Qichen-Kylin / **Python3-WebApp**

基于Python3的Web开发

Edit

Manage topics

62 commits | 13 branches | 0 packages | 0 releases | 1 contributor | View license

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

Qichen-Kylin Update README.md — Latest commit 08a503e 2 minutes ago

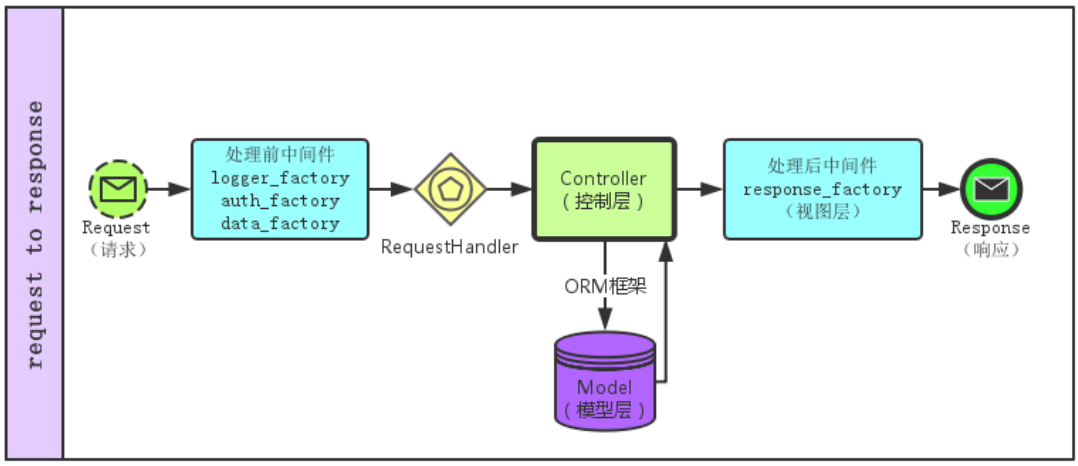
.github/workflows	Create pythonpackage.yml	2 months ago
.idea	dev-10	9 days ago
conf	dev-15	8 minutes ago
www	dev-15	8 minutes ago
.gitignore	调通Api	29 days ago
GitPsuh.sh	update	2 months ago
LICENSE	First commit	2 months ago
README.md	Update README.md	2 minutes ago
Schema.sql	update	2 months ago

README.md

Python3-WebApp

基于Python3的Web开发

HTTP请求的生命周期



1. 客户端（浏览器）发起请求
2. 路由分发请求（这个框架自动帮处理），add_routes函数就是注册路由。
3. 中间件预处理
   - 打印日志
   - 验证用户登陆
   - 收集Request（请求）的数据
4. RequestHandler清理参数并调用控制器（Django和Flask把这些处理请求的控制器称为view functions）

5. 控制器做相关的逻辑判断，有必要时通过ORM框架处理Model的事务。
6. 模型层的主要事务是数据库的查增改删。
7. 控制器再次接管控制权，返回相应的数据。
8. Response_factory根据控制器传过来的数据产生不同的响应。
9. 客户端（浏览器）接收到来自服务器的响应。

WebApp实战

Day-1 搭建开发环境

2019-11-19 星期二

1. 确认当前系统安装的Python版本：

```
(base) C:\Windows\system32>python -m pip install --upgrade pip
(base) C:\Windows\system32>python --version
Python 3.6.5 :: Anaconda, Inc.
```

2. 安装MySQL数据库：从MySQL官方网站下载并安装。

3. 然后，用`pip`安装开发Web App需要的第三方库：

- **异步框架** aiohttp: `pip install aiohttp`
- **前端模板引擎** jinja2: `pip install jinja2`
- MySQL**的python异步驱动程序**aiomysql： `pip install aiomysql`

4. 项目结构 选择一个工作目录，建立大致如下的目录结构：

```
python3-webapp/  <-- 根目录
|
+- backup/       <-- 备份目录
|
+- conf/         <-- 配置文件
|
+- dist/         <-- 打包目录
|
+- www/          <-- Web目录，存放.py文件
| |
| +- static/     <-- 存放静态文件
| |
| +- templates/  <-- 存放模板文件
|
+- LICENSE       <-- 代码LICENSE
```

创建好工作目录结构后，建立Git**仓库**并同步至`GitHub`，版本控制保证代码修改安全:

```
PD000731_陈麒@IT-13 MINGW64 /d/CCPD-G8.6/Python3-WebApp
$ git init
Initialized empty Git repository in D:/CCPD-G8.6/Python3-WebApp/.git/
PD000731_陈麒@IT-13 MINGW64 /d/CCPD-G8.6/Python3-WebApp (master)
$ git remote add origin https://github.com/Qichen-Kylin/Python3-WebApp.git
PD000731_陈麒@IT-13 MINGW64 /d/CCPD-G8.6/Python3-WebApp (master)
$ git pull origin master
PD000731_陈麒@IT-13 MINGW64 /d/CCPD-G8.6/Python3-WebApp (master)
$ git push -u origin master
PD000731_陈麒@IT-13 MINGW64 /d/CCPD-G8.6/Python3-WebApp (master)
$ mkdir -p backup conf dist www/static www/templates
PD000731_陈麒@IT-13 MINGW64 /d/CCPD-G8.6/Python3-WebApp (master)
$ touch LICENSE
PD000731_陈麒@IT-13 MINGW64 /d/CCPD-G8.6/Python3-WebApp (master)
$ ls
backup/  conf/  dist/  LICENSE  README.md  www/
```

Day-2 编写Web App骨架

该WebApp建立在`asyncio`的基础上，用`aiohttp`写一个基本骨架`app.py`：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

__author__ = 'Kylin'

'''
async web application.
Python 3.5开始的新语法:
1. 把 @asyncio.coroutine 替换为 async;
2. 把 yield from 替换为 await 。
'''

import logging; logging.basicConfig(level=logging.INFO)

import asyncio, os, json, time
from datetime import datetime

from aiohttp import web

def index(request):
    return web.Response(body=b'<h1>Awesome</h1>',headers={'content-type':'text/html'})

async def init(loop):
    app = web.Application(loop=loop)
    app.router.add_route('GET', '/', index)
    # loop.create_server则利用asyncio创建TCP服务
    srv = await loop.create_server(app.make_handler(), '127.0.0.1', 9000)
    logging.info('server started at http://127.0.0.1:9000...')
    return srv

loop = asyncio.get_event_loop()
loop.run_until_complete(init(loop))
loop.run_forever()
```

WebApp将在`9000`端口监听http请求，并对首页`/`进行响应。

- **协程（Coroutine）**

1. 协程，又称微线程。协程看上去也是子程序，但是执行过程中，在子程序内部可以中断，然后转而执行别的子程序，在适当的时间再返回来接着执行中断的子程序。

2. 协程的特点在于是一个线程执行，和多线程相比协程最大的优势就是极高的执行效率。因为子程序切换不是线程切换，而是程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显；第二大优势就是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

3. Python对协程的支持是通过`generator`实现的。在generator中，不但可以通过循环来迭代，还可以不断的调用`next()`函数获取有`yield`语句返回的下一个值。Python的yield还可以接收调用者发出的参数。

- asyncio

1. `asyncio`是直接内置对异步IO的支持。`asyncio`的编程模型就是一个消息循环。我们从`asyncio`模块中直接获取一个`EventLoop`的引用，然后把需要执行的`协程`扔到`EventLoop`中执行，就实现了异步IO。

2. `asyncio`提供了完善的的异步IO支持；异步操作需要在`coroutine(协同程序)`中通过`yield from`完成；多个`coroutine`可以分装成一组Task然后并发执行。

3. `asyncio`可以实现单线程并发IO操作。如果将其只用在客户端，则发挥的威力不大；如果将它放在服务器端，例如Web服务器，由于http连接就是IO操作，因此可以用`单线程+协程`实现多用户的高并发支持。

Day-3 编写ORM

ORM，即`Object-Relational Mapping（对象关系映射）`，它的作用是在关系型数据库和业务实体对象之间作一个映射，这样，我们在具体的操作业务对象的时候，就不需要再去和复杂的SQL语句打交道，只需简单的操作对象的属性和方法。 **总结：简单来说ORM就是封装数据库的操作。**

> - 优点：
>
> 1. 只需要面向对象编程, 不需要面向数据库编写代码. ： 对数据库的操作都转化成对类属性和方法的操作，不用编写各种数据库的sql语句。
> 2. 实现了数据模型与数据库的解耦，屏蔽了不同数据库操作的差异： 不在关注用的是mysql、oracle....等，通过简单的配置就可以轻松更换数据库，而不需要修改代码。
>
> - 缺点：
>
> 1. 相比较直接使用SQL语句操作数据库,有性能损失。
> 2. 根据对象的操作转换成SQL语句,根据查询的结果转化成对象, 在映射过程中有性能损失。

设计背景与思路

- 在一个Web App中，所有的数据，包括用户信息，用户发布的日志，评论都放在数据库中。*本次实战使用MySQL作为数据库。*
- Web App中，有许多地方都要用到数据库，访问数据要创建数据库连接，创建游标对象，然后执行SQL语句，最后要处理异常，清理资源等。这些访问数据库的的代码如果分散到各个函数之中，势必无法维护，也不利于代码复用。
- 所以，首先要把常用的 `SELECT,INSERT,UPDATE,DELETE` 操作用函数分装起来。
- 由于Web框架使用了基于 `asyncio` 的 `aiohttp` ，这是基于协程的异步模型（*异步编程的原则：一旦决定使用异步，则系统每一层都必须是异步*）。Web App框架采用异步IO编程，而 `aiomysql` 为MySQL数据库提供了异步IO的驱动。

创建连接池

- 需要创建一个全局的连接池，每个HTTP请求都可以从连接池中直接获取数据库连接。
- 使用连接池的好处是不必频繁地打开和关闭数据库连接，而是能复用就尽量复用。
- 连接池由全局变量 `__pool` 存储，缺省情况下将编码设置为 `utf8` ，自动提交事务。

封装SELECT操作

- 封装 `SELECT` 操作为 `select()` 函数执行，需要传入SQL语句及SQL参数：
- SQL语句占位符是 `?` ，而MySQL占位符是 `%s` ，`select()` 函数内部自动替换。*注意要始终坚持使用带参数的SQL，而不是自己拼接SQL字符串，这样可以防止SQL注入攻击。*
- `cur.execute('select * from user where id = %s', ('1',))`
- 如果传入size参数，就通过 `fetchmany` 获得最多指定数量的记录，否则就通过 `fetchall` 获得所有记录。

封装INSERT,UPDATE,DELETE操作

- 要执行 `INSERT` 、 `UPDATE` 、 `DELETE` 语句，可以定义一个通用的 `execute()` 函数，因为这3类SQL的执行都需要相同的参数，以及返回一个整数表示影响的行数。
- `execute()` 函数与 `select()` 函数所不同的是，cursor对象不返回结果集，而是通过 `rowcount` 返回结果数。

创建Field类

- `Field类` ：负责保存数据库表的字段名、字段类型、是否主键、默认值...
- 在Field的基础上，进一步定义各种类型的Field： `StringField` 、 `IntegerField` 等。

创建元类ModelMetaclass

- 元类：即创建类的类，创建类的时候只需要指定 `metaclass=元类` ，元类需要继承 `type` 而不是 `object` ,因为type就是元类。
- 元类 `ModelMetaclass` 负责分类、整理收集的数据并以此创建一些类属性(如SQL语句)供基类作为参数。

创建基类Model

- 基类 `Model` 负责执行操作，比如数据库的存储、读取，查找等操作等。
- 基类中方法都基于 `asyncio` 的装饰，所以方法都是协程。
- 任何继承自 `Model` 的类，会自动通过 `ModelMetaclass` 扫描映射关系，并存储到自身的类属性中。

Day-4 编写models

有了`ORM`，就可以把`WebApp`需要的表用`Model`表示出来：`models.py`

- 在编写`ORM`时，给每个`Field`增加一个`default`参数可以让ORM自己填入缺省值，非常方便。并且缺省值可以作为函数对象传入，在调用`save()`时自动计算。
- 比如：定义主键`Id`的缺省值是函数`next_id`,创建时间`created_at`的缺省值是函数`time.time`，可以自动设置当前时间。
- 日期和时间用`float`类型存储在数据库中，而不是`datatime`类型，这么做的好处是不必关心数据库的时区以及时区转换问题，排序非常简便，显示的时候，只需要做一个`float`到`str`的转换。

初始化数据库表

1. 连接数据库初始化：脚本见`Schema.sql`;
2. 用python的mysql-connector初始化：见`InitSchema.py`.

Day-5 编写Web框架

> 因为复杂的Web应用程序，光靠一个WSGI(Web Server Gateway Interface)函数来处理还是太底层了，我们需要在WSGI之上再抽象出Web框架(比如Aiohttp、Django、Flask等)，从而进一步简化Web开发。

在Day-1编写web app骨架因为要实现协程，所以运用的是aiohttpweb框架。那么现在为何又要重新编写一个新的web框架呢，这是因为从使用者的角度来说，aiohttp相对比较底层，想要使用框架时编写更少的代码，就只能在aiohttp框架上封装一个更高级的框架。

> Web框架的设计是完全从使用者出发，目的是让框架使用者编写尽可能少的代码。

因此我们希望框架使用者可以摒弃复杂的步骤，这次新创建的框架想要达到的预期效果是：只需编写函数(不然就要创建async def handle_url_xxx(request): ...这样的一大推东西)，透过新建的Web框架就可以实现相同的效果。同时，这样编写简单的函数而非引入request和web.Response还有一个额外的好处，就是可以单独测试，否则，需要模拟一个request才能测试。

因为是以aiohttp框架为基础，要达到上述预期的效果，也是需要符合aiohttp框架要求，因此就需要考虑如何在request对象中，提取使用者编写的函数中需要用到的参数信息，以及如何将函数的返回值转化web.response对象并返回。

1. 编写URL处理函数

1.1 aiohttp编写URL处理处理函数

Day-1的URL处理函数比较简单，因为Day-1的的URL处理函数没有真正意义上使用到request参数，但总体上差不多。 使用aiohttp框架，编写一个URL处理函数大概需要几步： 第一步，添加协程装饰器

```
async def handle_url_xxx(request):
       ...
```

第二步，对request参数进行操作，以获取相应的参数

```
url_param = request.match_info['key']
query_params = parse_qs(request.query_string)
```

第三步，就是构造Response对象并返回

```
text = render('template', data)
return web.Response(text.encode('utf-8'))
```

而新创建的web框架希望可以封装以上一些步骤，在使用时，更加方便快捷。

1.2 新建web框架编写URL处理函数

1.2.1 @get 和 @post

> Http定义了与服务器交互的不同方法，最基本的方法有4种，分别是GET，POST，PUT，DELETE。URL全称是资源描述符，我们可以这样认为：一个URL地址，它用于描述一个网络上的资源，而HTTP中的GET，POST，PUT，DELETE就对应着对这个资源的查，改，增，删4个操作。 建议：
>
> - 1、get方式的安全性较Post方式要差些，包含机密信息的话，建议用Post数据提交方式；
> - 2、在做数据查询时，建议用Get方式；而在做数据添加、修改或删除时，建议用Post方式；

把一个函数映射为一个URL处理函数，可以先构造一个装饰器，用来存储、附带URL信息

1.2.2 定义RequestHandler

> 参考：关于inspect

使用者编写的URL处理函数不一定是一个coroutine，因此用 `RequestHandler()` 来封装一个URL处理函数。 `RequestHandler` 是一个类，创建的时候定义了 `__call__()` 方法，因此可以将其实例视为函数。 `RequestHandler` 目的就是从URL函数中分析其需要接收的参数，从request中获取必要的参数，调用URL函数。(要完全符合aiohttp框架的要求，就需要把结果转换为web.Response对象)

1.2.3 封装APIError

从 `RequestHandler` 代码可以看出最后调用URL函数时，URL函数可能会返回一个名叫 `APIError` 的错误，那这个APIError又是什么来的呢，其实它的作用是用来返回诸如账号登录信息的错误。

2.编写add_route函数以及add_static函数

> 参考： 关于_import_ 关于rfind 关于add_static 关于Jinja2

由于新建的web框架时基于aiohttp框架，所以需要再编写一个 `add_route()` 函数，用来注册一个URL处理函数，主要起验证函数是否有包含URL的响应方法与路径信息，以及将函数变为协程。

通常a `dd_route()` 注册会调用很多次，而为了框架使用者更加方便，可以编写了一个可以批量注册的函数 `add_routes()` ，预期效果是：只需向这个函数提供要批量注册函数的文件路径，新编写的函数就会筛选，注册文件内所有符合注册条件的函数。

`add_static(add)`: 添加静态文件夹的路径。

`nit_jinja2(app, **kw)` : 添加完静态文件还需要初始化jinja2模板。

3. 编写middleware

> 参考： 关于middleware 关于response

如何将函数返回值转化为 `web.response` 对象呢？

这里引入aiohttp框架的web.Application()中的middleware参数。

middleware是一种拦截器，一个URL在被某个函数处理前，可以经过一系列的middleware的处理。一个middleware可以改变URL的输入、输出，甚至可以决定不继续处理而直接返回。middleware的用处就在于把通用的功能从每个URL处理函数中拿出来，集中放到一个地方。

middleware的感觉有点像装饰器，这与上面编写的RequestHandler有点类似。

从官方文档可以知道，当创建web.appliction的时候，可以设置middleware参数，而middleware的设置是通过创建一些middleware factory(协程函数)。这些middleware factory接受一个app实例，一个handler两个参数，并返回一个新的handler。

- 一个记录URL日志的logger可以作为middle factory；
- 转化得到response对象的middleware factory等。

参考原文链接：Day5-编写web框架

Day-6 编写配置文件

- 通常，一个 `Web App` 运行的时候都要读取配置文件，比如数据库的名字，口令等，在不同的环境中运行，可以读取不同的配置文件来获得正确的配置。
- 由于Python本身语法简单，完全可以用源代码来实现配置，而不需要解析一个单独的 `.properties` 或者 `.yaml` 等配置文件。
- 默认的配置文件应该完全符合本地开发环境，这样，无需任何设置，就可以立刻启动服务器。

实现思路

1. 编写默认配置文件 `config_default.py` ；
2. 编写覆盖配置文件 `config_override.py` ；
3. Merge配置文件为 `config.py` 。

把 `config_default.py` 作为开发环境的标准配置，把 `config_override.py` 作为生产环境的标准配置；就可以既方便的在本地开发，又可以随时把应用部署到服务器上。

应用程序读取配置文件需要优先从 `config_override.py` 读取，为了简化读取配置文件，可以把所有配置读取到统一的 `config.py` 中。

Day-7 编写MVC

> `MVC: Model-View-Controller`,中文名"模型-视图-控制器"。

- 其中Python处理的URL函数就是C： `Controller` ， `Controller` 主要负责业务逻辑，比如检查用户名是否存在，取出用户信息等等；
- 而 `View` 负责显示逻辑，通过一些简单的替换变量，View生成最终用户看到的HTML，那 `View` 实质就是HTML模板（如 `Django` 等），而在本次Web开发就是 `Jinja2` 模板；
- `Model` 是用来传给View的，这样View在替换变量的时候，就可以从 `Model` 中取出相应的数据。

当ORM框架、Web框架、配置就绪，就可以编写MVC，将其全部启动起来。对应的页面处理编写相对应的URL函数。使用 `'__template__'` 指定模板(`html`)。

```
import re, time, json, logging, hashlib, base64, asyncio

from coroweb import get, post

from models import User, Comment, Blog, next_id

@get('/')
async def index(request):
    users = await User.findAll()
    return {
        '__template__': 'test.html',
        'users': users
    }
```

Day-8 构建前端

- 简单的 `MVC` 页面不会令人满意。对于复杂的HTML前端页面来说，我们需要一套基础的 `CSS` 架构来完成页面布局和基本样式。另外 `jQuery` 作为操作DOM的 `JavaScript` 库也必不可少。
- 从零开始写 `CSS` 不如直接从一个已有的功能完善的 `CSS` 框架开始，有很多这样的框架可以选择，这里这次选择 `uikit` 这个强大的 `CSS` 框架。它具有完善的响应式布局、漂亮的UI，以及丰富的HTML组件。

> Uikit：A lightweight and modular front-end framework for developing fast and powerful web interfaces.（轻量级和模块化的前端框架，用于开发快速和强大的web接口。）

- 所有的静态资源文件统一放在 `www/app/static` 目录下，并按照类别归类：

```
static/
+- css/
| +- addons/
| | +- uikit.addons.min.css
| | +- uikit.almost-flat.addons.min.css
| | +- uikit.gradient.addons.min.css
| +- awesome.css
| +- uikit.almost-flat.addons.min.css
| +- uikit.gradient.addons.min.css
| +- uikit.min.css
+- fonts/
| +- fontawesome-webfont.eot
| +- fontawesome-webfont.ttf
| +- fontawesome-webfont.woff
| +- FontAwesome.otf
+- js/
+- awesome.js
+- html5.js
+- jquery.min.js
+- uikit.min.js
```

- 由于前端页面通常情况下不是首页一个页面，每个页面都有相同的页眉和页脚。常见的模板引擎已经考虑到了页面上重复`HTML`部分的复用问题。有的模板通过`include`把页面分为三个部分：

```
<html>
    <% include file="inc_header.html" %>
    <% include file="index_body.html" %>
    <% include file="inc_footer.html" %>
</html>
```

这样相同的部分`inc_header.html`和`inc_footer.html`就可以共享。

- 但是`include`方法不利于页面整体结构和维护。`jinjia2`的模板还有一种"继承"方式，实现模板的复用更简单。这种模板方式是编写一个"父模板"，在父模板中定义一些可替换的`Block`（块）。然后编写多个"子模板"，每个子模板都可以只替换父模板定义的`block`。比如父模板定义：

```
<!-- __base.html__ -->
<html>
    <head>
        {% block meta %}<!-- block meta  -->{% endblock %}
        <title>{% block title %} ? {% endblock %} - Awesome Python Webapp</title>
        {% block beforehead %}<!-- before head  -->{% endblock %}
    </head>
    <body>
        {% block content %} <!-- content -->{% endblock %}
    </body>
</html>
```

- 对于子模板，只需要将父模板中定义的对应`block`替换掉就行：

```
<!-- A.html -->
{% extends '__base__.html' %} <!-- 继承父模板标识  -->
{% block title %} A {% endblock %} <!-- 替换title block，覆盖页面的标题  -->
{% block beforehead %} <!-- 子页面在<head>标签关闭前插入JavaScript代码  -->
<script>
</script>
{% endblock %}
{% block content %} <!-- 子页面content布局和内容  -->
    ...
{% endblock %}
```

> 在浏览器看到的画面，都是有浏览器解释才呈现出来的。实质它是一段HTML代码，外加JavaScript、CSS构成。如果把网页比作一个人，那么HTML便是他的骨架；JavaScript是肌肉；CSS是衣服。

Day-9 编写API

- 如果一个URL返回的不是HTML，而是机器可以直接解析的数据，这个URL可以看成是一个WebAPI。比如读取`http://localhost:9000/api/blogs/123`，如果能直接返回Blog数据，那么机器就可以直接读取。
- `REST`(*英文全称是 Representational State Transfer ，翻译成中文是"表述性状态转移"*)就是一种设计API的模式。最常见的格式是`JSON`。由于JSON能直接被JavaScript读取，所以，以JSON格式编写的REST风格的API具有简单、易读、易用的特点。
- 编写API的好处：由于API就是把WebApp的功能全部封装了，所以，通过API操作数据，可以极大地把前端和后端代码隔离，使得后端代码易于测试，前端代码编写更加简单。
- 一个API就是一个URL处理函数，可以直接通过一个`@api`来吧函数变成JSON格式的REST API。

```
@get('/api/users')
def api_get_users():
    users = yield from User.findAll(orderBy='created_at desc')
    for u in users:
        u.passwd = '******'
    return dict(users=users)
```

- 只要返回一个`dict`，后续的`response`这个`middleware`就可以把结果序列化为JSON并返回。

Day-10 用户注册和登录

> 用户管理是绝大部分Web网站都需要解决的问题，涉及用户注册和登录。

- **用户注册**

1. 用户注册功能通过API先实现用户注册功能：代码中部分为`@post('/api/users')`。 *需要注意的是用户口令是客户端传递的经过`SHA1`计算后的40位Hash字符串，所以服务器端并不知道用户的原始口令。*
2. 接下来创建一个注册界面给用户填写注册表单，然后提交数据到注册用户的API。代码中部分为`templates/register.html`。

> 这里的流程是首先浏览器输入http://localhost:9000/，进入主页面，点击右上角注册，__base.html中通过链接跳转到http://localhost:9000/register网页，触发handlers.py文件中的@get('/register')请求，加载register.html网页，填写好信息后，点击提交按钮，会触发register.html的block beforehead部分的JavaScript代码。JavaScript代码的主要几个步骤为：1：校验输入值是否正确。 2：针对密码生成SHA1值。 3：执行@get('/api/users')函数，在该部分提交信息，并向浏览器返回cookie。 4：完成后返回主页。

- **用户登录**

1. 用户登录要比用户注册复杂。由于`HTTP协议`是一种无状态协议，而服务器要跟踪用户状态，就只能通过`cookie`实现。大多数Web框架提供了`Session`功能来封装保存用户状态的`cookie`。
2. Session的优点是简单易用，可以直接从Session中取出用户登录信息。
3. Session的缺点是服务器需要在内从中维护一个映射表来存储用户的登录信息，如果服务器是两台及以上，就需要对Session做集群，因此，使用Session的Web App很难扩展。
4. 这里采用直接读取cookie的方式来验证用户登录，每次用户访问任意URL，都会对cookie进行验证，这种方式的好处就是保证服务器处理任意的URL都是无状态的，可以拓展到多台服务器。
5. 由于登录成功后是由服务器生成一个cookie发送给浏览器，所以，要保证这个cookie不会被客户端伪造出来。实现防伪造的cookie的关键使用过一个单向算法（例如：SHA1），举例如下：

> 当用户输入了正确的口令登录成功后，服务器可以从数据库取到用户的id，并按照如下的方式计算出一个字符串：`"用户id"+"过期时间"+SHA1("用户id"+"用户口令"+"过期时间"+"SecretKey")` 当浏览器发送cookie到服务器端后，服务器可以拿到的信息包括：用户id、过期时间、SHA1值。如果未到过期时间，服务器就根据用户id查找用户口令，并计算：`SHA1("用户id"+"用户口令"+"过期时间"+"SecretKey")`，并与浏览器cookie中的哈希进行比较，如果相等，则说明用户已经登录，否则，cookie就是伪造的。 这个算法的关键就是在于SHA1是一种单向算法，即是可以通过原始字符串可以计算出SHA1结果，但是无法通过SHA1结果反推出原始字符串。

1. 登录的API实现在是代码中为`@post('/api/authenticate')`，以及计算加密cookie函数为`def user2cookie(user, max_age)`。
2. 对于每个URL处理函数，都去解析cookie的代码，那会导致代码重复好多次。
3. 如果利用middle在处理URL之前，把cookie解析出来，cookie解析协程函数为`async def cookie2user(cookie_str)`，并将登录用户绑定在request对象上，这样，后续的URL处理函数就可以直接拿到登录用户，对应程序中的协程函数为`async def auth_factory(app, handler)`。这样便完成了用户管理中的用户注册和登录功能。

> 密码生成 从上一节中，可以看出密码生成的步骤如下： 1 ：在register.html文件中的JavaScript将passward进行第一次包装，生成A，传递到@get('/api/users')中。 `CryptoJS.SHA1(email + ':' + this.password1).toString()` 2：@post('/api/users')绑定函数接下来对A进行第二次包装，生成B。 `sha1_passwd = '%s:%s' % (uid, passwd)` 3：user2cookie函数对B进行第三次包装，生成C。

```
#id-B-到期时间-秘钥
expires = str(int(time.time() + max_age))
s = '%s-%s-%s-%s' % (user.id, user.passwd, expires, _COOKIE_KEY)
```

4： 返回用户id-到期时间-C

```
#用户id-到期时间-C
L = [user.id, expires, hashlib.sha1(s.encode('utf-8')).hexdigest()]
```

> 密码比较 在用户cookie未到期时，对用户认证的时候，通过signin.html里的SHA1(email+password)值对password进行包装生成A。 `passwd: this.passwd==='' ? '' : CryptoJS.SHA1(email + ':' + this.passwd).toString()` 接下来运行handlers中@post('/api/authenticate')的绑定函数，对密码进行再次包装生成B。

```
#check passwd:
sha1 = hashlib.sha1()
sha1.update(user.id.encode('utf-8'))
sha1.update(b':')
sha1.update(passwd.encode('utf-8'))
```

然后对比两个密码，判断是否登陆。`if user.passwd != sha1.hexdigest()` 如果认证通过，更新`cookie`。最后通过`signin.html`中的`location.assign('/')`;来跳转到主页面，并传递用户信息到`blogs.html`中。完成右上角的信息请求。

原文链接

Day-11 编写日志创建页

- 在编写完ORM框架，web开发框架之后，后端代码写起来就相对的比较轻松了，现在编写一个REST API,用来创建blog：`@post('/api/blogs')`。
- Web开发真正困难的地方就是编写前端页面。前端页面需要混合HTML、CSS和JavaScript，如果对这三者没有深入地掌握，编写的前端页面将很快难以维护。
- 更大的问题在于前端页面通常是动态页面，也就是说，前端页面往往是由后端代码生成的。ASP、JSP、PHP等都是用以下这种模板方式生成前端页面。
- 如果在页面上要使用大量的javaScript，这样的模板会导致JavaScript代码与后端代码绑得非常紧密，以至于难以维护。**根本原因还是：负责显示的DOM模型(Document Object Model)与负责数据和交互的JavaScript代码没有分割清楚。**
- 所以新的MVVM：Model View ViewModel模式应运而生。个人认为这个MVVM就是基于前端页面的MVC。
- 在前端页面中，用纯JavaScript对象表示Model，而View则是纯HTML。
- Model表示数据，View负责显示，两者做到了最大限度的分离。
- ViewModel作用是把Model和View关联起来的。ViewModel负责把Model的数据同步到View显示出来，还负责把View的修改同步回Model。
- ViewModel如何编写？需要用JavaScript编写一个通用的ViewModel，这样，就可以复用整个MVVM模型了。
- 已经有很多成熟的MVVM框架可以使用，如AngularJS，KnockoutJS等，我们这里选择Vue这个简单易用的MVVM框架来实现创建Blog的页面`templates/manage_blog_edit.html`。
- 初始化Vue时，我们指定3个参数：

1. el:根据选择器查找绑定的View，这里是#VM，就是id为VM的DOM，对应的是一个 标签；
2. data：JavaScript对象表示的Model，我们初始化为`{name: '', summary: '', content: ''}`；
3. Models：View可以触发的JavaScript函数，submit就是提交表单时创建的函数。

- 接下来，我们在`<Form>`标签中，用几个简单的`v-model`,就可以让Vue把Model和View关联起来。
- Form表单通过`<form v-on="submit: submit">`把提交表单的事件关联到`submit`方法。
- 需要特别注意的是，在MVVM中，Model和View是双向绑定的。如果我们在Form中修改了文本框中的值，可以在Model中立刻拿到新的值。
- 双向绑定是MVVM框架最大的作用。借助MVVM，我们把最复杂的显示逻辑交给框架完成。由于后端编写了独立的REST API，所以，前端用AJAX提交表单非常容易，前后端分离得非常彻底。

Day-12 编写日志列表页

- `MVVM`模式不但可用于Form表单，在复杂的管理页面中也能大显身手。例如，分页显示Blog的功能，我们先把后端代码写出来： 在`apis.py`中定义一个Page类用于存储分页信息：`class Page(object)`。
- 在`handlers.py`中编写API实现接口用于数据库返回日志。`@get('/api/blogs')`。
- 编写管理页面 REST API：`@get('/manage/blogs')`。
- 模板页面首先通过API：`GET /api/blogs?page=?`拿到Model：

```
{
    "page": {
        "has_next": true,
        "page_index": 1,
        "page_count": 2,
        "has_previous": false,
        "item_count": 12
    },
    "blogs": [...]
}
```

- 然后，通过Vue初始化MVVM: `templates/manage_blogs.html`。View的容器是`#vm`，包含一个`table`，我们用`v-repeat`可以把`Model`的数组`blogs`直接变成多行的`<tr>`;可以把`v-repeat="blog: blogs"`看成循环代码，所以，可以在一个内部引用循环变量blog。`v-text`和`v-attr`指令分别用于生成文本和DOM节点属性。

Day-13 提升开发效率

- 每次修改代码，都必须在命令行先`Ctrl-C`停止服务器，再重启，改动才能生效。
- 有没有办法让服务器检测到代码修改后自动重新加载呢？思路是检测www目录下的代码改动，一旦有改动，就自动重启服务器。
- 按照思路，我们可以编写一个辅助程序`pymonitor.py`，让它启动`app.py`，并时刻监控www目录下的代码改动，有改动时，先把当前`app.py`进程杀掉，再重启，就完成了服务器进程的自动重启。
- 这里使用得是Python第三方库`watchdog`：`pip install watchdog`
- 利用`watchdog`接收文件变化的通知，如果是`.py`文件，就自动重启`app.py`进程。
- 利用`Python`自带的`subprocess`实现进程的启动和停止，并把输入输出重定向到当前进程的输入和输出。
- 启动服务器：

1. `$ python3 pymonitor.py app.py`;
2. `chomd u+x pymonitor.py`

   `$ pymonitor.py app.py`

Day-14 完成WebApp

- 在`WebApp`框架和基本流程跑通后，剩下的工作全部是体力活了：在`Debug`开发模式下完成后端所有API、前端所有页面。我们需要做的事情包括：
- 把当前用户绑定到`request`上，并对`URL/manage/`进行拦截，检查当前用户是否是管理员身份，功能实现为工厂函数：`async def data_factory(app, handler)`。
- 用户注册时，`users`表中`admin`字段置为0的，管理员用户需要将其修改为1.
- **后端API包括：**

- [x] •获取日志：GET /api/blogs
- [x] •创建日志：POST /api/blogs
- [x] •修改日志：POST /api/blogs/:blog_id
- [x] •删除日志：POST /api/blogs/:blog_id/delete
- [x] •获取评论：GET /api/comments
- [x] •创建评论：POST /api/blogs/:blog_id/comments
- [x] •删除评论：POST /api/comments/:comment_id/delete
- [x] •创建新用户：POST /api/users
- [x] •获取用户：GET /api/users

- **管理页面包括：**

- [x] •评论列表页：GET /manage/comments
- [x] •日志列表页：GET /manage/blogs
- [x] •创建日志页：GET /manage/blogs/create
- [x] •修改日志页：GET /manage/blogs/edit
- [x] •用户列表页：GET /manage/users

- **用户浏览页面包括：**

- [x] •注册页：GET /register
- [x] •登录页：GET /signin
- [x] •注销页：GET /signout
- [x] •首页：GET /
- [x] •日志详情页：GET /blog/:blog_id

把所有的功能实现，我们第一个Web App就宣告完成！

Day-15 部署WebApp

- Linux系统安装准备完毕后，保证ssh服务正在运行：

```
--Linux,Unbuntu有些许差异
[root@bdpadmin ~]# service sshd status
openssh-daemon (pid  4348) is running...
[root@bdpadmin ~]# ps -ef | grep ssh
root      4348     1  0  2018 ?        00:00:19 /usr/sbin/sshd
root     15386  4348  0 09:49 ?        00:00:00 sshd: root@pts/0
root     15801 15391  0 09:52 pts/0    00:00:00 grep ssh
```

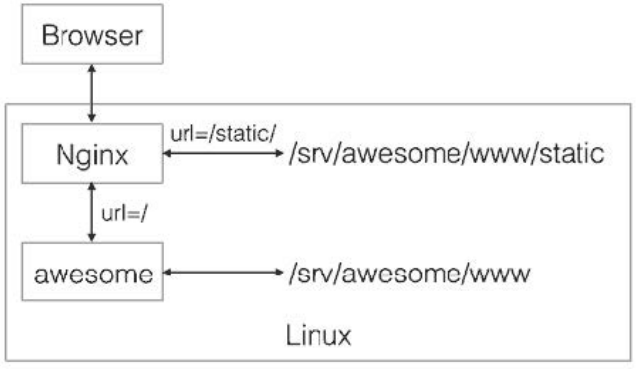
否则(安装)启动：

```
[root@bdpadmin ~]# sudo apt-get install openssh-server
[root@bdpadmin ~]# service sshd start
```

有了ssh服务，就可以从本地连接到服务器上。建议把公钥复制到服务器端用户的 `.ssh/authorized_keys` 中实现无密码认证链接。

部署方式

- 利用 `Python` 自带的 `asynico` ,以及完成了一个异步高性能服务器。但是还需要一个高性能的 `Web` 服务器，这里选择 `Nginx` ，它可以处理静态资源，同时作为反向代理把动态请求交给Python代码处理。Nginx负责分发请求模型如下：



。

- 在服务器端，我们需要定义好部署的目录结构：

```
/
+- srv/
```

```
    +- awesome/ <-- Web App根目录
        +- www/app/ <-- 存放Python源码
            | +- static/ <--存放静态资源文件
        +- log/ <-- 存放服务log
```

在服务器上部署，要考虑到新版本如果运行不正常，需要回退到旧版本时怎么办。每次用新的文件覆盖旧的文件是不行的，需要一个类似版本控制的机制。由于`Linux`系统提供了软链接功能，所以，把`www`目录作为一个软链接，它指向哪个版本目录，哪个就是当前运行的版本。而`Nginx`和`Python`代码的配置文件只需要指向`www`目录即可。

> 软链接相关参照：(软链接类似Windows下快捷方式) 1、软链接建立： `ln -s /原路径1 /软链接路径` 2、软链接修改： `ln -snf /原路径2 /软链接路径` 3、软链接删除： `rm [-rf] /软链接路径`

- Nginx可以作为服务进程直接启动，但是app.py还不行，所以，Supervisor登场。`Supervisor`是一个管理进程的工具，可以随系统的启动而启动服务，它还时刻监控服务进程，如果服务进程意外退出，`Supervisor`可以自动重启服务。
- 需要用到的服务有：

1. Nginx ： 高性能Web服务器+负责反向代理；
2. Supervisor ： 监控服务进程的工具；
3. MySQL ： 数据库服务；
4. Python ：基本运行环境。 在Linux服务器上用apt直接安装上述服务： `$ sudo apt-get install nginx supervisor python3 mysql-server`。

- 有了Python基础环境，然后将WebApp所用到的的Python库安装了： `$ sudo pip3 install jinja2 aiomysql aiohttp`。
- 在服务器上初始化和配置好MySQL数据库后，执行数据库初始化脚本： `$ mysql -u root -p < schema.sql`。
- ..... 服务器准备就绪。

部署

- 用FTP还是SCP还是rsync复制文件。（手动复制部署）
- 一般正确的部署方式是使用工具配合脚本完成自动化部署。Fabric是一个自动化部署工具。开发本地安装`Fabric`后，编写部署脚本`fabfile.py`，文件与www目录平级。

> Fabric is a high level Python (2.7, 3.4+) library designed to execute shell commands remotely over SSH, yielding useful Python objects in return。

- Fabric脚本编写：

  i. 首先导入Fabric的API，设置部署时的变量；

  ii. 每个Python函数都是一个任务，先写一个打包的任务：`def build()`；

  - Fabric提供`local('...')`来运行本地命令，with lcd(path)可以把当前命令的目录设定为`lcd()`指定的目录，注意Fabric只能运行命令行命令，Windows下可能需要Cgywin环境。

  iii. 在`Python3-WebApp`目录下运行：`$ fab build`。顺利的话会在dist目录下创建`dist-awesome.tar.gz`的文件；

  iv. 打包后，就可以继续编写`deploy`任务，把打包文件上传至服务器，解压、重置www软链接、重启相关服务；

  - Fabric提供`run()`函数执行的命令是在服务器上运行，`with cd(path)`和`with lcd(path)`类似，把当前目录在服务器端设置为`cd()`指定的目录。如果一个命令需要sudo权限，就不能用`run()`,而是用`sudo()`来执行。

配置Supervisor

- 编写一个Supervisor的配置文件`awesome.conf`,存放到`/etc/supervisor/conf.d/`目录下：

```
[program:awesome]
command = /srv/awesome/www/app/app.py
directory = /srv/awesome/www/app
user = webapp
startsecs = 3
redirect_stderr = true
stdout_logfile_maxbytes = 50MB
stdout_logfile_backups = 10
stdout_logfile = /srv/awesome/log/app.log
```

- 配置文件通过`[program:awesome]`指定服务名为`awesome`，`command`指定启动`app.py`。
- 然后重启Supervisor后，就可以随时启动和停止Supervisor管理的服务了。

```
$ sudo supervisorctl reload
$ sudo supervisorctl start awesome
$ sudo supervisorctl status
```

配置Nginx

- Supervisor只负责运行`app.py`,所以还需要配置Nginx，把配置文件`awesome`放到`/etc/nginx/sites-available/`目录下：

```
server {
        listen 80; # 监听80端口
        root /srv/awesome/www/app;
        access_log /srv/awesome/log/access_log;
        error_log /srv/awesome/log/error_log;
        #server_name awesome.kylin.com; # 配置域名
        #处理静态文件/favicon.ico:
        location /favicon.ico {
            root /srv/awesome/www/app;
        }
        #处理静态资源:
        location ~ ^\/static\/.*$ {
            root /srv/awesome/www/app;
        }
        # 动态请求转发到9000端口:
            location / {
            proxy_pass http://127.0.0.1:9000;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header Host $host;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }
    }
```

- 然后在`etc/nginx/sites-available/`该目录下创建软链接：

```
$ pwd
/etc/nginx/sites-enabled
$ sudo ln -s /etc/nginx/sites-available/awesome .
```

- 让Nginx重新加载配置文件：`$ sudo /etc/init.d/nginx reload`
- 如果有任何错误，都可以在`/srv/awesome/log`下查找Nginx和App本身的log。如果Supervisor启动时报错，可以在`/var/log/supervisor`下查看Supervisor的log。

如果一切顺利，就可以在浏览器中访问WebApp了！

- 如果开发环境更新了代码，只要在命令行执行如下两条指令，就完成自动部署服务：

```
$ fab build
$ fab deploy
```