 3](#sqlite3) 。

## 登入和登出

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

@app.route('/login', methods=['GET', 'POST']) def login(): error = None if request.method == 'POST': if request.form['username'] != app.config['USERNAME']: error = 'Invalid username' elif request.form['password'] != app.config['PASSWORD']: error = 'Invalid password' else: session['logged\_in'] = True flash('You were logged in') return redirect(url\_for('show\_entries')) return render\_template('login.html', error=error)

登出函数，做相反的事情，从会话中删除 logged\_in 键。我们这里使用了一个 简洁的方法：如果你使用字典的 [pop()](http://docs.python.org/dev/library/stdtypes.html#dict.pop) [http://docs.python.org/dev/library/stdtypes.html#dict.pop] 方法并传入第二个参数（默认）， 这个方法会从字典中删除这个键，如果这个键不存在则什么都不做。这很有用，因为 我们不需要检查用户是否已经登入。

@app.route('/logout') def logout(): session.pop('logged\_in', None) flash('You were logged out') return redirect(url\_for('show\_entries'))

继续 [步骤 6: 模板](#Bu_Zou__6__Mo_Ban__Jie_Xia_Lai_W) 。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_css_html) | [上一页](#Top_of_views_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [教程](#Top_of_index_html_1) »

# 步骤 6: 模板

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

我们也会使用模板继承在网站的所有页面中重用布局。

将下面的模板放在 templates 文件夹里:

## layout.html

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

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

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

## show\_entries.html

这个模板继承了上面的 layout.html 模板来显示消息。注意 for 循环会遍历并输出 所有 [render\_template()](#flask_render_template_template_n) 函数传入的消息。我们还告诉表单使用 HTTP 的 POST 方法提交信息到 add\_entry 函数:

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

## login.html

最后是登入模板，只是简单地显示一个允许用户登入的表单:

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

继续 [步骤 7: 添加样式](#Bu_Zou__7__Tian_Jia_Yang_Shi__Xi) 。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_testing_html) | [上一页](#Top_of_templates_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [教程](#Top_of_index_html_1) »

# 步骤 7: 添加样式

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

body { font-family: sans-serif; background: #eee; } a, h1, h2 { color: #377BA8; } h1, h2 { font-family: 'Georgia', serif; margin: 0; } h1 { border-bottom: 2px solid #eee; } h2 { font-size: 1.2em; } .page { margin: 2em auto; width: 35em; border: 5px solid #ccc; padding: 0.8em; background: white; } .entries { list-style: none; margin: 0; padding: 0; } .entries li { margin: 0.8em 1.2em; } .entries li h2 { margin-left: -1em; } .add-entry { font-size: 0.9em; border-bottom: 1px solid #ccc; } .add-entry dl { font-weight: bold; } .metanav { text-align: right; font-size: 0.8em; padding: 0.3em; margin-bottom: 1em; background: #fafafa; } .flash { background: #CEE5F5; padding: 0.5em; border: 1px solid #AACBE2; } .error { background: #F0D6D6; padding: 0.5em; }

继续 [福利: 应用测试](#Fu_Li___Ying_Yong_Ce_Shi__Xian_Z) 。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_templating_html) | [上一页](#Top_of_css_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [教程](#Top_of_index_html_1) »

# 福利: 应用测试

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

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_testing_html_1) | [上一页](#Top_of_testing_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 模板

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

本节只是快速地介绍 Jinja2 是如何集成到 Flask 中的。更多关于 Jinja2 语法本身的信息， 请参考官方文档 [Jinja2 模板引擎](http://jinja.pocoo.org/2/documentation/templates) [http://jinja.pocoo.org/2/documentation/templates] 。

## Jinja 配置

Jinja 2 默认配置如下:

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

## 标准上下文

下面的全局变量默认在 Jinja2 模板中可用:

config

当前的配置对象 (flask.config)

0.6 新版功能.

在 0.10 版更改: 现在这总是可用的，甚至在导入的模版里。

request

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

session

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

g

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

url\_for()

[flask.url\_for()](#flask_url_for_endpoint____values) 函数

get\_flashed\_messages()

[flask.get\_flashed\_messages()](#flask_get_flashed_messages_with) 函数

Jinja 上下文行为

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

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

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

与上下文中一起（with context）导入的方式如下:

{% from '\_helpers.html' import my\_macro with context %}

## 标准过滤器

这些过滤器在 Jinja2 中可用，也是 Jinja2 自带的过滤器:

tojson()

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

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

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

## 控制自动转义

自动转义的概念是自动转义特殊字符。 HTML （或 XML ，因此也有 XHTML ）意义下 的特殊字符是 & ， > ， < ， " 以及 ' 。因为这些字符在 文档中表示它们特定的含义，如果你想在文本中使用它们，应该把它们替换成相应 的“实体”。不这么做不仅会导致用户疲于在文本中使用这些字符，也会导致安全问题。 （见 [跨站脚本攻击（XSS）](#Kua_Zhan_Jiao_Ben_Gong_Ji__XSS) ）

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

我们有三种可行的解决方案:

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

在模板中禁用自动转义系统，可以使用 {%autoescape %} 块:

{% autoescape false %} <p>autoescaping is disabled here <p>{{ will\_not\_be\_escaped }} {% endautoescape %}

无论何时，都请务必格外小心这里的变量。

## 注册过滤器

如果你要在 Jinja2 中注册你自己的过滤器，你有两种方法。你可以把它们手动添加到 应用的 [jinja\_env](#jinja_env) 或者使用 [template\_filter()](#template_filter__args____kwargs) 装饰器。

下面两个例子作用相同，都是反转一个对象:

@app.template\_filter('reverse') def reverse\_filter(s): return s[::-1] def reverse\_filter(s): return s[::-1] app.jinja\_env.filters['reverse'] = reverse\_filter

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

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

## 上下文处理器

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

@app.context\_processor def inject\_user(): return dict(user=g.user)

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

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

@app.context\_processor def utility\_processor(): def format\_price(amount, currency=u'€'): return u'{0:.2f}{1}.format(amount, currency) return dict(format\_price=format\_price)

上面的上下文处理器使得 format\_price 函数在所有模板中可用:

{{ format\_price(0.33) }}

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

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_errorhandling_html) | [上一页](#Top_of_templating_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 测试 Flask 应用

没有经过测试的东西都是不完整的

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

Flask 提供了一种方法用于测试您的应用，那就是将 Werkzeug 测试 [Client](http://werkzeug.pocoo.org/docs/test/#werkzeug.test.Client) [http://werkzeug.pocoo.org/docs/test/#werkzeug.test.Client] 暴露出来，并且为您操作这些内容 的本地上下文变量。然后您就可以将自己最喜欢的测试解决方案应用于其上了。 在这片文档中，我们将会使用Python自带的 [unittest](http://docs.python.org/dev/library/unittest.html#module-unittest) [http://docs.python.org/dev/library/unittest.html#module-unittest] 包。

## 应用程序

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

## 测试的大框架

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

import os import flaskr import unittest import tempfile class FlaskrTestCase(unittest.TestCase): def setUp(self): self.db\_fd, flaskr.app.config['DATABASE'] = tempfile.mkstemp() flaskr.app.config['TESTING'] = True self.app = flaskr.app.test\_client() flaskr.init\_db() def tearDown(self): os.close(self.db\_fd) os.unlink(flaskr.app.config['DATABASE']) if \_\_name\_\_ == '\_\_main\_\_': unittest.main()

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

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

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

如果我们运行这套测试，我们应该会得到如下的输出:

$ python flaskr\_tests.py ---------------------------------------------------------------------- Ran 0 tests in 0.000s OK

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

## 第一个测试

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

class FlaskrTestCase(unittest.TestCase): def setUp(self): self.db\_fd, flaskr.app.config['DATABASE'] = tempfile.mkstemp() self.app = flaskr.app.test\_client() flaskr.init\_db() def tearDown(self): os.close(self.db\_fd) os.unlink(flaskr.DATABASE) def test\_empty\_db(self): rv = self.app.get('/') assert 'No entries here so far' in rv.data

注意到我们的测试函数以 test 开头，这允许 [unittest](http://docs.python.org/dev/library/unittest.html#module-unittest) [http://docs.python.org/dev/library/unittest.html#module-unittest] 模块自动 识别出哪些方法是一个测试方法，并且运行它。

通过使用 self.app.get 我们可以发送一个 HTTP GET 请求给应用的 某个给定路径。返回值将会是一个 [response\_class](#response_class) 对象。我们可以使用 [data](http://werkzeug.pocoo.org/docs/wrappers/#werkzeug.wrappers.BaseResponse.data) [http://werkzeug.pocoo.org/docs/wrappers/#werkzeug.wrappers.BaseResponse.data] 属性 来检查程序的返回值(以字符串类型)。在这里，我们检查 'No entries here so far' 是不是输出内容的一部分。

再次运行，您应该看到一个测试成功通过了:

$ python flaskr\_tests.py . ---------------------------------------------------------------------- Ran 1 test in 0.034s OK

## 登陆和登出

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

将如下两个方法加入到您的 FlaskrTestCase 类:

def login(self, username, password): return self.app.post('/login', data=dict( username=username, password=password ), follow\_redirects=True) def logout(self): return self.app.get('/logout', follow\_redirects=True)

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

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

## 测试消息的添加

我们同时应该测试消息的添加功能是否正常，添加一个新的 测试方法如下:

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

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

运行这个测试，我们应该得到三个通过的测试:

$ python flaskr\_tests.py ... ---------------------------------------------------------------------- Ran 3 tests in 0.332s OK

关于请求的头信息和状态值等更复杂的测试，请参考 [MiniTwit Example](http://github.com/mitsuhiko/flask/tree/master/examples/minitwit/) [http://github.com/mitsuhiko/flask/tree/master/examples/minitwit/] ，在这个例子的源代码里包含 一套更长的测试。

## 其他测试技巧

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

app = flask.Flask(\_\_name\_\_) with app.test\_request\_context('/?name=Peter'): assert flask.request.path == '/' assert flask.request.args['name'] == 'Peter'

所有其他的和上下文绑定的对象都可以使用同样的方法访问。

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

注意，尽管你在使用一个测试用的请求环境，函数 [before\_request()](#before_request__args____kwargs) 以及 [after\_request()](#after_request__args____kwargs) 都不会自动运行。 然而，[teardown\_request()](#teardown_request__args____kwargs) 函数在 测试请求的上下文离开 with 块的时候会执行。如果您 希望 [before\_request()](#before_request__args____kwargs) 函数仍然执行。 您需要手动调用 [preprocess\_request()](#preprocess_request) 方法:

app = flask.Flask(\_\_name\_\_) with app.test\_request\_context('/?name=Peter'): app.preprocess\_request() ...

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

如果您希望调用 [after\_request()](#after_request__args____kwargs) 函数， 您需要使用 [process\_response()](#process_response_response) 方法。 这个方法需要您传入一个 response 对象:

app = flask.Flask(\_\_name\_\_) with app.test\_request\_context('/?name=Peter'): resp = Response('...') resp = app.process\_response(resp) ...

这通常不是很有效，因为这时您可以直接转向使用 测试客户端。

## 伪造资源和上下文

0.10 新版功能.

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

def get\_user(): user = getattr(g, 'user', None) if user is None: user = fetch\_current\_user\_from\_database() g.user = user return user

对于测试，这样易于从外部覆盖这个用户，而不用修改代码。连接 [flask.appcontext\_pushed](#flask_appcontext_pushed) 信号可以很容易地完成这个任务:

from contextlib import contextmanager from flask import appcontext\_pushed @contextmanager def user\_set(app, user): def handler(sender, \*\*kwargs): g.user = user with appcontext\_pushed.connected\_to(handler, app): yield

并且之后使用它:

from flask import json, jsonify @app.route('/users/me') def users\_me(): return jsonify(username=g.user.username) with user\_set(app, my\_user): with app.test\_client() as c: resp = c.get('/users/me') data = json.loads(resp.data) self.assert\_equal(data['username'], my\_user.username)

## 保存上下文

0.4 新版功能.

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

app = flask.Flask(\_\_name\_\_) with app.test\_client() as c: rv = c.get('/?tequila=42') assert request.args['tequila'] == '42'

如果您仅仅使用 [test\_client()](#test_client_use_cookies_True) 方法，而 不使用 with 代码块， assert 断言会失败，因为 request 不再可访问(因为您试图在非真正请求中时候访问它)。

## 访问和修改 Sessions

0.8 新版功能.

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

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

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

with app.test\_client() as c: with c.session\_transaction() as sess: sess['a\_key'] = 'a value' # once this is reached the session was stored

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

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_config_html) | [上一页](#Top_of_testing_html_1) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 记录应用错误

0.3 新版功能.

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

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

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

但是你还可以做些别的，我们会介绍一些更好的设置来应对错误。

## 错误邮件

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

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

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

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

那么刚刚发生了什么？我们创建了一个新的 [SMTPHandler](http://docs.python.org/dev/library/logging.handlers.html#logging.handlers.SMTPHandler) [http://docs.python.org/dev/library/logging.handlers.html#logging.handlers.SMTPHandler] 来用监听 127.0.0.1 的邮件服务器 向所有的 ADMINS 发送发件人为 server-error@example.com ，主题 为 “YourApplication Failed” 的邮件。如果你的邮件服务器需要凭证，这些功能也 被提供了。详情请见 [SMTPHandler](http://docs.python.org/dev/library/logging.handlers.html#logging.handlers.SMTPHandler) [http://docs.python.org/dev/library/logging.handlers.html#logging.handlers.SMTPHandler] 的文档。

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

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

## 记录到文件

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

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

* [FileHandler](http://docs.python.org/dev/library/logging.handlers.html#logging.FileHandler) [http://docs.python.org/dev/library/logging.handlers.html#logging.FileHandler] - 在文件系统上记录日志
* [RotatingFileHandler](http://docs.python.org/dev/library/logging.handlers.html#logging.handlers.RotatingFileHandler) [http://docs.python.org/dev/library/logging.handlers.html#logging.handlers.RotatingFileHandler] - 在文件系统上记录日志， 并且当消息达到一定数目时，会滚动记录
* [NTEventLogHandler](http://docs.python.org/dev/library/logging.handlers.html#logging.handlers.NTEventLogHandler) [http://docs.python.org/dev/library/logging.handlers.html#logging.handlers.NTEventLogHandler] - 记录到 Windows 系统中的系 统事件日志。如果你在 Windows 上做开发，这就是你想要用的。
* [SysLogHandler](http://docs.python.org/dev/library/logging.handlers.html#logging.handlers.SysLogHandler) [http://docs.python.org/dev/library/logging.handlers.html#logging.handlers.SysLogHandler] - 发送日志到 Unix 的系统日志

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

if not app.debug: import logging from themodule import TheHandlerYouWant file\_handler = TheHandlerYouWant(...) file\_handler.setLevel(logging.WARNING) app.logger.addHandler(file\_handler)

## 控制日志格式

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

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

这里有一些配置实例:

### 邮件

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

### 日志文件

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

### 复杂日志格式

这里给出一个用于格式化字符串的格式变量列表。注意这个列表并不完整，完整的列 表请翻阅 [logging](http://docs.python.org/dev/library/logging.html#module-logging) [http://docs.python.org/dev/library/logging.html#module-logging] 包的官方文档。

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

如果你想深度定制日志格式，你可以继承 [Formatter](http://docs.python.org/dev/library/logging.html#logging.Formatter) [http://docs.python.org/dev/library/logging.html#logging.Formatter] 。 [Formatter](http://docs.python.org/dev/library/logging.html#logging.Formatter) [http://docs.python.org/dev/library/logging.html#logging.Formatter] 有三个需要关注的方法:

[format()](http://docs.python.org/dev/library/logging.html#logging.Formatter.format) [http://docs.python.org/dev/library/logging.html#logging.Formatter.format]:

处理实际上的格式。需要一个 [LogRecord](http://docs.python.org/dev/library/logging.html#logging.LogRecord) [http://docs.python.org/dev/library/logging.html#logging.LogRecord] 对象作为参数，并

必须返回一个格式化字符串。

[formatTime()](http://docs.python.org/dev/library/logging.html#logging.Formatter.formatTime) [http://docs.python.org/dev/library/logging.html#logging.Formatter.formatTime]:

控制 asctime 格式。如果你需要不同的时间格式，可以重载这个函数。

[formatException()](http://docs.python.org/dev/library/logging.html#logging.Formatter.formatException) [http://docs.python.org/dev/library/logging.html#logging.Formatter.formatException]

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

更多信息请见其官方文档。

## 其它的库

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

作为替代，我推荐你找出你有兴趣的日志记录器，用 [getLogger()](http://docs.python.org/dev/library/logging.html#logging.getLogger) [http://docs.python.org/dev/library/logging.html#logging.getLogger] 函数来获取日志记录器，并且遍历它们来附加处理程序:

from logging import getLogger loggers = [app.logger, getLogger('sqlalchemy'), getLogger('otherlibrary')] for logger in loggers: logger.addHandler(mail\_handler) logger.addHandler(file\_handler)

# 调试应用错误

对于生产应用，按照 [记录应用错误](#Ji_Lu_Ying_Yong_Cuo_Wu___0_3_Xin) 中的描述来配置你应用的日志记录和 通知。这个章节讲述了调试部署配置和深入一个功能强大的 Python 调试器的要点。

## 有疑问时，手动运行

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

## 调试器操作

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

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

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

如果你使用 Aptana/Eclipse 来调试，你会需要把 use\_debugger 和 user\_reloader 都设置为 False 。

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

FLASK: DEBUG: True DEBUG\_WITH\_APTANA: True

然后在你应用的入口（ main.py ），你可以写入下面的内容:

if \_\_name\_\_ == "\_\_main\_\_": # To allow aptana to receive errors, set use\_debugger=False app = create\_app(config="config.yaml") if app.debug: use\_debugger = True try: # Disable Flask's debugger if external debugger is requested use\_debugger = not(app.config.get('DEBUG\_WITH\_APTANA')) except: pass app.run(use\_debugger=use\_debugger, debug=app.debug, use\_reloader=use\_debugger, host='0.0.0.0')

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_signals_html) | [上一页](#Top_of_errorhandling_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 配置处理

0.3 新版功能.

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

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

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

## 配置基础

[config](#config___None) 实际上继承于字典，并且可以像修改字典一样修 改它:

app = Flask(\_\_name\_\_) app.config['DEBUG'] = True

给定的配置值会被推送到 [Flask](#class_flask_Flask_import_name__s) 对象中，所以你可以在那里读写它 们:

app.debug = True

你可以使用 [dict.update()](http://docs.python.org/dev/library/stdtypes.html#dict.update) [http://docs.python.org/dev/library/stdtypes.html#dict.update] 方法来一次性更新多个键:

app.config.update( DEBUG=True, SECRET\_KEY='...' )

## 内置的配置值

下列配置值是 Flask 内部使用的:

|  |  |
| --- | --- |
| DEBUG | 启用/禁用 调试模式 |
| TESTING | 启用/禁用 测试模式 |
| PROPAGATE\_EXCEPTIONS | 显式地允许或禁用异常的传播。如果没有设置 或显式地设置为 None ，当 TESTING 或 DEBUG 为真时，这个值隐式地为 true. |
| PRESERVE\_CONTEXT\_ON\_EXCEPTION | 默认情况下，如果应用工作在调试模式，请求 上下文不会在异常时出栈来允许调试器内省。 这可以通过这个键来禁用。你同样可以用这个 设定来强制启用它，即使没有调试执行，这对 调试生产应用很有用（但风险也很大） |
| SECRET\_KEY | 密钥 |
| SESSION\_COOKIE\_NAME | 会话 cookie 的名称。 |
| SESSION\_COOKIE\_DOMAIN | 会话 cookie 的域。如果不设置这个值，则 cookie 对 SERVER\_NAME 的全部子域名有效 |
| SESSION\_COOKIE\_PATH | 会话 cookie 的路径。如果不设置这个值，且 没有给 '/' 设置过，则 cookie 对 APPLICATION\_ROOT 下的所有路径有效。 |
| SESSION\_COOKIE\_HTTPONLY | 控制 cookie 是否应被设置 httponly 的标志， 默认为 True |
| SESSION\_COOKIE\_SECURE | 控制 cookie 是否应被设置安全标志，默认 为 False |
| PERMANENT\_SESSION\_LIFETIME | 以 [datetime.timedelta](http://docs.python.org/dev/library/datetime.html#datetime.timedelta) |
| SESSION\_REFRESH\_EACH\_REQUEST | 这个标志控制永久会话如何刷新。如果被设置为 True （这是默认值），每一个请求 cookie 都会被刷新。如果设置为 False ，只有当 cookie 被修改后才会发送一个 set-cookie 的标头。非永久会话不会受到这个配置项的影响 。 |
| USE\_X\_SENDFILE | 启用/禁用 x-sendfile |
| LOGGER\_NAME | 日志记录器的名称 |
| SERVER\_NAME | 服务器名和端口。需要这个选项来支持子域名 （例如： 'myapp.dev:5000' ）。注意 localhost 不支持子域名，所以把这个选项设 置为 “localhost” 没有意义。设置 SERVER\_NAME 默认会允许在没有请求上下文 而仅有应用上下文时生成 URL |
| APPLICATION\_ROOT | 如果应用不占用完整的域名或子域名，这个选项可 以被设置为应用所在的路径。这个路径也会用于会 话 cookie 的路径值。如果直接使用域名，则留作 None |
| MAX\_CONTENT\_LENGTH | 如果设置为字节数， Flask 会拒绝内容长度大于 此值的请求进入，并返回一个 413 状态码 |
| SEND\_FILE\_MAX\_AGE\_DEFAULT: | 默认缓存控制的最大期限，以秒计，在 [flask.Flask.send\_static\_file()](#send_static_file_filename) (默认的 静态文件处理器)中使用。对于单个文件分别在 [Flask](#class_flask_Flask_import_name__s) 或 [Blueprint](#class_flask_Blueprint_name__impo) 上使用 [get\_send\_file\_max\_age()](#get_send_file_max_age_filename) 来覆盖这个值。默认为 43200（12小时）。 |
| TRAP\_HTTP\_EXCEPTIONS | 如果这个值被设置为 True ，Flask不会执行 HTTP 异常的错误处理，而是像对待其它异常一样， 通过异常栈让它冒泡地抛出。这对于需要找出 HTTP 异常源头的可怕调试情形是有用的。 |
| TRAP\_BAD\_REQUEST\_ERRORS | Werkzeug 处理请求中的特定数据的内部数据结构会 抛出同样也是“错误的请求”异常的特殊的 key errors 。同样地，为了保持一致，许多操作可以 显式地抛出 BadRequest 异常。因为在调试中，你 希望准确地找出异常的原因，这个设置用于在这些 情形下调试。如果这个值被设置为 True ，你 只会得到常规的回溯。 |
| PREFERRED\_URL\_SCHEME | 生成URL的时候如果没有可用的 URL 模式话将使 用这个值。默认为 http |
| JSON\_AS\_ASCII | 默认情况下 Flask 使用 ascii 编码来序列化对 象。如果这个值被设置为 False ， Flask不会 将其编码为 ASCII，并且按原样输出，返回它的 unicode 字符串。比如 jsonfiy 会自动地采用 utf-8 来编码它然后才进行传输。 |
| JSON\_SORT\_KEYS | 默认情况下 Flask 按照 JSON 对象的键的顺序来序 来序列化它。这样做是为了确保键的顺序不会受到 字典的哈希种子的影响，从而返回的值每次都是 一致的，不会造成无用的额外 HTTP 缓存。你可 以通过修改这个配置的值来覆盖默认的操作。但 这是不被推荐的做法因为这个默认的行为可能会给 你在性能的代价上带来改善。 |
| JSONIFY\_PRETTYPRINT\_REGULAR | 如果这个配置项被 True （默认值）， 如果不是 XMLHttpRequest 请求的话（由 X-Requested-With 标头控制） json 字符串的返回值会被漂亮地打印出来。 |

关于 SERVER\_NAME 的更多

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

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

0.4 新版功能: LOGGER\_NAME

0.5 新版功能: SERVER\_NAME

0.6 新版功能: MAX\_CONTENT\_LENGTH

0.7 新版功能: PROPAGATE\_EXCEPTIONS, PRESERVE\_CONTEXT\_ON\_EXCEPTION

0.8 新版功能: TRAP\_BAD\_REQUEST\_ERRORS, TRAP\_HTTP\_EXCEPTIONS, APPLICATION\_ROOT, SESSION\_COOKIE\_DOMAIN, SESSION\_COOKIE\_PATH, SESSION\_COOKIE\_HTTPONLY, SESSION\_COOKIE\_SECURE

0.9 新版功能: PREFERRED\_URL\_SCHEME

0.10 新版功能: JSON\_AS\_ASCII, JSON\_SORT\_KEYS, JSONIFY\_PRETTYPRINT\_REGULAR

1.0 新版功能: SESSION\_REFRESH\_EACH\_REQUEST

## 从文件配置

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

则一个常见模式为如下:

app = Flask(\_\_name\_\_) app.config.from\_object('yourapplication.default\_settings') app.config.from\_envvar('YOURAPPLICATION\_SETTINGS')

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

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

在 Windows 下则使用其内置的 set 命令:

>set YOURAPPLICATION\_SETTINGS=\path\to\settings.cfg

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

这里是一个配置文件的例子:

# Example configuration DEBUG = False SECRET\_KEY = '?\xbf,\xb4\x8d\xa3"<\x9c\xb0@\x0f5\xab,w\xee\x8d$0\x13\x8b83'

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

## 配置的最佳实践

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

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

## 开发 / 生产

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

app = Flask(\_\_name\_\_) app.config.from\_object('yourapplication.default\_settings') app.config.from\_envvar('YOURAPPLICATION\_SETTINGS')

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

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

一个有意思的模式是在配置中使用类和继承:

class Config(object): DEBUG = False TESTING = False DATABASE\_URI = 'sqlite://:memory:' class ProductionConfig(Config): DATABASE\_URI = 'mysql://user@localhost/foo' class DevelopmentConfig(Config): DEBUG = True class TestingConfig(Config): TESTING = True

启用这样的配置你需要调用 [from\_object()](#from_object_obj)

app.config.from\_object('configmodule.ProductionConfig')

管理配置文件有许多方式，这取决于你。这里仍然给出一个好建议的列表:

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

## 实例文件夹

0.8 新版功能.

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

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

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

app = Flask(\_\_name\_\_, instance\_path='/path/to/instance/folder')

请注意给出的 一定 是绝对路径。

如果 instance\_path 参数没有赋值，会使用下面默认的位置:

* 未安装的模块:

/myapp.py /instance

* 未安装的包:

/myapp /\_\_init\_\_.py /instance

* 已安装的包或模块:

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

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

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

app = Flask(\_\_name\_\_, instance\_relative\_config=True)

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

app = Flask(\_\_name\_\_, instance\_relative\_config=True) app.config.from\_object('yourapplication.default\_settings') app.config.from\_pyfile('application.cfg', silent=True)

实例文件夹的路径可以在 Flask.instance\_path 找到。 Flask 也提供了 一个打开实例文件夹中文件的捷径，就是 Flask.open\_instance\_resource() 。

两者的使用示例:

filename = os.path.join(app.instance\_path, 'application.cfg') with open(filename) as f: config = f.read() # or via open\_instance\_resource: with app.open\_instance\_resource('application.cfg') as f: config = f.read()

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_views_html_1) | [上一页](#Top_of_config_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 信号

0.6 新版功能.

从 Flask 0.6 开始， Flask 集成了信号支持。这个支持由 [blinker](http://pypi.python.org/pypi/blinker) [http://pypi.python.org/pypi/blinker] 库提供， 并且当它不可用时会优雅地退回。

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

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

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

## 订阅信号

你可以使用信号的 [connect()](http://discorporate.us/projects/Blinker/docs/1.1/api.html#blinker.base.Signal.connect) [http://discorporate.us/projects/Blinker/docs/1.1/api.html#blinker.base.Signal.connect] 方法来订阅信号。该 函数的第一个参数是信号发出时要调用的函数，第二个参数是可选的，用于确定信号 的发送端。退订一个信号，可以使用 [disconnect()](http://discorporate.us/projects/Blinker/docs/1.1/api.html#blinker.base.Signal.disconnect) [http://discorporate.us/projects/Blinker/docs/1.1/api.html#blinker.base.Signal.disconnect] 方法。

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

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

from flask import template\_rendered from contextlib import contextmanager @contextmanager def captured\_templates(app): recorded = [] def record(sender, template, context, \*\*extra): recorded.append((template, context)) template\_rendered.connect(record, app) try: yield recorded finally: template\_rendered.disconnect(record, app)

这可以很容易地与一个测试客户端配对:

with captured\_templates(app) as templates: rv = app.test\_client().get('/') assert rv.status\_code == 200 assert len(templates) == 1 template, context = templates[0] assert template.name == 'index.html' assert len(context['items']) == 10

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

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

此外，也有一个方便的助手方法（ [connected\_to()](http://discorporate.us/projects/Blinker/docs/1.1/api.html#blinker.base.Signal.connected_to) [http://discorporate.us/projects/Blinker/docs/1.1/api.html#blinker.base.Signal.connected\_to] ） ，它允许你临时地把函数订阅到信号并使用信号自己的上下文管理器。因为这个上下文 管理器的返回值不能由我们决定，所以必须把列表作为参数传入:

from flask import template\_rendered def captured\_templates(app, recorded, \*\*extra): def record(sender, template, context): recorded.append((template, context)) return template\_rendered.connected\_to(record, app)

上面的例子会看起来是这样:

templates = [] with captured\_templates(app, templates, \*\*extra): ... template, context = templates[0]

Blinker API 变更

[connected\_to()](http://discorporate.us/projects/Blinker/docs/1.1/api.html#blinker.base.Signal.connected_to) [http://discorporate.us/projects/Blinker/docs/1.1/api.html#blinker.base.Signal.connected\_to] 方法出现于 Blinker 1.1 。

## 创建信号

如果你想要在自己的应用中使用信号，你可以直接使用 blinker 库。最常见的用法 是在自定义的 [Namespace](http://discorporate.us/projects/Blinker/docs/1.1/api.html#blinker.base.Namespace) [http://discorporate.us/projects/Blinker/docs/1.1/api.html#blinker.base.Namespace] 中命名信号。这也是大多数时候 推荐的做法:

from blinker import Namespace my\_signals = Namespace()

现在你可以这样创建新的信号:

model\_saved = my\_signals.signal('model-saved')

这里使用唯一的信号名，简化调试。可以用 [name](http://discorporate.us/projects/Blinker/docs/1.1/api.html#blinker.base.NamedSignal.name) [http://discorporate.us/projects/Blinker/docs/1.1/api.html#blinker.base.NamedSignal.name] 属性来访问信号名。

给扩展开发者

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

## 发送信号

如果你想要发出信号，调用 [send()](http://discorporate.us/projects/Blinker/docs/1.1/api.html#blinker.base.Signal.send) [http://discorporate.us/projects/Blinker/docs/1.1/api.html#blinker.base.Signal.send] 方法可以做到。 它接受发送端作为第一个参数，和一些推送到信号订阅者的可选关键字参数:

class Model(object): ... def save(self): model\_saved.send(self)

永远尝试选择一个合适的发送端。如果你有一个发出信号的类，把 self 作为发送 端。如果你从一个随机的函数发出信号，把 current\_app.\_get\_current\_object() 作为发送端。

传递代理作为发送端

永远不要向信号传递 [current\_app](#flask_current_app) 作为发送端，使用 current\_app.\_get\_current\_object() 作为替代。这样的原因是， [current\_app](#flask_current_app) 是一个代理，而不是真正的应用对象。

## 信号与 Flask 的请求上下文

信号在接收时，完全支持 [请求上下文](#Qing_Qiu_Shang_Xia_Wen__Zhe_Bu_F) 。上下文本地的变量在 [request\_started](#flask_request_started) 和 [request\_finished](#flask_request_finished) 一贯可用， 所以你可以信任 [flask.g](#flask_g) 和其它需要的东西。注意 [发送信号](#Fa_Song_Xin_Hao__Ru_Guo_Ni_Xiang) 和 [request\_tearing\_down](#flask_request_tearing_down) 信号中描述的限制。

## 基于装饰器的信号订阅

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

from flask import template\_rendered @template\_rendered.connect\_via(app) def when\_template\_rendered(sender, template, context, \*\*extra): print 'Template %s is rendered with %s' % (template.name, context)

## 核心信号

下列是 Flask 中存在的信号:

flask.template\_rendered

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

订阅示例:

def log\_template\_renders(sender, template, context, \*\*extra): sender.logger.debug('Rendering template "%s" with context %s', template.name or 'string template', context) from flask import template\_rendered template\_rendered.connect(log\_template\_renders, app)

flask.request\_started

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

订阅示例:

def log\_request(sender, \*\*extra): sender.logger.debug('Request context is set up') from flask import request\_started request\_started.connect(log\_request, app)

flask.request\_finished

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

订阅示例:

def log\_response(sender, response, \*\*extra): sender.logger.debug('Request context is about to close down. ' 'Response: %s', response) from flask import request\_finished request\_finished.connect(log\_response, app)

flask.got\_request\_exception

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

订阅示例:

def log\_exception(sender, exception, \*\*extra): sender.logger.debug('Got exception during processing: %s', exception) from flask import got\_request\_exception got\_request\_exception.connect(log\_exception, app)

flask.request\_tearing\_down

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

订阅示例:

def close\_db\_connection(sender, \*\*extra): session.close() from flask import request\_tearing\_down request\_tearing\_down.connect(close\_db\_connection, app)

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

flask.appcontext\_tearing\_down

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

订阅示例:

def close\_db\_connection(sender, \*\*extra): session.close() from flask import request\_tearing\_down appcontext\_tearing\_down.connect(close\_db\_connection, app)

如果有异常它会被传递一个 exc 关键字参数引用导致销毁的异常。

flask.appcontext\_pushed

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

用法示例:

from contextlib import contextmanager from flask import appcontext\_pushed @contextmanager def user\_set(app, user): def handler(sender, \*\*kwargs): g.user = user with appcontext\_pushed.connected\_to(handler, app): yield

测试代码:

def test\_user\_me(self): with user\_set(app, 'john'): c = app.test\_client() resp = c.get('/users/me') assert resp.data == 'username=john'

0.10 新版功能.

flask.appcontext\_popped

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

0.10 新版功能.

flask.message\_flashed

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

订阅示例:

recorded = [] def record(sender, message, category, \*\*extra): recorded.append((message, category)) from flask import message\_flashed message\_flashed.connect(record, app)

0.10 新版功能.

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_appcontext_html) | [上一页](#Top_of_signals_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 即插视图

0.7 新版功能.

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

## 基本原则

想象你有一个从数据库载入一个对象列表并渲染到视图的函数:

@app.route('/users/') def show\_users(page): users = User.query.all() return render\_template('users.html', users=users)

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

from flask.views import View class ShowUsers(View): def dispatch\_request(self): users = User.query.all() return render\_template('users.html', objects=users) app.add\_url\_rule('/users/', ShowUsers.as\_view('show\_users'))

如你所见，你需要做的是创建一个 [flask.views.View](#class_flask_views_View) 的子类， 并且实现 [dispatch\_request()](#dispatch_request_1) 。然后我们需要用类方法 [as\_view()](#classmethod_as_view_name___class) 把这个类转换到一个实际的视图函数。你传给 这个函数的字符串是视图之后的最终名称。但是用它自己实现的方法不够有效，所以 我们稍微重构一下代码:

from flask.views import View class ListView(View): def get\_template\_name(self): raise NotImplementedError() def render\_template(self, context): return render\_template(self.get\_template\_name(), \*\*context) def dispatch\_request(self): context = {'objects': self.get\_objects()} return self.render\_template(context) class UserView(ListView): def get\_template\_name(self): return 'users.html' def get\_objects(self): return User.query.all()

这当然不是那么有助于一个小例子，但是对于解释基本原则已经很有用了。当你有一 个基于类的视图，那么问题来了， self 指向什么。它工作的方式是，无论何时请 求被调度，会创建这个类的一个新实例，并且 [dispatch\_request()](#dispatch_request_1) 方法会以 URL 规则为参数调用。 这个类本身会用传递到 [as\_view()](#classmethod_as_view_name___class) 函数的参数来实例化。 比如，你可以像这样写一个类:

class RenderTemplateView(View): def \_\_init\_\_(self, template\_name): self.template\_name = template\_name def dispatch\_request(self): return render\_template(self.template\_name)

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

app.add\_url\_rule('/about', view\_func=RenderTemplateView.as\_view( 'about\_page', template\_name='about.html'))

## 方法提示

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

class MyView(View): methods = ['GET', 'POST'] def dispatch\_request(self): if request.method == 'POST': ... ... app.add\_url\_rule('/myview', view\_func=MyView.as\_view('myview'))

## 基于调度的方法

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

from flask.views import MethodView class UserAPI(MethodView): def get(self): users = User.query.all() ... def post(self): user = User.from\_form\_data(request.form) ... app.add\_url\_rule('/users/', view\_func=UserAPI.as\_view('users'))

如此，你可以不提供 [methods](#methods___None) 属性。它会自动的按照 类中定义的方法来设置。

## 装饰视图

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

def user\_required(f): """Checks whether user is logged in or raises error 401.""" def decorator(\*args, \*\*kwargs): if not g.user: abort(401) return f(\*args, \*\*kwargs) return decorator view = user\_required(UserAPI.as\_view('users')) app.add\_url\_rule('/users/', view\_func=view)

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

class UserAPI(MethodView): decorators = [user\_required]

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

## 用于 API 的方法视图

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

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

那么，你会想用 [MethodView](#class_flask_views_MethodView) 做什么？诀窍是利用你可以 对相同的视图提供多个规则的事实。

让我们假设这时视图看起来是这个样子:

class UserAPI(MethodView): def get(self, user\_id): if user\_id is None: # return a list of users pass else: # expose a single user pass def post(self): # create a new user pass def delete(self, user\_id): # delete a single user pass def put(self, user\_id): # update a single user pass

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

user\_view = UserAPI.as\_view('user\_api') app.add\_url\_rule('/users/', defaults={'user\_id': None}, view\_func=user\_view, methods=['GET',]) app.add\_url\_rule('/users/', view\_func=user\_view, methods=['POST',]) app.add\_url\_rule('/users/<int:user\_id>', view\_func=user\_view, methods=['GET', 'PUT', 'DELETE'])

如果你有许多看起来类似的 API ，你可以重构上述的注册代码:

def register\_api(view, endpoint, url, pk='id', pk\_type='int'): view\_func = view.as\_view(endpoint) app.add\_url\_rule(url, defaults={pk: None}, view\_func=view\_func, methods=['GET',]) app.add\_url\_rule(url, view\_func=view\_func, methods=['POST',]) app.add\_url\_rule('%s<%s:%s>' % (url, pk\_type, pk), view\_func=view\_func, methods=['GET', 'PUT', 'DELETE']) register\_api(UserAPI, 'user\_api', '/users/', pk='user\_id')

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_reqcontext_html) | [上一页](#Top_of_views_html_1) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 应用上下文

0.9 新版功能.

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

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

相反，到了第二个状态，在处理请求时，有一些其它的规则:

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

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

[current\_app](#flask_current_app) 上下文本地变量就是应用上下文驱动的。

## 应用上下文的作用

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

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

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

## 创建应用上下文

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

第二种是显式地调用 [app\_context()](#app_context) 方法:

from flask import Flask, current\_app app = Flask(\_\_name\_\_) with app.app\_context(): # within this block, current\_app points to app. print current\_app.name

在配置了 SERVER\_NAME 时，应用上下文也被用于 [url\_for()](#flask_url_for_endpoint____values) 函 数。这允许你在没有请求时生成 URL 。

## 应用上下文局部变量

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

更多详情见 [Flask 扩展开发](#extension_dev) 。

## 上下文用法

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

最常见的应用就是把资源的管理分成如下两个部分：

1. 一个缓存在上下文中的隐式资源
2. 当上下文被销毁时重新分配基础资源

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

如下是我们刚刚提到的连接数据库的例子:

import sqlite3 from flask import g def get\_db(): db = getattr(g, '\_database', None) if db is None: db = g.\_database = connect\_to\_database() return db @app.teardown\_appcontext def teardown\_db(exception): db = getattr(g, '\_database', None) if db is not None: db.close()

当 get\_db() 这个函数第一次被调用的时候数据库连接已经被建立了。 为了使得看起来更隐式一点我们可以使用 [LocalProxy](http://werkzeug.pocoo.org/docs/local/#werkzeug.local.LocalProxy) [http://werkzeug.pocoo.org/docs/local/#werkzeug.local.LocalProxy] 这 个类：

from werkzeug.local import LocalProxy db = LocalProxy(get\_db)

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

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_blueprints_html) | [上一页](#Top_of_appcontext_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 请求上下文

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

这里推荐先阅读 [应用上下文](#Ying_Yong_Shang_Xia_Wen___0_9_Xi) 章节。

## 深入上下文作用域

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

from flask import request, url\_for def redirect\_url(): return request.args.get('next') or \ request.referrer or \ url\_for('index')

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

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

这有很大意义，因为我们当前并没有可以访问的请求。所以我们需要制造一个 请求并且绑定到当前的上下文。 [test\_request\_context](#test_request_context__args____kw) 方 法为我们创建一个 [RequestContext](#class_flask_ctx_RequestContext_a):

>>> ctx = app.test\_request\_context('/?next=http://example.com/')

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

>>> ctx.push()

从这点开始，你可以使用请求对象:

>>> redirect\_url() u'http://example.com/'

直到你调用 pop:

>>> ctx.pop()

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

更多如何从交互式 Python shell 中利用请求上下文的信息，请见 [与 Shell 共舞](#Yu__Shell_Gong_Wu___0_3_Xin_Ban) 章节。

## 上下文如何工作

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

def wsgi\_app(self, environ): with self.request\_context(environ): try: response = self.full\_dispatch\_request() except Exception, e: response = self.make\_response(self.handle\_exception(e)) return response(environ, start\_response)

request\_context() 方法返回一个新的 [RequestContext](#class_flask_ctx_RequestContext_a) 对象，并结合 with 声明来绑定上下文。 从相同线程中被调用的一切，直到 with 声明结束前，都可以访问全局的请求 变量（ [flask.request](#class_flask_request) 和其它）。

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

另一件需要注意的事是，请求上下文被压入栈时，并且没有当前应用的应用上下文， 它会自动创建一个 [应用上下文](#Ying_Yong_Shang_Xia_Wen___0_9_Xi) 。

## 回调和错误

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

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

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

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

新的绑定于销毁请求的函数被认为是用于代替那些请求的最后绝对需要发生的事。

## 销毁回调

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

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

with app.test\_client() as client: resp = client.get('/foo') # the teardown functions are still not called at that point # even though the response ended and you have the response # object in your hand # only when the code reaches this point the teardown functions # are called. Alternatively the same thing happens if another # request was triggered from the test client

从这些命令行操作中，很容易看出它的行为:

>>> app = Flask(\_\_name\_\_) >>> @app.teardown\_request ... def teardown\_request(exception=None): ... print 'this runs after request' ... >>> ctx = app.test\_request\_context() >>> ctx.push() >>> ctx.pop() this runs after request >>>

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

## 留意代理

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

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

* 代理对象不会伪造它们继承的类型，所以如果你想运行真正的实例检查，你需要 在被代理的实例上这么做（见下面的 \_get\_current\_object ）。
* 如果对象引用是重要的（例如发送 [信号](#Xin_Hao___0_6_Xin_Ban_Gong_Neng) ）

如果你需要访问潜在的被代理的对象，你可以使用 [\_get\_current\_object()](http://werkzeug.pocoo.org/docs/local/#werkzeug.local.LocalProxy._get_current_object) [http://werkzeug.pocoo.org/docs/local/#werkzeug.local.LocalProxy.\_get\_current\_object] 方法:

app = current\_app.\_get\_current\_object() my\_signal.send(app)

## 错误时的上下文保护

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

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

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

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_extensions_html) | [上一页](#Top_of_reqcontext_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 用蓝图实现模块化的应用

0.7 新版功能.

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

## 为什么使用蓝图？

Flask 中的蓝图为这些情况设计:

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

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

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

## 蓝图的设想

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

## 我的第一个蓝图

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

from flask import Blueprint, render\_template, abort from jinja2 import TemplateNotFound simple\_page = Blueprint('simple\_page', \_\_name\_\_, template\_folder='templates') @simple\_page.route('/', defaults={'page': 'index'}) @simple\_page.route('/<page>') def show(page): try: return render\_template('pages/%s.html' % page) except TemplateNotFound: abort(404)

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

## 注册蓝图

那么你如何注册蓝图？像这样:

from flask import Flask from yourapplication.simple\_page import simple\_page app = Flask(\_\_name\_\_) app.register\_blueprint(simple\_page)

如果你检查已经注册到应用的规则，你会发现这些:

[<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>, <Rule '/<page>' (HEAD, OPTIONS, GET) -> simple\_page.show>, <Rule '/' (HEAD, OPTIONS, GET) -> simple\_page.show>]

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

不过，蓝图也可以在不同的位置挂载:

app.register\_blueprint(simple\_page, url\_prefix='/pages')

那么，这些果然是生成出的规则:

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

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

## 蓝图资源

蓝图也可以提供资源。有时候你会只为它提供的资源而引入一个蓝图。

### 蓝图资源文件夹

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

这个文件夹会从 Blueprint 的第二个参数中推断出来，通常是 \_\_name\_\_ 。 这个参数决定对应蓝图的是哪个逻辑的 Python 模块或包。如果它指向一个存在的 Python 包，这个包（通常是文件系统中的文件夹）就是资源文件夹。如果是一个模块， 模块所在的包就是资源文件夹。你可以访问 Blueprint.root\_path 属性来查看 资源文件夹是什么:

>>> simple\_page.root\_path '/Users/username/TestProject/yourapplication'

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

with simple\_page.open\_resource('static/style.css') as f: code = f.read()

### 静态文件

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

admin = Blueprint('admin', \_\_name\_\_, static\_folder='static')

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

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

url\_for('admin.static', filename='style.css')

### 模板

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

admin = Blueprint('admin', \_\_name\_\_, template\_folder='templates')

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

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

## 构造 URL

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

url\_for('admin.index')

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

url\_for('.index')

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

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_shell_html) | [上一页](#Top_of_blueprints_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# Flask 扩展

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

## 寻找扩展

[Flask Extension Registry](http://flask.pocoo.org/extensions/) [http://flask.pocoo.org/extensions/] 中列出了 Flask 扩展，并且可以通过 easy\_install 或 pip 下载。如果你把一个 Flask 扩展添加到 requirements.rst 或 setup.py 文件的依赖关系中，它们通常可以用一个 简单的命令或是在你应用安装时被安装。

## 使用扩展

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

from flask.ext import foo

## Flask 0.8 以前

如果你在使用 Flask 0.7 或更早的版本，包 flask.ext 并不存在，你不得不 从 flaskext.foo 或 flask\_foo 中导入，这取决与应用是如何分发的。如果你 想要开发支持 Flask 0.7 或更早版本的应用，你仍然应该从 flask.ext 中导 入。我们提供了一个兼容性模块来在 Flask 的老版本中提供这个包。你可以从 github 上下载它：[flaskext\_compat.py](https://github.com/mitsuhiko/flask/raw/master/scripts/flaskext_compat.py) [https://github.com/mitsuhiko/flask/raw/master/scripts/flaskext\_compat.py]

这里是使用它的方法:

import flaskext\_compat flaskext\_compat.activate() from flask.ext import foo

一旦激活了 flaskext\_compat 模块，就会存在 flask.ext ，并且你可以从 那里开始导入。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_index_html_2) | [上一页](#Top_of_extensions_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 与 Shell 共舞

0.3 新版功能.

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

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

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

一般来说，在阅读本章节之前还是建议大家先阅读 [请求上下文](#Qing_Qiu_Shang_Xia_Wen__Zhe_Bu_F) 相关章节。

## 创建一个请求上下文

从 Shell 创建一个合适的上下文，最简单的方法是使用 [test\_request\_context](#test_request_context__args____kw) 方法，此方法 会创建一个 [RequestContext](#class_flask_ctx_RequestContext_a) 对象:

>>> ctx = app.test\_request\_context()

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

>>> ctx.push()

从这里往后，您就可以使用这个请求对象直到您调用 pop 方法为止:

>>> ctx.pop()

## 激发请求发送前后的调用

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

您可以很容易的自己完成这件事，仅仅手动调用 [preprocess\_request()](#preprocess_request) 函数即可:

>>> ctx = app.test\_request\_context() >>> ctx.push() >>> app.preprocess\_request()

请注意， [preprocess\_request()](#preprocess_request) 函数可能会返回 一个响应对象。这时，忽略它就好了。

要关闭一个请求，您需要在请求后的调用函数(由 [process\_response()](#process_response_response) 函数激发)运行之前耍一些小小的把戏:

>>> app.process\_response(app.response\_class()) <Response 0 bytes [200 OK]> >>> ctx.pop()

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

## 进一步提升 Shell 使用体验

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

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

>>> from shelltools import \*

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_packages_html) | [上一页](#Top_of_shell_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# Flask 代码模式

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

这章提供了一些由用户贡献的代码片段和模板来加速开发 [Flask Snippet Archives](http://flask.pocoo.org/snippets/) [http://flask.pocoo.org/snippets/].

* [大型应用](#Top_of_packages_html)
  + [简单的包](#Jian_Dan_De_Bao__Jiang_Yi_Ge_Xia)
  + [与蓝图一起工作](#Yu_Lan_Tu_Yi_Qi_Gong_Zuo__Ru_Guo)
* [应用程序的工厂函数](#Top_of_appfactories_html)
  + [基础的工厂函数](#Ji_Chu_De_Gong_Chang_Han_Shu__Ni)
  + [使用应用程序](#Shi_Yong_Ying_Yong_Cheng_Xu__Suo)
  + [工厂函数的改进](#Gong_Chang_Han_Shu_De_Gai_Jin__Q)
* [应用调度](#Top_of_appdispatch_html)
  + [如何使用此文档](#Ru_He_Shi_Yong_Ci_Wen_Dang__Xia)
  + [合并应用](#He_Bing_Ying_Yong__Ru_Guo_Nin_Yo)
  + [通过子域名调度](#Tong_Guo_Zi_Yu_Ming_Diao_Du__You)
  + [使用路径来调度](#Shi_Yong_Lu_Jing_Lai_Diao_Du__To)
* [使用 URL 处理器](#Top_of_urlprocessors_html)
  + [国际化的应用程序 URL](#Guo_Ji_Hua_De_Ying_Yong_Cheng_Xu)
  + [多国语言化的 Blueprint URL](#Duo_Guo_Yu_Yan_Hua_De__Blueprint)
* [部署和分发](#Top_of_distribute_html)
  + [基础的安装脚本](#Ji_Chu_De_An_Zhuang_Jiao_Ben__Yi)
  + [分发代码](#Fen_Fa_Dai_Ma__Ru_Guo_Nin_Shi_Tu)
  + [声明依赖关系](#Sheng_Ming_Yi_Lai_Guan_Xi__Nin_X)
  + [安装 / 开发](#An_Zhuang____Kai_Fa__An_Zhuang_N)
* [使用 Fabric 部署](#Top_of_fabric_html)
  + [创建第一个 Fabfile](#Chuang_Jian_Di_Yi_Ge__Fabfile_Fa)
  + [运行 Fabfile](#Yun_Xing__Fabfile_Ru_He_Zhi_Xing)
  + [WSGI 文件](#WSGI_Wen_Jian__WSGI_Wen_Jian_Yin)
  + [配置文件](#Pei_Zhi_Wen_Jian__Zheng_Ru_Shang)
  + [第一次部署](#Di_Yi_Ci_Bu_Shu__Xian_Zai_Wo_Men)
  + [下一步操作](#Xia_Yi_Bu_Cao_Zuo__Cong_Xian_Zai)
* [在 Flask 中使用 SQLite 3](#Top_of_sqlite3_html)
  + [按需连接](#An_Xu_Lian_Jie__Shang_Shu_Fang_F)
  + [简化查询](#Jian_Hua_Cha_Xun__Xian_Zai_Zai_M)
  + [初始化数据库模型](#Chu_Shi_Hua_Shu_Ju_Ku_Mo_Xing__G)
* [在 Flask 中使用 SQLAlchemy](#Top_of_sqlalchemy_html)
  + [Flask-SQLAlchemy 扩展](#Flask_SQLAlchemy_Kuo_Zhan__Yin_W)
  + [显式调用](#Xian_Shi_Diao_Yong__SQLAlchemy_Z)
  + [手动实现 ORM](#Shou_Dong_Shi_Xian__ORM_Shou_Don)
  + [SQL 抽象层](#SQL_Chou_Xiang_Ceng__Ru_Guo_Nin)
* [上传文件](#Top_of_fileuploads_html)
  + [一点点介绍](#Yi_Dian_Dian_Jie_Shao__Rang_Wo_M)
  + [改进上传功能](#Gai_Jin_Shang_Chuan_Gong_Neng)
  + [上传进度条](#Shang_Chuan_Jin_Du_Tiao__Yi_Qian)
  + [更简单解决方案](#Geng_Jian_Dan_Jie_Jue_Fang_An__Y)
* [缓存](#Top_of_caching_html)
  + [配置缓存](#Pei_Zhi_Huan_Cun__Lei_Si_Yu_Jian)
  + [使用缓存](#Shi_Yong_Huan_Cun__You_Liang_Ge)
* [视图装饰器](#Top_of_viewdecorators_html)
  + [过滤未登录用户的装饰器](#Guo_Lu_Wei_Deng_Lu_Yong_Hu_De_Zh)
  + [缓存装饰器](#Huan_Cun_Zhuang_Shi_Qi__Shi_Xian)
  + [模板装饰器](#Mo_Ban_Zhuang_Shi_Qi__TurboGears)
  + [终端装饰器](#Zhong_Duan_Zhuang_Shi_Qi__Ru_Guo)
* [使用 WTForms 进行表单验证](#Top_of_wtforms_html)
  + [表单](#Biao_Dan__Yi_Xia_Shi_Yi_Ge_Dian)
  + [在视图里](#Zai_Shi_Tu_Li__Zai_Shi_Tu_Han_Sh)
  + [在模板中使用表单](#Zai_Mo_Ban_Zhong_Shi_Yong_Biao_D)
* [模板继承](#Top_of_templateinheritance_html)
  + [基础模板](#Ji_Chu_Mo_Ban__Zai_Zhe_Ge_Jiao_Z)
  + [子模板](#Zi_Mo_Ban__Zi_Mo_Ban_Kan_Qi_Lai)
* [消息闪现](#Top_of_flashing_html)
  + [简单的闪现](#Jian_Dan_De_Shan_Xian__Zhe_Li_Sh)
  + [分类闪现](#Fen_Lei_Shan_Xian___0_3_Xin_Ban)
  + [过滤闪现消息](#Guo_Lu_Shan_Xian_Xiao_Xi___0_9_X)
* [用 jQuery 实现 Ajax](#Top_of_jquery_html)
  + [加载 jQuery](#Jia_Zai__jQuery_Wei_Le_Shi_Yong)
  + [我的站点在哪?](#Wo_De_Zhan_Dian_Zai_Na___Nin_Zhi)
  + [JSON 视图函数](#JSON_Shi_Tu_Han_Shu__Xian_Zai_Ra)
  + [HTML 部分](#HTML_Bu_Fen__Nin_De__index_html)
* [自定义错误页面](#Top_of_errorpages_html)
  + [通常的错误代码](#Tong_Chang_De_Cuo_Wu_Dai_Ma__Xia)
  + [错误处理器](#Cuo_Wu_Chu_Li_Qi__Yi_Ge_Cuo_Wu_C)
* [延迟加载视图](#Top_of_lazyloading_html)
  + [转换到中央 URL 映射](#Zhuan_Huan_Dao_Zhong_Yang__URL_Y)
  + [延迟加载](#Yan_Chi_Jia_Zai__Mu_Qian_Wo_Men)
* [在 Flask 中使用 MongoKit](#Top_of_mongokit_html)
  + [显式调用](#Xian_Shi_Diao_Yong__MongoKit_De)
  + [PyMongo 兼容层](#PyMongo_Jian_Rong_Ceng__Ru_Guo_N)
* [添加 Favicon](#Top_of_favicon_html)
  + [参考](#Can_Kao___Wikipedia_Shang_You_Gu)
* [数据流](#Top_of_streaming_html)
  + [基本使用](#Ji_Ben_Shi_Yong__Xia_Mian_Shi_Yi)
  + [在模板中生成流](#Zai_Mo_Ban_Zhong_Sheng_Cheng_Liu)
* [延迟请求回调](#Top_of_deferredcallbacks_html)
  + [装饰器](#Zhuang_Shi_Qi__Xia_Mian_De_Zhuan)
  + [调用延迟函数](#Diao_Yong_Yan_Chi_Han_Shu__Xian)
  + [一个实际应用的例子](#Yi_Ge_Shi_Ji_Ying_Yong_De_Li_Zi)
* [添加 HTTP Method Overrides](#Top_of_methodoverrides_html)
* [请求内容校验码](#Top_of_requestchecksum_html)
* [基于 Celery 的后台任务](#Top_of_celery_html)
  + [安装 Celery](#An_Zhuang__Celery_Celery_Ti_Jiao)
  + [配置 Celery](#Pei_Zhi__Celery_Ni_Xu_Yao_De_Di)
  + [最简示例](#Zui_Jian_Shi_Li__Tong_Guo_Shang)
  + [运行 Celery 职程](#Yun_Xing__Celery_Zhi_Cheng__Xian)

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_appfactories_html) | [上一页](#Top_of_index_html_2) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 大型应用

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

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

## 简单的包

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

您最后得到的东西应该像下面这样:

/yourapplication /yourapplication /\_\_init\_\_.py /static /style.css /templates layout.html index.html login.html ...

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

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

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

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

这里是 \_\_init\_\_.py 的一个例子:

from flask import Flask app = Flask(\_\_name\_\_) import yourapplication.views

而 views.py 应该看起来像这样:

from yourapplication import app @app.route('/') def index(): return 'Hello World!'

您最终应该得到的程序结构应该是这样:

/yourapplication /runserver.py /yourapplication /\_\_init\_\_.py /views.py /static /style.css /templates layout.html index.html login.html ...

循环导入

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

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

## 与蓝图一起工作

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

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_appdispatch_html) | [上一页](#Top_of_packages_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 应用程序的工厂函数

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

这么做的目的在于:

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

那么该如何使用他们呢？

## 基础的工厂函数

您可以像下面展示的这样，从一个函数里启动这个应用:

def create\_app(config\_filename): app = Flask(\_\_name\_\_) app.config.from\_pyfile(config\_filename) from yourapplication.views.admin import admin from yourapplication.views.frontend import frontend app.register\_blueprint(admin) app.register\_blueprint(frontend) return app

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

from flask import current\_app, Blueprint, render\_template admin = Blueprint('admin', \_\_name\_\_, url\_prefix='/admin') @admin.route('/') def index(): return render\_template(current\_app.config['INDEX\_TEMPLATE'])

在这里我们从配置中查找一个网页模板文件的名字。

## 使用应用程序

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

from yourapplication import create\_app app = create\_app('/path/to/config.cfg') app.run()

## 工厂函数的改进

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

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

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_urlprocessors_html) | [上一页](#Top_of_appfactories_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 应用调度

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

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

## 如何使用此文档

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

from werkzeug.serving import run\_simple run\_simple('localhost', 5000, application, use\_reloader=True)

注意，[run\_simple](http://werkzeug.pocoo.org/docs/serving/#werkzeug.serving.run_simple) [http://werkzeug.pocoo.org/docs/serving/#werkzeug.serving.run\_simple] 函数不是为生产 用途设计的，发布应用时可以使用 [成熟的 WSGI 服务器](#Bu_Shu_Xuan_Ze__Qu_Jue_Yu_Ni_Xia) 。

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

from flask import Flask from werkzeug.serving import run\_simple app = Flask(\_\_name\_\_) app.debug = True @app.route('/') def hello\_world(): return 'Hello World!' if \_\_name\_\_ == '\_\_main\_\_': run\_simple('localhost', 5000, app, use\_reloader=True, use\_debugger=True, use\_evalex=True)

## 合并应用

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

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

from werkzeug.wsgi import DispatcherMiddleware from frontend\_app import application as frontend from backend\_app import application as backend application = DispatcherMiddleware(frontend, { '/backend': backend })

## 通过子域名调度

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

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

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

from threading import Lock class SubdomainDispatcher(object): def \_\_init\_\_(self, domain, create\_app): self.domain = domain self.create\_app = create\_app self.lock = Lock() self.instances = {} def get\_application(self, host): host = host.split(':')[0] assert host.endswith(self.domain), 'Configuration error' subdomain = host[:-len(self.domain)].rstrip('.') with self.lock: app = self.instances.get(subdomain) if app is None: app = self.create\_app(subdomain) self.instances[subdomain] = app return app def \_\_call\_\_(self, environ, start\_response): app = self.get\_application(environ['HTTP\_HOST']) return app(environ, start\_response)

调度器可以这样使用:

from myapplication import create\_app, get\_user\_for\_subdomain from werkzeug.exceptions import NotFound def make\_app(subdomain): user = get\_user\_for\_subdomain(subdomain) if user is None: # if there is no user for that subdomain we still have # to return a WSGI application that handles that request. # We can then just return the NotFound() exception as # application which will render a default 404 page. # You might also redirect the user to the main page then return NotFound() # otherwise create the application for the specific user return create\_app(user) application = SubdomainDispatcher('example.com', make\_app)

## 使用路径来调度

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

from threading import Lock from werkzeug.wsgi import pop\_path\_info, peek\_path\_info class PathDispatcher(object): def \_\_init\_\_(self, default\_app, create\_app): self.default\_app = default\_app self.create\_app = create\_app self.lock = Lock() self.instances = {} def get\_application(self, prefix): with self.lock: app = self.instances.get(prefix) if app is None: app = self.create\_app(prefix) if app is not None: self.instances[prefix] = app return app def \_\_call\_\_(self, environ, start\_response): app = self.get\_application(peek\_path\_info(environ)) if app is not None: pop\_path\_info(environ) else: app = self.default\_app return app(environ, start\_response)

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

from myapplication import create\_app, default\_app, get\_user\_for\_prefix def make\_app(prefix): user = get\_user\_for\_prefix(prefix) if user is not None: return create\_app(user) application = PathDispatcher(default\_app, make\_app)

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_distribute_html) | [上一页](#Top_of_appdispatch_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 使用 URL 处理器

0.7 新版功能.

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

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

## 国际化的应用程序 URL

试想如下一个网页应用:

from flask import Flask, g app = Flask(\_\_name\_\_) @app.route('/<lang\_code>/') def index(lang\_code): g.lang\_code = lang\_code ... @app.route('/<lang\_code>/about') def about(lang\_code): g.lang\_code = lang\_code ...

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

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

@app.url\_defaults def add\_language\_code(endpoint, values): if 'lang\_code' in values or not g.lang\_code: return if app.url\_map.is\_endpoint\_expecting(endpoint, 'lang\_code'): values['lang\_code'] = g.lang\_code

URL 映射的函数 [is\_endpoint\_expecting()](http://werkzeug.pocoo.org/docs/routing/#werkzeug.routing.Map.is_endpoint_expecting) [http://werkzeug.pocoo.org/docs/routing/#werkzeug.routing.Map.is\_endpoint\_expecting] 可以被用来 识别是否可以给末端的函数提供一个多国语言代号码。

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

@app.url\_value\_preprocessor def pull\_lang\_code(endpoint, values): g.lang\_code = values.pop('lang\_code', None)

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

from flask import Flask, g app = Flask(\_\_name\_\_) @app.url\_defaults def add\_language\_code(endpoint, values): if 'lang\_code' in values or not g.lang\_code: return if app.url\_map.is\_endpoint\_expecting(endpoint, 'lang\_code'): values['lang\_code'] = g.lang\_code @app.url\_value\_preprocessor def pull\_lang\_code(endpoint, values): g.lang\_code = values.pop('lang\_code', None) @app.route('/<lang\_code>/') def index(): ... @app.route('/<lang\_code>/about') def about(): ...

## 多国语言化的 Blueprint URL

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

from flask import Blueprint, g bp = Blueprint('frontend', \_\_name\_\_, url\_prefix='/<lang\_code>') @bp.url\_defaults def add\_language\_code(endpoint, values): values.setdefault('lang\_code', g.lang\_code) @bp.url\_value\_preprocessor def pull\_lang\_code(endpoint, values): g.lang\_code = values.pop('lang\_code') @bp.route('/') def index(): ... @bp.route('/about') def about(): ...

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_fabric_html) | [上一页](#Top_of_urlprocessors_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 部署和分发

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

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

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

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

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

## 基础的安装脚本

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

考虑这些操作可能会有风险，因此建议您参考 [你最好使用 virtualenv](#virtualenv_Ni_Zai_Kai_Fa_Zhong_H) 一文。

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

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

一个基本的 Flask 应用的 setup.py 文件看起来像如下这样:

from setuptools import setup setup( name='Your Application', version='1.0', long\_description=\_\_doc\_\_, packages=['yourapplication'], include\_package\_data=True, zip\_safe=False, install\_requires=['Flask'] )

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

from setuptools import setup, find\_packages setup( ... packages=find\_packages() )

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

## 分发代码

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

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

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

## 声明依赖关系

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

install\_requires=[ 'Flask>=0.2', 'SQLAlchemy>=0.6', 'BrokenPackage>=0.7,<=1.0' ]

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

dependency\_links=['http://example.com/yourfiles']

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

## 安装 / 开发

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

$ python setup.py install

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

$ python setup.py develop

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

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_sqlite3_html) | [上一页](#Top_of_distribute_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 使用 Fabric 部署

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

开始之前，请先检查如下列表中的事项是否都已经满足了:

* 在本地已经安装了 Fabric 1.0 。即这个教程完成时， Fabric 的最新版本。
* 应用程序已经被封装为包的形式，而且有一个有效的 setup.py 文件 (参考 [部署和分发](#Bu_Shu_He_Fen_Fa__distribute__ht))。
* 在下文中的例子里，我们使用 mod\_wsgi 作为远程服务器使用的服务端程序。 您当然也可以使用您喜欢的服务端程序，但是考虑到 Apache 和 mod\_wsgi 的 组合非常简单易用且容易安装配置，并且在无 root 权限的情况下，存在一个比较 简单的方法来重启服务器。

## 创建第一个 Fabfile

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

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

from fabric.api import \* # 远程服务器登陆使用的用户名 env.user = 'appuser' # 需要进行操作的服务器地址 env.hosts = ['server1.example.com', 'server2.example.com'] def pack(): # 以 tar 归档的方式创建一个新的代码分发 local('python setup.py sdist --formats=gztar', capture=False) def deploy(): # 之处发布产品的名称和版本 dist = local('python setup.py --fullname', capture=True).strip() # 将代码归档上传到服务器当中的临时文件夹内 put('dist/%s.tar.gz' % dist, '/tmp/yourapplication.tar.gz') # 创建一个文件夹，进入这个文件夹，然后将我们的归档解压到那里 run('mkdir /tmp/yourapplication') with cd('/tmp/yourapplication'): run('tar xzf /tmp/yourapplication.tar.gz') # 使用我们虚拟环境下的 Python 解释器安装我们的包 run('/var/www/yourapplication/env/bin/python setup.py install') # 现在我们的代码已经部署成功了，可以删除这个文件夹了 run('rm -rf /tmp/yourapplication /tmp/yourapplication.tar.gz') # 最终生成 .wsgi 文件，以便于 mod\_wsgi 重新加载应用程序 run('touch /var/www/yourapplication.wsgi')

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

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

## 运行 Fabfile

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

$ fab pack deploy

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

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

$ fab -H newserver.example.com bootstrap

初始化一个新的服务器，您大概需要执行如下几个步骤:

1. 在 /var/www 目录下创建目录结构:

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

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

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

## WSGI 文件

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

import os os.environ['YOURAPPLICATION\_CONFIG'] = '/var/www/yourapplication/application.cfg' from yourapplication import app

应用程序本身则应该向下面这样，通过查询环境变量来查找配置，以此初始化自己:

app = Flask(\_\_name\_\_) app.config.from\_object('yourapplication.default\_config') app.config.from\_envvar('YOURAPPLICATION\_CONFIG')

这种方法在本文档的 [配置处理](#Pei_Zhi_Chu_Li___0_3_Xin_Ban_Gon) 这节中进行了详细介绍。

## 配置文件

正如上文所属，应用程序将会通过查找 YOURAPPLICATION\_CONFIG 环境变量以 找到正确的配置文件。因此我们必须将配置文件放在应用程序可以找到的地方。 配置文件有在不同电脑上表现出不同效果的特质，所以您不应该以普通的方式 对它进行版本控制。

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

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

## 第一次部署

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

$ fab pack deploy

Fabric 现在就会连接到所有服务器，然后运行在 fabfile 文件中所指定的命令。 最初他会执行打包工作，为我们创建代码归档，然后他部署和上传代码到所有的 服务器，并在那里安装他们。归功于 setup.py ，所有引用依赖的包和库都将 自动被下载和安装到我们的虚拟环境中。

## 下一步操作

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

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

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

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_sqlalchemy_html) | [上一页](#Top_of_fabric_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 在 Flask 中使用 SQLite 3

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

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

import sqlite3 from flask import g DATABASE = '/path/to/database.db' def connect\_db(): return sqlite3.connect(DATABASE) @app.before\_request def before\_request(): g.db = connect\_db() @app.teardown\_request def teardown\_request(exception): if hasattr(g, 'db'): g.db.close()

注解

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

## 按需连接

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

with app.test\_request\_context(): app.preprocess\_request() # now you can use the g.db object

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

def get\_connection(): db = getattr(g, '\_db', None) if db is None: db = g.\_db = connect\_db() return db

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

## 简化查询

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

def query\_db(query, args=(), one=False): cur = g.db.execute(query, args) rv = [dict((cur.description[idx][0], value) for idx, value in enumerate(row)) for row in cur.fetchall()] return (rv[0] if rv else None) if one else rv

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

for user in query\_db('select \* from users'): print user['username'], 'has the id', user['user\_id']

如果您只希望得到一个单独的结果:

user = query\_db('select \* from users where username = ?', [the\_username], one=True) if user is None: print 'No such user' else: print the\_username, 'has the id', user['user\_id']

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

## 初始化数据库模型

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

from contextlib import closing def init\_db(): with closing(connect\_db()) as db: with app.open\_resource('schema.sql') as f: db.cursor().executescript(f.read()) db.commit()

然后您就可以在 Python 的交互式终端中创建一个这样的数据库:

>>> from yourapplication import init\_db >>> init\_db()

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_fileuploads_html) | [上一页](#Top_of_sqlite3_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 在 Flask 中使用 SQLAlchemy

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

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

## Flask-SQLAlchemy 扩展

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

您可以从 [PyPI](http://pypi.python.org/pypi/Flask-SQLAlchemy) [http://pypi.python.org/pypi/Flask-SQLAlchemy] 下载到 [Flask-SQLAlchemy](http://packages.python.org/Flask-SQLAlchemy/) [http://packages.python.org/Flask-SQLAlchemy/]

## 显式调用

SQLAlchemy 中的 declarative 扩展是最新的使用 SQLAlchemy 的方法。它允许您 同时定义表和模型，就像 Django 一样工作。除了下文所介绍的内容外，我们建议您 参考 [declarative](http://www.sqlalchemy.org/docs/orm/extensions/declarative.html) [http://www.sqlalchemy.org/docs/orm/extensions/declarative.html] 扩展的官方文档。

这是一个 database.py 模块的例子:

from sqlalchemy import create\_engine from sqlalchemy.orm import scoped\_session, sessionmaker from sqlalchemy.ext.declarative import declarative\_base engine = create\_engine('sqlite:////tmp/test.db', convert\_unicode=True) db\_session = scoped\_session(sessionmaker(autocommit=False, autoflush=False, bind=engine)) Base = declarative\_base() Base.query = db\_session.query\_property() def init\_db(): # 在这里导入所有的可能与定义模型有关的模块，这样他们才会合适地 # 在 metadata 中注册。否则，您将不得不在第一次执行 init\_db() 时 # 先导入他们。 import yourapplication.models Base.metadata.create\_all(bind=engine)

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

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

from yourapplication.database import db\_session @app.teardown\_request def shutdown\_session(exception=None): db\_session.remove()

这是一个模型的例子(将代码放入 models.py 或类似文件中):

from sqlalchemy import Column, Integer, String from yourapplication.database import Base class User(Base): \_\_tablename\_\_ = 'users' id = Column(Integer, primary\_key=True) name = Column(String(50), unique=True) email = Column(String(120), unique=True) def \_\_init\_\_(self, name=None, email=None): self.name = name self.email = email def \_\_repr\_\_(self): return '<User %r>' % (self.name)

您可以使用 init\_db 函数创建一个数据库:

>>> from yourapplication.database import init\_db >>> init\_db()

按照如下方式将数据实体插入数据库:

>>> from yourapplication.database import db\_session >>> from yourapplication.models import User >>> u = User('admin', 'admin@localhost') >>> db\_session.add(u) >>> db\_session.commit()

查询代码也很简单:

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

## 手动实现 ORM

手动实现 ORM (对象关系映射) 相比前面的显式调用方法，既有一些优点，也有一些缺点。 主要差别在于这里的数据表和模型是分开定义的，然后再将其映射起来。这提供了更大的灵活性， 但是会增加了代码量。通常来说它和上面显式调用的工作的方式很相似，所以请确保您的应用已经 被合理分割到了包中的不同模块中。

这是一个 database.py 模块的例子:

from sqlalchemy import create\_engine, MetaData from sqlalchemy.orm import scoped\_session, sessionmaker engine = create\_engine('sqlite:////tmp/test.db', convert\_unicode=True) metadata = MetaData() db\_session = scoped\_session(sessionmaker(autocommit=False, autoflush=False, bind=engine)) def init\_db(): metadata.create\_all(bind=engine)

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

from yourapplication.database import db\_session @app.teardown\_request def shutdown\_session(exception=None): db\_session.remove()

下面是一个数据表和模型的例子(将他们放到 models.py 当中):

from sqlalchemy import Table, Column, Integer, String from sqlalchemy.orm import mapper from yourapplication.database import metadata, db\_session class User(object): query = db\_session.query\_property() def \_\_init\_\_(self, name=None, email=None): self.name = name self.email = email def \_\_repr\_\_(self): return '<User %r>' % (self.name) users = Table('users', metadata, Column('id', Integer, primary\_key=True), Column('name', String(50), unique=True), Column('email', String(120), unique=True) ) mapper(User, users)

查询和插入操作和上面所给出的例子是一样的。

## SQL 抽象层

如果您仅用到数据库系统和 SQL 抽象层，那么您只需要引擎部分:

from sqlalchemy import create\_engine, MetaData engine = create\_engine('sqlite:////tmp/test.db', convert\_unicode=True) metadata = MetaData(bind=engine)

然后您就可以像上文的例子一样声明数据表，或者像下面这样自动加载他们:

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

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

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

SQLAlchemy 将会为我们自动提交对数据库的修改。

查询数据可以直接通过数据库引擎，也可以使用一个数据库连接:

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

返回的结果也是字典样式的元组:

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

您也可以将 SQL 语句的字符串传入到 execute() 函数中:

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

更多 SQLAlchemy 相关信息，请参考 [其网站](http://sqlalchemy.org/) [http://sqlalchemy.org/].

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_caching_html) | [上一页](#Top_of_sqlalchemy_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 上传文件

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

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

## 一点点介绍

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

import os from flask import Flask, request, redirect, url\_for from werkzeug import secure\_filename UPLOAD\_FOLDER = '/path/to/the/uploads' ALLOWED\_EXTENSIONS = set(['txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif']) app = Flask(\_\_name\_\_) app.config['UPLOAD\_FOLDER'] = UPLOAD\_FOLDER

首先我们导入一些东西，大多数内容都是直接而容易的。werkzeug.secure\_filename() 将会在稍后进行解释。 UPLOAD\_FOLDER 是我们储存上传的文件的地方，而 ALLOWED\_EXTENSIONS 则是允许的文件类型的集合。然后我们手动为应用添加一个的 URL 规则。我们 通常很少这样做，但是为什么这里要如此呢？原因是我们希望实际部署的服务器 (或者我们的开发服务器）来为我们提供这些文件的访问服务，所以我们只需要 一个规则用来生成指向这些文件的 URL 。

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

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

def allowed\_file(filename): return '.' in filename and \ filename.rsplit('.', 1)[1] in ALLOWED\_EXTENSIONS @app.route('/', methods=['GET', 'POST']) def upload\_file(): if request.method == 'POST': file = request.files['file'] if file and allowed\_file(file.filename): filename = secure\_filename(file.filename) file.save(os.path.join(app.config['UPLOAD\_FOLDER'], filename)) return redirect(url\_for('uploaded\_file', filename=filename)) return ''' <!doctype html> <title>Upload new File</title> <h1>Upload new File</h1> <form action="" method=post enctype=multipart/form-data> <p><input type=file name=file> <input type=submit value=Upload> </form> '''

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

关于文件名安全的更多信息

您对 [secure\_filename()](http://werkzeug.pocoo.org/docs/utils/#werkzeug.utils.secure_filename) [http://werkzeug.pocoo.org/docs/utils/#werkzeug.utils.secure\_filename] 的具体工作和您没使用它会造成的后果 感兴趣？试想一个人可以发送下列信息作为 filename 给您的应用:

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

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

现在我们来研究一下这个函数的功能:

>>> secure\_filename('../../../../home/username/.bashrc') 'home\_username\_.bashrc'

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

from flask import send\_from\_directory @app.route('/uploads/<filename>') def uploaded\_file(filename): return send\_from\_directory(app.config['UPLOAD\_FOLDER'], filename)

或者，您也可以选择为 uploaded\_file 注册 build\_only 规则，然后使用 [SharedDataMiddleware](http://werkzeug.pocoo.org/docs/middlewares/#werkzeug.wsgi.SharedDataMiddleware) [http://werkzeug.pocoo.org/docs/middlewares/#werkzeug.wsgi.SharedDataMiddleware] 类来实现下载服务。这种方法 同时支持更老版本的 Flask:

from werkzeug import SharedDataMiddleware app.add\_url\_rule('/uploads/<filename>', 'uploaded\_file', build\_only=True) app.wsgi\_app = SharedDataMiddleware(app.wsgi\_app, { '/uploads': app.config['UPLOAD\_FOLDER'] })

运行应用，不出意外的话，一切都应该像预期那样工作了。

## 改进上传功能

0.6 新版功能.

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

from flask import Flask, Request app = Flask(\_\_name\_\_) app.config['MAX\_CONTENT\_LENGTH'] = 16 \* 1024 \* 1024

上面的代码将会把上传文件限制为最大 16 MB 。 如果请求传输一个更大的文件， Flask 会抛出一个 [RequestEntityTooLarge](http://werkzeug.pocoo.org/docs/exceptions/#werkzeug.exceptions.RequestEntityTooLarge) [http://werkzeug.pocoo.org/docs/exceptions/#werkzeug.exceptions.RequestEntityTooLarge] 异常。

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

## 上传进度条

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

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

* [Plupload](http://www.plupload.com/) [http://www.plupload.com/] - HTML5, Java, Flash
* [SWFUpload](http://www.swfupload.org/) [http://www.swfupload.org/] - Flash
* [JumpLoader](http://jumploader.com/) [http://jumploader.com/] - Java

## 更简单解决方案

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

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_viewdecorators_html) | [上一页](#Top_of_fileuploads_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 缓存

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

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

## 配置缓存

类似于建立 [Flask](#class_flask_Flask_import_name__s) 的对象一样，您创建一个缓存对象，然后让他 保持存在。如果您使用的是开发服务器，您可以创建一个 [SimpleCache](http://werkzeug.pocoo.org/docs/contrib/cache/#werkzeug.contrib.cache.SimpleCache) [http://werkzeug.pocoo.org/docs/contrib/cache/#werkzeug.contrib.cache.SimpleCache] 对象，这个对象将元素缓存在 Python 解释器的控制的内存中:

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

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

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

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

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

## 使用缓存

有两个非常重要的函数可以用来使用缓存。那就是 [get()](http://werkzeug.pocoo.org/docs/contrib/cache/#werkzeug.contrib.cache.BaseCache.get) [http://werkzeug.pocoo.org/docs/contrib/cache/#werkzeug.contrib.cache.BaseCache.get] 函数和 [set()](http://werkzeug.pocoo.org/docs/contrib/cache/#werkzeug.contrib.cache.BaseCache.set) [http://werkzeug.pocoo.org/docs/contrib/cache/#werkzeug.contrib.cache.BaseCache.set] 函数。他们的使用方法 如下:

从缓存中读取项目，请使用 [get()](http://werkzeug.pocoo.org/docs/contrib/cache/#werkzeug.contrib.cache.BaseCache.get) [http://werkzeug.pocoo.org/docs/contrib/cache/#werkzeug.contrib.cache.BaseCache.get] 函数， 如果现在缓存中存在对应项目，它将会返回。否则函数将会返回 None

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

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

以下是一个通常情况下实现功能完整例子:

def get\_my\_item(): rv = cache.get('my-item') if rv is None: rv = calculate\_value() cache.set('my-item', rv, timeout=5 \* 60) return rv

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_wtforms_html) | [上一页](#Top_of_caching_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 视图装饰器

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

## 过滤未登录用户的装饰器

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

这个例子家丁登陆页面的名字是 'login' 并且当前用户被保存在 g.user 当中， 如果么有用户登陆， g.user 会是 None:

from functools import wraps from flask import g, request, redirect, url\_for def login\_required(f): @wraps(f) def decorated\_function(\*args, \*\*kwargs): if g.user is None: return redirect(url\_for('login', next=request.url)) return f(\*args, \*\*kwargs) return decorated\_function

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

@app.route('/secret\_page') @login\_required def secret\_page(): pass

## 缓存装饰器

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

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

被装饰器包装的函数将能做到如下几点:

1. 以当前请求和路径为基础生成缓存时使用的键。
2. 从缓存中取出对应键的值，如果缓存返回的不是空，我们就将它返回回去。
3. 如果缓存中没有这个键，那么最初的函数将会被执行，并且返回的值在指定时间 (默认5分钟内)被缓存起来。

代码如下:

from functools import wraps from flask import request def cached(timeout=5 \* 60, key='view/%s'): def decorator(f): @wraps(f) def decorated\_function(\*args, \*\*kwargs): cache\_key = key % request.path rv = cache.get(cache\_key) if rv is not None: return rv rv = f(\*args, \*\*kwargs) cache.set(cache\_key, rv, timeout=timeout) return rv return decorated\_function return decorator

注意，这段代码假定一个示例用的 cache 对象时可用的。请参考 [缓存](#Huan_Cun__Ru_Guo_Nin_De_Ying_Yon) 以获取更多信息。

## 模板装饰器

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

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

正如您所看到的，如果没有模板名被指定，那么他会使用 URL 映射的最后一部分， 然后将点转换为反斜杠，最后添加上 '.html' 作为模板的名字。当装饰器 包装的函数返回，返回的字典就会被传递给模板渲染函数。如果 None 被返回 了，那么相当于一个空的字典。如果非字典类型的对象被返回，函数将照原样 将那个对象再次返回。这样您就可以继续使用重定向函数或者返回简单的字符串了。

这是那个装饰器的源代码:

from functools import wraps from flask import request def templated(template=None): def decorator(f): @wraps(f) def decorated\_function(\*args, \*\*kwargs): template\_name = template if template\_name is None: template\_name = request.endpoint \ .replace('.', '/') + '.html' ctx = f(\*args, \*\*kwargs) if ctx is None: ctx = {} elif not isinstance(ctx, dict): return ctx return render\_template(template\_name, \*\*ctx) return decorated\_function return decorator

## 终端装饰器

如果您希望使用 werkzeug 路由系统来获得更多的灵活性。您需要将终点(Endpoint) 像 [Rule](http://werkzeug.pocoo.org/docs/routing/#werkzeug.routing.Rule) [http://werkzeug.pocoo.org/docs/routing/#werkzeug.routing.Rule] 中定义的那样映射起来。通过一个装饰器 是可以做到的，例如:

from flask import Flask from werkzeug.routing import Rule app = Flask(\_\_name\_\_) app.url\_map.add(Rule('/', endpoint='index')) @app.endpoint('index') def my\_index(): return "Hello world"

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_templateinheritance_html) | [上一页](#Top_of_viewdecorators_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 使用 WTForms 进行表单验证

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

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

挖掘 WTForms 的最大潜力

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

## 表单

以下是一个典型的注册页面的例子:

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

## 在视图里

在视图函数中，表单的使用是像下面这个样子的:

@app.route('/register', methods=['GET', 'POST']) def register(): form = RegistrationForm(request.form) if request.method == 'POST' and form.validate(): user = User(form.username.data, form.email.data, form.password.data) db\_session.add(user) flash('Thanks for registering') return redirect(url\_for('login')) return render\_template('register.html', form=form)

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

备忘表:

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

## 在模板中使用表单

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

以下是一个使用这种宏的 \_formhelpers.html 模板的例子:

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

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

以下是 register.html 模板，它对应于上面我们使用过的函数，同时也利用 了 \_formhelpers.html 模板:

{% from "\_formhelpers.html" import render\_field %} <form method=post action="/register"> <dl> {{ render\_field(form.username) }} {{ render\_field(form.email) }} {{ render\_field(form.password) }} {{ render\_field(form.confirm) }} {{ render\_field(form.accept\_tos) }} </dl> <p><input type=submit value=Register> </form>

关于 WTForms 的更多信息，请访问 [WTForms 网站](http://wtforms.simplecodes.com/) [http://wtforms.simplecodes.com/] 。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_flashing_html) | [上一页](#Top_of_wtforms_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 模板继承

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

听起来虽然复杂，但是其实非常初级。理解概念的最好方法就是通过例子。

## 基础模板

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

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

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

## 子模板

子模板看起来像这个样子:

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

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

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_jquery_html) | [上一页](#Top_of_templateinheritance_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 消息闪现

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

## 简单的闪现

这里是一个完成的例子:

from flask import Flask, flash, redirect, render\_template, \ request, url\_for app = Flask(\_\_name\_\_) app.secret\_key = 'some\_secret' @app.route('/') def index(): return render\_template('index.html') @app.route('/login', methods=['GET', 'POST']) def login(): error = None if request.method == 'POST': if request.form['username'] != 'admin' or \ request.form['password'] != 'secret': error = 'Invalid credentials' else: flash('You were successfully logged in') return redirect(url\_for('index')) return render\_template('login.html', error=error) if \_\_name\_\_ == "\_\_main\_\_": app.run()

这里的 layout.html 模板完成了所有的魔术:

<!doctype html> <title>My Application</title> {% with messages = get\_flashed\_messages() %} {% if messages %} <ul class=flashes> {% for message in messages %} <li>{{ message }}</li> {% endfor %} </ul> {% endif %} {% endwith %} {% block body %}{% endblock %}

这里是 index.html 模板:

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

这里是登陆模板:

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

## 分类闪现

0.3 新版功能.

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

要使用一个自定义的分类，只要使用 [flash()](#flask_flash_message__category__m) 函数的第二个参数:

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

在模板中，您接下来可以调用 [get\_flashed\_messages()](#flask_get_flashed_messages_with) 函数来返回 这个分类，在下面的情景中，循环看起来将会有一点点不一样:

{% with messages = get\_flashed\_messages(with\_categories=true) %} {% if messages %} <ul class=flashes> {% for category, message in messages %} <li class="{{ category }}">{{ message }}</li> {% endfor %} </ul> {% endif %} {% endwith %}

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

## 过滤闪现消息

0.9 新版功能.

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

{% with errors = get\_flashed\_messages(category\_filter=["error"]) %} {% if errors %} <div class="alert-message block-message error"> <a class="close" href="#">×</a> <ul> {%- for msg in errors %} <li>{{ msg }}</li> {% endfor -%} </ul> </div> {% endif %} {% endwith %}

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_errorpages_html) | [上一页](#Top_of_flashing_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 用 jQuery 实现 Ajax

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

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

如果您使用 Python 2.6 以上版本，JSON 的解析库是开箱即用的。在 Python 2.5 中 您则必须从 PyPI 安装 [simplejson](http://pypi.python.org/pypi/simplejson) [http://pypi.python.org/pypi/simplejson] 库。

## 加载 jQuery

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

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

另一个加载 jQuery 的技巧是使用 Google 的 [AJAX Libraries API](http://code.google.com/apis/ajaxlibs/documentation/) [http://code.google.com/apis/ajaxlibs/documentation/] :

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

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

## 我的站点在哪?

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

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

<script type=text/javascript> $SCRIPT\_ROOT = {{ request.script\_root|tojson|safe }}; </script>

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

可能有用的信息

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

## JSON 视图函数

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

from flask import Flask, jsonify, render\_template, request app = Flask(\_\_name\_\_) @app.route('/\_add\_numbers') def add\_numbers(): a = request.args.get('a', 0, type=int) b = request.args.get('b', 0, type=int) return jsonify(result=a + b) @app.route('/') def index(): return render\_template('index.html')

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

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

## HTML 部分

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

<script type=text/javascript> $(function() { $('a#calculate').bind('click', function() { $.getJSON($SCRIPT\_ROOT + '/\_add\_numbers', { a: $('input[name="a"]').val(), b: $('input[name="b"]').val() }, function(data) { $("#result").text(data.result); }); return false; }); }); </script> <h1>jQuery Example</h1> <p><input type=text size=5 name=a> + <input type=text size=5 name=b> = <span id=result>?</span> <p><a href=# id=calculate>calculate server side</a>

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

1. $(function() { ... }) 将会在浏览器加载完页面的基础内容之后立即执行。
2. $('selector') 选择一个用于操作的元素。
3. element.bind('event', func) 指定元素被单击时运行的函数，如果这个函数 返回 false ，那么单击操作的默认行为将被取消。在本例中，点击操作的默认 行为是导航到 # 链接标签。
4. $.getJSON(url, data, func) 发送一个 GET 请求给 url ，其中 data 对象的内容将以查询参数的形式发送。一旦数据抵达，它将以返回值作为参数执行 给定的函数。请注意，我们在这里可以使用我们先前设定的 $SCRIPT\_ROOT 变量。

如果您还没有完全了解这个例子，可以从 github 上下载 [本例源码](http://github.com/mitsuhiko/flask/tree/master/examples/jqueryexample) [http://github.com/mitsuhiko/flask/tree/master/examples/jqueryexample] 。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_lazyloading_html) | [上一页](#Top_of_jquery_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 自定义错误页面

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

依赖于错误代码的不同，用户看到某个错误的可能性大小也不同。

## 通常的错误代码

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

404 Not Found

经典的“哎呦，您输入的 URL 当中有错误”消息。这个消息太常见了，即使是 互联网的新手也知道 404 代号的意义: 该死，我寻找的东西不在那儿。确保 404 页面上有一些有用的信息是一个好主意，至少应该提供一个返回主页的链接。

403 Forbidden

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

410 Gone

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

500 Internal Server Error

通常在出现编程错误或者服务器过载的时候会返回这个错误代号。在这里放一个 漂亮的页面是一个非常好的主意。因为您的应用 总有一天 会出现错误(请参考 [记录应用错误](#Ji_Lu_Ying_Yong_Cuo_Wu___0_3_Xin) )

## 错误处理器

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

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

如下实现了一个 “404 Page Not Found” 错误处理的例子:

from flask import render\_template @app.errorhandler(404) def page\_not\_found(e): return render\_template('404.html'), 404

一个示例模板可能会如下所示:

{% extends "layout.html" %} {% block title %}Page Not Found{% endblock %} {% block body %} <h1>Page Not Found</h1> <p>What you were looking for is just not there. <p><a href="{{ url\_for('index') }}">go somewhere nice</a> {% endblock %}

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_mongokit_html) | [上一页](#Top_of_errorpages_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 延迟加载视图

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

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

用于激活中央 URL 映射的函数是 [add\_url\_rule()](#add_url_rule__args____kwargs) 方法。 您需要提供一个设置应用程序所有 URL 的文件，而不是使用装饰器。

## 转换到中央 URL 映射

假象现在的应用的样子如下所示:

from flask import Flask app = Flask(\_\_name\_\_) @app.route('/') def index(): pass @app.route('/user/<username>') def user(username): pass

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

def index(): pass def user(username): pass

然后使用一个文件初始化应用并将函数映射到 URLs:

from flask import Flask from yourapplication import views app = Flask(\_\_name\_\_) app.add\_url\_rule('/', view\_func=views.index) app.add\_url\_rule('/user/<username>', view\_func=views.user)

## 延迟加载

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

from werkzeug import import\_string, cached\_property class LazyView(object): def \_\_init\_\_(self, import\_name): self.\_\_module\_\_, self.\_\_name\_\_ = import\_name.rsplit('.', 1) self.import\_name = import\_name @cached\_property def view(self): return import\_string(self.import\_name) def \_\_call\_\_(self, \*args, \*\*kwargs): return self.view(\*args, \*\*kwargs)

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

现在您就可以定义您将视图整合到的位置，如下所示:

from flask import Flask from yourapplication.helpers import LazyView app = Flask(\_\_name\_\_) app.add\_url\_rule('/', view\_func=LazyView('yourapplication.views.index')) app.add\_url\_rule('/user/<username>', view\_func=LazyView('yourapplication.views.user'))

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

def url(url\_rule, import\_name, \*\*options): view = LazyView('yourapplication.' + import\_name) app.add\_url\_rule(url\_rule, view\_func=view, \*\*options) url('/', 'views.index') url('/user/<username>', 'views.user')

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

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_favicon_html) | [上一页](#Top_of_lazyloading_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 在 Flask 中使用 MongoKit

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

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

使用 MongoKit 有两种常用的方法，我们将会逐一介绍:

## 显式调用

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

下面是一个 app.py 模块的例子:

from flask import Flask from mongokit import Connection, Document # configuration MONGODB\_HOST = 'localhost' MONGODB\_PORT = 27017 # create the little application object app = Flask(\_\_name\_\_) app.config.from\_object(\_\_name\_\_) # connect to the database connection = Connection(app.config['MONGODB\_HOST'], app.config['MONGODB\_PORT'])

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

以下是一个文档的例子 (您可以将这个也放进 app.py 文件里):

def max\_length(length): def validate(value): if len(value) <= length: return True raise Exception('%s must be at most %s characters long' % length) return validate class User(Document): structure = { 'name': unicode, 'email': unicode, } validators = { 'name': max\_length(50), 'email': max\_length(120) } use\_dot\_notation = True def \_\_repr\_\_(self): return '<User %r>' % (self.name) # register the User document with our current connection connection.register([User])

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

向数据库里添加数据的方法如下所示:

>>> from yourapplication.database import connection >>> from yourapplication.models import User >>> collection = connection['test'].users >>> user = collection.User() >>> user['name'] = u'admin' >>> user['email'] = u'admin@localhost' >>> user.save()

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

查询也很简单:

>>> list(collection.User.find()) [<User u'admin'>] >>> collection.User.find\_one({'name': u'admin'}) <User u'admin'>

## PyMongo 兼容层

如果您想直接使用 PyMongo 。 您也可以利用 MongoKit 实现。如果您希望应用程序实现 最佳的表现，您也许希望使用这种方法。注意，例子并没有展示配合 Flask 使用的具体 方法。请参考上面 MongoKit 的例子代码:

from MongoKit import Connection connection = Connection()

插入数据可以使用 insert 方法。我们必须先获得一个连接。这跟 在 SQL 的世界使用表有些类似。

>>> collection = connection['test'].users >>> user = {'name': u'admin', 'email': u'admin@localhost'} >>> collection.insert(user)

print list(collection.find()) print collection.find\_one({‘name’: u’admin’})

MongoKit 将会为我们自动提交修改。

查询数据库，您要直接使用数据库连接:

>>> list(collection.find()) [{u'\_id': ObjectId('4c271729e13823182f000000'), u'name': u'admin', u'email': u'admin@localhost'}] >>> collection.find\_one({'name': u'admin'}) {u'\_id': ObjectId('4c271729e13823182f000000'), u'name': u'admin', u'email': u'admin@localhost'}

返回的结果也同样是类字典的对象:

>>> r = collection.find\_one({'name': u'admin'}) >>> r['email'] u'admin@localhost'

关于 MongoKit 的更多信息，请访问 [website](https://github.com/namlook/mongokit) [https://github.com/namlook/mongokit].

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_streaming_html) | [上一页](#Top_of_mongokit_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 添加 Favicon

“Favicon” 是指您的网页浏览器显示在标签页或者历史记录里的图标。 这个图标能帮助用户将您的网站与其他网站区分开，因此请使用一个 独特的标志

一个普遍的问题是如何将一个 Favicon 添加到您的 Flask 应用中。首先，您当然得 先有一个可用的图标，此图标应该是 16 x 16 像素的，且格式为 ICO 。这些虽然不是 必需的规则，但是是被所有浏览器所支持的事实标准。将这个图标放置到您的静态文件 目录下，文件名为 favicon.ico 。

现在，为了让浏览器找到您的图标，正确的方法是添加一个 Link 标签到 HTML 当中 例如:

<link rel="shortcut icon" href="{{ url\_for('static', filename='favicon.ico') }}">

对于大多数浏览器来说，这就足够了。然后一些非常老的浏览器不支持这个标准。 原来的标准是在网站的根路径下，查找 favicon 文件，并使用它。如果应用程序 不是挂在在域名的根路径，您要么需要配置 Web 服务器来在根路径提供这一图标， 要么您就很不幸地无法实现这一功能了。然而，如果您饿应用是在根路径，您就可以 简单的配置一条重定向的路由:

app.add\_url\_rule('/favicon.ico', redirect\_to=url\_for('static', filename='favicon.ico'))

如果想要保存额外的重定向请求，您也可以使用 [send\_from\_directory()](#flask_send_from_directory_direct) 函数写一个视图函数:

import os from flask import send\_from\_directory @app.route('/favicon.ico') def favicon(): return send\_from\_directory(os.path.join(app.root\_path, 'static'), 'favicon.ico', mimetype='image/vnd.microsoft.icon')

我们可以不详细指定 mimetype ，浏览器将会自行猜测文件的类型。但是我们也可以 指定它以便于避免额外的猜测，因为这个 mimetype 总是固定的。

以上的代码将会通过您的应用程序来提供图标文件的访问。然而，如果可能的话 配置您的网页服务器来提供访问服务会更好。请参考对应网页服务器的文档。

## 参考

* Wikipedia 上有关 [Favicon](http://en.wikipedia.org/wiki/Favicon) [http://en.wikipedia.org/wiki/Favicon] 的文章

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_deferredcallbacks_html) | [上一页](#Top_of_favicon_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 数据流

有时，您希望发送非常巨量的数据到客户端，远远超过您可以保存在内存中的量。 在您实时地产生这些数据时，如何才能直接把他发送给客户端，而不需要在文件 系统中中转呢?

答案是生成器和 Direct Response。

## 基本使用

下面是一个简单的视图函数，这一视图函数实时生成大量的 CSV 数据， 这一技巧使用了一个内部函数，这一函数使用生成器来生成数据，并且 稍后激发这个生成器函数时，把返回值传递给一个 response 对象:

from flask import Response @app.route('/large.csv') def generate\_large\_csv(): def generate(): for row in iter\_all\_rows(): yield ','.join(row) + '\n' return Response(generate(), mimetype='text/csv')

每一个 yield 表达式直接被发送给浏览器。现在，仍然有一些 WSGI 中间件可能 打断数据流，所以在这里请注意那些在带缓存快照的调试环境，以及其他一些您可能 激活了的东西。

## 在模板中生成流

Jinja2 模板引擎同样支持分块逐个渲染模板。Flask 没有直接暴露这一功能到 模板中，因为它很少被用到，但是您可以很轻易的自己实现:

from flask import Response def stream\_template(template\_name, \*\*context): app.update\_template\_context(context) t = app.jinja\_env.get\_template(template\_name) rv = t.stream(context) rv.enable\_buffering(5) return rv @app.route('/my-large-page.html') def render\_large\_template(): rows = iter\_all\_rows() return Response(stream\_template('the\_template.html', rows=rows))

这一技巧是从应用程序上的 Jinja2 的环境中得到那个模板对象，然后调用 stream() 函数而不是 render() 函数。前者返回的是一个流对象，而不是后者的字符串。因为我们绕过了 Flask 的模板渲染函数，而是直接使用了模板对象，所以我们手动必须调用 [update\_template\_context()](#update_template_context_context) 函数来确保更新了模板的渲染上下文。 这一模板随后以流的方式迭代直到结束。因为每一次您使用使用一个 yield 。服务器 都会将所有的已经产生的内容塞给给客户端，因可能希望在模板中缓冲一部分元素 之后再发送，而不是每次都直接发送。您可以使用 rv.enable\_buffering(size) 来实现，size 的较为合理的默认值是 5 。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_methodoverrides_html) | [上一页](#Top_of_streaming_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 延迟请求回调

Flask 的设计原则中有一条是响应对象被创建并在一条可能的回调链中传递，而在 这条回调链但中的任意一个回调，您都可以修改或者替换掉他们。当请求开始被 处理时，还没有响应对象，响应对象将在这一过程中，被某个视图函数或者系统 的其他组件按照实际需要来闯将。

但是，如果您想在响应过程的结尾修改响应对象，但是这是对象还不存在，那么会发生 什么呢？一个常见的例子是您可能需要在 before-request 函数当中在响应对象上 设定 Cookie 。

解决这一情况的一个常用方法是改变代码的逻辑，将这一部分代码迁移到 after-request 回调中。然而有些时候这种迁移并不是一个非常容易的敬礼 而且可能使代码看起来非常糟糕。

一个可能的替代方法是将一些回调函数绑定到 [g](#flask_g) 对象中。然后在 请求结束的时候调用他们。使用这种方法，您可以从应用里的任何一个地方来指定 代码延迟执行。

## 装饰器

下面的装饰器就是关键，它将一个函数注册到 [g](#flask_g) 对象上的 一个函数列表中:

from flask import g def after\_this\_request(f): if not hasattr(g, 'after\_request\_callbacks'): g.after\_request\_callbacks = [] g.after\_request\_callbacks.append(f) return f

## 调用延迟函数

现在您可以使用 after\_this\_request 装饰器来将一个函数标记为在请求结束之后 执行，但是我们仍然需要手动调用他们。为此，如下函数将被注册为 [after\_request()](#after_request__args____kwargs) 回调:

@app.after\_request def call\_after\_request\_callbacks(response): for callback in getattr(g, 'after\_request\_callbacks', ()): response = callback(response) return response

## 一个实际应用的例子

现在我们可以在任何时间点将一个函数注册为在某个特定请求结束后执行，例如您可以 在 before-request 中将用户当前语言的信息保存在 Cookie 中:

from flask import request @app.before\_request def detect\_user\_language(): language = request.cookies.get('user\_lang') if language is None: language = guess\_language\_from\_request() @after\_this\_request def remember\_language(response): response.set\_cookie('user\_lang', language) g.language = language

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_requestchecksum_html) | [上一页](#Top_of_deferredcallbacks_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 添加 HTTP Method Overrides

某些 HTTP 代理不支持任意的 HTTP 方法或更新的 HTTP 方法（比如 PATCH）。 这种情况下，通过另一种完全违背协议的 HTTP 方法来“代理” HTTP 方法是可行 的。

这个方法使客户端发出 HTTP POST 请求并设置 X-HTTP-Method-Override 标头的值为想要的 HTTP 方法（比如 PATCH ）。

这很容易通过一个 HTTP 中间件来完成:

class HTTPMethodOverrideMiddleware(object): allowed\_methods = frozenset([ 'GET', 'HEAD', 'POST', 'DELETE', 'PUT', 'PATCH', 'OPTIONS' ]) bodyless\_methods = frozenset(['GET', 'HEAD', 'OPTIONS', 'DELETE']) def \_\_init\_\_(self, app): self.app = app def \_\_call\_\_(self, environ, start\_response): method = environ.get('HTTP\_X\_HTTP\_METHOD\_OVERRIDE', '').upper() if method in self.allowed\_methods: method = method.encode('ascii', 'replace') environ['REQUEST\_METHOD'] = method if method in self.bodyless\_methods: environ['CONTENT\_LENGTH'] = '0' return self.app(environ, start\_response)

在 Flask 中使用它的必要步骤见下:

from flask import Flask app = Flask(\_\_name\_\_) app.wsgi\_app = HTTPMethodOverrideMiddleware(app.wsgi\_app)

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_celery_html) | [上一页](#Top_of_methodoverrides_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 请求内容校验码

许多代码可以消耗请求数据并对其进行预处理。例如最终出现在已读取的请求对 象上的 JSON 数据、通过另外的代码路径出现的表单数据。当你想要校验收到的 请求数据时，这似乎带来不便。而有时这对某些 API 是必要的。

幸运的是，无论如何可以包装输入流来简单地改变这种状况。

下面的例子计算收到数据的 SHA1 校验码，它从 WSGI 环境中读取数据并把校验 码存放到其中:

import hashlib class ChecksumCalcStream(object): def \_\_init\_\_(self, stream): self.\_stream = stream self.\_hash = hashlib.sha1() def read(self, bytes): rv = self.\_stream.read(bytes) self.\_hash.update(rv) return rv def readline(self, size\_hint): rv = self.\_stream.readline(size\_hint) self.\_hash.update(rv) return rv def generate\_checksum(request): env = request.environ stream = ChecksumCalcStream(env['wsgi.input']) env['wsgi.input'] = stream return stream.\_hash

要使用这段代码，所有你需要做的就是在请求消耗数据之前调用计算流。（例如： 小心访问 request.form 或其它此类的东西。例如，应注意避免 before\_request\_handlers 访问它）。

用法示例:

@app.route('/special-api', methods=['POST']) def special\_api(): hash = generate\_checksum(request) # Accessing this parses the input stream files = request.files # At this point the hash is fully constructed. checksum = hash.hexdigest() return 'Hash was: %s' % checksum

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_index_html_3) | [上一页](#Top_of_requestchecksum_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [Flask 代码模式](#Top_of_index_html_2) »

# 基于 Celery 的后台任务

Celery 是一个 Python 的任务队列，包含线程/进程池。曾经有一个 Flask 的集成， 但在 Celery 3 重构了内部细节后变得不必要了。本指导补充了如何妥善在 Flask 中使用 Celery 的空白，但假设你已经读过了 Celery 官方文档中的教程 [使用 Celery 的首要步骤](http://docs.celeryproject.org/en/master/getting-started/first-steps-with-celery.html) [http://docs.celeryproject.org/en/master/getting-started/first-steps-with-celery.html]

## 安装 Celery

Celery 提交到了 Python Package Index (PyPI)，所以可以通过标准 Python 工具 pip 或 easy\_install 安装:

$ pip install celery

## 配置 Celery

你需要的第一个东西是一个 Celery 实例，称为 Celery 应用。仅就 Celery 而言 其与 Flask 中的 [Flask](#class_flask_Flask_import_name__s) 对象有异曲同工之妙。因为这个实例用 于你在 Celery 中做任何事——诸如创建任务和管理职程（Worker）——的入口点， 它必须可以在其它模块中导入。

例如，你可以把它放置到 tasks 模块中。虽然你可以在不重新配置 Flask 的 情况下使用 Celery，但继承任务、添加对 Flask 应用上下文的支持以及关联 Flask 配置会让情况变得更好。

这就是把 Celery 集成到 Flask 的全部必要步骤:

from celery import Celery def make\_celery(app): celery = Celery(app.import\_name, broker=app.config['CELERY\_BROKER\_URL']) celery.conf.update(app.config) TaskBase = celery.Task class ContextTask(TaskBase): abstract = True def \_\_call\_\_(self, \*args, \*\*kwargs): with app.app\_context(): return TaskBase.\_\_call\_\_(self, \*args, \*\*kwargs) celery.Task = ContextTask return celery

该函数创建一个新的 Celery 对象，并用应用配置来配置中间人（Broker）， 用 Flask 配置更新其余的 Celery 配置，之后在应用上下文中创建一个封装任务 执行的任务子类。

## 最简示例

通过上面的步骤，下面即是在 Flask 中使用 Celery 的最简示例:

from flask import Flask app = Flask(\_\_name\_\_) app.config.update( CELERY\_BROKER\_URL='redis://localhost:6379', CELERY\_RESULT\_BACKEND='redis://localhost:6379' ) celery = make\_celery(app) @celery.task() def add\_together(a, b): return a + b

这项任务可以在后台调用:

>>> result = add\_together.delay(23, 42) >>> result.wait() 65

## 运行 Celery 职程

现在如果你行动迅速，已经执行过了上述的代码，你会失望地得知 .wait() 永远不会实际地返回。这是因为你也需要运行 Celery。你可以这样把 Celery 以职程运行:

$ celery -A your\_application worker

your\_application 字符串需要指向创建 celery 对象的应用所在包或模块。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_mod_wsgi_html) | [上一页](#Top_of_celery_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 部署选择

取决于你现有的，有多种途径来运行 Flask 应用。你可以在开发过程中使用内置的 服务器，但是你应该为用于生产的应用选择使用完整的部署。（不要在生产环境中使 用内置的开发服务器）。这里给出几个可选择的方法并且给出了文档。

如果你有一个不同的 WSGI 服务器，查阅文档中关于如何用它运行一个 WSGI 应用 度部分。请记住你的 Flask 应用对象就是实际的 WSGI 应用。

选择托管服务来快速配置并运行，参阅快速上手中的 [部署到 Web 服务器](#quickstart_deployment) 部分。

* [mod\_wsgi (Apache)](#Top_of_mod_wsgi_html)
  + [安装 mod\_wsgi](#An_Zhuang__mod_wsgi_Ru_Guo_Ni_Hu)
  + [创建一个 .wsgi 文件](#Chuang_Jian_Yi_Ge___wsgi_Wen_Jia)
  + [配置 Apache](#Pei_Zhi__Apache_Ni_Xu_Yao_Zuo_De)
  + [故障排除](#Gu_Zhang_Pai_Chu__Ru_Guo_Ni_De_Y)
  + [自动重加载支持](#Zi_Dong_Zhong_Jia_Zai_Zhi_Chi__N)
  + [使用虚拟环境](#Shi_Yong_Xu_Ni_Huan_Jing__Xu_Ni)
* [独立 WSGI 容器](#Top_of_wsgi_standalone_html)
  + [Gunicorn](#Gunicorn_Gunicorn__http___gunico)
  + [Tornado](#Tornado_Tornado__http___www_torn)
  + [Gevent](#Gevent_Gevent__http___www_gevent)
  + [代理设置](#Dai_Li_She_Zhi__Ru_Guo_Ni_Zai_Yi)
* [uWSGI](#Top_of_uwsgi_html)
  + [用 uwsgi 启动你的应用](#Yong__uwsgi_Qi_Dong_Ni_De_Ying_Y)
  + [配置 nginx](#Pei_Zhi__nginx_Yi_Ge_Ji_Ben_De)
* [FastCGI](#Top_of_fastcgi_html)
  + [创建一个 .fcgi 文件](#Chuang_Jian_Yi_Ge___fcgi_Wen_Jia)
  + [配置 lighttpd](#Pei_Zhi__lighttpd_Yi_Ge_Gei__lig)
  + [配置 nginx](#Pei_Zhi__nginx_Zai__nginx_Shang)
  + [运行 FastCGI 进程](#Yun_Xing__FastCGI_Jin_Cheng__Ji)
  + [调试](#Diao_Shi__FastCGI_Zai_Da_Duo_Shu)
* [CGI](#Top_of_cgi_html)
  + [创建一个 .cgi 文件](#Chuang_Jian_Yi_Ge___cgi_Wen_Jian)
  + [服务器配置](#Fu_Wu_Qi_Pei_Zhi__Tong_Chang_You)

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_wsgi_standalone_html) | [上一页](#Top_of_index_html_3) | [Flask 0.10.1 文档](#Top_of_index_html) » [部署选择](#Top_of_index_html_3) »

# mod\_wsgi (Apache)

如果你使用 [Apache](http://httpd.apache.org/) [http://httpd.apache.org/] web 服务器，请考虑使用 [mod\_wsgi](http://code.google.com/p/modwsgi/) [http://code.google.com/p/modwsgi/] 。

注意

请确保在任何 app.run() 调用之前，你应该把应用文件放在一个 if \_\_name\_\_ == `\_\_main\_\_`: 块中或移动到独立的文件。只确保它没被调用是 因为这总是会启动一个本地的 WSGI 服务器，而当我们使用 mod\_wsgi 部署应用 时并不想让它出现。

## 安装 mod\_wsgi

如果你还没有安装过 mod\_wsgi ，你需要使用包管理器来安装或手动编译它。 mod\_wsgi 的 [安装指引](http://code.google.com/p/modwsgi/wiki/QuickInstallationGuide) [http://code.google.com/p/modwsgi/wiki/QuickInstallationGuide] 涵盖了 UNIX 系统中的源码安装。

如果你使用 Ubuntu/Debian 你可以按照下面的命令使用 apt-get 获取并激活它:

# apt-get install libapache2-mod-wsgi

在 FreeBSD 上，通过编译 www/mode\_wsgi port 或使用 pkg\_add 来安装:

# pkg\_add -r mod\_wsgi

如果你在使用 pkgsrc 你可以编译 www/ap2-wsgi 包来安装 mod\_wsgi 。

如果你在 apache 第一次重加载后遇到子进程段错误，你可以安全地忽略它们。 只需要重启服务器。

## 创建一个 .wsgi 文件

你需要一个 yourapplication.wsgi 文件来运行你的应用。这个文件包含 mod\_wsgi 启动时执行的获取应用对象的代码。这个对象在该文件中名为 application ，并在 之后作为应用。

对于大多数应用，下面度文件就可以胜任:

from yourapplication import app as application

如果你没有一个用于创建应用的工厂函数而是单例的应用，你可以直接导入它为 application 。

把这个文件放在你可以找到的地方（比如 /var/www/yourapplication ）并确保 yourapplication 和所有使用的库在 python 载入的路径。如果你不想在系统全局 安装它，请考虑使用 [virtual python](http://pypi.python.org/pypi/virtualenv) [http://pypi.python.org/pypi/virtualenv] 实例。记住你也会需要在 virtualenv 中安装应用。可选地，你可以在 .wsgi 文件中在导入前修补路径:

import sys sys.path.insert(0, '/path/to/the/application')

## 配置 Apache

你需要做的最后一件事情就是为你的应用创建一个 Apache 配置文件。在本例中，考虑 安全因素，我们让 mod\_wsgi 来在不同度用户下执行应用:

<VirtualHost \*> ServerName example.com WSGIDaemonProcess yourapplication user=user1 group=group1 threads=5 WSGIScriptAlias / /var/www/yourapplication/yourapplication.wsgi <Directory /var/www/yourapplication> WSGIProcessGroup yourapplication WSGIApplicationGroup %{GLOBAL} Order deny,allow Allow from all </Directory> </VirtualHost>

更多信息请翻阅 [mod\_wsgi wiki](http://code.google.com/p/modwsgi/wiki/) [http://code.google.com/p/modwsgi/wiki/] 。

## 故障排除

如果你的应用不能运行，按照下面的指导来排除故障:

问题: 应用不能运行，错误日志显示, SystemExit 忽略

你的应用文件中有一个不在 if \_\_name\_\_ == '\_\_main\_\_': 声明保护下的 app.run() 调用。把 [run()](#run_host_None__port_None__debug) 从文件中移除，或者把 它移到一个独立的 run.py 文件，再或者把它放到这样一个 if 块中。

问题: 应用报出权限错误

可能是因为使用了错误的用户运行应用。确保需要访问的应用有合适的权限设置， 并且使用正确的用户来运行（ WSGIDaemonProcess 指令的 user 和 group 参数）

问题: 应用崩溃时打印一条错误

记住 mod\_wsgi 禁止对 [sys.stdout](http://docs.python.org/dev/library/sys.html#sys.stdout) [http://docs.python.org/dev/library/sys.html#sys.stdout] 和 [sys.stderr](http://docs.python.org/dev/library/sys.html#sys.stderr) [http://docs.python.org/dev/library/sys.html#sys.stderr] 做操作。 你可以通过设定配置中的 WSGIRestrictStdout 为 off 来禁用这个保护。

WSGIRestrictStdout Off

或者，你可以在 .wsgi 文件中用不同的流来替换标准输出:

import sys sys.stdout = sys.stderr

问题: 访问资源时报出 IO 错误

你的应用可能是一个你符号链接到 site-packages 文件夹的单个 .py 文件。 请注意这不会正常工作，除非把这个文件放进 pythonpath 包含的文件夹中， 或是把应用转换成一个包。

这个问题同样适用于非安装的包，模块文件名用于定位资源，而符号链接会获取 错误的文件名。

## 自动重加载支持

你可以激活自动重载入支持来协助部署工具。无论何时，当 .wsgi 文件， mod\_wsgi 会为我们自动重新加载所有的守护进程。

为此，只需要直接在你的 Directory 节中添加如下内容:

WSGIScriptReloading On

## 使用虚拟环境

虚拟环境的优势是它们永远不在系统全局安装所需的依赖关系，这样你可以更好 地控制使用什么。如果你想要同 mod\_wsgi 使用虚拟环境，你需要稍微修改一下 .wsgi 文件。

把下面的几行添加到你 .wsgi 文件的顶部:

activate\_this = '/path/to/env/bin/activate\_this.py' execfile(activate\_this, dict(\_\_file\_\_=activate\_this))

这根据虚拟环境的设置设定了加载路径。记住这个路径一经是绝对的。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_uwsgi_html) | [上一页](#Top_of_mod_wsgi_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [部署选择](#Top_of_index_html_3) »

# 独立 WSGI 容器

有用 Python 编写的流行服务器来容纳 WSGI 应用并提供 HTTP 服务。这些服务器在运行 时是独立的：你可以从你的 web 服务器设置到它的代理。如果你遇见问题，请注意 [代理设置](#Dai_Li_She_Zhi__Ru_Guo_Ni_Zai_Yi) 一节的内容。

## Gunicorn

[Gunicorn](http://gunicorn.org/) [http://gunicorn.org/] ‘Green Unicorn’ 是一个给 UNIX 用的 WSGI HTTP 服务器。这是一个从 Ruby 的 Unicorn 项目移植的 pre-fork worker 模式。它既支持 [eventlet](http://eventlet.net/) [http://eventlet.net/] ，也 支持 [greenlet](http://codespeak.net/py/0.9.2/greenlet.html) [http://codespeak.net/py/0.9.2/greenlet.html] 。在这个服务器上运行 Flask 应用是相当简单的:

gunicorn myproject:app

[Gunicorn](http://gunicorn.org/) [http://gunicorn.org/] 提供了许多命令行选项 —— 见 gunicorn -h 。 例如，用四个 worker 进程（ gunicorn -h ）来运行一个 Flask 应用，绑定 到 localhost 的4000 端口（ -b 127.0.0.1:4000 ）:

gunicorn -w 4 -b 127.0.0.1:4000 myproject:app

## Tornado

[Tornado](http://www.tornadoweb.org/) [http://www.tornadoweb.org/] 是一个开源的可伸缩的、非阻塞式的 web 服务器和工具集，它驱动了 [FriendFeed](http://friendfeed.com/) [http://friendfeed.com/] 。因为它使用了 epoll 模型且是非阻塞的，它可以处理数以千计 的并发固定连接，这意味着它对实时 web 服务是理想的。把 Flask 集成这个服务 是直截了当的:

from tornado.wsgi import WSGIContainer from tornado.httpserver import HTTPServer from tornado.ioloop import IOLoop from yourapplication import app http\_server = HTTPServer(WSGIContainer(app)) http\_server.listen(5000) IOLoop.instance().start()

## Gevent

[Gevent](http://www.gevent.org/) [http://www.gevent.org/] 是一个基于协同程序的 Python 网络库，使用 [greenlet](http://codespeak.net/py/0.9.2/greenlet.html) [http://codespeak.net/py/0.9.2/greenlet.html] 来在 [libevent](http://monkey.org/~provos/libevent/) [http://monkey.org/~provos/libevent/] 的事件循环上提供高层的同步 API

from gevent.wsgi import WSGIServer from yourapplication import app http\_server = WSGIServer(('', 5000), app) http\_server.serve\_forever()

## 代理设置

如果你在一个 HTTP 代理后把你的应用部署到这些服务器中的之一，你需要重写一些标头 来让应用正常工作。在 WSGI 环境中两个有问题的值通常是 REMOTE\_ADDR 和 HTTP\_HOST 。你可以配置你的 httpd 来传递这些标头，或者在中间件中手动修正。 Werkzeug 带有一个修正工具来解决常见的配置，但是你可能想要为特定的安装自己写 WSGI 中间件。

这是一个简单的 nginx 配置，它监听 localhost 的 8000 端口，并提供到一个应用的 代理，设置了合适的标头:

server { listen 80; server\_name \_; access\_log /var/log/nginx/access.log; error\_log /var/log/nginx/error.log; location / { proxy\_pass http://127.0.0.1:8000/; proxy\_redirect off; proxy\_set\_header Host $host; proxy\_set\_header X-Real-IP $remote\_addr; proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for; } }

如果你的 httpd 不提供这些标头，最常见的配置引用从 X-Forwarded-Host 设置的主机 名和从 X-Forwarded-For 设置的远程地址:

from werkzeug.contrib.fixers import ProxyFix app.wsgi\_app = ProxyFix(app.wsgi\_app)

信任标头

请记住在一个非代理配置中使用这样一个中间件会是一个安全问题，因为它盲目地信任 一个可能由恶意客户端伪造的标头。

如果你想从另一个标头重写标头，你可能会使用这样的一个修正程序:

class CustomProxyFix(object): def \_\_init\_\_(self, app): self.app = app def \_\_call\_\_(self, environ, start\_response): host = environ.get('HTTP\_X\_FHOST', '') if host: environ['HTTP\_HOST'] = host return self.app(environ, start\_response) app.wsgi\_app = CustomProxyFix(app.wsgi\_app)

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_fastcgi_html) | [上一页](#Top_of_wsgi_standalone_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [部署选择](#Top_of_index_html_3) »

# uWSGI

uWSGI 是在像 [nginx](http://nginx.org/) [http://nginx.org/] 、 [lighttpd](http://www.lighttpd.net/) [http://www.lighttpd.net/] 以及 [cherokee](http://www.cherokee-project.com/) [http://www.cherokee-project.com/] 服务器上的一个部署的选 择。更多选择见 [FastCGI](#deploying_fastcgi) 和 [独立 WSGI 容器](#deploying_wsgi_standalone) 。 你会首先需要一个 uWSGI 服务器来用 uWSGI 协议来使用你的 WSGI 应用。 uWSGI 是 一个协议，同样也是一个应用服务器，可以提供 uWSGI 、FastCGI 和 HTTP 协议。

最流行的 uWSGI 服务器是 [uwsgi](http://projects.unbit.it/uwsgi/) [http://projects.unbit.it/uwsgi/] ，我们会在本指导中使用。确保你已经安装 好它来跟随下面的说明。

注意

请提前确保你在应用文件中的任何 app.run() 调用在 if \_\_name\_\_ == '\_\_main\_\_': 块中或是移到一个独立的文件。这是因为它总会启动一个本地 的 WSGI 服务器，并且我们在部署应用到 uWSGI 时不需要它。

## 用 uwsgi 启动你的应用

uwsgi 被设计为操作在 python 模块中找到的 WSGI 可调用量。

已知在 myapp.py 中有一个 flask 应用，使用下面的命令:

$ uwsgi -s /tmp/uwsgi.sock --module myapp --callable app

或者，你喜欢这样:

$ uwsgi -s /tmp/uwsgi.sock -w myapp:app

## 配置 nginx

一个基本的 flaks uWSGI 的给 nginx 的 配置看起来是这样:

location = /yourapplication { rewrite ^ /yourapplication/; } location /yourapplication { try\_files $uri @yourapplication; } location @yourapplication { include uwsgi\_params; uwsgi\_param SCRIPT\_NAME /yourapplication; uwsgi\_modifier1 30; uwsgi\_pass unix:/tmp/uwsgi.sock; }

这个配置绑定应用到 /yourapplication 。如果你想要绑定到 URL 根会更简单，因 你不许要告诉它 WSGI SCRIPT\_NAME 或设置 uwsgi modifier 来使用它:

location / { try\_files $uri @yourapplication; } location @yourapplication { include uwsgi\_params; uwsgi\_pass unix:/tmp/uwsgi.sock; }

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_cgi_html) | [上一页](#Top_of_uwsgi_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [部署选择](#Top_of_index_html_3) »

# FastCGI

FastCGI 是在像 [nginx](http://nginx.org/) [http://nginx.org/] 、 [lighttpd](http://www.lighttpd.net/) [http://www.lighttpd.net/] 和 [cherokee](http://www.cherokee-project.com/) [http://www.cherokee-project.com/] 服务器上的一个部署选择。 其它选择见 [uWSGI](#deploying_uwsgi) 和 [独立 WSGI 容器](#deploying_wsgi_standalone) 章节。 在它们上的任何一个运行你的 WSGI 应用首先需要一个 FastCGI 服务器。最流行的一个 是 [flup](http://trac.saddi.com/flup) [http://trac.saddi.com/flup] ，我们会在本指导中使用它。确保你已经安装好它来跟随下面的说明。

注意

请提前确保你在应用文件中的任何 app.run() 调用在 if \_\_name\_\_ == '\_\_main\_\_': 块中或是移到一个独立的文件。这是因为它总会启动一个本地 的 WSGI 服务器，并且我们在部署应用到 uWSGI 时不需要它。

## 创建一个 .fcgi 文件

首先你需要创建一个 FastCGI 服务器文件。让我们把它叫做 yourapplication.fcgi:

#!/usr/bin/python from flup.server.fcgi import WSGIServer from yourapplication import app if \_\_name\_\_ == '\_\_main\_\_': WSGIServer(app).run()

这已经可以为 Apache 工作，而 nginx 和老版本的 lighttpd 需要传递一个 显式的 socket 来与 FastCGI 通信。为此，你需要传递 socket 的路径到 WSGIServer:

WSGIServer(application, bindAddress='/path/to/fcgi.sock').run()

这个路径一定与你在服务器配置中定义的路径相同。

把 yourapplication.fcgi 文件保存到你能找到的地方。保存在 /var/www/yourapplication 或类似的地方是有道理的。

确保这个文件有执行权限，这样服务器才能执行它:

# chmod +x /var/www/yourapplication/yourapplication.fcgi

## 配置 lighttpd

一个给 lighttpd 的基本的 FastCGI 配置看起来是这样:

fastcgi.server = ("/yourapplication.fcgi" => (( "socket" => "/tmp/yourapplication-fcgi.sock", "bin-path" => "/var/www/yourapplication/yourapplication.fcgi", "check-local" => "disable", "max-procs" => 1 )) ) alias.url = ( "/static/" => "/path/to/your/static" ) url.rewrite-once = ( "^(/static.\*)$" => "$1", "^(/.\*)$" => "/yourapplication.fcgi$1"

记得启用 FastCGI ，别名和重写模块。这份配置把应用绑定到 /yourapplication 。如果想要应用运行在 URL 根路径，你需要用 LighttpdCGIRootFix 中间件来处理 一个 lighttpd 的 bug 。

确保只在应用挂载到 URL 根路径时才应用它。同样，更多信息请翻阅 Lighty 的文档关于 [FastCGI and Python](http://redmine.lighttpd.net/wiki/lighttpd/Docs:ModFastCGI) [http://redmine.lighttpd.net/wiki/lighttpd/Docs:ModFastCGI] 的部分（注意显示传递一个 socket 到 run() 不再是必须的）。

## 配置 nginx

在 nginx 上安装 FastCGI 应用有一点不同，因为默认没有 FastCGI 参数被转 发。

一个给 nginx 的基本的 FastCGI 配置看起来是这样:

location = /yourapplication { rewrite ^ /yourapplication/ last; } location /yourapplication { try\_files $uri @yourapplication; } location @yourapplication { include fastcgi\_params; fastcgi\_split\_path\_info ^(/yourapplication)(.\*)$; fastcgi\_param PATH\_INFO $fastcgi\_path\_info; fastcgi\_param SCRIPT\_NAME $fastcgi\_script\_name; fastcgi\_pass unix:/tmp/yourapplication-fcgi.sock; }

这份配置把应用绑定到 /yourapplication 。如果你想要绑定到 URL 跟了路径 会更简单，因为你不需要指出如何获取 PATH\_INFO 和 SCRIPT\_NAME:

location / { try\_files $uri @yourapplication; } location @yourapplication { include fastcgi\_params; fastcgi\_param PATH\_INFO $fastcgi\_script\_name; fastcgi\_param SCRIPT\_NAME ""; fastcgi\_pass unix:/tmp/yourapplication-fcgi.sock; }

## 运行 FastCGI 进程

既然 Nginx 和其它服务器并不加载 FastCGI 应用，你需要手动这么做。 [Supervisor 可以管理 FastCGI 进程。](http://supervisord.org/configuration.html#fcgi-program-x-section-settings) [http://supervisord.org/configuration.html#fcgi-program-x-section-settings] 你可以寻找其它 FastCGI 进程管理器或写一个启动时运行 .fcgi 文件的脚本， 例如使用一个 SysV init.d 脚本。对于临时的解决方案，你总是可以在 GNU screen 中运行 .fcgi 。更多细节见 man screen ，注意这是一个手动 的解决方案，并且不会在系统重启后保留:

$ screen $ /var/www/yourapplication/yourapplication.fcgi

## 调试

FastCGI 在大多数 web 服务器上的部署，对于调试趋于复杂。服务器日志最经常 告诉发生的事就是成行的“未预期的标头结尾”。为了调试应用，唯一可以让你了解 什么东西破碎的方案就是切换到正确的用户并手动执行应用。

这个例子假设你的应用叫做 application.fcgi 并且你的 web 服务器用户是 www-data:

$ su www-data $ cd /var/www/yourapplication $ python application.fcgi Traceback (most recent call last): File "yourapplication.fcgi", line 4, in <module> ImportError: No module named yourapplication

在这种情况下，错误看起来是“yourapplication”不在 python 路径下。常见的 问题是:

* 使用了相对路径。不要依赖于当前工作目录
* 代码依赖于不是从 web 服务器设置的环境变量
* 使用了不同的 python 解释器

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_becomingbig_html) | [上一页](#Top_of_fastcgi_html) | [Flask 0.10.1 文档](#Top_of_index_html) » [部署选择](#Top_of_index_html_3) »

# CGI

如果所有其它的部署方式都不能奏效，那么 CGI 毫无疑问会奏效。 CGI 被 所有主流服务器支持，但通常性能欠佳。

这也是你在 Google 的 [App Engine](http://code.google.com/appengine/) [http://code.google.com/appengine/] 上使用 Flask 应用的方式，其执行方式 恰好是一个 CGI-like 的环境。

注意

请提前确保你在应用文件中的任何 app.run() 调用在 if \_\_name\_\_ == '\_\_main\_\_': 块中或是移到一个独立的文件。这是因为它总会启动一个本地 的 WSGI 服务器，并且我们在部署应用到 uWSGI 时不需要它。

## 创建一个 .cgi 文件

首先你需要创建一个 CGI 应用程序文件。我们把它叫做 yourapplication.cgi:

#!/usr/bin/python from wsgiref.handlers import CGIHandler from yourapplication import app CGIHandler().run(app)

## 服务器配置

通常有两种方式来配置服务器。直接把 .cgi 复制到 cgi-bin （并且使用 mod\_rewrite 或其它类似的东西来重写 URL ） 或让服务器直接指向这个文件。

例如，在 Apache 中你可以在配置中写入这样的语句:

ScriptAlias /app /path/to/the/application.cgi

更多信息请查阅你的 web 服务器的文档。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_api_html) | [上一页](#Top_of_cgi_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 聚沙成塔

这里是增长你的代码库或是扩大应用规模的一些选择。

## 阅读源码

Flask 的创建一定程度上是为了展示如何在现有的常用工具 Werkzeug（WSGI） 和 Jinja（模板）之上构建你自己的框架，并且当它开发出来之后，它对广大 受众很有用。当你增长你的代码库时，不要仅仅使用 Flask——去理解它。阅读源码。 Flask 的代码是为了阅读而写；它是发布的文档，所以你可以使用它的内部 API。 Flask 坚持为上游库里的 API 写文档，并且为内部工具写文档，这样你们可以 找到你的项目需要的钩子注册点。

## 钩子，继承

[API](#API_Zhe_Bu_Fen_Wen_Dang_Han_Gai) 文档里面全都是可用的覆盖、钩子注册点和 [信号](#Xin_Hao___0_6_Xin_Ban_Gong_Neng) 。 你可以提供诸如请求和响应对象的自定义类。深入你所用的 API，并且在 Flask 中探寻框架外可用的定制。去寻找把你的项目重构为实用工具集合和 Flask 扩展的方法，探索社区中的大量的 [扩展](http://flask.pocoo.org/extensions/) [http://flask.pocoo.org/extensions/] ， 如果你没找到你需要的工具，就去寻找可以用于构建你自己扩展的 [模式](http://docs.torriacg.org/docs/flask/patterns/) [http://docs.torriacg.org/docs/flask/patterns/]。

## 继承

[Flask](#class_flask_Flask_import_name__s) 类有许多为继承设计的方法。你可以继承 [Flask](#class_flask_Flask_import_name__s) 快速添加或自定义行为（见链接的方法文档），并且 无论你在哪里实例化一个应用类都会使用那个子类。这与 [应用程序的工厂函数](#Ying_Yong_Cheng_Xu_De_Gong_Chang) 工作良好。

## 用中间件包装

[应用调度](#Ying_Yong_Diao_Du__Ying_Yong_Dia) 章节描述了如何应用中间件的细节。你可以引入 WSGI 中 间件来包装你的 Flask 实例并在 Flask 应用和 HTTP 服务器之间的中间层引入 修正和变更。Werkzeug包含了一些 [中间件](http://werkzeug.pocoo.org/docs/middlewares/) [http://werkzeug.pocoo.org/docs/middlewares/] 。

## 分支

如果上述选择都不奏效，分支(fork) Flask。Flask 的大部分代码都限定在 Werkzeug 和 Jinja2 中。这些库做了大部分工作。Flask 只作为胶水把它们粘合在一起。 对每个项目，都有一个底层框架带来阻碍的点（归咎于原始开发者的假设）。 这很正常，因为如果不是这样，框架本身会是一个非常复杂的系统，导致学习曲 线陡峭，给用户带来许多挫折。

不仅仅是对 Flask，许多人用打了补丁的或修改过的框架来弥补短处。这个思路 也体现在 Flask 的许可证上。如果你决定修改这个框架，你不需要回馈任何的修 改。

分支的消极面当然就是 Flask 扩展会更容易不可用，因为新的框架有一个不同 的导入名称。此外，集成上游的修改可能是一个复杂的过程，取决于修改的数目。 为此，分支应该作为最后手段。

## 像专家一样扩大规模

对许多 web 应用，代码的复杂程度比起为预期的用户或数据条目而扩大规模就不 是问题了。 Flask 自己扩大规模的限制只在于你的应用代码、你想用的数据存储 和 Python 解释器以及你运行的 web 服务器。

良好的规模扩张意味着，如果你把服务器的数量加倍，你会得到大约两倍于原来的性能。 而糟糕的则意味着，当你添加了一台新的服务器，应用不会有任何性能提升或根本不 支持第二台服务器。

在 Flask 中关于应用的扩张只有一个制约因素，那就是上下文局部代理。它们依赖于在 Flask 中上下文是被定义为是线程、还是进程或 greenlet。如果你的服务器使用不是基于 线程或 greenlet 的并行计算， Flask 不再能支持这些全局代理。然而大多数 服务器使用线程、 greenlet 或独立进程来实现并发，而这些方法在底层的 Werkzeug 库中有着良好的支持。

## 与社区对话

Flask 开发者维护框架对大小代码库用户的可理解性，所以一旦你遇到了由 Flask 引起的麻烦，不要犹豫，用邮件列表或 IRC 频道联系开发者。 Flask 和 Flask 扩展开发者为更大型应用改进的最佳途径就是从用户那里获取反馈。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_design_html) | [上一页](#Top_of_becomingbig_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# API

这部分文档涵盖了 Flask 的所有接口。对于那些 Flask 依赖外部库的部分，我们 这里提供了最重要的部分的文档，并且提供其官方文档的链接。

## 应用对象

class flask.Flask(import\_name, static\_path=None, static\_url\_path=None, static\_folder='static', template\_folder='templates', instance\_path=None, instance\_relative\_config=False)

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()](#open_resource_resource__mode__rb).

Usually you create a [Flask](#class_flask_Flask_import_name__s) instance in your main module or in the \_\_init\_\_.py file of your package like this:

from flask import Flask app = Flask(\_\_name\_\_)

About the First Parameter

The idea of the first parameter is to give Flask an idea what belongs to your application. This name is used to find resources on the file system, 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:

app = Flask('yourapplication') 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 set up, that debugging information is lost. (For example it would only pick up SQL queries in yourapplication.app and not yourapplication.views.frontend)

0.7 新版功能: The static\_url\_path, static\_folder, and template\_folder parameters were added.

0.8 新版功能: The instance\_path and instance\_relative\_config parameters were added.

|  |  |
| --- | --- |
| 参数: | * 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. * 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. |

add\_template\_filter(\*args, \*\*kwargs)

Register a custom template filter. Works exactly like the [template\_filter()](#template_filter__args____kwargs) decorator.

|  |  |
| --- | --- |
| 参数: | name – the optional name of the filter, otherwise the function name will be used. |

add\_template\_global(\*args, \*\*kwargs)

Register a custom template global function. Works exactly like the [template\_global()](#template_global__args____kwargs) decorator.

0.10 新版功能.

|  |  |
| --- | --- |
| 参数: | name – the optional name of the global function, otherwise the function name will be used. |

add\_template\_test(\*args, \*\*kwargs)

Register a custom template test. Works exactly like the [template\_test()](#template_test__args____kwargs) decorator.

0.10 新版功能.

|  |  |
| --- | --- |
| 参数: | name – the optional name of the test, otherwise the function name will be used. |

add\_url\_rule(\*args, \*\*kwargs)

Connects a URL rule. Works exactly like the [route()](#route_rule____options) decorator. If a view\_func is provided it will be registered with the endpoint.

Basically this example:

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

Is equivalent to the following:

def index(): pass 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:

app.view\_functions['index'] = index

Internally [route()](#route_rule____options) invokes [add\_url\_rule()](#add_url_rule__args____kwargs) so if you want to customize the behavior via subclassing you only need to change this method.

For more information refer to [URL 路由注册](#url_route_registrations).

在 0.2 版更改: view\_func parameter added.

在 0.6 版更改: OPTIONS is added automatically as method.

|  |  |
| --- | --- |
| 参数: | * rule – the URL rule as string * endpoint – the endpoint for the registered URL rule. Flask itself assumes the name of the view function as endpoint * view\_func – the function to call when serving a request to the provided endpoint * options – the options to be forwarded to the underlying [Rule](http://werkzeug.pocoo.org/docs/routing/#werkzeug.routing.Rule)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. |

after\_request(\*args, \*\*kwargs)

Register a function to be run after each request. Your function must take one parameter, a [response\_class](#response_class) object and return a new response object or the same (see [process\_response()](#process_response_response)).

As of Flask 0.7 this function might not be executed at the end of the request in case an unhandled exception occurred.

after\_request\_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 open database connections or getting hold of the currently logged in user. To register a function here, use the [after\_request()](#after_request__args____kwargs) decorator.

app\_context()

Binds the application only. For as long as the application is bound to the current context the [flask.current\_app](#flask_current_app) points to that application. An application context is automatically created when a request context is pushed if necessary.

Example usage:

with app.app\_context(): ...

0.9 新版功能.

app\_ctx\_globals\_class

The class that is used for the [g](#flask_g) instance.

Example use cases for a custom class:

1. Store arbitrary attributes on flask.g.
2. Add a property for lazy per-request database connectors.
3. Return None instead of AttributeError on expected attributes.
4. Raise exception if an unexpected attr is set, a “controlled” flask.g.

In Flask 0.9 this property was called request\_globals\_class but it was changed in 0.10 to [app\_ctx\_globals\_class](#app_ctx_globals_class) because the flask.g object is not application context scoped.

0.10 新版功能.

\_AppCtxGlobals 的别名

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.

0.8 新版功能.

before\_first\_request(\*args, \*\*kwargs)

Registers a function to be run before the first request to this instance of the application.

0.8 新版功能.

before\_first\_request\_funcs = None

A lists of functions that should be called at the beginning of the first request to this instance. To register a function here, use the [before\_first\_request()](#before_first_request__args____kw) decorator.

0.8 新版功能.

before\_request(\*args, \*\*kwargs)

Registers a function to run before each request.

before\_request\_funcs = None

A dictionary with lists of functions that should be called at the beginning of the 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 open database connections or getting hold of the currently logged in user. To register a function here, use the [before\_request()](#before_request__args____kwargs) 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.

0.7 新版功能.

config = None

The configuration dictionary as [Config](#class_flask_Config_root_path__de). This behaves exactly like a regular dictionary but supports additional methods to load a config from files.

context\_processor(\*args, \*\*kwargs)

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()](#jinja_loader) function instead.

The global loader dispatches between the loaders of the application and the individual blueprints.

0.7 新版功能.

create\_jinja\_environment()

Creates the Jinja2 environment based on [jinja\_options](#jinja_options___ImmutableDict) and [select\_jinja\_autoescape()](#select_jinja_autoescape_filename). Since 0.7 this also adds the Jinja2 globals and filters after initialization. Override this function to customize the behavior.

0.5 新版功能.

create\_url\_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 set up so the request is passed explicitly.

0.6 新版功能.

在 0.9 版更改: This can now also be called without a request object when the URL adapter is created for the application context.

debug

The debug flag. Set this to True to enable debugging of the application. In debug mode the debugger will kick in when an unhandled exception occurs and the integrated server will automatically reload the application if changes in the code are detected.

This attribute can also be configured from the config with the DEBUG configuration key. Defaults to False.

debug\_log\_format = '--------------------------------------------------------------------------------\n%(levelname)s in %(module)s [%(pathname)s:%(lineno)d]:\n%(message)s\n--------------------------------------------------------------------------------'

The logging format used for the debug logger. This is only used when the application is in debug mode, otherwise the attached logging handler does the formatting.

0.3 新版功能.

default\_config = ImmutableDict({'JSON\_AS\_ASCII': True, 'USE\_X\_SENDFILE': False, 'SESSION\_COOKIE\_PATH': None, 'SESSION\_COOKIE\_DOMAIN': None, 'SESSION\_COOKIE\_NAME': 'session', 'LOGGER\_NAME': None, 'DEBUG': False, 'SECRET\_KEY': None, 'MAX\_CONTENT\_LENGTH': None, 'APPLICATION\_ROOT': None, 'SERVER\_NAME': None, 'PREFERRED\_URL\_SCHEME': 'http', 'JSONIFY\_PRETTYPRINT\_REGULAR': True, 'TESTING': False, 'PERMANENT\_SESSION\_LIFETIME': datetime.timedelta(31), 'PROPAGATE\_EXCEPTIONS': None, 'TRAP\_BAD\_REQUEST\_ERRORS': False, 'JSON\_SORT\_KEYS': True, 'SESSION\_COOKIE\_HTTPONLY': True, 'SEND\_FILE\_MAX\_AGE\_DEFAULT': 43200, 'PRESERVE\_CONTEXT\_ON\_EXCEPTION': None, 'SESSION\_COOKIE\_SECURE': False, 'TRAP\_HTTP\_EXCEPTIONS': 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()](#flask_make_response__args).

在 0.7 版更改: This no longer does the exception handling, this code was moved to the new [full\_dispatch\_request()](#full_dispatch_request).

do\_teardown\_appcontext(exc=None)

Called when an application context is popped. This works pretty much the same as [do\_teardown\_request()](#do_teardown_request_exc_None) but for the application context.

0.9 新版功能.

do\_teardown\_request(exc=None)

Called after the actual request dispatching and will call every as [teardown\_request()](#teardown_request__args____kwargs) decorated function. This is not actually called by the [Flask](#class_flask_Flask_import_name__s) object itself but is always triggered when the request context is popped. That way we have a tighter control over certain resources under testing environments.

在 0.9 版更改: Added the exc argument. Previously this was always using the current exception information.

enable\_modules = True

Enable the deprecated module support? This is active by default in 0.7 but will be changed to False in 0.8. With Flask 1.0 modules will be removed in favor of Blueprints

endpoint(\*args, \*\*kwargs)

A decorator to register a function as an endpoint. Example:

@app.endpoint('example.endpoint') def example(): return "example"

|  |  |
| --- | --- |
| 参数: | endpoint – the name of the endpoint |

error\_handler\_spec = None

A dictionary of all registered error handlers. The key is None for error handlers active on the application, otherwise the key is the name of the blueprint. Each key points to another dictionary where they key is the status code of the http exception. The special key None points to a list of tuples where the first item is the class for the instance check and the second the error handler function.

To register a error handler, use the [errorhandler()](#errorhandler__args____kwargs) decorator.

errorhandler(\*args, \*\*kwargs)

A decorator that is used to register a function give a given error code. Example:

@app.errorhandler(404) def page\_not\_found(error): return 'This page does not exist', 404

You can also register handlers for arbitrary exceptions:

@app.errorhandler(DatabaseError) def special\_exception\_handler(error): return 'Database connection failed', 500

You can also register a function as error handler without using the [errorhandler()](#errorhandler__args____kwargs) decorator. The following example is equivalent to the one above:

def page\_not\_found(error): return 'This page does not exist', 404 app.error\_handler\_spec[None][404] = page\_not\_found

Setting error handlers via assignments to [error\_handler\_spec](#error_handler_spec___None) however is discouraged as it requires fiddling with nested dictionaries and the special case for arbitrary exception types.

The first None refers to the active blueprint. If the error handler should be application wide None shall be used.

0.7 新版功能: One can now additionally also register custom exception types that do not necessarily have to be a subclass of the HTTPException class.

|  |  |
| --- | --- |
| 参数: | code – the code as integer for the handler |

extensions = None

a place where extensions can store application specific state. For example this is where an extension could store database engines and similar things. For backwards compatibility extensions should register themselves like this:

if not hasattr(app, 'extensions'): app.extensions = {} app.extensions['extensionname'] = SomeObject()

The key must match the name of the flaskext module. For example in case of a “Flask-Foo” extension in flaskext.foo, the key would be 'foo'.

0.7 新版功能.

full\_dispatch\_request()

Dispatches the request and on top of that performs request pre and postprocessing as well as HTTP exception catching and error handling.

0.7 新版功能.

get\_send\_file\_max\_age(filename)

Provides default cache\_timeout for the [send\_file()](#flask_send_file_filename_or_fp) functions.

By default, this function returns SEND\_FILE\_MAX\_AGE\_DEFAULT from the configuration of [current\_app](#flask_current_app).

Static file functions such as [send\_from\_directory()](#flask_send_from_directory_direct) use this function, and [send\_file()](#flask_send_file_filename_or_fp) calls this function on [current\_app](#flask_current_app) when the given cache\_timeout is None. If a cache\_timeout is given in [send\_file()](#flask_send_file_filename_or_fp), that timeout is used; otherwise, this method is called.

This allows subclasses to change the behavior when sending files based on the filename. For example, to set the cache timeout for .js files to 60 seconds:

class MyFlask(flask.Flask): def get\_send\_file\_max\_age(self, name): if name.lower().endswith('.js'): return 60 return flask.Flask.get\_send\_file\_max\_age(self, name)

0.9 新版功能.

got\_first\_request

This attribute is set to True if the application started handling the first request.

0.8 新版功能.

handle\_exception(e)

Default exception handling that kicks in when an exception occurs that is not caught. In debug mode the exception will be re-raised immediately, otherwise it is logged and the handler for a 500 internal server error is used. If no such handler exists, a default 500 internal server error message is displayed.

0.3 新版功能.

handle\_http\_exception(e)

Handles an HTTP exception. By default this will invoke the registered error handlers and fall back to returning the exception as response.

0.3 新版功能.

handle\_url\_build\_error(error, endpoint, values)

Handle BuildError on [url\_for()](#flask_url_for_endpoint____values).

handle\_user\_exception(e)

This method is called whenever an exception occurs that should be handled. A special case are HTTPExceptions which are forwarded by this function to the [handle\_http\_exception()](#handle_http_exception_e) method. This function will either return a response value or reraise the exception with the same traceback.

0.7 新版功能.

has\_static\_folder

This is True if the package bound object’s container has a folder named 'static'.

0.5 新版功能.

init\_jinja\_globals()

Deprecated. Used to initialize the Jinja2 globals.

0.5 新版功能.

在 0.7 版更改: This method is deprecated with 0.7. Override [create\_jinja\_environment()](#create_jinja_environment) instead.

inject\_url\_defaults(endpoint, values)

Injects the URL defaults for the given endpoint directly into the values dictionary passed. This is used internally and automatically called on URL building.

0.7 新版功能.

instance\_path = None

Holds the path to the instance folder.

0.8 新版功能.

jinja\_env

The Jinja2 environment used to load templates.

jinja\_loader

The Jinja loader for this package bound object.

0.5 新版功能.

jinja\_options = ImmutableDict({'extensions': ['jinja2.ext.autoescape', 'jinja2.ext.with\_']})

Options that are passed directly to the Jinja2 environment.

json\_decoder

The JSON decoder class to use. Defaults to [JSONDecoder](#class_flask_json_JSONDecoder_enc).

0.10 新版功能.

JSONDecoder 的别名

json\_encoder

The JSON encoder class to use. Defaults to [JSONEncoder](#class_flask_json_JSONEncoder_ski).

0.10 新版功能.

JSONEncoder 的别名

log\_exception(exc\_info)

Logs an exception. This is called by [handle\_exception()](#handle_exception_e) if debugging is disabled and right before the handler is called. The default implementation logs the exception as error on the [logger](#logger).

0.8 新版功能.

logger

A [logging.Logger](http://docs.python.org/dev/library/logging.html#logging.Logger) [http://docs.python.org/dev/library/logging.html#logging.Logger] object for this application. The default configuration is to log to stderr if the application is in debug mode. This logger can be used to (surprise) log messages. Here some examples:

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

0.3 新版功能.

logger\_name

The name of the logger to use. By default the logger name is the package name passed to the constructor.

0.4 新版功能.

make\_config(instance\_relative=False)

Used to create the config attribute by the Flask constructor. The instance\_relative parameter is passed in from the constructor of Flask (there named instance\_relative\_config) and indicates if the config should be relative to the instance path or the root path of the application.

0.8 新版功能.

make\_default\_options\_response()

This method is called to create the default OPTIONS response. This can be changed through subclassing to change the default behavior of OPTIONS responses.

0.7 新版功能.

make\_null\_session()

Creates a new instance of a missing session. Instead of overriding this method we recommend replacing the [session\_interface](#session_interface____flask_sessi).

0.7 新版功能.

make\_response(rv)

Converts the return value from a view function to a real response object that is an instance of [response\_class](#response_class).

The following types are allowed for rv:

|  |  |
| --- | --- |
| [response\_class](#response_class) | the object is returned unchanged |
| [str](http://docs.python.org/dev/library/stdtypes.html#str) | a response object is created with the string as body |
| unicode | a response object is created with the string encoded to utf-8 as body |
| a WSGI function | the function is called as WSGI application and buffered as response object |
| [tuple](http://docs.python.org/dev/library/stdtypes.html#tuple) | A tuple in the form (response, status, headers) where response is any of the types defined here, status is a string or an integer and headers is a list of a dictionary with header values. |
| 参数: | rv – the return value from the view function |

在 0.9 版更改: Previously a tuple was interpreted as the arguments for the response object.

name

The name of the application. This is usually the import name with the difference that it’s guessed from the run file if the import name is main. This name is used as a display name when Flask needs the name of the application. It can be set and overridden to change the value.

0.8 新版功能.

open\_instance\_resource(resource, mode='rb')

Opens a resource from the application’s instance folder ([instance\_path](#instance_path___None)). Otherwise works like [open\_resource()](#open_resource_resource__mode__rb). Instance resources can also be opened for writing.

|  |  |
| --- | --- |
| 参数: | * resource – the name of the resource. To access resources within subfolders use forward slashes as separator. * mode – resource file opening mode, default is ‘rb’. |

open\_resource(resource, mode='rb')

Opens a resource from the application’s resource folder. To see how this works, consider the following folder structure:

/myapplication.py /schema.sql /static /style.css /templates /layout.html /index.html

If you want to open the schema.sql file you would do the following:

with app.open\_resource('schema.sql') as f: contents = f.read() do\_something\_with(contents)

|  |  |
| --- | --- |
| 参数: | * resource – the name of the resource. To access resources within subfolders use forward slashes as separator. * mode – resource file opening mode, default is ‘rb’. |

open\_session(request)

Creates or opens a new session. Default implementation stores all session data in a signed cookie. This requires that the [secret\_key](#secret_key) is set. Instead of overriding this method we recommend replacing the [session\_interface](#session_interface____flask_sessi).

|  |  |
| --- | --- |
| 参数: | request – an instance of [request\_class](#request_class). |

permanent\_session\_lifetime

A [timedelta](http://docs.python.org/dev/library/datetime.html#datetime.timedelta) [http://docs.python.org/dev/library/datetime.html#datetime.timedelta] which is used to set the expiration date of a permanent session. The default is 31 days which makes a permanent session survive for roughly one month.

This attribute can also be configured from the config with the PERMANENT\_SESSION\_LIFETIME configuration key. Defaults to timedelta(days=31)

preprocess\_request()

Called before the actual request dispatching and will call every as [before\_request()](#before_request__args____kwargs) decorated function. If any of these function returns a value it’s handled as if it was the return value from the view and further request handling is stopped.

This also triggers the url\_value\_processor() functions before the actual [before\_request()](#before_request__args____kwargs) functions are called.

preserve\_context\_on\_exception

Returns the value of the PRESERVE\_CONTEXT\_ON\_EXCEPTION configuration value in case it’s set, otherwise a sensible default is returned.

0.7 新版功能.

process\_response(response)

Can be overridden in order to modify the response object before it’s sent to the WSGI server. By default this will call all the [after\_request()](#after_request__args____kwargs) decorated functions.

在 0.5 版更改: As of Flask 0.5 the functions registered for after request execution are called in reverse order of registration.

|  |  |
| --- | --- |
| 参数: | response – a [response\_class](#response_class) object. |
| 返回: | a new response object or the same, has to be an instance of [response\_class](#response_class). |

propagate\_exceptions

Returns the value of the PROPAGATE\_EXCEPTIONS configuration value in case it’s set, otherwise a sensible default is returned.

0.7 新版功能.

register\_blueprint(\*args, \*\*kwargs)

Registers a blueprint on the application.

0.7 新版功能.

register\_error\_handler(code\_or\_exception, f)

Alternative error attach function to the [errorhandler()](#errorhandler__args____kwargs) decorator that is more straightforward to use for non decorator usage.

0.7 新版功能.

register\_module(module, \*\*options)

Registers a module with this application. The keyword argument of this function are the same as the ones for the constructor of the Module class and will override the values of the module if provided.

在 0.7 版更改: The module system was deprecated in favor for the blueprint system.

request\_class

The class that is used for request objects. See [Request](#class_flask_Request_environ__pop) for more information.

[Request](#class_flask_Request_environ__pop) 的别名

request\_context(environ)

Creates a [RequestContext](#class_flask_ctx_RequestContext_a) from the given environment and binds it to the current context. This must be used in combination with the with statement because the request is only bound to the current context for the duration of the with block.

Example usage:

with app.request\_context(environ): do\_something\_with(request)

The object returned can also be used without the with statement which is useful for working in the shell. The example above is doing exactly the same as this code:

ctx = app.request\_context(environ) ctx.push() try: do\_something\_with(request) finally: ctx.pop()

在 0.3 版更改: Added support for non-with statement usage and with statement is now passed the ctx object.

|  |  |
| --- | --- |
| 参数: | environ – a WSGI environment |

response\_class

The class that is used for response objects. See [Response](#class_flask_Response_response_No) for more information.

[Response](#class_flask_Response_response_No) 的别名

route(rule, \*\*options)

A decorator that is used to register a view function for a given URL rule. This does the same thing as [add\_url\_rule()](#add_url_rule__args____kwargs) but is intended for decorator usage:

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

For more information refer to [URL 路由注册](#url_route_registrations).

|  |  |
| --- | --- |
| 参数: | * rule – the URL rule as string * endpoint – the endpoint for the registered URL rule. Flask itself assumes the name of the view function as endpoint * options – the options to be forwarded to the underlying [Rule](http://werkzeug.pocoo.org/docs/routing/#werkzeug.routing.Rule)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. |

run(host=None, port=None, debug=None, \*\*options)

Runs the application on a local development server. If the [debug](#debug) flag is set the server will automatically reload for code changes and show a debugger in case an exception happened.

If you want to run the application in debug mode, but disable the code execution on the interactive debugger, you can pass use\_evalex=False as parameter. This will keep the debugger’s traceback screen active, but disable code execution.

Keep in Mind

Flask will suppress any server error with a generic error page unless it is in debug mode. As such to enable just the interactive debugger without the code reloading, you have to invoke [run()](#run_host_None__port_None__debug) with debug=True and use\_reloader=False. Setting use\_debugger to True without being in debug mode won’t catch any exceptions because there won’t be any to catch.

在 0.10 版更改: The default port is now picked from the SERVER\_NAME variable.

|  |  |
| --- | --- |
| 参数: | * host – the hostname to listen on. Set this to '0.0.0.0' to have the server available externally as well. Defaults to '127.0.0.1'. * port – the port of the webserver. Defaults to 5000 or the port defined in the SERVER\_NAME config variable if present. * debug – if given, enable or disable debug mode. See [debug](#debug). * options – the options to be forwarded to the underlying Werkzeug server. See [werkzeug.serving.run\_simple()](http://werkzeug.pocoo.org/docs/serving/#werkzeug.serving.run_simple) |

save\_session(session, response)

Saves the session if it needs updates. For the default implementation, check [open\_session()](#open_session_request). Instead of overriding this method we recommend replacing the [session\_interface](#session_interface____flask_sessi).

|  |  |
| --- | --- |
| 参数: | * session – the session to be saved (a [SecureCookie](http://werkzeug.pocoo.org/docs/contrib/securecookie/#werkzeug.contrib.securecookie.SecureCookie) * response – an instance of [response\_class](#response_class) |

secret\_key

If a secret key is set, cryptographic components can use this to sign cookies and other things. Set this to a complex random value when you want to use the secure cookie for instance.

This attribute can also be configured from the config with the SECRET\_KEY configuration key. Defaults to None.

select\_jinja\_autoescape(filename)

Returns True if autoescaping should be active for the given template name.

0.5 新版功能.

send\_static\_file(filename)

Function used internally to send static files from the static folder to the browser.

0.5 新版功能.

session\_cookie\_name

The secure cookie uses this for the name of the session cookie.

This attribute can also be configured from the config with the SESSION\_COOKIE\_NAME configuration key. Defaults to 'session'

session\_interface = <flask.sessions.SecureCookieSessionInterface object at 0x3c9c7d0>

the session interface to use. By default an instance of [SecureCookieSessionInterface](#class_flask_sessions_SecureCooki) is used here.

0.8 新版功能.

should\_ignore\_error(error)

This is called to figure out if an error should be ignored or not as far as the teardown system is concerned. If this function returns True then the teardown handlers will not be passed the error.

0.10 新版功能.

teardown\_appcontext(\*args, \*\*kwargs)

Registers a function to be called when the application context ends. These functions are typically also called when the request context is popped.

Example:

ctx = app.app\_context() ctx.push() ... ctx.pop()

When ctx.pop() is executed in the above example, the teardown functions are called just before the app context moves from the stack of active contexts. This becomes relevant if you are using such constructs in tests.

Since a request context typically also manages an application context it would also be called when you pop a request context.

When a teardown function was called because of an exception it will be passed an error object.

0.9 新版功能.

teardown\_appcontext\_funcs = None

A list of functions that are called when the application context is destroyed. Since the application context is also torn down if the request ends this is the place to store code that disconnects from databases.

0.9 新版功能.

teardown\_request(\*args, \*\*kwargs)

Register a function to be run at the end of each request, regardless of whether there was an exception or not. These functions are executed when the request context is popped, even if not an actual request was performed.

Example:

ctx = app.test\_request\_context() ctx.push() ... ctx.pop()

When ctx.pop() is executed in the above example, the teardown functions are called just before the request context moves from the stack of active contexts. This becomes relevant if you are using such constructs in tests.

Generally teardown functions must take every necessary step to avoid that they will fail. If they do execute code that might fail they will have to surround the execution of these code by try/except statements and log occurring errors.

When a teardown function was called because of a exception it will be passed an error object.

Debug Note

In debug mode Flask will not tear down a request on an exception immediately. Instead if will keep it alive so that the interactive debugger can still access it. This behavior can be controlled by the PRESERVE\_CONTEXT\_ON\_EXCEPTION configuration variable.

teardown\_request\_funcs = None

A dictionary with lists of functions that are called after each request, even if an exception has occurred. The key of the dictionary is the name of the blueprint this function is active for, None for all requests. These functions are not allowed to modify the request, and their return values are ignored. If an exception occurred while processing the request, it gets passed to each teardown\_request function. To register a function here, use the [teardown\_request()](#teardown_request__args____kwargs) decorator.

0.7 新版功能.

template\_context\_processors = None

A dictionary with list of functions that are called without argument to populate the template context. The key of the dictionary is the name of the blueprint this function is active for, None for all requests. Each returns a dictionary that the template context is updated with. To register a function here, use the [context\_processor()](#context_processor__args____kwarg) decorator.

template\_filter(\*args, \*\*kwargs)

A decorator that is used to register custom template filter. You can specify a name for the filter, otherwise the function name will be used. Example:

@app.template\_filter() def reverse(s): return s[::-1]

|  |  |
| --- | --- |
| 参数: | name – the optional name of the filter, otherwise the function name will be used. |

template\_global(\*args, \*\*kwargs)

A decorator that is used to register a custom template global function. You can specify a name for the global function, otherwise the function name will be used. Example:

@app.template\_global() def double(n): return 2 \* n

0.10 新版功能.

|  |  |
| --- | --- |
| 参数: | name – the optional name of the global function, otherwise the function name will be used. |

template\_test(\*args, \*\*kwargs)

A decorator that is used to register custom template test. You can specify a name for the test, otherwise the function name will be used. Example:

@app.template\_test() def is\_prime(n): if n == 2: return True for i in range(2, int(math.ceil(math.sqrt(n))) + 1): if n % i == 0: return False return True

0.10 新版功能.

|  |  |
| --- | --- |
| 参数: | name – the optional name of the test, otherwise the function name will be used. |

test\_client(use\_cookies=True)

Creates a test client for this application. For information about unit testing head over to [测试 Flask 应用](#testing).

Note that if you are testing for assertions or exceptions in your application code, you must set app.testing = True in order for the exceptions to propagate to the test client. Otherwise, the exception will be handled by the application (not visible to the test client) and the only indication of an AssertionError or other exception will be a 500 status code response to the test client. See the [testing](#testing_1) attribute. For example:

app.testing = True client = app.test\_client()

The test client can be used in a with block to defer the closing down of the context until the end of the with block. This is useful if you want to access the context locals for testing:

with app.test\_client() as c: rv = c.get('/?vodka=42') assert request.args['vodka'] == '42'

See [FlaskClient](#class_flask_testing_FlaskClient) for more information.

在 0.4 版更改: added support for with block usage for the client.

0.7 新版功能: The use\_cookies parameter was added as well as the ability to override the client to be used by setting the [test\_client\_class](#test_client_class___None) attribute.

test\_client\_class = None

the test client that is used with when test\_client is used.

0.7 新版功能.

test\_request\_context(\*args, \*\*kwargs)

Creates a WSGI environment from the given values (see werkzeug.test.EnvironBuilder() for more information, this function accepts the same arguments).

testing

The testing flag. Set this to True to enable the test mode of Flask extensions (and in the future probably also Flask itself). For example this might activate unittest helpers that have an additional runtime cost which should not be enabled by default.

If this is enabled and PROPAGATE\_EXCEPTIONS is not changed from the default it’s implicitly enabled.

This attribute can also be configured from the config with the TESTING configuration key. Defaults to False.

trap\_http\_exception(e)

Checks if an HTTP exception should be trapped or not. By default this will return False for all exceptions except for a bad request key error if TRAP\_BAD\_REQUEST\_ERRORS is set to True. It also returns True if TRAP\_HTTP\_EXCEPTIONS is set to True.

This is called for all HTTP exceptions raised by a view function. If it returns True for any exception the error handler for this exception is not called and it shows up as regular exception in the traceback. This is helpful for debugging implicitly raised HTTP exceptions.

0.8 新版功能.

update\_template\_context(context)

Update the template context with some commonly used variables. This injects request, session, config and g into the template context as well as everything template context processors want to inject. Note that the as of Flask 0.6, the original values in the context will not be overridden if a context processor decides to return a value with the same key.

|  |  |
| --- | --- |
| 参数: | context – the context as a dictionary that is updated in place to add extra variables. |

url\_build\_error\_handlers = None

A list of functions that are called when [url\_for()](#flask_url_for_endpoint____values) raises a BuildError. Each function registered here is called with error, endpoint and values. If a function returns None or raises a BuildError the next function is tried.

0.9 新版功能.

url\_default\_functions = None

A dictionary with lists of functions that can be used as URL value preprocessors. The key None here is used for application wide callbacks, otherwise the key is the name of the blueprint. Each of these functions has the chance to modify the dictionary of URL values before they are used as the keyword arguments of the view function. For each function registered this one should also provide a [url\_defaults()](#url_defaults__args____kwargs) function that adds the parameters automatically again that were removed that way.

0.7 新版功能.

url\_defaults(\*args, \*\*kwargs)

Callback function for URL defaults for all view functions of the application. It’s called with the endpoint and values and should update the values passed in place.

url\_map = None

The [Map](http://werkzeug.pocoo.org/docs/routing/#werkzeug.routing.Map) [http://werkzeug.pocoo.org/docs/routing/#werkzeug.routing.Map] for this instance. You can use this to change the routing converters after the class was created but before any routes are connected. Example:

from werkzeug.routing import BaseConverter class ListConverter(BaseConverter): def to\_python(self, value): return value.split(',') def to\_url(self, values): return ','.join(BaseConverter.to\_url(value) for value in values) app = Flask(\_\_name\_\_) app.url\_map.converters['list'] = ListConverter

url\_rule\_class

The rule object to use for URL rules created. This is used by [add\_url\_rule()](#add_url_rule__args____kwargs). Defaults to [werkzeug.routing.Rule](http://werkzeug.pocoo.org/docs/routing/#werkzeug.routing.Rule) [http://werkzeug.pocoo.org/docs/routing/#werkzeug.routing.Rule].

0.7 新版功能.

Rule 的别名

url\_value\_preprocessor(\*args, \*\*kwargs)

Registers a function as URL value preprocessor for all view functions of the application. It’s called before the view functions are called and can modify the url values provided.

url\_value\_preprocessors = None

A dictionary with lists of functions that can be used as URL value processor functions. Whenever a URL is built these functions are called to modify the dictionary of values in place. The key None here is used for application wide callbacks, otherwise the key is the name of the blueprint. Each of these functions has the chance to modify the dictionary

0.7 新版功能.

use\_x\_sendfile

Enable this if you want to use the X-Sendfile feature. Keep in mind that the server has to support this. This only affects files sent with the [send\_file()](#flask_send_file_filename_or_fp) method.

0.2 新版功能.

This attribute can also be configured from the config with the USE\_X\_SENDFILE configuration key. Defaults to False.

view\_functions = None

A dictionary of all view functions registered. The keys will be function names which are also used to generate URLs and the values are the function objects themselves. To register a view function, use the [route()](#route_rule____options) decorator.

wsgi\_app(environ, start\_response)

The actual WSGI application. This is not implemented in \_\_call\_\_ so that middlewares can be applied without losing a reference to the class. So instead of doing this:

app = MyMiddleware(app)

It’s a better idea to do this instead:

app.wsgi\_app = MyMiddleware(app.wsgi\_app)

Then you still have the original application object around and can continue to call methods on it.

在 0.7 版更改: The behavior of the before and after request callbacks was changed under error conditions and a new callback was added that will always execute at the end of the request, independent on if an error occurred or not. See [回调和错误](#Hui_Diao_He_Cuo_Wu__Zai__Flask_Z).

|  |  |
| --- | --- |
| 参数: | * environ – a WSGI environment * start\_response – a callable accepting a status code, a list of headers and an optional exception context to start the response |

## 蓝图对象

class flask.Blueprint(name, import\_name, static\_folder=None, static\_url\_path=None, template\_folder=None, url\_prefix=None, subdomain=None, url\_defaults=None)

Represents a blueprint. A blueprint is an object that records functions that will be called with the BlueprintSetupState later to register functions or other things on the main application. See [用蓝图实现模块化的应用](#Yong_Lan_Tu_Shi_Xian_Mo_Kuai_Hua) for more information.

0.7 新版功能.

add\_app\_template\_filter(f, name=None)

Register a custom template filter, available application wide. Like [Flask.add\_template\_filter()](#add_template_filter__args____kwa) but for a blueprint. Works exactly like the [app\_template\_filter()](#app_template_filter_name_None) decorator.

|  |  |
| --- | --- |
| 参数: | name – the optional name of the filter, otherwise the function name will be used. |

add\_app\_template\_global(f, name=None)

Register a custom template global, available application wide. Like [Flask.add\_template\_global()](#add_template_global__args____kwa) but for a blueprint. Works exactly like the [app\_template\_global()](#app_template_global_name_None) decorator.

0.10 新版功能.

|  |  |
| --- | --- |
| 参数: | name – the optional name of the global, otherwise the function name will be used. |

add\_app\_template\_test(f, name=None)

Register a custom template test, available application wide. Like [Flask.add\_template\_test()](#add_template_test__args____kwarg) but for a blueprint. Works exactly like the [app\_template\_test()](#app_template_test_name_None) decorator.

0.10 新版功能.

|  |  |
| --- | --- |
| 参数: | name – the optional name of the test, otherwise the function name will be used. |

add\_url\_rule(rule, endpoint=None, view\_func=None, \*\*options)

Like [Flask.add\_url\_rule()](#add_url_rule__args____kwargs) but for a blueprint. The endpoint for the [url\_for()](#flask_url_for_endpoint____values) function is prefixed with the name of the blueprint.

after\_app\_request(f)

Like [Flask.after\_request()](#after_request__args____kwargs) but for a blueprint. Such a function is executed after each request, even if outside of the blueprint.

after\_request(f)

Like [Flask.after\_request()](#after_request__args____kwargs) but for a blueprint. This function is only executed after each request that is handled by a function of that blueprint.

app\_context\_processor(f)

Like [Flask.context\_processor()](#context_processor__args____kwarg) but for a blueprint. Such a function is executed each request, even if outside of the blueprint.

app\_errorhandler(code)

Like [Flask.errorhandler()](#errorhandler__args____kwargs) but for a blueprint. This handler is used for all requests, even if outside of the blueprint.

app\_template\_filter(name=None)

Register a custom template filter, available application wide. Like [Flask.template\_filter()](#template_filter__args____kwargs) but for a blueprint.

|  |  |
| --- | --- |
| 参数: | name – the optional name of the filter, otherwise the function name will be used. |

app\_template\_global(name=None)

Register a custom template global, available application wide. Like [Flask.template\_global()](#template_global__args____kwargs) but for a blueprint.

0.10 新版功能.

|  |  |
| --- | --- |
| 参数: | name – the optional name of the global, otherwise the function name will be used. |

app\_template\_test(name=None)

Register a custom template test, available application wide. Like [Flask.template\_test()](#template_test__args____kwargs) but for a blueprint.

0.10 新版功能.

|  |  |
| --- | --- |
| 参数: | name – the optional name of the test, otherwise the function name will be used. |

app\_url\_defaults(f)

Same as [url\_defaults()](#url_defaults_f) but application wide.

app\_url\_value\_preprocessor(f)

Same as [url\_value\_preprocessor()](#url_value_preprocessor_f) but application wide.

before\_app\_first\_request(f)

Like [Flask.before\_first\_request()](#before_first_request__args____kw). Such a function is executed before the first request to the application.

before\_app\_request(f)

Like [Flask.before\_request()](#before_request__args____kwargs). Such a function is executed before each request, even if outside of a blueprint.

before\_request(f)

Like [Flask.before\_request()](#before_request__args____kwargs) but for a blueprint. This function is only executed before each request that is handled by a function of that blueprint.

context\_processor(f)

Like [Flask.context\_processor()](#context_processor__args____kwarg) but for a blueprint. This function is only executed for requests handled by a blueprint.

endpoint(endpoint)

Like [Flask.endpoint()](#endpoint__args____kwargs) but for a blueprint. This does not prefix the endpoint with the blueprint name, this has to be done explicitly by the user of this method. If the endpoint is prefixed with a . it will be registered to the current blueprint, otherwise it’s an application independent endpoint.

errorhandler(code\_or\_exception)

Registers an error handler that becomes active for this blueprint only. Please be aware that routing does not happen local to a blueprint so an error handler for 404 usually is not handled by a blueprint unless it is caused inside a view function. Another special case is the 500 internal server error which is always looked up from the application.

Otherwise works as the [errorhandler()](#errorhandler__args____kwargs) decorator of the [Flask](#class_flask_Flask_import_name__s) object.

get\_send\_file\_max\_age(filename)

Provides default cache\_timeout for the [send\_file()](#flask_send_file_filename_or_fp) functions.

By default, this function returns SEND\_FILE\_MAX\_AGE\_DEFAULT from the configuration of [current\_app](#flask_current_app).

Static file functions such as [send\_from\_directory()](#flask_send_from_directory_direct) use this function, and [send\_file()](#flask_send_file_filename_or_fp) calls this function on [current\_app](#flask_current_app) when the given cache\_timeout is None. If a cache\_timeout is given in [send\_file()](#flask_send_file_filename_or_fp), that timeout is used; otherwise, this method is called.

This allows subclasses to change the behavior when sending files based on the filename. For example, to set the cache timeout for .js files to 60 seconds:

class MyFlask(flask.Flask): def get\_send\_file\_max\_age(self, name): if name.lower().endswith('.js'): return 60 return flask.Flask.get\_send\_file\_max\_age(self, name)

0.9 新版功能.

has\_static\_folder

This is True if the package bound object’s container has a folder named 'static'.

0.5 新版功能.

jinja\_loader

The Jinja loader for this package bound object.

0.5 新版功能.

make\_setup\_state(app, options, first\_registration=False)

Creates an instance of [BlueprintSetupState()](#class_flask_blueprints_Blueprint) object that is later passed to the register callback functions. Subclasses can override this to return a subclass of the setup state.

open\_resource(resource, mode='rb')

Opens a resource from the application’s resource folder. To see how this works, consider the following folder structure:

/myapplication.py /schema.sql /static /style.css /templates /layout.html /index.html

If you want to open the schema.sql file you would do the following:

with app.open\_resource('schema.sql') as f: contents = f.read() do\_something\_with(contents)

|  |  |
| --- | --- |
| 参数: | * resource – the name of the resource. To access resources within subfolders use forward slashes as separator. * mode – resource file opening mode, default is ‘rb’. |

record(func)

Registers a function that is called when the blueprint is registered on the application. This function is called with the state as argument as returned by the [make\_setup\_state()](#make_setup_state_app__options__f) method.

record\_once(func)

Works like [record()](#record_func) but wraps the function in another function that will ensure the function is only called once. If the blueprint is registered a second time on the application, the function passed is not called.

register(app, options, first\_registration=False)

Called by [Flask.register\_blueprint()](#register_blueprint__args____kwar) to register a blueprint on the application. This can be overridden to customize the register behavior. Keyword arguments from [register\_blueprint()](#register_blueprint__args____kwar) are directly forwarded to this method in the options dictionary.

route(rule, \*\*options)

Like [Flask.route()](#route_rule____options) but for a blueprint. The endpoint for the [url\_for()](#flask_url_for_endpoint____values) function is prefixed with the name of the blueprint.

send\_static\_file(filename)

Function used internally to send static files from the static folder to the browser.

0.5 新版功能.

teardown\_app\_request(f)

Like [Flask.teardown\_request()](#teardown_request__args____kwargs) but for a blueprint. Such a function is executed when tearing down each request, even if outside of the blueprint.

teardown\_request(f)

Like [Flask.teardown\_request()](#teardown_request__args____kwargs) but for a blueprint. This function is only executed when tearing down requests handled by a function of that blueprint. Teardown request functions are executed when the request context is popped, even when no actual request was performed.

url\_defaults(f)

Callback function for URL defaults for this blueprint. It’s called with the endpoint and values and should update the values passed in place.

url\_value\_preprocessor(f)

Registers a function as URL value preprocessor for this blueprint. It’s called before the view functions are called and can modify the url values provided.

## 进入的请求对象

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](#class_flask_request). If you want to replace the request object used you can subclass this and set [request\_class](#request_class) to your subclass.

The request object is a [Request](http://werkzeug.pocoo.org/docs/wrappers/#werkzeug.wrappers.Request) [http://werkzeug.pocoo.org/docs/wrappers/#werkzeug.wrappers.Request] subclass and provides all of the attributes Werkzeug defines plus a few Flask specific ones.

form

一个包含解析过的从 POST 或 PUT 请求发送的表单对象的 [MultiDict](http://werkzeug.pocoo.org/docs/datastructures/#werkzeug.datastructures.MultiDict) [http://werkzeug.pocoo.org/docs/datastructures/#werkzeug.datastructures.MultiDict] 。请注意上传的文件不会在这里，而是在 [files](#files) 属性中。

args

一个包含解析过的查询字符串（ URL 中问号后的部分）内容的 [MultiDict](http://werkzeug.pocoo.org/docs/datastructures/#werkzeug.datastructures.MultiDict) [http://werkzeug.pocoo.org/docs/datastructures/#werkzeug.datastructures.MultiDict] 。

values

一个包含 [form](#form) 和 [args](#args) 全部内容的 [CombinedMultiDict](http://werkzeug.pocoo.org/docs/datastructures/#werkzeug.datastructures.CombinedMultiDict) [http://werkzeug.pocoo.org/docs/datastructures/#werkzeug.datastructures.CombinedMultiDict] 。

cookies

一个包含请求中传送的所有 cookie 内容的 [dict](http://docs.python.org/dev/library/stdtypes.html#dict) [http://docs.python.org/dev/library/stdtypes.html#dict] 。

stream

如果表单提交的数据没有以已知的 mimetype 编码，为性能考虑，数据会不经 修改存储在这个流中。大多数情况下，使用可以把数据提供为字符串的 [data](#data) 是更好的方法。流只返回一次数据。

headers

进入请求的标头存为一个类似字典的对象。

data

如果进入的请求数据是 Flask 不能处理的 mimetype ，数据将作为字符串 存于此。

files

一个包含 POST 和 PUT 请求中上传的文件的 [MultiDict](http://werkzeug.pocoo.org/docs/datastructures/#werkzeug.datastructures.MultiDict) [http://werkzeug.pocoo.org/docs/datastructures/#werkzeug.datastructures.MultiDict] 。每个文件存储为 [FileStorage](http://werkzeug.pocoo.org/docs/datastructures/#werkzeug.datastructures.FileStorage) [http://werkzeug.pocoo.org/docs/datastructures/#werkzeug.datastructures.FileStorage] 对象。其基本的行为类似你在 Python 中 见到的标准文件对象，差异在于这个对象有一个 [save()](http://werkzeug.pocoo.org/docs/datastructures/#werkzeug.datastructures.FileStorage.save) [http://werkzeug.pocoo.org/docs/datastructures/#werkzeug.datastructures.FileStorage.save] 方法可以把文件存储到文件系统上。

environ

底层的 WSGI 环境。

method

当前请求的 HTTP 方法 (POST ， GET 等等)

path

script\_root

url

base\_url

url\_root

提供不同的方式来审视当前的 URL 。想象你的应用监听下面的 URL:

http://www.example.com/myapplication

并且用户请求下面的 URL:

http://www.example.com/myapplication/page.html?x=y

这个情况下，上面提到的属性的值会为如下:

|  |  |
| --- | --- |
| path | /page.html |
| script\_root | /myapplication |
| base\_url | http://www.example.com/myapplication/page.html |
| url | http://www.example.com/myapplication/page.html?x=y |
| url\_root | http://www.example.com/myapplication/ |

is\_xhr

当请求由 JavaScript 的 XMLHttpRequest 触发时，该值为 True 。 这只对支持 X-Requested-With 标头并把该标头设置为 XMLHttpRequest 的库奏效。这么做的库有 prototype 、 jQuery 以及 Mochikit 等更多。

blueprint

The name of the current blueprint

endpoint

The endpoint that matched the request. This in combination with [view\_args](#view_args___None) can be used to reconstruct the same or a modified URL. If an exception happened when matching, this will be None.

get\_json(force=False, silent=False, cache=True)

Parses the incoming JSON request data and returns it. If parsing fails the [on\_json\_loading\_failed()](#on_json_loading_failed_e) method on the request object will be invoked. By default this function will only load the json data if the mimetype is application/json but this can be overriden by the force parameter.

|  |  |
| --- | --- |
| 参数: | * force – if set to True the mimetype is ignored. * silent – if set to False this method will fail silently and return False. * cache – if set to True the parsed JSON data is remembered on the request. |

json

If the mimetype is application/json this will contain the parsed JSON data. Otherwise this will be None.

The [get\_json()](#get_json_force_False__silent_Fal) method should be used instead.

max\_content\_length

Read-only view of the MAX\_CONTENT\_LENGTH config key.

module

The name of the current module if the request was dispatched to an actual module. This is deprecated functionality, use blueprints instead.

on\_json\_loading\_failed(e)

Called if decoding of the JSON data failed. The return value of this method is used by [get\_json()](#get_json_force_False__silent_Fal) when an error occurred. The default implementation just raises a BadRequest exception.

在 0.10 版更改: Removed buggy previous behavior of generating a random JSON response. If you want that behavior back you can trivially add it by subclassing.

0.8 新版功能.

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](http://werkzeug.pocoo.org/docs/exceptions/#werkzeug.exceptions.NotFound) [http://werkzeug.pocoo.org/docs/exceptions/#werkzeug.exceptions.NotFound] exception or something similar.

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 a before/after handler (request.url\_rule.methods) etc.

0.6 新版功能.

view\_args = None

a dict of view arguments that matched the request. If an exception happened when matching, this will be None.

class flask.request

你可以使用全局 request 对象访问进入的请求数据。 Flask 处理进入的请求 数据并允许你用这个全局对象访问它。如果你工作在多线程环境，Flask 内部保证 你总会在当前线程上获取正确的数据，

这是一个代理。详情见 [留意代理](#Liu_Yi_Dai_Li__Flask_Zhong_Ti_Go) 。

请求对象是一个 [Request](http://werkzeug.pocoo.org/docs/wrappers/#werkzeug.wrappers.Request) [http://werkzeug.pocoo.org/docs/wrappers/#werkzeug.wrappers.Request] 子类的实例，提供所有 Werkzeug 定义的属性。这里只对最重要的展示了简要概述。

## 响应对象

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()](#make_response_rv) will take care of that for you.

If you want to replace the response object used you can subclass this and set [response\_class](#response_class) to your subclass.

headers

Headers 对象表示响应的标头。

status

字符串表示的响应状态。

status\_code

整数表示的响应状态。

data

A descriptor that calls get\_data() and set\_data(). This should not be used and will eventually get deprecated.

mimetype

The mimetype (content type without charset etc.)

set\_cookie(key, value='', max\_age=None, expires=None, path='/', domain=None, secure=None, httponly=False)

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.

|  |  |
| --- | --- |
| 参数: | * 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. * 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. * path – limits the cookie to a given path, per default it will span the whole domain. |

## 会话

如果你设置了 [Flask.secret\_key](#secret_key) ，你可以在 Flask 应用中使用会话。会话 主要使得在请求见保留信息成为可能。 Flask 的实现方法是使用一个签名的 cookie 。 这样，用户可以查看会话的内容，但是不能修改它，除非用户知道密钥。所以确保密钥 被设置为一个复杂且无法被容易猜测的值。

你可以使用 [session](#class_flask_session) 对象来访问当前的会话:

class flask.session

会话对象很像通常的字典，区别是会话对象会追踪修改。

这是一个代理。更多信息见 [留意代理](#Liu_Yi_Dai_Li__Flask_Zhong_Ti_Go) 。

下列属性是需要关注的:

new

如果会话是新的，该值为 True ，否则为 False 。

modified

当果会话对象检测到修改，这个值为 True 。注意可变结构的修改不会 被自动捕获，这种情况下你需要自行显式地设置这个属性为 True 。这 里有 一个例子:

# 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

如果设为 True ，会话存活 [permanent\_session\_lifetime](#permanent_session_lifetime) 秒。默认为 31 天。 如果是 False （默认选项），会话会在用户关闭浏览器时删除。

## 会话接口

0.8 新版功能.

会话接口提供了简单的途径来替换 Flask 正在使用的会话实现。

class flask.sessions.SessionInterface

The basic interface you have to implement in order to replace the default session interface which uses werkzeug’s securecookie implementation. The only methods you have to implement are [open\_session()](#open_session_app__request) and [save\_session()](#save_session_app__session__respo), the others have useful defaults which you don’t need to change.

The session object returned by the [open\_session()](#open_session_app__request) method has to provide a dictionary like interface plus the properties and methods from the [SessionMixin](#class_flask_sessions_SessionMixi). We recommend just subclassing a dict and adding that mixin:

class Session(dict, SessionMixin): pass

If [open\_session()](#open_session_app__request) returns None Flask will call into [make\_null\_session()](#make_null_session_app) to create a session that acts as replacement if the session support cannot work because some requirement is not fulfilled. The default [NullSession](#class_flask_sessions_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](#session_interface____flask_sessi):

app = Flask(\_\_name\_\_) app.session\_interface = MySessionInterface()

0.8 新版功能.

get\_cookie\_domain(app)

Helpful helper method that returns the cookie domain that should be used for the session cookie if session cookies are used.

get\_cookie\_httponly(app)

Returns True if the session cookie should be httponly. This currently just returns the value of the SESSION\_COOKIE\_HTTPONLY config var.

get\_cookie\_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.

get\_cookie\_secure(app)

Returns True if the cookie should be secure. This currently just returns the value of the SESSION\_COOKIE\_SECURE setting.

get\_expiration\_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.

is\_null\_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](#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](#null_session_class) by default.

null\_session\_class

[make\_null\_session()](#make_null_session_app) will look here for the class that should be created when a null session is requested. Likewise the [is\_null\_session()](#is_null_session_obj) method will perform a typecheck against this type.

[NullSession](#class_flask_sessions_NullSession) 的别名

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](#class_flask_sessions_SessionMixi).

pickle\_based = False

A flag that indicates if the session interface is pickle based. This can be used by flask extensions to make a decision in regards to how to deal with the session object.

0.10 新版功能.

save\_session(app, session, response)

This is called for actual sessions returned by [open\_session()](#open_session_app__request) 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.

class flask.sessions.SecureCookieSessionInterface

The default session interface that stores sessions in signed cookies through the itsdangerous module.

digest\_method()

the hash function to use for the signature. The default is sha1

key\_derivation = 'hmac'

the name of the itsdangerous supported key derivation. The default is hmac.

salt = 'cookie-session'

the salt that should be applied on top of the secret key for the signing of cookie based sessions.

serializer = <flask.sessions.TaggedJSONSerializer object at 0x3c9c390>

A python serializer for the payload. The default is a compact JSON derived serializer with support for some extra Python types such as datetime objects or tuples.

session\_class

SecureCookieSession 的别名

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 an accessors that are expected by Flask extensions and users for the session.

modified = True

for some backends this will always be True, but some backends will default this to false and detect changes in the dictionary for as long as changes do not happen on mutable structures in the session. The default mixin implementation just hardcodes True in.

new = False

some session backends can tell you if a session is new, but that is not necessarily guaranteed. Use with caution. The default mixin implementation just hardcodes False in.

permanent

this reflects the '\_permanent' key in the dict.

flask.sessions.session\_json\_serializer

Notice

PERMANENT\_SESSION\_LIFETIME 配置键从 Flask 0.8 开始可以是一个整数。 你可以自己计算值，或用应用上的 [permanent\_session\_lifetime](#permanent_session_lifetime) 属性来自动转换结果为 一个整数。

## 测试客户端

class flask.testing.FlaskClient(application, response\_wrapper=None, use\_cookies=True, allow\_subdomain\_redirects=False)

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](http://werkzeug.pocoo.org/docs/test/#werkzeug.test.Client) [http://werkzeug.pocoo.org/docs/test/#werkzeug.test.Client].

Basic usage is outlined in the [测试 Flask 应用](#testing) chapter.

session\_transaction(\*args, \*\*kwds)

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 test request context and since session handling could depend on request variables this function accepts the same arguments as [test\_request\_context()](#test_request_context__args____kw) which are directly passed through.

## 应用全局变量

只在一个请求内，从一个函数到另一个函数共享数据，全局变量并不够好。因为这 在线程环境下行不通。 Flask 提供了一个特殊的对象来确保只在活动的请求中 有效，并且每个请求都返回不同的值。一言蔽之：它做正确的事情，如同它对 [request](#class_flask_request) 和 [session](#class_flask_session) 做的那样。

flask.g

在这上存储你任何你想要存储的。例如一个数据库连接或者当前登入的用户。

从 Flask 0.10 起，对象 g 存储在应用上下文中而不再是请求上下文中，这 意味着即使在应用上下文中它也是可访问的而不是只能在请求上下文中。在 结合 [伪造资源和上下文](#Wei_Zao_Zi_Yuan_He_Shang_Xia_Wen) 模式使用来测试时这尤为有用。

另外，在 0.10 中你可以使用 get() 方法来获取一个属性或者如果这 个属性没设置的话将得到 None (或者第二个参数)。 这两种用法现在是没有区别的:

user = getattr(flask.g, 'user', None) user = flask.get.get('user', None)

现在也能在 g 对象上使用 in 运算符来确定它是否有某个属性，并且它 将使用 yield 关键字来生成这样一个可迭代的包含所有keys的生成器。

这是一个代理。详情见 [留意代理](#Liu_Yi_Dai_Li__Flask_Zhong_Ti_Go) 。

## 有用的函数和类

flask.current\_app

指向正在处理请求的应用。这对于想要支持同时运行多个应用的扩展有用。 它由应用上下文驱动，而不是请求上下文，所以你可以用 [app\_context()](#app_context) 方法 修改这个代理的值。

这是一个代理。详情见 [留意代理](#Liu_Yi_Dai_Li__Flask_Zhong_Ti_Go) 。

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](#class_flask_request) or [g](#flask_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

0.7 新版功能.

flask.has\_app\_context()

Works like [has\_request\_context()](#flask_has_request_context) but for the application context. You can also just do a boolean check on the [current\_app](#flask_current_app) object instead.

0.9 新版功能.

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](#url_building).

To integrate applications, [Flask](#class_flask_Flask_import_name__s) has a hook to intercept URL build errors through Flask.build\_error\_handler. 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](#flask_current_app) calls its build\_error\_handler 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.build\_error\_handler = 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 NotFound errors.

0.10 新版功能: The \_scheme parameter was added.

0.9 新版功能: The \_anchor and \_method parameters were added.

0.9 新版功能: Calls Flask.handle\_build\_error() on BuildError.

|  |  |
| --- | --- |
| 参数: | * 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 SERVER\_NAME configuration variable which defaults to localhost. * \_scheme – a string specifying the desired URL scheme. The \_external parameter must be set to True or a ValueError is raised. * \_anchor – if provided this is added as anchor to the URL. * \_method – if provided this explicitly specifies an HTTP method. |

flask.abort(code)

抛出一个给定状态代码的 [HTTPException](http://werkzeug.pocoo.org/docs/exceptions/#werkzeug.exceptions.HTTPException) [http://werkzeug.pocoo.org/docs/exceptions/#werkzeug.exceptions.HTTPException] 。 例如想要用一个页面未找到异常来终止请求，你可以调用 abort(404) 。

|  |  |
| --- | --- |
| 参数: | code – the HTTP error code. |

flask.redirect(location, code=302)

Return a response object (a WSGI application) that, if called, redirects the client to the target location. Supported codes are 301, 302, 303, 305, and 307. 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.

0.6 新版功能: The location can now be a unicode string that is encoded using the iri\_to\_uri() function.

|  |  |
| --- | --- |
| 参数: | * location – the location the response should redirect to. * code – the redirect status code. defaults to 302. |

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 error code:

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 view decorators:

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()](#make_response_rv) is invoked with it.
* if more than one argument is passed, the arguments are passed to the [flask.Flask.make\_response()](#make_response_rv) function as tuple.

0.6 新版功能.

flask.send\_file(filename\_or\_fp, mimetype=None, as\_attachment=False, attachment\_filename=None, add\_etags=True, cache\_timeout=None, conditional=False)

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](#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.

Please never pass filenames to this function from user sources without checking them first. Something like this is usually sufficient to avoid security problems:

if '..' in filename or filename.startswith('/'): abort(404)

0.2 新版功能.

0.5 新版功能: The add\_etags, cache\_timeout and conditional parameters were added. The default behavior is now to attach etags.

在 0.7 版更改: 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

在 0.9 版更改: cache\_timeout pulls its default from application config, when None.

|  |  |
| --- | --- |
| 参数: | * 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 fall back to the traditional method. Make sure that the file pointer is positioned at the start of data to send before calling [send\_file()](#flask_send_file_filename_or_fp). * mimetype – the mimetype of the file if provided, otherwise auto detection happens. * 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()](#get_send_file_max_age_filename) of [current\_app](#flask_current_app). |

flask.send\_from\_directory(directory, filename, \*\*options)

Send a file from a given directory with [send\_file()](#flask_send_file_filename_or_fp). 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.

0.5 新版功能.

|  |  |
| --- | --- |
| 参数: | * 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_send_file_filename_or_fp). |

flask.safe\_join(directory, filename)

Safely join directory and filename.

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...

|  |  |
| --- | --- |
| 参数: | * directory – the base directory. * filename – the untrusted filename relative to that directory. |
| Raises: | NotFound if the resulting path would fall out of directory. |

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

Marks a string as being safe for inclusion in HTML/XML output without needing to be escaped. This implements the \_\_html\_\_ interface a couple of frameworks and web applications use. [Markup](#class_flask_Markup) is a direct subclass of unicode and provides all the methods of unicode just that it escapes arguments passed and always returns Markup.

The escape function returns markup objects so that double escaping can’t happen.

The constructor of the [Markup](#class_flask_Markup) class can be used for three different things: When passed an unicode object it’s assumed to be safe, when passed an object with an HTML representation (has an \_\_html\_\_ method) that representation is used, otherwise the object passed is converted into a unicode string and then assumed to be safe:

>>> Markup("Hello <em>World</em>!") Markup(u'Hello <em>World</em>!') >>> class Foo(object): ... def \_\_html\_\_(self): ... return '<a href="#">foo</a>' ... >>> Markup(Foo()) Markup(u'<a href="#">foo</a>')

If you want object passed being always treated as unsafe you can use the [escape()](#flask_escape_s____markup) classmethod to create a [Markup](#class_flask_Markup) object:

>>> Markup.escape("Hello <em>World</em>!") Markup(u'Hello &lt;em&gt;World&lt;/em&gt;!')

Operations on a markup string are markup aware which means that all arguments are passed through the [escape()](#flask_escape_s____markup) function:

>>> em = Markup("<em>%s</em>") >>> em % "foo & bar" Markup(u'<em>foo &amp; bar</em>') >>> strong = Markup("<strong>%(text)s</strong>") >>> strong % {'text': '<blink>hacker here</blink>'} Markup(u'<strong>&lt;blink&gt;hacker here&lt;/blink&gt;</strong>') >>> Markup("<em>Hello</em> ") + "<foo>" Markup(u'<em>Hello</em> &lt;foo&gt;')

classmethod escape(s)

Escape the string. Works like [escape()](#flask_escape_s____markup) with the difference that for subclasses of [Markup](#class_flask_Markup) this function would return the correct subclass.

striptags()

Unescape markup into an text\_type string and strip all tags. This also resolves known HTML4 and XHTML entities. Whitespace is normalized to one:

>>> Markup("Main &raquo; <em>About</em>").striptags() u'Main \xbb About'

unescape()

Unescape markup again into an text\_type string. This also resolves known HTML4 and XHTML entities:

>>> Markup("Main &raquo; <em>About</em>").unescape() u'Main \xbb <em>About</em>'

## 消息闪现

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()](#flask_get_flashed_messages_with).

在 0.3 版更改: category parameter added.

|  |  |
| --- | --- |
| 参数: | * 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](#Xiao_Xi_Shan_Xian__Hao_De_Ying_Y) for examples.

在 0.3 版更改: with\_categories parameter added.

在 0.9 版更改: category\_filter parameter added.

|  |  |
| --- | --- |
| 参数: | * with\_categories – set to True to also receive categories. * category\_filter – whitelist of categories to limit return values |

## JSON 支持

Flask 使用 simplejson 来实现JSON。自从 simplejson 既在标准库中提供也在 Flask 的拓展中提供。Flask将首先尝试自带的simplejson，如果失败了就使用标准 库中的json模块。除此之外，为了更容易定制它还会委托访问当前应用的JSON的编码 器和解码器。

所以首先不要这样用：

try:

import simplejson as json

except ImportError:

import json

你可以这样

from flask import json

For usage examples, read the [json](http://docs.python.org/dev/library/json.html#module-json) [http://docs.python.org/dev/library/json.html#module-json] documentation. 关于更多的用法，请阅读标准库中的 [json](http://docs.python.org/dev/library/json.html#module-json) [http://docs.python.org/dev/library/json.html#module-json] 文档。下面的拓展已经默认被集成 到了标准库中JSON模块里：

1. datetime 对象被序列化为 [RFC 822](http://tools.ietf.org/html/rfc822.html) [http://tools.ietf.org/html/rfc822.html] 字符串。
2. 任何带有 \_\_html\_\_ 方法(比如 [Markup](#class_flask_Markup))将在序列化的时候 调用这个方法然后返回的字符串将会被序列化为字符串。

这个 htmlsafe\_dumps() 方法也能在 Jinja2 的过滤器中使用，名字为 |tojson 。请注意在 script 标签内部的内容将不会被转义，所以如果你想在 script 内部使用的话请确保它是不可用的通过 |safe 来转义，除非你正在 使用 Flask 0.10，如下：

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

flask.json.jsonify(\*args, \*\*kwargs)

Creates a [Response](#class_flask_Response_response_No) with the JSON representation of the given arguments with an application/json mimetype. The arguments to this function are the same as to the [dict](http://docs.python.org/dev/library/stdtypes.html#dict) [http://docs.python.org/dev/library/stdtypes.html#dict] constructor.

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 }

For security reasons only objects are supported toplevel. For more information about this, have a look at [JSON 安全](#json_security).

This function’s response will be pretty printed if it was not requested with X-Requested-With: XMLHttpRequest to simplify debugging unless the JSONIFY\_PRETTYPRINT\_REGULAR config parameter is set to false.

0.2 新版功能.

flask.json.dumps(obj, \*\*kwargs)

Serialize obj to a JSON formatted str by using the application’s configured encoder ([json\_encoder](#json_encoder)) if there is an application on the stack.

This function can return unicode strings or ascii-only bytestrings by default which coerce into unicode strings automatically. That behavior by default is controlled by the JSON\_AS\_ASCII configuration variable and can be overriden by the simplejson ensure\_ascii parameter.

flask.json.dump(obj, fp, \*\*kwargs)

Like [dumps()](#flask_json_dumps_obj____kwargs) but writes into a file object.

flask.json.loads(s, \*\*kwargs)

Unserialize a JSON object from a string s by using the application’s configured decoder ([json\_decoder](#json_decoder)) if there is an application on the stack.

flask.json.load(fp, \*\*kwargs)

Like [loads()](#flask_json_loads_s____kwargs) 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, encoding='utf-8', default=None, use\_decimal=True, namedtuple\_as\_object=True, tuple\_as\_array=True)

The default Flask JSON encoder. This one extends the default simplejson encoder by also supporting datetime objects, UUID as well as Markup objects which are serialized as RFC 822 datetime strings (same as the HTTP date format). In order to support more data types override the [default()](#default_o) 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(encoding=None, 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 encoder. Consult the [json](http://docs.python.org/dev/library/json.html#module-json) [http://docs.python.org/dev/library/json.html#module-json] documentation for more information. This decoder is not only used for the load functions of this module but also [Request](#class_flask_Request_environ__pop).

## 模板渲染

flask.render\_template(template\_name\_or\_list, \*\*context)

Renders a template from the template folder with the given context.

|  |  |
| --- | --- |
| 参数: | * 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.render\_template\_string(source, \*\*context)

Renders a template from the given template source string with the given context.

|  |  |
| --- | --- |
| 参数: | * source – the sourcecode of the template to be rendered * context – the variables that should be available in the context of the template. |

flask.get\_template\_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')

0.2 新版功能.

|  |  |
| --- | --- |
| 参数: | * template\_name – the name of the template * attribute – the name of the variable of macro to access |

## 配置

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()](#from_object_obj) 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 X use the export statement:

export YOURAPPLICATION\_SETTINGS='/path/to/config/file'

On windows use set instead.

|  |  |
| --- | --- |
| 参数: | * 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 |

from\_envvar(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'])

|  |  |
| --- | --- |
| 参数: | * variable\_name – name of the environment variable * silent – set to True if you want silent failure for missing files. |
| 返回: | bool. True if able to load config, False otherwise. |

from\_object(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.

Just the uppercase variables in that object are stored in the config. Example usage:

app.config.from\_object('yourapplication.default\_config') from yourapplication import default\_config app.config.from\_object(default\_config)

You should not use this function to load the actual configuration but rather configuration defaults. The actual config should be loaded with [from\_pyfile()](#from_pyfile_filename__silent_Fal) and ideally from a location not within the package because the package might be installed system wide.

|  |  |
| --- | --- |
| 参数: | obj – an import name or object |

from\_pyfile(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()](#from_object_obj) function.

|  |  |
| --- | --- |
| 参数: | * 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. |

0.7 新版功能: silent parameter.

## 扩展

flask.ext

这个模块重定向导入模块到 Flask 扩展。它在 0.8 中被加入，作为导入 Flask 扩展的权威方式，并使得我们在发布扩展时能有更大的灵活性。

如果你想使用名为 “Flask-Foo” 的扩展，你应按照下述从 [ext](#flask_ext) 导入:

from flask.ext import foo

0.8 新版功能.

## 流的辅助函数

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()))

0.9 新版功能.

## 有用的内构件

class flask.ctx.RequestContext(app, environ, request=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()](#test_request_context__args____kw) and [request\_context()](#request_context_environ) 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()](#teardown_request__args____kwargs)).

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()](#test_client_use_cookies_True) 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.

0.10 新版功能.

match\_request()

Can be overridden by a subclass to hook into the matching of the request.

pop(exc=None)

Pops the request context and unbinds it by doing that. This will also trigger the execution of functions registered by the [teardown\_request()](#teardown_request__args____kwargs) decorator.

在 0.9 版更改: Added the exc argument.

push()

Binds the request context to the current context.

flask.\_request\_ctx\_stack

Flask 中使用的所有的上下文局部对象，都由内部的 [LocalStack](http://werkzeug.pocoo.org/docs/local/#werkzeug.local.LocalStack) [http://werkzeug.pocoo.org/docs/local/#werkzeug.local.LocalStack] 实现。这是一个带文档的实例，并且可以 在扩展和应用的代码中使用，但一般来说是不推荐这样使用的。

下面的属性在栈的每层上都存在:

app

活动的 Flask 应用

url\_adapter

用于匹配请求的 URL 适配器

request

当前的请求对象

session

当前的会话对象

g

拥有 [flask.g](#flask_g) 对象上全部属性的对象

flashes

闪现消息的内部缓存

用法示例:

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 implicitly to the current thread or greenlet, similar to how the RequestContext binds request information. The application context is also implicitly created if a request context is created but the application is not on top of the individual application context.

pop(exc=None)

Pops the app context.

push()

Binds the app context to the current context.

flask.\_app\_ctx\_stack

类似请求上下文，但是只跟应用绑定。主要为扩展提供数据存储。

0.9 新版功能.

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()](#make_setup_state_app__options__f) method and later passed to all register callback functions.

add\_url\_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()](#register_blueprint__args____kwar) 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.

url\_prefix = None

The prefix that should be used for all URLs defined on the blueprint.

## 信号

0.6 新版功能.

flask.signals\_available

当信号系统可用时为 True ，即在 [blinker](http://pypi.python.org/pypi/blinker) [http://pypi.python.org/pypi/blinker] 已经被安装的情况下。

flask.template\_rendered

当一个模板成功渲染的时候，这个信号会发出。这个信号带着一个模板实例 template 和为一个字典的上下文（叫 context ）两个参数被调用。

flask.request\_started

这个信号在处建立请求上下文之外的任何请求处理开始前发送。因为请求上下文 这个信号在任何对请求的处理前发送，但是正好是在请求的上下文被建立的时候。 因为请求上下文已经被约束了，用户可以使用 [request](#class_flask_request) 之类的标 准全局代理访问请求对象。

flask.request\_finished

这个信号恰好在请求发送给客户端之前发送。它传递名为 response 的将被发送 的响应。

flask.got\_request\_exception

这个信号在请求处理中抛出异常时发送。它在标准异常处理生效 之前 ，甚至是 在不会处理异常的调试模式下也是如此。这个异常会被将作为一个 exception 传递到用户那。

flask.request\_tearing\_down

这个信号在请求销毁时发送。它总会被调用，即使发生异常。在这种清况下，造 成teardown的异常将会通过一个叫 exc 的关键字参数传递出来。

在 0.9 版更改: 添加了 exc 参数

flask.appcontext\_tearing\_down

这个信号在应用上下文销毁时发送。它总会被调用，即使发生异常。在这种清况 下，造成teardown的异常将会通过一个叫 exc 的关键字参数传递出来。发送 者是application对象。

flask.appcontext\_pushed

当应用上下文被压入栈后会发送这个信号。发送者是application对象

0.10 新版功能.

flask.appcontext\_popped

当应用上下文出栈后会发送这个信号。发送者是application对象。这常常与 [appcontext\_tearing\_down](#flask_appcontext_tearing_down) 这个信号一致。

0.10 新版功能.

flask.message\_flashed

This signal is sent when the application is flashing a message. The messages is sent as message keyword argument and the 当闪现一个消息时会发送这个信号。消息的内容将以 message 关键字参数 发送，而消息的种类则是 category 关键字参数。

0.10 新版功能.

class flask.signals.Namespace

[blinker.base.Namespace](http://discorporate.us/projects/Blinker/docs/1.1/api.html#blinker.base.Namespace) [http://discorporate.us/projects/Blinker/docs/1.1/api.html#blinker.base.Namespace] 的别名，如果 blinker 可用的话。否则， 是一个发送伪信号的伪造的类。这个类对想提供与 Flask 相同的备用系统的 Flask扩展有用。

signal(name, doc=None)

在此命名空间中创建一个新信号，如果 blinker 可用的话。否则返回一个 带有不做任何事的发送方法，任何操作都会（包括连接）报错为 [RuntimeError](http://docs.python.org/dev/library/exceptions.html#RuntimeError) [http://docs.python.org/dev/library/exceptions.html#RuntimeError] 的伪信号。

## 基于类的视图

0.7 新版功能.

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()](#add_url_rule__args____kwargs) 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 as\_view(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.

0.8 新版功能.

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 for which methods this pluggable view can handle.

class flask.views.MethodView

Like a regular class-based view but that dispatches requests to particular methods. For instance if you implement a method called get() it means you will response to 'GET' requests and the dispatch\_request() implementation will automatically forward your request to that. Also options is set for you automatically:

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'))

## URL 路由注册

在路由系统中定义规则可以的方法可以概括为三种:

1. 使用 [flask.Flask.route()](#route_rule____options) 装饰器
2. 使用 [flask.Flask.add\_url\_rule()](#add_url_rule__args____kwargs) 函数
3. 直接访问暴露为 [flask.Flask.url\_map](#url_map___None) 的底层的 Werkzeug 路由系统

路由中的变量部分可以用尖括号指定（ /user/<username>）。默认情况下，URL 中的变量部分接受任何不带斜线的字符串，而 <converter:name> 也可以指定不 同的转换器。

变量部分以关键字参数传递给视图函数。

下面的转换器是可用的:

|  |  |
| --- | --- |
| string | 接受任何不带斜线的字符串（默认的转换器） |
| int | 接受整数 |
| float | 同 int ，但是接受浮点数 |
| path | 和默认的相似，但也接受斜线 |

这里是一些例子:

@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

需要注意的一个重要细节是 Flask 处理结尾斜线的方式。你可以应用下面两个 规则来保证 URL 的唯一:

1. 如果规则以斜线结尾，当用户以不带斜线的形式请求，用户被自动重定向到 带有结尾斜线的相同页面。
2. 如果规则结尾没有斜线，当用户以带斜线的形式请求，会抛出一个 404 not found 。

这与 web 服务器处理静态文件的方式一致。这使得安全地使用相对链接地址成为 可能。

你可以为同一个函数定义多个规则。无论如何，他们也要唯一。也可以给定默认值。 这里给出一个接受可选页面的 URL 定义:

@app.route('/users/', defaults={'page': 1}) @app.route('/users/page/<int:page>') def show\_users(page): pass

这指定了 /users/ 为第一页的 URL ，/users/page/N 为第 N 页的 URL 。

以下是 [route()](#route_rule____options) 和 [add\_url\_rule()](#add_url_rule__args____kwargs) 接受的参数。两者唯一的区别是，带有路由参数的视图函数用装饰器定义，而不是 view\_func 参数。

|  |  |
| --- | --- |
| rule | URL 规则的字符串 |
| endpoint | 注册的 URL 规则的末端。如果没有显式地规定，Flask 本身假设 末端的名称是视图函数的名称，。 |
| view\_func | 当请求呈递到给定的末端时调用的函数。如果没有提供，可以 在用在 [view\_functions](#view_functions___None) 字典中以末端 作为键名存储，来在之后设定函数。 |
| defaults | 规则默认值的字典。上面的示例介绍了默认值如何工作。 |
| subdomain | 当使用子域名匹配的时候，为子域名设定规则。如果没有给定，假 定为默认的子域名。 |
| \*\*options | 这些选项会被推送给底层的 [Rule](http://werkzeug.pocoo.org/docs/routing/#werkzeug.routing.Rule)GET ， POST 等等）。默认情 况下，规则只监听 GET （也隐式地监听 HEAD ）。从 Flask 0.6 开始，OPTIONS 也被隐式地加入，并且做标准的请求处理。 它们需要作为关键字参数来给定。 |

## 视图函数选项

对内部使用，视图函数可以有一些属性，附加到视图函数通常没有控制权的自定义的 行为。下面的可选属性覆盖 [add\_url\_rule()](#add_url_rule__args____kwargs) 的默认值或一般 行为:

* \_\_name\_\_: 函数的名称默认用于末端。如果显式地提供末端，这个值会使用。 此外，它默认以蓝图的名称作为前缀，并且不能从函数本身自定义。
* methods: 如果没有在添加 URL 规则时提供 methods 参数。 Flask 会在视 图函数对象上寻找是否存在 methods 属性。如果存在，它会从上面拉取方法 的信息。
* provide\_automatic\_options: 如果设置了这个属性， Flask 会强制禁用或 启用 HTTP OPTIONS 响应的自动实现。这对于对单个视图自定义 OPTIONS 响应而使用装饰器的情况下是有用的。
* required\_methods: 如果这个属性被设置了， 当注册一个 URL 规则的时候， Flask 将总是会添加这些 methods 即使 methods 参数在 route() 调用 的时候被显式的覆盖了。

完整的例子:

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)

0.8 新版功能: 加入了 provide\_automatic\_options 功能。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_htmlfaq_html) | [上一页](#Top_of_api_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# Flask 中的设计决策

如果你好奇 Flask 为什么用它的方式做事情，而不是别的方法，那么这节是为你准 备的。这节应该给你一些设计决策的想法，也许起初是武断且令人惊讶的，特别是 直接与其它框架相比较。

## 显式的应用对象

一个基于 WSGI 的 Python web 应用必须有一个中央的可调用对象来实现实际的应 用。在 Flask 中，这是一个 [Flask](#class_flask_Flask_import_name__s) 类的实例。每个 Flask 应用 必须创建一个该类的实例，并传给它模块的名称，但是为什么 Flask 不自己这么 做？

当不是像下面的代码这样使用一个显式的应用对象时:

from flask import Flask app = Flask(\_\_name\_\_) @app.route('/') def index(): return 'Hello World!'

看起来会是这样:

from hypothetical\_flask import route @route('/') def index(): return 'Hello World!'

这样做有三个主要的原因。最重要的一个是，显式的应用对象需要在同一时刻只存在 一个实例。有许多方法来用单个应用对象来仿造多个应用，像维护一个应用的栈一样， 但这会导致一些问题，这里不会赘述。现在问题是：什么时候一个微框架在同一时刻需 要至少一个应用？一个很好的例子是单元测试。当你想要测试什么的时候，创建一个 最小化的应用来测试特定的行为非常有用。当应用对象删除时，它分配的一切都会被 再次释放。

当你的代码中有一个显式的对象时，继承基类（ [Flask](#class_flask_Flask_import_name__s) ）来更改 特定行为将成为可能。如果基于一个不暴露给你的类的对象在你之前创建，这么做只 能通过 hack 。

此外， Flask 依赖于一个那个类的显式实例还有一个非常重要的原因是：包名称。无 论何时你创建一个 Flask 实例，你通常传给它 \_\_name\_\_ 作为包名。 Flask 依赖 这个信息来正确地加载相对于你模块的资源。在 Python 对反射的杰出支持下，它可 以访问包来找出模板和静态文件存储在哪（见 [open\_resource()](#open_resource_resource__mode__rb) ）。当前显然有许多框架不需要任何配置，且能载入相对于你应用的模块的模板。但 是它们需要为此使用当前工作目录，一种非常不值得信赖的决定应用在哪的方式。当 前工作目录是进程间的，而且如果你想要在同一个进程中运行多个应用（这会在你不 知道的一个 web 服务器中发生），路径会断开。更可怕的是：许多 web 服务器不把 你应用的目录，而是文档根目录设定为工作目录，但两者不一定是一个文件夹。

第三个原因是“显明胜于隐含”。那个对象是你的 WSGI 应用，你不需要记住别的东西。 如果你想要应用一个 WSGI 中间件，只需要封装它（虽然有更好的方式来这么做来不 丢失应用对象的引用 [wsgi\_app()](#wsgi_app_environ__start_response) ）。

此外，这个设计使得用工厂函数来创建应用成为可能，这对单元测试和类似的东西 （ [应用程序的工厂函数](#Ying_Yong_Cheng_Xu_De_Gong_Chang) ）十分有用。

## 路由系统

Flask 使用 Werkzeug 路由系统，其被设计为按复杂度自动为路由排序。这意味着， 你可以任意顺序声明路由，而且他们仍会按期望工作。这在你想正确地实现基于装饰 器的路由是必须的，因为当应用被分割为多个模块时装饰器可以以未定义的顺序调用。

另一个 Werkzeug 路由系统的设计决策是， Werkzeug 中的路由试图确保 URL 是唯 一的。 Werkzeug 对此会做的足够多，因为它在路由不明确时自动重定向到一个规 范的 URL 。

## 某个模板引擎

Flask 在模板引擎上做了决定： Jinja2 。为什么 Flask 没有一个即插的模板引擎 接口？显然，你可以使用一个不同的模板引擎，但是 Flask 仍然会为你配置 Jinja2 。虽然 Jinja2 总是 配置的限制可能会消失，但绑定一个模板引擎并使用 的决策不会。

模板引擎与编程语言类似，每个模板引擎都有特定的理解事物工作的方式。表面上， 它们以相同方式工作：你给引擎一个变量的集合让它为模板求值，并返回一个字符 串。

然而，关于相同点的论述结束了。例如 Jinja2 有一个全面的过滤器系统，一个可靠 的模板继承方式，可以从模板内和 Python 代码内使用复用块（宏）的支持，对所有 操作使用 Unicode，支持迭代模板渲染，可配置的语法等等。其它的引擎，一个类似 Genshi——基于 XML 流求值的引擎，模板继承要考虑 XPath 可用性等等。而 Mako 像 对待 Python 模块一样处理模板。

当把一个模板引擎跟一个应用或框架联系到一起，就不只是渲染模板了。比如， Flask 使用 Jinja2 全面的自动转义支持。同样，也提供了从 Jinja2 模板中 访问宏的途径。

不去掉模板引擎的独特特性的模板抽象层是一门对自身的科学，也是像 Flask 的微框架的巨大事业。

此外，扩展也可以简易地依赖于一个现有的模板语言。你可以简单地使用你自己的 模板语言，而扩展会始终依赖于 Jinja 本身。

## 微与依赖

为什么 Flask 把自己叫做微框架，并且它依赖于两个库（也就是 Werkzeug 和 Jinja 2）。为什么不能？如果我们仔细审查 Ruby 的 web 开发，有一个非常 类似 WSGI 的协议。被称作 Rack 的就是它，但是除此之外，它看起来非常像 一个 WSGI 的 Ruby 实现。但是几乎所有的 Ruby 应用不直接使用 Rack ，而是 基于一个相同名字的库。这个 Rack 库与 Python 中的两个库不相伯仲： WebOb （以前叫 Paste ） 和 Werkzeug。 Paste 依然在使用，但是从我的理解，它有 些过时，而赞同 WebOb 。 WebOb 和 Werzeug 的开发是一起开始的，也有着同样 的理念：为其它应用的利用做一个 WSGI 的良好实现。

Flask 是一个受益于 Werkzeug 妥善实现 WSGI 接口（有时是一个复杂的任务） 既得成果的框架。感谢 Python 包基础建设中近期的开发，包依赖不再是问题， 并且只有很少的原因反对依赖其它库的库。

## 线程局域变量

Flask 为请求、会话和一个额外对象（你可以在 [g](#flask_g) 上放置自己的东 西）使用线程局域对象（实际上是上下文局域对象，它们也支持 greenlet 上下文）。 为什么是这样，这不是一个坏主意吗？

是的，通常情况下使用线程局域变量不是一个明智的主意。它们在不基于线程概念的 服务器上会导致问题，并且使得大型应用难以维护。但 Flask 不仅为大型应用或异步 服务器设计。 Flask 想要使得编写一个传统 web 应用的过程快速而简单。

一些关于基于 Flask 大型应用的灵感，见文档的 [聚沙成塔](#Ju_Sha_Cheng_Ta__Zhe_Li_Shi_Zeng) 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向的其它东西。 Flask 只建立 Werkezug 和 Jinja2 的桥梁，前者实现一个合适的 WSGI 应用，后者处理 模板。 Flask 也绑定了一些通用的标准库包，比如 logging 。其它所有一切取决 于扩展。

为什么是这样？众口难调，因此 Flask 不强制把特异的偏好和需求囊括在核心里。 大多数 web 应用都可以说需要一个模板引擎，但并不是每个应用都需要一个 SQL 数据库。

Flask 的思想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和扩展。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_security_html) | [上一页](#Top_of_design_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# HTML/XHTML 常见问题

Flask 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签 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，关于 HTML 和 XHTML 有许多混淆，本文档试图回答一些主要的疑问。

## XHTML 的历史

一段时间， XHTML 的出现欲取代 HTML 。然而，Internet 上几乎没有任何实 际的 XHTML （用 XML 规则处理的 HTML ）网站。这种情况有几个主要的原因。 其一是 Internet Explorer 缺乏对 XHTML 妥善的支持。 XHTML 规范要求 XHTML 必须由 MIME 类型 application/xhtml+xml 来承载，但是 Internet Explorer 拒绝读取这个 MIME 类型下的文件。

虽然配置 Web 服务器来提供正确的 XHTML 相对简单，但很少有人这么做。这可能 是因为正确地使用 XHTML 会是一件痛苦的事。

痛苦的最重要的原因之一是 XML 苛刻的（严格而残忍）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 并显示出能显示的。web 上大多数的 (X)HTML 生成基于非 XML 的模板引擎（比如 Flask 所使用的 Jinja）并不会防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，但是它们经常具有更大的运行时开销， 并且不能直接使用，因为它们要遵守 XML 规则。

大多数用户，不管怎样，假设它们正在正确地使用 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 <br> 要 写为 <br /> 或 <br></br> ）。然而，即使文档可以正确地通过 XHTML 验证，真正决定浏览器中 XHTML/HTML 处理的是前面说到的，经常不被正确设置的 MIME 类型。所以有效的 XHTML 会被视为有效的 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 规范的开发在 2004 年就以 “Web 应用1.0”之名由网页超文本技术工作小组 （Web Hypertext Application Technology Working Group），或 WHATWG（由主要 的浏览器供应商苹果、 Mozilla 以及 Opera 组成）启动了，目的是编写一个新的改 良的HTML 规范，基于现有的浏览器行为，而不是不切实际和不向后兼容的规范。

例如，在 HTML4 中 <title/Hello/ 理论上与 <title>Hello</title> 处理 得完全相同。然而，由于人们已然使用了诸如 <link /> 的 XHTML-like 标签， 浏览器供应商在规范语法之上实现了 XHTML 语法。

在 2007 年，这个标准被 W3C 收入一个新的 HTML 规范，也就是 HTML5 。现在， 随着 XHTML 2 工作组解散和 HTML5 被所有主流浏览器供应商实现，XHTML 正在失去 吸引力。

## HTML vs. XHTML

下面的表格给你一个 HTML 4.01 、 XHTML 1.1 和 HTML5 中可用特性的简要综述。 （不包括 XHTML 1.0 ，因为它被 XHTML 1.1 和几乎不使用的 XHTML5 代替 ）

|  |  |  |  |
| --- | --- | --- | --- |
|  | HTML4.01 | XHTML1.1 | HTML5 |
| <tag/value/ == <tag>value</tag> | Yes [[1]](#Top_of_htmlfaq_html) | No | No |
| 支持 <br/> | No | Yes | Yes [[2]](#Top_of_htmlfaq_html) |
| 支持 <script/> | No | Yes | No |
| 应该使用的 MIME 类型： text/html | Yes | No [[3]](#Top_of_htmlfaq_html) | Yes |
| 应该使用的 MIME 类型： application/xhtml+xml | No | Yes | No |
| 严格的错误处理 | No | Yes | No |
| 内联 SVG | No | Yes | Yes |
| 内联 MathML | No | Yes | Yes |
| <video> 标签 | No | No | Yes |
| <audio> 标签 | No | No | Yes |
| 新的语义标签，比如 <article> | No | No | Yes |
| [[1]](#_1_1) | 这是一个从 SGML 中继承过来的鲜为人知的特性。由于上述的原因，它通常不 被浏览器支持。 |
| [[2]](#_2) | 这用于兼容生成 <br> 之类的服务器代码。它不应该在新代码中出现。 |
| [[3]](#_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 严格地定义了处理规则，并准确地指定了一个浏览器应该如何应对处理中的错 误——不像 XHTML，只简单声明将要中断解析。一些人因显然无效的语法仍生成期望中 结果而困惑（比如，缺失结尾标签或属性值未用引号包裹）。

这些工作是因为大多数浏览器遭遇一个标记错误时的错误处理是宽容的，其它的实际 上也指定了。下面的结构在 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>, <li>, <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 " ）。
* 需要布尔属性来设定一个值。

这意味着下面的页面在 HTML5 中是完全有效的:

<!doctype html> <title>Hello HTML5</title> <div class=header> <h1>Hello HTML5</h1> <p class=tagline>HTML5 is awesome </div> <ul class=nav> <li><a href=/index>Index</a> <li><a href=/downloads>Downloads</a> <li><a href=/about>About</a> </ul> <div class=body> <h2>HTML5 is probably the future</h2> <p> There might be some other things around but in terms of browser vendor support, HTML5 is hard to beat. <dl> <dt>Key 1 <dd>Value 1 <dt>Key 2 <dd>Value 2 </dl> </div>

## HTML5 中的新技术

HTML5 添加了许多新特性来使得 Web 应用易于编写和使用。

* <audio> 和 <video> 标签提供了不使用 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> 以及 <time> 之类的 语义化元素，使得内容易于理解。
* <canvas> 标签，支持强大的绘图 API ，减少了服务器端生成图像来图形化 显示数据的必要。
* 新的表单控件类型，比如 <input type="data"> 使得用户代理记录和验证 其值更容易。
* 高级 JavaScript API ，诸如 Web Storage 、 Web Workers 、 Web Sockets 、 地理位置以及离线应用。

除此之外，也添加了许多其它的特性。 Mark Pilgrim 即将出版的书 [Dive Into HTML5](http://www.diveintohtml5.org/) [http://www.diveintohtml5.org/] 是 HTML5 中新特性的优秀入门书。并不是所有的这些特性已经 都被浏览器支持，无论如何，请谨慎使用。

## 应该使用什么？

一般情况下，答案是 HTML 5 。考虑到 web 浏览器最新的开发，几乎没有理由再去 使用 XHTML 。总结上面给出的原因:

* 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 ，由于它需要复杂的命名空间 API 。
* HTML 添加了数个新特性，包括语义标签和期待已久的 <audio> 和 <video> 标签。
* 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，使用 HTML5 无疑比 XHTML 要好。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_unicode_html) | [上一页](#Top_of_htmlfaq_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 安全注意事项

Web 应用通常面临所有种类的安全问题，并且很难把所有事做的正确。 Flask 试图 为你解决这些事情中的一些，但你仍需要关心更多的问题。

## 跨站脚本攻击（XSS）

跨站脚本攻击的概念是在一个网站的上下文中注入任意的 HTML （以及附带的 JavaScript ）。开发者需要正确地转义文本，使其不能包含任意 HTML 标签来避免 这种攻击。更多的信息请阅读维基百科上关于 [Cross-Site Scripting](http://en.wikipedia.org/wiki/Cross-site_scripting) [http://en.wikipedia.org/wiki/Cross-site\_scripting] 的文章。

Flask 配置 Jinja2 自动转义所有值，除非显式地指明不转义。这就排除了模板导 致的所有 XSS 问题，但是你仍需要在其它的地方小心:

* 生成 HTML 而不使用 Jinja2
* 在用户提交的数据上调用了 [Markup](#class_flask_Markup)
* 发送上传的 HTML 文件，永远不要这么做，使用 Content-Disposition: attachment 标头来避免这个问题
* 发送上传的文本文件。一些浏览器使用基于开头几个字节的 content-type 猜测，所以用户可能欺骗浏览器执行 HTML

另一件非常重要的事情是未用引号包裹的属性。虽然 Jinja2 可以通过转义 HTML 来保护你免受 XSS 问题，仍有一种情况，它不能保护你: 属性注入的 XSS 。为了 应对这种攻击媒介，确保当在属性中使用 Jinja 表达式时，始终用单引号或双引号 包裹属性:

<a href="{{ href }}">the text</a>

为什么这是必要的？因为如果你不这么做，攻击者可以容易地注入自制的 JavaScript 处理器。譬如一个攻击者可以注入这段 HTML+JavaScript:

onmouseover=alert(document.cookie)

当用户鼠标经过这个链接， 会在警告窗口里把 cookie 显示给用户。一个精明的 攻击者可能也会执行其它的 JavaScript 代码，而不是把 cookie 显示给用户。 同 CSS 注入联系在一起，攻击者甚至使得元素填满整个页面，这样用户鼠标在页面 上的任何地方都会触发攻击。

## 跨站请求伪造（CSRF）

另一个大问题是 CSRF 。这是一个非常复杂的话题，我不会在此详细介绍，而只会 提及 CSRF 是什么和理论上如何避免它。

如果你的验证信息存储在 cookie 中，你有隐式的状态管理。“已登入”状态由一个 cookie 控制，并且这个 cookie 在每个页面的请求中都会发送。不幸的是，在第三 方站点触发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 社会工程学来诱导你应用的用户在他们不知道的情况下做一些蠢事。

比如你有一个指定的 URL ，当你发送 POST 请求时会删除一个用户的资料（比如 http://example.com/user/delete 。如果一个攻击者现在创造一个页面来用 JavaScript 发送这个 post 请求，他们只是诱骗一些用户加载那个页面，而他们 的资料最终会被删除。

想象你在运行 Facebook ，有数以百万计的并发用户，并且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猫的图片时，他们的资料就被删除。

你怎样才能阻止这呢？基本上，对于每个修改服务器上内容的请求，你应该使用 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，你要比较两个令牌，并确保它们相等。

为什么 Flask 没有为你这么做？理想情况下，这应该是表单验证框架做的事，而 Flask 中并不存在表单验证。

## JSON 安全

ECMAScript 5 的变更

从 ECMAScript 5 开始，常量的行为变化了。现在它们不由 Array 或其它 的构造函数构造，而是由 Array 的内建构造函数构造，关闭了这个特殊的 攻击媒介。

JSON 本身是一种高级序列化格式，所以它几乎没有什么可以导致安全问题，对吗？ 你不能声明导致问题的递归结构，唯一可能导致破坏的就是在接受者角度上，非常 大的响应可以导致某种意义上的拒绝服务攻击。

然而有一个陷阱。由于浏览器在 CSRF 问题上工作的方式， JSON 也不能幸免。幸运 的是， JavaScript 规范中有一个怪异的部分可以用于简易地解决这一问题。 Flask 通过避免你做危险的事情上为你解决了一些。不幸的是，只有在 jsonify() 中有这样的保护，所以如果你用其它方法生成 JSON 仍然 有风险。

那么，问题是什么，并且怎样避免？问题是 JSON 中数组是一等公民。想象你在 一个 JSON 请求中发送下面的数据。比如 JavaScript 实现的用户界面的一部分， 导出你所有朋友的名字和邮件地址。并不罕见:

[ {"username": "admin", "email": "admin@localhost"} ]

这当然只在你登入的时候，且只为你这么做。而且，它对一个特定 URL 上的所有 GET 请求都这么做。比如请求的 URL 是 http://example.com/api/get\_friends.json

那么如果一个聪明的黑客把这个嵌入到他自己的网站上，并用社会工程学使得受害 者访问他的网站，会发生什么:

<script type=text/javascript> var captured = []; var oldArray = Array; function Array() { var obj = this, id = 0, capture = function(value) { obj.\_\_defineSetter\_\_(id++, capture); if (value) captured.push(value); }; capture(); } </script> <script type=text/javascript src=http://example.com/api/get\_friends.json></script> <script type=text/javascript> Array = oldArray; // now we have all the data in the captured array. </script>

如果你懂得一些 JavaScript 的内部工作机制，你会知道给构造函数打补丁和为 setter 注册回调是可能的。一个攻击者可以利用这点（像上面一样上）来获取 所有你导出的 JSON 文件中的数据。如果在 script 标签中定义了内容类型是 text/javascript ，浏览器会完全忽略 application/json 的 mimetype ，而把其作为 JavaScript 来求值。因为顶层数组元素是允许的（虽然 没用）且我们在自己的构造函数中挂钩，在这个页面载入后， JSON 响应中的数据 会出现在 captured 数组中。

因为在 JavaScript 中对象字面量（ {...} ）处于顶层是一个语法错误，攻 击者可能不只是用 script 标签加载数据并请求一个外部的 URL 。所以， Flask 所做的只是在使用 jsonify() 时允许对象作为顶层元素。确保使用 普通的 JSON 生成函数时也这么做。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_extensiondev_html) | [上一页](#Top_of_security_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# Flask 中的 Unicode

Flask 与 Jinja2 、 Werkzeug 一样，文本方面完全基于 Unicode ，大多数 web 相关的 Python 库同样这样处理文本。如果你还不知道 Unicode 是什么，可能需要阅读 [The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets](http://www.joelonsoftware.com/articles/Unicode.html) [http://www.joelonsoftware.com/articles/Unicode.html] 。 这些文档对基本知识做了一些封装，保证你处理 Unicode 相关的事情有愉快的经历。

## 自动转换

为了提供基本无痛的 Unicode 支持，Flask做了这些假设：

* 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 字符的字面量字符串
* 只要协议会话需要传送字节，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，不仅是说协议，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 系统与数据库会话，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 也会顾及到。

## 金科玉律

经验法则：如果你不需要处理二进制数据，请一律使用 Unicode 。在 Python 2.x 中，使用 Unicode 意味着什么？

* 使用 ASCII charpoints （基本是数字、非变音或非奇特的拉丁字母），你可以使用常规的 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 的前缀（比如 u'Hänsel und Gretel' ）
* 如果你在 Python 文件中使用了非 Unicode 字符，你需要告诉 Python 你的文 件使用了何种编码。我再次建议为此使用 UTF-8 。你可以在你 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 被配置为从 UTF-8 解码模板文件，所以确保你的编辑器也保存文件为 UTF-8 编码。

## 自行编解码

如果你的工作环境是一个不真正基于 Unicode 的文件系统之类的话，你需要确保使用 Unicode 接口妥善地解码。比如，当你想要在文件系统中加载一个文件，并嵌入到 Jinja2 模板时， 你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有效的编码， 所以限定你在文本文件中使用 UTF-8 也是在帮自己的忙。

无论如何，以 Unicode 加载这样文件，你可以使用内置的 str.decode() 方法:

def read\_file(filename, charset='utf-8'): with open(filename, 'r') as f: return f.read().decode(charset)

从 Unicode 转换成指定的字符集，你可以使用 unicode.encode() 方法:

def write\_file(filename, contents, charset='utf-8'): with open(filename, 'w') as f: 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没有配置为这样，你 需要更改它。这里是设置你编辑器存储为 UTF-8 的通用做法:

* Vim: 在你的 .vimrc 文件中加入 set enc=utf-8
* Emacs: 使用 encoding cookie，或者把这段文字加入到你的 .emacs 配置文件:

(prefer-coding-system 'utf-8) (setq default-buffer-file-coding-system 'utf-8)

* Notepad++:
  1. 打开 设置 -> 首选项 ...
  2. 选择“新建”选项卡
  3. 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。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_styleguide_html) | [上一页](#Top_of_unicode_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# Flask 扩展开发

Flask，一个微框架，通常需要一些重复的步骤来让第三方库工作。因为在很多时候， 这些步骤可以被分离出，来支持多个项目，就有了 [Flask Extension Registry](http://flask.pocoo.org/extensions/) [http://flask.pocoo.org/extensions/] 。

如果你想要为还没有的功能创建你自己的 Flask 扩展，这份扩展开发指南会帮助你 在很短的时间内让你的应用跑起来并且感到像用户一样期待你的扩展运转。

## 剖析扩展

所有的扩展都位于一个叫做 flask\_something 的包，其中“ something ”是你 想要连接的库的名字。那么，例如当你计划要为 Flask 添加一个叫做 simplexml 的库的支持时，你应该把你扩展的包命名为 flask\_simplexml 。

实际的扩展名（人类可读的名称）无论如何会是“Flask-SimpleXML”之类的东西。 确保在名字中包含“Flask”并注意大小写。这是用户可以在他们的 setup.py 文 件中注册你的扩展为依赖的方式。

Flask 设立了一个叫做 flask.ext 的重定向包，用户应该从这个包导入 扩展。例如，如果你有一个叫做 flask\_something 的包，用户应该用 flask.ext.something 的方式导入。这样做是为了从老命名空间的包过度。 详情见 [扩展导入的过渡](#Kuo_Zhan_Dao_Ru_De_Guo_Du__Yi_Du) 。

但是扩展如何看起来像扩展？一个扩展必须保证它可以同时在多个 Flask 应用中工 作。这是必要条件，因为许多人会使用类似 [应用程序的工厂函数](#Ying_Yong_Cheng_Xu_De_Gong_Chang) 的模式来创建 应用来进行单元测试或是支持多套配置。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 文件一起装配，并且在 PyPI 上注册。同 样，开发 checkout 链接也应该能工作，这样才可以在 virtualenv 中容易地安装开 发版本，而不是手动下载库。

Flask 扩展必须以 BSD 或 MIT 或更自由的许可证来许可，这样才能被列入到 Flask Extension Registry 。记住 Flask Extension Registry 是一个人工维护的地方， 并且会视这些库的行为来决定是否进行必要的提前审查。

## “Hello Flaskext!”

那么让我们开始创建这样一个 Flask 扩展。我们这里想要创建的扩展会提供 SQLite3 最基础的支持。

首先我们创建下面的目录结构:

flask-sqlite3/ flask\_sqlite3.py LICENSE README

这里是最重要的文件的内容:

### setup.py

下一个绝对需要的文件是 setup.py ，用于安装你的 Flask 扩展。你可以使用下 面的内容:

""" Flask-SQLite3 ------------- This is the description for that library """ from setuptools import setup setup( name='Flask-SQLite3', version='1.0', url='http://example.com/flask-sqlite3/', license='BSD', author='Your Name', author\_email='your-email@example.com', description='Very short description', long\_description=\_\_doc\_\_, py\_modules=['flask\_sqlite3'], # if you would be using a package instead use packages instead # of py\_modules: # packages=['flask\_sqlite3'], zip\_safe=False, include\_package\_data=True, platforms='any', install\_requires=[ 'Flask' ], classifiers=[ 'Environment :: Web Environment', 'Intended Audience :: Developers', 'License :: OSI Approved :: BSD License', 'Operating System :: OS Independent', 'Programming Language :: Python', 'Topic :: Internet :: WWW/HTTP :: Dynamic Content', 'Topic :: Software Development :: Libraries :: Python Modules' ] )

这有相当多的代码，但是你实际上可以从现有的扩展中直接复制/粘贴，并修改相应的 内容。

### flask\_sqlite3.py

现在这个是你的扩展放代码的位置。但是这样一个扩展到底看起来是什么样？ 最佳实践是什么？继续阅读，你会有一些认识。

## 初始化扩展

许多扩展会需要某种类型的初始化步骤。比如，想象一个应用像文档中建议的一样 ([在 Flask 中使用 SQLite 3](#sqlite3)) 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：你传递应用对象到它。

有两种推荐的初始化应用的方式:

初始化函数:

如果你的扩展叫做 helloworld ，你应该有一个名为 init\_helloworld(app[, extra\_args]) 的函数来为应用初始化扩展。它 可以附加在处理器前/后等位置。

类:

类的工作大多像初始化函数，但可以在之后进一步更改其行为。例如 [OAuth 扩展](http://packages.python.org/Flask-OAuth/) [http://packages.python.org/Flask-OAuth/] 的工作方式，一个 OAuth 对象提供一些诸如 OAuth.remote\_app 的助手函数来创建一个使用 OAuth 的远程应用的引用。

用什么取决于你想要什么。对于 SQLite 3 扩展，我们会使用基于类的方法，因为它 提供用户一个可以承担打开和关闭数据库连接的对象。

关于类，重要的是它们鼓励在模块层内共享。这种情况下，对象本身在任何情况下 不得存储任何应用的特定状态，而必须可以在不同的应用间共享。

## 扩展的代码

下面是用来复制/粘贴的 flask\_sqlite3.py 的内容:

import sqlite3 from flask import current\_app # Find the stack on which we want to store the database connection. # Starting with Flask 0.9, the \_app\_ctx\_stack is the correct one, # before that we need to use the \_request\_ctx\_stack. try: from flask import \_app\_ctx\_stack as stack except ImportError: from flask import \_request\_ctx\_stack as stack class SQLite3(object): def \_\_init\_\_(self, app=None): self.app = app if app is not None: self.init\_app(app) def init\_app(self, app): app.config.setdefault('SQLITE3\_DATABASE', ':memory:') # Use the newstyle teardown\_appcontext if it's available, # otherwise fall back to the request context if hasattr(app, 'teardown\_appcontext'): app.teardown\_appcontext(self.teardown) else: app.teardown\_request(self.teardown) def connect(self): return sqlite3.connect(current\_app.config['SQLITE3\_DATABASE']) def teardown(self, exception): ctx = stack.top if hasattr(ctx, 'sqlite3\_db'): ctx.sqlite3\_db.close() @property def connection(self): ctx = stack.top if ctx is not None: if not hasattr(ctx, 'sqlite3\_db'): ctx.sqlite3\_db = self.connect() return ctx.sqlite3\_db

那么这是这些代码做的事情:

1. \_\_init\_\_ 方法接受一个可选的应用对象，并且如果提供，会调用 init\_app 。
2. 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 会为数据库设定配置，如果不提供配置，默 认是一个内存中的数据库。此外， init\_app 方法附加了 teardown 处理器。 它会试图使用新样式的应用上下文处理器，并且如果它不存在，退回到请求上下文处理 器。
3. 接下来，我们定义了 connect 方法来打开一个数据库连接。
4. 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 在上下文。这也是处理资源的推荐方式：在资源第一次使用时惰性获取资源。

注意这里，我们把数据库连接通过 \_app\_ctx\_stack.top 附加到应用上下文 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。注意如果应用使用不支持它的老版本的 Flask 我们退回到 \_request\_ctx\_stack.top 。

那么为什么我们决定在此使用基于类的方法？因为使用我们的扩展的情况看起来 会是这样:

from flask import Flask from flask\_sqlite3 import SQLite3 app = Flask(\_\_name\_\_) app.config.from\_pyfile('the-config.cfg') db = SQLite3(app)

你之后可以在视图中这样使用数据库:

@app.route('/') def show\_all(): cur = db.connection.cursor() cur.execute(...)

同样地，如果你在请求之外，而你在使用支持应用上下文 Flask 0.9 或之后的版本， 你可以用同样的方法使用数据库:

with app.app\_context(): cur = db.connection.cursor() cur.execute(...)

在 with 块的最后，销毁处理器会自动执行。

此外， init\_app 方法用于支持创建应用的工厂模式:

db = Sqlite3() # Then later on. app = create\_app('the-config.cfg') db.init\_app(app)

记住已审核的 Flask 扩展需要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。

当扩展需要找出当前的应用且它没有一个指向其的引用时，必须使用 [current\_app](#flask_current_app) 上下文局域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 sqlite3\_db 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。在请求销毁时， sqlite3\_db 连接被关闭。 通过使用这个模式， 相同 的 sqlite3 数据库连接在请求期间对任何需要它的东 西都是可访问的。

如果 [\_app\_ctx\_stack](#flask__app_ctx_stack) 因为用户使用了老版本的 Flask 不存在， 建议退化到限定在请求中的 [\_request\_ctx\_stack](#flask__request_ctx_stack) 。

## 销毁行为

这只在你想要支持 Flask 0.6 和更老版本时有关

由于在 Flask 0.7 中关于在请求的最后运行的函数的变更，你的应用需要在此格外 小心，如果要继续支持 Flask 的更老版本。下面的模式是一个兼顾新旧的好方法:

def close\_connection(response): ctx = \_request\_ctx\_stack.top ctx.sqlite3\_db.close() return response if hasattr(app, 'teardown\_request'): app.teardown\_request(close\_connection) else: app.after\_request(close\_connection)

严格地讲，上面的代码是错误的，因为销毁函数接受异常且典型地不返回任何东西。 尽管如此，因为返回值被丢弃，这刚好会工作，假设中间的代码不触碰传递的参数。

## 他山之石，可以攻玉

本文档只接触了扩展开发中绝对的最小部分，如果你想要了解更多，一个非常好的 主意是查看 [Flask Extension Registry](http://flask.pocoo.org/extensions/) [http://flask.pocoo.org/extensions/] 上已有的扩展。如果你感到失落，也有 [邮件列表](http://flask.pocoo.org/mailinglist/) [http://flask.pocoo.org/mailinglist/] 和 [IRC 频道](http://flask.pocoo.org/community/irc/) [http://flask.pocoo.org/community/irc/] 来获取一些漂亮 API 的想法。特别是当你在做之前 没人做过的东西，这会是一个非常好的主意来获得更多投入。这不仅获得人们会想 从扩展中得到什么的想法，也可避免多个开发者重复发明轮子。

记住：良好的 API 设计是困难的，所以请在邮件列表里介绍你的项目，让 其它开发者在 API 设计上助你一臂之力。

最好的 Flask 扩展是那些为 API 共享通用风格的扩展，并且这只在起初就协作时 奏效。

## 已审核的扩展

Flask 也有已审核的扩展的概念。已审核的扩展被作为 Flask 自身的一部分来测 试来保证在新版本中不会破坏。这些已审核的扩展会在 [Flask Extension Registry](http://flask.pocoo.org/extensions/) [http://flask.pocoo.org/extensions/] 中列出，并有相应的标记。如果你想要自己的扩展 通过审核，你需要遵守下面的指导方针:

1. 一个通过审核的 Flask 扩展需要一个维护者。如果一个扩展作者想要超越项目， 项目应该寻找一个新的维护者，包括完整的源码托管过渡和 PyPI 访问。如果没 有可用的维护者，请给 Flask 核心团队访问权限。
2. 一个通过审核的 Flask 扩展必须确切地提供一个名为 flask\_extensioname 的 包或模块。它们也可能驻留在 flaskext 命名空间包内部，虽然现在这不被推荐。
3. 它必须伴随一个可以使用 make test 或 python setup.py test 的调用测 试套件。对于用 make test 测试的套件，扩展必须确保所有测试需要的依赖关 系都被自动处理好。如果测试由 python setup.py test 调用，测试的依赖关系 由 setup.py 文件指定。测试套件也必须是发行版的一部分。
4. 通过审核的扩展的 API 可以通过下面特性的检查: - 一个通过审核的扩展必须支持在同一个 Python 进程中支持多个应用 - 必须支持使用工厂模式创建应用
5. 必须以 BSD/MIT/WTFPL 许可
6. 官方扩展的命名模式是 Flask-ExtensionName 或 ExtensionName-Flask
7. 通过审核的扩展必须在 setup.py 文件里定义好它们的依赖关系，除非因 其在 PyPI 上不可用而不能满足这个依赖。
8. 扩展的文档必须使用两种 Flask 的 Sphinx 文档主题中的一个
9. setup.py 描述（因此PyPI 描述同）必须链接到文档、网站（如果有）， 并且必须有一个链接来自动安装开发版本（ PackageName==dev ）
10. 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
11. 现行扩展必须支持 Python 2.6 以及 2.7

## 扩展导入的过渡

一段时间，我们推荐对 Flask 扩展使用命名空间包。这在实践中被证明是有问题 的，因为许多不同命名空间包系统存在竞争，并且 pip 会自动在不同的系统中切 换，这给用户导致了许多问题。

现在，我们推荐命名包为 flask\_foo 替代过时的 flaskext.foo 。Flask 0.8 引入了重定向导入系统，允许从 flask.ext.foo 导入，并且如果 flaskext.foo 失败时，会首先尝试 flask\_foo 。

Flask 扩展应该力劝用户从 flask.ext.foo 导入，而不是 flask\_foo 或 flaskext\_foo ，这样扩展可以迁移到新的包名称而不烦扰用户。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_python3_html) | [上一页](#Top_of_extensiondev_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# Pocoo 风格指引

Pocoo 风格指引是所有 Pocoo 项目的风格指引，包括 Flask 。这份风格指引 在 Flask 补丁中是必须的，并且推荐在 Flask 扩展中使用。

一般而言， Pocoo 风格指引遵循 [PEP 8](http://www.python.org/dev/peps/pep-0008) [http://www.python.org/dev/peps/pep-0008] ，有一些小差异和扩充。

## 总体布局

缩进:

4个空格。没有制表符，没有例外。

最大行长:

79字符，软限制是 84 ，如果绝对必要。尝试合理放置 break 、 continue 和 return 声明来避免代码过度嵌套。

可续长语句:

你可以使用反斜线来继续一个语句，在这种情况下，你应该对齐下一行到最后一个 点或等号或缩进四个空格:

this\_is\_a\_very\_long(function\_call, 'with many parameters') \ .that\_returns\_an\_object\_with\_an\_attribute MyModel.query.filter(MyModel.scalar > 120) \ .order\_by(MyModel.name.desc()) \ .limit(10)

如果在一个带括号的语句中换行，对齐到括号:

this\_is\_a\_very\_long(function\_call, 'with many parameters', 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:

items = [ 'this is the first', 'set of items', 'with more items', 'to come in this line', 'like this' ]

空行:

顶层函数和类由两个空行分隔，其它东西一行。不要使用太多的空行来分隔 代码中的逻辑段。示例:

def hello(name): print 'Hello %s!' % name def goodbye(name): print 'See you %s.' % name class MyClass(object): """This is a simple docstring""" def \_\_init\_\_(self, name): self.name = name def get\_annoying\_name(self): return self.name.upper() + '!!!!111'

## 表达式和语句

常规空格规则:

* 不对一元运算符使用空格（例如 - 、 ~ 等等），对圆括号同理
* 在二元运算符间使用空格

Good

exp = -1.05 value = (item\_value / item\_count) \* offset / exp value = my\_list[index] value = my\_dict['key']

Bad

exp = - 1.05 value = ( item\_value / item\_count ) \* offset / exp value = (item\_value/item\_count)\*offset/exp value=( item\_value/item\_count ) \* offset/exp value = my\_list[ index ] value = my\_dict ['key']

禁止使用 Yoda 语句:

永远不要用变量与常量做比较，而是把常量与变量做比较:

God

if method == 'md5': pass

Bad

if 'md5' == method: pass

比较:

* 跟任意类型: == 和 !=
* 跟单例，使用 is 和 is not （例如 foo is not None ）
* 永远不要与 True 或 False 做比较（比如永远不要写 foo == False ，而使用 not foo ）

否定包含检查:

使用 foo not in bar 而不是 not foo in bar

实例检查:

用 isinstance(a, C) 而不是 type(A) is C ， 但通常试图避免 实例检查，请对特性检查。

## 命名约定

* 类名: CamelCase ，缩写词大写（ HTTPWriter 而非 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，双下划线为 mixin 类保留。

有关键字的类上，在末尾添加下划线。允许与内置组建冲突，并且 一定不要 在用在变量名后添加下划线的方式解决。如果函数需要访问一个隐蔽 的内置构件，重绑定内置构件到一个不同的名字作为替代。

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
* 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

## 文档注释

文档字符串约定:

所有的文档注释应为 Sphinx 可理解的 reStructuredText 格式，其格式根据注释行数而变化。 如果只有一行，闭合的三引号和开头的三引号在同一行， 否则开头的三引号与文本在同一行，而闭合的三引号另起一行:

def foo(): """This is a simple docstring""" def bar(): """This is a longer docstring with so much information in there that it spans three lines. In this case the closing triple quote is on its own line. """

模块标头:

模块标头包含一个 utf-8 编码声明（即使没有使用非 ASCII 字符，也始终推荐这么做） 和一个标准的文档注释:

# -\*- coding: utf-8 -\*- """ package.module ~~~~~~~~~~~~~~ A brief description goes here. :copyright: (c) YEAR by AUTHOR. :license: LICENSE\_NAME, see LICENSE\_FILE for more details. """

请留意，合适的版权和许可证文件对于 Flask 扩展通过审核是必须的。

## 注释

注释的规则和文档注释类似。两者都使用 reStructuredText 格式。如果一个 注释被用于一个属性的文档，在起始的井号（ # ）后加一个冒号:

class User(object): #: the name of the user as unicode string name = Column(String) #: the sha1 hash of the password + inline salt pw\_hash = Column(String)

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_upgrading_html) | [上一页](#Top_of_styleguide_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# Python 3 支持

Flask 与其所有依赖包都支持 Python 3，所以理论上你已经可以在 Python 3 中 工作了。尽管如此，在你开始为新项目采用 Python 3 之前，仍有一些事情应该 了解。

## 需求

如果你想要在 Python 3 中使用 Flask，你需要安装 Python 3.3 或更高的版本。 不支持 3.2 及更古老版本。

此外你需要使用最新且最大版本的 itsdangerous 、 Jinja2 和 Werkzeug 。

## API 稳定性

Python 3 中做出的一些关于 Unicode 和 byte 的修改使得编写底层 代码变得困难。这主要影响 WSGI 中间件和与 WSGI 所提供信息的交互。Werkzeug 把所有 WSGI 信息封转成高层的辅助函数，但其中的一些是特地为 Python 3 支持而 新添加的。

很多关于 WSGI 使用的文档是在 WSGI 升级至 Python 3 编写的，未顾及这些细节。 虽然 Werkzeug 和 Flask 在 Python 2.x 中的 API 不会做出大改动，但我们不能保 证 Python 3 中不会发生。

## 少数派

根据 PyPI 下载统计，Python 3 用户只有不足 Python 2 用户的 1%。因此，你遭遇的 Python 3 特定的问题将很难在网上搜索到。

## 小生态系统

大多数的 Flask 扩展、所有的文档和绝大多数 PyPI 提供的库尚未支持 Python 3。 即使你在已知所有所需要的都支持 Python 3 的情况下开始项目，你也不知道接下来 的 6 个月发生什么。如果你富有冒险精神，你可以开始自行移植那些库，但意志薄弱者 则无计可施。

## 建议

除非你已经熟悉各个版本的差异，否则，我们推荐在生态系统完善前，固守当前版 本的 Python。

大多数升级的痛苦在于诸如 Flask 和 Werkzeug 这样的底层库，而非实际的高层应用代码中。 例如 Flask 代码库中所有的 Flask 实例无需修改一行代码就可以在 2.x 和 3.x 中良好运行。

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_changelog_html) | [上一页](#Top_of_python3_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 升级到最新版本

Flask 如同其它软件一样，会随着时间不停地更新自己，其中大部分都会是非常体贴的， 你无需改动一行自身代码就可以应用新版本的 Flask。

不过，每当 Flask 有更新时，我们都建议你适当地修改自己的代码，以充分地利用这些新功能， 提高自己代码地质量。

本章节文档为您列举了 Flask 各版本之间地差异，以及你如何修改自身代码才能无痛地升级。

如果你使用 easy\_install 命令更新、安装 Flask，确保命令中包括 -U 参数

$ easy\_install -U Flask

## Version 0.10

版本 0.9 到 0.10 最大变化在于 cookie 序列化格式从 pickle 转变为了专门的 JSON 格式， 这一更新是为了避免密钥丢失时遭受黑客攻击带来损失。在更新是你会注意到以下两大主要变化： 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.

为了避免破坏用户的 session 数据，你可以使用 Flask 扩展 Flask-OldSessions\_ 来代替原先的 session。

## Version 0.9

从函数中返回元组的操作被简化了，返回元组时你不再需要为你创建的 response 对象定义参数了， 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. 如果你的代码依赖于旧版本，可以通过创建 Flask 的子类简单地解决这个问题

class TraditionalFlask(Flask): def make\_response(self, rv): if isinstance(rv, tuple): return self.response\_class(\*rv) return Flask.make\_response(self, rv)

如果你维护的扩展曾使用 [\_request\_ctx\_stack](#flask__request_ctx_stack) ，可以考虑降至替换为 [\_app\_ctx\_stack](#flask__app_ctx_stack)，但仍须检查是否可行。例如，对于操作数据的扩展来说，app context stack 更说得通，在请求无关的用例中使用它比 request stack 处理起来更容易。

## 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](#class_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](http://werkzeug.pocoo.org/docs/exceptions/#werkzeug.exceptions.BadRequest) [http://werkzeug.pocoo.org/docs/exceptions/#werkzeug.exceptions.BadRequest] exception instead of letting the default [ValueError](http://docs.python.org/dev/library/exceptions.html#ValueError) [http://docs.python.org/dev/library/exceptions.html#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 testsuites 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 an 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:

1. Download the script: [flask-07-upgrade.py](https://raw.github.com/mitsuhiko/flask/master/scripts/flask-07-upgrade.py) [https://raw.github.com/mitsuhiko/flask/master/scripts/flask-07-upgrade.py]
2. Run it in the directory of your application:

python flask-07-upgrade.py > patchfile.diff

1. Review the generated patchfile.
2. Apply the patch:

patch -p1 < patchfile.diff

1. 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 enabled them with the [warnings](http://docs.python.org/dev/library/warnings.html#module-warnings) [http://docs.python.org/dev/library/warnings.html#module-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](http://docs.python.org/dev/library/exceptions.html#RuntimeError) [http://docs.python.org/dev/library/exceptions.html#RuntimeError] instead of an [AttributeError](http://docs.python.org/dev/library/exceptions.html#AttributeError) [http://docs.python.org/dev/library/exceptions.html#AttributeError] when they are unbound. If you caught these exceptions with [AttributeError](http://docs.python.org/dev/library/exceptions.html#AttributeError) [http://docs.python.org/dev/library/exceptions.html#AttributeError] before, you should catch them with [RuntimeError](http://docs.python.org/dev/library/exceptions.html#RuntimeError) [http://docs.python.org/dev/library/exceptions.html#RuntimeError] now.

Additionally the [send\_file()](#flask_send_file_filename_or_fp) function is now issuing deprecation warnings if you depend on functionality that will be removed in Flask 1.0. 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:

return send\_file(my\_file\_object) return send\_file(my\_file\_object)

New code:

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()](#after_request__args____kwargs) decorators continue to work as expected, but for things that absolutely must happen at the end of request we introduced the new [teardown\_request()](#teardown_request__args____kwargs) 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:

@app.after\_request def after\_request(response): g.db.close() return response

You are now encouraged to use this instead:

@app.teardown\_request def after\_request(exception): if hasattr(g, 'db'): 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:

with app.test\_client() as client: resp = client.get('/') # g.db is still bound if there is such a thing # 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 generaly 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:

app.error\_handlers[403] = handle\_error

Into this:

app.register\_error\_handler(403, handle\_error)

Alternatively you should just attach the function with a decorator:

@app.errorhandler(403) def handle\_error(e): ...

(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 guesssing 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 searchpath. 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 postprocessing. 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 as 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 refactoring 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()](#select_jinja_autoescape_filename) 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 make 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()](#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:

app.debug = DEBUG 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 [配置处理](#Pei_Zhi_Chu_Li___0_3_Xin_Ban_Gon).

### 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 [记录应用错误](#Ji_Lu_Ying_Yong_Cuo_Wu___0_3_Xin).

### 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](#Xiao_Xi_Shan_Xian__Hao_De_Ying_Y) pattern.

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_license_html) | [上一页](#Top_of_upgrading_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# Flask Changelog

Here you can see the full list of changes between each Flask release.

## Version 1.0

(release date to be announced, codename to be selected)

* 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.

## Version 0.10.2

(bugfix release, release date to be announced)

* Fixed broken test\_appcontext\_signals() test case.
* Raise an [AttributeError](http://docs.python.org/dev/library/exceptions.html#AttributeError) [http://docs.python.org/dev/library/exceptions.html#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.

## Version 0.10.1

(bugfix release, released on June 14th 2013)

* 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 on June 13nd 2013, 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](#upgrading_to_010) 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 sessions 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 on July 1st 2012, codename Campari.

* The [flask.Request.on\_json\_loading\_failed()](#on_json_loading_failed_e) now returns a JSON formatted response by default.
* The [flask.url\_for()](#flask_url_for_endpoint____values) function now can generate anchors to the generated links.
* The [flask.url\_for()](#flask_url_for_endpoint____values) 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_flask_Flask_import_name__s) 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()](#app_template_filter_name_None).
* The Flask and Blueprint classes now have a non-decorator method for adding custom template filters application wide, [flask.Flask.add\_template\_filter()](#add_template_filter__args____kwa) and [flask.Blueprint.add\_app\_template\_filter()](#add_app_template_filter_f__name).
* The [flask.get\_flashed\_messages()](#flask_get_flashed_messages_with) function now allows rendering flashed message categories in separate blocks, through a category\_filter argument.
* The [flask.Flask.run()](#run_host_None__port_None__debug) 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()](#flask_render_template_template_n) method now accepts a 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()](#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](#class_flask_Response_response_No). This allows for returning jsonify(error="error msg"), 400 from a view function.
* [Flask](#class_flask_Flask_import_name__s) and [Blueprint](#class_flask_Blueprint_name__impo) now provide a [get\_send\_file\_max\_age()](#get_send_file_max_age_filename) hook for subclasses to override behavior of serving static files from Flask when using [flask.Flask.send\_static\_file()](#send_static_file_filename) (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](#flask_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()](#flask_stream_with_context_genera) and the ability to push contexts multiple times without producing unexpected behavior.

## Version 0.8.1

Bugfix release, released on July 1st 2012

* 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 on September 29th 2011, 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 (uploads etc.). Also this is conceptionally 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](#Shi_Li_Wen_Jian_Jia___0_8_Xin_Ba).
* 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](#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 unittests.

## Version 0.7.3

Bugfix release, release date to be decided

* Fixed the Jinja2 environment’s list\_templates method not returning the correct names when blueprints or modules were involved.

## Version 0.7.2

Bugfix release, released on July 6th 2011

* Fixed an issue with URL processors not properly working on blueprints.

## Version 0.7.1

Bugfix release, released on June 29th 2011

* Added missing future import that broke 2.5 compatibility.
* Fixed an infinite redirect issue with blueprints.

## Version 0.7

Released on June 28th 2011, codename Grappa

* Added [make\_default\_options\_response()](#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](http://docs.python.org/dev/library/exceptions.html#RuntimeError) [http://docs.python.org/dev/library/exceptions.html#RuntimeError] instead of an [AttributeError](http://docs.python.org/dev/library/exceptions.html#AttributeError) [http://docs.python.org/dev/library/exceptions.html#AttributeError].
* Mimetype guessing and etag support based on file objects is now deprecated for [flask.send\_file()](#flask_send_file_filename_or_fp) 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](#Upgrading_to_new_Teardown_Handli)
* Implemented [flask.has\_request\_context()](#flask_has_request_context)
* Deprecated init\_jinja\_globals. Override the [create\_jinja\_environment()](#create_jinja_environment) method instead to achieve the same functionality.
* Added [flask.safe\_join()](#flask_safe_join_directory__filen)
* The automatic JSON request data unpacking now looks at the charset mimetype parameter.
* Don’t modify the session on [flask.get\_flashed\_messages()](#flask_get_flashed_messages_with) 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 [即插视图](#Ji_Cha_Shi_Tu___0_7_Xin_Ban_Gong) (class-based views).

## Version 0.6.1

Bugfix release, released on December 31st 2010

* 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 where exposed from.

## Version 0.6

Released on July 27th 2010, 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 .yml file.
* the [config](#config___None) 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()](#flask_make_response__args) 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](http://pypi.python.org/pypi/blinker) [http://pypi.python.org/pypi/blinker] installed.
* refactored the way URL adapters are created. This process is now fully customizable with the [create\_url\_adapter()](#create_url_adapter_request) 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

Bugfix Release, released on July 15th 2010

* fixed another issue with loading templates from directories when modules were used.

## Version 0.5.1

Bugfix Release, released on July 6th 2010

* fixes an issue with template loading from directories when modules where used.

## Version 0.5

Released on July 6th 2010, 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()](#flask_send_file_filename_or_fp) 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 on June 18th 2010, codename Rakia

* added the ability to register application wide error handlers from modules.
* [after\_request()](#after_request__args____kwargs) 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

Bugfix release, released on May 28th 2010

* fixed a error reporting bug with [flask.Config.from\_envvar()](#from_envvar_variable_name__silen)
* removed some unused code from flask
* release does no longer include development leftover files (.git folder for themes, built documentation in zip and pdf file and some .pyc files)

## Version 0.3

Released on May 28th 2010, codename Schnaps

* added support for categories for flashed messages.
* the application now configures a logging.Handler and will log request handling exceptions to that logger when not in debug mode. This makes it possible to receive mails on server errors for example.
* added support for context binding that does not require the use of the with statement for playing in the console.
* the request context is now available within the with statement making it possible to further push the request context or pop it.
* added support for configurations.

## Version 0.2

Released on May 12th 2010, codename Jägermeister

* various bugfixes
* integrated JSON support
* added [get\_template\_attribute()](#flask_get_template_attribute_tem) helper function.
* [add\_url\_rule()](#add_url_rule__args____kwargs) can now also register a view function.
* refactored internal request dispatching.
* server listens on 127.0.0.1 by default now to fix issues with chrome.
* added external URL support.
* added support for [send\_file()](#flask_send_file_filename_or_fp)
* module support and internal request handling refactoring to better support 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

First public preview release.

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [下一页](#Top_of_glossary_html) | [上一页](#Top_of_changelog_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 许可证

Flask 由一个三条款的 BSD 许可证许可。基本上可以认为：你可以用它做任何事情， 只要版权在 Flask 的支持范围内，条款不能被修改，并且提供免责声明。 此外，你不可以在没有书面同意的情况下使用作者的名字来推广衍生作品。

完整的许可证可以在下面找到（ [Flask License](#Flask_License_Copyright__c__2012) ）。对于文档和艺术作品， 使用不同的许可证。

## 作者

Flask is written and maintained by Armin Ronacher and various contributors:

### Development Lead

* Armin Ronacher <armin.ronacher@active-4.com>

### Patches and Suggestions

* Adam Zapletal
* Ali Afshar
* Chris Edgemon
* Chris Grindstaff
* Christopher Grebs
* Florent Xicluna
* Georg Brandl
* Justin Quick
* Kenneth Reitz
* Marian Sigler
* Matt Campell
* Matthew Frazier
* Michael van Tellingen
* Ron DuPlain
* Sebastien Estienne
* Simon Sapin
* Stephane Wirtel
* Thomas Schranz
* Zhao Xiaohong

## 通用许可证定义

下面的章节包含完整的 Flask 及其文档的许可证文本。

* 在此 [作者](#Zuo_Zhe__Flask_is_written_and_ma) 节中列出的所有作者引用自 “AUTHORS” 。
* “[Flask License](#Flask_License_Copyright__c__2012)” 适用于所有作为 Flask 一部分的源代码（ Flask 本 身，同样也包含实例和单元测试），以及文档。
* “[Flask Artwork License](#artwork_license)” 适用于 Flask 的长角标志（Horn-Logo）。

## Flask License

Copyright (c) 2012 by Armin Ronacher and contributors. See AUTHORS for more details.

Some rights reserved.

Redistribution and use in source and binary forms of the software as well as documentation, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
* The names of the contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE AND DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## Flask Artwork License

Copyright (c) 2010 by Armin Ronacher.

Some rights reserved.

This logo or a modified version may be used by anyone to refer to the Flask project, but does not indicate endorsement by the project.

Redistribution and use in source (the SVG file) and binary forms (rendered PNG files etc.) of the image, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice and this list of conditions.
* The names of the contributors to the Flask software (see AUTHORS) may not be used to endorse or promote products derived from this software without specific prior written permission.

Note: we would appreciate that you make the image a link to <http://flask.pocoo.org/> if you use it on a web page.

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [上一页](#Top_of_license_html) | [Flask 0.10.1 文档](#Top_of_index_html) »

# 术语表

Application / App

应用

Application Context

应用上下文

Blueprint

蓝图

Bubble up

冒泡

Configuration / Config

配置

Context Locals

上下文局部变量

Context-local

上下文本地（局域）的

Debugger

调试器

Handler

处理程序

Handling

处理

Header

标头

Logger

日志记录器

Message Flashing

消息闪现

Object Literal

对象字面量

Processor

处理器

Redirect

重定向

Request Context

请求上下文

Sender

（信号的）发送者

Set / Setting

设定

Signal

信号

Token

令牌

Worker

职程

© 版权所有 2013, Armin Ronacher.

[索引](#Top_of_genindex_html) [Flask 0.10.1 文档](#Top_of_index_html) »

# 索引

\_ | [A](#A) | [B](#B) | [C](#C) | [D](#D) | [E](#E) | [F](#F) | [G](#G) | [H](#H) | [I](#I) | [J](#J) | [K](#K) | [L](#L) | [M](#M) | [N](#N) | [O](#O) | [P](#P) | [R](#R) | [S](#S) | [T](#T) | [U](#U) | [V](#V) | [W](#W) | [Y](#Y) | [符号](#Fu_Hao)

## \_

|  |  |
| --- | --- |
| [\_app\_ctx\_stack() (在 flask 模块中)](#flask__app_ctx_stack) | [\_request\_ctx\_stack() (在 flask 模块中)](#flask__request_ctx_stack) |

## A

|  |  |
| --- | --- |
| [abort() (在 flask 模块中)](#flask_abort_code)  [add\_app\_template\_filter() (flask.Blueprint 方法)](#add_app_template_filter_f__name)  [add\_app\_template\_global() (flask.Blueprint 方法)](#add_app_template_global_f__name)  [add\_app\_template\_test() (flask.Blueprint 方法)](#add_app_template_test_f__name_No)  [add\_template\_filter() (flask.Flask 方法)](#add_template_filter__args____kwa)  [add\_template\_global() (flask.Flask 方法)](#add_template_global__args____kwa)  [add\_template\_test() (flask.Flask 方法)](#add_template_test__args____kwarg)  [add\_url\_rule() (flask.Blueprint 方法)](#add_url_rule_rule__endpoint_None)  [(flask.Flask 方法)](#add_url_rule__args____kwargs)  [(flask.blueprints.BlueprintSetupState 方法)](#add_url_rule_rule__endpoint_None_1)  [after\_app\_request() (flask.Blueprint 方法)](#after_app_request_f)  [after\_request() (flask.Blueprint 方法)](#after_request_f)  [(flask.Flask 方法)](#after_request__args____kwargs)  [after\_request\_funcs (flask.Flask 属性)](#after_request_funcs___None)  [app (flask.blueprints.BlueprintSetupState 属性)](#app___None)  [app\_context() (flask.Flask 方法)](#app_context)  [app\_context\_processor() (flask.Blueprint 方法)](#app_context_processor_f) | [app\_ctx\_globals\_class (flask.Flask 属性)](#app_ctx_globals_class)  [app\_errorhandler() (flask.Blueprint 方法)](#app_errorhandler_code)  [app\_template\_filter() (flask.Blueprint 方法)](#app_template_filter_name_None)  [app\_template\_global() (flask.Blueprint 方法)](#app_template_global_name_None)  [app\_template\_test() (flask.Blueprint 方法)](#app_template_test_name_None)  [app\_url\_defaults() (flask.Blueprint 方法)](#app_url_defaults_f)  [app\_url\_value\_preprocessor() (flask.Blueprint 方法)](#app_url_value_preprocessor_f)  [AppContext (flask.ctx 中的类)](#class_flask_ctx_AppContext_app)  [appcontext\_popped() (在 flask 模块中)](#flask_appcontext_popped)  [appcontext\_pushed() (在 flask 模块中)](#flask_appcontext_pushed)  [appcontext\_tearing\_down() (在 flask 模块中)](#flask_appcontext_tearing_down)  [args (flask.Request 属性)](#args)  [as\_view() (flask.views.View 类方法)](#classmethod_as_view_name___class)  [auto\_find\_instance\_path() (flask.Flask 方法)](#auto_find_instance_path) |

## B

|  |  |
| --- | --- |
| [base\_url (flask.Request 属性)](#base_url)  [before\_app\_first\_request() (flask.Blueprint 方法)](#before_app_first_request_f)  [before\_app\_request() (flask.Blueprint 方法)](#before_app_request_f)  [before\_first\_request() (flask.Flask 方法)](#before_first_request__args____kw)  [before\_first\_request\_funcs (flask.Flask 属性)](#before_first_request_funcs___Non)  [before\_request() (flask.Blueprint 方法)](#before_request_f)  [(flask.Flask 方法)](#before_request__args____kwargs) | [before\_request\_funcs (flask.Flask 属性)](#before_request_funcs___None)  [Blueprint (flask 中的类)](#class_flask_Blueprint_name__impo)  [blueprint (flask.blueprints.BlueprintSetupState 属性)](#blueprint___None)  [(flask.Request 属性)](#blueprint)  [blueprints (flask.Flask 属性)](#blueprints___None)  [BlueprintSetupState (flask.blueprints 中的类)](#class_flask_blueprints_Blueprint) |

## C

|  |  |
| --- | --- |
| [Config (flask 中的类)](#class_flask_Config_root_path__de)  [config (flask.Flask 属性)](#config___None)  [context\_processor() (flask.Blueprint 方法)](#context_processor_f)  [(flask.Flask 方法)](#context_processor__args____kwarg)  [cookies (flask.Request 属性)](#cookies)  [copy() (flask.ctx.RequestContext 方法)](#copy) | [create\_global\_jinja\_loader() (flask.Flask 方法)](#create_global_jinja_loader)  [create\_jinja\_environment() (flask.Flask 方法)](#create_jinja_environment)  [create\_url\_adapter() (flask.Flask 方法)](#create_url_adapter_request)  [current\_app() (在 flask 模块中)](#flask_current_app) |

## D

|  |  |
| --- | --- |
| [data (flask.Request 属性)](#data)  [(flask.Response 属性)](#data_1)  [debug (flask.Flask 属性)](#debug)  [debug\_log\_format (flask.Flask 属性)](#debug_log_format)  [decorators (flask.views.View 属性)](#decorators)  [default() (flask.json.JSONEncoder 方法)](#default_o)  [default\_config (flask.Flask 属性)](#default_config___ImmutableDict) | [digest\_method() (flask.sessions.SecureCookieSessionInterface 方法)](#digest_method)  [dispatch\_request() (flask.Flask 方法)](#dispatch_request)  [(flask.views.View 方法)](#dispatch_request_1)  [do\_teardown\_appcontext() (flask.Flask 方法)](#do_teardown_appcontext_exc_None)  [do\_teardown\_request() (flask.Flask 方法)](#do_teardown_request_exc_None)  [dump() (在 flask.json 模块中)](#flask_json_dump_obj__fp____kwarg)  [dumps() (在 flask.json 模块中)](#flask_json_dumps_obj____kwargs) |

## E

|  |  |
| --- | --- |
| [enable\_modules (flask.Flask 属性)](#enable_modules___True)  [endpoint (flask.Request 属性)](#endpoint)  [endpoint() (flask.Blueprint 方法)](#endpoint_endpoint)  [(flask.Flask 方法)](#endpoint__args____kwargs)  [environ (flask.Request 属性)](#environ) | [error\_handler\_spec (flask.Flask 属性)](#error_handler_spec___None)  [errorhandler() (flask.Blueprint 方法)](#errorhandler_code_or_exception)  [(flask.Flask 方法)](#errorhandler__args____kwargs)  [escape() (flask.Markup 类方法)](#classmethod_escape_s)  [(在 flask 模块中)](#flask_escape_s____markup)  [extensions (flask.Flask 属性)](#extensions___None) |

## F

|  |  |
| --- | --- |
| [files (flask.Request 属性)](#files)  [first\_registration (flask.blueprints.BlueprintSetupState 属性)](#first_registration___None)  [flash() (在 flask 模块中)](#flask_flash_message__category__m)  [Flask (flask 中的类)](#class_flask_Flask_import_name__s)  [flask (模块)](#module_flask)  [flask.ext() (在 flask 模块中)](#flask_ext)  [flask.json (模块)](#JSON_Zhi_Chi__Flask_Shi_Yong__si)  [flask.signals.Namespace (內置类)](#class_flask_signals_Namespace) | [FlaskClient (flask.testing 中的类)](#class_flask_testing_FlaskClient)  [FLASKR\_SETTINGS](#index_0)  [form (flask.Request 属性)](#form)  [from\_envvar() (flask.Config 方法)](#from_envvar_variable_name__silen)  [from\_object() (flask.Config 方法)](#from_object_obj)  [from\_pyfile() (flask.Config 方法)](#from_pyfile_filename__silent_Fal)  [full\_dispatch\_request() (flask.Flask 方法)](#full_dispatch_request) |

## G

|  |  |
| --- | --- |
| [g() (在 flask 模块中)](#flask_g)  [get\_cookie\_domain() (flask.sessions.SessionInterface 方法)](#get_cookie_domain_app)  [get\_cookie\_httponly() (flask.sessions.SessionInterface 方法)](#get_cookie_httponly_app)  [get\_cookie\_path() (flask.sessions.SessionInterface 方法)](#get_cookie_path_app)  [get\_cookie\_secure() (flask.sessions.SessionInterface 方法)](#get_cookie_secure_app)  [get\_expiration\_time() (flask.sessions.SessionInterface 方法)](#get_expiration_time_app__session) | [get\_flashed\_messages() (在 flask 模块中)](#flask_get_flashed_messages_with)  [get\_json() (flask.Request 方法)](#get_json_force_False__silent_Fal)  [get\_send\_file\_max\_age() (flask.Blueprint 方法)](#get_send_file_max_age_filename_1)  [(flask.Flask 方法)](#get_send_file_max_age_filename)  [get\_template\_attribute() (在 flask 模块中)](#flask_get_template_attribute_tem)  [got\_first\_request (flask.Flask 属性)](#got_first_request)  [got\_request\_exception() (在 flask 模块中)](#flask_got_request_exception) |

## H

|  |  |
| --- | --- |
| [handle\_exception() (flask.Flask 方法)](#handle_exception_e)  [handle\_http\_exception() (flask.Flask 方法)](#handle_http_exception_e)  [handle\_url\_build\_error() (flask.Flask 方法)](#handle_url_build_error_error__en)  [handle\_user\_exception() (flask.Flask 方法)](#handle_user_exception_e) | [has\_app\_context() (在 flask 模块中)](#flask_has_app_context)  [has\_request\_context() (在 flask 模块中)](#flask_has_request_context)  [has\_static\_folder (flask.Blueprint 属性)](#has_static_folder_1)  [(flask.Flask 属性)](#has_static_folder)  [headers (flask.Request 属性)](#headers)  [(flask.Response 属性)](#headers_1) |

## I

|  |  |
| --- | --- |
| [init\_jinja\_globals() (flask.Flask 方法)](#init_jinja_globals)  [inject\_url\_defaults() (flask.Flask 方法)](#inject_url_defaults_endpoint__va)  [instance\_path (flask.Flask 属性)](#instance_path___None) | [is\_null\_session() (flask.sessions.SessionInterface 方法)](#is_null_session_obj)  [is\_xhr (flask.Request 属性)](#is_xhr) |

## J

|  |  |
| --- | --- |
| [jinja\_env (flask.Flask 属性)](#jinja_env)  [jinja\_loader (flask.Blueprint 属性)](#jinja_loader_1)  [(flask.Flask 属性)](#jinja_loader)  [jinja\_options (flask.Flask 属性)](#jinja_options___ImmutableDict)  [json (flask.Request 属性)](#json)  [json\_decoder (flask.Flask 属性)](#json_decoder) | [json\_encoder (flask.Flask 属性)](#json_encoder)  [JSONDecoder (flask.json 中的类)](#class_flask_json_JSONDecoder_enc)  [JSONEncoder (flask.json 中的类)](#class_flask_json_JSONEncoder_ski)  [jsonify() (在 flask.json 模块中)](#flask_json_jsonify__args____kwar) |

## K

|  |
| --- |
| [key\_derivation (flask.sessions.SecureCookieSessionInterface 属性)](#key_derivation____hmac) |

## L

|  |  |
| --- | --- |
| [load() (在 flask.json 模块中)](#flask_json_load_fp____kwargs)  [loads() (在 flask.json 模块中)](#flask_json_loads_s____kwargs)  [log\_exception() (flask.Flask 方法)](#log_exception_exc_info) | [logger (flask.Flask 属性)](#logger)  [logger\_name (flask.Flask 属性)](#logger_name) |

## M

|  |  |
| --- | --- |
| [make\_config() (flask.Flask 方法)](#make_config_instance_relative_Fa)  [make\_default\_options\_response() (flask.Flask 方法)](#make_default_options_response)  [make\_null\_session() (flask.Flask 方法)](#make_null_session)  [(flask.sessions.SessionInterface 方法)](#make_null_session_app)  [make\_response() (flask.Flask 方法)](#make_response_rv)  [(在 flask 模块中)](#flask_make_response__args)  [make\_setup\_state() (flask.Blueprint 方法)](#make_setup_state_app__options__f)  [Markup (flask 中的类)](#class_flask_Markup)  [match\_request() (flask.ctx.RequestContext 方法)](#match_request)  [max\_content\_length (flask.Request 属性)](#max_content_length) | [message\_flashed() (在 flask 模块中)](#flask_message_flashed)  [method (flask.Request 属性)](#method)  [methods (flask.views.View 属性)](#methods___None)  [MethodView (flask.views 中的类)](#class_flask_views_MethodView)  [mimetype (flask.Response 属性)](#mimetype)  [modified (flask.session 属性)](#modified)  [(flask.sessions.SessionMixin 属性)](#modified___True)  [module (flask.Request 属性)](#module) |

## N

|  |  |
| --- | --- |
| [name (flask.Flask 属性)](#name)  [new (flask.session 属性)](#new)  [(flask.sessions.SessionMixin 属性)](#new___False) | [null\_session\_class (flask.sessions.SessionInterface 属性)](#null_session_class)  [NullSession (flask.sessions 中的类)](#class_flask_sessions_NullSession) |

## O

|  |  |
| --- | --- |
| [on\_json\_loading\_failed() (flask.Request 方法)](#on_json_loading_failed_e)  [open\_instance\_resource() (flask.Flask 方法)](#open_instance_resource_resource)  [open\_resource() (flask.Blueprint 方法)](#open_resource_resource__mode__rb_1)  [(flask.Flask 方法)](#open_resource_resource__mode__rb) | [open\_session() (flask.Flask 方法)](#open_session_request)  [(flask.sessions.SessionInterface 方法)](#open_session_app__request)  [options (flask.blueprints.BlueprintSetupState 属性)](#options___None) |

## P

|  |  |
| --- | --- |
| [path (flask.Request 属性)](#path)  [permanent (flask.session 属性)](#permanent)  [(flask.sessions.SessionMixin 属性)](#permanent_1)  [permanent\_session\_lifetime (flask.Flask 属性)](#permanent_session_lifetime)  [pickle\_based (flask.sessions.SessionInterface 属性)](#pickle_based___False)  [pop() (flask.ctx.AppContext 方法)](#pop_exc_None_1)  [(flask.ctx.RequestContext 方法)](#pop_exc_None)  [preprocess\_request() (flask.Flask 方法)](#preprocess_request) | [preserve\_context\_on\_exception (flask.Flask 属性)](#preserve_context_on_exception)  [process\_response() (flask.Flask 方法)](#process_response_response)  [propagate\_exceptions (flask.Flask 属性)](#propagate_exceptions)  [push() (flask.ctx.AppContext 方法)](#push_1)  [(flask.ctx.RequestContext 方法)](#push)  Python 提高建议  [PEP 8](#index_0_3) |

## R

|  |  |
| --- | --- |
| [record() (flask.Blueprint 方法)](#record_func)  [record\_once() (flask.Blueprint 方法)](#record_once_func)  [redirect() (在 flask 模块中)](#flask_redirect_location__code_30)  [register() (flask.Blueprint 方法)](#register_app__options__first_reg)  [register\_blueprint() (flask.Flask 方法)](#register_blueprint__args____kwar)  [register\_error\_handler() (flask.Flask 方法)](#register_error_handler_code_or_e)  [register\_module() (flask.Flask 方法)](#register_module_module____option)  [render\_template() (在 flask 模块中)](#flask_render_template_template_n)  [render\_template\_string() (在 flask 模块中)](#flask_render_template_string_sou)  [request (flask 中的类)](#class_flask_request)  [Request (flask 中的类)](#class_flask_Request_environ__pop)  [request\_class (flask.Flask 属性)](#request_class) | [request\_context() (flask.Flask 方法)](#request_context_environ)  [request\_finished() (在 flask 模块中)](#flask_request_finished)  [request\_started() (在 flask 模块中)](#flask_request_started)  [request\_tearing\_down() (在 flask 模块中)](#flask_request_tearing_down)  [RequestContext (flask.ctx 中的类)](#class_flask_ctx_RequestContext_a)  [Response (flask 中的类)](#class_flask_Response_response_No)  [response\_class (flask.Flask 属性)](#response_class)  RFC  [RFC 822](#index_0_2)  [route() (flask.Blueprint 方法)](#route_rule____options_1)  [(flask.Flask 方法)](#route_rule____options)  [routing\_exception (flask.Request 属性)](#routing_exception___None)  [run() (flask.Flask 方法)](#run_host_None__port_None__debug) |

## S

|  |  |
| --- | --- |
| [safe\_join() (在 flask 模块中)](#flask_safe_join_directory__filen)  [salt (flask.sessions.SecureCookieSessionInterface 属性)](#salt____cookie_session)  [save\_session() (flask.Flask 方法)](#save_session_session__response)  [(flask.sessions.SessionInterface 方法)](#save_session_app__session__respo)  [script\_root (flask.Request 属性)](#script_root)  [secret\_key (flask.Flask 属性)](#secret_key)  [SecureCookieSessionInterface (flask.sessions 中的类)](#class_flask_sessions_SecureCooki)  [select\_jinja\_autoescape() (flask.Flask 方法)](#select_jinja_autoescape_filename)  [send\_file() (在 flask 模块中)](#flask_send_file_filename_or_fp)  [send\_from\_directory() (在 flask 模块中)](#flask_send_from_directory_direct)  [send\_static\_file() (flask.Blueprint 方法)](#send_static_file_filename_1)  [(flask.Flask 方法)](#send_static_file_filename)  [serializer (flask.sessions.SecureCookieSessionInterface 属性)](#serializer____flask_sessions_Tag)  [session (flask 中的类)](#class_flask_session)  [session\_class (flask.sessions.SecureCookieSessionInterface 属性)](#session_class)  [session\_cookie\_name (flask.Flask 属性)](#session_cookie_name)  [session\_interface (flask.Flask 属性)](#session_interface____flask_sessi) | [session\_json\_serializer() (在 flask.sessions 模块中)](#flask_sessions_session_json_seri)  [session\_transaction() (flask.testing.FlaskClient 方法)](#session_transaction__args____kwd)  [SessionInterface (flask.sessions 中的类)](#class_flask_sessions_SessionInte)  [SessionMixin (flask.sessions 中的类)](#class_flask_sessions_SessionMixi)  [set\_cookie() (flask.Response 方法)](#set_cookie_key__value_____max_ag)  [should\_ignore\_error() (flask.Flask 方法)](#should_ignore_error_error)  [signal() (flask.signals.Namespace 方法)](#signal_name__doc_None)  [signals\_available() (在 flask 模块中)](#flask_signals_available)  [status (flask.Response 属性)](#status)  [status\_code (flask.Response 属性)](#status_code)  [stream (flask.Request 属性)](#stream)  [stream\_with\_context() (在 flask 模块中)](#flask_stream_with_context_genera)  [striptags() (flask.Markup 方法)](#striptags)  [subdomain (flask.blueprints.BlueprintSetupState 属性)](#subdomain___None) |

## T

|  |  |
| --- | --- |
| [teardown\_app\_request() (flask.Blueprint 方法)](#teardown_app_request_f)  [teardown\_appcontext() (flask.Flask 方法)](#teardown_appcontext__args____kwa)  [teardown\_appcontext\_funcs (flask.Flask 属性)](#teardown_appcontext_funcs___None)  [teardown\_request() (flask.Blueprint 方法)](#teardown_request_f)  [(flask.Flask 方法)](#teardown_request__args____kwargs)  [teardown\_request\_funcs (flask.Flask 属性)](#teardown_request_funcs___None)  [template\_context\_processors (flask.Flask 属性)](#template_context_processors___No)  [template\_filter() (flask.Flask 方法)](#template_filter__args____kwargs)  [template\_global() (flask.Flask 方法)](#template_global__args____kwargs) | [template\_rendered() (在 flask 模块中)](#flask_template_rendered)  [template\_test() (flask.Flask 方法)](#template_test__args____kwargs)  [test\_client() (flask.Flask 方法)](#test_client_use_cookies_True)  [test\_client\_class (flask.Flask 属性)](#test_client_class___None)  [test\_request\_context() (flask.Flask 方法)](#test_request_context__args____kw)  [testing (flask.Flask 属性)](#testing_1)  [trap\_http\_exception() (flask.Flask 方法)](#trap_http_exception_e) |

## U

|  |  |
| --- | --- |
| [unescape() (flask.Markup 方法)](#unescape)  [update\_template\_context() (flask.Flask 方法)](#update_template_context_context)  [url (flask.Request 属性)](#url)  [url\_build\_error\_handlers (flask.Flask 属性)](#url_build_error_handlers___None)  [url\_default\_functions (flask.Flask 属性)](#url_default_functions___None)  [url\_defaults (flask.blueprints.BlueprintSetupState 属性)](#url_defaults___None)  [url\_defaults() (flask.Blueprint 方法)](#url_defaults_f)  [(flask.Flask 方法)](#url_defaults__args____kwargs)  [url\_for() (在 flask 模块中)](#flask_url_for_endpoint____values) | [url\_map (flask.Flask 属性)](#url_map___None)  [url\_prefix (flask.blueprints.BlueprintSetupState 属性)](#url_prefix___None)  [url\_root (flask.Request 属性)](#url_root)  [url\_rule (flask.Request 属性)](#url_rule___None)  [url\_rule\_class (flask.Flask 属性)](#url_rule_class)  [url\_value\_preprocessor() (flask.Blueprint 方法)](#url_value_preprocessor_f)  [(flask.Flask 方法)](#url_value_preprocessor__args)  [url\_value\_preprocessors (flask.Flask 属性)](#url_value_preprocessors___None)  [use\_x\_sendfile (flask.Flask 属性)](#use_x_sendfile) |

## V

|  |  |
| --- | --- |
| [values (flask.Request 属性)](#values)  [View (flask.views 中的类)](#class_flask_views_View) | [view\_args (flask.Request 属性)](#view_args___None)  [view\_functions (flask.Flask 属性)](#view_functions___None) |

## W

|  |
| --- |
| [wsgi\_app() (flask.Flask 方法)](#wsgi_app_environ__start_response) |

## Y

|  |
| --- |
| [YOURAPPLICATION\_SETTINGS](#index_0_1) |

## 符号

|  |
| --- |
| 环境变量  [FLASKR\_SETTINGS](#index_0)  [YOURAPPLICATION\_SETTINGS](#index_0_1) |

© 版权所有 2013, Armin Ronacher.