

Python 实战

20180327

🌸 Day 1. 搭建开发环境

① 开发环境

pip 安装开发 Web App 需要的第三方库:

异步框架 aiohttp;

前端模板引擎 jinja2;

MySQL 5.x 数据库;

MySQL 的 Python 异步驱动程序 aiomysql:

② 项目结构

myblog/			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
	ios/		存放 iOS App 工程
	LICENSE		代码 LICENSE

创建项目目录后, 建立 git 仓库并同步至 GitHub, 保证代码修改的安全。

🌸 Day 2. 编写 Web App 骨架

www/app.py:

```
import logging; logging.basicConfig(level=logging.INFO)
import asyncio
from aiohttp import web

def index(req):
    return web.Response(body='<h1 style="color:red">hikari\'s web</h1>',
content_type='text/html')

async def init(loop):
    app = web.Application(loop=loop)
    app.router.add_route('GET', '/', index)
    server = await loop.create_server(app.make_handler(), '127.0.0.1', 8000)
    logging.info('Server started at http://127.0.0.1:8000...')
    return server

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

结果:

hikari's web

🌸 Day 3. 编写 ORM

在一个 Web App 中，所有数据，包括用户信息、发布的日志、评论等，都存储在数据库中。此处选择 MySQL。

首先把常用的 SELECT、INSERT、UPDATE 和 DELETE 操作用函数封装。

由于 Web 框架使用 aiohttp，是基于协程的异步模型。在协程中，不能调用普通的同步 IO 操作，因为所有用户都是由一个线程服务的，协程的执行速度必须非常快，才能处理大量用户的请求。而耗时的 IO 操作不能在协程中以同步的方式调用；否则等待一个 IO 操作时，系统无法响应任何其他用户。

这就是异步编程的一个原则：一旦决定使用异步，则系统每一层都必须是异步。aiomysql 模块为 MySQL 数据库提供了异步 IO 的驱动。

① 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：
www/orm.py:

```
import logging
import aiomysql

def log(sql, args=()): # 控制台打印 SQL 语句
    logging.info('SQL: {}'.format(sql))

async def create_pool(loop, **kw): # 创建全局连接池
    logging.info('create database connection pool...')
    global __pool
    __pool = await aiomysql.create_pool(
        host=kw.get('host', 'localhost'),
        port=kw.get('port', 3306),
        user=kw['user'],
        password=kw['password'],
        db=kw['db'],
        charset=kw.get('charset', 'utf8'),
        autocommit=kw.get('autocommit', True),
        maxsize=kw.get('maxsize', 10),
        minsize=kw.get('minsize', 1),
        loop=loop
    )
```

② SELECT 查询

```

async def select(sql, args=(), size=None): # 查询
    log(sql, args)
    global __pool
    async with __pool.get() as conn:
        async with conn.cursor(aiomysql.DictCursor) as cur:
            # SQL 语句的占位符是?,而 MySQL 的占位符是%s
            await cur.execute(sql.replace('?', '%s'), args)
            # 获取指定数量的数据,若未指定获取全部
            # 调用子协程(在一个协程中调用另一个协程)并获得子协程的返回结果
            ret = await cur.fetchmany(size) if size else await cur.fetchall()
            logging.info('rows returned: {}'.format(len(ret)))
        return ret

```

async 和 await 看着还是很蛋疼...

③ Insert, Update, Delete

```

async def execute(sql, args=(), autocommit=True): # 增删改
    log(sql, args)
    async with __pool.get() as conn:
        if not autocommit: # 如果不是自动提交,手动开始提交回滚
            await conn.begin()
        try:
            async with conn.cursor(aiomysql.DictCursor) as cur:
                await cur.execute(sql.replace('?', '%s'), args)
                affected = cur.rowcount # 影响的行数
            if not autocommit:
                await conn.commit()
        except BaseException:
            if not autocommit:
                await conn.rollback()
            raise # 有异常,回滚并抛出异常
    return affected

```

④ ORM

设计 ORM 需要从上层调用者角度来设计(自顶向下?)。
 考虑如何定义一个 User 对象,将数据库表 user 和它关联。
 一般是这样定义:

```

from orm import Model, StringField, IntegerField
class User(Model):
    __table__ = 'user'
    id = IntegerField(primary_key=True)
    name = StringField()

```

注意: __table__、id 和 name 是类属性,不是实例属性。所以,在类级别上定义的属性用来描述 User 对象和表的映射关系,而实例属性必须通过 __init__() 方法去初始化,两者互不干扰。

⑤ 定义 Model

首先定义所有 ORM 映射的基类 Model:

```
# 可用函数或类当做元类,此处用类继承于 type
class ModelMetaclass(type):
    def __new__(cls, name, bases, attrs): # 3 个参数分别为类名,父类,类属性和值的字典
        if name == 'Model': # Model 类是父类,没有字段,不需要处理
            return type.__new__(cls, name, bases, attrs)
        table_name = attrs.get('__table__', None) or name # 获取表名,没有就用类名做表名
        logging.info('found model: {} (table: {})'.format(name, table_name))
        mappings = dict() # 保存所有字段的映射
        fields = [] # 保存非主键字段
        primary_key = None
        for k, v in attrs.items():
            if isinstance(v, Field): # 类属性值是字段的实例
                logging.info('found mapping: {} ==> {}'.format(k, v))
                mappings[k] = v # 保存字段的映射,类属性名-->Field 对象实例
                if v.primary_key: # 找到主键保存
                    if primary_key: # 若之前已经找到主键,又发现主键,则主键不唯一,抛出异常
                        raise RuntimeError('Duplicate primary key for field: {}'.format(k))
                    primary_key = k
                else:
                    fields.append(k) # 保存非主键字段
        if not primary_key: # 遍历完所有字段没有发现主键
            raise RuntimeError('Primary key not found.')
        for k in mappings.keys(): # 将原来类属性字典是字段的清空
            attrs.pop(k)
        escaped_fields = list(map(lambda f: '{}'.format(f), fields))
        # 重新设置类属性与值的字典,将新创建的类返回
        attrs['__mappings__'] = mappings # 保存属性和列的映射关系
        attrs['__table__'] = table_name
        attrs['__primary_key__'] = primary_key # 主键属性名
        attrs['__fields__'] = fields # 除主键外的属性名
        # 以下四种方法保存了默认的增删改查操作,反引号``是为了避免与 sql 关键字冲突
        attrs['__select__'] = 'select `{}`, {} from {}'.format(primary_key, ','.join(escaped_fields),
            table_name)
        attrs['__insert__'] = 'insert into `{}` ({}, `{}`) values {}'.format(table_name,
            ','.join(escaped_fields), primary_key, create_args_string(len(escaped_fields) + 1))
        attrs['__update__'] = 'update `{}` set {} where `{}`=?'.format(table_name,
            ','.join(map(lambda f: '%s=?' % (mappings.get(f).name or f), fields)), primary_key)
        attrs['__delete__'] = 'delete from `{}` where `{}`=?'.format(table_name, primary_key)
        return type.__new__(cls, name, bases, attrs)

# 这样,任何继承自 Model 的类(比如 User),会自动通过 ModelMetaclass 扫描映射关系
# 并存储到自身的类属性如__table__、__mappings__中
```

```

class Model(dict, metaclass=ModelMetaclass):
    # 所有 ORM 映射的基类 Model
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def __getattr__(self, item): # 获取值
        try:
            return self[item]
        except KeyError:
            raise AttributeError("'Model' object has no attribute '{}'.format(item))

    def __setattr__(self, key, value): # 设置值
        self[key] = value

    def get_value(self, key): # 获取值,没有此 key 则为 None
        return getattr(self, key, None)

    def get_value_or_default(self, key): # 使用字段的默认值
        value = getattr(self, key, None)
        if value is None:
            field = self.__mappings__[key] # 获取字段字符串对应字段实例
            if field.default is not None: # 如果此类型有默认值,采用该默认值
                value = field.default() if callable(field.default) else field.default # 函数()或值
                logging.debug('using default value for {}: {}'.format(key, value))
                setattr(self, key, value)
        return value

    # 定义类方法
    @classmethod
    async def find_all(cls, where=None, args=None, **kw):
        sql = [cls.__select__]
        if where: # where 条件查询
            sql.append('where')
            sql.append(where)
        if args is None:
            args = []
        order_by = kw.get('order_by', None)
        if order_by: # order 排序
            sql.append('order by')
            sql.append(order_by)
        limit = kw.get('limit', None)
        if limit is not None: # 指定偏移,限制查询条数
            sql.append('limit')
            if isinstance(limit, int):

```

```

        sql.append('?')
        args.append(limit)
    elif isinstance(limit, tuple) and len(limit) == 2:
        sql.append('?, ?')
        args.extend(limit)
    else:
        raise ValueError('Invalid limit value: {}'.format(limit))
ret = await select(' '.join(sql), args) # 拼接 sql 语句,查询
return [cls(**i) for i in ret]

@classmethod # 这是什么方法?难道是 count?
async def find_number(cls, select_field, where=None, args=None):
    sql = ['select {} _num_ from {}'.format(select_field, cls.__table__)]
    if where:
        sql.append('where')
        sql.append(where)
    ret = await select(' '.join(sql), args, 1)
    if len(ret) == 0:
        return None
    return ret[0]['_num_']

@classmethod
async def find(cls, pk): # 根据指定主键获取一行数据
    ret = await select('{} where `{}`=?'.format(cls.__select__, cls.__primary_key__), [pk], 1)
    if len(ret) == 0:
        return None
    return cls(**ret[0])

# 以下为实例方法
async def save(self): # 插入数据到数据库
    args = list(map(self.get_value_or_default, self.__fields__))
    args.append(self.get_value_or_default(self.__primary_key__))
    rows = await execute(self.__insert__, args)
    if rows != 1:
        logging.warning('failed to insert record: affected rows: {}'.format(rows))

# 修改数据
async def update(self):
    args = list(map(self.get_value, self.__fields__))
    args.append(self.get_value(self.__primary_key__))
    rows = await execute(self.__update__, args)
    if rows != 1:
        logging.warning('failed to update by primary key: affected rows: {}'.format(rows))

```

```
# 删除数据
async def remove(self):
    args = [self.get_value(self.__primary_key__)]
    rows = await execute(self.__delete__, args)
    if rows != 1:
        logging.warning('failed to remove by primary key: affected rows: {}'.format(rows))
```

⑥ 定义字段

```
class Field(object): # 字段父类,感觉上面与数据库交互字段不用此处 name,而是用类属性名
    def __init__(self, name, column_type, primary_key, default):
        self.name = name # 字段名字,也就是此处 name 没卵用
        self.column_type = column_type # 字段类型
        self.primary_key = primary_key # 字段是否为主键
        self.default = default # 字段的默认值
    def __str__(self):
        return '<{}, {}>'.format(self.__class__.__name__, self.column_type, self.name)

class StringField(Field): # 映射 varchar 的 StringField
    def __init__(self, name=None, primary_key=False, default=None, ddl='varchar(100)'):
        super().__init__(name, ddl, primary_key, default)

class BooleanField(Field): # 布尔类型
    def __init__(self, name=None, default=False):
        super().__init__(name, 'boolean', False, default)

class IntegerField(Field): # 整数
    def __init__(self, name=None, primary_key=False, default=0):
        super().__init__(name, 'bigint', primary_key, default)

class FloatField(Field): # 实数
    def __init__(self, name=None, primary_key=False, default=0.0):
        super().__init__(name, 'real', primary_key, default)

class TextField(Field): # 文本
    def __init__(self, name=None, default=None):
        super().__init__(name, 'text', False, default)
```

⑦ 测试：不知道怎么测试协程...

在 create_pool()后面添加：

```
async def destroy_pool(): # 程序结束之前手动关闭 mysql 连接池
    global __pool
    if __pool:
        __pool.close()
        await __pool.wait_closed()
```

以 User 为例测试：

```

import asyncio
class User(Model):
    __table__ = 'user'
    id = IntegerField('id', primary_key=True)
    name = StringField('name')

async def test(loop, db, lst):
    await create_pool(loop, **db)
    for i in range(len(lst)):
        user = User()
        user.id = i + 1
        user.name = lst[i]
        await user.save()
    print('test insert over!')

async def show(loop, db):
    await asyncio.sleep(1)
    ret = await User.find_all('id between 3 and 5')
    print(ret)
    await destroy_pool()
    print('show over!')

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    database = {
        'user': 'root',
        'password': 'mysql',
        'db': 'test'
    }
    lst = ['rin', 'maki', 'nozomi', 'nico', 'umi', 'kotori']
    task = [test(loop, database, lst), show(loop, database)]
    loop.run_until_complete(asyncio.wait(task))
    loop.close()

```

结果:

```

test insert over!
[{'id': '3', 'name': 'nozomi'}, {'id': '4', 'name': 'nico'}, {'id': '5', 'name': 'umi'}]
show over!

```

查看 User.__dict__:

```

mappingproxy({
    '__module__': '__main__',
    '__table__': 'user',
    '__mappings__': {'id': <__main__.IntegerField object at 0x062C3C30>,
    'name': <__main__.StringField object at 0x062C3C50>},
    '__primary_key__': 'id',
    '__fields__': ['name'],

```



```

'__select__': 'select `id`, `name` from `user`,
'__insert__': 'insert into `user` (`name`, `id`) values (?, ?)',
'__update__': 'update `user` set `name`=? where `id`=?',
'__delete__': 'delete from `user` where `id`=?',
'__doc__': None})

```

元类编写 ORM 对新手不友好啊...元类一般很少用，算 Python 黑魔法，但据说很多源码底层也用元类？

重点：

- ① ORM 对象映射关系：数据库表映射为一个类，一行数据映射为一个实例；
- ② 元类 metaclass：继承于 type，用来创建类；一个类继承了元类，会调用元类的方法初始化、创建类；
- ③ 协程异步，一处异步，处处异步。

20180328

🌸 Day 4. 编写 Model

- ① 定义 3 个模型：User、Blog、Comment 继承于 orm 的 Model 类

```

import time
import uuid
from orm import Model, StringField, BooleanField, FloatField, TextField

def next_id(): # 生成唯一 id
    # uuid4()通过伪随机数得到 uuid,32 位 16 进制
    return '{:015d}{}000'.format(int(time.time() * 1000), uuid.uuid4().hex)

class User(Model): # 用户模型
    __table__ = 'users'
    id = StringField(primary_key=True, default=next_id, ddl='varchar(50)')
    email = StringField(ddl='varchar(50)')
    pwd = StringField(ddl='varchar(50)')
    admin = BooleanField() # 是不是管理员
    name = StringField(ddl='varchar(50)')
    image = StringField(ddl='varchar(500)')
    created_at = FloatField(default=time.time) # 创建时间

class Blog(Model): # 博客模型
    __table__ = 'blogs'
    id = StringField(primary_key=True, default=next_id, ddl='varchar(50)')
    user_id = StringField(ddl='varchar(50)')
    user_name = StringField(ddl='varchar(50)')
    user_image = StringField(ddl='varchar(500)')
    name = StringField(ddl='varchar(50)')
    summary = StringField(ddl='varchar(200)')
    content = TextField()
    created_at = FloatField(default=time.time)

```

```
class Comment(Model): # 评论模型
    __table__ = 'comments'
    id = StringField(primary_key=True, default=next_id, ddl='varchar(50)')
    blog_id = StringField(ddl='varchar(50)')
    user_id = StringField(ddl='varchar(50)')
    user_name = StringField(ddl='varchar(50)')
    user_image = StringField(ddl='varchar(500)')
    content = TextField()
    created_at = FloatField(default=time.time)
```

② 初始化数据库表

如果表的数量很少，可以手动创建表的 SQL 脚本。

data.sql:

```
drop database if exists myblog;
create database myblog charset=utf8;
use myblog;
-- 将 myblog 数据库下的所有表的操作权限授予 root 用户,认证密码为 mysql
grant select, insert, update, delete on myblog.* to 'root'@'localhost' identified by 'mysql';

create table users (
    `id` varchar(50) not null,
    `email` varchar(50) not null,
    `pwd` varchar(50) not null,
    `admin` bool not null,
    `name` varchar(50) not null,
    `image` varchar(500) not null,
    `created_at` real not null,
    unique key `idx_email` (`email`),
    key `idx_created_at` (`created_at`), -- key 是索引吗?
    primary key (`id`)
);

create table blogs (
    `id` varchar(50) not null,
    `user_id` varchar(50) not null,
    `user_name` varchar(50) not null,
    `user_image` varchar(500) not null,
    `name` varchar(50) not null,
    `summary` varchar(200) not null,
    `content` mediumtext not null,
    `created_at` real not null,
    key `idx_created_at` (`created_at`),
    primary key (`id`)
);

create table comments (
    `id` varchar(50) not null,
```

```

`blog_id` varchar(50) not null,
`user_id` varchar(50) not null,
`user_name` varchar(50) not null,
`user_image` varchar(500) not null,
`content` mediumtext not null,
`created_at` real not null,
key `idx_created_at` (`created_at`),
primary key (`id`)
);

```

如果表的数量很多，可以从 Model 对象直接通过脚本自动生成 SQL 脚本。
 命令行切换到 data.sql 的根目录：mysql -u root -p < data.sql

③ 编写数据访问代码

以 User 对象为例，test.py:

```

import orm
from models import User
import asyncio

async def test(loop):
    await orm.create_pool(loop, user='root', password='mysql', db='myblog')
    u = User(name='hikari', email='hikari@example.com', pwd='1234', image='about:blank')
    await u.save()
    await orm.destroy_pool()

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(test(loop))
    loop.close()

```

🔧 Day 5. 编写 Web 框架

aiohttp 相对比较底层，用其编写 URL 处理(视图)函数步骤：

```

# ① 编写一个用 async/await 装饰的函数
async def handle_url_xxx(request):
    # ② 传入的参数需要自己从 request 中获取
    url_param = request.match_info['key']
    query_params = parse_qs(request.query_string)
    # ③ 自己构造 Response 对象
    text = render('template', data)
    return web.Response(text.encode('utf-8'))

```

这些重复的工作可以由框架完成。

Web 框架的设计完全从使用者出发，让使用者编写尽可能少的代码。

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。

② @get 和 @post

先编写一个构造视图函数的装饰器@get('/path'): coroweb.py:

```
import functools
def get(path):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kw):
            return func(*args, **kw)
        wrapper.__method__ = 'GET'
        wrapper.__route__ = path
        return wrapper
    return decorator
```

一个函数通过@get()的装饰就附带了 URL 信息。

@post 与@get 定义类似。

利用偏函数统一 GET 和 POST 装饰器:

```
# 建立视图函数装饰器,用来存储、附带 URL 信息,GET 和 POST 统一为一个装饰器
def handler_decorator(path, *, method):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kw):
            return func(*args, **kw)
        wrapper.__method__ = method
        wrapper.__route__ = path
        return wrapper
    return decorator

# 偏函数, GET,POST 方法的路由装饰器
get = functools.partial(handler_decorator, method='GET')
post = functools.partial(handler_decorator, method='POST')
```

这样就可以直接通过装饰器, 将一个函数映射成视图函数。

```
@get('/')
def index(request):
    return 'hello!'
```

③ 定义 RequestHandler

视图函数仍无法从 request 中获取参数, 所以还要从 request 对象中提取视图函数所需的参数, 并且视图函数并非都是 coroutine。

需要定义一个能处理 request 请求的类来对视图函数进行封装。

RequestHandler 类, 分析视图函数所需的参数, 再从 request 对象中将参数提取, 调用视图函数, 并返回 web.Response 对象。这样就完全符合 aiohttp 框架的要求。由于其定义了__call__()方法, 其实例对象可以看作函数。

1) 解析视图函数

使用 Python 自带的 inspect 模块, 可以用来解析函数的参数。

20180329

插曲: inspect 模块的使用

① `inspect.signature(f)`: 返回一个 `inspect.Signature` 类型的对象, 值为函数 `f` 的所有参数

```
import inspect
def func(a, b=0, *c, d, e=1, **f):
    pass
sig = inspect.signature(func)
print(sig) # (a, b=0, *c, d, e=1, **f),函数 f 所有参数
print(type(sig)) # <class 'inspect.Signature'>
```

② `inspect.Signature` 对象的 `parameters` 属性是一个 `mappingproxy`(映射)类型的对象, 值为一个有序字典(`OrderedDict`)。字典里的 `key` 是 `str` 类型的参数名, `value` 是一个 `inspect.Parameter` 类型的对象。

`inspect.Parameter` 对象的 `kind` 属性是一个 `_ParameterKind` 枚举类型的对象, 值为这个参数的类型。

1) **POSITIONAL_OR_KEYWORD**: 位置或关键字参数, Python 最普通的参数类型, 可以通过位置或关键字传参数;

```
def f(a):
    print(a)
f(1) # 位置传参调用
f(a=1) # 关键字传参调用
```

2) **VAR_POSITIONAL**: 可变参数 `*args`, 位置参数的元组, 不能用关键字传参

```
def f(*args):
    print(args)
# 可以传入任意个位置参数调用,不传参数也可以,传入关键字参数报错
f() # ()
f(1, 'a', True) # (1, 'a', True)
f(a=1) # TypeError: f() got an unexpected keyword argument 'a'
```

3) **KEYWORD_ONLY**: 关键字参数, 在`*`或`*args` (**VAR_POSITIONAL**)后面的参数, 只能用关键字传参数, 因为位置参数被前面的`*args` 全部接收了;

```
def f(*, a): # VAR_POSITIONAL 不需要使用时,可以匿名化
    print(a)
# 只能关键字传参,位置传参报错
f(a=1) # 1
f(1) # TypeError: f() takes 0 positional arguments but 1 was given
```

4) **VAR_KEYWORD**: 可变关键字参数`**kwargs`, 字典形式, 此类型的参数只允许有一个, 只能在函数最后声名

```
def f(**kwargs):
    print(kwargs)
# 可以传入任意个关键字参数,不传也可以,传入位置参数报错
f() # {}
f(a=1, b='b', c=False, d=[1, 2]) # {'a': 1, 'b': 'b', 'c': False, 'd': [1, 2]}
f(1) # TypeError: f() takes 0 positional arguments but 1 was given
```

5) **POSITIONAL_ONLY**: 位置参数, 不重要, 历史遗留产物, 高版本 Python 无法使用此类参数, 推荐用 **VAR_POSITIONAL** 来代替。

默认参数

- 1) VAR 类型不允许设置默认参数
- 2) 默认参数靠后放
- 3) 默认参数不要设为可变类型(如 list、dict 等), 因为如果在函数内改变了默认参数, 下次再调用时就不再是默认值。

```
def f(a, lst=[]):
    lst.append(a)
    print(lst)
f(1) # [1]
f(2) # [1, 2]
f(3, ['a', 'b']) # ['a', 'b', 3]
f(4) # [1, 2, 4]
```

inspect.Parameter 对象的 default 属性: 如果这个参数有默认值, 返回这个默认值; 没有则返回一个 inspect._empty 类。

```
params = sig.parameters
print(params) # OrderedDict([('a', <parameter "a">), ('b', <parameter "b=0">), ('c',
<parameter "*c">), ('d', <parameter "d">), ('e', <parameter "e=1">), ('f', <parameter "**f">)])
print(type(params)) # <class 'mappingproxy'>
for k, v in params.items():
    print('k: {}, type is {}\nv: {}, type is {}'.format(k, type(k), v, type(v)))
    kind = v.kind
    print('kind is', kind, ', kind-type is', type(kind))
    default = v.default
    print('default is {}, default-type is {}'.format(default, type(default)))
print('*' * 50)
```

此处每个 k 都是字符串参数名, 类型为<class 'str'>; 每个 v 为 a,b=0,*c,d,e=1,**f, 类型都是<class 'inspect.Parameter'>;
a,b 的 kind 是 POSITIONAL_OR_KEYWORD, c 是 VAR_POSITIONAL, d,e 是 KEYWORD_ONLY, f 是 VAR_KEYWORD;
kind 类型都是<enum '_ParameterKind'>;
a,c,d,f 没有默认参数, default 是<class 'inspect._empty'>, 类型是<class 'type'>; b 默认 0, e 默认 1, 都是整数, 类型是<class 'int'>

coroweb.py 解析视图函数:

```
import inspect # 使用 inspect 模块, 检查视图函数的参数

def get_required_kw_args(f): # 获取无默认值的关键字参数
    args = []
    params = inspect.signature(f).parameters
    for name, param in params.items():
        # 如果视图函数存在关键字参数,且无默认值,获取它的参数名
        if param.kind == inspect.Parameter.KEYWORD_ONLY and param.default == inspect.Parameter.empty:
```

```

        args.append(name)
    return tuple(args)

def get_named_kw_args(f): # 获取关键字参数
    args = []
    params = inspect.signature(f).parameters
    for name, param in params.items():
        if param.kind == inspect.Parameter.KEYWORD_ONLY:
            args.append(name)
    return tuple(args)

def has_named_kw_arg(f): # 判断是否有关键字参数
    params = inspect.signature(f).parameters
    for name, param in params.items():
        if param.kind == inspect.Parameter.KEYWORD_ONLY:
            return True

def has_var_kw_arg(f): # 判断是否有可变关键词参数**kwargs
    params = inspect.signature(f).parameters
    for name, param in params.items():
        if param.kind == inspect.Parameter.VAR_KEYWORD:
            return True

def has_request_arg(f): # 判断是否有名叫 request 的参数,且位置在最后
    sig = inspect.signature(f)
    params = sig.parameters
    found = False
    for name, param in params.items():
        if name == 'request':
            found = True
            continue
    if found and (
        param.kind != inspect.Parameter.VAR_POSITIONAL and
        param.kind != inspect.Parameter.KEYWORD_ONLY and
        param.kind != inspect.Parameter.VAR_KEYWORD):
        # param 是普通参数,但 param 位于 request 之后,即 request 位置不在最后,报错
        raise ValueError('request parameter must be the last named parameter in
function:{}'.format(f.__name__, sig))
    return found

```

2) 提取 request 中的参数

request 是经 aiohttp 封装后的对象，其本质是一个 HTTP 请求。
request 由请求状态(status)、请求头(header)、请求体(body)三部分组成。
需要的参数包含在 body 和 status 的 URI 中。

RequestHandler 需要处理以下问题:

1. 确定 HTTP 请求的方法是 **GET** 还是 **POST**(用 request.method 获取);
2. 根据 HTTP 请求的 content_type 字段(用 request.content_type 获取), 选用不同解析方法获取参数;
3. 将获取的参数经处理使其完全符合视图函数接收的参数形式;
4. 调用视图函数。

coroweb.py 的 RequestHandler 类:

```
from aiohttp import web
from urllib.parse import parse_qs
import logging

# 定义 RequestHandler 类,从视图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
# 视图函数,把结果转换为 web.Response 对象,符合 aiohttp 框架要求
class RequestHandler(object):
    def __init__(self, app, f):
        self.app = app
        self.func = f
        self.required_kw_args = get_required_kw_args(f)
        self.named_kw_args = get_named_kw_args(f)
        self.has_request_arg = has_request_arg(f)
        self.has_named_kw_arg = has_named_kw_arg(f)
        self.has_var_kw_arg = has_var_kw_arg(f)

    async def __call__(self, request):
        kw = None # ① 定义 kw , 用于保存 request 中参数
        # ② 判断视图函数是否存在关键字参数,如果存在根据 GET 或 POST 方法将 request 请求内
        # 容保存到 kw
        if self.has_named_kw_arg or self.has_var_kw_arg: # 若视图函数有关键字参数
            if request.method == 'POST':
                # 根据 request 参数中的 content_type 使用不同解析方法
                if not request.content_type: # 如果 content_type 不存在返回 400 错误
                    return web.HTTPBadRequest(text='Missing Content_Type.')
                ct = request.content_type.lower() # 统一小写,便于检查
                if ct.startswith('application/json'): # json 格式数据
                    params = await request.json() # 仅解析 body 字段的 json 数据
                    if not isinstance(params, dict):
                        return web.HTTPBadRequest(text='JSON body must be object.')
                    kw = params
                # form 表单请求的编码形式
                elif ct.startswith('application/x-www-form-urlencoded') or
                ct.startswith('multipart/form-data'):
                    params = await request.post() # 返回 post 中解析后的数据,dict-like 对象
                    kw = dict(**params) # 组成 dict,统一 kw 格式
```



```

        else:
            return web.HTTPBadRequest(text='Unsupported Content-Type:
{}'.format(request.content_type))
        if request.method == 'GET':
            qs = request.query_string # url 查询字符串
            if qs:
                kw = dict()
                """
                >>> qs='a=1&b=2&b=3&c=haha'
                >>> parse_qs(qs,True)
                {'a': ['1'], 'b': ['2', '3'], 'c': ['haha']}
                """
                # 值 v 是一个 list,第 2 个参数 keep_blank_values 为 True 表示不忽略空格
                for k, v in parse_qs(qs, True).items(): # 返回查询字符串键值对,dict 对象
                    kw[k] = v[0]
            # ③ 如果 kw 为空,说明 request 无请求内容,则将 match_info 里的资源映射给 kw
            if kw is None:
                # request.match_info 返回 dict 对象,键为可变路由中可变字段(variable)的参数名,
                # 值为传入 request 请求的 path 的对应值,比如路由为/user/{name},请求 path 为
                # /user/hikari,匹配路由,则 request.match_info 返回{'name':'hikari'}
                kw = dict(**request.match_info)
            else:
                if self._has_named_kw_arg and (not self._has_var_kw_arg): # 若视图函数只有命名
                    关键字参数没有可变关键词参数
                    tmp = dict()
                    for name in self._named_kw_args:
                        if name in kw:
                            tmp[name] = kw[name]
                    kw = tmp # 只保留命名关键字参数
                # 将 request.match_info 中的参数传入 kw
                for k, v in request.match_info.items():
                    # 检查 kw 中的参数是否和 match_info 中的重复
                    if k in kw: # 貌似和 if k in kw.keys() 一样
                        logging.warning('Duplicate arg name in named arg and kw args:
{}'.format(k))
                    kw[k] = v
            # ④ 善后工作
            if self._has_request_arg: # 视图函数存在 request 参数
                kw['request'] = request
            if self._required_kw_args: # 视图函数存在无默认值的关键字参数
                for name in self._required_kw_args:
                    if name not in kw: # 若未传入必须关键字参数值,报错
                        return web.HTTPBadRequest(text='Missing argument: {}'.format(name))
            # 至此 kw 为视图函数 f 真正能调用的参数

```

```
# 也就是 request 请求中的参数终于传递给了视图函数
logging.info('call with args: {}'.format(str(kw)))
return await self._func(**kw)
```

④ 编写注册函数(添加路由)

1) 视图函数注册函数 add_route(): coroweb.py:

```
import asyncio
def add_route(app, f): # 注册一个视图函数
    method = getattr(f, '__method__', None)
    path = getattr(f, '__route__', None)
    # 验证视图函数是否有 method 和 path 参数
    if method is None or path is None:
        raise ValueError('@get or @post not defined in {}'.format(f.__name__))
    # 判断视图函数是否协程并且是生成器
    if not asyncio.iscoroutinefunction(f) and not inspect.isgeneratorfunction(f):
        f = asyncio.coroutine(f) # 将视图函数转为协程
    logging.info('add route {} {} --> {}'.format(method, path, f.__name__,
                                                    ', '.join(inspect.signature(f).parameters.keys())))
    # 在 app 中注册经 RequestHandler 类封装的视图函数
    app.router.add_route(method, path, RequestHandler(app, f))
```

2) 批量注册视图函数

add_route()每次只能注册一个视图函数;批注册函数 add_routes()实现只提供模块路径,自动导入其中的视图函数进行注册。

```
# 导入模块,批量注册视图函数
def add_routes(app, module_name):
    n = module_name.rfind('.') # 从右侧检索返回索引
    if n == -1:
        # __import__ 作用同 import 语句,但__import__是一个函数,参数为模块的字符串名字
        # __import__('urllib',globals(),locals(),['request'],0)等价于 from urllib import request
        mod = __import__(module_name, globals(), locals())
    else:
        # 比如'urllib.request', name='request', 获取 urllib 模块对象的 request 属性得到
        # urllib.request 模块对象
        name = module_name[(n + 1):]
        # 获取对应的模块对象
        mod = getattr(__import__(module_name[:n], globals(), locals(), [name], 0), name)
    # dir()获取模块所有类、实例、函数等对象的 str 形式
    for attr in dir(mod):
        if attr.startswith('_'):
            continue # 忽略 '_'开头的对象
        f = getattr(mod, attr)
        if callable(f): # f 可以被调用
            # 确保视图函数存在 method 和 path
            method = getattr(f, '__method__', None)
```

```

path = getattr(f, '__route__', None)
if method and path:
    add_route(app, f) # 注册视图函数

```

3) 静态文件注册函数

add_static()用于注册静态文件，只提供文件路径即可进行注册

```

import os

# 添加静态文件, 如 image, css, js 等文件
def add_static(app):
    # 获取本文件绝对路径-->获取根目录-->拼接同目录的 static 目录
    # path = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'static')
    # 上面太麻烦了, abspath('.')可以直接获取当前文件根目录的绝对路径
    path = os.path.join(os.path.abspath('.'), 'static')
    app.router.add_static('/static/', path) # 注册静态文件
    logging.info('add static {} --> {}'.format('/static/', path))

```

⑤ 初始化 jinja2 模板

使用 jinja2 作为模板引擎，在自己写的框架中对 jinja2 模板进行初始化设置。

app.py:

```

import logging; logging.basicConfig(level=logging.INFO)
import os
from jinja2 import Environment, FileSystemLoader

def init_jinja2(app, **kw):
    logging.info('init jinja2...')
    # ① 配置 options 参数,字典形式,是 Environment(**options)的参数
    options = dict(
        # 自动转义 xml/html 的特殊字符
        autoescape=kw.get('autoescape', True),
        # 代码块的开始、结束标志
        block_start_string=kw.get('block_start_string', '{%'),
        block_end_string=kw.get('block_end_string', '%}'),
        # 变量的开始、结束标志
        variable_start_string=kw.get('variable_start_string', '{{'),
        variable_end_string=kw.get('variable_end_string', '}}'),
        # 自动加载修改后的模板文件
        auto_reload=kw.get('auto_reload', True)
    )
    path = kw.get('path', None) # 获取模板文件目录路径
    if not path:
        # path = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'templates')
        path = os.path.join(os.path.abspath('.'), 'templates')
    # Environment 类是 jinja2 的核心类,用来保存配置、全局对象、模板文件的路径
    # ② 模板加载器 FileSystemLoader 类加载 path 路径中的模板文件

```

```

env = Environment(loader=FileSystemLoader(path), **options)
# ③ 创建 Environment 对象, 添加过滤器
ft = kw.get('filters', None)
if ft:
    for name, f in ft.items():
        env.filters[name] = f # filters 是 Environment 类的属性, 过滤器字典
# jinja2 的环境配置都在 env 对象中, 把 env 对象添加到 app 类字典对象中,
# 这样 app 就知道模板的路径和解析模板的方法
app['__template__'] = env # app 是一个 dict-like 对象

```

编写一个过滤器:

```

import time
from datetime import datetime

def datetime_filter(t): # 模板语言的过滤器, 用于时间戳转为字符串, 显示博客发表时间
    delta = int(time.time() - t)
    if delta < 0: # 所以不可能未来发表的...此句注释吧
        return u'未来的某天'
    if delta < 60:
        return u'1 分钟前'
    if delta < 3600:
        return u'%s 分钟前' % (delta // 60)
    if delta < 86400:
        return u'%s 小时前' % (delta // 3600)
    if delta < 604800:
        return u'%s 天前' % (delta // 86400)
    dt = datetime.fromtimestamp(t)
    return u'%s 年%s 月%s 日' % (dt.year, dt.month, dt.day)

```

⑥ 编写 middleware

middleware 是符合 WSGI 定义的中间件, 位于服务端和客户端之间对数据进行拦截处理的一个桥梁。可以看作服务器端的数据, 经 middleware 一层层封装, 最终传递给客户端。

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。

web 框架正是由一层层 middleware 的封装, 才具备各种完善的功能。

app.py:

```

# 编写一个用于打印日志的 middleware, 和装饰器类似
async def logger_factory(app, handler): # handler 是视图函数
    async def logger(request):
        logging.info('Request: {} {}'.format(request.method, request.path))
        return await handler(request)
    return logger

```

```

# 打印日志的中间件, 打印 POST 请求 json 或表单数据
async def data_factory(app, handler):
    async def parse_data(request):
        if request.method == 'POST':
            if request.content_type.startswith('application/json'):
                request.__data__ = await request.json()
                logging.info('request json: {}'.format(request.__data__))
            elif request.content_type.startswith('application/x-www-form-urlencoded'):
                request.__data__ = await request.post()
                logging.info('request form: {}'.format(request.__data__))
        return await handler(request)
    return parse_data

from aiohttp import web
import json
# response 中间件把返回值转换为 web.Response 对象再返回, 保证满足 aiohttp 的要求
async def response_factory(app, handler):
    async def response(request):
        logging.info('Response handler...')
        res = await handler(request)
        if isinstance(res, web.StreamResponse): # StreamResponse 是所有 Response 的父类
            return res
        if isinstance(res, bytes):
            # Response 继承于 StreamResponse, 接收 body 参数, 构造 HTTP 响应内容
            res = web.Response(body=res)
            res.content_type = 'application/octet-stream'
            return res
        if isinstance(res, str):
            if res.startswith('redirect:'): # 若返回重定向字符串, 重定向至目标 url
                return web.HTTPFound(res[9:])
            res = web.Response(body=res.encode('utf-8'))
            res.content_type = 'text/html;charset=utf-8' # utf-8 编码的 text 格式
            return res
        if isinstance(res, dict):
            # 视图函数返回值会带有 __template__ 值, 表示选择渲染的模板
            template = res.get('__template__')
            if template is None: # 不带模板信息返回 json 对象
                res = web.Response(
                    body=json.dumps(res, ensure_ascii=False, default=lambda obj:
obj.__dict__).encode('utf-8'))
                res.content_type = 'application/json;charset=utf-8'
                return res
            else: # 带模板信息, 渲染模板
                # 获取已初始化的 Environment 对象, 调用 get_template() 返回 Template 对象;

```

调用 Template 对象的 render()方法, 传入 res 渲染模板, 返回 unicode 格式字符串,用 utf-8 编码

```

    tpl = app['__template__'].get_template(template)
    res = web.Response(body=tpl.render(**res).encode('utf-8'))
    res.content_type = 'text/html;charset=utf-8'
    return res

if isinstance(res, int) and 100 <= res < 600:
    return web.Response(status=res) # 返回响应码
if isinstance(res, tuple) and len(res) == 2:
    code, msg = res # 返回了响应码和原因的元组, 如(200, 'OK'), (404, 'Not Found')
    if isinstance(code, int) and 100 <= code < 600:
        return web.Response(status=code, text=msg)
# 均以上条件不满足, 默认返回
res = web.Response(body=str(res).encode('utf-8'))
res.content_type = 'text/plain;charset=utf-8'
return res
return response

```

⑦ 测试: www/handlers.py 存放视图函数, 暂时不涉及数据库

```

from coroweb import get

@get('/')
async def index(request):
    return '<h1 style="color:red">hikari\'s website</h1>'

@get('/hello')
async def hello(request):
    return '<h1>hello!</h1>'

@get('/hello/{name}')
async def hello2(name, request):
    return '<h1>hello! {}</h1>'.format(name)

```

app.py:

```




import asyncio
from coroweb import add_routes, add_static

async def init(loop):
    app = web.Application(loop=loop, middlewares=[logger_factory, response_factory])
    init_jinja2(app, filters=dict(datetime=datetime_filter))
    add_routes(app, 'handlers') # 将 handlers.py 的所有视图函数添加路由
    add_static(app) # 需要创建 static 目录
    server = await loop.create_server(app.make_handler(), 'localhost', 8000)
    logging.info('server started at http://127.0.0.1:8000...')
    return server

```

```
if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(init(loop))
    loop.run_forever()
```

居然可以执行...

 127.0.0.1:8000
  127.0.0.1:8000/hello
  127.0.0.1:8000/hello/maki

hikari's website hello! hello! maki

输入 `http://127.0.0.1:8000/hello/maki` 打印的日志:

```
INFO:root:init jinja2...
INFO:root:add route GET /hello --> hello(request)
INFO:root:add route GET /hello/{name} --> hello2(name,request)
INFO:root:add route GET / --> index(request)
INFO:root:add static /static/ --> D:\hikari_web_day5\www\static
INFO:root:server started at http://127.0.0.1:8000...
INFO:root:Request: GET /hello/maki
INFO:root:Response handler...
INFO:root:call with args: {'name': 'maki', 'request': <Request GET /hello/maki >}
INFO:aiohttp.access:127.0.0.1 [30/Mar/2018:02:59:32 +0000] "GET /hello/maki HTTP/1.1" 200
169 "-" "Mozilla/5.0 (Windows NT 10.0; WOW64) ..."
```

瞎猜上面响应过程:

- ① 创建 `app` 对象, 附带中间件; `jinja2` 初始化, `app['__template__']`;
- ② `@get@post` 装饰器将 `handler.py` 的函数附加 `__method__` 和 `__route__` 属性, 附带 URL 信息, 变为视图函数;
- ③ `RequestHandler` 装饰 `handler.py` 的视图函数, 用于提取 `request` 参数;
- ④ `add_routes` 在 `app` 中注册视图函数; 注册静态路由;
- ⑤ 开启服务器; 浏览器输入 `http://127.0.0.1:8000/hello/maki`, 服务器收到请求;
- ⑥ 中间件 `response_factory` 收到请求, 根据路由发给相应的 `handler` (`RequestHandler` 装饰的视图函数)处理, 也就是执行其对象的 `__call__()` 方法, 发现 `request` 没有关键字参数、没有查询字符串, 匹配 `url` 得到 `kw = {'name': 'maki'}`, 对应视图函数为 `@get` 装饰的 `hello2()` 函数, 调用返回 `<h1>hello! maki</h1>`;
- ⑦ 请求向装饰器内部传递, 相反响应向装饰器外面传递, 经中间件 `response_factory` 对字符串响应处理, 构造真正的 `web.Response` 返回浏览器。

20180330

🌸 Day 6. 编写配置文件

通常一个 Web App 在运行时都需要读取配置文件, 比如数据库的用户名、密码等。在不同的环境中运行时, Web App 可以通过读取不同的配置文件来获得正确的配置。

可以直接用 Python 源代码实现配置; 默认的配置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。

创建一个默认配置文件 `config_default.py`:

```

configs = {
    'debug': True,
    'db': {
        'host': 'localhost',
        'port': 3306,
        'user': 'root',
        'password': 'mysql',
        'db': 'myblog',
    },
    'session': {
        'secret': 'hiKari'
    }
}

```

如果要部署到服务器，通常需要修改数据库的 host 等信息，直接修改 config_default.py 不是好方法，更好的方法是编写一个 config_override.py 用来覆盖某些默认设置：

```

# 覆盖默认设置
configs = {
    'db': {
        'host': '192.168.1.101',
        'user': 'hikari',
    }
}

```

config_default.py 作为开发环境的标准配置，config_override.py 作为生产环境的标准配置，既能方便在本地开发，又能随时部署到服务器上。

应用程序读取配置文件要优先从 config_override.py 读取。

编写 config.py 用以整合配置文件：

```

import config_default

class Dict(dict):
    def __init__(self, names=(), values=(), **kw):
        super().__init__(**kw)
        for k, v in zip(names, values):
            self[k] = v # names 每个元素为键, values 对应位置元素为值
    # 可以 dct.key 或 dct[key] 获取或设置属性
    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Dict' object has no attribute '%s'" % key)
    def __setattr__(self, key, value):
        self[key] = value

```



```

# 合并设置
def merge(defaults, override):
    ret = {}
    for k, v in defaults.items():
        if k in override:
            if isinstance(v, dict):
                ret[k] = merge(v, override[k]) # 如果 v 是 dict, 递归
            else:
                ret[k] = override[k] # k 有新值, 覆盖默认值
        else:
            ret[k] = v # override 没有 k, 使用默认值
    return ret

def to_dict(configs):
    d = Dict()
    for k, v in configs.items():
        d[k] = to_dict(v) if isinstance(v, dict) else v # 如果 v 是 dict, 递归
    return d

configs = config_default.configs
try:
    import config_override
    configs = merge(configs, config_override.configs) # 获得合并的 configsdict, dict 类型
except ImportError:
    pass

configs = to_dict(configs) # 将 dict 类型的 configs 变为 Dict 类的实例, 可以通过 configs.k 直接
                           获取 v, 增加易用性, 不是必需
if __name__ == '__main__':
    print(configs.db.user) # hikari
    print(configs['db']['host']) # 192.168.1.101
    print(configs.db['port']) # 3306

```

🌸 Day 7. 编写 MVC

Controller: 控制业务逻辑, 决定与前端进行数据交互的形式和方法。如检查数据, 存取数据等。

View: 负责显示页面。通过接收 Controller 传来的数据, 渲染后呈现给用户。

Model: 在后端 Controller 和前端 View 之间被传递的数据。

在本项目中: Model 是之前编写的 orm 框架: 它从 MySQL 中取出数据, 并以对象的形式被传递; View 是 jinja2 模板引擎, 能接收从后端传来的数据, 渲染呈现出页面; Controller 是视图函数, 以及用来封装视图函数的 HandRequest 类和 middleware 等。

① 修改之前 handlers.py 的视图函数:

```
from coroweb import get
```

```

from models import User

@get('/')
async def index(request):
    users = await User.find_all()
    return { # 视图函数返回值是 dict
        '__template__': 'test.html', # 在 response_factory 中会搜索模板
        'users': users # 传入模板的数据
    }

```

② www/templates/目录下编写一个 jinja2 模板 test.html:

```

<style type="text/css">
    ul { list-style: none; font: 24px/36px "Microsoft YaHei";}
    .users { font: 36px/36px "Microsoft YaHei"; color: red;}
    .idx { color: #ff00ff;}
</style>

```

```

<h1 class="users">All users</h1>
<ul>
    {% for u in users %}
        <li><span class="idx">{{ loop.index }}. </span>{{ u.name }} / {{ u.email }}</li>
    {% endfor %}</ul>

```

③ 修改 aap.py 的 init, 增加访问数据库

```

import asyncio
from coroweb import add_routes, add_static
import orm
from config import configs

async def init(loop):
    await orm.create_pool(loop, **configs.db) # 从配置文件获取数据库信息,创建连接池
    app = web.Application(loop=loop, middlewares=[logger_factory, response_factory])
    init_jinja2(app, filters=dict(datetime=datetime_filter))
    add_routes(app, 'handlers') # 将 handlers 的视图函数添加路由
    add_static(app) # 需要创建 static 目录
    server = await loop.create_server(app.make_handler(), 'localhost', 8000)
    logging.info('server started at http://127.0.0.1:8000...')
    return server

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(init(loop))
    loop.run_forever()

```

用之前的 test.py 向 MySQL 插入几条数据;

运行 app.py, 浏览器输入 127.0.0.1:8000:

< > ↻ ☆ 127.0.0.1:8000

All users

1. hikari / hikari@example.com
2. maki / maki@example.com
3. rin / rin@example.com

🌸 Day 8. 构建前端

之前是一个最简单的 MVC, 但页面效果肯定不会让人满意。

对于复杂的 HTML 前端页面, 需要一套基础的 CSS 框架来完成页面布局和基本样式; jQuery 作为操作 DOM 的 JS 库也必不可少。

uikit 是一个强大的 CSS 框架, 具备完善的响应式布局、漂亮的 UI 和丰富的 HTML 组件, 能轻松设计出美观而简洁的页面。官网下载。
所有静态文件统一放到 `www/static/` 目录下, 并按照类别归类。

由于前端页面肯定不止首页一个页面, 每个页面都有相同的页眉和页脚。如果每个页面都是独立的 HTML 模板, 那么修改页眉和页脚就需要把每个模板都改一遍, 效率太低。

常见的模板引擎已经考虑到了页面上重复的 HTML 部分的复用问题。有的模板通过 `include` 把页面拆成三部分:

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

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的维护。

jinjia2 的模板继承使模板的复用更简单。见 201803.pdf~7

① 用 uikit 编写父模板 `base.html`:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8"/>
  {% block meta %}{# 挖坑 #}{% endblock %}
  <title>{% block title %} ? {% endblock %} - hikari webapp</title>
  {# 代码和静态文件都是拷过来的 #}
  <link rel="stylesheet" href="/static/css/uikit.min.css">
  <link rel="stylesheet" href="/static/css/uikit.gradient.min.css">
  <link rel="stylesheet" href="/static/css/myblog.css"/>
  <script src="/static/js/jquery.min.js"></script>
  <script src="/static/js/sha1.min.js"></script>
```

```

<script src="/static/js/uikit.min.js"></script>
<script src="/static/js/sticky.min.js"></script>
<script src="/static/js/vue.min.js"></script>
<script src="/static/js/myblog.js"></script>
{% block beforehead %}{# 此处一般是 css 或 js #}{% endblock %}
</head>
<body>
{# 导航条 #}
<nav class="uk-navbar uk-navbar-attached uk-margin-bottom">
  <div class="uk-container uk-container-center">
    <a href="/" class="uk-navbar-brand">hikari blog</a>
    <ul class="uk-navbar-nav">
      {# 各种链接页面都没写 #}
      <li data-url="blogs"><a href="/"><i class="uk-icon-home"></i> 日志</a></li>
      <li><a target="_blank" href="#"><i class="uk-icon-book"></i> 教程</a></li>
      <li><a target="_blank" href="#"><i class="uk-icon-code"></i> 源码</a></li>
    </ul>
    <div class="uk-navbar-flip">
      <ul class="uk-navbar-nav">
        {% if __user__ %}
          <li class="uk-parent" data-uk-dropdown>
            <a href="#0"><i class="uk-icon-user"></i> {{ __user__.name }}</a>
            <div class="uk-dropdown uk-dropdown-navbar">
              <ul class="uk-nav uk-nav-navbar">
                <li><a href="/signout"><i class="uk-icon-sign-out"></i>
              登出</a></li>
              </ul></div></li>
            {% else %}
              <li><a href="/signin"><i class="uk-icon-sign-in"></i> 登录</a></li>
              <li><a href="/register"><i class="uk-icon-edit"></i> 注册</a></li>
            {% endif %}
          </ul></div></div></nav>
{# 正文部分,每个页面都不同,先挖坑 #}
<div class="uk-container uk-container-center">
  <div class="uk-grid">
    {% block content %}{% endblock %}</div></div>
{# 底部 #}
<div class="uk-margin-large-top" style="background-color:#eee; border-top:1px solid #ccc;">
  <div class="uk-container uk-container-center uk-text-center">
    <div class="uk-panel uk-margin-top uk-margin-bottom">
      <p>{# 微博、github、领英、twitter 链接, 图标都是 uikit 里有的 #}
      <a target="_blank" href="#" class="uk-icon-button uk-icon-weibo"></a>
      <a target="_blank" href="https://github.com/hoshizorahikari" class="uk-

```

```

icon-button uk-icon-github"></a>
        <a target="_blank" href="#" class="uk-icon-button uk-icon-linkedin-square"></a>
        <a target="_blank" href="#" class="uk-icon-button uk-icon-twitter"></a>
    </p>
    <p>Powered by <a href="#">hikari webapp</a>. Copyright &copy; 2018. [<a href="/manage/" target="_blank">Manage</a>]
    </p>
    <p><a href="http://www.liaoxuefeng.com/" target="_blank">www.liaoxuefeng.com</a>. All rights reserved.</p>
        <a target="_blank" href="http://www.w3.org/TR/html5/"><i class="uk-icon-html5" style="font-size:64px; color: #444;"></i></a></div></div></div>
</body>
</html>

```

② 首页 myblog.html 继承于 base.html:

```

{% extends 'base.html' %}
{% block title %}日志{% endblock %}
{% block beforehead %}<script></script>{% endblock %}
{% block content %}
    <div class="uk-width-medium-3-4">{# 栅格化?左边占 3/4, 右边 1/4? #}
        {% for blog in blogs %}
            <article class="uk-article">
                <h2><a href="/blog/{{ blog.id }}">{{ blog.name }}</a></h2>
                {# 此处 datetime 是一个 filter(过滤器),就是之前写的那个 #}
                <p class="uk-article-meta">发表于{{ blog.created_at|datetime }}</p>
                <p>{{ blog.summary }}</p>
                <p><a href="/blog/{{ blog.id }}">继续阅读 <i class="uk-icon-angle-double-right"></i></a></p>
            </article>
            <hr class="uk-article-divider">
        {% endfor %}</div>

    <div class="uk-width-medium-1-4">
        <div class="uk-panel uk-panel-header">
            <h3 class="uk-panel-title">友情链接</h3>
            <ul class="uk-list uk-list-line">
                <li><i class="uk-icon-thumbs-o-up"></i> <a target="_blank" href="#">编程</a></li>
                <li><i class="uk-icon-thumbs-o-up"></i> <a target="_blank" href="#">读书</a></li>
                <li><i class="uk-icon-thumbs-o-up"></i> <a target="_blank" href="#">Python 教程</a></li>
                <li><i class="uk-icon-thumbs-o-up"></i> <a target="_blank" href="#">Git

```

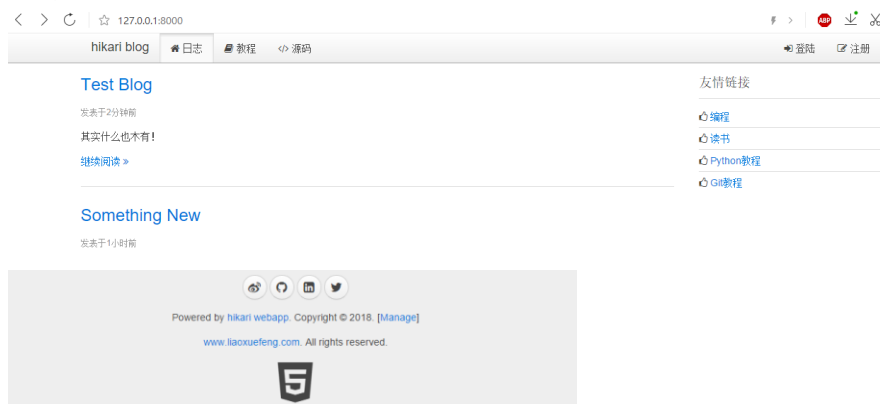
```
教程</a></li>
    </ul></div></div>
{% endblock %}
```

③ handler.py 的首页视图函数:

```
from coroweb import get
from models import User, Blog
import time

@get('/')
def index(request):
    summary = '其实什么也没有!'
    blogs = [
        Blog(id='1', name='Test Blog', summary=summary, created_at=time.time() - 120),
        Blog(id='2', name='Something New', summary=summary, created_at=time.time() - 3600),
        Blog(id='3', name='Learn Swift', summary=summary, created_at=time.time() - 7200)
    ]
    return {
        '__template__': 'myblog.html',
        'blogs': blogs
    }
```

结果:



20180331

🔧 Day 9. 编写 API

REST(Representational State Transfer, 表述性状态传递)是 Roy Fielding 博士在 2000 年其博士论文中提出的一种软件架构风格。

因为 REST 模式的 Web 服务与复杂的 SOAP 和 XML-RPC 相比更加简洁, 越来越多的 web 服务开始采用 REST 风格设计和实现, 成为 Web API 的标准。

什么是 Web API?

如果想要获取一篇 Blog, 输入 `http://localhost:8000/blog/123`, 就可以看到 id 为 123 的 Blog 页面; 但这个结果是 HTML 页面, 它同时混合包含了 Blog 的数据和 Blog 的展示两个部分。对于用户来说, 阅读起来没有问题; 但是对于机器,

就很难从 HTML 中解析出 Blog 的数据。

如果一个 URL 返回的不是 HTML，而是机器能直接解析的数据，这个 URL 就可以看成是一个 Web API。比如，读取 `http://localhost:8000/api/blogs/123`，如果能直接返回 Blog 的数据，那么机器就可以直接读取。

REST 是一种设计 API 的模式。最常用的数据格式是 JSON。由于 JSON 能直接被 JS 读取，所以 JSON 格式编写的 REST 风格的 API 简单、易读、易用。

由于 API 把 Web App 的功能全部封装，通过 API 操作数据，可以极大地把前端和后端的代码分离，使后端代码易于测试，前端代码编写更简单。

一个 API 也是视图函数，希望能直接通过 `@api` 装饰器把函数变成 JSON 格式的 REST API。获取注册用户可以用一个 API 实现如下：

handlers.py 添加：

```
@get('/api/users')
async def api_get_users(): # 此处貌似要用协程额, 不用报错...
    users = await User.find_all(order_by='created_at desc') # 按创建时间降序
    for u in users:
        u.pwd = '*****' # 将查询到的用户密码隐藏
    return dict(users=users)
```

只要返回一个 dict，中间件就可以把结果序列化为 JSON 并返回。

因此定义一个 APIError 处理 API 调用时发生了逻辑错误(比如用户不存在)，其他的 Error 视为 Bug，返回的错误代码为 `internalerror`。

www/apis.py:

```
class APIError(Exception):
    # APIError 父类
    def __init__(self, error, data='', message=''):
        super(APIError, self).__init__(message)
        self.error = error
        self.data = data
        self.message = message

class APIValueError(APIError):
    # 输出值错误或无效
    def __init__(self, field, message=''):
        super(APIValueError, self).__init__('value : invalid', field, message)

class APIResourceNotFoundError(APIError):
    # 资源没有找到
    def __init__(self, field, message=''):
        super(APIResourceNotFoundError, self).__init__('value : not found', field, message)

class APIPermissionError(APIError):
    # api 没有权限
    def __init__(self, message=''):
```

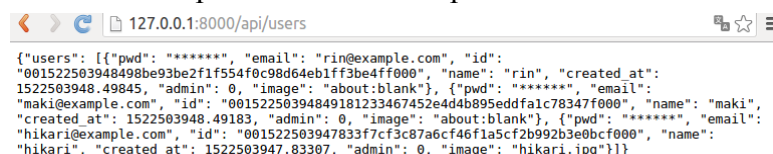
```
super(APIPermissionError, self).__init__('permission : forbidden', 'permission',
message)
```

修改 coroweb.py 的 RequestHandler 类的 `__call__()` 方法:

```
from apis import APIError
class RequestHandler(object):
    async def __call__(self, request):
        # 之前一样, 在最后添加 try
        try:
            return await self._func(**kw)
        except APIError as e:
            return dict(error=e.error, data=e.data, message=e.message)
```

该项目第一次 ubuntu 测试:

- ① 创建虚拟环境: `mkvirtualenv -p /usr/bin/python3.5 hikari_blog`
- ② 将需要的模块安装; 貌似由于 python3.5 的虚拟环境, 直接 `pip` 就可以; 用 `pip3` 反而装到全局了;
- ③ `python test.py` 向 MySQL 插入数据;
- ④ `python app.py` 运行服务器;
- ⑤ 浏览器输入 `http://localhost:8000/api/users` 测试 API:



```
{
  "users": [
    {
      "pwd": "*****",
      "email": "rin@example.com",
      "id": "001522503948498be93be2f1f554f0c98d64eb1ff3be4ff000",
      "name": "rin",
      "created_at": "1522503948.49845",
      "admin": 0,
      "image": "about:blank"
    },
    {
      "pwd": "*****",
      "email": "maki@example.com",
      "id": "00152250394849181233467452e4d4b895eddfalc78347f000",
      "name": "maki",
      "created_at": "1522503948.49183",
      "admin": 0,
      "image": "about:blank"
    },
    {
      "pwd": "*****",
      "email": "hikari@example.com",
      "id": "001522503947833f7cf3c87a6cf46f1a5cf2b992b3e0bcf000",
      "name": "hikari",
      "created_at": "1522503947.83307",
      "admin": 0,
      "image": "hikari.jpg"
    }
  ]
}
```

这算成功了? 之前的 blog 首页也显示木有问题!

- ⑥ `deactivate` 退出虚拟环境

客户端调用 API 时, 必须通过错误代码来区分 API 调用是否成功。错误代码用来告诉调用者出错的原因。很多 API 用一个整数表示错误码, 需要查表得知错误信息。更好的方式是用字符串表示错误代码, 不需要看文档也能猜到错误原因。

20180402

🌸 Day 10. 用户注册和登录

用户注册相对简单:

- ① 注册模板: `templates/register.html`:

```
{% extends 'base.html' %}
{% block title %}注册{% endblock %}
{% block beforehead %}
    <script>
        function validateEmail(email) {
            var re = /^[^w+@\w+(\.[a-z]{2,3}){1,2}]$/;
            return re.test(email.toLowerCase());
        }
        $(function () {
            var vm = new Vue({
```



```

        el: '#vm',
        data: {
            name: '',
            email: '',
            password1: '',
            password2: ''
        },
        methods: {
            submit: function (event) {
                event.preventDefault();
                var $form = $('#vm');
                if (!this.name.trim()) {
                    return $form.showFormError('请输入名字');
                }
                if (!validateEmail(this.email.trim().toLowerCase())) {
                    return $form.showFormError('请输入正确的 Email 地址');
                }
                if (this.password1.length < 6) {
                    return $form.showFormError('密码长度至少为 6 个字符');
                }
                if (this.password1 !== this.password2) {
                    return $form.showFormError('两次输入的密码不一致');
                }
                var email = this.email.trim().toLowerCase();
                $form.postJSON('/api/users', {
                    name: this.name.trim(),
                    email: email,
                    pwd: CryptoJS.SHA1(email + ':' + this.password1).toString()
                }, function (err, r) {
                    if (err) {
                        return $form.showFormError(err);
                    }
                    return location.assign('/'); {# 注册成功返回主页? #}
                });
            }
        }
    });
    $('#vm').show();
});
</script>
{% endblock %}

{% block content %}
<div class="uk-width-2-3">

```

```

<h1>欢迎注册！</h1>
<form id="vm" v-on="submit: submit" class="uk-form uk-form-stacked">
  <div class="uk-alert uk-alert-danger uk-hidden"></div>
  <div class="uk-form-row">
    <label class="uk-form-label">名字:</label>
    <div class="uk-form-controls">
      <input v-model="name" type="text" maxlength="50"
placeholder="your name" class="uk-width-1-1"></div></div>
  <div class="uk-form-row">
    <label class="uk-form-label">电子邮件:</label>
    <div class="uk-form-controls">
      <input v-model="email" type="text" maxlength="50"
placeholder="your-name@example.com" class="uk-width-1-1"></div></div>
  <div class="uk-form-row">
    <label class="uk-form-label">输入密码:</label>
    <div class="uk-form-controls">
      <input v-model="password1" type="password" maxlength="50"
placeholder="输入密码" class="uk-width-1-1"></div></div>
  <div class="uk-form-row">
    <label class="uk-form-label">确认密码:</label>
    <div class="uk-form-controls">
      <input v-model="password2" type="password" maxlength="50"
placeholder="确认密码" class="uk-width-1-1"></div></div>
  <div class="uk-form-row">
    <button type="submit" class="uk-button uk-button-primary"><i class="uk-
icon-user"></i> 注册</button></div></form></div>
{% endblock %}

```

② handlers.py 用户注册视图函数：

```

from coroweb import get, post
from models import User, Blog, Comment, next_id
from apis import APIValueError, APIError

@get('/register')
def register():
    return {
        '__template__': 'register.html'
    }

_re_email = r'^\w+@\w+(\.[a-z]{2,3}){1,2}$'
_re_sha1 = r'^[0-9a-f]{40}$'

@post('/api/users')
async def api_register_user(*, email, name, pwd):
    # 用户注册 api

```

```

name = name.strip()
if not name:
    raise APIValueError('name')
if not email or not re.match(_re_email, email):
    raise APIValueError('email')
if not pwd or not re.match(_re_sha1, pwd):
    raise APIValueError('password')
users = await User.find_all('email=?', [email]) # 查询邮箱是不是已经存在
if len(users) > 0: # 邮箱已经存在用户
    raise APIError('register:failed', 'email', 'Email is already in use.')
uid = next_id()
new_pwd = '{}:{}'.format(uid, pwd)
user = User(id=uid, name=name, email=email, pwd=hashlib.sha1(new_pwd.encode('utf-8')).hexdigest(), image='http://www.gravatar.com/avatar/{}?s=80&d=identicon&r=g'.format(hashlib.md5(email.encode('utf-8')).hexdigest()))
await user.save() # 注册用户信息保存至数据库
# 创建 session 的 cookie
res = web.Response()
res.set_cookie(COOKIE_NAME, user2cookie(user, 86400), max_age=86400, httponly=True)
user.pwd = '*****'
res.content_type = 'application/json'
res.body = json.dumps(user, ensure_ascii=False).encode('utf-8')
return res

_COOKIE_KEY = configs.session.secret
COOKIE_NAME = 'hikari_session'

def user2cookie(user, max_age):
    # 计算加密 cookie, cookie 字符串为: id-expires-sha1
    expires = str(int(time.time()) + max_age) # 当前时间+最大寿命即为过期时间
    s = '{}-{}-{}'.format(user.id, user.pwd, expires, _COOKIE_KEY)
    lst = [user.id, expires, hashlib.sha1(s.encode('utf-8')).hexdigest()]
    return '-'.join(lst)

```

结果:

127.0.0.1:8000/register

hikari blog 日志 教程 源码

欢迎注册!

名字:

电子邮件:

输入密码:

确认密码:

 注册

注册之后跳转到首页，但不是处于登录状态，还显示登录和注册...
由于 HTTP 协议无状态，而服务器要跟踪用户状态，可以通过 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。
Session 的优点是简单易用，可以直接从 Session 中取出用户登录信息。
Session 的缺点是服务器需要在内存中维护一个映射表来存储用户登录信息，如果有两台以上服务器，就需要对 Session 做集群。
因此，使用 Session 的 Web App 很难扩展。

此处采用直接读取 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，这种方式的好处是保证服务器处理任意的 URL 都是无状态的，可以扩展到多台服务器。
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 cookie 的关键是通过一个单向算法(例如 SHA1)
比如：当用户输入正确密码登录成功后，服务器可以从数据库取到用户的 id，并以如下方式计算出一个字符串：

"用户 id" + "过期时间" + SHA1("用户 id" + "用户密码" + "过期时间" + "SecretKey")

当浏览器发送 cookie 到服务器后，服务器拿到的信息包括：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 查找用户密码，并计算：

SHA1("用户 id" + "用户密码" + "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登录，否则 cookie 就是伪造的。

此算法关键在于 SHA1 是一种单向算法，即可以通过原始字符串计算出 SHA1 结果，但无法通过 SHA1 结果反推出原始字符串。

③ 登录 API：登录成功设置 cookie

```
# 登录认证 api
@post('/api/authenticate')
async def authenticate(*, email, pwd):
    if not email:
        raise APIValueError('email', 'Invalid email.')
    if not pwd:
        raise APIValueError('password', 'Invalid password.')
    users = await User.find_all('email=?', [email])
    if len(users) == 0:
        raise APIValueError('email', 'Email not exist.')
    user = users[0]
    # 检查密码
    new_pwd = '{}:{}'.format(user.id, pwd)
    if user.pwd != hashlib.sha1(new_pwd.encode('utf-8')).hexdigest():
```

```

        raise APIValueError('password', 'Invalid password.')
    # 登录成功,设置 cookie
    res = web.Response()
    res.set_cookie(COOKIE_NAME, user2cookie(user, 86400), max_age=86400, httponly=True)
    user.pwd = '*****'
    res.content_type = 'application/json'
    res.body = json.dumps(user, ensure_ascii=False).encode('utf-8')
    return res

```

对于每个 URL 处理函数，如果都去写解析 cookie 的代码，那会导致代码重复。利用中间件在处理 URL 之前，把 cookie 解析，并将登录用户绑定到 request 对象上；这样后续的视图函数就可以直接拿到登录用户：

④ app.py 添加认证 cookie 中间件：获取 cookie 转为 user 对象绑定到 request

```

from handlers import cookie2user, COOKIE_NAME

# 利用中间件在处理 URL 之前，把 cookie 解析，并将登录用户绑定到 request 对象上
# 这样后续的视图就可以直接拿到登录用户
async def auth_factory(app, handler):
    async def auth(request):
        logging.info('check user: {}'.format(request.method, request.path))
        request.__user__ = None
        cookie_str = request.cookies.get(COOKIE_NAME)
        if cookie_str:
            user = await cookie2user(cookie_str)
            if user:
                logging.info('set current user: {}'.format(user.email))
                request.__user__ = user
            if request.path.startswith('/manage/') and (request.__user__ is None or not
request.__user__.admin):
                return web.HTTPFound('/signin')
            return await handler(request)
        return auth

```

⑤ app.py 的 response 中间件：响应是 dict 且模板非空情况，响应从 request 获取 user 字段，传给模板渲染：

```

async def response_factory(app, handler):
    async def response(request):
        # 前面一样...
        if isinstance(res, dict):
            template = res.get('__template__')
            if template is None: # 不带模板信息返回 json 对象
                res = web.Response(
                    body=json.dumps(res, ensure_ascii=False, default=lambda obj:
obj.__dict__).encode('utf-8'))

```

```

        res.content_type = 'application/json;charset=utf-8'
        return res
    else: # 带模板信息, 渲染模板
        res['__user__'] = request.__user__ # 新加一句, 从 request 获取用户信息
        tpl = app['__template__'].get_template(template)
        res = web.Response(body=tpl.render(**res).encode('utf-8'))
        res.content_type = 'text/html;charset=utf-8'
        return res

# .....
return res
return response

```

app.py 的 init 将认证中间件添加:

```

app = web.Application(loop=loop, middlewares=[logger_factory, auth_factory,
response_factory])

```

⑥ handlers.py 的函数将 cookie 字符串转为 user 对象:

```

async def cookie2user(cookie_str):
    # 解析 cookie
    if not cookie_str:
        return
    try:
        lst = cookie_str.split('-')
        if len(lst) != 3:
            return
        uid, expires, sha1 = lst
        if int(float(expires)) < time.time():
            print('过期啦')
            return
        user = await User.find(uid)
        if user is None:
            return
        s = '{}-{}-{}'.format(uid, user.pwd, expires, _COOKIE_KEY)
        if sha1 != hashlib.sha1(s.encode('utf-8')).hexdigest():
            logging.info('invalid sha1')
            return
        user.pwd = '*****'
        return user
    except Exception as e:
        logging.exception(e)
    return

```

⑦ handlers.py 登录视图:

```

@get('/signin')

```

```
def signin():
    return {
        '__template__': 'signin.html'
    }
```

⑧ templates/signin.html: 重新写，不继承

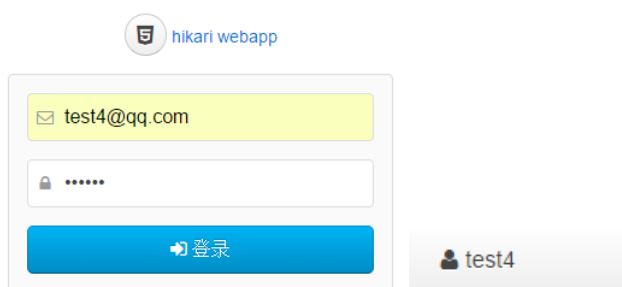
```
<!DOCTYPE html>
<html class="uk-height-1-1">
<head>
    <meta charset="utf-8"/>
    <title>登录 -hikari webapp</title>
    <link rel="stylesheet" href="/static/css/uikit.min.css">
    <link rel="stylesheet" href="/static/css/uikit.gradient.min.css">
    <script src="/static/js/jquery.min.js"></script>
    <script src="/static/js/sha1.min.js"></script>
    <script src="/static/js/uikit.min.js"></script>
    <script src="/static/js/vue.min.js"></script>
    <script src="/static/js/myblog.js"></script>
    <script>
        $(function () {
            var vmAuth = new Vue({
                el: '#vm',
                data: {
                    email: '',
                    pwd: ''
                },
                methods: {
                    submit: function (event) {
                        event.preventDefault();
                        var
                            $form = $('#vm'),
                            email = this.email.trim().toLowerCase(),
                            data = {
                                email: email,
                                pwd: this.pwd === '' ? '' : CryptoJS.SHA1(email + ':' +
this.pwd).toString()
                            };
                        $form.postJSON('/api/authenticate', data, function (err, r) {
                            if (!err) {
                                location.assign('/');
                            }
                        });
                    }
                }
            })
        })
    </script>
```

```

    });
  });
</script>
</head>
<body class="uk-height-1-1">
<div class="uk-vertical-align uk-text-center uk-height-1-1">
  <div class="uk-vertical-align-middle" style="width: 320px">
    <p><a href="/" class="uk-icon-button"><i class="uk-icon-html5"></i></a> <a
href="/">hikari webapp</a></p>
    <form id="vm" v-on="submit: submit" class="uk-panel uk-panel-box uk-form">
      <div class="uk-alert uk-alert-danger uk-hidden"></div>
      <div class="uk-form-row">
        <div class="uk-form-icon uk-width-1-1">
          <i class="uk-icon-envelope-o"></i>
          <input v-model="email" name="email" type="text" placeholder="电子邮箱"
maxlength="50" class="uk-width-1-1 uk-form-large"></div></div>
        <div class="uk-form-row">
          <div class="uk-form-icon uk-width-1-1">
            <i class="uk-icon-lock"></i>
            <input v-model="pwd" name="pwd" type="password" placeholder="密码"
maxlength="50" class="uk-width-1-1 uk-form-large"></div></div>
        <div class="uk-form-row">
          <button type="submit" class="uk-width-1-1 uk-button uk-button-primary
uk-button-large"><i class="uk-icon-sign-in"></i> 登录</button>
        </div></form></div></div>
</body>
</html>

```

结果:



每个继承于 base.html 的模板都会显示用户登录信息，根据 response 从 request 获取的 user，如果为 None 显示登录注册，否则显示用户名和登出。
是不是该去学一下 Vue.js 了？

⑨ 登出视图:

设置 cookie 为-deleted- (随意)，寿命为 0，其实就是删除 cookie

```

@get('/signout')
def signout(request): # 登出,重定向首页

```



```

referer = request.headers.get('Referer')
res = web.HTTPFound(referer or '/')
res.set_cookie(COOKIE_NAME, '-deleted-', max_age=0, httponly=True) # 删除 cookie
logging.info('user signed out.')
return res

```

结果:



20180409

🌸 Day 11. 编写日志创建页

在 Web 开发中, 后端代码写起来其实是相当容易的。

例如编写一个 REST API, 用于创建一个 Blog:

① handlers.py:

```

from apis import APIPermissionError

# 检查是不是管理员,有没有权限
def check_admin(request):
    if request.__user__ is None or not request.__user__.admin:
        raise APIPermissionError()

# REST API, 用于创建一个 Blog
@post('/api/blogs')
async def api_create_blog(request, *, name, summary, content):
    check_admin(request) # 管理员才能创建 blog
    # name,summary, content 都不能为空
    if not name or not name.strip():
        raise APIValueError('name', 'name cannot be empty.')
    if not summary or not summary.strip():
        raise APIValueError('summary', 'summary cannot be empty.')
    if not content or not content.strip():
        raise APIValueError('content', 'content cannot be empty.')
    # 创建 blog 对象, 保存到数据库
    blog = Blog(user_id=request.__user__.id, user_name=request.__user__.name,
                user_image=request.__user__.image,
                name=name.strip(), summary=summary.strip(), content=content.strip())
    await blog.save()
    return blog

```

编写后端 Python 代码简单而且容易测试, 上面的 API: `api_create_blog()` 本身只是一个普通函数。

Web 开发真正困难在于编写前端页面。前端页面需要混合 HTML、CSS 和 JavaScript，如果对它们没有深入掌握，编写的前端页面将很快难以维护。前端页面通常是动态页面，也就是前端页面往往是由后端代码生成的。

1) 生成前端页面最早方式是拼接字符串：

```
s = '<html><head><title>' + title + '</title></head><body>' + body + '</body></html>'
```

显然这种方式完全没有可维护性。

2) 模板方式：

```
<html>
<head><title>{{ title }}</title></head>
<body>{{ body }}</body>
</html>
```

ASP、JSP、PHP 等都是用这种模板方式生成前端页面。

如果在页面上大量使用 JavaScript (事实上大部分页面都会)，模板方式仍然会导致 JavaScript 代码与后端代码耦合紧密，以至于难以维护。其根本原因在于负责显示的 HTML DOM 模型与负责数据和交互的 JavaScript 代码没有分割清楚。

和后端结合的 **MVC 模式** 已经无法满足复杂页面逻辑的需要了，所以新的 **MVVM (Model-View-ViewModel)** 模式应运而生。

MVVM 最早由微软提出，它借鉴了桌面应用程序的 MVC 思想，在前端页面中，把 Model 用纯 JavaScript 对象表示：

```
<script>
  let blog = {
    name: 'hello',
    summary: 'this is summary',
    content: 'this is content...'
  };
</script>
```

View 是纯 HTML：

```
<form action="/api/blogs" method="post">
  <p><input name="name" id="name"></p>
  <p><input name="summary" id="summary"></p>
  <p><textarea name="content" id="content"></textarea></p>
  <button type="submit">OK</button>
</form>
```

由于 Model 表示数据，View 负责显示，两者做到了最大限度的分离。

把 Model 和 View 关联起来的的就是 ViewModel。ViewModel 负责把 Model 数据同步到 View 显示，还负责把 View 的修改同步回 Model。

需要用 JavaScript 编写一个通用的 ViewModel，可以复用整个 MVVM 模型。

已有许多成熟的 MVVM 框架，例如 AngularJS，KnockoutJS 等。

此处选择 **Vue** 这个简单易用的 MVVM 框架实现创建 Blog 的页面。

② templates/manage_blog_edit.html:

```
{% extends 'base.html' %}
```

```

{% block title %}编辑日志{% endblock %}
{% block beforehead %}
    <script>
        let
            ID = '{{ id }}',
            action = '{{ action }}';

        function initVM(blog) {
            let $vm = $('#vm');
            {# 让 vm 变为全局变量方便调试... #}
            vm = new Vue({
                {# 初始化 Vue, 指定 3 个参数 : ① el : 根据选择器查找绑定的 View, 这里是#vm,
                id 为 vm 的<div>标签 ② data : JavaScript 对象表示的 Model, 初始化为{ name: '', summary: '',
                content: ''} ③ methods : View 可以触发的 JavaScript 函数, submit 是提交表单时触发的函数 #}
                el: '#vm',
                data: blog,
                methods: {
                    submit: function (event) {
                        event.preventDefault();
                        let $form = $vm.find('form');
                        {# 将数据 POST 提交到 action #}
                        $form.postJSON(action, this.$data, function (err, r) {
                            if (err) {
                                $form.showFormError(err);
                            }
                            else {
                                {# 提交成功, 转到该 blog 文章页面 #}
                                return location.assign('/blogs/' + r.id);
                            }
                        });
                    }
                }
            });
            $vm.show();
        }

        $(function () {
            let $loading = $('#loading');
            if (ID) {
                getJSON('/api/blogs/' + ID, function (err, blog) {
                    if (err) {
                        return fatal(err);
                    }
                    $loading.hide();
                    initVM(blog);
                }
            }
        });
    </script>

```

```

    });
  }
  else {
    $loading.hide();
    initVM({
      name: '',
      summary: '',
      content: ''
    });
  }
});
</script>
{% endblock %}

{% block content %}
  <div class="uk-width-1-1 uk-margin-bottom">
    <div class="uk-panel uk-panel-box">
      <ul class="uk-breadcrumb">
        <li><a href="/manage/comments">评论</a></li>
        <li><a href="/manage/blogs">日志</a></li>
        <li><a href="/manage/users">用户</a></li></ul></div></div>
      <div id="error" class="uk-width-1-1"></div>
      <div id="loading" class="uk-width-1-1 uk-text-center">
        <span><i class="uk-icon-spinner uk-icon-medium uk-icon-spin"></i> 正在加
载...</span></div>
      <div id="vm" class="uk-width-2-3">
        {# 把提交表单的事件关联到 submit 方法 #}
        <form v-on="submit: submit" class="uk-form uk-form-stacked">
          <div class="uk-alert uk-alert-danger uk-hidden"></div>
          {# v-model 使 Vue 把 Model 和 View 关联起来, Model 的 name 为键, input 的 value 为值 #}
          <div class="uk-form-row">
            <label class="uk-form-label">标题:</label>
            <div class="uk-form-controls">
              <input v-model="name" name="name" type="text" placeholder="标题"
class="uk-width-1-1"></div></div>
            <div class="uk-form-row">
              <label class="uk-form-label">摘要:</label>
              <div class="uk-form-controls">
                <textarea v-model="summary" rows="4" name="summary"
placeholder="摘要" class="uk-width-1-1" style="resize:none;"></textarea></div></div>
            <div class="uk-form-row">
              <label class="uk-form-label">内容:</label>
              <div class="uk-form-controls">
                <textarea v-model="content" rows="16" name="content" placeholder="

```

```

内容" class="uk-width-1-1" style="resize:none;"></textarea></div></div>
    <div class="uk-form-row">
        <button type="submit" class="uk-button uk-button-primary"><i class="uk-
icon-save"></i> 保存</button>
        <a href="/manage/blogs" class="uk-button"><i class="uk-icon-times"></i>
取消</a></div></form></div>
{% endblock %}

```

③ base.html 在登出前面添加一个创建博客按钮(管理员权限):

```

<ul class="uk-nav uk-nav-navbar">
    {% if __user__.admin %}
        <li><a href="/manage/blogs/create"><i class="uk-icon-pencil"></i> 创建</a></li>
    {% endif %}
    <li><a href="/signout"><i class="uk-icon-sign-out"></i> 登出</a></li></ul>

```

④ handlers.py:

```

@get('/manage/blogs/create') # 创建 blog 的视图函数
def manage_create_blog():
    return {
        '__template__': 'manage_blog_edit.html',
        'id': '',
        'action': '/api/blogs'
    }

@get('/api/blogs/{id}')
async def api_get_blog(*, id):
    blog = await Blog.find(id)
    return blog

def text2html(text):
    # text 按行拆成列表, 滤去空字符, 将特殊字符转义, 加上 p 标签, 再拼接字符串
    lines = map(lambda s: '<p>{}</p>'.format(s.replace('&', '&amp;').replace('<',
'&lt;').replace('>', '&gt;')),
                filter(lambda s: s.strip() != '', text.split('\n')))
    return ''.join(lines)

import markdown2
@get('/blog/{id}')
async def get_blog(id): # 获取指定 id 的 blog
    blog = await Blog.find(id)
    comments = await Comment.find_all('blog_id=?', [id], order_by='created_at desc')
    for c in comments:
        c.html_content = text2html(c.content)
    blog.html_content = markdown2.markdown(blog.content)

```

```
return {
  '__template__': 'blog.html', # 需要写一个 blog.html 模板
  'blog': blog,
  'comments': comments
}
```

⑤ 注册新账号；MySQL: `update users set admin=1 where name='hikari 星'`;



点击创建就可以写日志

注意：在 MVVM 中，Model 和 View 是双向绑定的。如果在 Form 中修改了文本框的值，可以在 Model 中立刻拿到新值。在表单中输入文本，F12-Console，可以通过 `vm.name` 访问单个属性，或者通过 `vm.$data` 访问整个 Model：

```
Extends $form...
> vm.name
< "Test"
> vm.summary
< "This is summary"
> vm.content
< "Hello, world!"
> JSON.stringify(vm.$data)
< '{"name":"Test","summary":"This is summary","content":"Hello, world!"}'
```

如果在 JavaScript 逻辑中修改了 Model，这个修改会立刻反映到 View 上。比如在 Console 输入 `vm.name = '测试用'`，可以看到文本框的内容自动被同步了。



点击保存跳转到 `http://127.0.0.1:8000/api/blogs/{id}` 了，一串 JSON。

⑥ 将首页视图修改，从数据库获取：

```
@get('/')
async def index(request):
    blogs = await Blog.find_all()
    return {
        '__template__': 'myblog.html',
        'blogs': blogs,
    }
```

测试用

发表于1分钟前

This is summary

[继续阅读 >](#)

需要再建立 `blog/{id}` 的模板，点击继续阅读才不会报错。

双向绑定是 MVVM 框架最大的作用。借助 MVVM 将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：

① apis.py: 定义 Page 类用于存储分页信息：

```
class Page(object):
    # Page 类用于存储分页信息
    def __init__(self, item_count, page_index=1, page_size=10):
        self.item_count = item_count # 总条目数
        self.page_size = page_size # 一页的条目数
        # 总共多少页
        self.page_count = item_count // page_size + (1 if item_count % page_size > 0 else 0)
        # 如果总条目数为 0 或当前页数超过总页数
        if (item_count == 0) or (page_index > self.page_count):
            self.offset = 0
            self.limit = 0
            self.page_index = 1
        else:
            self.page_index = page_index # 当前页数
            self.offset = self.page_size * (page_index - 1) # 偏移, 当前页之前总条目数
            self.limit = self.page_size
        # 是否有前一页或后一页
        self.has_next = self.page_index < self.page_count
        self.has_previous = self.page_index > 1

    def __str__(self):
        return 'item_count: {}, page_count: {}, page_index: {}, page_size: {}, offset: {}, limit: {}'.format(self.item_count, self.page_count, self.page_index, self.page_size, self.offset, self.limit)
    __repr__ = __str__
```

② handlers.py: blog API

```
def get_page_index(page_str):
    # 字符串页数变为整数, 非法页数全变为 1
    p = 1
    try:
        p = int(page_str)
    except ValueError:
        pass
    if p < 1:
        p = 1
    return p

from apis import Page
```

```

@get('/api/blogs')
async def api_blogs(*, page='1'):
    page_index = get_page_index(page) # 指定页数
    num = await Blog.find_number('count(id)') # 总条目数?
    p = Page(num, page_index) # 创建 Page 对象
    if num == 0:
        return dict(page=p, blogs=())
    # 从 offset 开始 limit 条
    blogs = await Blog.find_all(order_by='created_at desc', limit=(p.offset, p.limit))
    return dict(page=p, blogs=blogs) # 返回 Page 对象和 Blog 对象

```

③ handlers.py 管理页面视图函数:

```

@get('/manage/blogs')
def manage_blogs(*, page='1'):
    return {
        '__template__': 'manage_blogs.html',
        'page_index': get_page_index(page)
    }

```

④ manage_blogs.html:

```

{% extends 'base.html' %}
{% block title %}日志{% endblock %}

{% block beforehead %}
<script>
    function initVM(data) {
        let vm = new Vue({
            el: '#vm',
            data: {
                blogs: data.blogs,
                page: data.page
            },
            methods: {
                {# 编辑 blog 和删除 blog #}
                edit_blog: function (blog) {
                    location.assign('/manage/blogs/edit?id=' + blog.id);
                },
                delete_blog: function (blog) {
                    if (confirm('确认要删除 "' + blog.name + '" ? 删除后不可恢复!')) {
                        postJSON('/api/blogs/' + blog.id + '/delete', function (err, r) {
                            if (err) {
                                return alert(err.message || err.error || err);
                            }
                        });
                    }
                }
            }
        });
    }
    refresh();

```



```

        });
    }
}

});
$('#vm').show();
}

$(function () {
    {# 通过 API GET /api/blogs?page=?拿到 Model #}
    getJSON('/api/blogs', {
        page: {{ page_index }}
    }, function (err, results) {
        if (err) {
            return fatal(err);
        }
        $('#loading').hide();
        {# 初始化 View #}
        initVM(results);
    });
});
</script>
{% endblock %}

{% block content %}
<div class="uk-width-1-1 uk-margin-bottom">
    <div class="uk-panel uk-panel-box">
        <ul class="uk-breadcrumb">
            <li><a href="/manage/comments">评论</a></li>
            <li class="uk-active"><span>日志</span></li>
            <li><a href="/manage/users">用户</a></li></ul></div></div>
    <div id="error" class="uk-width-1-1"></div>
    <div id="loading" class="uk-width-1-1 uk-text-center">
        <span><i class="uk-icon-spinner uk-icon-medium uk-icon-spin"></i> 正在加
载...</span></div>
    <div id="vm" class="uk-width-1-1">
        <a href="/manage/blogs/create" class="uk-button uk-button-primary"><i
class="uk-icon-plus"></i> 新日志</a>
        <table class="uk-table uk-table-hover">
            <thead>
                <tr>
                    <th class="uk-width-5-10">标题 / 摘要</th>
                    <th class="uk-width-2-10">作者</th>
                    <th class="uk-width-2-10">创建时间</th>
                    <th class="uk-width-1-10">操作</th></tr></thead>

```

```

<tbody>
{# 用 v-repeat 可以把 Model 的 blogs 数组直接变成多行<tr> #}
<tr v-repeat="blog: blogs"> {# v-repeat="blog: blogs"可以看成循环代码 #}
  <td>{# v-text 和 v-attr 分别用于生成文本和 DOM 结点属性 #}
    <a target="_blank" v-attr="href: '/blog/'+blog.id" v-
text="blog.name"></a></td>
    <td>
      <a target="_blank" v-attr="href: '/user/'+blog.user_id" v-
text="blog.user_name"></a></td>
    <td>
      <span v-text="blog.created_at.toDateTime()"></span></td>
    <td>
      <a href="#0" v-on="click: edit_blog(blog)"><i class="uk-icon-
edit"></i></a>
      <a href="#0" v-on="click: delete_blog(blog)"><i class="uk-icon-trash-
o"></i></a></td></tr></tbody></table>
<div v-component="pagination" v-with="page"></div></div>
{% endblock %}

```

结果:

444	hikari星	2018-04-10 09:49:21	✎ 🗑
555	hikari星	2018-04-10 09:49:25	✎ 🗑
666	hikari星	2018-04-10 09:49:31	✎ 🗑

<< 1 >>

标题 / 摘要	作者	创建时间	操作
777	hikari星	2018-04-10 09:49:49	✎ 🗑
888	hikari星	2018-04-10 09:49:54	✎ 🗑

<< 2 >>

编辑删除 blog, /user/{id}, /blog/{id}, 评论都需要实现

20180410

🌸 Day 13. 提升开发效率

现在已经把一个 Web App 的框架搭建好了, 从后端的 API 到前端的 MVVM, 流程已经跑通了。但是每次修改 Python 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, 开发阶段这样严重降低开发效率。

Django 的开发环境在 Debug 模式可以自动重新加载, 如果自己编写的服务器也能实现这个功能, 就大大提升开发效率; 可惜 Django 没把这个功能独立出来。Python 也提供了重新载入模块的功能, 但不是所有模块都能被重新载入。

另一种思路是一旦检测到 www 目录下代码改动, 就自动重启服务器。可以编写一个辅助程序 pymonitor.py, 让它启动 wsgiapp.py, 并时刻监控 www 目录下代码的改动; 有改动时先把当前 wsgiapp.py 进程杀掉再重启, 就完成了服务器进程的自动重启。

Python 的第三方库 [watchdog](#) 可以利用操作系统的 API 来监控目录文件的变化，并发送通知。利用 watchdog 接收文件变化的通知，如果是.py 文件，就自动重启 wsgiapp.py 进程；利用 Python 自带 subprocess 实现进程的启动和终止，并把输入输出重定向到当前进程的输入输出中：

```
import os
import subprocess
import sys
import time
from watchdog.events import FileSystemEventHandler
from watchdog.observers import Observer

def log(s):
    print('[Monitor] {}'.format(s))

class MyFileSystemEventHandler(FileSystemEventHandler):
    def __init__(self, f):
        super().__init__()
        self.restart = f # 传入 restart_process 函数
    def on_any_event(self, event):
        # 检测到 py 文件有改动, 调用传入的函数
        if event.src_path.endswith('.py'):
            log('Python source file changed: {}'.format(event.src_path))
            self.restart()

command = ['echo', 'ok']
process = None

def kill_process(): # 关闭进程
    global process
    if process:
        log('Kill process [{}]...'.format(process.pid))
        process.kill()
        process.wait()
        log('Process ended with code {}'.format(process.returncode))
        process = None

def start_process(): # 启动进程
    global process, command
    log('Start process {}'.format(' '.join(command)))
    process = subprocess.Popen(command, stdin=sys.stdin, stdout=sys.stdout,
stderr=sys.stderr)

def restart_process(): # 重启进程
```

```

kill_process()
start_process()

def start_watch(path, callback): # 监视 path 路径文件变化
    observer = Observer()
    observer.schedule(MyFileSystemEventHandler(restart_process), path, recursive=True)
    observer.start()
    log('Watching directory {}'.format(path))
    start_process()
    try:
        while True:
            time.sleep(0.5)
    except KeyboardInterrupt:
        observer.stop() # ctrl-c 停止监视
    observer.join()

if __name__ == '__main__':
    argv = sys.argv[1:] # 命令行参数
    if not argv:
        print('Usage: ./pymonitor your-script.py')
        exit(0)
    if argv[0] != 'python':
        argv.insert(0, 'python')
    command = argv
    path = os.path.abspath('.') # 监视当前 py 文件所在目录
    start_watch(path, None)

```

./pymonitor.py app.py 启动服务，在 handlers.py 尾部敲一个回车：

INFO:root:server started at http://127.0.0.1:8000...

[Monitor] Python source file changed: D:\hikari 星\hikari_web_day13\www\handlers.py

[Monitor] Kill process [10728]...

[Monitor] Process ended with code 1.

[Monitor] Start process python app.py...

也就是一保存代码，自动重启服务，大大提升了开发效率。

目前为止的路由有：

INFO:root:add route GET /api/blogs --> api_blogs(page)

INFO:root:add route POST /api/blogs --> api_create_blog(request,name,summary,content)

INFO:root:add route GET /api/blogs/{id} --> api_get_blog(id)

INFO:root:add route GET /api/users --> api_get_users()

INFO:root:add route POST /api/users --> api_register_user(email,name,pwd)

INFO:root:add route POST /api/authenticate --> authenticate(email,pwd)

INFO:root:add route GET /blog/{id} --> get_blog(id)

INFO:root:add route GET / --> index(request)

INFO:root:add route GET /manage/blogs --> manage_blogs(page)

```
INFO:root:add route GET /manage/blogs/create --> manage_create_blog()
INFO:root:add route GET /register --> register()
INFO:root:add route GET /signin --> signin()
INFO:root:add route GET /signout --> signout(request)
INFO:root:add static /static/ --> D:\hikari 星\hikari_web_day13\www\static
```

🌸 Day 14. 完成 Web App

在 Web App 框架和基本流程跑通后，剩下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。

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，这个之前在中间件 auth_factory 已经做过了...

1. 后端 API

- ① 获取日志：GET /api/blogs
- ② 创建日志：POST /api/blogs
- ③ 修改日志：POST /api/blogs/{blog_id}

```
@post('/api/blogs/{id}')
async def api_update_blog(id, *, name, summary, content):
    # 获取指定 id 的 blog 修改
    blog = await Blog.find(id)
    blog.name = name
    blog.summary = summary
    blog.content = content
    await blog.update()
    return blog
```

- ④ 删除日志：POST /api/blogs/{blog_id}/delete

```
@post('/api/blogs/{id}/delete')
async def api_delete_blog(id, request):
    # 删除指定 id 的 blog
    check_admin(request)
    b = await Blog.find(id)
    if b is None:
        raise APIResourceNotFoundError('Blog')
    await b.remove()
    return dict(id=id)
```

- ⑤ 获取评论：GET /api/comments

```
@get('/api/comments')
async def api_comments(request, *, page='1'):
    check_admin(request)
    # 获取所有评论分页显示
    page_index = get_page_index(page)
```

```

num = await Comment.find_number('count(id)')
p = Page(num, page_index)
if num == 0:
    return dict(page=p, comments=())
comments = await Comment.find_all(order_by='created_at desc', limit=(p.offset, p.limit))
return dict(page=p, comments=comments)

```

⑥ 创建评论：POST /api/blogs/{blog_id}/comments

```

@post('/api/blogs/{id}/comments')
async def api_create_comment(id, request, *, content):
    user = request.__user__
    if user is None: # 未登录不能评论
        raise APIPermissionError('Please signin first.')
    if not content or not content.strip(): # 没有内容
        raise APIValueError('content')
    blog = await Blog.find(id)
    if blog is None:
        raise APIResourceNotFoundError('Blog')
    comment = Comment(blog_id=blog.id, user_id=user.id, user_name=user.name,
user_image=user.image, content=content.strip())
    await comment.save()
    return comment

```

⑦ 删除评论：POST /api/comments/{comment_id}/delete

```

@post('/api/comments/{id}/delete')
async def api_delete_comments(id, request):
    check_admin(request)
    c = await Comment.find(id)
    if c is None:
        raise APIResourceNotFoundError('Comment')
    await c.remove()
    return dict(id=id)

```

⑧ 创建新用户：POST /api/users

⑨ 获取用户：GET /api/users

```

@get('/api/users')
async def api_get_users(request, *, page='1'):
    check_admin(request) # 非管理员不能直接访问显示
    # 获取所有用户信息分页显示
    page_index = get_page_index(page)
    num = await User.find_number('count(id)')
    p = Page(num, page_index)
    if num == 0:
        return dict(page=p, users=())

```

```

users = await User.find_all(order_by='created_at desc', limit=(p.offset, p.limit))
for u in users:
    u.pwd = '*****'
return dict(page=p, users=users)

```

2. 管理页面：

① 评论列表页：GET /manage/comments

```

@get('/manage/comments')
def manage_comments(*, page='1'): # 管理评论
    return {
        '__template__': 'manage_comments.html',
        'page_index': get_page_index(page)
    }

```

② 日志列表页：GET /manage/blogs

③ 创建日志页：GET /manage/blogs/create

④ 修改日志页：GET /manage/edit

```

@get('/manage/blogs/edit')
def manage_edit_blog(*, id):
    return {
        '__template__': 'manage_blog_edit.html',
        'id': id,
        'action': '/api/blogs/{}'.format(id)
    }

```

⑤ 用户列表页：GET /manage/users

```

@get('/manage/users')
def manage_users(*, page='1'):
    return {
        '__template__': 'manage_users.html',
        'page_index': get_page_index(page)
    }

```

⑥ 底部/manage 默认重定向到/manage/comments

```

@get('/manage/')
def manage():
    return 'redirect:/manage/comments'

```

3. 用户浏览页面包括：

① 注册页：GET /register

② 登录页：GET /signin

③ 注销页：GET /signout

④ 首页：GET /

```

@get('/')

```

```

async def index(*, page='1'):
    # 获取所有 blog 分页显示
    page_index = get_page_index(page)
    num = await Blog.find_number('count(id)')
    page = Page(num, page_index)
    if num == 0:
        blogs = []
    else:
        blogs = await Blog.find_all(order_by='created_at desc', limit=(page.offset, page.limit))
    return {
        '__template__': 'myblog.html',
        'page': page,
        'blogs': blogs
    }

```

⑤ 日志详情页：GET /blog/{blog_id}

4 模板部分

① base.html 加入分页的宏定义：

```

{% macro pagination(url, page) %}
    <ul class="uk-pagination">
        {% if page.has_previous %}
            <li><a href="{{ url }}{{ page.page_index - 1 }}"><i class="uk-icon-angle-double-left"></i></a></li>
        {% else %}
            <li class="uk-disabled"><span><i class="uk-icon-angle-double-left"></i></span></li>
        {% endif %}
        <li class="uk-active"><span>{{ page.page_index }}</span></li>
        {% if page.has_next %}
            <li><a href="{{ url }}{{ page.page_index + 1 }}"><i class="uk-icon-angle-double-right"></i></a></li>
        {% else %}
            <li class="uk-disabled"><span><i class="uk-icon-angle-double-right"></i></span></li>
        {% endif %}
    </ul>
{% endmacro %}

```

② blog.html：显示单个 blog 和评论

```

{% extends 'base.html' %}

{% block title %}{{ blog.name }}{% endblock %}

```



```

{% block beforehead %}
<script>
  let comment_url = '/api/blogs/{{ blog.id }}/comments';
  $(function () {
    let $form = $('#form-comment');
    {# 提交评论事件 #}
    $form.submit(function (e) {
      e.preventDefault();
      $form.showFormError("");
      let content = $form.find('textarea').val().trim();
      if (content === '') {
        return $form.showFormError('请输入评论内容！');
      }
      $form.postJSON(comment_url, {content: content}, function (err, result) {
        if (err) {
          return $form.showFormError(err);
        }
        refresh();
      });
    });
  });
</script>
{% endblock %}

{% block content %}
<div class="uk-width-medium-3-4">
  <article class="uk-article">
    <h2>{{ blog.name }}</h2>
    <p class="uk-article-meta">发表于{{ blog.created_at|datetime }}</p>
    {# safe 过滤器不用乱用, 自己写的无所谓... #}
    <p>{{ blog.html_content|safe }}</p></article>
    <hr class="uk-article-divider">
    {% if __user__ %}
      <h3>发表评论</h3>
      <article class="uk-comment">
        <header class="uk-comment-header">
          
          <h4 class="uk-comment-title">{{ __user__.name }}</h4>
        </header>
        <div class="uk-comment-body">
          <form id="form-comment" class="uk-form">
            <div class="uk-alert uk-alert-danger uk-hidden"></div>
            <div class="uk-form-row">

```

```

        <textarea rows="6" placeholder="说点什么吧"
style="width:100%;resize:none;"></textarea></div>
        <div class="uk-form-row">
            <button type="submit" class="uk-button uk-button-
primary"><i class="uk-icon-comment"></i>
                发表评论</button></div></form></div></article>
        <hr class="uk-article-divider">
    {% endif %}
    <h3>最新评论</h3>
    <ul class="uk-comment-list">
        {% for comment in comments %}
            <li>
                <article class="uk-comment">
                    <header class="uk-comment-header">
                        
                        <h4 class="uk-comment-title">{{ comment.user_name }} {% if
comment.user_id==blog.user_id %}(作者){% endif %}</h4>
                        <p class="uk-comment-
meta">{{ comment.created_at|datetime }}</p>
                    </header>
                    <div class="uk-comment-body">
                        {{ comment.html_content|safe }}</div></article></li>
                {% else %}
                    <p>还没有人评论...</p>
                {% endfor %}</ul></div>
    <div class="uk-width-medium-1-4">
        <div class="uk-panel uk-panel-box">
            <div class="uk-text-center">
                
                <h3>{{ blog.user_name }}</h3></div></div>
        <div class="uk-panel uk-panel-header">
            <h3 class="uk-panel-title">友情链接</h3>
            <ul class="uk-list uk-list-line">
                <li><i class="uk-icon-link"></i> <a href="#">编程</a></li>
                <li><i class="uk-icon-link"></i> <a href="#">思考</a></li>
                <li><i class="uk-icon-link"></i> <a href="#">读书</a></li>
            </ul></div></div>
    {% endblock %}

```

③ manage_comments.html 与 manage_blogs.html 和 manage_users.html 类似:

```
{% extends 'base.html' %}
```

```

{% block title %}评论{% endblock %}

{% block beforehead %}
<script>
    function initVM(data) {
        $('#vm').show();
        let vm = new Vue({
            el: '#vm',
            data: {
                comments: data.comments,
                page: data.page
            },
            methods: {
                delete_comment: function (comment) {
                    let content = comment.content.length > 20 ?
comment.content.substring(0, 20) + '...' : comment.content;
                    if (confirm('确认要删除评论 "' + comment.content + '" ? 删除后不可
恢复！')) {
                        postJSON('/api/comments/' + comment.id + '/delete', function
(err, r) {

                            if (err) {
                                return error(err);
                            }
                            refresh();
                        });
                    }
                }
            }
        });
        $(function () {
            getJSON('/api/comments', {
                page: {{ page_index }}
            }, function (err, results) {
                if (err) {
                    return fatal(err);
                }
                $('#loading').hide();
                initVM(results);
            });
        });
    }
</script>
{% endblock %}

```

```

{% block content %}
  <div class="uk-width-1-1 uk-margin-bottom">
    <div class="uk-panel uk-panel-box">
      <ul class="uk-breadcrumb">
        <li class="uk-active"><span>评论</span></li>
        <li><a href="/manage/blogs">日志</a></li>
        <li><a href="/manage/users">用户</a></li></ul></div></div>

    <div id="error" class="uk-width-1-1"></div>
    <div id="loading" class="uk-width-1-1 uk-text-center">
      <span><i class="uk-icon-spinner uk-icon-medium uk-icon-spin"></i> 正在加
载...</span></div>
    <div id="vm" class="uk-width-1-1" style="display:none">
      <table class="uk-table uk-table-hover">
        <thead>
          <tr>
            <th class="uk-width-2-10">作者</th>
            <th class="uk-width-5-10">内容</th>
            <th class="uk-width-2-10">创建时间</th>
            <th class="uk-width-1-10">操作</th></tr></thead>
        <tbody>
          <tr v-repeat="comment: comments">
            <td><span v-text="comment.user_name"></span></td>
            <td><span v-text="comment.content"></span></td>
            <td><span v-text="comment.created_at.toDateTime()"></span></td>
            <td>
              <a href="#0" v-on="click: delete_comment(comment)"><i class="uk-
icon-trash-o"></i></a></td></tr></tbody></table>
          <div v-component="pagination" v-with="page"></div></div>
{% endblock %}

```

④ manage_users.html

```

{% extends 'base.html' %}

{% block title %}用户{% endblock %}

{% block beforehead %}
  <script>
    function initVM(data) {
      $('#vm').show();
      let vm = new Vue({
        el: '#vm',
        data: {
          users: data.users,
          page: data.page

```

```

    }
    });
}
$(function () {
    getJSON('/api/users', {
        page: {{ page_index }}
    }, function (err, results) {
        if (err) {
            return fatal(err);
        }
        $('#loading').hide();
        initVM(results);
    });
});
</script>
{% endblock %}

{% block content %}
<div class="uk-width-1-1 uk-margin-bottom">
    <div class="uk-panel uk-panel-box">
        <ul class="uk-breadcrumb">
            <li><a href="/manage/comments">评论</a> </li>
            <li><a href="/manage/blogs">日志</a> </li>
            <li class="uk-active"><span>用户</span> </li> </ul> </div> </div>
        <div id="error" class="uk-width-1-1"></div>
        <div id="loading" class="uk-width-1-1 uk-text-center">
            <span><i class="uk-icon-spinner uk-icon-medium uk-icon-spin"></i> 正在加
载...</span></div>
        <div id="vm" class="uk-width-1-1">
            <table class="uk-table uk-table-hover">
                <thead>
                    <tr>
                        <th class="uk-width-4-10">名字</th>
                        <th class="uk-width-4-10">电子邮件</th>
                        <th class="uk-width-2-10">注册时间</th> </tr> </thead>
                <tbody>
                    <tr v-repeat="user: users">
                        <td>
                            <span v-text="user.name"></span>
                            <span v-if="user.admin" style="color:#d05"><i class="uk-icon-
key"></i> 管理员</span></td>
                        <td>
                            <a v-attr="href: 'mailto:'+user.email" v-text="user.email"></a></td>
                        <td><span v-text="user.created_at.toDateTime()"></span></td>

```

```

</tr></tbody></table>
<div v-component="pagination" v-with="page"></div></div>
{% endblock %}

```

⑤ 首页 myblog.html 添加分页

```

{% for blog in blogs %}
    {# 与之前一样 #}
{% endfor %}
{{ pagination('/?page=', page) }}{# 添加分页 #}

```

大部分功能都实现了，如增删改博客，用户注册登录退出，评论管理等，主要问题：

- ① markdown 貌似很奇怪...
- ② 用户头像没有管理，需要一个/user/{id}显示用户信息
- ③ 需要添加几个易用性的按钮

20180411

🌸 Day 15. 部署 Web App

需要把 Web App 部署到远程服务器上，广大用户才能访问到网站。

很多做开发的童鞋把部署看成是运维的工作，这是完全错误的。壹. 最近流行 DevOps 理念，意思是开发和运维要变成一个整体。贰. 运维的难度其实跟开发质量有很大的关系。代码写得垃圾，运维再好也扛不住。叁. DevOps 理念需要把运维、监控等功能融入到开发中。想服务器升级时不中断用户服务?那就需要在开发时考虑到这一点。

🌸 将 hikari_web_app 部署到 Linux 服务器

① 搭建 Linux 服务器

首先得有一台 Linux 服务器。要在公网上体验的童鞋，可以在 Amazon 的 AWS 申请一台 EC2 虚拟机(免费使用 1 年)，或者使用国内的一些云服务器，一般都提供 Ubuntu Server 的镜像。想在本地部署的同学，请安装虚拟机，推荐使用 VirtualBox。

Linux 安装完成后，确保 ssh 服务正在运行，否则需要通过 apt 安装：

```
sudo apt-get install openssh-server
```

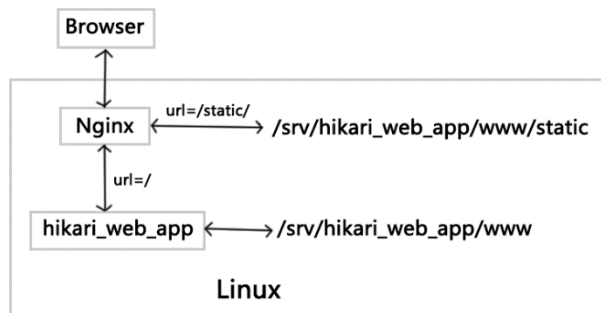
有了 ssh 服务，就可以从本地连接到服务器。建议把公钥复制到服务器端用户的.ssh/authorized_keys 中，就可以通过证书实现无密码连接。

② 部署方式

利用 Python 自带的 asyncio 已经编写了一个异步高性能服务器，但还需要一个高性能的 Web 服务器。此处选择 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：



服务器端定义好部署的目录结构：

```
/
+- srv/
  +- hikari_web_app/      <-- Web App 根目录
    +- www/              <-- 存放 Python 源码
      +- static/         <-- 存放静态资源文件
    +- log/              <-- 存放 log
```

在服务器上部署，要考虑到如果新版本运行不正常，需要回退到旧版本时怎么办。每次用新的代码覆盖掉旧的文件是不行的，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` 作为一个软链接，它指向哪个目录，哪个目录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。

Nginx 可以作为服务进程直接启动，但 `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。

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
```

然后，把用到的 Python 库安装：

```
sudo pip3 install jinja2 aiomysql aiohttp
```

在服务器上创建目录 `/srv/hikari_web_app/` 以及相应的子目录。

在服务器上初始化 MySQL 数据库，把数据库初始化脚本 `schema.sql` 复制到服务器上执行：`mysql -u root -p < schema.sql`

服务器端准备就绪。

③ 部署

用 FTP 还是 SCP 还是 `rsync` 复制文件？如果要手动复制，用一次两次还行，一天如果部署 50 次不但速度慢、效率低，而且容易出错。

正确的部署方式是使用工具配合脚本完成自动化部署。[Fabric](#) 就是一个自动化部

署工具。由于 Fabric 是用 Python 2.x 开发的,所以部署脚本要用 Python 2.7 编写,本机(自己电脑,不是服务器)还必须安装 Python 2.7 版本;Linux 服务器不需要安装 Fabric, Fabric 使用 SSH 直接登录服务器并执行部署命令。

// 好像现在已经支持 Python3 了...

编写部署脚本。Fabric 的部署脚本叫 fabfile.py,把它放到 hikari_web_app 目录下,与 www 目录平级:

1) 导入 Fabric 的 API, 设置部署时的变量:

```
# 导入 Fabric API
from fabric.api import *
# 服务器登录用户名
env.user = 'hikari'
# sudo 用户为 root
env.sudo_user = 'root'
# 服务器地址可以有多个, 依次部署
env.hosts = ['192.168.1.101']
# 服务器 MySQL 用户名和密码
db_user = 'root'
db_password = 'mysql'
```

每个 Python 函数都是一个任务。

2) 打包任务:

```
import os
_TAR_FILE = 'myblog.tar.gz'
def build():
    # 打包任务
    includes = ['static', 'templates', 'transwarp', 'favicon.ico', '*.py']
    excludes = ['test', '.*', '*.pyc', '*.pyo']
    local('rm -f dist/{}'.format(_TAR_FILE))
    # 把当前命令的目录设定为 lcd()指定的目录
    with lcd(os.path.join(os.path.abspath('.'), 'www')):
        cmd = ['tar', '--dereference', '-czvf', './dist/{}'.format(_TAR_FILE)]
        cmd.extend(['--exclude={}{}'.format(x) for x in excludes])
        cmd.extend(includes)
    local(' '.join(cmd)) # local()运行本地命令
```

注意: Fabric 只能运行命令行命令, Windows 下可能需要 Cgywin 环境。

在 hikari_web_app 目录下运行: fab build

在 dist 目录下创建了 myblog.tar.gz 的文件。

3) 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:

```
from datetime import datetime
_REMOTE_TMP_TAR = '/tmp/{}'.format(_TAR_FILE)
```



```

_REMOTE_BASE_DIR = '/srv/hikari_web_app'
def deploy():
    newdir = 'www-{}'.format(datetime.now().strftime('%Y-%m-%d_%H.%M.%S'))
    # run()函数执行的命令是在服务器上运行
    run('rm -f {}'.format(_REMOTE_TMP_TAR)) # 删除已有的 tar 文件
    put('dist/{}'.format(_TAR_FILE), _REMOTE_TMP_TAR) # 上传新的 tar 文件
    # 根据时间创建新目录
    with cd(_REMOTE_BASE_DIR):
        sudo('mkdir {}'.format(newdir))
    # 解压到新目录
    with cd('{}{}'.format(_REMOTE_BASE_DIR, newdir)):
        sudo('tar -xzf {}'.format(_REMOTE_TMP_TAR))
    # 重置软链接
    with cd(_REMOTE_BASE_DIR):
        sudo('rm -f www')
        sudo('ln -s {} www'.format(newdir))
        # 将 www 的拥有者设为 users 群体的 hikari
        sudo('chown hikari:users www')
        sudo('chown -R hikari:users {}'.format(newdir)) # -R 对指定目录递归改变拥有者
    # 重启 Python 服务和 nginx 服务器
    with settings(warn_only=True):
        sudo('supervisorctl stop hikari_web_app')
        sudo('supervisorctl start hikari_web_app')
        sudo('/etc/init.d/nginx reload')

```

注意：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，要用 sudo()来执行。

20180412

④ 配置 Supervisor

上面让 Supervisor 重启 hikari_web_app 会失败，因为还没有配置 Supervisor。

编写一个 Supervisor 的配置文件/etc/supervisor/conf.d/hikari_web_app.conf:

```

[program:hikari_web_app]

command      = /srv/hikari_web_app/www/app.py
directory    = /srv/hikari_web_app/www
user         = root
startsecs    = 3

redirect_stderr      = true
stdout_logfile_maxbytes = 50MB
stdout_logfile_backups = 10

```

```
stdout_logfile      = /srv/hikari_web_app/log/app.log
```

配置文件通过[program: hikari_web_app]指定服务名为 hikari_web_app，command 指定启动 app.py。

重启 Supervisor 后，就可以随时启动和停止 Supervisor 管理的 services 了：

```
$ sudo supervisorctl reload
$ sudo supervisorctl start hikari_web_app
$ sudo supervisorctl status
```

⑤ 配置 Nginx

配置文件/etc/nginx/sites-available/hikari_web_app:

```
server {
    listen      80;# 监听 80 端口
    root        /srv/hikari_web_app/www;
    access_log  /srv/hikari_web_app/log/access_log;
    error_log   /srv/hikari_web_app/log/error_log;
    # server_name www.hikari-blog.com; # 配置域名
    client_max_body_size 1m;
    gzip        on;
    gzip_min_length 1024;
    gzip_buffers 4 8k;
    gzip_types   text/css application/x-javascript application/json;
    sendfile on;
    location /favicon.ico { # 处理静态文件/favicon.ico
        root /srv/hikari_web_app/www;
    }
    location ~ ^\static\.*$ { # 处理静态资源
        root /srv/hikari_web_app/www;
    }
    location / { # 动态请求转发到 8000 端口
        proxy_pass      http://127.0.0.1:8000;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

在/etc/nginx/sites-enabled/目录下创建软链接：

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

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/hikari_web_app/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，可以在浏览器中访问 Linux 服务器上的 hikari_web_app

如果在开发环境更新了代码，只需要在命令行执行：

```
$ fab build
```

```
$ fab deploy
```

自动部署完成！刷新浏览器就可以看到服务器代码更新后的效果。

嫌国外网速慢的童鞋请移步[网易](#)和[搜狐](#)的镜像站点。

// [为什么阿里云和腾讯云的服务器辣么的贵...](#)

🌸 Day 16. 编写移动 App

在移动互联网浪潮席卷而来的今天，一个网站没有上线移动 App，出门根本不好意思跟人打招呼。所以必须得有一个移动 App 版本！

开发 iPhone App 前置条件：一台 Mac 电脑，安装 XCode 和最新的 iOS SDK。
在使用 MVVM 编写前端页面时，用 REST API 封装网站后台的功能，能清晰地分离前端页面和后台逻辑；现在这个好处更加明显，移动 App 也可以通过 REST API 从后端拿到数据。

设计一个简化版的 iPhone App，包含两个屏幕：列出最新日志和阅读日志的详细内容。只需要调用 API: /api/blogs。在 XCode 中完成 App 编写：

关于如何开发 iOS，请移步 [Develop Apps for iOS](#)。

如何编写 Android App？这个当成作业了。

// so difficult!

20180417

🌸 markdown 部分修改

handlers.py:

```
import markdown
@get('/blog/{id}')
async def get_blog(id): # 获取指定 id 的 blog
    blog = await Blog.find(id)
    comments = await Comment.find_all('blog_id=?', [id], order_by='created_at desc')
    for c in comments:
        c.html_content = text2html(c.content)
    # 需要添加扩展的列表
    exts = ['markdown.extensions.extra', 'markdown.extensions.codehilite',
'markdown.extensions.tables',
'markdown.extensions.toc']
    blog.html_content = markdown.markdown(blog.content, extensions=exts)
    return {
        '__template__': 'blog.html',
```

```
'blog': blog,
'comments': comments
}
```

安装 markdown 和 pygments

命令行: `pygmentize -S default -f html > default.css`

生成一个默认的语法高亮 css 文件

然后在 base.html 中导入:

```
<link rel="stylesheet" href="/static/css/default.css"/>
```

效果:

冒泡排序

发表于5天前

```
def bubble_sort(arr):
    for i in range(len(arr) - 1, 0, -1):
        exchange = False
        for j in range(i):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                exchange = True
        if not exchange:
            break
```

⚙️ 管理按键修改, 默认跳转到博客管理

handlers.py:

```
@get('/manage/') # 点击管理默认重定向到博客管理页面
def manage():
    return 'redirect:/manage/blogs'
```

⚙️ 在单个博客页面添加编辑按钮

blog.html:

```
<h2>{{ blog.name }}</h2>
{% if __user__.admin %}
    <p><a href="/manage/blogs/edit?id={{ blog.id }}">
        <i class="uk-icon-pencil"></i> 编辑</a></p>
{% endif %}
<p class="uk-article-meta">发表于{{ blog.created_at|datetime }}</p>
```