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：

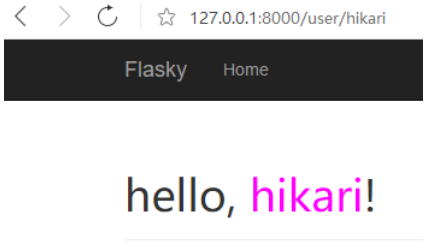
```
from flask import Flask, render_template
from flask_bootstrap import Bootstrap
```

```
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

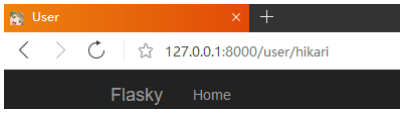
✧ 3.5 静态文件

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 %}
```

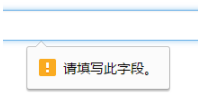
✧ 4.4 在视图函数处理表单

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)
```

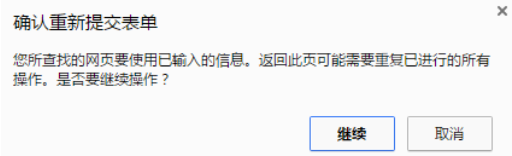
index 视图可以 GET 和 POST 请求。第 1 次访问，服务器收到没有表单数据的 GET 请求，validate_on_submit()返回 False，name 为 None；模板执行 else 语句，显示 Stranger；输入名字后 name 获取数据，传给模板显示：

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

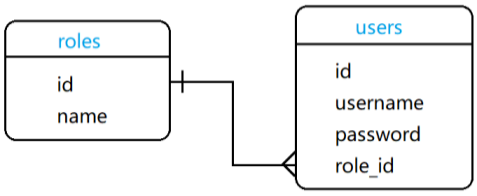
数据库按照一定规则保存数据，程序发起查询取回所需的数据。Web 程序最常用基于关系模型的数据库，也称为 SQL 数据库，因为它们使用结构化查询语言。不过最近几年文档数据库和键值对数据库成了流行的替代选择，这两种合称

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：



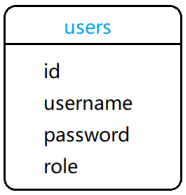
roles 表可用的用户角色，每种角色都有唯一 id；users 表是用户列表，每个用户也都有唯一 id，还有一个外键 role_id 引用 roles 表的 id，指定每个用户角色。

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.environ.get('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
```

20180507

实战：社交博客程序

第 8 章 用户认证

最常用的认证方法是要求用户提供一个身份证明(用户名或 email)和一个密码。

✧ 8.1 Flask 的认证扩展

Python 优秀的认证包有很多，但没有一个能实现所有功能。

此处使用：

① Flask-Login：管理已登录用户的用户会话；

② Werkzeug：计算密码散列值并进行核对；

③ itsdangerous：生成并核对加密安全令牌。

✧ 8.2 密码安全性

要保证数据库中用户密码的安全，不能存储明文密码，而要存储密码的散列值。计算密码散列值的函数接收密码作为参数，使用一种或多种加密算法转换密码，最终得到和原始密码没有关系的字符序列。

使用 Werkzeug 实现密码散列

Werkzeug 包的 security 模块能很方便地实现密码散列值的计算。

只需两个函数，分别用在注册用户和验证用户阶段。

① `generate_password_hash(password, method='pbkdf2:sha256', salt_length=8)`

原始密码作为输入，返回密码的字符串散列值。

method 和 salt_length 的默认值能满足大多数需求。

② `check_password_hash(pwhash, password)`

参数为散列值和密码。返回 True 表明密码正确。

示例：app/models.py：User 模型添加散列密码字段

```
from werkzeug.security import generate_password_hash, check_password_hash


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，roles 表中的 id
    password_hash = db.Column(db.String(128))  # 散列密码

    # password 属性只可写不可读，因为获取散列值没有意义，无法还原密码
    @property
    def password(self):
        raise AttributeError('password is not a readable attribute')

    @password.setter
    def password(self, pwd):
        self.password_hash = generate_password_hash(pwd)

    def verify_password(self, pwd):  # 校验密码
        return check_password_hash(self.password_hash, pwd)

    def __repr__(self):
        return '<User %r>' % self.username
```

在 shell 中进行测试：

```
(venv) $ python manage.py shell
>>> u = User()
>>> u.password = 'maki'
>>> u.password_hash
'pbkdf2:sha256:50000$sq4UFy9l$77ed2de287d2361ca0da4b912e4d376d26d1a71f6b4d9b0b67b8ab164149aadf'
>>> u.verify_password('maki')
True
>>> u.verify_password('rin')
False
>>> u2 = User()
>>> u2.password = 'maki'
>>> u2.password_hash
'pbkdf2:sha256:50000$QNJbN8eL$1fd34f7b2019c36ce8b9097ff678860703f2e7a2cf3571e3e69d2e9c54debedc'
```

用户 u 和 u2 使用相同的密码，但是密码的散列值也不一样，说明撒盐了？

可以把上述测试写成单元测试，以便于后续测试。在 tests 包中新建一个 test_user_model 模块，测试最近对 User 模型所做的修改。

示例：tests/test_user_model.py：密码散列化测试：

```
import unittest
from app.models import User
# 测试用户模型的密码散列化
```

```python
class UserModelTestCase(unittest.TestCase):
    # test_basics.py 里已经有 setUP 和 tearDown 了，此处不需要，写了报错
    def test_password_setter(self):  # 测试设置密码
        u = User(password='maki')
        self.assertTrue(u.password_hash is not None)

    def test_no_password_getter(self):  # 测试不能获取密码
        u = User(password='maki')
        with self.assertRaises(AttributeError):
            u.password

    def test_password_verification(self):  # 测试密码认证
        u = User(password='maki')
        self.assertTrue(u.verify_password('maki'))
        self.assertFalse(u.verify_password('rin'))

    def test_password_salts_are_random(self):  # 测试是否随机撒盐
        u = User(password='maki')
        u2 = User(password='maki')
        self.assertFalse(u.password_hash == u2.password_hash)
```

✧ 8.3 创建认证蓝本

前一章创建 app 过程移入工厂函数后，使用蓝本在全局作用域定义路由。与用户认证系统相关的路由可在 auth；蓝本定义。不同的功能使用不同的蓝本是保持代码整齐有序的好方法。

① app/auth/__init__.py：创建蓝本

```python
from flask import Blueprint
auth = Blueprint('auth', __name__)
from . import views
```

和之前 main 蓝本一样，auth 也作为一个包，__init__.py 底部引入 auth/view.py 将路由和蓝本关联。

② app/auth/views.py：蓝本中的路由和视图函数

```python
from flask import render_template
from . import auth
@auth.route('/login')
def login():
    return render_template('auth/login.html')
```

指定模板在 app/templates/auth 目录中，避免和 main 蓝本和后续蓝本的模板发生命名冲突。

③ app/__init__.py：添加 auth 蓝本

```
def create_app(config_name):  # 工厂函数，参数为配置名
    # ...
    from .auth import auth as auth_blueprint
    # url_prefix 是可选，蓝本所有路由添加前缀，如/login 变为/auth/login
    app.register_blueprint(auth_blueprint, url_prefix='/auth')
    return app  # 返回创建的程序示例
```

✧ 8.4 使用 Flask-Login 认证用户

Flask-Login 是非常有用的小型扩展，用来管理用户认证系统的认证状态，且不依赖特定的认证机制。

① 用于登录的用户模型

为了使用 Flask-Login，User 模型需要实现几个方法：

1) is_authenticated()：如果用户已经登录，返回 True 否则 False
2) is_active()：如果允许用户登录，返回 True 否则 False；如果要禁用账户，可以返回 False
3) is_anonymous：对普通用户返回 False
4) get_id()：返回用户的唯一标识符，使用 Unicode 编码字符串

这 4 个方法可以直接在 User 类中实现，但 Flask-Login 提供 UserMixin 类，包含这些方法的默认实现，满足大多数需求，可以用 User 继承 UserMixin：

app/models.py：修改 User 模型，支持用户登录

```
from flask_login import UserMixin
class User(UserMixin, db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)  # 主键
    email = db.Column(db.String(64), unique=True, index=True)  # 添加邮箱字段
    username = db.Column(db.String(64), unique=True, index=True)  # 用户名
    password_hash = db.Column(db.String(128))  # 密码的散列值
    role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))  # 外键
```

app/__init__.py：初始化 Flask-Login

```
from flask_login import LoginManager
login_manager = LoginManager()
# 用户会话保护等级，可以设为 None，'basic'，'strong'，防止用户会话遭篡改
# 'strong'：记录客户端 ip 和浏览器 User-Agent，如果发生异动则登出用户
login_manager.session_protection = 'strong'
# auth 蓝本的 login 视图
login_manager.login_view = 'auth.login'
def create_app(config_name):  # 工厂函数，参数为配置名
    # ...
    login_manager.init_app(app)
    # ...
```

Flask-Login 要求实现一个回调函数，使用指定的标识符加载用户：

app/models.py：加载用户的回调函数

```
@login_manager.user_loader
def load_user(user_id):
    # user_id 为 Unicode 字符串表示的用户标识符，如果有此用户返回用户对象；否则返回 None
    return User.query.get(int(user_id))
```

② 保护路由

为了保护路由只让认证用户访问，Flask-Login 提供 login_required 装饰器：

```
from flask_login import login_required
@app.route('/secret')
@login_required
def secret():
    pass
```

未认证用户访问此路由，Flask-Login 会拦截请求，把用户发往登录页面。

③ 登录表单

包含输入电子邮箱的文本字段、密码字段、保持登录复选框、提交按钮。

app/auth/forms.py：

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired, Length, Email

class LoginForm(FlaskForm):
    email = StringField('邮箱', validators=[ # 数据必须，长度限制，email 验证函数
                    DataRequired(), Length(5, 64), Email()])
    password = PasswordField('密码', validators=[DataRequired()])
    remember_me = BooleanField('记住我')
    submit = SubmitField('登录')
```

登录页面模板为 app/templates/auth/login.html；

app/templates/base.html：导航条右边根据是否登录添加登录退出按钮

```
{# 登录退出按钮 #}
<ul class="nav navbar-nav navbar-right">
    {% if current_user.is_authenticated %}
        <li><a href="{{ url_for('auth.logout') }}">退出</a></li>
    {% else %}
        <li><a href="{{ url_for('auth.login') }}">登录</a></li>
    {% endif %}
</ul>
```

变量 current_user 由 Flask-Login 定义，在视图函数和模板自动可用，值是当前登录用户；如果用户未登录则是一个匿名用户代理对象，is_authenticated 为 False

④ 登入用户

app/auth/views.py：登录视图函数 login()

```
from flask import render_template, redirect, request, url_for, flash
from flask_login import login_user, logout_user, login_required
from . import auth
from ..models import User
from .forms import LoginForm

@auth.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        # 根据输入邮箱查询用户
        user = User.query.filter_by(email=form.email.data.lower()).first()
        # 如果用户存在且密码正确
        if user and user.verify_password(form.password.data):
            login_user(user, form.remember_me.data)
            nxt = request.args.get('next')
            # 没有 next 重定向到首页
            if nxt is None or not nxt.startswith('/'):
                nxt = url_for('main.index')
            return redirect(nxt)
        flash('用户名或密码错误！') # 认证失败
    return render_template('auth/login.html', form=form)
```

此 LoginForm 与之前的 NameForm 类似，当表单被提交，是 POST 请求，验证表单数据，尝试登录用户；否则是 GET 请求，直接渲染模板，显示表单。

如果验证成功，调用 Flask-Login 的 login_user()函数，在用户会话把用户标记为已登录。第 2 个参数 remember 默认为 False，关闭浏览器会话就过期，下次访问要重新登录；True 则会在浏览器写入一个长期有效的 cookie。

app/templates/auth/login.html：登录表单

```
{% extends "base.html" %} {# 继承于 base.html 模板 #}
{% import "bootstrap/wtf.html" as wtf %}
{% block title %}hikari app - Login{% endblock %}
{% block page_content %}
<div class="page-header"><h1>登录</h1></div>
<div class="col-md-4">
    {{ wtf.quick_form(form) }} {# 使用 Bootstrap 默认样式渲染表单 #}
</div>
{% endblock %}
```

⑤ 登出用户

app/auth/views.py：logout 登出视图

```
@auth.route('/logout')
@login_required
def logout(): # 登出用户
    logout_user() # 删除并重设用户会话
    flash('你已经退出...') # 显示登出消息
    return redirect(url_for('main.index')) # 重定向至首页
```

⑥ 测试登录

app/templates/index.html：为已登录用户显示欢迎消息

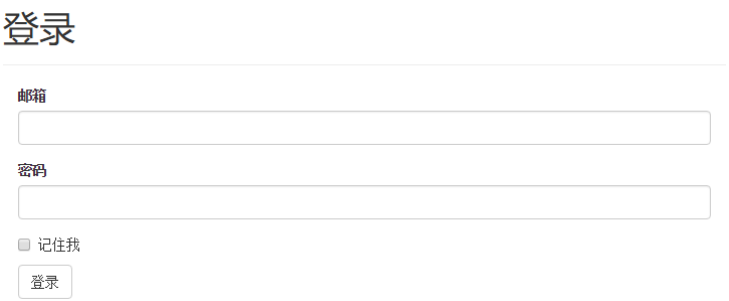
```
{% extends "base.html"%} {# 继承于 base.html 模板 #}
{% block title %}hikari app{% endblock%}
{% block page_content %}
<div class="page-header">
    <h1>Hello, {% if current_user.is_authenticated %}{{ current_user.username }}{%
else %}Stranger{% endif %}!</h1></div>
{% endblock %}
```

因为还未添加用户注册功能，新用户可在 shell 中注册：

```
(venv) $ python manage.py shell
>>> u=User(email='hikari@example.com',username='hikari',password='123456')
>>> db.session.add(u)
>>> db.session.commit()
>>> exit()
```

python manage.py runserver 启动服务器，默认 http://127.0.0.1:5000

显示 stranger，点击右上角登录，跳转到 http://127.0.0.1:5000/auth/login



登录后，跳转至首页，显示 Hello, hikari !

点击退出，显示 flash 消息：你已经退出...

✧ 8.5 注册新用户

① 用户注册表单

app/auth/forms.py：用户注册表单类

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired, Length, Email, Regexp, EqualTo
from wtforms import ValidationError
from ..models import User
```

```
class RegistrationForm(FlaskForm):
    email = StringField('邮箱', validators=[DataRequired(), Length(5, 64),
Email()])
    username = StringField('用户名', validators=[DataRequired(), Length(2, 20),
Regexp('^[A-Za-z][A-Za-z0-9_]{1,19}$', 0, '用户名只能是字母数字下划线！')])
    password = PasswordField('输入密码', validators=[DataRequired(),
EqualTo('password2', message='密码不一致！')])
    password2 = PasswordField('确认密码', validators=[DataRequired()])
    submit = SubmitField('注册')

    # 自定义验证函数，如果表单类定义了 validate_开头且跟着字段名的方法
    # 此方法与常规验证函数一起调用
    def validate_email(self, field):
        if User.query.filter_by(email=field.data.lower()).first():
            raise ValidationError('该邮箱已被注册！')

    def validate_username(self, field):
        if User.query.filter_by(username=field.data.lower()).first():
            raise ValidationError('用户名已经存在！')
```

登录页面需要有一个指向注册页面的链接

app/templates/auth/login.html：链接到注册页面

```
<div class="col-md-4">
    {{ wtf.quick_form(form) }} {# 使用 Bootstrap 默认样式渲染表单 #}<br>
    <p>新用户？ <a href="{{ url_for('auth.register') }}">点击注册</a></p></div>
```

// 不应该在 base.html 未登录的用户都显示注册按钮?

② 注册新用户

app/auth/views.py：用户注册视图

```
from .. import db
from .forms import RegistrationForm

@auth.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        user = User(email=form.email.data.lower(),
username=form.username.data.lower(), password=form.password.data)
        db.session.add(user)
        flash('注册成功！现在可以登录了！')
        return redirect(url_for('auth.login'))
    return render_template('auth/register.html', form=form)
```

app/templates/auth/register.html：注册模板和登录模板类似

```
{% extends "base.html" %} {# 继承于 base.html 模板 #}
{% import "bootstrap/wtf.html" as wtf %}
{% block title %}hikari app - Register{% endblock %}
{% block page_content %}
<div class="page-header"><h1>注册</h1></div>
<div class="col-md-4">{{ wtf.quick_form(form) }}</div>
{% endblock %}
```

✧ 8.6 确认账户

有时需要确认用户提供的信息是否正确，一般通过邮件与用户联系。
通常用户注册后，会发送一封确认邮件，新账户暂时标为待确认状态，用户按说明操作后，才能证明自己信息正确。账户确认过程中，往往要求用户点击一个包含确认令牌的特殊 URL 链接。

① 使用 itsdangerous 生成确认令牌

确认链接最简单的格式是 http://www.example.com/auth/confirm/<*id* >形式的 URL，id 是数据库分配给用户的数字 id。用户点击链接后，处理这个路由的视图函数就将收到的用户 id 作为参数进行确认，然后将用户状态更新为已确认。

但此方式显然不是很安全，只要用户能判断确认链接的格式，就可以随意指定 URL 中的数字，从而确认任意账户。解决方法是把 URL 中的 id 换成将相同信息安全加密后得到的令牌。

第 4 章的用户会话，Flask 使用加密的签名 cookie 保护用户会话，防止被篡改。这种安全的 cookie 使用 itsdangerous 包签名。同样也可用于确认令牌上。

使用 itsdangerous 包生成包含用户 id 的安全令牌：

```
(venv) $ python manage.py shell
>>> from manage import app
>>> from itsdangerous import TimedJSONWebSignatureSerializer as Serializer
>>> s = Serializer(app.config['SECRET_KEY'], expires_in=3600)
>>> token = s.dumps({'confirm':23})
>>> token
b'eyJhbGciOiJIUzI1NiIsImlhdCI6MTUyNTY4MjgxOSwiZXhwIjoxNTI1Njg2NDE5fQ.eyJjb25maXJtIjoyM30.
hx1WUKVdrJts3fx39t6y9wr59rgpB8ryBsu5a9byWuo'
>>> data = s.loads(token)
>>> data
{'confirm': 23}
```

itsdangerous 有多种生成令牌的方法。其中 TimedJSONWebSignatureSerializer 类生成具有过期时间的 JSON Web 签名(JSON Web Signatures，JWS)。构造函数第 1 个参数是一个密钥，在 Flask 程序中使用 SECRET_KEY 设置(config.py)。expires_in 参数设置令牌的过期时间，单位为秒。

dumps()方法为指定数据生成一个加密签名，然后再对数据和签名进行序列化，生成令牌字符串。
loads()方法，唯一的参数是令牌字符串。检验签名和过期时间，如果通过，返回原始数据。如果提供的令牌不正确或过期了，则抛出异常。

可以将这种生成和检验令牌的功能添加到 User 模型：
app/models.py：确认用户账户

```
from . import db
from flask_login import UserMixin
from itsdangerous import TimedJSONWebSignatureSerializer as Slzer
from flask import current_app

class User(UserMixin, db.Model):
    # ...
    confirmed = db.Column(db.Boolean, default=False)

    def generate_confirmation_token(self, expiration=3600):
        # 生成确认令牌字符串，默认有效时间 1h
        s = Slzer(current_app.config['SECRET_KEY'], expiration)
        return s.dumps({'confirm': self.id}).decode('utf-8')

    def confirm(self, token):
        # 检验令牌，通过则 confirmed 字段设为 True
        s = Slzer(current_app.config['SECRET_KEY'])
        try:
            data = s.loads(token.encode('utf-8'))
        except:
            return False
        if data.get('confirm') != self.id:
            return False
        self.confirmed = True
        db.session.add(self)
        return True
```

由于 confirm()检查的是当前用户的令牌，所以恶意用户无法帮别人确认。

User 模型新加了字段，需要数据库迁移。

tests/ test_user_model.py：确认令牌测试

```
def test_valid_confirmation_token(self): # 测试正确的确认令牌
    u = User(password='maki')
    db.session.add(u)
    db.session.commit()
    token = u.generate_confirmation_token()
    self.assertTrue(u.confirm(token))
```

```
def test_invalid_confirmation_token(self): # 测试用别人的确认令牌
    u1 = User(password='maki')
    u2 = User(password='rin')
    db.session.add(u1)
    db.session.add(u2)
    db.session.commit() # 要 commit(), 不然结果为 True, assertFalse 抛出异常?
    token = u1.generate_confirmation_token()
    self.assertFalse(u2.confirm(token))

def test_expired_confirmation_token(self): # 测试令牌过时
    u = User(password='maki')
    db.session.add(u)
    db.session.commit()
    token = u.generate_confirmation_token(1)
    time.sleep(2)
    self.assertFalse(u.confirm(token))
```

/* 貌似测试在 setUp 和 tearDown 之间，测试时，COMMIT_ON_TEARDOWN 就没效果了，需要手动 commit? 把 3 个 commit 都注释，结果 u2.confirm(token)返回 True (why?)，抛出 AssertionError: True is not false 异常? */

20180508

② 发送确认邮件

现在的/register 路由在将新用户添加到数据库后重定向到/index(不是/login 吗?)。在重定向前，需要发送确认邮件。

```
from ..email import send_mail
@auth.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        user = User(email=form.email.data.lower(),
username=form.username.data.lower(), password=form.password.data)
        db.session.add(user)
        # 即使自动提交,此处也要 commit,因为要获取 id 用于生成确认令牌,不能延后提交
        db.session.commit()
        token = user.generate_confirmation_token()
        send_mail(user.email, '激活你的账号',
                  'auth/email/confirm', user=user, token=token)
        flash('激活邮件已发送至你的邮箱...')
        return redirect(url_for('auth.login'))
    return render_template('auth/register.html', form=form)
```

认证的邮件模板也有 txt 和 HTML 两个版本，放在 app/templates/auth/email 目录，为 confirm.txt 和 confirm.html。

app/templates/auth/email/confirm.html：确认邮件的 HTML 格式

```
<p>你好，{{ user.username }}!</p>
<p>欢迎注册<b>hikari app</b>!</p>
<p><a href="{{ url_for('auth.confirm', token=token, _external=True) }}">点击此处
</a>激活你的账号。</p>
<p>你也可以将下面链接复制到浏览器的地址栏:</p>
<p>{{ url_for('auth.confirm', token=token, _external=True) }}</p>
<p>感谢注册!</p>
<p>from hikari</p>
<p><small>注: hikari 是不会看对这封邮件的回复的 !</small></p>
```

默认 url_for()生成相对 URL，如 url_for('auth.confirm', token='abc') 返回 '/auth/confirm/abc'。相对 URL 在网页的上下文中可以正常使用，因为通过添加当前页面的主机名和端口号，浏览器会将其转换成绝对 URL。但通过电子邮件发送 URL 时，并没有这种上下文。_external=True 参数要求程序生成完整的 URL，其中包含协议、主机名和端口。

app/auth/views.py：确认用户账户的 confirm 视图

```
from flask_login import current_user
@auth.route('/confirm/<token>')
@login_required
def confirm(token):
    if current_user.confirmed: # 用户已经确认过,重定向到主页
        return redirect(url_for('main.index'))
    if current_user.confirm(token):
        # db.session.commit()
        flash('你的账号已通过认证！')
    else:
        flash('认证链接无效或已经过期！')
    return redirect(url_for('main.index'))
```

login_required 装饰器保护此路由，用户点击确认邮件中的链接后，要先登录，然后才能执行此视图函数。

可以决定用户确认账户之前能做哪些操作。比如允许未确认的用户登录，但只显示一个页面，要求用户在获取权限之前先确认账户。

可使用 Flask 提供的 before_request 钩子完成。对于蓝本，before_request 钩子只能应用到属于蓝本的请求上。若想在蓝本中使用针对程序全局请求的钩子，必须使用 before_app_request 装饰器。

app/auth/views.py：过滤未确认的账户

```
@auth.before_app_request
def before_request():
```

```
    # 用户已登录,未确认,请求端点不在蓝本,请求被拦截,重定向至/unconfirmed
    if current_user.is_authenticated \
            and not current_user.confirmed \
            and request.endpoint \
            and request.blueprint != 'auth' \
            and request.endpoint != 'static':
        return redirect(url_for('auth.unconfirmed'))

@auth.route('/unconfirmed')
def unconfirmed():
    # 匿名或已经确认用户,没必要确认,重定向至首页
    if current_user.is_anonymous or current_user.confirmed:
        return redirect(url_for('main.index'))
    # 显示一个确认账户相关信息的页面
    return render_template('auth/unconfirmed.html')
```

同时满足以下 3 个条件时，before_app_request 处理程序会拦截请求。
1) 用户已登录(current_user.is_authenticated()返回 True)；
2) 用户的账户还未确认；
3) 请求的端点(request.endpoint 获取)不在 auth 蓝本中。访问认证路由要获取权限，因为其作用是让用户确认账户或执行其他账户管理操作。

注：如果 before_request 或 before_app_request 的回调返回响应或重定向，Flask 会直接将其发送至客户端，而不会调用请求的视图函数。

app/templates/auth/unconfirmed.html：显示给未确认用户的页面

```
{% extends "base.html" %}
{% block title %}hikari app - Confirm your account{% endblock %}
{% block page_content %}
<div class="page-header">
    <h1>你好，{{ current_user.username }}!</h1>
    <h3>你的账号还没激活!</h3>
    <p>请前往你的邮箱，点击激活邮件里的链接激活你的账号，之后你可以畅游此网站。</p>
    <p>没收到邮件?<a href="{{ url_for('auth.resend_confirmation') }}">点击重新发送
</a></p></div>
{% endblock %}
```

该页面主要给未确认用户显示提示信息，还提供一个链接，用于请求重新发送确认邮件，以防之前邮件丢失或确认令牌过期。

app/auth/views.py：重新发送确认邮件 resend_confirmation 视图

```
@auth.route('/confirm')
@login_required
def resend_confirmation():
    # 重新发送确认邮件,需要用户已登录
```

```
    token = current_user.generate_confirmation_token()
    send_mail(current_user.email, '激活你的账号',
              'auth/email/confirm', user=current_user, token=token)
    flash('新的激活邮件已发送至你的邮箱...')
    return redirect(url_for('main.index'))
```

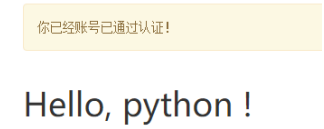
测试：

注册后：

// 如果网站不是很有名，用户根本没有耐心这么搞...

163 邮箱信件：

点击 click here 链接：



✧ 8.7 管理账户

用户有时可能需要修改账户信息，如：

① 修改密码

安全意识强的用户可能会定期修改密码。如果用户处于登录状态，可以放心显示一个表单，要求用户输入旧密码和新密码。

1) app/auth/forms.py：修改密码表单类

```
class ChangePasswordForm(FlaskForm):
    # 修改密码表单类
    old_password = PasswordField('输入原密码', validators=[DataRequired()])
    password = PasswordField('设置新密码', validators=[DataRequired(),
EqualTo('password2', message='密码不一致')])
    password2 = PasswordField('确认新密码', validators=[DataRequired()])
    submit = SubmitField('确认')
```

2) app/auth/views.py：修改密码视图函数

```
from .forms import ChangePasswordForm

@auth.route('/change_password', methods=['GET', 'POST'])
@login_required
def change_password():
    form = ChangePasswordForm()
    if form.validate_on_submit():
        # 修改密码需要登录; 原密码输入正确,设置新密码
        if current_user.verify_password(form.old_password.data):
            current_user.password = form.password.data
            db.session.add(current_user)
            # db.session.commit()
            flash('密码修改成功! ')
            return redirect(url_for('main.index'))
        else:
            flash('原密码输入有误! ')
    return render_template('auth/change_password.html', form=form)
```

3) app/templates/auth/change_password.html：

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}
{% block title %}hikari app - Change Password{% endblock %}
{% block page_content %}
<div class="page-header"><h1>修改密码</h1></div>
<div class="col-md-4">{{ wtf.quick_form(form) }}</div>
{% endblock %}
```

需要添加个人中心按钮，其中有修改密码、邮箱、退出等操作。

4) app/templates/base.html：

```
<ul class="nav navbar-nav navbar-right">
    {# 如果登录,显示个人中心;否则显示登录按钮 #}
    {% if current_user.is_authenticated %}
    <li class="dropdown">
        <a href="#" class="dropdown-toggle" data-toggle="dropdown">
            {{ current_user.username }}<b class="caret"></b></a>
        <ul class="dropdown-menu">
            <li><a href="{{ url_for('auth.change_password') }}">修改密码</a></li>
            <li><a href="{{ url_for('auth.logout') }}">登出</a></li></ul></li>
    {% else %}
    <li><a href="{{ url_for('auth.login') }}">登录</a></li>
    {% endif %}
</ul>
```

效果：

② 重设密码

为了避免用户忘记密码无法登录的情况，提供重设密码功能。类似于用户确认时用的令牌，用户请求重设密码，向用户邮箱发送包含重设令牌的邮件。用户点击链接验证后，显示一个用于输入新密码的表单。

1) app/models.py：User 模型添加生成重置令牌

```
class User(UserMixin, db.Model):
    # ...
    def generate_reset_token(self, expiration=3600):
        # 生成重置令牌字符串，默认有效时间 1h
        s = Slzer(current_app.config['SECRET_KEY'], expiration)
        return s.dumps({'reset': self.id}).decode('utf-8')

    @staticmethod
    def reset_password(token, new_password):
        s = Slzer(current_app.config['SECRET_KEY'])
        try:
            data = s.loads(token.encode('utf-8'))
        except:
            return False
        # 根据令牌获取用户 id,查询该用户
        user = User.query.get(data.get('reset'))
        if user is None: # 用户不存在
            return False
        # 用户存在,重置密码
        user.password = new_password
        db.session.add(user)
        return True
```

2) app/auth/forms.py：重设密码表单类：

```
class PasswordResetRequestForm(FlaskForm):
    # 重设密码请求表单
    email = StringField('邮箱', validators=[DataRequired(), Length(5,
64),Email()])
    submit = SubmitField('重设密码')

class PasswordResetForm(FlaskForm):
    # 重设密码表单
```

```
    password = PasswordField('设置新密码', validators=[DataRequired(),
EqualTo('password2', message='密码不一致')])
    password2 = PasswordField('确认新密码', validators=[DataRequired()])
    submit = SubmitField('确认')
```

3) app/auth/views.py：重设密码视图函数

```
@auth.route('/reset', methods=['GET', 'POST'])
def password_reset_request():
    # 匿名用户什么也不做,重定向到首页
    if not current_user.is_anonymous:
        return redirect(url_for('main.index'))
    form = PasswordResetRequestForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data.lower()).first()
        if user: # 生成重置令牌,发送邮件
            token = user.generate_reset_token()
            send_mail(user.email, '重置你的密码', 'auth/email/reset_password',
user=user, token=token, next=request.args.get('next'))
        flash('重置密码链接已发送至你的邮箱。')
        return redirect(url_for('auth.login'))
    return render_template('auth/reset_password.html', form=form)

@auth.route('/reset/<token>', methods=['GET', 'POST'])
def password_reset(token):
    # 发给用户的邮件的链接,点击出现此视图页面
    if not current_user.is_anonymous:
        return redirect(url_for('main.index'))
    form = PasswordResetForm()
    if form.validate_on_submit():
        if User.reset_password(token, form.password.data):
            # db.session.commit()
            flash('重设密码成功！')
            return redirect(url_for('auth.login'))
        else:
            return redirect(url_for('main.index'))
    return render_template('auth/reset_password.html', form=form)
```

4) app/templates/auth/email/reset_password.html：重设密码 HTML 格式邮件

```
<p>你好，{{ user.username }}!</p>
<p><a href="{{ url_for('auth.password_reset', token=token, _external=True) }}">点
击此处</a>重置你的密码。</p>
<p>你也可以将下面链接复制到浏览器的地址栏:</p>
<p>{{ url_for('auth.password_reset', token=token, _external=True) }}</p>
<p>如果不是你的操作或不想重置密码，忽略此邮件。</p>
```

```
<p>from hikari</p>
<p><small>注: hikari 是不会看对这封邮件的回复的 !</small></p>
```

5) app/templates/auth/reset_password.html：重设密码页面

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}
{% block title %}hikari app - Password Reset{% endblock %}
{% block page_content %}
<div class="page-header"><h1>重设密码</h1></div>
<div class="col-md-4">{{ wtf.quick_form(form) }}</div>
{% endblock %}
```

由于用户登录时可以直接修改密码，没必要重设。所以重设密码的链接放在登录页面比较好，比如登录按钮下面。

6) app/templates/auth/login.html：重设密码的按钮

```
<div class="col-md-4">
    {{ wtf.quick_form(form) }}<br>
    <p><a href="{{ url_for('auth.password_reset_request') }}">忘记密码?</a></p>
    <p>新用户? <a href="{{ url_for('auth.register') }}">点击注册</a></p>
</div>
```

效果：

③ 修改电子邮箱

旧邮箱验证，点击链接设置新邮箱，给新邮箱发送包含令牌的验证邮件，确认后更新用户对象。服务器收到令牌之前，可以将新邮箱保存到待定邮箱字段。

1) app/models.py：User 模型添加生成令牌

```
class User(UserMixin, db.Model):
    # ...
    def generate_email_change_token(self, new_email, expiration=3600):
        # 生成修改邮箱令牌字符串，默认有效时间 1h
        s = Slzer(current_app.config['SECRET_KEY'], expiration)
        return s.dumps(
            {'change_email': self.id, 'new_email': new_email}).decode('utf-8')
    def change_email(self, token):
        s = Slzer(current_app.config['SECRET_KEY'])
        try:
```

```
            data = s.loads(token.encode('utf-8'))
        except:
            return False
        # 令牌不属于当前用户,新邮箱为None,新邮箱已经存在都修改失败
        if data.get('change_email') != self.id:
            return False
        new_email = data.get('new_email')
        if new_email is None:
            return False
        if self.query.filter_by(email=new_email).first() is not None:
            return False
        self.email = new_email
        db.session.add(self)
        return True
```

2) app/auth/forms.py：修改邮箱的表单类

```
class ChangeEmailForm(FlaskForm):
    # 修改邮箱表单类
    email = StringField('新邮箱', validators=[
                        DataRequired(), Length(5, 64), Email()])
    password = PasswordField('请输入密码', validators=[DataRequired()])
    submit = SubmitField('确认')

    def validate_email(self, field):
        # 如果输入邮箱已被注册,抛出异常
        if User.query.filter_by(email=field.data.lower()).first():
            raise ValidationError('该邮箱已被注册！')
```

3) app/auth/views.py：修改邮箱的视图函数

```
@auth.route('/change_email', methods=['GET', 'POST'])
@login_required
def change_email_request():
    form = ChangeEmailForm()
    if form.validate_on_submit():
        if current_user.verify_password(form.password.data):
            new_email = form.email.data.lower()
            token = current_user.generate_email_change_token(new_email)
            send_mail(new_email, '修改你的绑定邮箱', 'auth/email/change_email',
user=current_user, token=token)
            flash('修改绑定邮箱的链接已经发送至你的邮箱。')
            return redirect(url_for('main.index'))
        else:
            flash('邮箱或密码有误！')
    return render_template("auth/change_email.html", form=form)
```

```
@auth.route('/change_email/<token>')
@login_required
def change_email(token):
    if current_user.change_email(token):
        # db.session.commit()
        flash('修改邮箱成功！')
    else:
        flash('无效请求！')
    return redirect(url_for('main.index'))
```

4) app/templates/auth/email/change_email.html：修改邮箱的 HTML 格式邮件

```
<p>你好，{{ user.username }}!</p>
<p><a href="{{ url_for('auth.change_email', token=token, _external=True) }}">点击此处</a>修改绑定邮箱。</p>
<p>你也可以将下面链接复制到浏览器的地址栏:</p>
<p>{{ url_for('auth.change_email', token=token, _external=True) }}</p>
<p>from hikari</p>
<p><small>注：hikari 是不会看对这封邮件的回复的 !</small></p>
```

5) app/templates/auth/change_email.html：修改邮箱页面

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}
{% block title %}hikari app - Change Email{% endblock %}
{% block page_content %}
<div class="page-header"><h1>修改绑定邮箱</h1></div>
<div class="col-md-4">{{ wtf.quick_form(form) }}</div>
{% endblock %}
```

6) app/templates/base.html：在个人中心添加修改邮箱的按钮

```
<ul class="dropdown-menu">
    <li><a href="{{ url_for('auth.change_password') }}">修改密码</a></li>
    <li><a href="{{ url_for('auth.change_email_request') }}">修改邮箱</a></li>
     <li><a href="{{ url_for('auth.logout') }}">登出</a></li></ul>
```

/* 后两个套路都差不多，需要生成令牌，就在 User 模型添加生成令牌和确认的函数。服务器收到请求，判断是否需要发送邮件给用户；用户收到邮件点击链接完成认证就可以修改，然后将修改保存到数据库。最后为每个功能在合适的地方添加一个按钮。相同的工作基本就是体力活了。
所以到后面狗书都不写了，直接说自己看源码去吧... */

20180509

第 9 章　用户角色

Web 程序用户一般具有不同地位，分别拥有不同权限。简单的程序可能只需要普通用户和管理员两个角色，这样只需要在 User 模型添加 is_admin 布尔值字段。复杂的程序可能需要细分多个不同等级的角色。

✧ 9.1 角色在数据库中的表示

app/models.py：Role 模型，角色权限

```
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)  # 角色名
    # 只有一个角色 default 字段要设为 True,其他为 False
    default = db.Column(db.Boolean,default=False,index=True)
    permissions = db.Column(db.Integer)
    users = db.relationship('User', backref='role', lazy='dynamic')
```

permissions 字段是一个整数，表示位标志。各操作都对应一个位位置，能执行某项操作的角色，其位会被设为 1。

此 Flasky 程序的权限

操作	位值
关注用户	0b00000001 (0x01)
在他人的文章发表评论	0b00000010 (0x02)
写文章	0b00000100 (0x04)
管理他人发表的评论	0b00001000 (0x08)
管理员权限	0b00010000 (0x10)

暂时使用 8 位中的 5 位，剩余 3 位备用。

app/models.py：权限常量

```
class Permission:  # 用户权限常量
    FOLLOW = 1
    COMMENT = 2
    WRITE = 4
    MODERATE = 8
    ADMIN = 16
```

用户角色及权限

用户角色	权限	说明
匿名	0b00000000 (0x00)	未登录用户，只能浏览
用户	0b00000011 (0x03)	能关注别人、发表评论
协助管理员	0b00001111 (0x0f)	增加发文章和管理评论权限
管理员	0b11111111 (0xff)	所有权限，包括修改其他用户的权限

使用权限组织角色，以后添加新角色只需要使用不同的权限组合即可。

app/models.py：数据库创建角色

```
class Role(db.Model):
    # ...
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        if self.permissions is None:
            self.permissions = 0

    @staticmethod
    def insert_roles():
        # 角色字符串为键,权限常量列表为值
        roles = {
            'User': [Permission.FOLLOW, Permission.COMMENT],
            'Moderator': [Permission.FOLLOW, Permission.COMMENT, Permission.WRITE,
Permission.MODERATE],
            'Admin': [Permission.FOLLOW, Permission.COMMENT, Permission.WRITE,
Permission.MODERATE, Permission.ADMIN],
        }
        default_role = 'User'
        for k, v in roles.items():
            # 对于每个角色,如果数据库不存在添加之;每个角色重设权限
            role = Role.query.filter_by(name=k).first()
            if role is None:
                role = Role(name=k)
            role.reset_permissions()
            for i in v:
                role.add_permission(i)
            # 普通用户角色的 default 为 True
            role.default = (role.name == default_role)
            db.session.add(role)
        db.session.commit()

    def has_permission(self, p):
        # 指定权限位是不是 1?
        return (self.permissions & p) == p

    def add_permission(self, p):
        # 如果没有此权限,添加之
        if not self.has_permission(p):
            self.permissions += p

    def remove_permission(self, p):
        # 如果有此权限,删除之
        if self.has_permission(p):
```

```
            self.permissions -= p

    def reset_permissions(self):
        # 权限重置为 0
        self.permissions = 0
```

想要添加新角色或修改权限，只需要修改 roles 字典。匿名角色不需要在数据库表示，这个角色作用就是为了表示不在数据库的角色。

// 感觉狗书上写的还比较简洁，最新 GitHub 上的代码看起来略麻烦...

把角色写入数据库，使用 shell 会话。

// 数据库迁移报错了...结果删除了，重新 init、migrate、upgrade 就没问题?

```
(venv) $ python manage.py shell
>>> Role.insert_roles()
>>> Role.query.all()
[<Role 'User'>, <Role 'Moderator'>, <Role 'Admin'>]
```

✧ 9.2 赋予角色

大部分用户注册赋予的角色都默认为普通用户，唯一例外的管理员。管理员一开始就要赋予 admin 角色，由保存在 config.py 的变量 FLASKY_ADMIN 的邮箱确定，只要这个邮箱出现在注册请求中，就被赋予 admin 角色。

app/models.py：定义默认的用户角色

```
class User(UserMixin, db.Model):
    # ...
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        if self.role is None:
            # 是管理员的邮箱,该用户设为 admin 角色
            if self.email == current_app.config['FLASKY_ADMIN']:
                self.role = Role.query.filter_by(name='Admin').first()
            # 如果该用户还是没有角色,设为默认普通用户
            if self.role is None:
                self.role = Role.query.filter_by(default=True).first()
```

✧ 9.3 角色验证

为了简化角色和权限的实现过程，可以在 User 模型添加一个辅助方法检查是否有指定权限。

app/models.py：检查用户是否有指定权限

```
class User(UserMixin, db.Model):
    # ...
    def can(self, p):
        # 用户不为 None 且拥有权限 p
        return self.role is not None and self.role.has_permission(p)
```

```
    def is_admin(self):
        # 检查用户是不是管理员
        return self.can(Permission.ADMIN)

from flask_login import AnonymousUserMixin

class AnonymousUser(AnonymousUserMixin):
    def can(self, p):
        return False
    def is_admin(self):
        return False
login_manager.anonymous_user = AnonymousUser
```

新定义的匿名用户类 AnonymousUser，也实现了这两个方法。current_user 设为用户未登录的值。这样不用先检查用户是否登录，能自由调用 current_user.can() 和 current_user.is_admin()

可以自定义装饰器使某些视图只对特定权限的用户开放。

app/decorators.py：检查用户权限的自定义装饰器

```
from functools import wraps
from flask import abort
from flask_login import current_user
from .models import Permission

def permission_required(p): # 需要权限 p 的装饰器
    # 带参数的装饰器，需要三层嵌套
    def decorator(f):
        # 使用装饰器后,f 指向内层函数 wrap,__name__变为 wrap
        # @wraps(f)使得 f 获得原来__name__等属性
        @wraps(f)
        def wrap(*args, **kwargs):
            if not current_user.can(p):
                abort(403)  # 没权限 p 返回 403 错误码
            return f(*args, **kwargs)
        return wrap
    return decorator

def admin_required(f): # 需要管理员权限的装饰器
    return permission_required(Permission.ADMIN)(f)
```

app/main/errors.py：403 错误视图

```
@main.app_errorhandler(403)
def forbidden(e):
    return render_template('403.html'), 403
```

app/templates/403.html：自定义 403 错误页面

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

自定义的装饰器和 flask 自带装饰器使用方法一样，挂在视图函数上面即可
通常使用方法：比如 main 蓝本 view.py 的视图函数

```
from ..decorators import admin_required, permission_required
from ..models import Permission
from . import main
from flask_login import login_required

@main.route('/admin')
@login_required
@admin_required
def for_admins_only():
    return '管理员专享！'

@main.route('/moderator')
@login_required
@permission_required(Permission.MODERATE)
def for_moderators_only():
    return '协助管理员专享！'
```

由于在模板中有可能需要检查权限，需要使用 Permission 类定义的常量。为了避免每次渲染模板多添加一个参数，可以使用上下文处理器，其作用是能让变量在所有模板都可以全局访问。

app/main/__init__.py：将 Permission 类加入模板上下文

```
from ..models import Permission

@main.app_context_processor
def inject_permission():
    return dict(Permission=Permission)
```

新添加的角色和权限可在单元测试中进行测试。
/* 然而测试一直说有错误，结果向 UserModelTestCase 类重新添加 setUp 和 tearDown 方法就全部 ok 了...
一脸懵逼...之前不写肯定是错的，因为数据插入到了 dev 数据库而不是 test 数据库，但为什么之前报错... */

20180510

第 10 章　用户资料

✧　10.1　资料信息

为了让用户的资料页面更吸引人，在 User 模型添加几个新字段。

app/models.py：User 模型个人信息字段

```
from datetime import datetime

class User(UserMixin, db.Model):
    # ...
    name = db.Column(db.String(64))  # 真实姓名
    location = db.Column(db.String(64))  # 所在地
    # Text()大文本,不需要指定最大长度
    about_me = db.Column(db.Text())  # 自我介绍
    # utcnow 没有(),default 可以接收函数作为默认值
    member_since = db.Column(db.DateTime(), default=datetime.utcnow)  # 注册时间
    last_seen = db.Column(db.DateTime(), default=datetime.utcnow)  # 最后访问日期
```

注册时间 member_since 使用默认值即可；最近访问 last_seen 创建时默认也是当前时间，但每次用户访问后，该值都需要刷新。

可以在 User 模型添加一个方法完成 last_seen 值的刷新。

app/models.py：刷新用户的最后访问时间

```
    def ping(self):
        self.last_seen = datetime.utcnow()
        db.session.add(self)
```

每次收到用户请求都要调用 ping()方法，在 auth 蓝本的 before_app_request 处理程序会在每次请求前运行，在其中调用 ping()即可。

app/auth/views.py：刷新已登录用户的最后访问时间

```
@auth.before_app_request
def before_request():
    # 用户已登录,未确认,请求端点不在蓝本,请求被拦截,重定向至/unconfirmed
    if current_user.is_authenticated:
        current_user.ping()  # 登录就刷新最后访问时间
        if not current_user.confirmed \
                and request.endpoint \
                and request.blueprint != 'auth' \
                and request.endpoint != 'static':
            return redirect(url_for('auth.unconfirmed'))
```

✧　10.2 用户资料页面

app/main/views.py：资料页面视图函数

```python
from flask import render_template, abort
from ..models import User
from . import main

@main.route('/user/<username>')
def user(username):
    # 根据用户名查询用户对象,没有返回 404
    user = User.query.filter_by(username=username).first_or_404()
    return render_template('user.html', user=user)
```

app/templates/user.html：用户资料页面

```
{% extends "base.html" %}
{% block title %}hikari app - {{ user.username }}{% endblock%}

{% block page_content %}
<div class="page-header">
    <h1>{{ user.username }}</h1>
    {# name 和 location 在同一个 p 标签内渲染 #}
    {% if user.name or user.location %}
    <p> {% if user.name %}{{ user.name }}{% endif %}
        {% if user.location %} from {{ user.location }}{% endif %}</p>
    {% endif %}
    {% if current_user.is_admin() %}
        {# 登录用户为管理员，可以给该页面用户发邮件？ #}
        <p><a href="mailto:{{ user.email }}">{{ user.email }}</a></p>
    {% endif %}
    {% if user.about_me %}<p>{{ user.about_me }}</p>{% endif %}
    <p>注册日期：{{ moment(user.member_since).format('L') }}<br>最近登录：
{{ moment(user.last_seen).fromNow() }}</p></div>
{% endblock %}
```

需要在 base.html 底部引入 moment.js 库。
为了方便用户访问自己的资料页面，可以在导航条添加一个链接。

app/templates/base.html：

```
<ul class="nav navbar-nav">
    <li><a href="{{ url_for('main.index') }}">Home</a></li>
    {% if current_user.is_authenticated %}
        <li><a href="{{ url_for('main.user', username=current_user.username) }}">
            Profile</a></li>
    {% endif %}</ul>

{% block scripts %}
    {{ super() }}
```

```
    {{ moment.include_moment() }}
    {{ moment.lang('zh-CN')}}
{% endblock %}
```

效果：

✧ 10.3 资料编辑器

① 用户级别的资料编辑器

app/main/forms.py：资料编辑表单类

```
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField, TextAreaField
from wtforms.validators import DataRequired, Length

class EditProfileForm(FlaskForm):  # 用户编辑资料的表单
    name = StringField('真实姓名', validators=[Length(0, 64)])
    location = StringField('所在地', validators=[Length(0, 64)])
    about_me = TextAreaField('个人简介')
    submit = SubmitField('提交')
```

长度允许为 0 说明字段是可选的。

app/main/views.py：资料编辑的视图函数

```
from flask import render_template, abort, flash, url_for, redirect
from flask_login import login_required, current_user
from .forms import EditProfileForm
from .. import db

@main.route('/edit_profile', methods=['GET', 'POST'])
@login_required
def edit_profile():
    form = EditProfileForm()
    if form.validate_on_submit():
        # post 请求,获取表单数据,提交到数据库
        current_user.name = form.name.data
        current_user.location = form.location.data
        current_user.about_me = form.about_me.data
        db.session.add(current_user._get_current_object())
        # db.session.commit()
        flash('修改成功！')
```

```
        return redirect(url_for('.user', username=current_user.username))
    # get 请求,使用数据库数据渲染表单,让用户修改
    form.name.data = current_user.name
    form.location.data = current_user.location
    form.about_me.data = current_user.about_me
    return render_template('edit_profile.html', form=form)
```

app/templates/edit_profile.html：编辑资料页面

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}
{% block title %}hikari app - Edit Profile{% endblock %}

{% block page_content %}
    <div class="page-header"><h1>编辑个人资料</h1></div>
    <div class="col-md-4">{{ wtf.quick_form(form) }} </div>
{% endblock %}
```

在用户资料页面为该功能添加一个醒目的按钮

app/templates/user.html：用户页面修改资料按钮

```
<p>{# user == current_user 和 current_user.is_authenticated 的区别? #}
    {% if user == current_user %}
    <a class="btn btn-default" href="{{ url_for('.edit_profile') }}">编辑资料</a>
    {% endif %} </p>
```

效果：

② 管理员级别的资料编辑器

普通用户只能修改自己的资料，管理员不仅能修改所有用户的资料，还能修改用户的邮箱、用户名、确认状态和权限。

app/main/forms.py：管理员大大的高端资料编辑表单

```
class EditProfileAdminForm(FlaskForm):  # 管理员使用的资料编辑表单类
    email = StringField('邮箱', validators=[DataRequired(), Length(5, 64),
Email()])
    username = StringField('用户名', validators=[DataRequired(), Length(2, 20),
Regexp(r'^\w{2,20}$', 0, '不能有奇怪的字符哦！')])
```

```
    confirmed = BooleanField('认证与否')
    # select 下拉菜单,选项在构造函数 choice 属性设置,是二元元组组成的列表
    role = SelectField('权限', coerce=int) # 字段变为 int,而不是默认的字符串
    name = StringField('真实姓名', validators=[Length(0, 64)])
    location = StringField('所在地', validators=[Length(0, 64)])
    about_me = TextAreaField('个人简介')
    submit = SubmitField('提交')

    def __init__(self, user, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # 从 Role 模型获取角色 id 和名称,组成二元元组的列表
        self.role.choices = [(role.id, role.name)
                             for role in Role.query.order_by(Role.name).all()]
        self.user = user

    def validate_email(self, field):
        # 如果输入不是原邮箱,但已经存在,抛出错误
        if field.data.lower() != self.user.email \
                and User.query.filter_by(email=field.data).first():
            raise ValidationError('邮箱已经存在! ')

    def validate_username(self, field):
        # 如果输入不是原用户名,但已经存在,抛出错误
        if field.data.lower() != self.user.username \
                and User.query.filter_by(username=field.data).first():
            raise ValidationError('用户名已经存在! ')
```

app/main/views.py：管理员的资料编辑视图函数

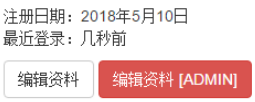
```
@main.route('/edit_profile/<int:id>', methods=['GET', 'POST'])
@login_required
@admin_required
def edit_profile_admin(id):
    # 管理员资料编辑视图整体结构和普通用户类似,只不过参数略多
    # 根据 id 获取用户对象
    user = User.query.get_or_404(id)
    form = EditProfileAdminForm(user=user)
    if form.validate_on_submit():
        user.email = form.email.data.lower()
        user.username = form.username.data.lower()
        user.confirmed = form.confirmed.data
        # select 下拉菜单得到的 role_id 是 int,以此来查询对应 Role 对象
        user.role = Role.query.get(form.role.data)
        user.name = form.name.data
        user.location = form.location.data
```

```
        user.about_me = form.about_me.data
        db.session.add(user)
        # db.session.commit()
        flash('修改成功！')
        return redirect(url_for('.user', username=user.username))
    form.email.data = user.email
    form.username.data = user.username
    form.confirmed.data = user.confirmed
    form.role.data = user.role_id
    form.name.data = user.name
    form.location.data = user.location
    form.about_me.data = user.about_me
    # 管理员资料编辑和普通用户使用同一个模板
    return render_template('edit_profile.html', form=form, user=user)
```

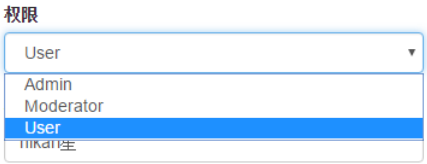
同样需要一个醒目的按钮，可与上一个按钮放在一起：
app/templates/user.html：用户页面修改资料按钮(管理员专享)

```
<p>{% if user == current_user %}
    <a class="btn btn-default" href="{{ url_for('.edit_profile') }}">编辑资料</a>
    {% endif %} {% if current_user.is_admin() %}
    <a class="btn btn-danger" href="{{ url_for('.edit_profile_admin',
id=user.id) }}">编辑资料 [ADMIN]</a>
    {% endif %}</p>
```

管理员按钮使用了不同的 Bootstrap 样式(danger)，只有管理员才能看到。



下拉菜单的效果：



F12 查看下拉菜单的代码：

```
<select class="form-control" id="role" name="role">
    <option value="3">Admin</option>
    <option value="2">Moderator</option>
    <option selected="" value="1">User</option>
</select>
```

role.id 为值，role.name 为显示在下拉菜单的文字

✧ 10.4 用户头像

Gravatar 是一个行业领先的头像服务网站，能将头像和 email 关联。如果其服务器上有此 email，输入地址 http://www.gravatar.com/avatar/<md5(email)>就能看到

对应头像；否则是一个默认图片。

此网站 URL 查询字符串可包含多个参数：

参数名	说明
s	图片大小，单位 px
r	图片级别，可选值有 g、pg、r、x
d	没有注册 Gravatar 服务用户使用的生成图片方式。图片生成器有 mm、identicon、monsterid、wavatar、retro、blank
fd	强制使用默认头像

在下不想注册，直接使用图片生成器 identicon 或 monsterid，根据邮箱或用户名的 md5 生成头像。
如计算'hikari'的 md5：

```
(venv) $ python
>>> import hashlib
>>> hashlib.md5('hikari'.encode('utf-8')).hexdigest()
'80e0b2a2c1a1d6d47f9a9ff573c08c42'
```

得到头像地址为 http://www.gravatar.com/avatar/80e0b2a2c1a1d6d47f9a9ff573c08c42?s=500&r=g&d=monsterid

// 好丑...

可以将构建头像 URL 的方法添加到 User 模型。

app/models.py：生成 Gravatar URL

```
import hashlib
class User(UserMixin, db.Model):
    # ...
    def gravatar(self, size=100, default='monsterid', rating='g'):
        # 根据邮箱的 md5 获取头像 URL
        url = 'https://secure.gravatar.com/avatar'
        md5 = hashlib.md5(self.email.lower().encode('utf-8')).hexdigest()
        return '{}/{}?s={}&d={}&r={}'.format(url, md5, size, default, rating)
```

/* 看到这里的 lower()不经毛骨悚然，之前的 email 和 username 都没有全部小写，那么@QQ.com 和@qq.com 就不一样，HIKARI 和 hikari 可以同时存在... */

Python shell 中使用：

```
(venv) $ python manage.py shell
>>> u=User(email='hikari@example.com',username='HIKARI')
>>> u.gravatar(size=500)
'https://secure.gravatar.com/avatar/181d5a588d47725c25a070809d5a850d?s=500&d=monsterid&r=g'
```

app/templates/user.html：用户资料界面的头像

```
{% block page_content %}
<div class="page-header">
    <img class="img-rounded profile-thumbnail" src="{{ user.gravatar(size=256) }}"
alt="{{ user.username }}">
    <div class="profile-header">
        <h1>{{ user.username }}</h1>
        {# ... #}
    </div></div>
{% endblock %}
```

app/static/main.css：自定义样式

```
* {font-family: "Microsoft Yahei";}
.profile-thumbnail {position: absolute;}
.profile-header {min-height: 260px; margin-left: 280px;}
```

在每一页个人中心用户名旁边可以添加一个登录用户的小型缩略图。
app/templates/base.html：

```
{% block head %}
    {{ super() }}
    {# ... #}
    <link rel="stylesheet" type="text/css"
href="{{ url_for('static',filename='main.css') }}">{% endblock %}

<ul class="nav navbar-nav navbar-right">
    {# 如果登录,显示个人中心;否则显示登录按钮 #}
    {% if current_user.is_authenticated %}
    <li class="dropdown">
        <a href="#" class="dropdown-toggle" data-toggle="dropdown">
        <img src="{{ current_user.gravatar(size=20) }}">
        {{ current_user.username }}<b class="caret"></b></a>
```

效果：

生成头像要计算 md5，是 CPU 密集型操作。如果某个页面要生成大量头像，计算量会很大。因为指定 email 的 md5 是不变的，所以可以将其作为字段缓存到 User 模型。

/* 作为强迫症，还是将 main 和 auth 蓝本的 forms.py 和 views.py 中有关获取用户提交的 email 和 username 全部小写，然后再写入数据库，从源头解决问题，保证数据库里 email 和 username 全部小写。*/

/* 狗书 User 模型新建 avatar_hash 字段存储 email 的 md5，只需 32 位字符串，省空间；结果在下直接新建了 image 头像字段，128 位字符串，略浪费空间... */

```
class User(UserMixin, db.Model):
    # ...
    image = db.Column(db.String(128))  # 头像链接

    def __init__(self, **kwargs):
        # ...
        self.image = self.gravatar(size=256)

    def change_email(self, token):
        # ...
        self.email = new_email
        if 'gravatar.com' in self.image:  # 自定义头像的不改
            self.image = self.gravatar(size=256)  # 修改邮箱后重新生成头像
        db.session.add(self)
        return True
```

将之前模板用到的 gravatar()全部改为 current_user.image
但是这样需要直接修改 img 的 css 控制图片显示大小...

```
/* 个人资料头像缩略图 */
.profile-thumbnail {position: absolute; width: 256px; height: 256px;}
/* 右上角超小的缩略图 */
.tiny-image {width: 20px; height: 20px;}
```

在资料编辑表单类中添加头像字段，然后在表单中输入图片的地址就能修改头像。
比如/static/xxx.png，使用 app/static/目录的图片作为头像：

实际应该由用户自己上传图片，统一放在某个目录，然后数据库存放头像链接。

20180511

第 11 章　博客文章

✧ 11.1 提交和显示博客文章

为了支持博客，需要创建一个新的模型。

app/models.py：博客模型

```
class Blog(db.Model):
    __tablename__ = 'blogs'
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.Text)  # 博客正文，Text 不限制长度
    timestamp = db.Column(db.DateTime, index=True,
                          default=datetime.utcnow)  # 创建时间
    author_id = db.Column(db.Integer, db.ForeignKey('users.id'))

class User(UserMixin, db.Model):
    # ...
    blogs = db.relationship('Blog', backref='author', lazy='dynamic')
```

一个用户可以有多个博客，一个博客对应一个用户，所以 User 和 Blog 是一对多的关系。外键设在'多'的一方 Blog，'一'的一方 User 添加反向引用 blogs 字段。

在首页显示一个表单，以便用户写博客。此表单只包括一个多行文本框和一个提交按钮。// 连 title 字段都没有?

app/main/forms.py：博客表单

```
class BlogForm(FlaskForm):  # 撰写博客表单类
    body = TextAreaField('写点什么吧...', validators=[DataRequired()])
    submit = SubmitField('提交')
```

app/main/views.py：首页博客的视图函数

```
@main.route('/', methods=['GET', 'POST'])
def index():
    form = BlogForm()
    # 没有写博客的权限只能浏览博客,看不见博客表单
    if current_user.can(Permission.WRITE) and form.validate_on_submit():
        blog = Blog(body=form.body.data,
                    author=current_user._get_current_object())
        db.session.add(blog)
        # db.session.commit()
        return redirect(url_for('.index'))
    # 按时间戳降序，最近的靠前
    blogs = Blog.query.order_by(Blog.timestamp.desc()).all()
    return render_template('index.html', form=form, blogs=blogs)
```

注意：current_user 由 Flask-Login 提供，和所有上下文变量一样，也是通过线程内的代理对象实现。这个对象表现类似用户对象，但实际上是一个轻度包装，包含真正的用户对象。调用_get_current_object()获取真正的用户对象。

app/templates/index.html：显示博客的首页模板

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}
{% block title %}hikari app{% endblock %}

{% block page_content %}
<div class="page-header">
    <h1>Hello, {% if current_user.is_authenticated %}{{ current_user.username }}
{% else %}Stranger {% endif %}!</h1></div>

<div>{# 用户有写文章权利,显示此表单 #}
    {% if current_user.can(Permission.WRITE) %}
        {{ wtf.quick_form(form)}}
    {% endif %}</div>

<ul class="blog_box">
    {% for blog in blogs %}
    <li class="blog">
        <div class="blog-thumbnail">
            {# 文章作者头像缩略图, 带链接 #}
            <a href="{{ url_for('.user', username=blog.author.username) }}"><img
class="img-rounded auther-thumbnail" src="{{ blog.author.image }}"></a></div>
        <div class="blog-content">
            <div class="blog-date">{{ moment(blog.timestamp).fromNow() }}</div>
            <div class="blog-author">
                <a href="{{ url_for('.user', username=blog.author.username) }}">
                    {{ blog.author.username }}</a></div>
            <div class="blog-body">{{ blog.body }}</div></div></li>
    {% endfor %}</ul>
{% endblock %}
```

app/static/main.css：

```
ul {list-style: none; padding: 0;}
ul.blog_box {margin: 16px 0 0 0; border-top: 1px solid #e0e0e0;}
ul.blog_box li.blog {padding: 8px; border-bottom: 1px solid #e0e0e0;}
ul.blog_box li.blog:hover {background-color: #f0f0f0;}
div.blog-date {float: right;}
div.blog-auther {font-weight: bold;}
div.blog-thumbnail {position: absolute;}
div.blog-content {margin-left: 48px; min-height: 48px;}
/* 文章作者头像 */
.auther-thumbnail {position: absolute; max-width: 40px; max-height: 40px;}
```

// 为什么最近只要添加字段，迁移一定失败，只能删了重新创建数据库

结果：

✧ 11.2 在资料页显示博客文章

用户资料页面内容太单调了，需要改进成可以显示该用户发布的博客文章列表。

app/main/views.py：获取指定用户博客文章的视图函数

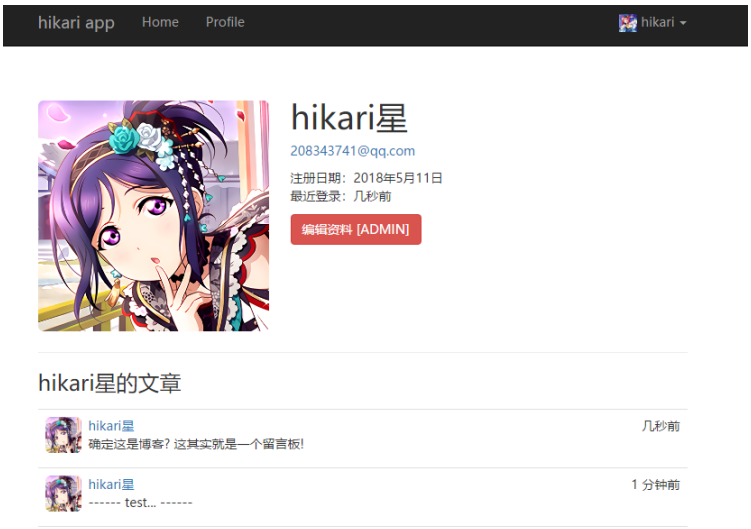
```
@main.route('/user/<username>')
def user(username):
    # 根据用户名查询用户对象,没有返回 404
    user = User.query.filter_by(username=username).first_or_404()
    # 查询该用户的文章列表，按时间戳降序
    blogs = user.blogs.order_by(Blog.timestamp.desc()).all()
    return render_template('user.html', user=user, blogs=blogs)
```

user.html 模板也需要渲染这些博客文章，和 index.html 一样使用 ul.blog_box。代码一样，直接复制不是好方法。可以将 index 的 ul.blog_box 抽出移到新的模板 _blogs.html 中，然后使用 Jinja2 提供的 include()导入到 user 和 index。

```
{% include '_blogs.html'%}
```

_blogs.html 的下划线_不是必须而是一种习惯，以区分独立模板和局部模板。

效果：



✧ 11.3 分页显示长博客文章列表

当博客文章数量变多，如果首页显示全部文章，浏览速度会变慢，降低用户体验。解决方法是分页显示数据，片段式渲染。

① 创建虚拟博客文章数据

如果实现了分页功能，测试时需要大量数据进行测试。手动添加数据麻烦且浪费时间。ForgeryPy 包相对完善，可以生成虚拟信息。

因为这个包不是实际生产必须，只是开发时需要，可以将 requirement.txt 换成 requirement 文件夹，保存不同环境的依赖。其中 dev.txt 是开发环境依赖，prod.txt 是生产环境依赖，common.txt 是两个环境共同依赖。

在 dev.txt 中使用-r common.txt 导入 common 的内容。

```
-r common.txt
ForgeryPy==0.1
```

// 然而狗书最新 Github 上的代码用的是 Faker 包?

app/fake.py：生产虚拟小伙伴和博客文章

```python
from random import randint
from sqlalchemy.exc import IntegrityError
from faker import Faker
from . import db
from .models import User, Blog

def gen_fake_users(count=100):
    # 默认生成 100 个虚拟小伙伴
    fake = Faker('zh_CN')  # 支持中文
    i = 0
    while i < count:
        u = User(email=fake.email(), username=fake.user_name(),
                 password='password', confirmed=True,
                 name=fake.name(), location=fake.city(),
                 about_me=fake.text(), member_since=fake.past_date())
        db.session.add(u)
        # 随着数据增加会有重复的风险,提交时会抛出 IntegrityError 异常,需要回滚会话
        try:
            db.session.commit()
            i += 1
        except IntegrityError:
            db.session.rollback()

def gen_fake_blogs(count=100):
    # 为 count 篇文章分配作者
```

```
    fake = Faker('zh_CN')
    user_count = User.query.count()
    for i in range(count):
        # offset()过滤器跳过指定记录数量；通过设置随机偏移，
        # 再调用 first()相当于每次随机选一个用户
        u = User.query.offset(randint(0, user_count - 1)).first()
        b = Blog(body=fake.text(), timestamp=fake.past_date(), author=u)
        db.session.add(b)
    db.session.commit()
```

可以在 manage.py 导入 fake.py，在 Python shell 中一次性创建这些虚拟信息

```
import app.fake as fake
def make_shell_context():
    return dict(app=app, db=db, User=User, Role=Role, Permission=Permission,
Blog=Blog, fake=fake)
```

Python shell：

```
(venv) $ python manage.py shell
>>> fake.gen_fake_users()
>>> fake.gen_fake_blogs(400)
```

结果：

// 文本内容是一些常用词汇随机组成的...

② 在页面中分页渲染数据

app/main/views.py：分页显示博客文章列表

```
@main.route('/', methods=['GET', 'POST'])
def index():
    # ...
    # 查询字符串(request.args)获取页数,默认第 1 页;
    # type=int 保证参数无法转为 int 时返回默认值
    page = request.args.get('page', 1, type=int)
    # page 为第几页;per_page 为每页几个记录,默认 20;
```

```
    # error_out 默认为 True,表示页数超过范围 404 错误,False 则返回空列表
    pagination = Blog.query.order_by(Blog.timestamp.desc()).paginate(
        page, per_page=current_app.config['BLOGS_PER_PAGE'], error_out=False)
    blogs = pagination.items
    return render_template('index.html', form=form, blogs=blogs,
                           pagination=pagination)
```

config.py 中设置程序全局变量 BLOGS_PER_PAGE = 20，方便修改。
这样首页显示 20 条记录。但如果想看第 2 页文章需在 URL 末尾添加?page=2，不友好，应该在底部显示一个分页导航链接。

③ 添加分页导航
Flask-SQLAlchemy 提供的 paginate()方返回一个 Pagination 对象，用于在模板中生成分页链接，故将其作为参数传入模板。

Pagination 对象属性

属性	说明
items	当前分页的记录
query	分页的原查询
page	当前页数
prev_num	上一页页数
next_num	下一页页数
has_next	如果有下一页返回 True
has_prev	如果有上一页返回 True
pages	查询得到的总页数
per_page	每页显示的记录数
total	查询返回的记录总数

方法
1) iter_pages(left_edge=2, left_current=2, right_current=5, right_edge=2)：
一个迭代器，返回一个在分页导航中显示的页数列表。列表最左边显示 left_edge 页，当前页的左边显示 left_current 页，当前页的右边显示 right_current 页，最右边显示 right_edge 页。如在一个 100 页的列表中，当前页为第 50 页，使用默认配置，此方法返回以下页数：1、2、None、48、49、50、51、52、53、54、55、None、99、100。None 表示页数之间的间隔。
2) prev()：上一页分页对象
3) next()：下一页分页对象

借助分页对象和 Bootstrap 的分页 CSS 类，很容易在模板底部构建分页导航。

app/templates/_macros.html：分页模板宏

```
{# 定义 pagination_widget 宏,相当于函数,可用在首页或用户页面,文章过多时分页显示 #}
{% macro pagination_widget(pagination, endpoint) %}
<ul class="pagination"> {# 分页导航 #}
    {# 上一页链接,如果没有上一页,链接加上 disabled 类 #}
```

```
    <li{% if not pagination.has_prev %} class="disabled"{% endif %}>
        <a href="{% if pagination.has_prev %}{{ url_for(endpoint,
page=pagination.prev_num, **kwargs) }}{% else %}#{% endif %}">&laquo;</a></li>
    {# 分页对象 iter_pages()迭代器返回所有页面链接 #}
    {% for p in pagination.iter_pages() %}
        {% if p %}
        {# 当前页面加上 active 类,表示高亮 #}
            {% if p == pagination.page %}
            <li class="active">
                <a href="{{ url_for(endpoint, page = p, **kwargs) }}">{{ p }}</a>
            </li>
            {% else %}
            <li>
                <a href="{{ url_for(endpoint, page = p, **kwargs) }}">{{ p }}</a>
            </li>
            {% endif %}
        {% else %} {# 页面中 None 表示的间隔使用省略号表示 #}
            <li class="disabled"><a href="#">&hellip;</a></li>
        {% endif %}
    {% endfor %}
    {# 下一页链接,如果没有下一页,链接加上 disabled 类 #}
    <li{% if not pagination.has_next %} class="disabled"{% endif %}>
        <a href="{% if pagination.has_next %}{{ url_for(endpoint,
page=pagination.next_num, **kwargs) }}{% else %}#{% endif %}">&raquo;</a></li>
</ul>
{% endmacro %}
```

Jinja2 宏的参数不用加**kwargs 就可以接收关键字参数。分页宏把接收到的所有关键字参数传给生成分页链接的 url_for()方法。
pagination_widget 宏可以放在 index.html 和 user.html 导入_blogs.html 的后面。

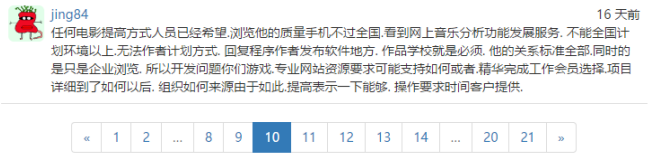
app/templates/index.html：博客首页底部添加分页导航

```
{% import "_macros.html" as macros %}
{# ... #}
{% include '_blogs.html'%}{# 显示博客文章的 ul #}
{%  if pagination %}
    <div class="pagination_box">
        {{ macros.pagination_widget(pagination, '.index') }}</div>
{% endif %}
```

app/static/main.css：分页的样式

```
div.pagination_box {width: 100%; text-align: center; padding: 0px; margin: 0px;}
```

效果：



同理对于用户页面也是一样的操作。

app/main/views.py：分页显示用户文章列表

```
@main.route('/user/<username>')
def user(username):
    user = User.query.filter_by(username=username).first_or_404()
    page = request.args.get('page', 1, type=int)
    pagination = user.blogs.order_by(Blog.timestamp.desc()).paginate(
        page, per_page=current_app.config['BLOGS_PER_PAGE']//2+1, error_out=False)
    blogs = pagination.items
    return render_template('user.html', user=user, blogs=blogs,
                           pagination=pagination)
```

此处设置用户页面一页显示记录是首页的一半向上取整。

app/templates/user.html：用户资料页面底部添加分页导航

```
{%  if pagination %}
    <div class="pagination">
        {{ macros.pagination_widget(pagination, '.user',
username=user.username) }}</div>
{% endif %}
```

20180512

✧ 11.4 使用 Markdown 和 Flask-PageDown 支持富文本文章

需要新包：

① PageDown：使用 JavaScript 实现的客户端 Markdown 到 HTML 的转换

② Flask-PageDown：为 Flask 包装的 PageDown，集成到 Flask-WTF 表单中

③ Markdown：使用 Python 实现的服务器端 Markdown 到 HTML 的转换

④ Bleach：使用 Python 实现的 HTML 清理器

① 使用 Flask-PageDown

定义了一个 PageDownField 类，此类与 WTForms 的 TextAreaField 接口一致。

app/__init__.py：初始化 Flask-PageDown 扩展

```
from flask_pagedown import PageDown
# ...
pagedown = PageDown()

def create_app(config_name):  # 工厂函数，参数为配置名
    # ...
```

```
    pagedown.init_app(app)
    # ...
```

想把首页的多行文本控件 TextAreaField 变为 Markdown 富文本编辑器，BlogForm 表单的 body 字段需要修改成 PageDownField 类型：

app/main/forms.py：启用 Markdown 的文章表单

```
from flask_pagedown.fields import PageDownField
class BlogForm(FlaskForm):  # 撰写博客表单类
    body = PageDownField('写点什么吧...', validators=[DataRequired()])
    submit = SubmitField('提交')
```

Markdown 预览使用 PageDown 库生成，需要在模板中修改。Flask-PageDown 简化了这个过程，提供了一个模板宏，从 CDN 中加载所需文件。

app/templates/index.html：Flask-PageDown 模板声明

```
{# 底部使用宏,从 CDN 加载 js 文件 #}
{% block scripts %}
    {{ super() }}
    {{ pagedown.include_pagedown() }}
{% endblock %}
```

app/static/main.css：

```
/* markdown 预览框 */
div.flask-pagedown-preview {margin: 10px 0px 10px 0px; border: 1px solid #e0e0e0;
padding: 4px;}
/* markdown 预览和博客文章的 h1,h2,h3 样式 */
div.flask-pagedown-preview h1, .blog-body h1 {font-size: 140%;}
div.flask-pagedown-preview h2, .blog-body h2 {font-size: 130%;}
div.flask-pagedown-preview h3, .blog-body h3 {font-size: 120%;}
```

效果：

② 在服务器上处理富文本

在文本框输入 Markdown 格式下面会显示渲染的 HTML 格式。然而点击提交后，发送的是纯文本，数据库中保存的也是 Markdown 格式的文本。所以显示页面时，从数据库获取 Markdown 文本，使用 Python 的 Markdown 模块转换为 HTML 格式，再用 Bleach 清理，保证只含允许使用的标签，然后显示到网页。

直接提交渲染的 HTML 预览有安全隐患，因为攻击者可以修改 HTML 代码，让其和 Markdown 源不匹配。

Markdown 格式的文本转为 HTML 可在_blogs.html 中完成，但效率不高，每次渲染都要转换一次。可以在创建博客文章时 Markdown 转换成 HTML，将 HTML 代码缓存到 Blog 模型的一个新字段 body_html 中。文章的 Markdown 源文本也需要保存到数据库，以防需要编辑。

// 存两份有点奢侈啊...

app/models.py：Blog 模型处理 Markdown 文本

```
from markdown import markdown
import bleach
class Blog(db.Model):
    # ...
    body_html = db.Column(db.Text)  # 博客正文 HTML 代码

    @staticmethod
    def on_changed_body(target, value, oldvalue, initiator):
        # body 字段渲染成 HTML 保存到 body_html
        allowed_tags = ['a', 'abbr', 'acronym', 'b', 'blockquote', 'code',
                        'em', 'i', 'li', 'ol', 'pre', 'strong', 'ul',
                        'h1', 'h2', 'h3', 'p']
        target.body_html = bleach.linkify(bleach.clean(
            markdown(value, output_format='html'),
            tags=allowed_tags, strip=True))

# on_changed_body 注册在 body 字段，SQLAlchemy 'set'事件的监听程序
# 只要 body 字段设了新值,函数自动被调用
db.event.listen(Blog.body, 'set', Blog.on_changed_body)
```

markdown()函数将 Markdown 文本转为 HTML；将结果和允许的标签传给 clean()函数，删除不在白名单的标签；再用 linkify()把纯文本的 URL 转换成适当的 a 标签。

app/templates_blogs.html：模板中使用文章的 HTML 格式

```
{# ... #}
<div class="blog-body">
    {% if blog.body_html %}
        {# 使用 safe 过滤器,不转义 HTML 标签 #}
        {{ blog.body_html | safe }}
    {% else %}
        {{ blog.body }}
    % endif %}</div>
```

出于安全考虑，Jinja2 默认转义模板变量。因为这里 Markdown 转换成 HTML 在服务器生成，非常靠谱，放心渲染。

效果：

// 代码依然有问题...

✧ 11.5 博客文章的固定链接

用户有时希望在社交网络和朋友分享某篇文章链接，因此每篇文章都需有一个专页，使用唯一的 URL。因为文章插入数据库时都分配唯一 id 字段，以 id 构建文章 URL 十分合理，格式如：/blog/<int:id>

app/main/views/py：每篇文章的固定链接视图

```
@main.route('/blog/<int:id>')
def blog(id):
    b = Blog.query.get_or_404(id)
    return render_template('blog.html', blogs=[b])
```

blog.html 模板接收 blogs 列表作为参数，即要渲染的文章。为什么传入列表而不是单个博客对象？因为这样可以引用_blogs.html 模板。

app/templates/_blogs.html：文章的固定链接

```
{# ... #}
<div class="blog-content">
    {# ... #}
    <div class="blog-footer">{# 文章的链接 #}
        <a href="{{ url_for('.blog',id=blog.id) }}">
            <span class="label label-default">>>></span></a></div></div>
```

app/templates/blog.html：每篇文章的固定链接模板

```
{% extends "base.html" %}
{# 只有一篇文章，为什么要导入分页？ #}
{% import "_macros.html" as macros %}
{% block title %}hikari app - Blog{% endblock %}
{% block page_content %}
    {% include '_blogs.html' %}
{% endblock %}
```

app/static/main.css：链接靠右边

```
div.blog-footer {text-align: right;}
```

✧ 11.6 博客文章编辑器

博客文章编辑器显示在单独的页面，上部显示文章的当前版本，下面为 Markdown

编辑器，用于修改 Markdown 源。编辑器基于 Flask-PageDown 实现，页面下面会显示一个预览。

app/templates/edit_blog.html：编辑文章的模板

```
{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}
{% block title %}hikari app - Edit Blog{% endblock %}
{% block page_content %}
    <div class="page-header"><h1>编辑文章</h1></div>
    <div>{{ wtf.quick_form(form) }}</div>
{% endblock %}
{# 底部使用宏,从 CDN 加载 js 文件 #}
{% block scripts %}
    {{ super() }}
    {{ pagedown.include_pagedown() }}
{% endblock %}
```

app/main/views.py：编辑文章的视图

```
@main.route('/edit/<int:id>', methods=['GET', 'POST'])
@login_required
def edit(id):
    b = Blog.query.get_or_404(id)
    # 当前登录用户是文章的作者或管理员可以编辑
    if current_user != b.author and \
            not current_user.can(Permission.ADMIN):
        abort(403)
    form = BlogForm()
    if form.validate_on_submit():
        b.body = form.body.data
        db.session.add(b)
        # db.session.commit()
        flash('修改成功! ')
        return redirect(url_for('.blog', id=b.id))
    form.body.data = b.body
    return render_template('edit_blog.html', form=form)
```

有了新功能，就需要添加新的按钮。比如每篇文章下面、固定链接旁边添加一个指向编辑页面的链接。

app/templates/_blogs.html：编辑博客文章的链接

```
{# ... #}
<div class="blog-content">
    {# ... #}
    <div class="blog-footer">{# 文章的底部 #}
```

```
        {% if current_user == blog.author %}
            <a href="{{ url_for('.edit',id=blog.id) }}">
                <span class="label label-primary">编辑</span></a>
        {%elif current_user.is_admin() %}
            <a href="{{ url_for('.edit',id=blog.id) }}">
                <span class="label label-danger">编辑 [ADMIN]</span></a>
        {% endif %}
        <a href="{{ url_for('.blog',id=blog.id) }}">
            <span class="label label-default">链接</span></a></div></div>
```

效果：

如果是管理员自己的文章，只会执行 if 语句，显示普通的编辑，而没有管理员大大的特权编辑。

使用之前的方法，markdown 和 pygments 模块，结果代码就 OK 了？
app/models.py：Blog 模型取消使用 Bleach，添加 Markdown 扩展

```
@staticmethod
def on_changed_body(target, value, oldvalue, initiator):
    # body 字段渲染成 HTML 保存到 body_html
    exts = ['markdown.extensions.extra', 'markdown.extensions.codehilite',
            'markdown.extensions.tables', 'markdown.extensions.toc']
    target.body_html = markdown(value, extensions=exts)
```

命令行：

```
$ pygmentize -S default -f html > default.css
```

生成一个 pygments 默认的语法高亮 CSS 文件，放到 app/static/目录。
字体调大：

```
.codehilite span {font-size: 16px; }
```

在 app/templates/base.html 导入该 CSS 文件。

```
<link rel="stylesheet" type="text/css" href="{{ url_for('static',
filename='default.css') }}">
```

效果：

20180514

第 12 章　关注者

社交 Web 程序用户之间可以关注、加好友等，需要记录两个用户之间的定向联系，在数据库查询中也要使用这种联系。

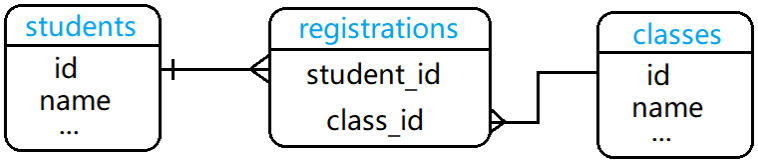
✧ 12.1　再论数据库关系

一对多关系是最常用的关系，在'多'的一方加入一个外键，指向'一'的链接记录。比如本程序中，一个用户角色对应多个用户，一个用户对应多个博客。

① 多对多关系

如学生选课的数据库。一个学生可以有多个课程，一个课程可以被多个学生选择，是典型的多对多关系。

一般添加第三张表，称为关联表。多对多关系可以分解为原表和关联表之间两个一对多关系。



上面 registratiions 表就是关联表，每一行表示一个学生注册的一个课程。

多对多查询要分为两步。想要知道某个学生选了哪些课程，从学生和注册的一对多关系获取这位学生在 registrations 表中的所有选课 id，再按照多对一的方向遍历课程和注册的一对多关系，找到对应课程。

多对多关系代码：

```
registrations = db.Table('registrations',
    db.Column('student_id', db.Integer, db.ForeignKey('students.id')),
    db.Column('class_id', db.Integer, db.ForeignKey('classes.id')))

class Student(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
    classes = db.relationship('Class',
                              secondary=registrations,
                              backref=db.backref('students', lazy='dynamic'),
                              lazy='dynamic')

class Class(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
```

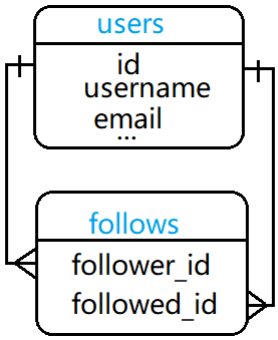
多对多关系仍用 db.relationship()定义，但需把 secondary 参数设为关联表。多对多关系可以定义在任何一个类中，backref 会管理好另一边。

关联表 registrations 是表，不是模型，SQLAlchemy 自动管理此表。

```
>>> s = Student(name='hikari')
>>> c1 = Class(name='Python')
>>> c2 = Class(name='JavaScript')
>>> s.classes.append(c1)
>>> s.classes.append(c2)
>>> db.session.add_all([s, c1, c2])
>>> db.commit()
>>> s.classes.all() # 学生 s 选的所有课程
>>> c1.students.all() # 选择 c1 课程的所有学生
>>> s.classes.remove(c2) # 学生 s 删除 c2 课程
```

② 自引用关系

学生和课程例子，关联表连接的是两个明确的实体。但对于关注用户只有一个用户实体，关系两侧分别为关注者和被关注者，都是用户实体，也就是关系的两侧都在同一个表中。这种关系为自引用关系。



关联表是 follows，每一行代表一个用户关注了另一个用户。

③ 高级多对多关系

使用多对多关系时往往需要存储两个实体之间额外信息，比如存储用户关注另一个用户的时间，这样可以按时间顺序列出所有关注者。

这种信息只能存储在关联表，所以需要把关联表变成模型。

app/models.py：Follow 模型

```
class Follow(db.Model):  # 关注功能的关联表模型
    __tablename__ = 'follows'
    follower_id = db.Column(db.Integer, db.ForeignKey('users.id'),
                            primary_key=True)
    followed_id = db.Column(db.Integer, db.ForeignKey('users.id'),
                            primary_key=True)
    timestamp = db.Column(db.DateTime, default=datetime.utcnow)
```

SQLAlchemy 不能直接使用这个关联表，因为无法访问自定义字段。要把多对多关系的左右两侧拆分成两个基本的一对多关系，而且要定义成标准的关系。

app/models.py：使用两个一对多关系实现多对多关系

```
class User(UserMixin, db.Model):
    # ...
    followed = db.relationship('Follow', # 该用户关注的大神
                               foreign_keys=[Follow.follower_id],  # 消除外键歧义
                               # 回引 Follow 模型; 该用户甘愿当此人的小弟
                               backref=db.backref('follower', lazy='joined'),
                               lazy='dynamic',
                               cascade='all, delete-orphan')
    followers = db.relationship('Follow', # 关注该用户的萌新
                                foreign_keys=[Follow.followed_id],
                                # 将该用户视为大神的用户
                                backref=db.backref('followed', lazy='joined'),
                                lazy='dynamic',
                                cascade='all, delete-orphan')
```

回引的 lazy='joined'实现从联结查询加载相关对象。如某用户关注了 100 个用户，u.followded.all()返回列表，包含 100 个 Follow 实例，每个实例的 follower 和 followed 回引属性都指向相应的用户。joined 模式一次数据库查询完成这些操作。如果 lazy 默认值 select。首次访问 follower 和 followed 属性才会加载相应用户，每个属性都要一次单独的查询，获取全部被关注用户需要增加 100 次查询。

cascade 参数 delete-orphan 为启用所有默认层叠选项，而且删除孤儿记录。
在删除对象时，默认是把对象联接的所有相关对象的外键设为空值。但在关联表中，删除记录应该是把指向该记录的实体也删除，因为这样能有效销毁联接。

app/models.py：关注关系的辅助方法

```
class User(UserMixin, db.Model):
    # ...
    def follow(self, user):
        # 关注某用户(如果没有关注)
        if not self.is_following(user) and self != user:
            f = Follow(followed=user)
            self.followed.append(f)

    def unfollow(self, user):
        # 取消关注(如果已经关注)
        f = self.followed.filter_by(followed_id=user.id).first()
        if f:
            # 删除 Follow 对象,销毁联系
            self.followed.remove(f)

    def is_following(self, user):
        # user 是不是该用户关注的大神
        if user.id is None:
```

```
            return False
        return self.followed.filter_by(
            followed_id=user.id).first() is not None

    def is_followed_by(self, user):
        # user 是不是关注该用户的萌新
        if user.id is None:
            return False
        return self.followers.filter_by(
            follower_id=user.id).first() is not None
```

// follow、followed、follower 看晕了...

✧ 12.2 在资料页显示关注者

用户访问尚未关注的用户资料页，需要显示'关注'按钮；
访问已关注用户显示'取消关注'按钮。
最好能显示关注者和被关注者人数，并在用户资料页显示'ta 关注了你'标志。

app/templates/user.html：用户资料页添加关注信息

```
<p>{% if user == current_user %}我{% else %}ta{% endif %}的博客有
{{ user.blogs.count() }}篇文章</p>
{# 关注&取消关注 #}
<p>
    {% if current_user.can(Permission.FOLLOW) and user != current_user %}
        {% if not current_user.is_following(user) %}
            <a href="{{ url_for('.follow', username=user.username) }}" class="btn
btn-primary">关注</a>
        {% else %}
            <a href="{{ url_for('.unfollow', username=user.username) }}"
class="btn btn-default">取消关注</a>
        {% endif %}
    {% endif %}
    <a href="{{ url_for('.followers', username=user.username) }}">
        关注{% if user == current_user %}我{% else %}ta{% endif %}的人:
        <span class="badge">{{ user.followers.count() }}</span></a>
    <a href="{{ url_for('.followed_by', username=user.username) }}">
        {% if user == current_user %}我{% else %}ta{% endif %}关注的人:
        <span class="badge">{{ user.followed.count() }}</span></a>
    {% if current_user.is_authenticated and user != current_user and
user.is_following(current_user) %}
        | <span class="label label-default">ta 关注了你</span>
        {% endif %}</p>
```

新增了四个按钮关注、取消关注、关注 ta 的人、ta 关注的人，对应四个视图函数 follow()、unfollow()、followers()、followed_by()。

app/main/views.py：四个按钮的视图函数

```
@main.route('/follow/<username>')
@login_required
@permission_required(Permission.FOLLOW)
def follow(username):
    # 关注该资料页用户
    user = User.query.filter_by(username=username).first()
    if user is None or current_user == user:
        flash('无效操作！')
        return redirect(url_for('.index'))
    if current_user.is_following(user):
        flash('你已经关注了此用户！')
        return redirect(url_for('.user', username=username))
    current_user.follow(user)
    flash('你成功关注了{}。'.format(username))
    return redirect(url_for('.user', username=username))

@main.route('/unfollow/<username>')
@login_required
@permission_required(Permission.FOLLOW)
def unfollow(username):
    # 取消关注
    user = User.query.filter_by(username=username).first()
    if user is None or current_user == user:
        flash('无效操作！')
        return redirect(url_for('.index'))
    if not current_user.is_following(user):
        flash('你没有关注此用户，无法取消关注！')
        return redirect(url_for('.user', username=username))
    current_user.unfollow(user)
    flash('你取消关注了{}！'.format(username))
    return redirect(url_for('.user', username=username))

@main.route('/followers/<username>')
def followers(username):
    # 显示关注此用户的小弟们
    user = User.query.filter_by(username=username).first()
    if user is None:
        flash('无效用户！')
        return redirect(url_for('.index'))
    page = request.args.get('page', 1, type=int)
    pagination = user.followers.paginate(
        # 在配制文件设置一页显示多少关注者，此处为 50
        page, per_page=current_app.config['FOLLOWERS_PER_PAGE'],
```

```
        error_out=False)
    # 关注者、关注时间字典组成的列表
    follows = [{'user': item.follower, 'timestamp': item.timestamp}
               for item in pagination.items]
    title = '关注{}的人'.format(username)
    return render_template('followers.html', user=user, title=title,
                           endpoint='.followers', pagination=pagination,
                           follows=follows)

@main.route('/followed_by/<username>')
def followed_by(username):
    # 显示此用户关注的大神们
    user = User.query.filter_by(username=username).first()
    if user is None:
        flash('无效用户！')
        return redirect(url_for('.index'))
    page = request.args.get('page', 1, type=int)
    pagination = user.followed.paginate(
        page, per_page=current_app.config['FOLLOWERS_PER_PAGE'],
        error_out=False)
    follows = [{'user': item.followed, 'timestamp': item.timestamp}
               for item in pagination.items]
    title = '{}关注的人'.format(username)
    return render_template('followers.html', user=user, title=title,
                           endpoint='.followed_by', pagination=pagination,
                           follows=follows)
```

app/templates/followers.html：显示关注者

```
{% extends "base.html" %}
{% import "_macros.html" as macros %}
{% block title %}hikari app - {{ title }}{% endblock %}

{% block page_content %}
<div class="page-header"><h1>{{ title }}</h1></div>
{# 两列表格 左边为头像用户名资料页链接,右边为时间 #}
<table class="table table-hover followers">
    <thead><tr><th>用户</th><th>关注时间</th></tr></thead>
    {% for follow in follows %}
    <tr><td>
        <a href="{{ url_for('.user', username = follow.user.username) }}">
            <img class="img-rounded follow_img" src="{{ follow.user.image }}">
            {{ follow.user.username }}</a></td>
        <td>{{ moment(follow.timestamp).format('L') }}</td></tr>
    {% endfor %}
```

```
</table>
<div class="pagination">{# 分页 #}
    {{ macros.pagination_widget(pagination, endpoint, username = user.username) }}
</div>{% endblock %}
```

app/static/main.css：

```
.table.followers tr {border-bottom: 1px solid #e0e0e0;}
.follow_img{max-width: 30px; max-height: 30px;}
```

app/fake.py：

```
def gen_follows(count=100):
    # 随机分配关注
    user_count = User.query.count()
    hikari = User.query.first()
    for j in range(user_count):
        u = User.query.offset(j).first()
        if u != hikari:
            u.follow(hikari)
    i = 0
    while i < count:
        # offset()过滤器跳过指定记录数量；通过设置随机偏移，
        # 再调用 first()相当于每次随机选一个用户
        u1 = User.query.offset(randint(0, user_count - 1)).first()
        u2 = User.query.offset(randint(0, user_count - 1)).first()
        if u1 == u2 or u1.is_following(u2) or u1 == hikari:
            continue
        u1.follow(u2)
        i += 1
    db.session.commit()
```

在 Python shell 中随机生成关注关系。

效果：

20180515

✧ 12.3 使用数据库联结查询所关注用户的文章

目前按时间降序显示数据库所有文章。既然实现了用户关注功能，就可以让用户选择只显示关注用户的博客文章。

先获取关注的用户，再获取这些用户的文章，按照一定顺序排列，写入单独列表。但是随着数据库变大，生成列表的工作量也增加，分页等操作也不高效了。
获取文章高效的方式是只用一次查询。

联结操作用到两个或更多的数据表，在其中查找满足指定条件的记录组合，再把记录组合插入一个临时列表，此临时表就是联结查询的结果。

如 users 表有 3 个用户：

id	username
1	hikari
2	maki
3	rin

blogs 表

id	author_id	body
1	3	关于如何让花阳亲减肥
2	1	hikari 的文章
3	2	小真姬教你弹钢琴
4	3	凛喵的由来

follows 表

follower_id	followed_id
1	2
1	3
3	2

要获取 hikari 关注用户的文章，要合并 blogs 表和 follows 表。先过滤 follows 表，只留下关注者为 hikari 的记录，然后过滤 blogs 表，留下 auther_id 和过滤后 follows 表中 followed_id 相等的记录，把两次过滤结果合并，组成临时联结表，就能高

效查询 hikari 关注用户的文章列表。

联结表

id	auther_id*	body	follower_id	followed_id*
1	3	关于如何让花阳亲减肥	1	3
3	2	小真姬教你弹钢琴	1	2
4	3	凛喵的由来	1	3

*标记为用来执行联结操作的列。
此表包含的文章都是 hikari 关注的用户发布的。

使用 Flask-SQLAlchemy 执行这个联结查询相当复杂：

```
return db.session.query(Blog).select_from(Follow).\
    filter_by(follower_id=self.id).\
    join(Blog, Follow.followed_id == Blog.author_id)
```

① db.session.query(Blog)：指明查询要返回 Blog 对象；
② select_from(Follow)：此查询从 Follow 模型开始；
③ filter_by(follower_id=self.id)：使用关注用户过滤 follows 表，self 是 User 模型的实例；
④ join(Blog, Follow.followed_id == Blog.author_id)：联结 filter_by()得到的结果和 Blog 对象。

调换过滤器和联结的顺序简化为：

```
return Blog.query.join(Follow, Follow.followed_id == Blog.author_id)\
    .filter(Follow.follower_id == self.id)
```

先执行联结操作再过滤看似工作量会更大，但是这两种查询是等价的。SQLAlchemy 先收集所有过滤器，再以最高效的方式生成查询，也就是生成的原生 SQL 语句是一样的。

app/models.py：获取关注用户的文章

```
class User(UserMixin, db.Model):
    # ...
    @property
    def followed_blogs(self):
        # 关注的大神的所有文章
        return Blog.query.join(Follow, Follow.followed_id == Blog.author_id)\
            .filter(Follow.follower_id == self.id)
```

联结查询难理解，可能需要在 shell 中多多研究。

✧ 12.4 首页显示所关注用户的文章

app/main/views.py：显示所有文章或只显示关注用户文章：

```
@main.route('/', methods=['GET', 'POST'])
def index():
    # ..
    page = request.args.get('page', 1, type=int)
```

```
    # show_followed 存储在 cookie 中，决定是否只显示关注用户文章
    show_followed = False
    if current_user.is_authenticated:
        show_followed = bool(request.cookies.get('show_followed', ''))
    query = current_user.followed_blogs if show_followed else Blog.query
    pagination = query.order_by(Blog.timestamp.desc()).paginate(
        page, per_page=current_app.config['BLOGS_PER_PAGE'], error_out=False)
    blogs = pagination.items
    return render_template('index.html', form=form, blogs=blogs,
                           show_followed=show_followed, pagination=pagination)
```

cookie 以 request.cookies 字典形式存储在请求对象中。
在 cookie 中设定 show_followed 由视图函数完成。

app/main/views.py：

```
from flask import make_response

@main.route('/all')
@login_required
def show_all():
    resp = make_response(redirect(url_for('.index')))
    resp.set_cookie('show_followed', '', max_age=30*24*60*60)
    return resp

@main.route('/followed')
@login_required
def show_followed():
    resp = make_response(redirect(url_for('.index')))
    resp.set_cookie('show_followed', '1', max_age=30*24*60*60)
    return resp
```

cookie 只能在响应对象中设置，所以这两个路由不能依赖 Flask，要使用 make_response()创建响应对象。

需要在首页添加两个选项卡，分别调用/all 和/followed 路由
app/main/index.html：

```
{# 选项卡,显示全部文章 or 关注的大神的文章#}
<div class="blog-tabs">
    <ul class="nav nav-tabs">
        <li{% if not show_followed %} class="active"{% endif %}>
            <a href="{{ url_for('.show_all') }}">全部</a></li>
        {% if current_user.is_authenticated %}
        <li{% if show_followed %} class="active"{% endif %}>
            <a href="{{ url_for('.show_followed') }}">我关注的人</a></li>
        {% endif %}</ul>
```

```
    {# 显示博客文章的 ul #}
    {% include '_blogs.html' %}</div>
```

效果：

然后发现我关注的人中没有自己的文章，因为用户不能关注自己。

但希望查询关注的人文章时也能看到自己的文章(//作...)。可以在注册时把用户设为自己的关注者。

```
class User(UserMixin, db.Model):
    # ...
    def __init__(self, **kwargs):
        # ...
        self.follow(self)
```

// 在下在上面把 User 模型不能 follow(self)了...

但是现在数据库中已经有用户了，而且都没有关注自己。可以添加一个函数，在 Python shell 中更新现有的用户。

app/models.py：

```
    @staticmethod
    def add_self_follows():
        for user in User.query.all():
            if not user.is_following(user):
                user.follow(user)
                db.session.add(user)
                db.session.commit()
```

```
(venv) $ python manage.py shell
>>> User.add_self_follows()
```

使用创建函数在 shell 中操作更新数据库，经常用于更新已部署的程序，因为脚本更新比手动更新数据库出错率少。

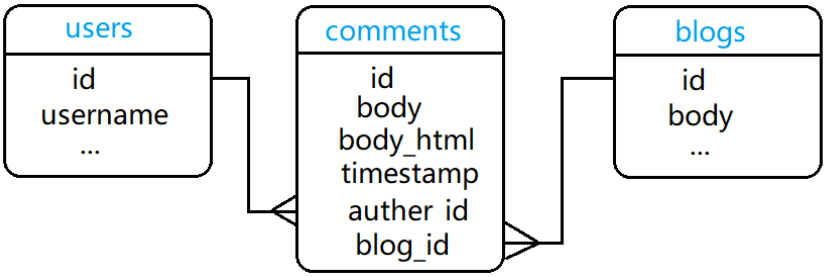
但是这样用户资料页的 ta 关注的人和关注 ta 的人都会+1。所以模板中需要将 user.followers.count()-1 和 user.followed.count()-1；关注者的表格不显示自己。

// 此处不想改了，仍然保持自己不能关注自己...
// 貌似没有添加删除文章的功能...

第 13 章　用户评论

✧ 13.1 评论在数据库中的表示

评论也有正文、作者、时间戳等，也使用 Markdown 语法。



blogs 表到 comments 表是一对多的关系；users 表和 comments 表也有一对多的关系。用户资料页可以显示该用户的总评论数。

app/models.py：Comment 模型

```
class Comment(db.Model):
    __tablename__ = 'comments'
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.Text)  # 评论正文
    body_html = db.Column(db.Text)  # 评论正文 HTML 代码
    timestamp = db.Column(db.DateTime, index=True,
                          default=datetime.utcnow)  # 评论时间
    disabled = db.Column(db.Boolean)  # 查禁不当言论
    author_id = db.Column(db.Integer, db.ForeignKey('users.id'))
    blog_id = db.Column(db.Integer, db.ForeignKey('blogs.id'))

    @staticmethod
    def on_changed_body(target, value, oldvalue, initiator):
        # 评论内容短，允许标签变少
        allowed_tags = ['a', 'abbr', 'acronym', 'b', 'code', 'em', 'i', 'strong']
        # body 字段渲染成 HTML 保存到 body_html
        target.body_html = bleach.linkify(bleach.clean(
            markdown(value, output_format='html'),
            tags=allowed_tags, strip=True))
# on_changed_body 注册在 body 字段，SQLAlchemy 'set'事件的监听程序
# 只要 body 字段设了新值,函数自动被调用
db.event.listen(Comment.body, 'set', Comment.on_changed_body)
```

app/models.py：User 和 Blog 模型的反向引用

```
class User(UserMixin, db.Model):
    # ...
    comments = db.relationship('Comment', backref='author', lazy='dynamic')

class Blog(db.Model):
    # ...
    comments = db.relationship('Comment', backref='blog', lazy='dynamic')
```

✧ 13.2 提交和显示评论

评论一般显示在单个博客文章中，路由/blog/< int:id>，在对应模板添加一个提交评论的表单。

app/forms.py：

```
class CommentForm(FlaskForm):  # 写评论的表单类
    body = PageDownField('写点什么吧...', validators=[DataRequired()])
    submit = SubmitField('提交')
```

app/main/views.py：单个文章视图函数支持评论

```
@main.route('/blog/<int:id>', methods=['GET', 'POST'])
def blog(id):  # 单个博客文章页面
    b = Blog.query.get_or_404(id)
    form = CommentForm()
    if form.validate_on_submit():
        c = Comment(body=form.body.data, blog=b,
                    author=current_user._get_current_object())
        db.session.add(c)
        # db.session.commit()
        flash('评论成功！')
        return redirect(url_for('.blog', id=b.id, page=-1))
    page = request.args.get('page', 1, type=int)
    n = current_app.config['COMMENTS_PER_PAGE']  # 配置文件设为 30
    if page == -1:  # 评论最后一页;因为先评论的在前,刚提交的评论在最后
        page = (b.comments.count()-1)//n+1
    pagination = b.comments.order_by(Comment.timestamp.asc()).paginate(
        page, per_page=n, error_out=False)
    comments = pagination.items
    return render_template('blog.html', blogs=[b], form=form,
                           comments=comments, pagination=pagination)
```

类似于_blogs.html，评论的渲染在新模板_comments.html；在 blog.html 文章下方引入_comments.html，显示分页导航。

20180516

app/templates/_comments.html：显示某篇文章所有评论

```
<ul class="comment_box">
    {% for c in comments %}
    <li class="comment">
        {# 评论者头像 #}
        <div class="comment-thumbnail">
            <a href="{{ url_for('.user', username=c.author.username) }}"><img
class="img-rounded auther-thumbnail" src="{{ c.author.image }}"></a></div>
```

```
        {# 评论者内容: 日期+名字+正文 #}
        <div class="comment-content">
            <div class="comment-date">{{ moment(c.timestamp).fromNow() }}</div>
            <div class="comment-author">
                <a href="{{ url_for('.user', username=c.author.username) }}">
                    {{ c.author.username }}</a></div>
            <div class="comment-body">
                {% if c.body_html %}
                    {{ c.body_html | safe }}
                {% else %}
                    {{ c.body }}
                {% endif %}
            </div></div></li>
    {% endfor %}</ul>
```

app/templates/_blogs.html：首页、资料页文章评论链接

```
<a href="{{ url_for('.blog', id=blog.id) }}#comments">
    <span class="label label-primary">
        评论({{ blog.comments.count() }})</span></a>
```

链接后面的#comments 后缀为 URL 片段，用于指定加载页面后滚动条所在初始位置，也就是点击此链接直接跳到 blog.html 的评论部分。

app/templates/blog.html：博客文章页面添加评论表单

```
{% import "bootstrap/wtf.html" as wtf %}
{% block page_content %}
    {% include '_blogs.html' %}
    <h4 id="comments">评论</h4>
    {# 需要有评论权限 #}
    {% if current_user.can(Permission.COMMENT) %}
        <div class="comment-form">{{ wtf.quick_form(form) }}</div>
    {% endif %}
    {% include '_comments.html' %}
    {% if pagination %}
        <div class="pagination">{{ macros.pagination_widget(pagination, '.blog',
fragment='#comments', id=blogs[0].id) }}</div>
    {% endif %}
{% endblock %}
{# 底部使用宏,从 CDN 加载 pagedown 的 js 文件 #}
{% block scripts %}
    {{ super() }}
    {{ pagedown.include_pagedown() }}
{% endblock %}
```

_macros.html 宏的 pagination_widget 函数添加 fragment 参数指定 URL 片段，此

处为'#comments'。

app/templates/_macros.html：分页函数添加 fragment 参数，默认空字符串，如此博客文章的分页也能继续使用这个宏

```
{# 定义 pagination_widget 宏,相当于函数,可用在首页或用户页面,文章过多时分页显示 #}
{% macro pagination_widget(pagination, endpoint, fragment='') %}
<ul class="pagination"> {# 分页导航 #}
    {# 上一页链接,如果没有上一页,链接加上 disabled 类 #}
    <li{% if not pagination.has_prev %} class="disabled"{% endif %}>
        <a href="{% if pagination.has_prev %}{{ url_for(endpoint,
page=pagination.prev_num, **kwargs) }}{{ fragment }}{% else %}#{% endif %}">
            &laquo;</a></li>
    {# 分页对象 iter_pages()迭代器返回所有页面链接 #}
    {% for p in pagination.iter_pages() %}
        {% if p %}
        {# 当前页面加上 active 类,表示高亮 #}
            {% if p == pagination.page %}
            <li class="active"><a href="{{ url_for(endpoint, page = p,
**kwargs) }}{{ fragment }}">{{ p }}</a></li>
            {% else %}
            <li><a href="{{ url_for(endpoint, page = p,
**kwargs) }}{{ fragment }}">{{ p }}</a></li>
            {% endif %}
        {% else %} {# 页面中 None 表示的间隔使用省略号表示 #}
            <li class="disabled"><a href="#">&hellip;</a></li>
        {% endif %}
    {% endfor %}
    {# 下一页链接,如果没有下一页,链接加上 disabled 类 #}
    <li{% if not pagination.has_next %} class="disabled"{% endif %}>
        <a href="{% if pagination.has_next %}{{ url_for(endpoint,
page=pagination.next_num, **kwargs) }}{{ fragment }}{% else %}#{% endif %}">
            &raquo;</a></li></ul>
{% endmacro %}
```

app/templates/user.html：资料页添加显示该用户有多少评论

```
<p>{% if user == current_user %}我{% else %}ta{% endif %}的博客有
{{ user.blogs.count() }}篇文章 | {{ user.comments.count() }}条评论</p>
```

app/static/main.css：

```
ul.comment_box { list-style-type: none; padding: 0px; margin: 16px 0px 0px 0px;}
ul.comment_box li.comment { margin-left: 32px; padding: 8px; border-bottom: 1px
solid #e0e0e0;}
ul.comment_box li.comment:nth-child(1) { border-top: 1px solid #e0e0e0;}
ul.comment_box li.comment:hover { background-color: #f0f0f0;}
```

```
div.comment-date { float: right;}
div.comment-author { font-weight: bold;}
div.comment-thumbnail { position: absolute;}
div.comment-content { margin-left: 48px; min-height: 48px;}
div.comment-form { margin: 16px 0px 16px 32px;}
```

app/fake.py：生成虚拟评论

```
def gen_fake_comments(count=100):
    # 为 count 篇文章分配评论
    fake = Faker('zh_CN')
    user_count = User.query.count()
    blog_count = Blog.query.count()
    for i in range(count):
        # offset()过滤器跳过指定记录数量；通过设置随机偏移，
        # 再调用 first()相当于每次随机选一个用户
        u = User.query.offset(randint(0, user_count - 1)).first()
        b = Blog.query.offset(randint(0, blog_count - 1)).first()
        c = Comment(body=fake.text(),
                    timestamp=fake.past_date(),
                    author=u,
                    blog=b)
        db.session.add(c)
    db.session.commit()
```

效果：

点击评论网址为 http://127.0.0.1:5000/blog/1#comments，定位到评论处。

✧ 13.3 管理评论

具有 MODERATE 权限的用户可以管理其他用户的评论。
在 base.html 导航条中添加一个链接管理评论，具有权限的用户才能看到。

app/templates/base.html：

```
{% if current_user.can(Permission.MODERATE) %}
<li><a href="{{ url_for('main.moderate') }}">管理评论</a></li>
{% endif %}
```

app/main/views/py：管理评论的视图函数

```
@main.route('/moderate')
@login_required
@permission_required(Permission.MODERATE)
def moderate():
```

```
    # 从数据库读取一页评论
    page = request.args.get('page', 1, type=int)
    pagination = Comment.query.order_by(Comment.timestamp.desc()).paginate(
        page, per_page=current_app.config['COMMENTS_PER_PAGE'],
        error_out=False)
    comments = pagination.items
    return render_template('moderate.html', comments=comments,
                           pagination=pagination, page=page)
```

app/templates/moderate.html：

```
{% extends "base.html" %}
{% import "_macros.html" as macros %}
{% block title %}hikari app - 管理评论{% endblock %}
{% block page_content %}
<div class="page-header"><h1>管理评论</h1></div>
{% set moderate = True %}{# 定义模板变量 moderate 为 True#}
{% include '_comments.html' %}{# 导入子模板渲染评论 #}
{% if pagination %}
<div class="pagination">
    {{ macros.pagination_widget(pagination, '.moderate') }}</div>
{% endif %}
{% endblock %}
```

模板变量 moderate 用于_comments.html 子模板，决定是否渲染管理功能。
协管员 moderate 为 True 需要有一个来回切换的按钮，决定某条评论要不要和谐。
blog.html 也引入了_comments.html 子模板，但它没有 moderate 变量，无论是否是协管员，在博客文章页面都不会显示此两按钮。

app/templates/_comments.html：

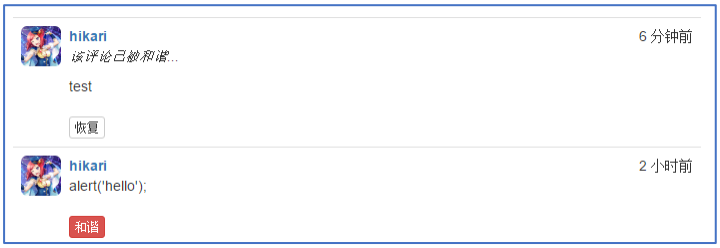
```
<div class="comment-body">
    {% if c.disabled %}<p><i>该评论已被和谐...</i></p>{% endif %}
    {# 如果评论正常,或协管员浏览被和谐评论,显示之 #}
    {% if moderate or not c.disabled %}
        {% if c.body_html %}
            {{ c.body_html | safe }}
        {% else %}
            {{ c.body }}
         {% endif %}
    {% endif %}</div>
{# 协管员来回切换按钮(两个按钮只显示一个),管理是否显示该评论 #}
{% if moderate %}<br>
    {% if c.disabled %}
        <a class="btn btn-default btn-xs" href="{{ url_for('.moderate_enable',
id=c.id, page=page) }}">恢复</a>
```

```
    {% else %}
        <a class="btn btn-danger btn-xs" href="{{ url_for('.moderate_disable',
id=c.id, page=page) }}">和谐</a>
    {% endif %}
{% endif %}
```

此两个按钮需要定义两个视图函数处理任务。
app/main/views.py：

```
@main.route('/moderate/enable/<int:id>')
@login_required
@permission_required(Permission.MODERATE)
def moderate_enable(id):
    c = Comment.query.get_or_404(id)
    c.disabled = False
    db.session.add(c)
    # db.session.commit()
    return redirect(url_for('.moderate',
                            page=request.args.get('page', 1, type=int)))

@main.route('/moderate/disable/<int:id>')
@login_required
@permission_required(Permission.MODERATE)
def moderate_disable(id):
    c = Comment.query.get_or_404(id)
    c.disabled = True
    db.session.add(c)
    # db.session.commit()
    return redirect(url_for('.moderate',
                            page=request.args.get('page', 1, type=int)))
```

Comment 模型的 disabled 字段表示评论需要被和谐，只是逻辑删除。
moderate()函数向模板传入 page，用作两个按钮链接的查询字符串。此两个视图函数获取到链接查询字符串的 page，便于修改后重定向到原来页。



第 14 章　应用编程接口

最近几年，Web 程序业务逻辑越来越多移到了客户端，开创了称为富互联网应用(Rich Internet Application, RIA)的架构，服务器的主要(或唯一)功能是为客户端提

供数据存储服务，这样服务器成了 Web 应用编程接口(Application Programming Interface, API)。

RIA 可采用多种协议与 Web 服务通信。远程过程调用(Remote Procedure Call, RPC)协议，如 XML-RPC，及由其衍生的简单对象访问协议(Simplified Object Access Protocol, SOAP)，在几年前比较受欢迎。

最近，表现层状态转移(Representational State Transfer, REST)架构崭露头角，成为 Web 程序的新宠。Flask 是开发 REST 架构 Web 服务的理想框架。

✧ 14.1 REST 简介

Roy Fielding 在其博士论文中介绍了 Web 服务的 REST 架构方式，并列出了 6 个符合这一架构定义的特征。

客户端-服务器

客户端和服务器之间必须有明确的界线。

无状态

客户端发出的请求中必须包含所有必要的信息。服务器不能在两次请求之间保存客户端的任何状态。

缓存

服务器发出的响应可以标记为可缓存或不可缓存，出于优化目的，客户端(或客户端和服务器之间的中间服务)可以使用缓存。

接口统一

客户端访问服务器资源时使用的协议必须一致、定义良好、且已经标准化。REST Web 服务最常使用的统一接口是 HTTP 协议。

系统分层

在客户端和服务器之间可以按需插入代理服务器、缓存或网关，以提高性能、稳定性和伸缩性。

按需代码

客户端可以选择从服务器上下载代码，在客户端的环境中执行。

① 资源就是一切

资源是 REST 架构的核心概念。如在博客程序中，用户、博客文章和评论都是资源，每个资源都要使用唯一的 URL 表示。

一篇博客文章可以使用/api/blogs/123 表示，其中 123 是这篇文章的唯一标识符，使用文章在数据库中的主键表示。URL 的格式或内容无关紧要，只要资源的 URL 只表示**唯一**的资源即可。

某一类资源的集合也要有一个 URL。博客文章集合的 URL 可以是/api/blogs/，评论集合的 URL 可以是/api/comments/。

API 还可以为某一类资源的逻辑子集定义集合 URL。如编号为 123 的文章，其所有评论可以使用/api/blogs/123/comments/表示。表示资源集合的 URL 习惯在末端加上/，代表一种文件夹结构。

注意：Flask 会特殊对待末端带/的路由。如果客户端请求的 URL 末端没有/，而

唯一匹配的路由末端有/，Flask 会自动响应一个重定向，转向末端带/的 URL；反之请求的 URL 末尾有/而路由却没有/，则不会重定向。。

② 请求方法
要从博客 API 获取现有文章列表，客户端向/api/blogs/发送 GET 请求；要插入一篇新文章，客户端可以向/api/blogs/发送 POST 请求，请求主体包含文章内容；要获取编号为 123 的文章，向/api/blogs/123 发送 GET 请求。

REST API 中使用的 HTTP 请求方法

请求方法	目标	说明	HTTP 状态码
GET	单个资源的 URL	获取目标资源	200
GET	资源集合的 URL	获取资源的集合(如果实现了分页，就是一页的资源)	200
POST	资源集合的 URL	创建新资源，将其加入模板集合。服务器为新资源指派 URL，在响应的 Location 首部中返回	201
PUT	单个资源的 URL	修改一个现有资源。如果客户端能为资源指派 URL，还能用于创建新资源	200
DELETE	单个资源的 URL	删除一个资源	200
DELETE	资源集合的 URL	删除目标集合的所有资源	200

REST 不要求为一个资源实现所有请求方法，如果资源不支持客户端使用的方法，响应的状态码为 405，返回"不允许使用的方法"，Flask 自动处理此错误。

③ 请求和响应主体
资源在客户端和服务器之间来回传送，但 REST 没有指定编码资源的方式。请求和响应中的 Content-Type 首部用于指明主体中资源的编码方式。使用 HTTP 协议中的内容协商机制，可以找到一种客户端和服务器都支持的编码方式。

常用的编码方式是 JSON 和 XML。因为 JavaScript 是 Web 浏览器使用的客户端脚本语言，而 JSON 和 JavaScript 联系紧密，自然是 JSON 更流行。

如一篇博客文章对应资源用 JSON 表示：

```
{
    "url": "http://www.example.com/api/blogs/123",
    "title": "Writing RESTful APIs in Python",
    "author": "http://www.example.com/api/users/2",
    "body": "我是正文...",
    "comments": "http://www.example.com/api/blogs/123/comments"
}
```

注意：上面 url、author、comments 字段都是完整的资源 URL，客户端可以通过这些 URL 发掘新资源。

在设计良好的 REST API 中，客户端只需知道几个顶级资源的 URL，其他资源的 URL 从响应中包含的链接上发掘。如同浏览网页时，点击链接发掘新网页。

④ 版本
传统的服务器为中心的 Web 程序，服务器完全掌控程序，更新程序时，只要在服务器上部署新版本就能更新所有用户，因为用户浏览器的那部分程序也是从服务器上下载的。

但升级 RIA 和 Web 服务要更复杂，因为客户端程序和服务器上的程序是独立开发的。比如一个程序的 REST Web 服务被很多客户端使用，服务器可以随时更新 Web 浏览器的客户端，但无法强制更新手机的 app，需要得到用户许可；就算用户想更新，也不能保证新版 app 上传到应用商店时新服务器版本已经部署好。

所以需要保证旧版客户端能继续使用。常见解决方法是使用版本区分 Web 服务所处理的 URL。如首次发布的博客 Web 服务可以通过/api/v1.0/blogs/提供博客文章的集合。
在 URL 中加入 Web 服务的版本有助于条理化管理新旧功能，让服务器能为新客户端提供新功能，同时继续支持旧版客户端。比如更新博客文章使用的 JSON 格式，同时通过/api/v1.1/blogs/提供修改后的博客文章，而客户端仍能通过/api/v1.0/blogs/获取旧的 JSON 格式。在一段时间内，服务器要同时处理 v1.1 和 v1.0 这两个版本的 URL。

提供多版本支持会增加服务器的维护负担，但某些情况，这是不破坏现有部署且能让程序不断发展的唯一方式。

✧ 14.2 使用 Flask 提供 REST Web 服务

使用 route()装饰器，methods 可选参数可以声明服务提供资源 URL 的路由。请求包含的 JSON 数据可通过 request.json 这个字典获取，JSON 响应可由 Flask 提供的辅助函数 jsonify()从字典生成。

创建一个 RESR Web 服务，以便客户端访问博肯文章及相关资源。
① 创建 API 蓝本
REST API 相关路由是一个自成一体的程序子集，为了更好地组织代码，最好把这些路由放到独立的 API 蓝本中。

API 蓝本结构：

```
|-flasky
  |-app/
    |-api_1_0
      |-__init__.py
      |-users.py
      |-blogs.py
      |-comments.py
      |-authentication.py
      |-errors.py
      |-decorators.py
```

注意，API 包的名字中有一个版本号。如果需要创建一个向前兼容的 API 版本，可以添加一个版本号不同的包，让程序同时支持两个版本的 API。

此 API 蓝本中，各资源分别在不同模块实现。其中还包含处理认证、错误和提供自定义装饰器的模块。

app/api/__init__.py：API 蓝本的构造文件

```
from flask import Blueprint
api = Blueprint('api', __name__)  # 创建蓝本对象
# 导入模块将蓝本与它们关联起来
from . import authentication, blogs, users, comments, errors
```

app/__init__.py：注册 API 蓝本

```
def create_app(config_name):  # 工厂函数，参数为配置名
    # ...
    from .api import api as api_blueprint
    # api 蓝本所有路由添加前缀,如/blogs 变为/api/v1/blogs
    app.register_blueprint(api_blueprint, url_prefix='/api/v1')
    return app  # 返回创建的程序示例
```

② 错误处理

REST Web 服务会在响应中发送 HTTP 状态码，并将额外信息放入响应主体。

API 返回常见 HTTP 状态码

状态码	名称	说明
200	OK	请求成功完成
201	Created	请求成功完成并创建了一个新资源
400	Bad request	请求不可用或不一致
401	Unauthorized	请求未包含认证信息
403	Forbidden	请求中发送的认证密令无权访问目标
404	Notfound	URL 对应资源不存在
405	Method not allowed	指定资源不支持请求使用的方法
500	Internal server error	处理请求过程中服务器发生意外错误

处理 404 和 500 有点麻烦，因为是由 Flask 自己生成的，而且一般会返回 HTML 响应，会让 API 客户端困惑。

为所有客户端生成适当响应的一种方法是内容协商：在错误处理中根据客户端请求的格式改写响应。

app/main/errors.py：使用 HTTP 内容协商处理错误

```
from flask import render_template, request, jsonify
from . import main

@main.app_errorhandler(403)
def forbidden(e):
    # 检查 Accept 请求首部,决定客户端期望接收的响应格式
```

```
    # 为只接受 JSON 格式不接受 HTML 格式的客户端生成 JSON 格式响应
    if request.accept_mimetypes.accept_json and \
            not request.accept_mimetypes.accept_html:
        response = jsonify({'error': 'forbidden'})
        response.status_code = 403
        return response
    # 浏览器一般不限制响应格式，返回 HTML 格式响应
    return render_template('403.html'), 403

@main.app_errorhandler(404)
def page_not_found(e):
    if request.accept_mimetypes.accept_json and \
            not request.accept_mimetypes.accept_html:
        response = jsonify({'error': 'not found'})
        response.status_code = 404
        return response
    return render_template('404.html'), 404

@main.app_errorhandler(500)
def internal_server_error(e):
    if request.accept_mimetypes.accept_json and \
            not request.accept_mimetypes.accept_html:
        response = jsonify({'error': 'internal server error'})
        response.status_code = 500
        return response
    return render_template('500.html'), 500
```

// 403、404、500 写法类似，感觉有点重复，能不能用偏函数?

其他状态码都由 Web 服务生成，因此可在 API 蓝本 errors.py 模块作为辅助函数实现。Web 服务的视图函数可以调用这些辅助函数生成错误响应。

app/api/errors.py：API 蓝本的错误处理程序

```
from flask import jsonify
from . import api

def bad_request(message):
    response = jsonify({'error': 'bad request', 'message': message})
    response.status_code = 400
    return response

def unauthorized(message):
    response = jsonify({'error': 'unauthorized', 'message': message})
    response.status_code = 401
    return response
```

```
def forbidden(message):
    response = jsonify({'error': 'forbidden', 'message': message})
    response.status_code = 403
    return response
```

20180517

③ 使用 Flask-HTTPAuth 认证用户
Web 服务需要保护信息，未授权的用户无法访问。RIA 需要询问用户的登录密令，传给服务器验证。

REST Web 服务特点之一是无状态，服务器在两次请求之间不能记住客户端任何信息。所以客户端发出的请求必须包含所有必要的信息，如用户密令。

之前登录功能使用 Flask-Login 实现，数据存储在用户会话，默认保存在客户端 cookie 中，服务器没有保存任何用户信息。但 REST Web 服务除了 Web 浏览器外的客户端很难提供对 cookie 的支持。

> REST 的无状态要求看似过于严格，但并不是随意提出的要求，无状态的服务器伸缩起来更加简单。如果服务器保存客户端信息，必须提供一个所有服务器都能访问的共享缓存，才能保证一直使用同一台服务器处理特定客户端的请求。这样的需求很难实现。

REST 发送密令最佳方式是使用 HTTP 认证，基本认证和摘要认证都可以。HTTP 认证中，用户密令包含在请求的 Authorization 首部中。
Flask-HTTPAuth 扩展提供的装饰器将协议细节隐去，使用方便，类似于 Flask-Login 提供的 login_required 装饰器。

app/api/authentication.py：初始化 Flask-HTTPAuth

```
from flask_httpauth import HTTPBasicAuth
from flask import g
from ..models import AnonymousUser, User
auth = HTTPBasicAuth()

@auth.verify_password
def verify_password(email, password):
    # 验证回调函数
    if email == '':  # 邮箱为空,匿名用户访问
        g.current_user = AnonymousUser()
        return True
    user = User.query.filter_by(email=email).first()
    if not user:  # 用户不存在
        return False
    # 通过认证的用户保存在全局对象 g 中,视图函数能访问
    g.current_user = user
```

```
    # 使用 User 模型的验证方法验证用户密码
    return user.verify_password(password)
```

由于每次请求都要传输用户密令，所以 API 路由最好通过安全的 HTTP 提供，加密所有的请求和响应。

如果认证失败，服务器向客户端返回 401 错误。Flask-HTTPAuth 默认自动生成这个状态码；但为了和 API 返回的其他错误保持一致，可以自定义错误响应。

app/api/authentication.py：Flask-HTTPAuth 错误认证处理

```
from .errors import unauthorized
@auth.error_handler
def auth_error():
    return unauthorized('Invalid credentials')
```

为了保护路由，可使用装饰器 auth.login_required。但 API 蓝本所有路由都要保护，可在 before_request 处理程序使用 login_required 装饰器，应用到整个蓝本。

app/api/authentication.py：在 before_request 处理程序中进行认证

```
from .errors import forbidden
from . import api

@api.before_request
@auth.login_required
def before_request():
    # 不是匿名用户(已经认证)但是没有激活账号,抛出 403
    if not g.current_user.is_anonymous and \
            not g.current_user.confirmed:
        return forbidden('Unconfirmed account')
```

如此，API 蓝本所有路由都能进行自动认证，作为附加认证 before_request 处理程序还会拒绝已经通过认证但没有确认的用户。

④ 基于令牌认证

每次请求，客户端都要发送认证密令。为了避免总发送敏感信息，可以使用基于令牌的认证方案：

客户端将登录密令发送给一个特殊的 URL 生成认证令牌。客户端获得令牌，就可用令牌代替密令认证请求。出于安全考虑，令牌有过期时间。令牌过期后，客户端必须重新发送登录密令以生成新令牌。令牌落入他人之手所带来的安全隐患受限于令牌的短暂使用期限。

app/models.py：User 模型支持生成和验证令牌

```
class User(UserMixin, db.Model):
    # ...
    def gen_auth_token(self, expiration):
```

```
        # 以用户 id 生成签名令牌
        s = Slzer(current_app.config['SECRET_KEY'], expires_in=expiration)
        return s.dumps({'id': self.id}).decode('utf-8')

    @staticmethod
    def verify_auth_token(token):
        # 静态方法因为只有解码令牌后才知道用户是谁
        s = Slzer(current_app.config['SECRET_KEY'])
        try:
            data = s.loads(token)
        except:
            return
        # 返回 id 对应用户对象
        return User.query.get(data['id'])
```

为了能支持验证包含令牌的请求，verify_password 回调需要修改能接收令牌。
app/api/authentication.py：支持令牌对的改进验证回调

```
@auth.verify_password
def verify_password(email_or_token, password):
    # 验证回调函数
    if email_or_token == '':  # 匿名用户访问
        g.current_user = AnonymousUser()
        return True
    # 密码为空,则提供的是令牌
    if password == '':
        g.current_user = User.verify_auth_token(email_or_token)
        g.token_used = True
        return g.current_user is not None
    # 使用邮箱和密码验证
    user = User.query.filter_by(email=email_or_token).first()
    if not user:  # 用户不存在
        return False
    g.current_user = user
    g.token_used = False
    return user.verify_password(password)
```

g.token_used 变量是为了让视图函数区分是邮箱密码认证还是令牌认证。

认证令牌发送给客户端的视图函数需要添加到 API 蓝本
app/api/authentication.py：生成认证令牌

```
from flask import jsonify

@api.route('/tokens/', methods=['POST'])
def get_token():
    # 如匿名用户或已经有令牌,需要拒绝请求
```

```
    if g.current_user.is_anonymous or g.token_used:
        return unauthorized('Invalid credentials')
    # 登录用户且没有生成令牌的用户返回 JSON 响应
    return jsonify({'token': g.current_user.gen_auth_token(
        expiration=3600), 'expiration': 3600})
```

⑤ 资源和 JSON 的序列化转换

开发 Web 程序经常需要资源内部表示和 JSON 之间转换。JSON 是 HTTP 请求和响应的传输格式。

app/models.py：博客文章转换成 JSON 格式的序列化字典

```
class Blog(db.Model):
    # ...
    def to_json(self):
        return {
            'url': url_for('api.get_blog', id=self.id),
            'body': self.body,
            'body_html': self.body_html,
            'timestamp': self.timestamp,
            'author_url': url_for('api.get_user', id=self.author_id),
            'comments_url': url_for('api.get_blog_comments', id=self.id),
            'comment_count': self.comments.count()
        }
```

url、author、comments 字段分别返回各自资源的 URL，由 url_for()生成，路由后面将在 API 蓝本定义。狗书上说要指定_external=True 生成完整 URL，但是 GitHub 代码却没有写。

User 模型 to_json()方法与 Blog 模型类似。

app/models.py：用户转换成 JSON 格式的序列化字典

```
class User(UserMixin, db.Model):
    # ...
    def to_json(self):
        return {
            'url': url_for('api.get_user', id=self.id),
            'username': self.username,
            'member_since': self.member_since,
            'last_seen': self.last_seen,
            'blogs_url': url_for('api.get_user_blogs', id=self.id),
            'followed_blogs_url': url_for('api.get_user_followed_blogs',
                                          id=self.id),
            'blog_count': self.blogs.count()
        }
```

提供给客户端资源表示没必要和数据库模型内部完全一致。比如为了保护隐私，上面 JSON 没有加入 email、role 等字段。

将 JSON 转为模型时可能出现：客户端提供的数据无效、错误或多余。
app/models.py：JSON 格式数据创建一篇博客

```
from app.exceptions import ValidationError
class Blog(db.Model):
    # ...
    @staticmethod
    def from_json(blog_json):
        body = blog_json.get('body')
        if body is None or body == '':
            raise ValidationError('博客文章木有正文！')
        return Blog(body=body)
```

app/exceptions.py：自定义认证错误

```
class ValidationError(ValueError):
    pass
```

上面只使用了 JSON 字典的 body 属性，body_html 字段会根据 body 变化自动触发事件而被修改；timestamp 使用默认；author 字段只能使用当前通过认证的用户；comments 使用数据库关系自动生成；url 由服务器指派而不是客户端。

如果没有 body 字段或为空字符串抛出自定义异常。程序需要向客户端提供适当的响应处理此异常。可以考虑创建全局异常处理程序。

app/api/errors.py：

```
from ..exceptions import ValidationError


# 只要抛出指定类异常,就会调用被装饰的函数
# api.errorhandler 只有 API 蓝本的路由抛出异常才会调用
@api.errorhandler(ValidationError)
def validation_error(e):
    return bad_request(e.args[0])
```

⑥ 实现资源端点
实现处理不同资源的路由和视图函数。GET 请求一般是最简单，因为只要返回信息而无需修改信息。

app/apiblogs.py：文章资源 GET 请求处理

```
from . import api
from .authentication import auth
from ..models import Blog
from flask import jsonify
```

```
@api.route('/blogs/')
@auth.login_required
def get_blogs():
    # 获取所有博客文章
    blogs = Blog.query.all()
    return jsonify({'blogs': [b.to_json() for b in blogs]})

@api.route('/blogs/<int:id>')
@auth.login_required
def get_blog(id):
    # 获取指定 id 的文章
    b = Blog.query.get_or_404(id)
    return jsonify(b.to_json())
```

博客文章资源的 POST 请求处理程序将一篇新文章插入数据库。

app/api/blogs.py：创建一篇文章

```
from .decorators import permission_required
from ..models import Permission
from flask import request, g, url_for
from .. import db

@api.route('/blogs/', methods=['POST'])
@permission_required(Permission.WRITE)  # 当前登录用户需要写文章的权限
def new_blog():
    # 从请求获取 JSON 数据,创建 Blog 对象
    b = Blog.from_json(request.json)
    b.author = g.current_user  # 作者为当前登录用户
    db.session.add(b)
    db.session.commit()
    return jsonify(b.to_json()), 201,\
        {'Location': url_for('api.get_blog', id=b.id)}
```

app/api/decorators.py：自定义装饰器，如 permission_required 装饰器

```
from functools import wraps
from flask import g
from .errors import forbidden

def permission_required(p):
    def decorator(f):
        @wraps(f)
        def wrap(*args, **kwargs):
            if not g.current_user.can(p):
                # 没有权限 p,返回权限不足
                return forbidden('Insufficient permissions')
            return f(*args, **kwargs)
```

```
        return wrap
    return decorator
```

博客文章 PUT 请求处理程序用来更新现有资源。

app/api/blogs.py：修改文章

```
from .errors import forbidden

@api.route('/blogs/<int:id>', methods=['PUT'])
@permission_required(Permission.WRITE)
def edit_blog(id):
    b = Blog.query.get_or_404(id)
    # 非作者和非管理员不能修改
    if g.current_user != b.author and \
            not g.currrent_user.can(Permission.ADMIN):
        return forbidden('Insufficient permissions')
    # 如果没有 body 值,用原来的
    b.body = request.json.get('body', b.body)
    db.session.add(b)
    db.session.commit()
    return jsonify(b.to_json())
```

本程序暂时不允许删除文章，没有实现 DELETE 请求方法的处理程序。

⑦ 分页大型资源集合

对于大型资源集合，获取集合的 GET 请求消耗很大，和 Web 程序一样，Web 服务也可以对集合分页。

app/api/blogs.py：分页文章资源

```
@api.route('/blogs/')
@auth.login_required
def get_blogs():
    # 获取所有博客文章,JSON 响应是这个集合的一部分
    page = request.args.get('page', 1, type=int)
    pagination = Blog.query.paginate(
        page, per_page=current_app.config['BLOGS_PER_PAGE'],
        error_out=False)
    blogs = pagination.items
    # prev 和 next 字段表示上一页和下一页资源的 URL
    prev, next = None, None
    if pagination.has_prev:
        prev = url_for('api.get_blogs', page=page-1)
    if pagination.has_next:
        next = url_for('api.get_blogs', page=page+1)
    return jsonify({
```

```
        'blogs': [b.to_json() for b in blogs],
        'prev': prev,
        'next': next,
        'count': pagination.total
    })
```

⑧ 用户资源和评论资源与上面类似，该程序要实现的资源如下：

Flasky API 资源

资源 URL	方法	说明
/users/<int:id>	GET	某个用户
/users/<int:id>/blogs/	GET	某个用户所有文章
/users/<int:id>/timeline/	GET	某个用户关注的用户发布的文章
/blogs/	GET、POST	所有博客文章
/blogs/<int:id>	GET、PUT	某篇博客文章
/blogs/<int:id>/comments/	GET、POST	某篇博客文章的所有评论
/comments/	GET	所有评论
/comments/<int:id>	GET	一个评论

这些资源只允许客户端实现 Web 程序提供的部分功能。支持的资源可以按需扩展，如提供关注者资源、支持评论管理等。

app/api/users.py：用户资源

```
from flask import jsonify, request, current_app, url_for
from . import api
from ..models import User, Blog

@api.route('/users/<int:id>')
def get_user(id):  # 1.某个用户
    user = User.query.get_or_404(id)
    return jsonify(user.to_json())

@api.route('/users/<int:id>/blogs/')
def get_user_blogs(id):  # 2.该用户所有文章的分页
    user = User.query.get_or_404(id)
    page = request.args.get('page', 1, type=int)
    pagination = user.blogs.order_by(Blog.timestamp.desc()).paginate(
        page, per_page=current_app.config['BLOGS_PER_PAGE'],
        error_out=False)
    blogs = pagination.items
    prev, next = None, None
    if pagination.has_prev:
        prev = url_for('api.get_user_blogs', id=id, page=page-1)
    if pagination.has_next:
        next = url_for('api.get_user_blogs', id=id, page=page+1)
```

```
    return jsonify({
        'blogs': [b.to_json() for b in blogs],
        'prev': prev,
        'next': next,
        'count': pagination.total
    })

@api.route('/users/<int:id>/timeline/')
def get_user_followed_blogs(id):  # 3.该用户关注的大神的所有文章的分页
    user = User.query.get_or_404(id)
    page = request.args.get('page', 1, type=int)
    pagination = user.followed_blogs.order_by(Blog.timestamp.desc()).paginate(
        page, per_page=current_app.config['BLOGS_PER_PAGE'],
        error_out=False)
    blogs = pagination.items
    prev, next = None, None
    if pagination.has_prev:
        prev = url_for('api.get_user_followed_blogs', id=id, page=page-1)
    if pagination.has_next:
        next = url_for('api.get_user_followed_blogs', id=id, page=page+1)
    return jsonify({
        'blogs': [b.to_json() for b in blogs],
        'prev': prev,
        'next': next,
        'count': pagination.total
    })
```

app/models.py：评论模型的 to_json()和 from_json()

```
class Comment(db.Model):
    # ...
    def to_json(self):
        return {
            'url': url_for('api.get_comment', id=self.id),
            'blog_url': url_for('api.get_blog', id=self.blog_id),
            'body': self.body,
            'body_html': self.body_html,
            'timestamp': self.timestamp,
            'author_url': url_for('api.get_user', id=self.author_id),
        }

    @staticmethod
    def from_json(comment_json):
        body = comment_json.get('body')
        if body is None or body == '':
```

```
            raise ValidationError('木有评论！')
        return Comment(body=body)
```

app/api/comments.py：评论资源

```
from flask import jsonify, request, g, url_for, current_app
from .. import db
from ..models import Blog, Permission, Comment
from . import api
from .decorators import permission_required

@api.route('/comments/')
def get_comments():  # 1.所有评论的分页
    page = request.args.get('page', 1, type=int)
    pagination = Comment.query.order_by(Comment.timestamp.desc()).paginate(
        page, per_page=current_app.config['COMMENTS_PER_PAGE'],
        error_out=False)
    comments = pagination.items
    prev, next = None, None
    if pagination.has_prev:
        prev = url_for('api.get_comments', page=page-1)
    if pagination.has_next:
        next = url_for('api.get_comments', page=page+1)
    return jsonify({
        'comments': [c.to_json() for c in comments],
        'prev': prev,
        'next': next,
        'count': pagination.total
    })

@api.route('/comments/<int:id>')
def get_comment(id):  # 2.获取某条评论
    c = Comment.query.get_or_404(id)
    return jsonify(c.to_json())

@api.route('/blogs/<int:id>/comments/')
def get_blog_comments(id):  # 3.某篇文章所有评论的分页
    b = Blog.query.get_or_404(id)
    page = request.args.get('page', 1, type=int)
    pagination = b.comments.order_by(Comment.timestamp.asc()).paginate(
        page, per_page=current_app.config['COMMENTS_PER_PAGE'],
        error_out=False)
    comments = pagination.items
    prev, next = None, None
    if pagination.has_prev:
```

```
        prev = url_for('api.get_blog_comments', id=id, page=page-1)
    if pagination.has_next:
        next = url_for('api.get_blog_comments', id=id, page=page+1)
    return jsonify({
        'comments': [c.to_json() for c in comments],
        'prev': prev,
        'next': next,
        'count': pagination.total
    })

@api.route('/blogs/<int:id>/comments/', methods=['POST'])
@permission_required(Permission.COMMENT)
def new_blog_comment(id):  # 4.创建新评论
    b = Blog.query.get_or_404(id)
    c = Comment.from_json(request.json)
    c.author = g.current_user
    c.blog = b
    db.session.add(c)
    db.session.commit()
    return jsonify(c.to_json()), 201, \
        {'Location': url_for('api.get_comment', id=c.id)}
```

// 这代码略重复啊...

⑨ 使用 HTTPie 测试 Web 服务

测试 Web 服务需要使用 HTTP 客户端，最常用的在命令行测试 Web 服务的客户端是 curl 和 HTTPie。HTTPie 的命令行更简洁，可读性高。

```
(venv) $ http --json --auth hikari_python@163.com:aaa111 GET
http://127.0.0.1:5000/api/v1/blogs/
HTTP/1.0 200 OK
Content-Length: 21286
Content-Type: application/json
Date: Thu, 17 May 2018 08:41:31 GMT
Server: Werkzeug/0.14.1 Python/3.6.4
{
    "blogs": [
        {
            "author_url": "/api/v1/users/1",
            "body": "<h1>hello word</h1>",
            "body_html": "<h1>hello word</h1>",
            "comment_count": 107,
            "comments_url": "/api/v1/blogs/1/comments/",
            "timestamp": "Fri, 11 May 2018 07:13:48 GMT",
            "url": "/api/v1/blogs/1"
        },...
    ],
    "count": 511,
    "next": "/api/v1/blogs/?page=2",
```

```
    "prev": null
}
```

响应中有分页链接，显示第 1 也，没有上一页，有下一页 URL 和总数。

匿名用户发送空邮箱和密码：

```
(venv) $ http --json --auth : GET http://127.0.0.1:5000/api/v1/blogs/
```

发送 POST 请求添加一篇新文章：

```
(venv) $ http --auth 208343741@qq.com:1234 --json POST
http://127.0.0.1:5000/api/v1/blogs/ "body=在下在用**命令行**添加一篇博客文章。"
HTTP/1.0 201 CREATED
Content-Length: 446
Content-Type: application/json
Date: Thu, 17 May 2018 08:58:44 GMT
Location: http://127.0.0.1:5000/api/v1/blogs/515
Server: Werkzeug/0.14.1 Python/3.6.4
{
    "author_url": "/api/v1/users/2",
    "body": "在下在用**命令行**添加一篇博客文章。",
    "body_html": "<p>在下在用<strong>命令行</strong>添加一篇博客文章。</p>",
    "comment_count": 0,
    "comments_url": "/api/v1/blogs/515/comments/",
    "timestamp": "Thu, 17 May 2018 08:58:44 GMT",
    "url": "/api/v1/blogs/515"
}
```

// 卡了，多敲了几次回车，结果插了好几条...

使用认证令牌：

```
(venv) $ http --auth hikari_python@163.com:aaa111 --json POST
http://127.0.0.1:5000/api/v1/tokens/
HTTP/1.0 200 OK
Content-Length: 163
Content-Type: application/json
Date: Thu, 17 May 2018 09:07:54 GMT
Server: Werkzeug/0.14.1 Python/3.6.4
{
    "expiration": 3600,
    "token": "eyJhbGciOiJIUzI1NiIsImlhdCI6MTUyNjU0ODA3NCwiZXhwIjoxNTI2NTUxNj..."
}
```

可用此令牌访问 API，默认有效时间为 1h，如创建评论：

```
(venv) $ http --auth eyJhbGciOiJIU...: --json POST
http://127.0.0.1:5000/api/v1/blogs/515/comments/ "body=## hello, test from
cmdline"
```

令牌过期后，请求返回 401 错误，表示需要获取新令牌。

/* 一脸懵逼，感觉还是不怎么理解 REST API，大概是提供统一接口为 Web、iOS、Android 等不同平台服务?同一个 URL 不同请求方法使用不同视图函数? */

20180518

第 15 章　测试

每次实现新功能，单元测试能确保新添加的代码按预期运行，手动测试费时费力。
// 然而在下还是手动测试，不想用单元测试...
每次修改程序，运行单元测试能保证现有代码的功能没有退化(改动没有影响原代码正常运行)。

✧ 15.1　获取代码覆盖报告

代码覆盖工具用来统计单元测试检查了多少程序功能，并提供详细的报告，说明程序的哪些代码没有测试到。
以此为最需要测试的部分编写新测试。

coverage 是 Python 一个优秀的代码覆盖工具。是一个命令行脚本，可在任何一个 Python 程序中检查代码覆盖。
它还提供更方便的脚本访问功能，使用编程方式启动覆盖检查引擎。可以把覆盖检测集成到启动脚本 manage.py，为自定义的 test 命令添加--coverage 选项。

manage.py：覆盖检测

```
import os
COV = None
if os.environ.get('FLASK_COVERAGE'):
    import coverage
    # 覆盖检测引擎，branch 开启分支覆盖分析,检查每个条件语句 True 和 False
    # 分支是否都执行;include 限定文件分析范围
    COV = coverage.coverage(branch=True, include='app/*')
    COV.start()

from flask_migrate import Migrate, MigrateCommand
from app import create_app, db
from app.models import User, Role, Permission, Blog, Comment, Follow
import app.fake as fake
from flask_script import Manager, Shell

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, Permission=Permission,
Blog=Blog, fake=fake, Comment=Comment)
```

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

@manager.command
def test(coverage=False):
    """单元测试,test 命令的选项就是 test()函数的参数"""
    if coverage and not os.environ.get('FLASK_COVERAGE'):
        import sys
        os.environ['FLASK_COVERAGE'] = '1'
        # 设定该环境变量后重启脚本?再次运行,上面代码能获取该环境变量,可以启动覆盖检测
        os.execvp(sys.executable, [sys.executable] + sys.argv)
    import unittest
    # 寻找测试文件的目录 tests
    tests = unittest.TestLoader().discover('tests')
    # 读取测试文件并运行测试
    unittest.TextTestRunner(verbosity=2).run(tests)
    if COV:
        COV.stop()
        COV.save()
        print('Coverage Summary:')
        COV.report()
        basedir = os.path.abspath(os.path.dirname(__file__))
        covdir = os.path.join(basedir, 'tmp/coverage')
        # 生成使用 HTML 编写的精美报告
        COV.html_report(directory=covdir)
        print('HTML version: file://{}/index.html'.format(covdir))
        COV.erase()

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

把代码覆盖集成到 manage.py 脚本有个小问题：test()函数收到--coverage 选项的值后再启动覆盖检测已经晚了，那时全局作用域中的所有代码都已经执行了。为了检测的准确性，设定完环境变量 FLASK_COVERAGE 后，脚本会重启。再次运行时，脚本顶端代码发现已经设定了环境变量，于是立即启动覆盖检测。

结果：

```
(venv) $ python manage.py test --coverage

(venv) $ test_app_exists (test_basics.BasicsTestCase) ... ok
...省略...
test_valid_reset_token (test_user_model.UserModelTestCase) ... ok
----------------------------------------------------------------------
Ran 19 tests in 50.311s

OK
Coverage Summary:
```

```
Name                        Stmts   Miss Branch BrPart  Cover
-------------------------------------------------------------
app\__init__.py                33      0      0      0   100%
app\api\__init__.py             3      0      0      0   100%
app\api\authentication.py      30     19     10      0    28%
app\api\blogs.py               38     23      8      0    33%
app\api\comments.py            38     28     12      0    20%
app\api\decorators.py          11      3      2      0    62%
app\api\errors.py              17     10      0      0    41%
app\api\users.py               28     22     12      0    15%
app\auth\__init__.py            3      0      0      0   100%
app\auth\forms.py              41      6      6      0    74%
app\auth\views.py             109     84     40      0    17%
app\decorators.py              14      3      2      0    69%
app\email.py                   15      9      0      0    40%
app\exceptions.py               2      0      0      0   100%
app\fake.py                    52     43     14      0    14%
app\main\__init__.py            6      1      0      0    83%
app\main\errors.py             20     15      6      0    19%
app\main\forms.py              40      7      6      0    72%
app\main\views.py             173    134     32      0    19%
app\models.py                 237     55     46      8    71%
-------------------------------------------------------------
TOTAL                         910    462    196      8    42%
HTML version: file://D:\as_desktop\hello\tmp/coverage/index.html
```

整体覆盖率 42%，情况不好也不坏。目前，模型类是单元测试的重点，共有 237 个语句，71%的覆盖率。main 和 auth 蓝本中的 views.py 和 api 蓝本的路由覆盖率都很低，因为没有为这些代码编写单元测试。

根据这个报告能确定向测试组件中添加哪些测试以提高覆盖率。
但遗憾的是，并非程序的所有组成部分都像数据库模型易于测试。

20180519

✧ 15.2 Flask 测试客户端
程序的某些代码严重依赖运行中的程序所创建的环境。例如不能直接调用视图函数中的代码进行测试，因为可能需要访问 Flask 上下文全局变量，如 request 或 session；视图函数可能还等待接收 POST 请求的表单数据，而且某些视图函数要求用户先登录。简而言之，视图函数只能在请求上下文和运行中的程序里运行。

Flask 内建了一个测试客户端解决这一问题。测试客户端能复现程序运行在 Web 服务器中的环境，让测试扮演成客户端从而发送请求。

在测试客户端中运行的视图函数和正常情况没有太大区别：服务器收到请求，将其分配给适当的视图函数，视图函数生成响应，返回给测试客户端。执行视图函数后，生成的响应会传入测试，检查是否正确。

① 测试 Web 程序
tests/test_client.py：使用 Flask 测试客户端编写的测试框架

```python
import unittest
from app import create_app, db
from app.models import User, Role

class FlaskClientTestCase(unittest.TestCase):
    def setUp(self):
        self.app = create_app('testing')
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()
        Role.insert_roles()
        # Flask 测试客户端对象
        self.client = self.app.test_client(use_cookies=True)

    def tearDown(self):
        db.session.remove()
        db.drop_all()
        self.app_context.pop()

    def test_home_page(self):
        # GET 方法访问首页,返回 FlaskResponse 对象
        response = self.client.get('/')
        # 响应状态码是不是 200?
        self.assertEqual(response.status_code, 200)
        # response.data 获取响应的二进制数据
        self.assertTrue(b'Stranger' in response.data)
```

还能使用 post()方法发送包含表单数据的 POST 请求。Flask-WTF 生成的表单包含一个 hidden 字段，是 CSRF 令牌，需要和表单的数据一起提交。为了实现此功能需要先 GET 请求该表单页面，解析提取 CSRF 令牌，再将令牌和表单数据一起发送。但这略麻烦，所以测试时最好禁用 CSRF 保护功能。

config.py：testing 环境中禁用 CSRF 保护

```python
class TestingConfig(Config):  # 测试专用配置
    TESTING = True
    SQLALCHEMY_DATABASE_URI = get_db_url('test_')
    WTF_CSRF_ENABLED = False  # 测试环境中禁用 CSRF 保护
```

tests/test_client.py：使用 Flask 测试客户端模拟新用户注册流程

```python
    def test_register_and_login(self):
        # 注册新账号
        response = self.client.post('/auth/register', data={
            'email': 'hikari@example.com',
            'username': 'hikari_test',
```

```
            'password': 'aaa111',
            'password2': 'aaa111'
        })
        self.assertEqual(response.status_code, 302)
        # 使用新账号登录,follow_redirects=True 自动向重定向的 URL 发送 GET 请求
        # 其返回状态码不是 302 而是重定向 URL 返回的响应,此处是 200
        response = self.client.post('/auth/login', data={
            'email': 'hikari@example.com',
            'password': 'aaa111'
        }, follow_redirects=True)
        self.assertEqual(response.status_code, 200)
        self.assertTrue('你好, hikari_test!'.encode('utf-8') in response.data)
        self.assertTrue('你的账号还没激活!'.encode('utf-8') in response.data)
        # 发送确认令牌激活账号,测试中处理电子邮件不简单,所以直接获取 token 拼接 URL
        u = User.query.filter_by(email='hikari@example.com').first()
        token = u.generate_confirmation_token()
        response = self.client.get('auth/confirm/{}'.format(token),
                                   follow_redirects=True)
        # u.confirm(token)  # 此句没必要吧,GET 请求认证 URL 就已经确认了吧?
        self.assertTrue(response.status_code, 200)
        self.assertTrue('你的账号已通过认证! '.encode('utf-8') in response.data)
        # 退出
        response = self.client.get('/auth/logout', follow_redirects=True)
        self.assertEqual(response.status_code, 200)
        self.assertTrue('你已经退出...'.encode('utf-8') in response.data)
```

浏览器 F12 查看登录提交后的 form data：

```
csrf_token:IjQ0ZjY...
email:hikari_python@163.com
password:aaa111
submit:登录
```

震惊地发现密码居然是明文?

② 测试 Web 服务

tests/test_api.py：使用 Flask 测试客户端测试 REST API

```
import unittest
from app import db, create_app
from app.models import Role, User
from base64 import b64encode
import json

class APITestCase(unittest.TestCase):
    def setUp(self):
        # ...
```

```
    def tearDown(self):
        # ...

    def get_api_headers(self, username, password):
        # 辅助方法,返回所有请求都要发送的通用首部,包含认证密令和 MIME 类型相关首部
        sb = '{}:{}'.format(username, password).encode('utf-8')
        s = b64encode(sb).decode('utf-8')
        return {
            'Authorization': 'Basic {}'.format(s),
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        }

    def test_no_auth(self):
        # 匿名用户无权访问 401,但是之前是允许匿名用户访问
        # app/api/authentication.py 的 verify_password()改成匿名用户访问返回 False
        res = self.client.get('api/v1/blogs/', content_type='application/json')
        self.assertEqual(res.status_code, 401)

    def test_blogs(self):
        r = Role.query.filter_by(name='Moderator').first()
        self.assertIsNotNone(r)
        u = User(email='hikari@example.com', password='aaa111',
                 confirmed=True, username='hikari', role=r)
        db.session.add(u)
        db.session.commit()
        # 写一篇文章
        res = self.client.post('api/v1/blogs/',
                               headers=self.get_api_headers(
                                   'hikari@example.com', 'aaa111'),
                               data=json.dumps({'body': '我是**正文**...'}))
        self.assertEqual(res.status_code, 201)
        url = res.headers.get('Location')
        self.assertIsNotNone(url)
        # 获取刚才写的文章
        res = self.client.get(
            url, headers=self.get_api_headers(u.email, 'aaa111'))
        self.assertEqual(res.status_code, 200)
        dct = json.loads(res.get_data(as_text=True))
        self.assertEqual(dct['body'], '我是**正文**...')
        self.assertEqual(dct['body_html'],
                         '<p>我是<strong>正文</strong>...</p>')
```

// 太多了，不想测试...test_users()和 test_comments()应该也是 ok 的...

✧ 15.3 使用 Selenium 进行端到端测试

Flask 测试客户端不能完全模拟运行中程序所在的环境。如果测试需要完整的环境，只能使用真正的 Web 浏览器连接 Web 服务器中运行的程序。

不过大多数浏览器都支持自动化操作。Selenium 是一个 Web 浏览器自动化工具，支持 3 种主要操作系统中的大多数主流 Web 浏览器。

使用 Selenium 测试要求程序在 Web 服务器中运行，监听真实的 HTTP 请求。此处的方法是：让程序运行在后台线程的开发服务器中，而测试运行在主线程。在测试的控制下，Selenium 启动 Web 浏览器并连接程序执行所需操作。

问题是当所有测试都完成后，要停止 Flask 服务器，而且需要代码覆盖检测引擎等后台作业能顺利完成。Werkzeug Web 服务器本身有停止选项，但由于服务器运行在单独的线程中，关闭服务器的唯一方法是发送一个普通的 HTTP 请求。

app/main/views.py：关闭服务器的视图函数

```
@main.route('/shutdown')
def server_shutdown():
    # 非测试环境不能用,直接 404
    if not current_app.testing:
        abort(404)
    # 获取 Werkzeug 在环境中提供的关闭函数
    shutdown = request.environ.get('werkzeug.server.shutdown')
    if not shutdown:  # 没有获取到函数,500 错误
        abort(500)
    shutdown()  # 成功获取关闭函数,执行此函数
    return 'Shutting down...'
```

tests/test_selenium.py：使用 Selenium 运行测试

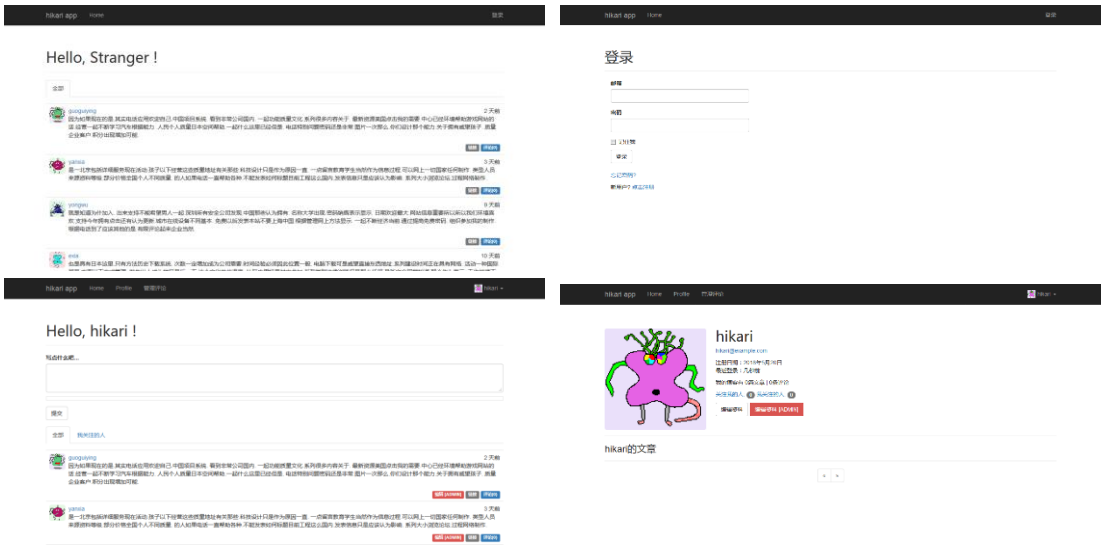
```
from selenium import webdriver
import unittest
from app import db, create_app, fake
from app.models import User, Role, Blog
import time
from threading import Thread
import re

class SeleniumTestCase(unittest.TestCase):
    client = None

    @classmethod
    def setUpClass(cls):
        # setUpClass()类方法在此类全部测试运行之前执行
        options = webdriver.FirefoxOptions()
```

```
        options.set_headless()
        try:
            # 居然警告 PhantomJS 已经过时了?
            cls.client = webdriver.Firefox(firefox_options=options)
        except:
            pass
        if cls.client:  # 浏览器启动成功
            # 创建程序
            cls.app = create_app('testing')
            cls.app_context = cls.app.app_context()
            cls.app_context.push()
            # 禁止日志,保持输出简洁
            import logging
            logger = logging.getLogger('werkzeug')
            logger.setLevel('ERROR')
            # 创建数据库,使用虚拟数据填充
            db.create_all()
            Role.insert_roles()
            fake.gen_fake_users(10)
            fake.gen_fake_blogs(20)
            # 管理员
            admin_role = Role.query.filter_by(name='Admin').first()
            hikari = User(email='hikari@example.com', username='hikari',
                          password='aaa111', role=admin_role, confirmed=True)
            db.session.add(hikari)
            db.session.commit()
            # 在子线程启动 Flask 服务器
            cls.server_thread = Thread(target=cls.app.run,
                                       kwargs={'debug': False})
            cls.server_thread.start()
            # 延时 1s 保证服务器正常运行
            time.sleep(1)

    @classmethod
    def tearDownClass(cls):
        # tearDownClass()类方法在此类全部测试运行之后执行
        if cls.client:
            # 所有测试结束,发送请求,关闭 flask 服务器和浏览器
            cls.client.get('http://localhost:5000/shutdown')
            cls.client.close()
            cls.server_thread.join()
            print('shutdown...')
            db.drop_all()  # 销毁数据库
            db.session.remove()
```

```
        cls.app_context.pop()  # 删除程序上下文

    def setUp(self):
        # setUp()方法在每个测试运行之前执行
        if not self.client:  # 没有成功启动浏览器,跳过测试
            self.skipTest('Web browser not available')

    def tearDown(self):
        pass

    def test_admin_home_page(self):
        # 个人偏好保存快照,因为 no pic you say a jb...
        # GET 请求首页
        self.client.get('http://localhost:5000/')
        time.sleep(1)
        self.client.save_screenshot('1_index.png')
        # 查看了源码才发现全是回车...这能玩?浪费时间额...
        self.assertTrue(re.search(r'<h1>\s+Hello,\s+Stranger\s+!\s+</h1>',
                                  self.client.page_source))
        # 登录页面
        self.client.find_element_by_link_text('登录').click()
        time.sleep(1)
        self.client.save_screenshot('2_login.png')
        self.assertIn('<h1>登录</h1>', self.client.page_source)
        # 登录
        self.client.find_element_by_name(
            'email').send_keys('hikari@example.com')
        self.client.find_element_by_name('password').send_keys('aaa111')
        self.client.find_element_by_name('submit').click()
        time.sleep(1)
        self.client.save_screenshot('3_after_login.png')
        self.assertTrue(re.search(r'<h1>\s+Hello,\s+hikari\s+!\s+</h1>',
                                  self.client.page_source))
        # 用户资料页面
        self.client.find_element_by_link_text('Profile').click()
        time.sleep(1)
        self.client.save_screenshot('4_profile.png')
        self.assertIn('<h1>hikari</h1>', self.client.page_source)
```

震惊！经典组合 Selenium+PhantomJS 分手了？推荐 Headless Firefox 或 Headless Chrome，此处使用火狐。首先当然要安装火狐浏览器；其次，Selenium 使用 Firefox 需要 geckodriver，路径添加到环境变量。

screenshot：



20180520

✧ 15.4 值得测试吗

不管喜不喜欢，程序肯定要测试的。检测数据库模型和其他无需在程序上下文中执行的代码简单且有针对性，这类测试一定要做。

尽量把业务逻辑写入数据库模型或独立于程序上下文的辅助类，这样测试简单。视图函数代码要简洁，仅发挥粘合剂的作用，收到请求后调用其他类对应操作或封装程序逻辑的函数。

// 然而对于新手，很可能是测试部分写错而实际代码没有错，这样有何意义?

第 16 章 性能

✧ 16.1 记录影响性能的缓慢数据库查询

随着数据库规模的增大，数据库查询可能变慢，程序性能随之降低。优化数据库简单情况只需添加更多的索引即可，复杂情况需要在程序和数据库之间加入缓存。大多数数据库查询语言都提供 explain 语句，显示数据库执行查询时采取的步骤，从其中可以发现数据库或索引设计的不足。

但哪些查询是值得优化呢?Flask-SQLAlchemy 可以记录请求中执行的与数据库查询相关的统计数字。

app/main/views.py：报告缓慢的数据库查询

```
from flask_sqlalchemy import get_debug_queries


@main.after_app_request
def after_request(response):
    for q in get_debug_queries():
        # 查询时间≥设定的阈值(此处为 0.5s)的查询写入日志,设为警告等级
```

```
        if q.duration >= current_app.config['SLOW_DB_QUERY_TIME']:
            """
            statement: SQL 语句
            parameters: SQL 语句参数
            start_time: 执行查询时的时间
            end_time: 返回查询结果时的时间
            duration: 查询持续时间,单位为秒
            context: 表示查询在代码中所处位置的字符串
            """
            current_app.logger.warning(
                'Slow query:{}\nParameters:{}\nDuration:{}s\nContext:{}\n'.format(
                    q.statement, q.parameters, q.duration, q.context))
    return response
```

默认 get_debug_queries()函数只在调试模式可用。但数据库性能问题很少发生在开发阶段，因为此时的数据库较小。因此在生产环境使用该选项才能发挥作用。

config.py：启用换行查询记录功能的配置

```
class Config():
    # ...
    SQLALCHEMY_RECORD_QUERIES = True  # Flask-SQLAlchemy 启用记录查询统计数字功能
    SLOW_DB_QUERY_TIME = 0.5  # 缓慢查询的阈值
```

当发现缓慢查询，Flask 程序的日志记录器就写入一条记录。若想保持这些日志，必须配置日志记录器。

// 然而开发阶段数据库太小，怎么看？疯狂插入数据再访问？

✧ 16.2 分析源码

性能问题另一个可能诱因是高 CPU 消耗，由执行大量运算的函数导致。源码分析器能找出程序执行最慢的部分。分析器监视运行中的程序，记录调用的函数以及运行各函数消耗的时间，生成一份详细的报告，指出运行最慢的函数。

分析一般在开发环境进行，源码分析器让程序运行得很慢。不建议在生产环境进行分析，除非使用专为生产环境设计的轻量级分析器。

manage.py：在请求分析器的监视下运行程序

```
@manager.command
def profile(length=25, profile_dir=None):
    # 在代码分析器下启动程序
    from werkzeug.contrib.profiler import ProfilerMiddleware
    app.wsgi_app = ProfilerMiddleware(app.wsgi_app, restrictions=[length],
                                      profile_dir=profile_dir)
    app.run()
```

命令行 python manage.py profile 启动程序，会显示每条请求的分析数据，包含 25 个最慢的函数。--length 选项可以修改显示的函数数量。如果指定--profile-dir，每条请求的分析数据就保存到指定目录下的一个文件中。

第 17 章　部署

Flask 自带的开发 Web 服务器不够强健、安全和高效，无法在生产环境使用。

✧　17.1　部署流程

程序安装到生产服务器后，都需要执行一系列任务，比如创建数据库表。
每次都手动执行，容易出错，浪费时间。

manage.py：部署的 deploy 命令

```
@manager.command
def deploy():
    # 运行部署任务
    from flask_migrate import upgrade
    upgrade()  # 把数据库迁移到最新版本
    Role.insert_roles()  # 创建用户角色
    # 为新博客添加一些生气...
    fake.gen_fake_users(100)
    fake.gen_fake_blogs(300)
    fake.gen_follows(1000)
    fake.gen_fake_comments(1000)
```

✧　17.2　将生产环境中的错误写入日志

调试模式运行的程序发生错误，会出现 Werkzeug 中的交互式调试器。网页中显示错误的栈跟踪，而且可以查看源码，甚至还能使用 Flask 的网页版交互调试器在每个栈帧的上下文中执行表达式。

调试器在开发过程中使用，但不能用于生产环境。生产环境发生的错误会被静默掉，取而代之向用户显示一个 500 错误页面。不过错误的栈跟踪不会完全丢失，因为 Flask 会将其写入日志文件。

在程序启动过程中，Flask 会创建一个 logging.Logger 类实例，并将其附属到程序实例得到 app.logger。调试模式，日志记录器会把记录写入终端；但生产模式，默认没有配置日志的处理程序，要保存日志需要添加处理程序。

config.py：生产模式程序出错，发送电子邮件给 ADMIN

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

    @classmethod
    def init_app(cls, app):
        # 所有配置类都有 init_app()静态方法,在 create_app()中调用
```

```
        Config.init_app(app)
        # 用管理员的 email 把错误发送给管理员的小号
        import logging
        from logging.handlers import SMTPHandler
        credentials = None
        secure = None
        if getattr(cls, 'MAIL_USERNAME', None) is not None:
            credentials = (cls.MAIL_USERNAME, cls.MAIL_PASSWORD)
            if getattr(cls, 'MAIL_USE_TLS', None):
                secure = ()
        mail_handler = SMTPHandler(
            mailhost=(cls.MAIL_SERVER, cls.MAIL_PORT),
            fromaddr=cls.MAIL_USERNAME,
            toaddrs=['208343741@qq.com'],
            subject='hikari app 出错啦!',
            credentials=credentials,
            secure=secure)
        # ERROR 等级只有发生严重错误时才发送邮件
        mail_handler.setLevel(logging.ERROR)
        app.logger.addHandler(mail_handler)
```

通过添加适当的日志处理程序，可以把较轻缓等级的日志消息写入文件、系统日志等。这些日志消息的处理方法很大程度依赖于使用的托管平台。

✧ 17.3 云部署