**20180502**

**第1章 Flask安装**

[Flask](http://flask.pocoo.org/)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](http://werkzeug.pocoo.org/)提供；模板系统由[Jinja2](http://werkzeug.pocoo.org/)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'&lt;h1&gt;maki&lt;/h1&gt;'，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](http://momentjs.com/)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：



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](http://momentjs.com/docs/#/displaying/)学习全部格式化选项。

**第4章 Web表单**

[Flask-WTF](http://pythonhosted.org/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](http://wtforms.simplecodes.com)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 %} |



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](http://pythonhosted.org/Flask-SQLAlchemy/)，其集成了[SQLAlchemy](http://www.sqlalchemy.org/)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](https://alembic.readthedocs.org/en/latest/index.html)；[Flask-Migrate](http://flask-migrate.readthedocs.org/en/latest/)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](http://www.celeryproject.org/)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：



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：

|  |
| --- |
| import os  *# flask程序根目录*  basedir = os.path.abspath(os.path.dirname(\_\_file\_\_))  my\_mail, my\_pwd = None, None  with open('mail.hikari') as f:  data = eval(f.read())  my\_mail = data.get('mail') *# xxx@qq.com*  my\_pwd = data.get('password') *# 授权码, 不是密码*  def get\_db\_url(a):  return os.environ.get('{}DATABASE\_URL'.format(a.upper())) or 'sqlite:///{}'.format(os.path.join(basedir, '{}data.sqlite'.format(a)))  class Config(): *# 父类通用配置*  *# 敏感信息从环境变量获取, 但还是给了默认值*  SECRET\_KEY = os.environ.get('SECRET\_KEY') or 'hoshizora rin'  MAIL\_SERVER = os.environ.get('MAIL\_SERVER') or 'smtp.qq.com'  MAIL\_PORT = int(os.environ.get('MAIL\_PORT', 465)) *# SMTP的加密SSL端口*  MAIL\_USE\_SSL = True *# SSL*  MAIL\_USERNAME = os.environ.get('MAIL\_USERNAME') or my\_mail  MAIL\_PASSWORD = os.environ.get('MAIL\_PASSWORD') or my\_pwd  SQLALCHEMY\_COMMIT\_ON\_TEARDOWN = True  SQLALCHEMY\_TRACK\_MODIFICATIONS = False  FLASKY\_ADMIN = os.environ.get('FLASKY\_ADMIN') or 'hikari\_python@163.com'  @staticmethod  def init\_app(app): *# 对当前环境配置初始化*  pass  *# 3种环境使用不同数据库*  class DevelopmentConfig(Config): *# 开发专用配置*  DEBUG = True  SQLALCHEMY\_DATABASE\_URI = get\_db\_url('dev\_')  class TestingConfig(Config): *# 测试专用配置*  TESTING = True  SQLALCHEMY\_DATABASE\_URI = get\_db\_url('test\_')  class ProductionConfig(Config): *# 生产专用配置*  SQLALCHEMY\_DATABASE\_URI = get\_db\_url('')  config = {  'development': DevelopmentConfig,  'testing': TestingConfig,  'production': ProductionConfig,  'default': DevelopmentConfig *# 默认开发环境的配置*  } |

* 7.3 程序包

程序包用来保存程序所有代码、模块和静态文件，这个包可以称为app (应用)。templates和static目录是程序包一部分，需要移到app目录；数据库模型和电子邮件支持函数model.py和email.py也需要移到app目录。

① 使用程序工厂函数

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

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

示例：程序包的构造文件app/\_\_init\_\_.py

|  |
| --- |
| from flask import Flask  from flask\_bootstrap import Bootstrap  from flask\_mail import Mail  from flask\_moment import Moment  from flask\_sqlalchemy import SQLAlchemy  from config import config  bootstrap = Bootstrap()  mail = Mail()  moment = Moment()  db = SQLAlchemy()  def create\_app(config\_name): *# 工厂函数, 参数为配置名*  app = Flask(\_\_name\_\_)  app.config.from\_object(config[config\_name]) *# 导入配置*  config[config\_name].init\_app(app)  bootstrap.init\_app(app)  mail.init\_app(app)  moment.init\_app(app)  db.init\_app(app)  *# 附加路由和自定义的错误页面*  return app *# 返回创建的程序示例* |

② 在蓝本中实现程序功能

转换成工厂函数的操作让定义路由变得复杂了，因为只有调用create\_app()后才能使用app.route装饰器(错误页面处理用app.errorhandler装饰器)，这时定义路由太晚了。

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

蓝本可以在单个文件中定义，也可以定义成包，包中多个模块，比如在app包中创建子包main，用于保存蓝本。

示例1：创建蓝本：app/main/\_\_init\_\_.py

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

蓝本在工厂函数create\_app()中注册到程序。

示例2：注册蓝本：app/\_\_init\_\_.py：

|  |
| --- |
| def create\_app(config\_name):  *# ...*  from .main import main as main\_blueprint  app.register\_blueprint(main\_blueprint)  return app *# 返回创建的程序示例* |

示例3：蓝本中的错误处理：app/main/errors.py：

|  |
| --- |
| from flask import render\_template  from . import main  *# 如果使用errorhandler装饰器, 只能处理蓝本中的错误;*  *# 使用app\_errorhandler装饰器, 注册全局错误处理*  @main.app\_errorhandler(404)  def page\_not\_found(e):  return render\_template('404.html'), 404  @main.app\_errorhandler(500)  def internal\_server\_error(e):  return render\_template('500.html'), 500 |

示例4：蓝本中定义路由：app/main/views.py

|  |
| --- |
| from flask import render\_template, session, redirect, url\_for, current\_app  from .. import db  from ..models import User  from ..email import send\_mail  from . import main  from .forms import NameForm  @main.route('/', methods=['GET', 'POST']) *# 支持GET和POST*  def index():  form = NameForm()  if form.validate\_on\_submit(): *# 提交表单,数据被验证函数接受*  user = User.query.filter\_by(username=form.name.data).first() *# 数据库查询*  if user is None: *# 数据库不存在用户则添加*  user = User(username=form.name.data)  db.session.add(user)  session['known'] = False  admin = current\_app.config['FLASKY\_ADMIN']  if admin:  send\_mail(admin, 'New User', 'mail/new\_user', user=user)  else:  session['known'] = True  session['name'] = form.name.data *# 获取字段data属性存入session*  *# 蓝本端点自动添加命名空间, 为main.index; 当前蓝本简写为.index*  return redirect(url\_for('.index')) *# 重定向, GET方式请求index*  return render\_template('index.html', form=form, name=session.get('name'), known=session.get('known', False)) |

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

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

示例5：蓝本中的表单类：app/main/forms.py：直接复制

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

示例6：app/email.py

|  |
| --- |
| from threading import Thread  from flask import current\_app, render\_template  from flask\_mail import Message  from . import mail *# \_\_init\_\_.py中的mail对象*  *# 异步发送邮件*  def send\_async\_mail(app, msg):  with app.app\_context():  mail.send(msg)  def send\_mail(to, subject, template, \*\*kwargs):  *# to为接收方,subject为邮件主题,template为渲染邮件正文的模板*  app = current\_app.\_get\_current\_object() *# 将app传给子线程?*  msg = Message(  subject, sender=app.config['FLASKY\_MAIL\_SENDER'], recipients=[to])  msg.body = render\_template(template + '.txt', \*\*kwargs)  msg.html = render\_template(template + '.html', \*\*kwargs)  t = Thread(target=send\_async\_mail, args=[app, msg])  t.start()  return t |

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

|  |
| --- |
| from . import db |

将static目录和templates目录移到app目录

7.4 启动脚本

manage.py用于启动程序：

|  |
| --- |
| import os  from app import create\_app, db  from app.models import User, Role  from flask\_script import Manager, Shell  from flask\_migrate import Migrate, MigrateCommand  app = create\_app(os.getenv('FLASK\_CONFIG') or 'default')  manager = Manager(app)  migrate = Migrate(app, db)  def make\_shell\_context():  return dict(app=app, db=db, User=User, Role=Role)  manager.add\_command("shell", Shell(make\_context=make\_shell\_context))  manager.add\_command('db', MigrateCommand)  if \_\_name\_\_ == '\_\_main\_\_':  manager.run() |

* 7.5 需求文件

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

自动生成需求文件：

|  |
| --- |
| (venv) $ pip freeze >requirements.txt |

* 7.6 单元测试

这个程序很小，没什么可测试的。

使用Python标准库的[unittest](https://docs.python.org/3.6/library/unittest.html)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 |