基于 flask 架构的博客系统软件说明

作者：张靖祥

目　录

第一章 项目简介

1.1 文章的目的

文章用于详细解释该 flask 项目的结构、注意事项、未来的拓展方向等。用于向其他使用了该架构的人提供设计思路。阅读本文之前请务必确保已经获得了源程序。

1.2 业务需求

本项目使用 python 语言，采用 flask 架构设计网站。本项目仅用于学习使用，预计并发量小于 10 QPS（小于每秒 10 次查询）。因此项目完整的程序只部署在一台服务器中。

1.3 参考资料

本项目参考了常见的网站设计架构，包括了 MVC 架构、SSM 架构等常用服务器架构，同时参考了 Miguel Grinberg 的《Flask Web Development》中对于诸多 flask 第三方库的使用方法。

第二章 项目启动

本章内容用于配置、启动网站，详细的记录在项目根目录下的 README.MD 文件中，如果已经阅读 README.MD 文件，则可以直接跳过本章。请确保电脑中已经安装了 python3 及以上的版本，并添加了 pip 到环境变量。在命令行中输入 pip

如果没有添加环境变量，或者有多个 python，需要找到 python 根目录，以 Windows 系统 C:\Program Files\python\为例，将下面全部的 pip 全部替换为 C:\Program Files\python\Scripts\pip，例如 pip install -r requirements.txt，则替换为 C:\Program Files\python\Scripts\pip install -r requirements.txt

1. 下载配置文件。

用于安装项目所需的全部第三方库，若安装速度太慢，可以自行搜索更换 pip 源的方法。（linux 系统使用 pip3 命令）

pip install -r requirements.txt

2. 配置第三方库中对于跨域请求的路径地址。

第三方库，例如 flask_bootstrap 中，记录了 bootstrap 的地址，该地址位于境外，访问速度通常在 10s 以上，需要将其替换为国内的源。

Windows 系统下，使用 pycharm 开发工具，在 import flask_bootstrap 中点击 flask_bootstrap 直接进入该库的源文件，可以直接进行修改。如果没有类似的开发工具，则需要手动进入 python 安装文件夹下的 \Lib\site-packages\ 文件夹，进入对应的文件夹，并打开 __init__.py 文件进行修改。

Linux 系统下，第三方库被安装在各自用户单独的目录下。在 Ubuntu 系统中，位于 /home/user_name/.local/lib/python3.x/site-packages 目录下，需要使用 vim 打开后用命令行修改，也可以直接使用远程连接工具（Bitwise SSH 等）中的 SFTP 打开，直接双击打开.py 文件，修改后保存即可（该方法还需要设置 Windows 默认的.py 文件打开格式为记事本）。

需要修改的第三方库如下：

（1）找到 bootstrap 文件下的 init 文本（在文件结尾），修改下面的地址为：

```
    bootstrap = lwrap(
        WebCDN('//cdn.bootcss.com/bootstrap/%s/' % BOOTSTRAP_VERSION),
local)
    jquery = lwrap(
        WebCDN('//cdn.bootcss.com/jquery/%s/' % JQUERY_VERSION), local)
    html5shiv = lwrap(
        WebCDN('//cdn.bootcss.com/html5shiv/%s/' % HTML5SHIV_VERSION))
    respondjs = lwrap(
        WebCDN('//cdn.bootcss.com/respond.js/%s/' % RESPONDJS_VERSION))
```

（2）找到 flask_moment 库，用以下地址替换其中 cdn 地址。（去掉版本号变量和%s，因为不同服务商的文件存储路径和版本号格式可能不一样）
https://cdn.bootcss.com/moment.js/2.18.1/locale/af.js

但本文没有成功找到可以修改的地方，因此推荐使用下面的方法。在所有前端页面引用 moment 的时候加入 local_js 参数，并赋值为国内源。

{{ moment.include_moment(local_js="https://cdn.bootcss.com/moment.js/2.22.1/moment.min.js") }}

（3）找到 flask_pagedown 库文件，用以下地址替换其中 cdn 地址（就在文件开头）:（去掉版本号变量和%s，因为不同服务商的文件存储路径和版本号格式可能不一样）
//cdn.bootcdn.net/ajax/libs/pagedown/1.0/Markdown.Converter.min.js
//cdn.bootcdn.net/ajax/libs/pagedown/1.0/Markdown.Sanitizer.min.js

3. 配置参数文件
在目录 /scripts/start_config.py 中，修改数据库 url，即 sql_dev 行，格式为：
mysql+pymysql://数据库用户名:密码@服务器地址:3306/数据库名?charset=utf8

同时修改与 email 有关的项，本项目采用 qq 邮箱发送邮件（163 邮箱审核严格，很难使用 SMTP 协议发送邮件）。必须修改的有：
MAIL_USERNAME、MAIL_DEFAULT_SENDER 你的 qq 邮箱
MAIL_PASSWORD 你的 qq 邮箱申请的 SMTP 协议的密码

修改第 8 行的 t＝Write(key="****")，key 为配置文件的 AES 加密密码，如果 key 为 None 或不赋值，则不进行加密。
修改 img 中 AVATAR_PATH，用于保存图像数据，该文件夹必须存在。

直接运行该文件，会在项目根目录生成 config.data 的文件，即服务器启动配置。

4. 保存服务器密码
服务器密码需要保存在 /key.txt 中，该密码为上述的 key。

4. 生成虚假数据
若不需要生成虚假数据，则跳过该步骤，在命令行输入以下启动命令。
python manager.py db generate_fake

5. 配置服务器启动地址与端口
打开根目录下的 manager.py 文件，修改 run 函数中的 app.run 的参数，
若本地运行，则将 hosts 改为 `127.0.0.1`，
若将其挂到服务器，则将 hosts 改为 `0.0.0.0`。

6. 启动服务器
直接启动命令：python3 manager.py run
linux 系统后台运行命令： nohup python3 -u manager.py run > out.log 2>&1 &
或者直接运行：sh start.sh

第三章 项目架构详细说明及可扩展内容

3.1 总体结构

/app/ 程序主体架构

controller/	存储全部的 route 路径以及对应处理的函数
models/	存储所有数据模型的结构
service/	存储所有的数据库访问函数
webapp/	存储与前端页面有关的文件
__init__.py	创建 flask 的对象 app，初始化 flask 第三方库，引入蓝图
decorators.py	创建修饰器，用于函数执行前的检测（用户权限检测）
email.py	用于发送邮件
exception.py	用于创建自定义的错误类型
/migrations/	数据库迁移过程中由 flask_migrate 库自动创建的文件夹
/scripts/	与项目启动无关的其他功能
/config.data	由/scripts/start_config.py 生成的启动配置文件
/key.txt	由 start.sh 生成的对于 config.data 解密的秘钥
/start.sh	启动服务器
/manager.py	主函数

3.2 各部分结构及其说明

3.2.1 /app/controller/ 模块

该部分的结构为（以下内容省略/app/controller/ 根目录路径）：

/api/	用于 api 访问，非浏览器访问
/user/	仅以 user 路径做例子，用于用户定义的路径
/__init__.py	初始化
/errors.py	用于错误处理

3.2.1.1 /app/controller/__init__.py

前面提到，controller 用于管理全部的 route 路径，在用户每次请求的过程中，服务器根据用户提供的路径查找 controller 中对应的 route 函数进行执行。

（1） 引入 /app/controller/user/__init__.py 中的 user_blueprint 蓝图对象

```
from .user import user_blueprint
from .api import api_blueprint
```

（2） 对外提供对象接口，这里面的蓝图将用于在/app/__init__.py 中登记到 flask 中

```
__all__ = ['user_blueprint',
           'api_blueprint']
```

（3） 添加每次浏览器访问过程中，一个周期的上下文参数

```
@error_blueprint.app_context_processor
def inject_permissions():
    return dict(Permission=Permission)
```

（4）在完成每一次请求的时候，进行判断，数据库连接是否超时

```
@main_blueprint.after_app_request
    def after_request(response):
        pass
```

3.2.1.2 /app/controller/error.py

用于配置发送请求错误时返回的页面，分别包括 403 权限错误、404 无效路径、500 服务器内部错误。发生错误时，先判断页面类型为 html 类型（浏览器访问），还是 json（api 访问）

3.2.1.3 /app/controller/user/

这里仅以 user 的 route 举例：

/__init__.py 创建蓝图，蓝图名为 user，当浏览器访问 localhost:5000/user/test 时，

将会找到该 user 蓝图下注册的 test 路径（由于本内容简单，后面不再做赘述）

/forms.py 使用 wtforms 第三方库快速进行网页表单的创建与验证（存在的不足将在下一页 PPT 说明）

/views.py 保存多个 route 路径，用于用户访问后对应的执行函数

3.2.1.4 /app/controller/user/views.py

该部分包含全部的 route 程序，完整的一次函数执行，包括但不限于：
函数执行前：定义路径及其参数、修饰器进行用户登录和权限的检测、记录用户登录时间、判断用户邮箱是否授权；
函数执行时：数据库操作、flask_form 表单；
函数执行后：记录数据库查询是否超时、返回 render_template、重定向、返回错误页面。

（1） 函数执行前
① 定义路径及其参数

```
@user_blueprint.route('/edit-profile', methods=['GET', 'POST'])
```

② 修饰器进行用户登录和权限的检测

```
@login_required
@admin_required
```

③ 判断用户权限，同时记录用户登录时间，路径：/app/controller/auth/views.py

```
@auth_blueprint.before_app_request
def before_request():
    if current_user.is_authenticated:
        current_user.ping()
        if not current_user.confirmed \
                and request.endpoint \
                and request.blueprint != 'auth' \
                and request.endpoint != 'static':
            return redirect(url_for('auth.unconfirmed'))
```

（2） 函数执行时

① 数据库操作

数据库操作应尽量在/app/service/user/中进行，在/app/controller/user/views.py 中调用。方便在未来的数据库查询的改进。

注：查询的改进可以包括设计合理的缓存，适当的优化查询语句（例如用 join 替换 in）。以及数据库方面的优化，包括建立视图以增加代码执行效率，建立合适顺序的索引提高查找效率（索引顺序务必要适宜），建立聚簇等。

② flask_form 表单，详见：3.3.1 flask_form 与数据交互

（3） 函数执行后

① 记录数据库查询是否超时

位于 app/controller/__init__.py 中

```
@main_blueprint.after_app_request
def after_request(response):
    for query in get_debug_queries():
        if query.duration >=
            current_app.config['FLASKY_SLOW_DB_QUERY_TIME']:
            current_app.logger.warning(
                'Slow query: %s\nParameters: %s\nDuration: %fs\nContext: %s\n'
                % (query.statement, query.parameters, query.duration,
                    query.context))
    return response
```

② 返回 render_template 与重定向

每次在 views 中判断表单是否提交后都使用了重定向，即便是返回当前页面也使用了重定向，因为如果不使用重定向，直接返回 render_template，那么页面的状态会处于表单提交状态，用户在浏览器中刷新页面会提示是否重复提交表单。使用重定向会重新载入页面，再次调用后端的 route 函数并进入 form.validate_on_submit()的分支，重新返回页面，不会提示重复提交表单。

③ 返回错误页面

错误页面可以有多种返回方式，除了 flask 服务器默认的在出现错误后返回对应

的错误页面以外，还可以查询无结果直接返回错误：User.query.get_or_404(id)

3.2.2 /app/models/ 模块

Models 中应包含所有的数据模型，且结构如下图所示，不一定每一个数据库的表都用一个.py 文件，同时过于复杂的情况下也可以采用多级目录结构，只需要将每一个 model 文件都在__init__.py 中导入即可。

/__init__.py　　向外提供所有 models 接口
/user.py　　用户表
/role.py　　角色表
……

3.2.2.1 结构

数据模型应继承 db.Model 基类，包含数据项以及数据项的描述（primary_key、index、default、comment 等）：

```
class Comment(db.Model):
    __tablename__ = 'comments'
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.Text)
```

3.2.2.2 ForeignKey、relationship 与连表查询

flask_sqlalchemy 采用了 ForeignKey 和 relationship 来进行对象的调用。例如用户 user 类，每一个 user 都会发表 n 个 post 发言，user 和 post 属于 1 对 n 关系，数据模型中 post 应当有 user_id 外键。在 flask 中该关系在 post 中描述：

```
class Post(db.Model):
    __tablename__ = 'posts'
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('users.id'))
```

上述关系中，post 表中包含了 user_id 外键，用于描述 post->user 的关系，但该关系仅用在模型逆向到数据库过程中建立外键使用，与 flask 实际使用无关。

flask_sqlalchemy 提供了从 user->post 的关系，该关系不会出现在数据库中（因为 mysql 并没有反向关系连接），并定义为 relationship，该关系将用于 flask 中数据查询使用，定义如下：

```
class User(UserMixin, db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    posts = db.relationship('Post', backref='author', lazy='dynamic')
```

当获取 user 对象的全部 posts 时，只需 user.posts 即可，而获取 post 对象的 user 内容，并非使用 post.user_id，而是使用 post.author，利用 backref 反向指代名称获取。同时连表查询，若需要连接的两表存在 relationship，则可以直接用对象调用的方式获取。

例如，已知 post 的 id，获取发表了该 post 的 user 的电话：

（1）没有定义 relationship 情况下的连表查询，没有办法借助定义的数据对象，只能使用全局的 db 对象直接生成查询语句：

```
db.session.query(post.id, post.user_id, user.id, user.tel). \
    filter(post.id == post_id). \
    filter(user.id == post.user_id). \
    first()
```

（2）定义 relationship 情况下的连表查询

```
post = Post.query.get_or_404(id)
user_tel = post.author.tel
```

其中 filter 函数括号内需要以 **对象.属性==’值’** 的方式，filter_by 则是 **属性=’值’** 。请参考：

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	使用指定的值限定原查询返回的结果
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

常用的SQLAlchemy查询执行器

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，如果未查到，返回None
first_or_404()	返回查询的第一个结果，如果未查到，返回404
get()	返回指定主键对应的行，如不存在，返回None
get_or_404()	返回指定主键对应的行，如不存在，返回404
count()	返回查询结果的数量
paginate()	返回一个Paginate对象，它包含指定范围内的结果

3.2.2.3 密码管理

涉及到密码的属性，例如 user 的 pwd 以及加密 token 需要进行加密

（1） 密码类：使用 werkzeug.security 的 generate_password_hash 生成加密后密码，check_password_hash 生成解密后密码。具体请参考/app/models/user.py：

```
@property
def password(self):
    raise AttributeError('password is not a readable attribute')
@password.setter
def password(self, password):
    self.password_hash = generate_password_hash(password)
def verify_password(self, password):
    return check_password_hash(self.password_hash, password)
```

调用时，使用以下方式输入密码，而不是改变 password_hash 值

```
user = User(email=form.email.data.lower(),
            username=form.username.data,
            password=form.password.data)
db.session.add(user)
```

db.session.commit()

使用如下方式改变密码：
current_user.password = new_password

使用如下方式判断密码是否正确：
user.verify_password(user_input_password)

（2） 生成加密 token，用于邮箱验证，api 验证等
from itsdangerous import TimedJSONWebSignatureSerializer as Serializer

创建一个对象，包含密码与到期时间，使用该对象进行加密，采用和 json 类似的使用方式：

```
def generate_confirmation_token(self, expiration=3600):
    s = Serializer(current_app.config['SECRET_KEY'], expiration)
    return s.dumps({'confirm': self.id}).decode('utf-8')
```

确认 token 信息：

```
def confirm(self, token):
    s = Serializer(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
```

3.2.2.4 数据查询

部分查询可以写在 models 中，不过这将有悖于数据查询与视图层面分离的原则，虽然这样会让 views 中的函数调用模型时便捷的调用到模型中自带的查询，

但是将来在查看哪些模块用到数据查询的时候可能会漏掉隐藏在模型中的查询。因此应尽量将查询放到 service 层，这样即便是在 views 中重复使用的查询也能确保效率。

3.2.3 /app/webapp/ 模块

以 /app/webapp/ 为根目录，其下的文件（片段）分别为：

/static/	存储所有静态文件
css/	存储 css 样式表，内部建议采用二级以上目录
font/	存储字体
img/	存储图像
userprofile/	该目录下存储的为用户默认头像
js/	存储 js 代码
/template/	所有的 render_template 需要用到的模板
components/	render_template 不直接使用的 html 文件
include/	用于 include
macro/	用于 import
base.html	用于 extend
routes/	所有 render_template 直接使用的 html 文件
user/	user 目录
user.html	user 主页
post.html	post 主页
404.html	路径错误
403.html	权限错误
500.html	服务器内部错误
Index.html	首页

3.2.3.1 设立首页路径

首页在 /app/controller/main/views.py 中确定

```
@main_blueprint.route('/', methods=['GET', 'POST'])
def index():
    return render_template('index.html')
```

如何在 render_template 中提供 index.html 参数后找到 /webapp/templates 根目录，以及如何使用 url_for 定位静态图像后找到 static 根目录，需要在 /app/__init__.py 中创建 Flask 对象之初进行设定：

```
base_dir = os.path.abspath(os.path.dirname(__file__))
templates_dir = os.path.join(base_dir, 'webapp/templates')
static_dir = os.path.join(base_dir, 'webapp/static')
app = Flask(__name__, template_folder=templates_dir, static_folder=static_dir)
```

3.2.3.2 页面组件包含的三种方式：extends

flask 对于页面的渲染类似于 react 的组件化模式，充分的保证了代码的重复利用，将经常用到的模块写成组件来调用，使得整个页面如搭积木一般。页面之间的关系有三种，分别是 extends、import 和 include。

extends 用于子页面继承父页面。使用方式参考：
/app/webapp/templates/components/base.html

必须在第一行标明继承的父页面，本项目基于 bootstrap 因此继承 boostrap 基类：

```
{% extends "bootstrap/base.html" %}
```

继承父页面后，所有 html 代码必须写在父页面已经提供好的位置中，之后由编译器将该内容插入到父页面相应的位置中。否则代码无效，例如：
将 title 部分的空缺插入 Flasky 六个字母

```
{% block title %}Flasky{% endblock %}
```

在父页面的 content 空缺位置插入一段 html，同时，预留一个 page_content 的 block 区间为继承该 base 页面的子页面填充。

```
{% block content %}
<div class="container">
    {% block page_content %}{% endblock %}
</div>
{% endblock %}
```

在 /app/webapp/templates/index.html 中，继承了该父页面，同时重写了 page_content 内容，将父类中空缺的 page_content 覆盖。

```
{% extends "components/base.html" %}
{% block page_content %}
<div class="page-header">
    <h1>Hello！</h1>
</div>
{% endblock %}
```

3.2.3.3 页面组件包含的三种方式：import

使用 macro 模板宏，定义类似于函数的模块，参考：
/app/webapp/templates/components/macro/paging.html

例如函数名（模板名 pagination_widget），包含 3 个参数：

```
{% macro pagination_widget(pagination, endpoint, fragment='') %}
    {% for p in pagination.iter_pages() %}
        <li class="active">
            <a href="{{ url_for(endpoint, page = p, **kwargs) }}{{ fragment }}">
                {{ p }}
            </a>
        </li>
    {% endfor %}
{% endmacro %}
```

调用模板页 D:\学习记录\7.0 科研助理/app/webapp/templates/routes/user/user.html 的最后几行，使用 import 的方式，把 paging 当做函数调用：

```
{% import "components/macro/paging.html" as paging %}
{% if pagination %}
<div class="pagination">
    {{ paging.pagination_widget(pagination, '.user', username=user.username) }}
</div>
```

3.2.3.4 页面组件包含的三种方式：include

include 用于页面完全不变的直接插入进来，例如：
/app/webapp/templates/components/include/user/comments.html
直接插入到/app/webapp/templates/routes/user/user.html 的最后几行
{% include 'components/include/user/posts.html' %}

3.2.3.5 页面组件引用方式对比

上述三种页面组件中，extends 是把自己的代码插入到父类页面，而 import 与 include 都是把别的页面插入到自己的代码里。因此 extends 中全部的内容都需要写入到父类页面预设好的空中，而 import 和 include 则不需要。

3.2.3.6 页面结构阅读指南

代码阅读应当从 app/controller/user/views.py 中开始，route 中返回 return render_template('routes/user/user.html')，找到 app/webapp/routes/user/user.html。route 中存在的页面都是会被 render_template 访问到的。

页面第一行固定继承基类 {% extends "components/base.html" %}

第二行固定重写父类的 title 位置，{% block title %}Flasky - {{ user.username }}{% endblock %}，此时重写的 title 的 block 并不源于 components/base.html，而是源于父类继承的 bootstrap/base.html 中预留的 title 位置。后续主要代码要包含在{% block page_content %} {% endblock %}中，该 page_content 则是父类{% block page_content %}{% endblock %}中预留的位置。

注意，不要在后续继承了 components/base.html 页面的页面中重写 scripts 模块，这将覆盖 base 页面中已经写过的模块。引用 css 外联样式表和 js 应当直接写在 page_content 中。

3.2.3.7 url_for 控制的相对路径链接

url_for 动态获取相对地址，存在两种情况：
（1） 访问某个 route 路径
格式：url_for('蓝图名.蓝图中的函数名', 参数 1=值 1,参数 2=值 2) }}"
如果没有用蓝图，直接使用函数名即可。
例如：
user_blueprint = Blueprint('user', __name__)

@user_blueprint.route('/<username> /<pwd> ')
def user(username, pwd):
    pass
调用时，使用 url_for('user.user',username=123,pwd=456)
user.user 中的第一个 user 代表 Blueprint('user', __name__)中的名，第二个 user 代表函数名，并非路径名。

若使用 app 直接定义 route，如：
@app.route('/')
def index():
    pass
则使用 url_for('index')
这里的 index 同样指代了函数名 index

（2） 访问 static 文件
url_for('static', filename='以/app/webapp/static 为根目录的文件相对路径')
例如获取用户头像
url_for('static', filename='img/userprofile/blank.png')

3.2.3.8 ajax 异步数据加载

使用 ajax 可以异步加载数据，局部进行页面刷新，可以：
（1）避免 post 提交表单后强制刷新页面；
（2）优先加载文字和框架，将需要加载时间长的大量图片后续加载，提供良好的用户体验；
（3）减小服务器负载，需要大量加载的框架只在第一次加载，框架中的列表在翻页后只需要动态加载少量的表格数据即可；
使用方式请参考 /app/webapp/static/js/avatar.js
function user_avatar(t_url, uid){
    $.ajax({
        url:t_url,
        type:"POST",
        data:"id=" + uid,
        dataType: 'json',

```
            success:function(obj){
                $("#avatar_user").attr("src", obj.data);
            },
            error:function(e){
                alert("internal error happened when retrieving user avatar");
            }
        });
}
```

其中 t_url 用 url_for 获取：

```
user_avatar("{{ url_for('user.avatar') }}", "{{ user.id }}");
```

注意：ajax 本身也算一种 js 语句，因此在 $.ajax()的结尾一定要加分号，否则将无法执行。同时括号中各个部分之间用逗号分隔。
详细的 ajax 教程请自行查找

3.2.3.9 html 加载完成后执行的函数 onload

一个页面只能有一个 onload 函数，在页面完成渲染后被执行。因此在异步数据获取时，由于需要在页面加载完成后再获取用户头像，所以需要将该内容写在 onload 函数中：

```
<script type="text/javascript">
    window.onload = function(){
        user_avatar("{{ url_for('user.avatar') }}", "{{ user.id }}");
    }
</script>
```

但由于页面存在继承关系，请不要在不被继承的页面中使用 onload，这样在最终页面使用 onload 后父类的 onload 会被覆盖，导致难以察觉的错误。

3.2.4 /app/decorators.py

项目中需要用到的修饰器目前只有权限判断，未来有更多需要修饰器的地方可以将修饰器统一写到该文件中。修饰器的具体写法请参考该文件。

3.2.5 /app/exception.py

因为尚不存在特殊的异常类型，因此文件为空。异常类如果数量过多也可以建立新的目录单独存放，异常类应当继承 Exception 基类，通过接受相关的异常来做出更为准确的错误诊断。

例如，创建一个用户类异常，实现用户登录错误的处理：

```
class UserException(Exception):
    def __init__(self, error_reason):
        assert(type(error_reason) == str)
        self.reason = error_reason
    def __str__(self):
        return(self.reason)
```

处理请求：

```
def login(username, password):
    if login_user(username, password) == False:
        raise UserException(“login fail”)
```

接收该异常：

```
try:
    login(username, password)
    do_something()
except UserException as UE:
    print(“error: ”, str(UE))
    # print(“error: ”, UE.reason)
```

这样可以只过滤 UserException，将其他的错误类型向上提交，直到能处理为止。

3.2.6 /scripts/ 模块

该模块为与服务器启动无关的工具

3.2.6.1 创建服务器初始化配置

目录结构：

/start_config.py	用于配置服务器初始化数据
/config/__init__.py	导出读写配置的 Read、Write 和密码检测 CheckPwd

/config/AES.py	使用 AES 加密的类，包括加密与解密方法
/config/check_pwd.py	简化操作，用户输入密码、读取配置、返回参数组
/config/errortype.py	错误类型定义
/config/read_file.py	读取、解密配置文件，tile 函数分组取出配置内容
/config/write_file.py	写配置，推荐加密后写入配置
/config/xml_create.py	创建用于配置文件的 xml

更改其中内容请自行阅读代码。
使用方法详见/start_config.py，

```
t = Write(key="test_pwd")
t.add_item("key", "SECRET_KEY", "123456")
t.add_item("sql", "SQLALCHEMY_TRACK_MODIFICATIONS", "F")
t.add_item("sql", "SQLALCHEMY_RECORD_QUERIES", "T")
t.write(path)
```

其中 t.add_item 的第一个参数为分组参数，在读取配置数据时，可以根据分组选择性读取数据：

```
from scripts.config. read_file import Read
file = Read(“test_pwd”)
config_data = Read.tile(xml_=file, element_list=[“sql”], interpret=True)
```

此时读取到的只有 element_list 中的”sql”组，即：

```
{
    "SQLALCHEMY_TRACK_MODIFICATIONS"=False,
    "SQLALCHEMY_RECORD_QUERIES"=True
}
```

也可以使用/manager.py 中的方法：

```
config_dict = CheckPwd.check_pwd_input_with_force("config.data", password=key,
                group=("emile", "sql", "key", "sql_dev", "page", "img"))
```

3.2.6.2 生成测试数据

生成具体代码详见 /scripts/fake/，使用过程详见/manager.py。该内容非主要结构，不做详细介绍。

3.2.6.3 单元测试

单元测试模块被我删除了，因为我认为单元测试过于复杂且冗长，详细的测试方式请参考 https://github.com/miguelgrinberg/flasky 中的 tests 模块

3.2.6.3 route 测试

get_route 函数，用于测试每一个可以访问到的路径。具体用法参考/manager.py 中 run 函数中使用的 get_route

3.2.7 /manager.py

该文件为主函数。文件中的 @manager.command 用于设置命令行参数，详细使用方法请自行查找。

3.3 其他结构

3.3.1 flask_form 与数据交互

flask_form 与数据交互，使用 wtforms 的使用方法，优点、缺点，以及数据交互的代替方法。

flask_form 使用方法：
（1）定义：继承 FlaskForm 基类，所有的页面表单均由 StringField 等快速定义，支持 validators 提供的简单验证与正则表达式验证，以及 validate_开头的函数用于对应表单项的复杂验证；
（2）判断用户表单：在 views 中创建 form = EditProfileForm() 表单对象，通过 form.validate_on_submit() 判断是否有用户输入。
例如：浏览器首次访问/user/edit-profile 路径修改用户资料，服务器调用该处理函数，首先创建 form = EditProfileForm() 表单对象，然后判断 form.validate_on_submit()是否有用户提交。返回 render_template 模板（即返回的页面），并附带 form 参数。在 template 前端页面直接使用 quick_form 函数生成附带了 name 参数的表单。如果用户提交了表单数据，不同于前端 html 的表单需要定义跳转路径，wtforms 会直接将表单数据以 post 的方式提交到当前路径

/user/edit-profile，此时服务器会第二次调用该处理函数，form.validate_on_submit()判断为真，返回 redirect 重定向一个新的页面，实现表单提交后页面跳转。

优点：
（1）使用方便；
（2）安全性高，所有的安全性验证自动生成，避免手写出 bug

缺点（由本人总结，并非官方认定的缺点）：
（1） 固定的跳转方式，使用不灵活。生成的表单会直接提交到当前地址，并由后端判断是否有表单提交；
（2）前端页面难以改变。除了 quick_form 的方式生成前端页面显示的表单，定制化的生成前端的表单难度很大；
（3）对于非文本类数据（图像、视频、文件）很难处理，需要配合其他第三方库使用，大大增加了复杂度，使得本来以简洁为优势的 wtforms 变得更加复杂。
（4）难以实现前端页面更加灵活的 js 触发；
（5）无法实现异步数据传输，即：每次提交表单，都需要强制性刷新页面，页面数据一次性加载，无法局部加载、刷新。

代替方法：
（1）使用 js 或 jquery 获取表单数据，发送 get 请求，将数据以拼接字符串的形式显示的放在路径里，例如 localhost:5000/user?username=100&pwd=abc；
（2）使用 post 方法，直接使用 html 中的 form，当然可以用 bootstrap 中的 form，更加美观；
（3）使用 ajax，实现异步数据加载，页面局部刷新，该内容在后面介绍本项目前端部分有详细说明。

3.3.2 控制用户登录的 flask_login

flask_login 需要设置的内容很多：
/app/__init__.py 中第一次初始化 login 模块，并设置登录界面 auth.login

```
from flask_login import LoginManager
login_manager = LoginManager()
login_manager.login_view = 'auth.login'
```

在/app/controller/models/user.py 中，user 继承 flask_login 的 UserMixin

```
from flask_login import UserMixin
class User(UserMixin, db.Model):
    __tablename__ = 'users'
```

在 /app/controller/models/user.py 中定义 login_required 函数，每次使用 @login_required 的时候会调用该函数，下面为固定写法：

```
@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))
```

在 /app/controller/auth/views.py 中设置 login 和 logout

使用时常见操作为：

```
from flask_login import login_required, current_user
```

其中 @ login_required 的用法强制用户登录

current_user 则为继承了 UserMixin 的对象（本项目中为 User）

3.3.3 配置 flask_mail

注意，不同的邮箱需要配置的内容不同，本项目采用 qq 邮箱，因为 qq 邮箱对于 SMTP 邮件协议发送的邮件不做限制，网易邮箱会限制 SMTP 的使用导致邮件无法正常发送。qq 邮箱 SMTP 服务的授权码存在有效时间，所以当邮件无法发送时应当检查是否为授权码到期导致的。

邮件初始化配置请参考：3.2.5.1 创建服务器初始化配置

邮件发送采用异步方式，提高浏览器响应速度。浏览器发送邮件发送请求后，服务器会直接返回发送结果，同时调用另一个线程发送邮件，避免发邮件过程缓慢导致浏览器前端长时间无响应。需要注意的是，当需要发送的邮件数量过大时，开启大量的线程将会大大降低效率，应当**采用任务队列进行优化**。

3.3.4 数据库迁移

数据库由模型导入数据库、数据库更新，详见 /manager.py。

以下为固定用法

```
import flask_migrate
app = create_app(config_dict)
flask_migrate.Migrate(app, db)
```

数据库初始化：

```
flask_migrate.init()            #生成 /migrations/ 目录
flask_migrate.migrate()         # 将模型导入到 /migrations/ 目录
flask_migrate.upgrade()         # 将迁移保存到数据库
```

当模型更新时，不需要删除数据库后重新导入，而使用：

```
flask_migrate.migrate()
flask_migrate.upgrade()
```

注：需要在 /app/scripts/start_config.py 中正确的输入数据库 url

3.3.5 多媒体以及文件数据存储

为了确保系统 IO 速度、以及某些文件系统单级目录的限制，图片或其他多媒体数据采用多级目录存储。详见 app/controller/user/views.py 中的 upload_avatar 函数，计算用户上传图像的哈希值，取前两 8 比特进行分级，这样可以确保第一层目录有至多 256 个（2^8）子目录，每个子目录下存储对应的图片。该部分内容可以新建一个 py 文件或写到 service 中，使得程序更加直观清晰。

3.3.6 大量图片加载问题

大量图片加载时引起的卡顿，可以有多种解决办法，除了 4.3.1 前后端分离 中提到前后端分离的做法，还有以下几种可以直接应用的方法：

（1） 将图片压缩为 base64 格式，然后一次请求直接获取全部数据，在前端进行处理。这样可以节约请求次数。如果每张图片都用 url 方式加载，浏览器将会发出很多次请求。该方法被应用到项目中，详见后端 /app/controller/user/views.py 中的 get_batch_avatar 函数发送批量图片数据，前端 /app/webapp/static/js/avatar.js 的 user_avatar_batch 函数使用 ajax 异步的从后端获取批量图片数据，以及在 /app/webapp/routes/index.html 中调用该 js 函数；

（2） 图片懒加载，滚动到下一行的时候才用 ajax 动态加载图片；

（3） 采用更加优化的压缩算法。

3.4 api 访问

api 根目录为：app/controller/api
部分结构为：

/v1_0/	表示 api 版本号，url 路径为 api/v1.0/
/routes	该 api 版本下全部 routes
/users.py	处理 user 相关的 api 的 route
/posts.py	处理 post 相关的 api 的 route
/service	向 routes 提供功能
/users.py	向 routes/user 提供功能
/posts.py	向 routes/post 提供功能
__init__.py	创建 api 蓝图，加载全部 route 文件
authorization.py	登录验证
decorators.py	权限验证，请自行查看
errors.py	错误页面处理，请自行查看
__init__.py	向外提供蓝图接口，请自行查看

3.4.1 api 版本号与创建

传统的浏览器对于服务器的访问时动态的，服务器的改动不会对用户造成影响，但是调用 api 的则被写入到程序中，api 接口往往不能随意修改。在推出新版本 api 的同时，必须要兼顾老版本的 api。因此 api 必须要引入版本号。

在 /app/controller/api/v1_0/__init__.py 中创建完 api 后，要手动 import 其他在 routes 目录下的文件，因为在程序执行过程中，该页面会被 /app/controller/api/__init__.py 引用，而这个页面又会被/app/controller/ __init__.py 引用。但是 routes 中的其他文件不会被 python 解释器自动运行，会导致无法加载 route。因此这里必须要手动引入全部的 route 中的 py 文件。

3.4.2 无 session 的用户权限验证

Session 用于存储临时的用户数据，session 的存储有两种形式：

（1）存储在服务器端。此时需要考虑服务器集群之间的 session 一致性问题；
（2）存储在用户端。此时需要考虑 session 大小的问题与加密的问题。存储在客户端的 session 容易受到攻击，同时每次访问传输大量 session 数据会导致网络拥塞。
由于 api 访问没有 session 存在，因此必须在每次访问的时候携带用户信息。在/app/controller/api/v1_0/authorization.py 中，引入了 HTTPBasicAuth 用于验证，HTTPBasicAuth 可用作无状态的密码验证：

```
from flask_httpauth import HTTPBasicAuth
auth = HTTPBasicAuth()
```

HTTPBasicAuth 提供了三个常用接口 ：
（1）@auth.error_handler，用于产生错误后调用；
（2）@auth.login_required，加上该修饰器的函数，会在执行前自动调用 verify_password 进行密码检验。而项目中 login_required 被加载了 before_request 里，因此每次执行 api/v1_0/的 url，都会自动触发 login_required，进而调用 verify_password；
（3）@auth.verify_password，

```
@auth.verify_password
def verify_password(email_or_token, password):
    pass
```

是@auth.verify_password 的回调函数，回调函数有多种，这里用了两个参数的回调函数。其他版本请咨询查找。当 password 为空时，说明用户携带了 token 访问，则需要验证 token 并从 token 中取出用户 id；当 password 不为空是，说明用户准备使用密码登录，则从数据库中查找用户邮箱，并取出密码进行验证。

该模块还有一个用于获取 token 的函数，get_token，用于不携带用户名密码的调用。

3.4.3 service 层的作用

该项目参考的程序中，用于将 post、user 对象转化为 json 数据返回用户的函数，被写在了 models 中。但实际使用时，不同版本的 api 可能返回不同的结果，因此没有共性的返回结果不应写到 models 中，应当单独写在 api/v1_0/目录里。

3.4.4 api 测试与 json 数据访问

一次基本的用户请求至少要分两次，第一次用于获取 token，第二次携带 token 用于数据获取。这里有三个注意点：

（1）用户的每次访问应携带密码或 token，用于提供@api_blueprint_v1_0.before_request 中调用的 verify_password 中的 email_or_token, password 这两个参数；

（2）在 /app/controller/api/v1_0/routes/posts.py 中，使用了 request.json，而 request 传递参数的方式有很多种，包括 request.args、request.form 以及 request.file 等，其中 request.json 要求访问的 http 报文的 Header 必须带有 Content-Type，值为 application/json，要明确告诉 flask 服务器，发送的就是 json 格式的数据，才能从 request.json 中获取。同时必须使用 post 或 put 的方法，不能使用 get，因为 get 方法的报文不能携带 body，也就无法携带除 url 地址栏的 args 以外的参数数据。数据放在报文的 body 中，以 json 格式传递，否则会导致解析错误；

（3）api 访问应当在在 http 请求的 Header 中将 Accept 的值改为 application/json，而不是 text/xml，因为在 app/controller/errors.py 中，发生错误时会判断是否为 json 类型的访问，如果为 json 类型的访问，会直接返回错误的原因，否则会返回一个网页。

综上所述，举例说明请求的字段，例如：

（1）请求获取 tokens

邮箱 123@qq.com，密码 888

服务器地址为本地服务器，127.0.0.1

服务器端口 5000

api 版本号为 1.0

则正确的请求为：

需要发送 post 请求 http://123@qq.com:888@127.0.0.1:5000/api/v1.0/tokens/

返回结果：

{
"expiration": 3600,
"token":"eyJhbGciOiJIUzI1NiIsImlhdCI6MTYzMzE3OTMxOSwiZXhwIjoxNjMzMTgyOTE5fQ.eyJpZCI6MX0.OC00FA3waPlzf26Fr_6PBcVXXF_v_BRsk18eJMxyY8"
}

（2）直接携带 token 访问，获取某用户信息，正确的请求为：

发送 get 请求

http://eyJhbGciOiJIUzI1NiIsImlhdCI6MTYzMzE3OTkxOSwiZXhwIjoxNjMzMTgzNTE5fQ.eyJpZCI6MX0.CUPBxvkgWUJrtUbrfaGeVrtnDdkMpCGcnyHzWmZga5U@127.0.0.1/api/v1.0/users/1

（3）json 数据 post 与 put 的方法：

```
res = requests.post("http://token 值:@服务器地址:服务器端口/api/v1.0/posts/",
                    json={"body":"待发表的 post 内容"})
```

使用 postman 测试，

1. 请发送 json 请求，在请求栏中的 Body 中选择 raw，
   下方框输入{"body":"待发表的 post 内容"}，
2. 选择 JSON，不是 Text，或者选择 Header，找到 Content-Type，
   将后面的值改为 application/json，
3. 将 Header 中的 Accept 值改为 application/json

（这步用于 /app/controller/errors.py 中对于请求类型的判断

第四章 可扩展部分

可扩展部分仅用于提供思路，具体操作方法请自行搜索

4.1 安全性

4.1.1 HTTPs 加密访问

使用 Https 之前，请先参考 https 基本原理：
https://www.bilibili.com/video/BV1P7411375j
https://www.bilibili.com/video/BV1q7411E7oa
https://www.bilibili.com/video/BV1C7411L7UZ
https://www.bilibili.com/video/BV1KE41177kK
https://www.bilibili.com/video/BV1xE411N7SP
参考/manager.py 中的 sslrun 函数

4.1.2 限流器

使用 flask_limiter 限制用户访问频率，控制粒度可以达到单个函数。但更为精细的控制需要自己写。

4.2 其他功能

4.2.1 flask-bable 多语言

使用 flask-bable 支持多国语言，默认为自动翻译，也可以手动查看翻译页进行订正。

4.3 服务器部署

4.3.1 Nginx 分布式部署

通过 Nginx 网关统一分配请求，实现多机之间的负载均衡。

4.3.2 前后端分离

将所有与磁盘 IO 无关的静态页面用一个服务器存储，所有需要数据库或 IO 操作的请求放到另一个服务器（前后端分离需要配置跨域请求，请自行查找）。

4.3.3 数据库

请参考数据库 CAP 原理，以及数据库的选择问题。在更换数据库后，如果所有数据库操作都在 service 层进行，那么可以方便快速的进行修改。对于较为灵活的 mysql 数据库，请根据需求选择数据库引擎（默认为 innodb）。数据库访问量大时，考虑的优先级可以为：

（1） 建立合适的索引；

（2） 配置缓存。

访问量过大时，请参考百万级并发服务器架构。