**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：



* 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...

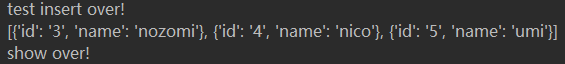
在create\_pool()后面添加：

|  |
| --- |
| 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：



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paramerters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', <paramsmeter "a">), ('b', <paramsmeter "b=0">), ('c', <paramsmeter "\*c">), ('d', <paramsmeter "d">), ('e', <paramsmeter "e=1">), ('f', <paramsmeter "\*\*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

middlerware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 |

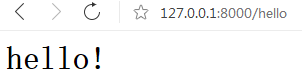
⑦ 测试：www/handlers.py存放视图函数，暂时不涉及数据库

|  |
| --- |
| 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…

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；默认的配置文件应该完全符合本地开发环境；这样无需任何设置，就可以立刻启动服务器。

-------------------------------------------------------------------------------------------------------

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middlerware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> |

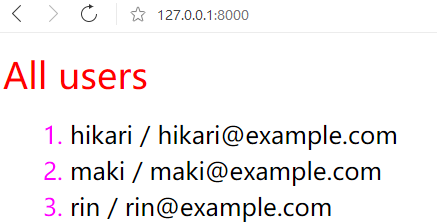
|  |
| --- |
| <h1 class="users">All users</h1> <ul>  {% for u **in** users %}  <li><span class="idx">{{ loop.index }}. </span>{{ u.name }} / {{ u.email }}</li>  {% endfor %}</ul> |

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：



* 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](https://getuikit.com)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> |

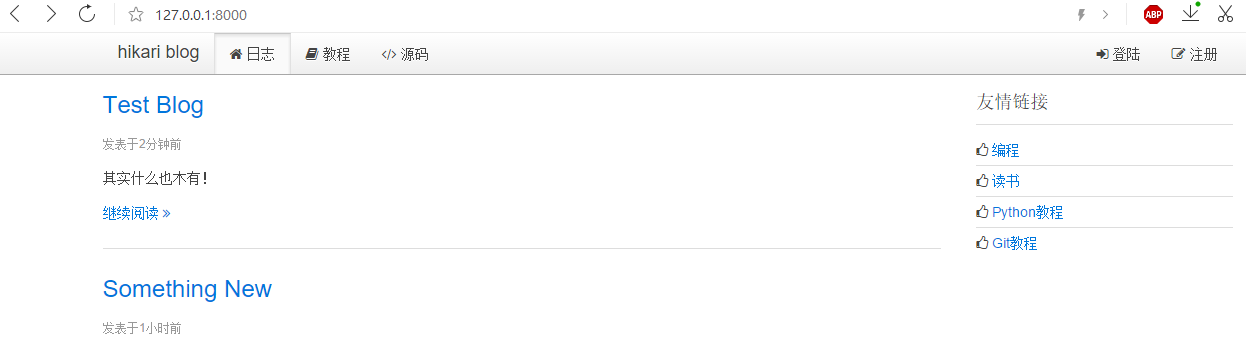
② 首页myblog.html继承于base.html：

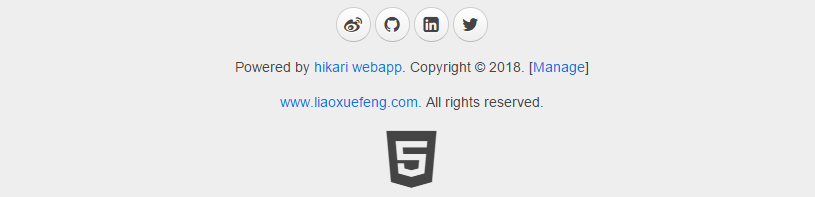
|  |
| --- |
| {% extends 'base.html' %} {% block title %}日志{% endblock %} {% block beforehead %}<script></script>{% endblock %} {% block content %}  <div class="uk-width-medium-3-4">{# 栅格化?左边占3/4, 右边1/4? #}  {% for blog **in** blogs %}  <article class="uk-article">  <h2><a href="/blog/{{ blog.id }}">{{ blog.name }}</a></h2>  {# 此处datatime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 = '其实什么也木有！'  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：

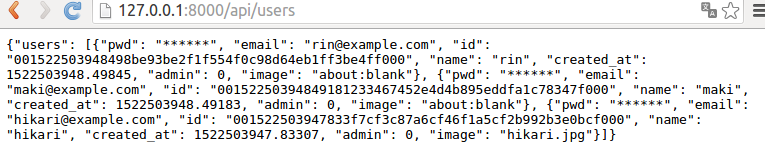
① 创建虚拟环境：mkvirtualenv -p /usr/bin/python3.5 hikari\_blog

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：



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：

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> </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="电子邮件" 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reques；获取的user，如果为None显示登录注册，否则显示用户名和登出。

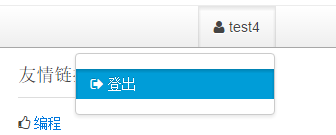
是不是该去学一下Vue.js了?

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);  });  }  **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> |

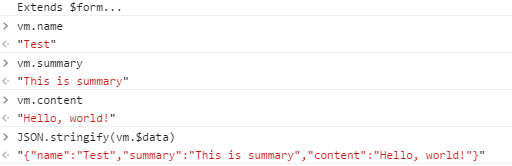
④ handlers.py：

|  |
| --- |
| @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：



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,  } |



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：





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 MyFileSystemEventHander(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(MyFileSystemEventHander(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">  <img class="uk-comment-avatar uk-border-circle" width="50" height="50" src="{{ \_\_user\_\_.image }}">  <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">  <img class="uk-comment-avatar uk-border-circle" width="50" height="50" src="{{ comment.user\_image }}">  <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">  <img class="uk-border-circle" width="120" height="120" src="{{ blog.user\_image }}">  <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](http://aws.amazon.com/)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](https://www.virtualbox.org/)。

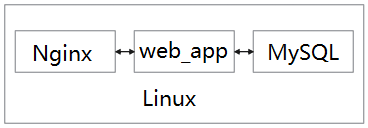
Linux安装完成后，确保ssh服务正在运行，否则需要通过apt安装：

sudo apt-get install openssh-server

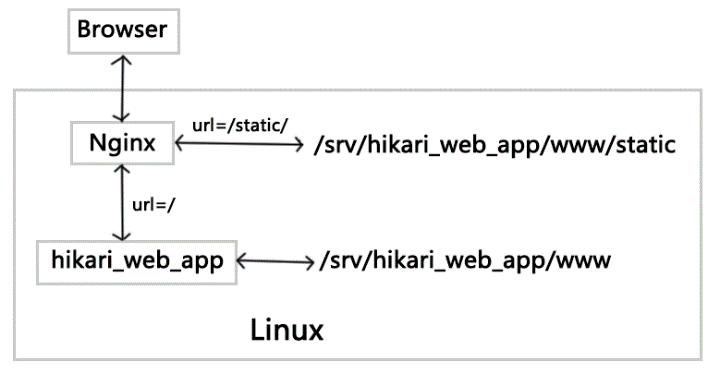
有了ssh服务，就可以从本地连接到服务器。建议把公钥复制到服务器端用户的.ssh/authorized\_keys中，就可以通过证书实现无密码连接。

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](http://supervisord.org/)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](http://www.fabfile.org/)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](http://cygwin.com/)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 -xzvf {}'.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管理的服务了：

$ 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](http://mirrors.163.com/)和[搜狐](http://mirrors.sohu.com/)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](https://developer.apple.com/technologies/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

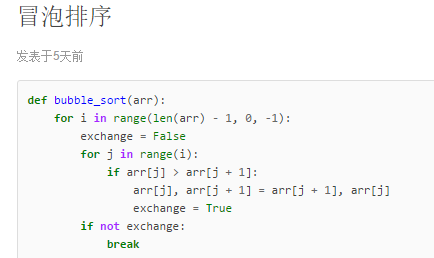
命令行：pygmentize -S default -f html > default.css

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：



* 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> |