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

✧ 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 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'))
```

✧ 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。

✧ 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。

✧ 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。

✧ 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。

✧ 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。

✧ 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。

✧ 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。

✧ 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：

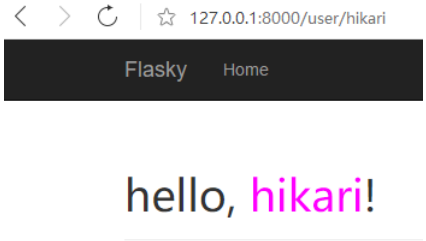
```
from flask import Flask, render_template
from flask_bootstrap import Bootstrap
```

```
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 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 %}
```

✧ 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

✧ 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

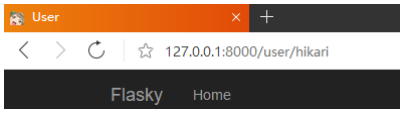
Flask 默认在程序根目录的 static 子目录寻找静态文件。

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：



✧ 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。

✧ 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'
```

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。

✧ 大型 Flask 程序项目结构

flasky/			
	app/		
		templates/	
		static/	
		main/	
			__init__.py
			errors.py

			forms.py
			views.py
		__init__.py	
		email.py	
		models.py	
	migrations/		
	tests/		
		__init__.py	
		test*.py	
	venv/		
	requirements.txt		
	config.py		
	manage.py		

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。

✧ 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。