my-web-app 实验报告

16337339 朱祎康

today

Contents

1	项目简介	1
2	项目结构	1
	2.1 基础架构	
	2.1.1 ORM	
	2.1.2 Web 框架	
	2.1.3 Middleware 中间件	
	2.2 后端 API	
	2.3 前端页面	5
3	实验心得	10
4	参考资料	10

1 项目简介

该项目使用 python 语言以及一些基础的通信架构,实现了自己的 web 框架,并且搭建了一个简单博客网站。

2 项目结构

整个项目主要包括三个部分:

- 1. 基础架构
- 2. 后端 API
- 3. 前端 html 页面

下面详细地对各个部分进行说明。

2.1 基础架构

由于考虑到大量页面访问的效率问题,所以使用的是异步 IO 机制,全部使用携程实现。

使用到了已有的 aiohttp 异步框架以及 mysql 的异步驱动程序 aiomysql,在此基础上实现了自己的 web 框架。

2.1.1 ORM

orm (Object Relational Mapping) 对象关系映射是建立起 python 语言与 mysql 之间的对象的相互转换以及操作。 mysql 数据类型在 python 中的抽象:

```
class Field(object):
    def __init__(self, name, column_type, primary_key, default):
        self.name = name
        self.column_type = column_type
        self.primary_key = primary_key
        self.default = default
```

数据库的基本操作抽象为 select 和 execute:

```
async def select(sql, args, size=None):
   log(sql, args)
    global __pool
    async with __pool.get() as conn:
        async with conn.cursor(aiomysql.DictCursor) as cur:
            await cur.execute(sql.replace('?', '%s'), args or ())
            if size:
                rs = await cur.fetchmany(size)
            else:
                rs = await cur.fetchall()
        logging.info("rows returned: %s" % len(rs))
        return rs
async def execute(sql, args, autocommit=True):
    log(sql, args)
   global __pool
    async with __pool.get() as conn:
        if not autocommit:
            await conn.begin()
        try:
            async with conn.cursor(aiomysql.DictCursor) as cur:
                await cur.execute(sql.replace('?', '%s'), args)
                affected = cur.rowcount
            if not autocommit:
                await conn.commit()
        except BaseException:
            if not autocommit:
                await conn.rollback()
            raise
        return affected
```

定义模型 Model 的元类 ModelMetaclass:

```
class ModelMetaclass(type):
    def __new__(cls, name, bases, attrs):
        if name == 'Model':
            return type.__new__(cls, name, bases, attrs)
        table_name = attrs.get('__table__', None) or name

mappings = ... # 保存属性和列的映射关系
    fields = ... # 所有属性名
    primary_key = ... # 主键属性名
    escaped_fields = ... # 除主键外的属性名

attrs['__mappings__'] = mappings
    attrs['__table__'] = table_name
```

将 mysql 中的表格 table 映射为 python 中的 Model,并且操作之间的映射如下:

```
• select \rightarrow find or findNumber
```

- insert \rightarrow save
- update \rightarrow update
- delete \rightarrow remove

```
class Model(dict, metaclass=ModelMetaclass):
    def __init__(self, **kw):
        super(Model, self).__init__(**kw)
    @classmethod
    async def findAll(cls, where=None, args=None, **kw):
        find all objects by where clause
        sql = [cls.__select__]
        if where:
            sql.append("where")
            sql.append(where)
        rs = await select(" ".join(sql), args)
        return [cls(**r) for r in rs]
    async def save(self):
        args = list(map(self.getValueOrDefault, self.__fields__))
        args.append(self.getValueOrDefault(self.__primary_key__))
        rows = await execute(self.__insert__, args)
        if rows != 1:
            logging.warning("failed to insert record: affected rows: %s" % rows)
    ... # 其他操作: update, remove 等等
```

2.1.2 Web 框架

由于已有的 aiohttp 框架过于底层,所以这里对它进行一定程度的封装。

原有的 aiohttp 需要进行以下几个步骤:

1. 编写使用 @asyncio.coroutine 修饰的函数

```
@asyncio.coroutine
def handle_url_xxx(request):
    pass
```

- 2. 从 request 中获取所需要的参数
- 3. 构造 Response 对象

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

这样显得很麻烦,对它进行封装,封装的效果如下:

1. 处理带参数的 URL /blog/{id}

```
@get('/blog/{id}')
def get_blog(id):
    pass
```

2. 处理 query_string 参数通过关键字参数接受

```
@get('/api/comments')
def api_get_users(*, page='1'):
    pass
```

- 3. 函数返回值不一定是 web.Response, 可以是 int、str 等其他对象
- 4. 需要渲染模板,可以通过返回一个 dict 实现

```
return {
    "__template__": "blog.html",
    "blog": blog,
    "comments": comments
}
```

具体的封装方法,是通过实现一个 RequestHandler 类,来对 URL 处理函数进行封装,使其装换为一个返回 web.Response 的携程:

```
class RequestHandler(object):
    def __init__(self, app, f):
        self._app = app
        self._func = f
        ...

async def __call__(self, request):
        kw = ... # 获取参数

    ret = await self._func(**kw)
        return ret
```

注册一个 URL 处理函数:

2.1.3 Middleware 中间件

 $\operatorname{middleware}$ 是一种拦截器,在 URL 被对应函数处理前后先进行加工处理,可以改变函数的输入、输出。本质上是位于前后端之间的一类函数。

例如:

• logger_factory: 用于在 URL 处理函数工作前,打上 log。

```
async def logger_factory(app, handler):
    async def logger(request):
    logging.info("Request: %s %s" % (request.method, request.path))
```

```
return (await handler(request))
return logger
```

• response_factory: 用于 URL 处理函数工作后,将结果加工返回。

```
async def response_factory(app, handler):
    async def response(request):
    logging.info("Response handler...")
    ret = await handler(request)
    if isinstance(ret, web.StreamResponse):
        return ret
    if isinstance(ret, bytes):
        tmp = web.Response(body=ret)
        tmp.content_type = "application/octet-stream"
        return tmp
    ... # 返回值为其他类型

tmp = web.Response(body=str(ret).encode("utf-8"))
    tmp.content_type = "text/plain; charset=utf-8"
    return tmp
    return response
```

2.2 后端 API

后端 API 如下:

```
获取日志: GET /api/blogs
创建日志: POST /api/blogs
修改日志: POST /api/blogs/{id}
删除日志: POST /api/blogs/{id}/delete
获取评论: GET /api/comments
创建评论: POST /api/blogs/{id}/comments
删除评论: POST /api/comments/{id}/delete
创建用户: POST /api/users
```

2.3 前端页面

前端使用uikit框架,并且使用 jinja2 模板进行渲染。

• 获取用户: GET /api/users

前端页面如下:

• 首页: GET /

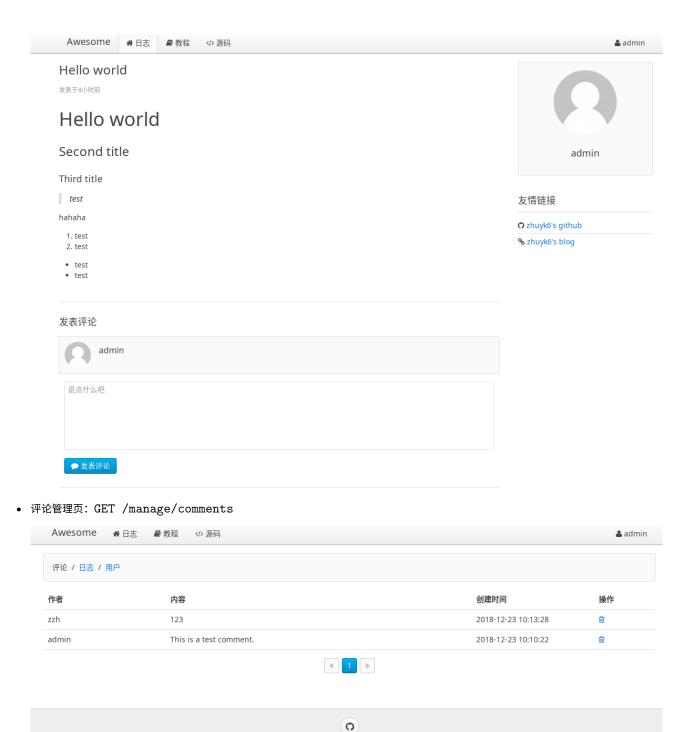


• 登录页: GET /signin

Awesome Python Webapp

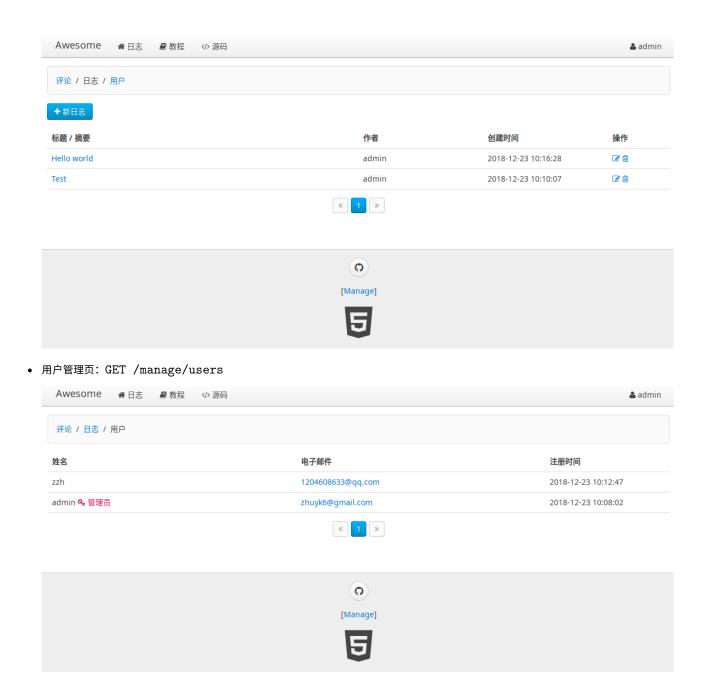


• 日志详情页: GET /blog/{id}

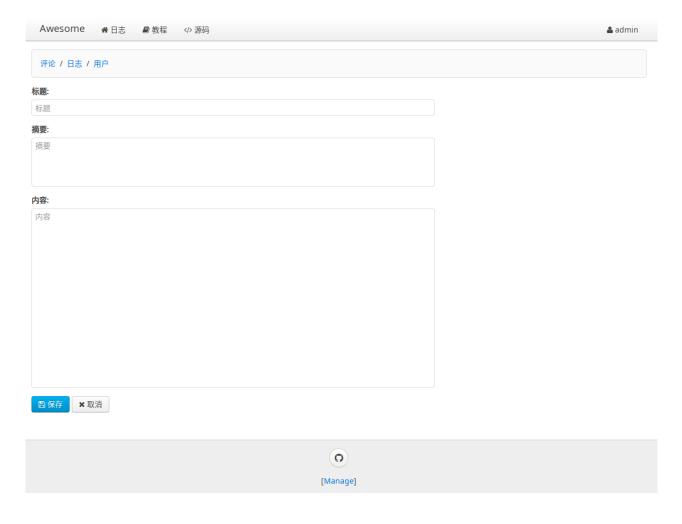


• 日志管理页: GET /manage/blogs

[Manage]



• 日志创建(修改)页: GET /manage/blogs/create



3 实验心得

- 1. 本次实验是我第一次写有关 web 的程序,学到了很多 web 的相关知识,例如:如何写前端 html 页面、javascript 代码,如何使用 css 框架。
- 2. 加深了对于 web 的理解,了解到了前后端是如何结合在一起良好工作的。
- 3. 学会了书写携程代码,异步 IO。
- 4. 学会了如何在 python 中使用数据库 mysql。
- 5. 也学会了很多很多关于 python 语言的技巧: 元类、修饰器、ducktype 等等。

4 参考资料

- [2] http://www.runoob.com/html/html-tutorial.html
- [3] https://getuikit.com/docs/introduction
- [4] https://cn.vuejs.org/index.html