20180502

第 1 章 Flask 安装

Flask 是小型框架，其有两个主要依赖：路由、调试和 Web 服务器网关接口(WSGI, Web Server Gateway Interface)子系统由 Werkzeug 提供；模板系统由 Jinja2 提供。Werkzeug 和 Jinjia2 都是由 Flask 的核心开发者开发而成。

Flask 并不原生支持数据库访问、Web 表单验证和用户认证等高级功能，需要以扩展的形式实现，然后再与核心包集成。

✧ 虚拟环境 virtualenv
① 安装 virtualenv 包：$ pip install virtualenv
② 创建虚拟环境 venv：$ virtualenv venv
当前目录(比如 hello 目录)生成一个 venv 子目录，虚拟环境名字为 venv
③ 激活虚拟环境 venv：$ venv\Scripts\activate
虚拟环境被激活后，命令提示符变为：(venv) $
其中 Python 解释器临时添加到 PATH
④ 退出虚拟环境：$ deactivate

以后就在虚拟环境 venv 安装 Python 第三方模块，如：

```
(venv) $ pip install flask
(venv) $ pip freeze
click==6.7
Flask==1.0.1
itsdangerous==0.24
Jinja2==2.10
MarkupSafe==1.0
Werkzeug==0.14.1
```

第 2 章 程序的基本结构

✧ 2.1 初始化
所有 Flask 程序必须创建一个程序实例。Web 服务器使用 WSGI 协议把接收自客户端的所有请求转交给这个对象处理。

Flask 类的构造函数只有一个必须指定的参数，即程序主模块或包的名字：

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

✧ 2.2 路由(route)和视图函数(view)
客户端把请求发给 Web 服务器，Web 服务器再把请求发给 Flask 程序实例。程序实例需要知道对每个 URL 请求运行哪些代码，所以保存了一个 URL 到 Python 函数的映射关系。处理 URL 和函数之间关系的程序称为路由。

定义路由最简便方式，是使用程序实例提供的 app.route 装饰器，把函数注册为路由。

```
@app.route('/')
def index():
    return '<h1 style=color:red>hello world!</h1>'
```

动态路由：URL 地址中可以包含可变部分，如/user/<name>，尖括号内容就是动态部分，Flask 将动态部分作为参数传入视图函数：

```
@app.route('/user/<name>')  # 动态路由
def user(name):
    return '<h1>hello, {}!</h1>'.format(name)
```

Flask 支持在路由中使用 int、float 和 path 类型。path 类型也是字符串，但不把斜线视作分隔符，而作为动态的一部分。

```
dct = {5: 'rin', 6: 'maki', 7: 'nozomi'}
@app.route('/user/<int:id>')
def user1(id):
    return '<h1>hello, {}!</h1>'.format(dct.get(id, 'world'))
```

✧ 2.3 启动服务器

```
if __name__ == '__main__':
    # 默认端口 5000, 可以修改
    app.run(debug=True, port=8888)
```

① 公认端口(Well Known Ports)：0~1023，紧密绑定于一些服务。通常这些端口的通讯明确表明了某种服务协议。80 端口实际上总是 HTTP 通讯。
② 注册端口(Registered Ports)：1024~49151，松散地绑定于一些服务。这些端口同样用于许多其它目的。如许多系统处理动态端口从 1024 左右开始。
③ 动态和/或私有端口(Dynamic and/or Private Ports)：49152~65535。理论上不应为服务分配这些端口。实际上机器通常从 1024 起分配动态端口。

服务器启动后，会进入轮询，等待并处理请求。轮询一直运行，直到程序停止。

✧ 2.4 请求-响应循环

请求对象封装了客户端发送的 HTTP 请求。要让视图函数能够访问请求对象，可以将其作为参数传入视图函数，但会导致每个视图函数都增加一个参数。
为了避免传入大量参数使得视图函数变得乱七八糟，Flask 使用上下文临时把某些对象变为全局可访问。

```
from flask import request
@app.route('/')
def index():
    user_agent = request.headers.get('User-Agent')
    return '<h1 style=color:red>hello world!</h1><br><p>{}</p>'.format(user_agent)
```

此视图函数把 request 当全局变量使用。但事实上 request 不可能是全局变量。比如多线程处理不同客户端不同请求时，每个线程的 request 对象一定不同

Flask 上下文全局变量：程序上下文和请求上下文

变量名	上下文	说明
current_app	程序上下文	当前激活程序的程序实例
g	程序上下文	处理请求时用作临时存储的对象。每次请求都会重设此变量
request	请求上下文	请求对象，封装了客户端发出的 HTTP 请求内容
session	请求上下文	用户会话，用于存储请求之间需要记住值的词典

Flask 在分发请求之前激活(或推送)程序和请求上下文，请求处理完成后再将其删除。程序上下文被推送后，可以在线程中使用 current_app 和 g 变量；请求上下文被推送后，可以使用 request 和 session 变量。如果使用这些变量时没有激活上下文，就会导致错误。

✧ 2.5 URL 映射

生成映射除了用 app.route 装饰器，还可以用 app.add_url_rule()方法
使用 app.url_map 可以查看 URL 映射(用 shell 或另一个 py 文件)
同目录的 test.py：

```
from hello import app
print(app.url_map)
```

结果：

```
Map([<Rule '/' (GET, OPTIONS, HEAD) -> index>,
 <Rule '/static/<filename>' (GET, OPTIONS, HEAD) -> static>,
 <Rule '/user/<id>' (GET, OPTIONS, HEAD) -> user1>,
 <Rule '/user/<name>' (GET, OPTIONS, HEAD) -> user>])
```

/static/<*filename*>是 Flask 添加的特殊路由，用于访问静态文件。
URL 映射中的 HEAD、Options、GET 是*请求方法*，由路由处理。Flask 为每个路由指定了请求方法，不同的请求方法发送到相同的 URL 上时，使用不同的视图函数处理。HEAD 和 OPTIONS 方法由 Flask 自动处理。

✧ 2.6 请求钩子

有时在处理请求之前或之后执行相同函数，为了避免每个视图函数都使用重复代码，Flask 提供注册通用函数的功能。

请求钩子通过装饰器实现：
① before_first_request：在处理第一个请求之前运行；
② before_request：在每次请求之前运行；
③ after_request：如果没有未处理的异常抛出，在每次请求之后运行；
④ teardown_request：即使有未处理的异常抛出，也在每次请求之后运行。

在请求钩子函数和视图函数之间共享数据一般使用上下文全局变量 g。
如 before_request 处理程序可从数据库中加载已登录用户，并将其保存到 g.user 中。之后调用视图函数再使用 g.user 获取用户。

✧ 2.7 响应

视图函数返回值作为响应内容，可以是一个简单的字符串，作为 HTML 页面返回客户端。

① 状态码是 HTTP 响应的重要部分，Flask 默认 200，表示成功处理请求。

状态码可以作为视图函数第 2 个返回值：

```
@app.route('/user/<int:id>')
def user1(id):
    name = dct.get(id)
    if name:
        return '<h1>hello, {}!</h1>'.format(name)
    return '<h1>Bad Request</h1>', 400
```

make_response()函数(参数和视图的返回值一样)可以返回一个 Response 对象，返回一个 Response 对象。可以在响应对象上调用各种方法，比如设置 cookie：

```
from flask import make_response
@app.route('/')
def index():
    res = make_response('<h1>F12 查看 cookie</h1>')   # 创建 Response 对象
    res.set_cookie('name', 'hikari')   # 设置 cookie
    return res
```

② 重定向，通常使用 302 状态码，通常在 Web 表单中使用

Flask 提供 redirect()辅助函数生成重定向响应

```
from flask import redirect
@app.route('/')
def index():
    return redirect('https://www.baidu.com')
```

③ abort()函数用于处理错误，生成特殊的响应

```
from flask import abort
@app.route('/user/<int:id>')
def user1(id):
    name = dct.get(id)
    if name is None:
        abort(404)
    return '<h1>hello, {}!</h1>'.format(name)
```

如果 URL 动态参数 id 对应用户不存在就返回 404。

abort 不会把控制权交还给调用的函数，而是抛出异常把控制权交给 Web 服务器。

✧ 2.8 Flask 扩展

Flask 设计为可扩展，没有提供一些重要的功能(如数据库和用户认证)，所以可以自由选择最适合的包，或者按需求自行开发。

使用 Flask-Script 支持命令行选项

Flask 支持很多启动设置选项，但只能在脚本中作为参数传给 app.run()，不方便，传递设置选项的理想方式是使用命令行参数。

// 为什么在下认为命令行反而不友好...

Flask-Script 是一个 Flask 扩展，为 Flask 程序添加一个命令行解析器。Flask-Script 自带一组常用选项，而且支持自定义命令。

```
from flask_script import Manager
app = Flask(__name__)
manager = Manager(app)
# 视图与之前一样
if __name__ == '__main__':
    manager.run()
```

运行 hello.py：

```
$ (venv) python hello.py
usage: hello.py [-?] {shell,runserver} ...
positional arguments:
  {shell,runserver}
    shell           Runs a Python shell inside Flask application context.
    runserver       Runs the Flask development server i.e. app.run()
optional arguments:
  -?, --help        show this help message and exit
```

① shell 命令用于在程序上下文启动 Python Shell 会话，可以测试或维护；

② runserver 命令启动 Web 服务器：

python hello.py runserver --help 可以查看用法：

```
(venv) $ python hello.py runserver --help
usage: hello.py runserver [-?] [-h HOST] [-p PORT] [--threaded]
                          [--processes PROCESSES] [--passthrough-errors] [-d]
                          [-D] [-r] [-R] [--ssl-crt SSL_CRT]
                          [--ssl-key SSL_KEY]
...
```

-h 或--host，默认指定服务器监听 localhost 的连接，所以只接受来自服务器所在计算机发起的连接。要允许同网其他计算机连接服务器指定--host 0.0.0.0：

```
(venv) $ python hello.py runserver --host 0.0.0.0
 * Serving Flask app "hello" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

这样，Web 服务器可使用 http://a.b.c.d:5000/网络中的任一台电脑访问，其中 a.b.c.d 是服务器所在计算机的外网 IP 地址。

第 3 章 模板

例如用户注册，视图函数需要访问数据库，添加新用户，此为业务逻辑；注册完将响应返回浏览器，此为表现逻辑。如果两者耦合太大，使代码难以理解和维护。把表现逻辑移到模板中能降低耦合，提高可维护性。

✧ 3.1 Jinja2 模板引擎

模板是一个包含响应文本的文件，其中包含用占位变量{{ variable }}表示的动态部分，其具体值只在请求上下文中才知道。

渲染：使用真实值替换变量，再返回最终得到的响应字符串。

① 渲染模板

默认 Flask 在程序根目录的 templates 子目录寻找模板。

hello.py：

```
from flask import Flask, render_template
app = Flask(__name__)
@app.route('/')
def index():
    return render_template('index.html')
@app.route('/user/<name>')
def user(name):
    return render_template('user.html', name=name)
if __name__ == '__main__':
    app.run(debug=True, port=8000)
```

index.html：

```
h1 {font: 36px/36px "Microsoft YaHei"; color: red;}
<h1>welcome to hikari's website!!!</h1>
```

user.html：

```
h1 {font: 36px/36px "Microsoft YaHei";}
.user {color: #ff00ff;}
<h1>hello,<span class="user">{{ name }}</span>!</h1>
```

② Jinja2 变量

类似于{{ name }}结构表示变量，是特殊的占位符，告诉模板引擎这个位置的值从渲染模板使用的数据中获取。

```
{{ dct['key'] }} {# 字典根据键获取值 #}
{{ lst[0] }} {# 列表指定索引 #}
{{ lst[i] }} {# 列表索引是变量 #}
{{ obj.func() }} {# 对象的方法 #}
```

③ 过滤器

可以使用过滤器修改变量，格式：`{{ variable|filter }}`

如：`{{ name|capitalize }}`

Jinja2 常用过滤器

常用过滤器	说明
safe	渲染值时不转义
capitalize	首字母转大写，其他字母小写
lower	转换成小写
upper	转换成大写
title	每个单词的首字母转换成大写
trim	去除首尾空格
striptags	渲染之前删除变量所有 HTML 标签

默认出于安全考虑，Jinja2 会转义变量。如果一个变量的值为`'<h1>maki</h1>'`，Jinja2 会将其渲染成`'<h1>maki</h1>'`，浏览器显示<h1>maki</h1>，没有解析成 h1 标签。但很多情况需要显示变量中存储的 HTML 代码，就可使用 safe 过滤器：`{{ name|safe }}`，浏览器显示 **maki**，将其作为 h1 标签解析。
注意：千万别在不可信的值上使用 safe 过滤器，例如用户在表单中输入的文本。

④ Jinja2 控制结构
1) if 语句：

```
{% if user %}
    Hello, {{ user }}!
{% else %}
    Hello, Stranger!
{% endif %}
```

2) for 语句：

```
<ul>
    {% for i in data %}
        <li>{{ i }}</li>
    {% endfor %}
</ul>
```

3) 宏，类似于函数

```
{% macro show(name) %}
<li>{{ name }}</li>
{% endmacro %}
<ul>
    {% for i in data %} {{ show(i) }} {% endfor %}
</ul>
```

为了重复使用宏，可以将其保存到单独文件如 macros.html；
需要使用时导入：

```
{% import 'macros.html' as macros %}
<ul>
    {% for i in data %}
        {{ macros.show(i) }}
    {% endfor %}
</ul>
```

需要多处重复使用的模板代码片段可以写入单独文件，再包含在所有模板中，以避免重复：`{% include 'common.html' %}`
另一种重复使用代码的强大方式是模板继承，类似于 Python 中的类继承。

⑤ 模板继承
base.html 父模板：

```
<!DOCTYPE html>
<html lang="en">
<head>
    {% block head %}
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>{% block title %}{% endblock %} - hikari app</title>
    {% endblock %}
</head>
<body>
    {% block body %} {% endblock %}
</body>
</html>
```

block 标签在父模板中挖坑，比如上面定义了 head, title, body 3 个坑。

index.html：

```
{% extends "base.html" %} {# 继承于base.html 模板 #}
{% block head %} {{super()}}{# 父模板中内容非空,使用super()获取原来的内容 #}
<style> h1 {font: 36px/36px "Microsoft YaHei"; color: red;}</style>
{% endblock %}
{% block title %}index{% endblock%}
{% block body %}<h1>hello world!</h1>{% endblock %}
```

index.html 继承于 base.html，在其中填坑。

✧ 3.2 Flask-Bootstrap
Bootstrap 是客户端框架，因此不会直接涉及服务器。服务器需要做的只是提供引用了 Bootstrap CSS 和 JavaScript 文件的 HTML 响应，并在 HTML、CSS 和 JavaScript 代码中实例化所需组件。这些操作最理想的执行场所就是模板。
要在程序中集成 Bootstrap，可以使用 Flask-Bootstrap 扩展。

初始化 Flask-Bootstrap 后，在程序中可以使用其提供的父模板 bootstrap/base.html。利用 Jinja2 的模板继承机制，子模板就引入了 Bootstrap 元素。

示例：使用 Flask-Bootstrap 修改 user.html
hello.py：

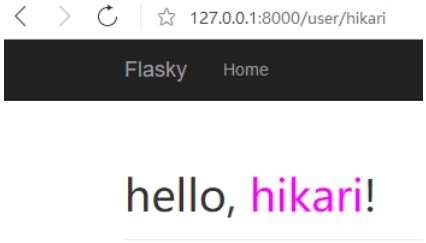
```
from flask import Flask, render_template
from flask_bootstrap import Bootstrap
```

```
app = Flask(__name__)
bootstrap = Bootstrap(app)
@app.route('/')
def index():
    return render_template('index.html')
@app.route('/user/<name>')
def user(name):
    return render_template('user.html', name=name)
if __name__ == '__main__':
    app.run(debug=True, port=8000)
```

user.html：

```
{% extends "bootstrap/base.html" %}
{# 提供网页框架，引入 Bootstrap 所有 CSS 和 JS 文件 #}
{% block title %}User{% endblock%}
{% block styles%}
{{super()}}
<style>
    h1 {font: 36px/36px "Microsoft YaHei";}
    .user {color: #ff00ff;}
</style>
{% endblock %}
{% block navbar %}{# 导航条，直接复制吧，太难记了 #}
<div class="navbar navbar-inverse navbar-static-top" role="navigation">
    <div class="container">
        <div class="navbar-header">
        {# 当设备宽度小，菜单内容折叠时出现此按钮，点击出现 data-target 指向 collapse #}
            <button type="button" class="navbar-toggle" data-toggle="collapse"
data-target=".navbar-collapse">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span></button>
            {# logo 区域#}
            <a class="navbar-brand" href="/">Flasky</a></div>
        <div class="navbar-collapse collapse">
            {# nav navbar-nav:导航条菜单 #}
            <ul class="nav navbar-nav"><li><a href="/">Home</a></li></ul>
        </div></div></div>
{% endblock %}
{% block content %}
<div class="container">
    <div class="page-header">
        <h1>hello, <span class="user">{{ name }}</span>!</h1></div></div>
{% endblock %}
```

效果：



Flask-Bootstrap 父模板中定义的 block：

块名	说明	块名	说明
doc	整个 HTML 文档	styles	css 样式
html_attribs	<html>标签的属性	body_attribs	<body>标签的属性
html	<html>标签的内容	body	<body>标签的内容
head	<head>标签的内容	navbar	用户定义的导航条
title	<title>标签的内容	content	用户定义的页面内容
metas	一组<meta>标签	scripts	文档底部的 JS 声明

其中很多块都是 Flask-Bootstrap 自用，如果直接重定义可能会导致问题。
如 Bootstrap 所需的 css 和 js 文件在 styles 和 scripts 块中声明。如果程序需要在已经有内容的块中添加新内容，必须使用 Jinja2 提供的 super()函数。

✧ 3.3 自定义错误页面

在浏览器输入没有配置的 url，会显示一个 404 错误页面，然而这个页面太丑了！
Flask 可以基于模板自定义错误页面。
常见错误代码：404：客户端请求未知页面；500：有未处理的异常。

① hello.py 自定义错误页面的视图

```
@app.errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404
@app.errorhandler(500)
def internal_server_error(e):
    return render_template('500.html'), 500
```

需要创建 404.html 和 500.html，可以复制 user.html 内容，但是太麻烦；可以使用模板继承，templates/base.html 继承于 bootstrap/base.html，然后 user.html、404.html 和 500.html 都继承此父模板。

② templates/base.html：

```
{% extends "bootstrap/base.html" %}
{% block title %}hikari app{% endblock%}
{% block styles %}
{{ super() }}
<style>
    h1 {font: 36px/36px "Microsoft YaHei";}
    .user {color: #ff00ff;}</style>
```

```
{% endblock %}
{% block navbar %}{# 导航条,直接复制吧,太难记了 #}
<div class="navbar navbar-inverse navbar-static-top" role="navigation">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle" data-toggle="collapse"
data-target=".navbar-collapse">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span></button>
            <a class="navbar-brand" href="/">Flasky</a></div>
        <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
                <li><a href="/">Home</a></li></ul>
        </div></div></div>
{% endblock %} {% block content %}
<div class="container">
    {% block page_content %}{% endblock %}</div>
{% endblock %}
```

和上面 templates/user.html 基本一样，主要最后在 content 坑里挖了一个 page_content 的坑。

③ templates/404.html：使用模板继承自定义 404 错误页面

```
{% extends "base.html" %} {# 继承于自定义的base.html 模板#}
{% block title %}hikari app - Page Not Found{% endblock %} {# 覆盖父模板的title#}
{% block page_content %}
<div class="page-header"><h1>Not Found</h1></div>
{% endblock %}
```

④ templates/500.html：使用模板继承自定义 500 错误页面

```
{% extends "base.html" %}
{% block title %}hikari app - Internal Server Error{% endblock %}
{% block page_content %}
<div class="page-header"><h1>Internal Server Error</h1></div>
{% endblock %}
```

⑤ 简化 templates/user.html：

```
{% extends "base.html" %}
{% block title %}User{% endblock %}
{% block page_content %}
<div class="page-header">
    <h1>hello,<span class="user">{{ name }}</span>!</h1></div>
{% endblock %}
```

✧ 3.4 链接

任何具有多路由的程序都需要可以链接到不同页面，例如导航条。
url_for()函数可以使用 URL 映射中保存的信息生成 URL。
最简单用法是以视图函数名作为参数：
如`{{ url_for('index') }}`返回/；
`{{ url_for('index', _external=True) }}`返回 http://localhost:5000/；
`_external=True`表示绝对地址。

使用 url_for()生成动态地址时，将动态部分作为关键字参数传入。
如`<a href="{{ url_for('user', name='hikari' , _external=True) }}">hikari</a>`
链接地址是 http://localhost:5000/user/hikari
还可以添加查询字符串：`{{ url_for('index', page=2) }}`结果是/?page=2

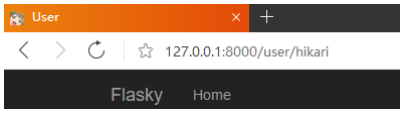
✧ 3.5 静态文件

对静态文件的引用作为特殊路由：*/static/<filename>*
如`{{ url_for('static', filename='css/main.css', _external=True) }}`
结果是：http://localhost:5000/static/css/main.css
Flask 默认在程序根目录的 static 子目录寻找静态文件。

示例：定义收藏夹图标
templates/base.html：

```
{% block head %}
    {{ super() }}
    <link rel="shortcut icon" href="{{ url_for('static', filename =
'shortcuticon.png') }}" type="image/png">
    <link rel="icon" href="{{ url_for('static', filename = 'shortcuticon.png') }}"
type="image/png">
{% endblock %}
```

左上角的小图标：



✧ 3.6 Flask-Moment 本地化日期和时间

服务器需要统一时间，一般使用协调世界时(Coordinated Universal Time，UTC)
但用户更希望看到当地时间，而且采用当地惯用的格式。

一个优雅的解决方案：把时间单位发送给 Web 浏览器，转换成当地时间，然后渲染。因为浏览器能获取用户计算机的时区和区域设置。

moment.js 是使用 JavaScript 开发的优秀客户端开源代码库，可以在浏览器中渲染日期和时间。Flask-Moment 是一个 Flask 程序扩展，把 moment.js 集成到 Jinja2 模板中。

示例：

① hello.py：初始化 Flask-Moment

```
from flask_moment import Moment
moment = Moment(app)
```

② templates/base.html：在底部引入 moment.js 库

```
{% block scripts %}
    {{ super() }}
    {{ moment.include_moment() }}
    {{ moment.lang('zh-CN') }} {# 指定时间戳本地化语言 #}
{% endblock %}
```

为了处理时间戳，Flask-Moment 向模板开放了 moment 类

③ hello.py：index 视图添加一个 now 变量：

```
from datetime import datetime
@app.route('/')
def index():
    return render_template('index.html', now=datetime.utcnow())
```

④ templates/index.html：使用 Flask-Moment 渲染时间戳

```
{% block page_content %}
    <div class="page-header">
        <h3>时间：{{ moment(now).format('YYYY 年 MM 月 DD 日 ddd HH:mm:ss') }}</h3>
        {# 根据电脑的时区和区域设置渲染日期和时间 #}
        <h3>时间：{{ moment(now).format('LLLL') }}</h3>
        <h3>那是{{ moment(now).fromNow(refresh=True) }}。</h3>
    </div>
{% endblock %}
```

结果：

时间：2018年05月02日 周三 17:06:48

时间：2018年5月2日星期三下午5点06分

那是1 分钟前。

fromNow()渲染相对时间戳，会随着时间的推移自动刷新显示的时间。最开始显示几秒前；但指定 refresh 参数后，会随着时间的推移而更新，比如 1 分钟前、2 分钟前等。

Flask-Moment 实现了 moment.js 中的 format()、fromNow()、fromTime()、calendar()、valueOf()和 unix()方法。查阅文档学习全部格式化选项。

第 4 章 Web 表单

Flask-WTF 扩展把处理 Web 表单的过程变成一种愉悦的体验。它对独立的 WTForms 包进行了包装，方便集成到 Flask 程序中。

✧ 4.1 跨站请求伪造保护

默认 Flask-WTF 能保护所有表单免受跨站请求伪造(Cross-Site Request Forgery，CSRF)的攻击。恶意网站把请求发送到被攻击者已登录的其他网站时就会引发 CSRF 攻击。

为了实现 CSRF 保护，Flask-WTF 需要程序设置一个密钥。Flask-WTF 使用这个密钥生成加密令牌，再用令牌验证请求中表单数据的真伪。

示例：hello.py：设置 Flask-WTF 密钥：

```
app = Flask(__name__)
app.config['SECRET_KEY'] = 'hoshizora rin'  # csrf 保护
```

app.config 字典可用来存储框架、扩展和程序本身的配置变量；还提供了一些方法，可以从文件或环境中导入配置值。

SECRET_KEY 配置变量是通用密钥，可在 Flask 和多个第三方扩展中使用。加密的强度取决于变量值的机密程度。不同的程序要使用不同的密钥，而且要保证其他人不知道所用的字符串。

注意：为了增强安全性，密钥不应该直接写入代码，而要保存在环境变量中。

20180503

✧ 4.2 表单类

每个表单都由一个继承自 FlaskForm 的类表示。这个类定义表单中的一组字段，每个字段都用对象表示。字段对象可附属一个或多个验证函数。验证函数用来验证用户提交的输入值是否符合要求。

示例：包含一个文本字段和一个提交按钮的简单表单

```
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField
from wtforms.validators import DataRequired
class NameForm(FlaskForm):  # 一个文本字段和一个提交按钮
    name = StringField('你的名字是？', validators=[DataRequired()])
    submit = SubmitField('提交')
```

StringField 类表示属性为 type="text"的<input>元素。SubmitField 类表示属性为 type="submit"的<input>元素。字段构造函数的第一个参数是把表单渲染成 HTML 时使用的标号。

可选参数 validators 指定一个由验证函数组成的列表，在接受用户提交的数据之前验证数据。验证函数 DataRequired()确保提交的字段不为空。

注：FlaskForm 基类由 Flask-WTF 扩展定义；字段和验证函数却可以直接从 WTForms 包中导入。

WTForms 支持的 HTML 标准字段有：StringField、TextAreaField、PasswordField、

HiddenField、IntegerField 等。
WTForms 验证函数有：Email、Length、Regexp、EqualTo 等。
具体查狗书 P35。

✧ 4.3 表单渲染成 HTML

Flask-Bootstrap 提供一个非常高端的辅助函数，可以使用 Bootstrap 中预先定义好的表单样式渲染整个 Flask-WTF 表单。

```
{% import "bootstrap/wtf.html" as wtf %}
{{ wtf.quick_form(form) }}
```

wtf.quick_form()函数的参数为 Flask-WTF 表单对象，使用 Bootstrap 默认样式渲染传入的表单。

示例：使用 Flask-WTF 和 Flask-Bootstrap 渲染表单
templates/index.html：

```
{% extends "base.html"%}
{% import "bootstrap/wtf.html" as wtf %}
{% block title %}index{% endblock%}
{% block page_content %}
<div class="page-header">
    <h1>Hello, {% if name %}{{ name }}{% else %}Stranger{% endif %}!</h1></div>
{{ wtf.quick_form(form) }} {# 使用Bootstrap 默认样式渲染表单 #}
{% endblock %}
```

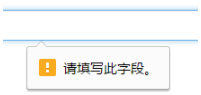
✧ 4.4 在视图函数处理表单

index 视图：

```
@app.route('/', methods=['GET', 'POST'])  # 支持GET 和POST
def index():
    name = None
    form = NameForm()
    if form.validate_on_submit():  # 提交表单,数据被验证函数接受
        name = form.name.data  # 获取字段data 属性
        form.name.data = ''  # 清空字段
    return render_template('index.html', form=form, name=name)
```

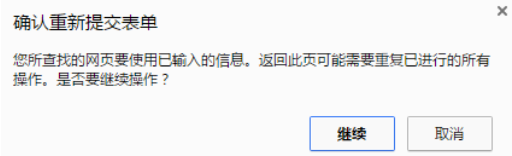
index 视图可以 GET 和 POST 请求。第 1 次访问，服务器收到没有表单数据的 GET 请求，validate_on_submit()返回 False，name 为 None；模板执行 else 语句，显示 Stranger；输入名字后 name 获取数据，传给模板显示：

不输入直接提交，提示：



✧ 4.5 重定向和用户会话

上面表单，提交之后刷新会出现警告，要求再次提交表单之前进行确认。



因为刷新页面，浏览器会重新发送之前已经发送过的最后一个请求。如果是包含表单数据的 POST 请求，刷新页面会再次提交表单。
最好不要让 POST 请求成为浏览器发送的最后一个请求。

可以使用重定向作为 POST 请求的响应。重定向是特殊的响应，响应内容是 URL，而不是 HTML 代码的字符串，浏览器收到重定向响应后，向重定向的 URL 发送 GET 请求。称为 Post/重定向/Get 模式。

但是处理 POST 请求时，使用 form.name.data 获取用户输入，但请求结束，数据随之丢失，所以需要程序能保存输入数据。这样重定向后的请求也可以获得这个数据，从而构建真正的响应。

程序可以把数据存储在用户会话中，在请求之间记住数据。用户会话是私有存储，存在每个连接到服务器的客户端中。用户会话是请求上下文中的变量 session，类似于字典。

注：默认用户会话保存在客户端 cookie 中，使用设置的 SECRET_KEY 进行加密签名。如果篡改了 cookie 的内容，签名就会失效，会话也随之失效。

示例：修改 index()，实现重定向和用户会话：

```
from flask import Flask, redirect, render_template, session, url_for
@app.route('/', methods=['GET', 'POST'])  # 支持 GET 和 POST
def index():
    form = NameForm()
    if form.validate_on_submit():  # 提交表单,数据被验证函数接受
        session['name'] = form.name.data  # 获取字段 data 属性存入 session
        return redirect(url_for('index'))  # 重定向,GET 方式请求 index
    return render_template('index.html', form=form, name=session.get('name'))
```

第 1 次 GET 请求，session 没有 name 字段，显示 stranger；输入名字 POST 提交表单后，name 被存入 session 中，随即重定向再次 GET 请求，显示 name；下次再刷新，仍然是同一个 name。浏览器的 cookies 保存有 session 字段。
session 在不同请求之间共享数据。

✧ 4.6 Flash 消息

flash()函数实现请求完成后告诉用户状态发生了变化，比如用户提交错误登录表单后，服务器发回响应重新渲染表单，显示消息提示用户名或密码错误。

示例：

① index 视图，Flash 消息：

```
from flask import Flask, redirect, render_template, session, url_for, flash
@app.route('/', methods=['GET', 'POST'])  # 支持 GET 和 POST
def index():
    form = NameForm()
    if form.validate_on_submit():  # 提交表单,数据被验证函数接受
        old_name = session.get('name')  # 获取 session 已有 name
        if old_name and old_name != form.name.data:  # 旧名字与获取名字不同
            flash('你似乎改名字了！')
        session['name'] = form.name.data  # 获取字段 data 属性存入 session
        return redirect(url_for('index'))  # 重定向,GET 方式请求 index
    return render_template('index.html', form=form, name=session.get('name'))
```

每次提交名字都会和之前存储在 session 的名字比较，如果不一样就调用 flash()函数，发给客户端下一个响应中显示消息。

仅调用 flash()函数不能显示消息，需要在模板渲染才能显示消息。最好在父模板 templates/base.html 中渲染 flash 消息，这样所有页面都能显示。

② templates/base.html 渲染 Flash 消息：
修改 content 坑位：

```
{% block content %}
    <div class="container">
        {% for message in get_flashed_messages() %}
            <div class="alert alert-warning">
                <button type="button" class="close" data-
dismiss="alert">&times;</button>
                {{ message }}
            </div>
        {% endfor %}
        {% block page_content %}{% endblock %}
    </div>
{% endblock %}
```


Hello, maki!

Flask 把 get_flashed_messages()函数开放给模板，用来获取并渲染消息。
在模板中使用循环是因为在之前的请求循环中每次调用 flash()函数都会生成一个消息，可能有多个消息在排队等待显示。

第 5 章 数据库

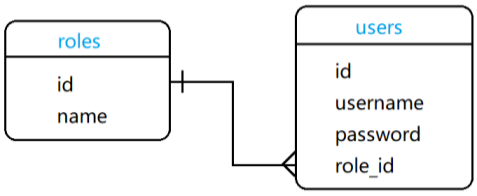
数据库按照一定规则保存数据，程序发起查询取回所需的数据。Web 程序最常用基于关系模型的数据库，也称为 SQL 数据库，因为它们使用结构化查询语言。不过最近几年文档数据库和键值对数据库成了流行的替代选择，这两种合称

NoSQL 数据库。

✧ 5.1 SQL 数据库

关系型数据库把数据存储在表中，表的列是固定的，行数是可变的。表中有个特殊的列称为主键，是各行唯一标识符。表中还可以有称为外键的列，引用同一个或不同表某行的主键。

如下图两个表：



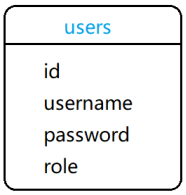
roles 表可用的用户角色，每种角色都有唯一 id；users 表是用户列表，每个用户也都有唯一 id，还有一个外键 role_id 引用 roles 表的 id，指定每个用户角色。

关系型数据库存储数据高效，避免重复。重命名角色很简单，只要修改 roles 表的角色名，users 表不需要修改，通过 role_id 立刻能看到更新。但数据分别存储在不同表中，需要连接查询，使用复杂。

✧ 5.2 NoSQL 数据库

NoSQL 数据库一般用集合代替表，使用文档代替记录。大多数 NoSQL 数据库不支持联结操作。

对于上面 users 和 roles 表，改写成 NoSQL，结构应该是：



每个用户都存储一个具体的角色，重命名操作将会非常耗时。
但 NoSQL 不用联结，虽数据重复量增加，但查询速度快，操作简单。

✧ 5.3 SQL or NoSQL

SQL 数据库擅于用高效且紧凑的形式存储结构化数据，需花费大量精力保证数据的一致性。NoSQL 数据库放宽了要求，性能上有优势。
对中小型程序，SQL 和 NoSQL 都是很好的选择，性能相当。

✧ 5.4 Python 数据库框架

选择数据库框架时要考虑的因素：

① 易用性

如果直接比较数据库引擎和数据库抽象层，显然后者取胜。抽象层，也称为对象关系映射(Object-Relational Mapper，ORM)或对象文档映射(Object-Document Mapper，ODM)，在用户不知觉的情况下把高层的面向对象操作转换成低层的数据库指令。

② 性能

ORM 和 ODM 把对象业务转换成数据库业务会有一定的损耗。大多数情况下，ORM 和 ODM 对生产率的提升远远超过性能的损耗。关键在于如何选择一个能直接操作低层数据库的抽象层，以防特定的操作需要直接使用数据库原生指令优化。

③ 可移植性

④ Flask 集成度

不一定非得选择已经集成了 Flask 的框架，但选择这些框架可以节省编写集成代码的时间，简化配置和操作。

狗书推荐使用 Flask-SQLAlchemy，其集成了 SQLAlchemy 框架。

✧ 5.5 Flask-SQLAlchemy 管理数据库

Flask-SQLAlchemy 简化了在 Flask 程序中使用 SQLAlchemy 的操作。
SQLAlchemy 是一个强大的关系型数据库框架，支持多种数据库后台。
其提供了高层 ORM 和原生 SQL 的低层功能。

在 Flask-SQLAlchemy 中，数据库使用 URL 指定。

常用 FLask-SQLAlchemy 数据库 URL

数据库引擎	URL
MySQL	*mysql://username:password@hostname/database*
Postgres	*postgresql://username:password@hostname/database*
SQLite(Unix)	*sqlite:////absolute/path/to/database*
SQLite(Windows)	*sqlite:///c:/absolute/path/to/database*

注：SQLite 数据库不需要使用服务器，因此不用指定 hostname、username 和 password。URL 中的 database 是硬盘上文件的文件名。

程序使用的数据库 URL 必须保存到 Flask 配置对象的 `SQLALCHEMY_DATABASE_URI` 键中。配置对象中还有一个很有用的选项，即 `SQLALCHEMY_COMMIT_ON_TEARDOWN` 键，设为 True 时，每次请求结束后都会自动提交数据库中的变动。

示例：配置数据库

```
from flask_sqlalchemy import SQLAlchemy
import os
basedir = os.path.abspath(os.path.dirname(__file__))  # flask 程序根目录
app = Flask(__name__)
# 配置 sqlite 数据库
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///' + \
    os.path.join(basedir, 'data.sqlite')  # 数据库名字为 data.sqlite
app.config['SQLALCHEMY_COMMIT_ON_TEARDOWN'] = True  # 自动提交修改
db = SQLAlchemy(app)
```

db 对象是 SQLAlchemy 类的实例，表示程序使用的数据库，同时还获得了 Flask-SQLAlchemy 提供的所有功能。

✧ 5.6 定义模型

模型表示程序使用的持久化实体。在 ORM 中模型一般是一个类，类的属性对应数据库表中的列。

Flask-SQLAlchemy 创建的数据库实例为模型提供了一个父类以及一系列辅助类和辅助函数，可用于定义模型的结构。

示例：定义 Role 和 User 模型

```
class Role(db.Model):
    __tablename__ = 'roles'  # 表名
    id = db.Column(db.Integer, primary_key=True)  # 主键
    name = db.Column(db.String(64), unique=True)
    def __repr__(self):
        return '<Role %r>' % self.name

class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)  # 主键
    username = db.Column(db.String(64), unique=True, index=True)
    def __repr__(self):
        return '<User %r>' % self.username
```

自定义模型继承于 db.Model 类，一个模型对应一张表，模型的属性对应表中的列，是 db.Column 类的实例。

__repr()__返回一个具有可读性的字符串表示模型，可在调试和测试时使用。

常见 SQLAlchemy 列(字段)类型有 String、Integer、Text、Date 等，见狗书 P48

常见 SQLAlchemy 列选项(约束)有：

选项名	说明
primary_key	设为 True 表示主键
unique	设为 True 不允许出现重复的值
index	设为 True 为这列创建索引，提升查询效率
nullable	设为 True，允许使用空值；设为 False，不允许使用空值
default	为这列设置默认值

✧ 5.7 关系

关系型数据库使用关系把不同表中的行联系起来。roles 表示用户角色，有主键 id 和 name 字段；users 表示用户信息，有主键 id、username、password 字段，还有一个外键 role_id 表示用户角色。显然 1 个角色对应 n 个用户，1 个用户只能有 1 个角色，是一对多的关系。

示例：Role 和 User 关系

```
class Role(db.Model):
    __tablename__ = 'roles'  # 表名
    id = db.Column(db.Integer, primary_key=True)  # 主键
```

```
    name = db.Column(db.String(64), unique=True)
    users = db.relationship('User', backref='role')
class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)  # 主键
    username = db.Column(db.String(64), unique=True, index=True)
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))  # 外键
```

添加到 User 模型的 role_id 为外键，db.ForeignKey()参数'role.id'表示这列值是 roles 表的 id 值。

添加到 Role 模型的 users 属性代表这个关系的面向对象视角。对于 Role 类的一个实例，其 users 属性将返回与角色相关联的用户组成的列表。

db.relationship()的第一个参数表示关系的另一端是哪个模型。如果模型类尚未定义，可使用字符串形式指定。
backref 参数向 User 模型中添加一个 role 属性，从而定义反向关系。这一属性可替代 role_id 访问 Role 模型，获取的是模型对象，而不是外键的值。

常用 SQLAlchemy 关系选项

选项名	说明
backref	在关系的另一个模型中添加反向引用
primaryjoin	明确指定两个模型之间使用的联结条件。只在模棱两可的关系中需要指定
lazy	指定如何加载相关记录。可选值有 select(首次访问时按需加载)、immediate(源对象加载后就加载)、joined(加载记录，但使用联结)、subquery(立即加载，但使用子查询)、noload(永不加载)和 dynamic(不加载记录，但提供加载记录的查询)
uselist	如果设为 False，不使用列表，而使用标量值
order_by	指定关系中记录的排序方式
secondary	指定多对多关系中关系表的名字
secondaryjoin	SQLAlchemy 无法自行决定时，指定多对多关系中的二级联结条件

一对一关系可以用一对多关系表示，但调用 db.relationship()时要把 uselist 设为 False，把多变成一。
多对多关系很复杂，需要用到第三张表，称为关系表。

✧ 5.8 数据库操作

学习使用模型最好方法是在 Python shell 中实际操作。
要使用 shell 用 Flask-Script 支持命令行：

```
from flask_script import Manager
manager = Manager(app)
if __name__ == '__main__':
    manager.run()
```

提示 SQLALCHEMY_TRACK_MODIFICATIONS 要设为 True 或 False…

```
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
```

① 创建表：create_all()

```
(venv) $ python hello.py shell
>>> from hello import *
>>> db.create_all()
```

程序根目录多了一个 data.sqlite 的文件。
如果数据库表已经存在于数据库，db.create_all()不会重新创建或者更新这个表。
如果修改模型后要应用到数据库，粗暴的方式是先删除再创建：

```
>>> db.drop_all()
>>> db.create_all()
```

但是这将数据库原来的数据删除了。

② 插入行

```
>>> admin_role = Role(name='Admin')
>>> mod_role = Role(name='Moderator')
>>> user_role = Role(name='User')
>>> hikari = User(username='hikari',role=admin_role)
>>> maki = User(username='maki',role=user_role)
>>> rin = User(username='rin',role=user_role)
```

id 没有明确指定，因为主键由 Flask-SQLAlchemy 管理。

这些对象只存在于 Python 中，还未写入数据库。

```
>>> print(admin_role.id)
None
```

id 没有被赋值，说明数据的确未写入数据库。

通过数据库会话(也称事务)db.session 管理对数据库所做的改动

```
>>> db.session.add(admin_role)
>>> db.session.add(mod_role)
>>> db.session.add(user_role)
>>> db.session.add(hikari)
>>> db.session.add(maki)
>>> db.session.add(rin)
```

可以简写为：

```
>>> db.session.add_all([admin_role, mod_role, user_role, hikari, maki, rin])
```

调用 commit()方法提交会话，将对象写入数据库：

```
>>> db.session.commit()
>>> print(admin_role.id)
1
```

说明 id 已经被赋值了，数据存入了数据库。

数据库会话能保证数据库的一致性。提交操作使用原子方式把会话中的对象全部写入数据库。如果在写入过程中发生了错误，整个会话都会失效。

db.session.rollback()：数据库会话回滚

③ 修改行

数据库会话使用 add()方法也可以更新模型。

比如将'Admin'角色重命名为'Administrator'：

```
>>> admin_role.name='Administrator'
>>> db.session.add(admin_role)
>>> db.session.commit()
```

④ 删除行

数据库会话使用 delete()方法删除行。

比如将"Moderator"字段从数据库删除：

```
>>> db.session.delete(mod_role)
>>> db.session.commit()
```

⑤ 查询行

每个模型类都有 query 对象，查询字段所有行使用 all()方法：

```
>>> Role.query.all()
[<Role 'Administrator'>, <Role 'User'>]
>>> User.query.all()
[<User 'hikari'>, <User 'maki'>, <User 'rin'>]
```

使用过滤器可以对 query 对象进行更精确的数据库查询。

如查询 User 中所有角色是 user_role 的用户：

```
>>> User.query.filter_by(role=user_role).all()
[<User 'maki'>, <User 'rin'>]
```

将 query 对象转换成字符串可以查看 SQLAlchemy 生成的原生 SQL 查询语句：

```
>>> str(User.query.filter_by(role=user_role))
'SELECT users.id AS users_id, users.username AS users_username, users.role_id AS
users_role_id \nFROM users \nWHERE ? = users.role_id'
```

如果退出 shell 会话后，再打开新的 shell 会话，此时没有 Python 对象，需要从数据库读取行，再重新创建 Python 对象。

```
>>> from hello import *
>>> user_role = Role.query.filter_by(name='User').first()
>>> user_role.users
[<User 'maki'>, <User 'rin'>]
```

常用的 SQLAlchemy 查询过滤器

过滤器	说明
filter()	把过滤器添加到原查询上，返回一个新查询
filter_by()	把等值过滤器添加到原查询上，返回一个新查询
limit()	使用指定的值限制原查询返回的结果数量，返回一个新查询
offset()	偏移原查询返回的结果，返回一个新查询
order_by()	根据指定条件对原查询结果进行排序，返回一个新查询
group_by()	根据指定条件对原查询结果进行分组，返回一个新查询

在查询上应用指定的过滤器后，通过调用 all()执行查询，以列表的形式返回结果。还有其他方法能触发查询执行。

常用的 SQLAlchemy 查询执行函数

方法	说明
all()	以列表形式返回查询的所有结果
first()	返回查询的第一个结果；没有返回 None
first_or_404()	返回查询的第一个结果；没有结果则终止请求，返回 404 错误响应
get()	返回指定主键对应的行，没有则返回 None
get_or_404()	返回指定主键对应的行，没有则终止请求，返回 404 错误响应
count()	返回查询结果的数量
paginate()	返回一个 Paginate 对象，包含指定范围内的结果

关系和查询的处理方式类似。
示例：分别从关系的两端查询角色和用户之间的一对多关系：

```
>>> users = user_role.users
>>> users
[<User 'maki'>, <User 'rin'>]
>>> users[0].role
<Role 'User'>
```

但是此时执行 user_role.users 表达式，隐含的查询会调用 all()返回一个用户列表。query 对象是隐藏的，无法指定更精确的查询过滤器。

示例：修改 Role 模型的动态关系

```
class Role(db.Model):
    __tablename__ = 'roles'  # 表名
    id = db.Column(db.Integer, primary_key=True)  # 主键
    name = db.Column(db.String(64), unique=True)
    users = db.relationship('User', backref='role', lazy='dynamic')
```

加入了 lazy = 'dynamic'参数，禁止自动执行查询，可以添加过滤器：

```
>>> users = user_role.users
>>> users
<sqlalchemy.orm.dynamic.AppenderBaseQuery object at 0x055DC210>
>>> users.order_by(User.username.desc()).all()
[<User 'rin'>, <User 'maki'>]
>>> users.count()
2
```

✧ 5.9 在视图函数中操作数据库
① index 视图

```
@app.route('/', methods=['GET', 'POST'])  # 支持 GET 和 POST
def index():
    form = NameForm()
```

```
    if form.validate_on_submit():  # 提交表单,数据被验证函数接受
        user = User.query.filter_by(username=form.name.data).first()  # 数据库查询
        if user is None:  # 数据库不存在用户则添加
            user = User(username=form.name.data)
            db.session.add(user)
            session['known'] = False
        else:
            session['known'] = True
        session['name'] = form.name.data  # 获取字段 data 属性存入 session
        form.name.data = ''  # 清空文本框
        return redirect(url_for('index'))  # 重定向, GET 方式请求 index
    return render_template('index.html', form=form, name=session.get('name'),
known=session.get('known', False))
```

② templates.index.html：

```
{% extends "base.html"%} {# 继承于 base.html 模板 #}
{% import "bootstrap/wtf.html" as wtf %}
{% block title %}index{% endblock%}
{% block page_content %}
<div class="page-header">
    <h1>hello, {% if name %}{{ name }}{% else %}Stranger{% endif %}!</h1>
    {% if not known %}
        <p>很高兴见到你! </p> {# 新用户 #}
    {% else %}
        <p>非常高兴再次见到你! </p> {# 数据库已知用户 #}
    {% endif %}
</div>
{{ wtf.quick_form(form) }} {# 使用 Bootstrap 默认样式渲染表单 #}
{% endblock %}
```

✧ 5.10 集成 Python shell

shell 练习用尚可，但每次都要重复导入模型和实例，枯燥、不友善。可以做些配置，让 Flask-Script 的 shell 命令自动导入特定的对象。

想把对象添加到导入列表中，要为 shell 命令注册一个 make_context 回调函数

```
from flask_script import Shell
# 注册了应用 app、数据库实例 db、模型 User 和 Role
def make_shell_context():
    return dict(app=app, db=db, User=User, Role=Role)
manager.add_command("shell", Shell(make_context=make_shell_context))
```

该函数注册了 app、db、User、Role，因此可以直接导入 shell

```
(venv) $ python hello.py shell
>>> app
<Flask 'hello'>
```

```
>>> db
<SQLAlchemy engine=sqlite:///D:\hello\data.sqlite>
>>> User
<class '__main__.User'>
>>> User.query.all()
[<User 'hikari'>, <User 'rin'>, <User 'maki'>, <User 'haha'>, <User 'nozomi'>]
```

✧ 5.11 Flask-Migrate 实现数据库迁移

开发过程中，有时需要修改数据库模型，修改之后还需要更新数据库。

仅当数据库表不存在时，Flask-SQLAlchemy 才会创建表。因此，更新表的唯一方式就是先删除旧表，不过会丢失数据库中的所有数据。

更好的方法是使用数据库迁移框架。数据库迁移框架能跟踪数据库模式的变化，然后增量式的把变化应用到数据库中。

SQLAlchemy 的主力开发人员编写了一个迁移框架 Alembic；Flask-Migrate 扩展对 Alembic 做了轻量级包装，并集成到 Flask-Script 中。

① 创建迁移仓库

示例：配置 Flask-Migrate

```
from flask_migrate import Migrate, MigrateCommand
migrate = Migrate(app, db)
manager.add_command('db', MigrateCommand)
```

为了导出数据库迁移命令，Flask-Migrate 提供了一个 MigrateCommand 类，可附加到 Flask-Script 的 manager 对象上。
此处，MigrateCommand 类使用 db 命令附加。

在维护数据库迁移之前，要使用 init 子命令创建迁移仓库：

```
(venv) $ python hello.py db init
```

然后生成 migrations 文件夹，所有迁移脚本都存放其中。

注：数据库迁移仓库中的文件要和程序其他文件一起纳入版本控制。

② 创建迁移脚本

在 Alembic 中，数据库迁移用迁移脚本表示。脚本有两个函数：upgrade()把迁移中的改动应用到数据库中；downgrade()将改动删除。
Alembic 可以添加和删除改动，数据库可重设到历史的任意一点。

可以使用 revision 命令手动创建迁移，也可使用 migrate 命令自动创建。
手动创建的迁移只是一个骨架，upgrade()和 downgrade()函数都是空的，要使用 Alembic 提供的 Operations 对象指令实现具体操作。自动创建的迁移会根据模型定义和数据库当前状态之间的差异生成 upgrade()和 downgrade()函数的内容。

注意：自动创建的迁移不一定正确，有可能会漏掉一些细节。自动生成迁移脚本后一定要进行检查。

migrate 子命令自动创建迁移脚本：

```
(venv) $ python hello.py db migrate -m "initial migration"
```

③ 更新数据库

使用 db upgrade 命令把迁移应用到数据库：

```
(venv) $ python hello.py db upgrade
```

对第一个迁移，其作用和调用 db.create_all()方法一样。但后续迁移中，upgrade 命令能把改动应用到数据库中，且不影响其中保存的数据。

第 6 章 电子邮件

很多应用都需要在特定事件发生时提醒用户，比如注册成功、异地登录等，常用的通信方式是电子邮件。虽然 Python 标准库的 smtplib 包可用在 Flask 程序中发送电子邮件；但包装了 smtplib 的 Flask-Mail 扩展能更好地和 Flask 集成。

✧ 6.1 使用 Flask-Mail 提供电子邮件支持

Flask-Mail 连接到 SMTP (Simple Mail Transfer Protocol，简单邮件传输协议)服务器，并把邮件交给这个服务器发送。如果不进行配置，Flask-Mail 会连接 *localhost* 上的端口 25，无需验证即可发送电子邮件。

Flask-Mail SMTP 服务器的配置

配置	默认值	说明
MAIL_SERVER	*localhost*	电子邮件服务器的主机名或 IP 地址
MAIL_PORT	25	电子邮件服务器的端口
MAIL_USE_TLS	False	启用 TLS(Transport Layer Security，传输层安全)协议
MAIL_USE_SSL	False	启用 SSL(Secure Sockets Layer，安全套接层)协议
MAIL_USERNAME	None	邮件账户的用户名
MAIL_PASSWORD	None	邮件账户的密码

示例：配置 Flask-Mail 使用 163 邮箱：

```
from flask_mail import Mail
app.config['MAIL_SERVER'] = 'smtp.163.com'
app.config['MAIL_PORT'] = 465  # SMTP 的加密 SSL 端口
app.config['MAIL_USE_SSL'] = True
app.config['MAIL_USE_TLS'] = False
app.config['MAIL_USERNAME'] = os.environ.get('MAIL_USERNAME')  # xxx@163.com
app.config['MAIL_PASSWORD'] = os.environ.get('MAIL_PASSWORD')  # 授权码，不是密码
mail = Mail(app)
```

注意：邮箱需要手动开启 SMTP 发信功能。

千万不要把账户密码直接写入脚本，特别是要把源码传递 GitHub 或博客时。为了保护账户信息，需要让脚本从环境变量中导入敏感信息。

设置环境变量：

Linux：

```
(venv) $ export MAIL_USERNAME=abc@163.com
```

Windows：

```
(venv) $ set MAIL_USERNAME=abc@163.com
```

20180504

✧ 6.2 在程序中集成发送电子邮件

// 用 163 邮箱按照例子发送邮件总是失败，换 QQ 邮箱了...

QQ 邮箱-->设置-->账户-->开启服务-->POP3/SMTP 服务-->手机发短信-->得到 16 位授权码

① 发送邮件函数 send_mail()：

```
from flask_mail import Mail, Message
app.config['MAIL_SERVER'] = 'smtp.qq.com'
app.config['MAIL_PORT'] = 465  # SMTP 的加密 SSL 端口
app.config['MAIL_USE_SSL'] = True
with open('mail.hikari') as f:
    data = eval(f.read())  # 不想用环境变量...
    my_mail = data.get('mail')  # xxx@qq.com
    my_pwd = data.get('password')  # 授权码，不是密码
app.config['MAIL_USERNAME'] = my_mail
app.config['MAIL_PASSWORD'] = my_pwd
ADMIN = 'hikari_python@163.com'
mail = Mail(app)
def send_mail(to, subject, template, **kwargs):
    # to 为接收方，subject 为邮件主题，template 为渲染邮件正文的模板
    msg = Message(subject, sender=my_mail, recipients=[to])
    msg.body = render_template(template + '.txt', **kwargs)
    msg.html = render_template(template + '.html', **kwargs)
    mail.send(msg)
```

每次有新用户，就用 QQ 邮箱给管理员 hikari_python@163.com 发邮件。

// 为什么不是用管理员邮箱给新用户发邮件?

② index 视图修改：

```
@app.route('/', methods=['GET', 'POST'])  # 支持 GET 和 POST
def index():
    form = NameForm()
    if form.validate_on_submit():  # 提交表单，数据被验证函数接受
        user = User.query.filter_by(username=form.name.data).first()  # 数据库查询
        if user is None:  # 数据库不存在用户则添加
            user = User(username=form.name.data)
            db.session.add(user)
            session['known'] = False
```

```
            if ADMIN:
                send_mail(ADMIN, 'New User', 'mail/new_user', user=user)
        else:
            session['known'] = True
        session['name'] = form.name.data  # 获取字段 data 属性存入 session
        form.name.data = ''  # 清空文本框
        return redirect(url_for('index'))  # 重定向，GET 方式请求 index
    return render_template('index.html', form=form, name=session.get('name'),
known=session.get('known', False))
```

③ 邮件正文

templates/mail/new_user.html：

```
html: User <b>{{ user.username }}</b> has joined.
```

templates/mail/new_user.txt：

```
txt: User {{ user.username }} has joined.
```

分别渲染纯文本和 HTML 版本的邮件正文。

电子邮件的模板中要有一个模板参数是 user，因此调用 send_mail()函数时要以关键字参数的形式传入 user。

结果：

// txt 纯文本呢?

✧ 6.3 异步发送电子邮件

如果发送几封邮件，可能会发现 mail.send()在发送电子邮件时停滞了几秒，为了避免处理请求时不必要得到延迟，可以把发送电子邮件的函数移到后台线程。

示例：异步发送电子邮件

```
from threading import Thread
def send_async_mail(app,msg):
    with app.app_context():
        mail.send(msg)
def send_mail(to, subject, template, **kwargs):
    # to 为接收方，subject 为邮件主题，template 为渲染邮件正文的模板
    msg = Message(subject, sender=my_mail, recipients=[to])
    msg.body = render_template(template + '.txt', **kwargs)
    msg.html = render_template(template + '.html', **kwargs)
    t=Thread(target=send_async_mail,args=[app,msg])
    t.start()
    return t
```

很多 Flask 扩展都假设已经存在激活的程序上下文和请求上下文。 Flask-Mail 中

的 send()函数使用 current_app，因此必须激活程序上下文。但在不同线程中执行 send()函数时，程序上下文要使用 app.app_context()手动创建。

如果程序要发送大量电子邮件，使用专门发送电子邮件的作业要比给每封邮件都新建一个线程更合适。比如可以把执行 send_async_mail()函数操作发给 Celery 任务队列。

第 7 章 大型程序的结构

之前一直在 hello.py 一个文件里搬砖，一旦砖堆多了，看得头昏眼花，后期修改变得复杂而困难。

✧ 7.1 项目结构

大型 Flask 程序项目基本结构：

```
flasky
├── config.py                配置，配置类，继承，组成字典
├── manage.py                生成程序实例，定义shell菜单
├── requirements.txt         依赖包列表
├── venv                     Python环境
├── migrations               数据库迁移脚本
├── app                      应用文件夹
│   ├── __init__.py          创建实例，初始化，引用蓝本
│   ├── email.py             邮件功能
│   ├── models.py            数据库模型
│   ├── main                 蓝本目录
│   │   ├── __init__.py      蓝本初始化
│   │   ├── errors.py        自定义错误路由
│   │   ├── forms.py         表单
│   │   └── views.py         蓝本视图函数、路由
│   ├── static               静态文件,Image/CSS/JS
│   └── templates            模板
└── tests                    单元测试
```

① Flask 程序一般保存在名为 app 的包中；
② migrations 目录包含数据库迁移脚本；
③ 单元测试编写在 test 包；
④ venv 目录包含 Python 虚拟环境；
⑤ requirements.txt 列出所有依赖包，便于在其他电脑生成相同的虚拟环境；
⑥ config.py：配置文件；
⑦ manage.py：用于启动程序和其他的程序任务。

✧ 7.2 配置选项

程序经常需要设定多个配置，如开发、测试、生产环境需要不同的数据库。不再使用 hello.py 中简单的字典结构配置，而使用层次结构的配置类。

示例：配置文件 config.py：

```python
import os
# flask 程序根目录
basedir = os.path.abspath(os.path.dirname(__file__))
my_mail, my_pwd = None, None
with open('mail.hikari') as f:
    data = eval(f.read())
```

```
    my_mail = data.get('mail')  # xxx@qq.com
    my_pwd = data.get('password')  # 授权码，不是密码

def get_db_url(a):
    return os.environ.get('{}DATABASE_URL'.format(a.upper())) or
'sqlite:///{}'.format(os.path.join(basedir, '{}data.sqlite'.format(a)))

class Config():  # 父类通用配置
    # 敏感信息从环境变量获取，但还是给了默认值
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'hoshizora rin'
    MAIL_SERVER = os.environ.get('MAIL_SERVER') or 'smtp.qq.com'
    MAIL_PORT = int(os.environ.get('MAIL_PORT', 465))  # SMTP 的加密SSL 端口
    MAIL_USE_SSL = True  # SSL
    MAIL_USERNAME = os.environ.get('MAIL_USERNAME') or my_mail
    MAIL_PASSWORD = os.environ.get('MAIL_PASSWORD') or my_pwd
    SQLALCHEMY_COMMIT_ON_TEARDOWN = True
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    FLASKY_ADMIN = os.environ.get('FLASKY_ADMIN') or 'hikari_python@163.com'

    @staticmethod
    def init_app(app):  # 对当前环境配置初始化
        pass

# 3 种环境使用不同数据库
class DevelopmentConfig(Config):  # 开发专用配置
    DEBUG = True
    SQLALCHEMY_DATABASE_URI = get_db_url('dev_')

class TestingConfig(Config):  # 测试专用配置
    TESTING = True
    SQLALCHEMY_DATABASE_URI = get_db_url('test_')

class ProductionConfig(Config):  # 生产专用配置
    SQLALCHEMY_DATABASE_URI = get_db_url('')

config = {
    'development': DevelopmentConfig,
    'testing': TestingConfig,
    'production': ProductionConfig,
    'default': DevelopmentConfig  # 默认开发环境的配置
}
```

✧ 7.3 程序包

程序包用来保存程序所有代码、模块和静态文件，这个包可以称为 app (应用)。

templates 和 static 目录是程序包一部分，需要移到 app 目录；数据库模型和电子邮件支持函数 model.py 和 email.py 也需要移到 app 目录。

① 使用程序工厂函数
单个文件开发的缺点是运行脚本时，程序实例已经创建，无法动态修改配置。如在单元测试时，为了提高测试覆盖度，需在不同配置环境中运行程序。

解决方法是延迟创建程序实例，将创建过程移到可显式调用的工厂函数。这样不仅给配置留有时间，还能创建多个程序实例。

示例：程序包的构造文件 app/__init__.py

```
from flask import Flask
from flask_bootstrap import Bootstrap
from flask_mail import Mail
from flask_moment import Moment
from flask_sqlalchemy import SQLAlchemy
from config import config

bootstrap = Bootstrap()
mail = Mail()
moment = Moment()
db = SQLAlchemy()

def create_app(config_name):  # 工厂函数，参数为配置名
    app = Flask(__name__)
    app.config.from_object(config[config_name])  # 导入配置
    config[config_name].init_app(app)
    bootstrap.init_app(app)
    mail.init_app(app)
    moment.init_app(app)
    db.init_app(app)
    # 附加路由和自定义的错误页面
    return app  # 返回创建的程序示例
```

② 在蓝本中实现程序功能
转换成工厂函数的操作让定义路由变得复杂了，因为只有调用 create_app()后才能使用 app.route 装饰器(错误页面处理用 app.errorhandler 装饰器)，这时定义路由太晚了。

Flask 使用蓝本提供了更好的解决方法。蓝本中定义的路由处于休眠状态，直到蓝本注册到程序后，路由才真正成为程序的一部分。使用位于全局作用域中的蓝本时，定义路由的方法几乎和单脚本程序一样。

蓝本可以在单个文件中定义，也可以定义成包，包中多个模块，比如在 app 包中

创建子包 main，用于保存蓝本。

示例 1：创建蓝本：app/main/__init__.py

```
from flask import Blueprint
# 创建蓝本对象，参数为蓝本的名字和蓝本所在包或模块
main = Blueprint('main', __name__)
# app/main/view.py 保存路由和视图函数；app/main/errors.py 保存错误处理
# 导入这两个模块是为了将路由和错误处理与蓝本关联起来
# 末尾导入为了避免循环导入，因为views.py 和errors.py 要导入蓝本main
from . import views, errors
```

蓝本在工厂函数 create_app()中注册到程序。

示例 2：注册蓝本：app/__init__.py：

```
def create_app(config_name):
    # ...
    from .main import main as main_blueprint
    app.register_blueprint(main_blueprint)
    return app  # 返回创建的程序示例
```

示例 3：蓝本中的错误处理：app/main/errors.py：

```
from flask import render_template
from . import main
# 如果使用errorhandler 装饰器，只能处理蓝本中的错误；
# 使用app_errorhandler 装饰器，注册全局错误处理
@main.app_errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404
@main.app_errorhandler(500)
def internal_server_error(e):
    return render_template('500.html'), 500
```

示例 4：蓝本中定义路由：app/main/views.py

```
from flask import render_template, session, redirect, url_for, current_app
from .. import db
from ..models import User
from ..email import send_mail
from . import main
from .forms import NameForm

@main.route('/', methods=['GET', 'POST'])  # 支持GET 和POST
def index():
    form = NameForm()
    if form.validate_on_submit():  # 提交表单,数据被验证函数接受
        user = User.query.filter_by(username=form.name.data).first()  # 数据库查询
```

```
        if user is None:  # 数据库不存在用户则添加
            user = User(username=form.name.data)
            db.session.add(user)
            session['known'] = False
            admin = current_app.config['FLASKY_ADMIN']
            if admin:
                send_mail(admin, 'New User', 'mail/new_user', user=user)
        else:
            session['known'] = True
        session['name'] = form.name.data  # 获取字段 data 属性存入 session
        # 蓝本端点自动添加命名空间，为 main.index；当前蓝本简写为.index
        return redirect(url_for('.index'))  # 重定向，GET 方式请求 index
    return render_template('index.html', form=form, name=session.get('name'),
known=session.get('known', False))
```

蓝本中编写视图函数区别：1. 路由装饰器由蓝本提供，是 main.route()而不是 app.route()；2. url_for()函数用法不同，其第 1 参数是路由的端点名。在程序路由中默认为视图函数名；而 Flask 为蓝本的所有端点加上一个命名空间，这样不同蓝本可以有相同的函数名而不冲突。

所以上面 index 视图注册的端点名是 main.index，在同一蓝本可以简写为.index，而跨蓝本重定向需要带有命名空间。

示例 5：蓝本中的表单类：app/main/forms.py：直接复制

```
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField
from wtforms.validators import DataRequired
class NameForm(FlaskForm):  # 一个文本字段和一个提交按钮
    name = StringField('你的名字是？', validators=[DataRequired()])
    submit = SubmitField('提交')
```

示例 6：app/email.py

```
from threading import Thread
from flask import current_app, render_template
from flask_mail import Message
from . import mail  # __init__.py 中的 mail 对象

# 异步发送邮件
def send_async_mail(app, msg):
    with app.app_context():
        mail.send(msg)

def send_mail(to, subject, template, **kwargs):
    # to 为接收方,subject 为邮件主题,template 为渲染邮件正文的模板
    app = current_app._get_current_object()  # 将 app 传给子线程?
    msg = Message(
```

```
        subject, sender=app.config['FLASKY_MAIL_SENDER'], recipients=[to])
    msg.body = render_template(template + '.txt', **kwargs)
    msg.html = render_template(template + '.html', **kwargs)
    t = Thread(target=send_async_mail, args=[app, msg])
    t.start()
    return t
```

示例 7：app/models.py：将 User 类和 Role 类复制，导入 db 对象

```
from . import db
```

将 static 目录和 templates 目录移到 app 目录

7.4 启动脚本

manage.py 用于启动程序：

```
import os
from app import create_app, db
from app.models import User, Role
from flask_script import Manager, Shell
from flask_migrate import Migrate, MigrateCommand

app = create_app(os.getenv('FLASK_CONFIG') or 'default')
manager = Manager(app)
migrate = Migrate(app, db)

def make_shell_context():
    return dict(app=app, db=db, User=User, Role=Role)

manager.add_command("shell", Shell(make_context=make_shell_context))
manager.add_command('db', MigrateCommand)

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

✧ 7.5 需求文件

程序中必须包含一个 requirements.txt 文件，记录所有依赖包及其精确的版本号。
用于在另一台电脑重新生成虚拟环境，如部署程序时使用的电脑。

自动生成需求文件：

```
(venv) $ pip freeze >requirements.txt
```

✧ 7.6 单元测试

这个程序很小，没什么可测试的。
使用 Python 标准库的 unittest 包测试。

示例：单元测试：tests/test_basics.py：

```
import unittest
from flask import current_app
from app import create_app, db

class BasicsTestCase(unittest.TestCase):
    def setUp(self):  # 创建一个测试环境
        self.app = create_app('testing')
        self.app_context = self.app.app_context()  # 激活上下文
        self.app_context.push()
        db.create_all()  # 创建数据库

    def tearDown(self):
        # 删除数据库和上下文
        db.session.remove()
        db.drop_all()
        self.app_context.pop()

    def test_app_exists(self):  # 测试 app 实例是否存在
        self.assertFalse(current_app is None)

    def test_app_is_testing(self):  # 程序是否在测试配置环境运行
        self.assertTrue(current_app.config['TESTING'])
```

要把 tests 目录当做包，需要添加 tests/__init__.py 文件，可以为空，unittest 会扫描所有模块并查找测试。

为了运行单元测试，可以在 manage.py 中添加一个自定义命令。
示例：启动单元测试命令：manage.py：

```
@manager.command
def test():
    """Run the unit tests."""
    import unittest
    tests = unittest.TestLoader().discover('tests')
    unittest.TextTestRunner(verbosity=2).run(tests)
```

manager.command 装饰器装饰的函数名就是命令名，函数的文档字符串会显示在帮助消息中。test()函数的定义体中调用了 unittest 包提供的测试运行函数。

单元测试可使用下面的命令运行：

```
(venv) $ python manage.py test
test_app_exists (test_basics.BasicsTestCase) ... ok
test_app_is_testing (test_basics.BasicsTestCase) ... ok
----------------------------------------------------------------------
Ran 2 tests in 1.263s
OK
```

✧ 7.7 创建数据库

重组后的程序和单脚本版本使用不同的数据库。从环境变量读取数据库 URL，或默认的 SQLite 数据库。

不管从哪里获取数据库 URL，都要在新数据库中创建数据表。如果使用 Flask-Migrate 跟踪迁移，可使用如下命令创建数据表或者升级到最新修订版本：

```
(venv) $ python manage.py db upgrade
```