框架使用说明

本框架集成最常见的开发流程、模式，旨在以最少的侵入度，约束团队以统一的开发规范进行API开发，减少因个人编码习惯差异导致一系列问题。将重复的工作交给框架，利用框架内置的能力实现快速开发，同时对质量可控。

只需稍加配置，即可快速启动一个web服务

一、快速开始：

```
# 安装：安装依赖包较多，时间较长
pip install lcyframe
# 创建项目
lcyframe my_project
# 启动项目
cd my_project && python app_start.py
```

二、文件说明：

my_project：

- - context/：环境配置
- __init__.py 开发、测试、生成环境对应的配置文件逻辑在这里实现

- - api_schame/：定义API参数、返回结构。所有api开发都从这里开始

- - handler/：业务逻辑代码

- - model/：数据模块代码
- -shema/：数据模型，所有数据库表定义都在这里

- - producer/：消息队列生产者
- - mqtt/
- event.py 事件定义

- - works/：消息队列消费者
- - mqtt/
- event.py 一对多订阅事件
- - celery/
- task.py 演示代码

- - utils/：工具集合
- helper.py 项目工具助手
- keys 缓存key定义文件
- constant.py 常量定义文件
- errors.py 异常码定义文件

- - service/：依赖服务定义，比如新增一个框架未内置的orcale数据库封装

- - SDK/：第三方代码
- - test_script/：测试脚本
- - logs/：请求日志目录，由配置文件指定
- base.py 项目基类
- app_start.py app服务文件
- celery_start.py 队列服务文件

三、演示用户注册、查看的API开发步骤

1、定义shema：`api_schema/user.yml`

```
- model：用户模块，每一个yml文件只能包含一个'model'标签。该标签将作为一级菜单显示在开发文档左侧导航菜单中

  apis: "/user"     # url定义，最终访问形式类似：http://www.domain.com/user
  name: "用户基本"  # 模块名称 二级菜单
  description: 用户 # 模块描述
  handler: UserHandler  # Handler类名称，通常只需写前缀'User'，框架在生成时自动补齐为'UserHandler'
  method:                             # 定义api的方法，允许：get、post、、delete
    post:                             # 方法名
        summary: 注册用户             # 功能名称
        description: 新增一个用户     # 功能描述
        parameters:                   # 参数定义，以下带*的为必须设置
          - name: phone               # * 参数名
            description: 手机号       # * 说明
            required: false           # * 是否必填 true、false
            type: int                 # * 参数类型：int、string、float、list、json、file
            default: 1                # 默认值，当required=false时生效
            regex: xxxxxx             # 正则表达式,如 ^(0|86|17951)····|14[57])[0-9]{8}$
            message: "手机号不符合规则" # 当参数值不满足正则表达式时，自动抛出该错误提示
          - name: username
            description: 用户名
            required: true
            type: string
          - name: password
            description: 密码，加密后传输
            required: true
            type: string
          - name: nickname
            description: 昵称
            required: true
            type: string
```

```
        responses:                      # 返回结构中data字段的定义，{"code":0,"data":
responses}
          {
            "uid": 349|int,             # 349|int 数字代表返回值，'|int'代表值类型为整形。
            "username": "woshishui",    # 末尾不含|str,框架自动用当前的值类型作为输出。即字符串类
型
            "nickname": "woshishui|str",
            "sex": 0女1男|int,          # 将值当做描述或者枚举，以便前端更好理解，必须标明类
型，|int
            "address": "地址",
            "phone": "13888888888",
            "email": "",
            "company": "企业名称",
            "state": 1 正常 -1 冻结,
            "create_at": 1634217171,
            "head_img": "http://www.xxx.com/2ed024d6d44a0d706776fac4495659d9.png"
          }

    get:
        summary: 查看
        description:
        parameters:
          - name: uid
            in: query
            description: uid
            required: true
            type: integer
        responses:
          {
            "uid": 349,
            "username": "1cylln",
            "nickname": "",
            "sex": 0女1男,
            "address": "地址",
            "phone": "13888888888",
            "email": "",
            "company": "企业名称",
            "state": 1 正常 -1 冻结,
            "create_at": 1634217171,
            "head_img": "http://www.xxx.com/2ed024d6d44a0d706776fac4495659d9.png"
          }
```

当定义完成后，重新启动: `python app_start.py`,框架会自动生成对应的handler、model、schema文件。

这时候我们请求查看一下用户详情：`http://www.domain.com/user?uid=100`

```
{
    code: 200,          # 全局约定，在基类base.py中可以自定义状态值
    msg: "OK",          #
```

```
    data:  {          # 在user.yml response中定义的输出结构
        uid: 100,
        username: "admin",
        nickname: "admin",
        sex: 1,
        address: "",
        phone: "13688888888",
        email: "",
        company: "易华有限公司",
        state: 1,
        create_at: 1636599201,
        head_img: "http://0.0.0.0:6600/data/images/head/9be22df595967d39e9.png"
    }
}
```

2、schema 数据模型层： `model/schema/user.py`,该文件由框架生成。数据库字段定义、分表逻辑在这里配置，基于mongo

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

from bson.objectid import ObjectId
from base import BaseSchema
from lcyframe.libs import utils

class UserSchema(BaseSchema):
    """
    user表
    """

    collection = "user"                 # 表名称
    is_shard = False                    # 是否分表，True时，自动按照分表存储，整个过程对业务透
明
    shard_key = ""                      # 分表字段，当is_shard=True时有效

    def __init__(self):                 # 以下字段可以通过model.fields()获取
        self._id = ObjectId()
        self.uid = 0
        self.username = ""
        self.password = ""
        self.salt = utils.gen_salt(6)
        self.nickname = ""
        self.sex = 1
        self.head_img = ""
        self.gid = 1
        self.address = ""
        self.phone = ""
        self.email = ""
        self.company = ""
```

```
        self.state = 1      # 1 正常 -1 禁止
        self.create_at = int(utils.now())

    @classmethod
    def shard_rule(cls, shard_key_value):     # 默认分表方法，模10。当is_shard=True时有效
      return cls.mod10(shard_key_value)
```

3、Handler 业务逻辑层 `handler/user.py`,该文件自动生成

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
import os
from lcyframe import route
from lcyframe import funts
from base import BaseHandler, helper
import re

@route("/user")
class UserHandler(BaseHandler):
    """
    用户基本
    """

    @helper.admin()
    def post(self):
        """
        用户注册
        :return:
        :rtype:
        """
        # self.params 包含所有参数与值，类型dict。注意，不在schema中定义的参数，不在此列
        # self.model 为全站所有模块的挂载点，以user.yml名称作为前缀，即可在app所在的命名空间内调用
        # 例如：调用用户模块的方法为：self.model.UserModel.function(**args)

        data = self.model.UserModel.create(**self.params)
        if data == -1:
            raise self.api_error.UserCommon("用户和密码不能为空")  # 抛出内置异常，且自定义说
明


        if data == -2:
            raise self.api_error.UserExists                    # 抛出内置异常

        self.write_success(data)      # 响应成功，data按照你在schema.response定义的格式输出

    @helper.admin()
    def get(self):
        """
        个人资料
```

```
        :return:
        :rtype:
        """

        data = self.model.UserModel.get(self.params["uid"])
        if not data:
            raise self.api_error.UserNotExists
        self.write_success(data)
```

4、model 数据层：`model/user.py`，该文件自动生成。所有数据操作应该在这里完成，且要保证复用性

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
import os
from base import BaseModel
from model.schema.user_schema import UserSchema


class UserModel(BaseModel, UserSchema):   # 固定格式，以user.yml名称作为前缀
    # 以下几个方法由框架自动生成，你也可以新增其他方法，通常需要事先指定某种约定

    @classmethod
    def get(cls, uid):
        """
        单条记录
        :return:
        :rtype:
        """
        return cls._parse_data(cls.find_one({"uid": int(uid)}))

    @classmethod
    def get_user_list_by_page(cls, page, count, **kwargs):
        """
        列表
        :return:
        :rtype:
        """
        spec = {}                                                    #
        spec.update(kwargs.get("spec", {}))
        data_list, pages = cls.find_list_by_page(spec,               # 查询条件
                                                 page,               # 第几页
                                                 count,              # 每页显示条
数
                                                 sort=[("create_at", -1), ],# 排序字段
                                                 fields=False)       # 需返回的字
段
        return [cls._parse_data(d) for d in data_list if d], pages   # 清洗每一条
记录
```

```python
    @classmethod
    def create(cls, *args, **kwargs):
        """
        创建
        :return:
        :rtype:
        """
        password = str(cls.aes.decode(kwargs.pop("password")))              # 内置aes加密模块
        if not password or not kwargs["username"]:
            return -1

        if cls.find_one({"username": kwargs["username"]}):
            return -2

        docs = cls.fields()
        docs["uid"] = cls.model.IdGeneratorModel.gen_uid_id()               # 内置的id生成模块
        docs["password"] = cls.utils.gen_salt_pwd(docs["salt"], password) # 使用工具包utils
        docs.update(kwargs)
        cls.insert(docs)
        return docs

    @classmethod
    def modify(cls, uid, **kwargs):
        """
        修改
        :return:
        :rtype:
        """
        user = cls.get(uid)
        if not user:
            return None

        spec = {"uid": uid}
        update_docs = {}
        docs = cls.fields()                                                 # 索取表字段与值

        return cls.update(spec, update_docs)

    @classmethod
    def delete(cls, *args, **kwargs):
        """
        删除
        :return:
        :rtype:
        """
```

```
        pass

    @classmethod
    def _parse_data(cls, d, **kwargs):
        """
        组装数据、清洗
        :return:
        :rtype:
        """
        if not d:
            return d

        d["head_img"] = cls.helper.get_imge_url(
            cls.app_config["wsgi"]["host"],
            d["head_img"] or "default.jpg",
            os.path.join(cls.app_config["data_config"]["base"], "images/head")
        )
        return d

    # 更多方法
    """
    # 获取当前表所有字段
    fields = cls.fields()                       # 获取自身模块的表字段
    fields = self.model.DemoSchema.fields()     # 获取其他模块的表字段

    # mongo
    # 新增记录（1条或者多条）
    # 1、框架的方法
    # id = cls.insert({"uid": 104})
    # ids = cls.insert([{"uid": 102}, {"uid": 103}], ordered=False)   # ordered并发写入,
不保证顺序
    # 2、原生方法
    # cls.mongo.user.insert_one({"uid": 102})
    # cls.mongo.user.insert_many([{"uid": 102}, {"uid": 103}])

    # 删除多条
    # deleted_count = cls.remove({"uid": 103}, multi=False)
    # 删除单条
    # deleted_count = cls.remove({"uid": 102})

    # 查看
    # data = cls.find_one({"yyy": 102})
    # datas = cls.find({"uid": 102}, fields={"uid": 1}, skip=1, limit=10, sort={"uid":
1})

    # 替换/覆盖/新增  result.matched_count,result.modified_count,result.upserted_id
    # result = cls.replace_one({"uid": 102}, {"uidx": 102})                   # 修改字段
    # result = cls.replace_one({"xxx": 100}, {"xxx": 100}, upsert=True)       # 没有则新增
```

```
    # result = cls.replace_one({"xxx": 200}, {"yyy": 200, "zzz": 100}, upsert=True)
# 没有则新增一条，并新增一个字段

    # 更新数据
    # update_state = cls.update({"yyy": 200},cls.find {"$inc": {"xxx": 1}})
    # update_state = cls.update({"yyy": 200}, {"$inc": {"xxx": 1}}, multi=True)
    # <class 'dict'>: {'n': 6, 'nModified': 6, 'ok': 1.0, 'updatedExisting': True}
    # result = cls.update({"yyy": 200}, {"$inc": {"xxx": 1}}, multi=True,
check_updated_state=False)
    # 没有则新增
    # update_state = cls.update({"yyy": 203}, {"$inc": {"xxx": 1}}, upsert=True)

    # 原子操作
    # 更新,默认没有则新增，upsert=True，new=True返回更新后的文档
    # result = cls.find_and_modify({"yyy": 300}, {"$inc": {"xxx": 1}}, new=True)  #
    # 替换/覆盖/新增
    # result = cls.find_one_and_replace({"yyy": 300}, {"xxx": 1}, return_document=True)
    # result = cls.find_one_and_replace({"yyy": 400}, {"xxx": 1}, sort={"a": -1},
return_document=True)
    # result = cls.find_one_and_replace({"yyy": 500}, {"xxx": 1}, sort=[("a", -1)],
return_document=True)
    # 原子更新
    # result = cls.find_one_and_update({"yyy": 400}, {"$inc": {"xxx": 1}}, sort=[("a",
1)], return_document=True)
    # 原子删除
    # result = cls.find_one_and_delete({"yyy": 400}, sort=[("a", 1)])
    # result = cls.find_one_and_delete({"yyy": 400}, sort={"a": -1}, projection={"_id":
0})

    # 过滤重复记录
    # result = cls.distinct({"a": 2}, "yyy")
    # result = cls.distinct({"a": 2}, "_id")
    # result = cls.distinct({"a": 2}, "yyy", **{"maxTimeMS": 5000})

    # 统计result.upserted_id
    # count = cls.count()
    # count = cls.count({"uid": 102}, skip=10)

    # myqsl数据库
    # 插入一条或多条数据
    # cls.mysql.insert(cls.collection, [{"name": 2}, {"name": 3}])

    # 查询记录，按关键字传参
    # data = cls.mysql.query_sql(sql="select * from demo where name=(name)", params=
{"name": 2})
    # 查询记录，按顺序传参
    # data = cls.mysql.query_sql(sql="select * from demo where name=%s", params=[2, ])
    """
```

四、消息队列

1、生产者定义：

以celery为例，由于celery的设计模式，决定了生产者即为消费者，所以这里不需要单独定义生产者。

2、消费者定义：`works/celery/tasks.py`，手动创建celery目录，并新建task.py文件，写入示例代码

```
# -*- coding: utf-8 -*-
import random, time
from celery import shared_task
from lcyframe.libs.celery_route import BaseTask
from lcyframe.libs.celery_route import BaseEvent
from celery.utils.log import get_task_logger

logger = get_task_logger(__name__)

@shared_task(ignore_result=False)  # 必须使用shared_task注册任务
def func(x):                       # 简单的任务函数
    """
    任务函数
    """
    n = round(random.random(), 2)
    time.sleep(n)
    logger.info('Adding {0} + {1}'.format(x, "e"))
    return x, n

class DefaultEvent(BaseEvent):     # 以类为纬度的任务集合，通常是以业务线为单位的场景
    queue = "for_task_A"           # 指定消费队列名称，默认default

    @staticmethod                  # 必须以staticmethod装饰器
    @shared_task(bind=True)        # True时，self作为第一个参数自动传入，可以使用任务对象的属性
    def bind(self, x):
        """
        bind=True时，可以读取任务对象属性
        :param self:
        :param x:
        :return:
        """
        print(self.app)
        return x
```

3、使用队列：实现异步任务和调度

handler/user_handler.py：

```
self.celery.Events.func.delay(1)                # 以函数定义的任务
self.celery.Events.DefaultEvent.bind.delay(111) # 以类定义的任务
```

model/user_model.py：

```
cls.celery.Events.func.delay(1)                  # 以函数定义的任务
cls.celery.Events.DefaultEvent.bind.delay(111)  # 以类定义的任务
```