# 说明：

手册中有详细内容，详细内容查看手册

# 一、异步网络库使用

# 项目

## 极简项目

import tornado #引入库

from tornado import web #引入库

class IndexHandler(web.RequestHandler):

async def get(self):

pass

if \_\_name\_\_ == "\_\_main\_\_":

app = web.Application([

(r"/",IndexHandler)

])

app.listen(8000)

tornado.ioloop.IOLoop.current().start() #循环

## server.py

import tornado.web

import tornado.ioloop

# import tornado.options

import tornado.httpserver

## 路由配置

**/url.py**

from application.IndexHandler import Indexhandler

urls = [

(r”/”, IndexHandler)

(r”/d/(.\*)”, IndexHandler) #正则、提取路由种的参数

(r”/d/？”，IndexHandler) #路径最后是否添加/

(r”/d”,oneHandler, name=”index”) #通过name跳转

(r”/d”,two, value1,value2) # 传值

(r”/d/(:?P<name>\d)”, three) #起名

]

=====================

**/applincation/IndexHandler.py**

async def get(self, id): -->IndexHandler

self.write(id)

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

async def get(self): -->通过name跳转

# 跳转加传值

a = self.reverse\_url(“index”,”one”,...))

self.redirect(a)

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

async def two(self):--> 传值

self.write(self.data) #通过initialize方法设定的值

def initialize(self, data): # 接受路由配置的值

#不定义成协程

self.data = data

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

async def three(self, name)-->起名

#name与url中起别名一样

pass

=====================

**/server.py**

from url import urls # 导入路由

app = tornado.web.Application( urls ) #加载路由

## 4、命令行参数配置

全局只有一个option，可以

/server.py

from tornado.options import define, option

define(“port”, type=int, default=8000, help=”端口”)

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

**option.parse\_command\_line() #获取命令行参数**

----二选一-------

# parse\_config\_file只能读取文件中设置的内容

# 必须有define() 定义后才会生效，基本无用

option.parse\_config\_file(“config.cfg”) #读取文件配置

**/config.cfg**

**port=8000**

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

命令行

**python server.py --port=8000**

## requestHandler

from tornado.web import RequestHandler

class IndexHandler(RequestHandler):

def initialize(self):

# 初始化类

def prepare(self):

# 用于真正调用请求处理之前的初始化方法

# 所有请求都经过这个方法

def write\_error(self):

# 重新-指定错误页面

def set\_default\_headers(self):

self.set\_header( # 设置请求头

“Content-Type”,”application/json;charset=UTF-8”)

def on\_finish(self):

# 当请求结束时调用

# 关闭句柄，清理内存等

def get\_current\_user(self):

**# 通过session判断用户是否为成功登录**

*@property #作为属性，self.db即可调用*

*def db(self):*

*return self.application.db*

## setting.py

**/setting.py**

import tornado

import redis

# import pymysql

class **Application**(tornado.web.Application）：

def \_\_init\_\_(self, \*args, \*\*kargs):

super(Application, self).\_\_init\_\_(\*args, \*\*kargs)

# self.db = pymysql.Connect( //连接mysql

# \*\*mysql\_options

# )

self.redis = redis.StrictRedis( //连接redis

\*\*redis\_options // 解包

)

# 配置application参数

setting = dick(

“static\_path”: 路径,

“template\_path”: 路径,

“debug”:True, #bug

“cookie\_secret”:”安全字符串uuid”,

“xsrf\_cookie”:”True”

)

# 配置mysql

**mysql\_options** = dict(host=”127.0.0.1”, database=”home”)

# 配置redis

**redis\_options** = dict(host=”127.0.0.1”, port=”6369”)

log\_file = “路径” // 日志位置

session\_expires\_seconds = 86400 #session过期时间

## server.py

import tornado.web

import tornado.ioloop

import tornado.httpserver

from url import urls // 导入同级目录下的路由文件

# from config import mysql\_options //导入配置文件

from config import redis\_options //导入配置文件

from config import settings

from config import Application //导入类

// 导入option，主要用于获取 命令行 参数

from tornado.options import define, option

define(“port”, type=int, default=8000, help=”端口”)

def main():

options.logging = configs.log\_level # 设置日志等级

# 设置读取配置文件中的

options.log\_file\_prefix = configs.log\_file

#读取配置参数 tornado必要

tornado.options.parse\_command\_line()

app = Application(urls, \*\*settings)

http\_server = tornado.httpserver.HTTPServer(app) #多进程

http\_server.listen( options.port ) #简写

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

**app.listen(8000)** 相当于：<单进程>

http\_server = tornado.httpserver.HTTPServer(app)

http\_server.bind(8000)

http\_server.start(1)

**http\_server.listen(8000)** 相当于：<单进程>

http\_server.bind(8000)

http\_server.start(1)

<多进程>

http\_server = tornado.httpserver.HTTPServer(app)

http\_server.bind(8000)

http\_server.start(0) # 少于或等于0，根据CPU核数开启进程

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

tornado.ioloop.IOLoop.current().start() **#主循环**

# 项目开启

## 4、/handler/BaseHandler.py

**from tornado.web import RequestHandler**

**from util import SESSION #导入session文件**

class BaseHandler(**RequestHandler**):

**@property #作为属性，self.db即可调用**

def db(self):

return self.application.db

**@property #作为属性，self.redis即可调用**

def redis(self):

return self.application.redis

def prepare(self):

self.xsrf\_token //手动触发一下 ,放大下面执行

if self.request.headers.get(“Content-Type”, “”)

.startwith(“application/json”):

self.json\_args = json.loads(self.request.body)

else:

self.json\_args = None

def write\_error(self):

def set\_default\_headers(self):

self.set\_header( # 设置请求头

“Content-Type”,”application/json;charset=UTF-8”)

def initialize(self)

def on\_finish(self):

def get\_current\_user(self):

**# 通过session判断用户是否为成功登录**

self.session = session(self)

return self.session.data

class StaticFileHandler(tornado.web.StaticFileHandler):

def \_\_init\_\_(self, \*args, \*\*kargs):

super(StaticFileHandler, self).\_\_init\_\_(\*args,\*\*kargs)

self.xsrf\_token

## 6、日志功能

from tornado.options import option

options.logging = configs.log\_level # 设置日志等级

options.log\_file\_prefix = configs.log\_file # 设置读取配置文件中的

import logging

logging.error(“输出信息”)

logging.debug(“输出信息”)

logging.info(“输出信息”)

logging.waring(“输出信息”)

tail -f 日志名 //cmd中查看

# 生命周期

无异常

1. set\_default\_headers
2. initialize
3. prepare
4. get/post/patch

有异常

set\_default\_headers

write\_error

on\_finish

无异常

on\_finish

对于一个请求的处理过程代码调用次序如下：

1. 程序为每一个请求创建一个 **RequestHandler** 对象；
2. initialize() 函数，这个函数的参数是 Application 配置中的关键字参数定义。

程序调用 ，。（initialize 方法是 Tornado 1.1 中新添加的，旧版本中你需要重写 \_\_init\_\_ 以达到同样的目的） initialize 方法一般只是把传入的参数存到成员变量中，而不会产生一些输出或者调用像 send\_error 之类的方法。

程序调用 prepare()。无论使用了哪种 HTTP 方法，prepare 都会被调用到，因此这个方法通常会被定义在一个基类中，然后在子类中重用。prepare可以产生输出信息。如果它调用了finish（或send\_error` 等函数），那么整个处理流程就此结束。

程序调用某个 HTTP 方法：例如 get()、post()、put() 等。如果 URL 的正则表达式模式中有分组匹配，那么相关匹配会作为参数传入方法。

# 五、接收参数

self.get\_query\_argument(name) #从url获取值

url：[www.xx.com/a?key1=value1&key2=value2](http://www.xx.com/a?key1=value1&key2=value2)

self.get\_body\_argument(name) #获取表单

HTTP报文头Header中的"Content-Type"为以下任意一种

============

application/x-www-form-urlencoded #POST方式

multipart/form-data #POST方式

=============

# 获取**get\_body\_argument**与**get\_query\_argument**的参数

self.get\_argument(name) #获取所有数据，不特别指定header

**要想获取json，需要使用self.request.bod**

## 1、内置方法

#获取key为name的值，strip去掉空格

**self.get\_body\_argument(name, default=””, strip=True)**

#获取多个值

**self.get\_body\_arguments(name, strip=True)**

#获取key为name的值，strip去掉空格

**self.get\_query\_argument(name, default=””, strip=True)**

#获取多个值

**self.get\_query\_arguments(name, strip=True)**

**-------------------推荐使用一下方法--------------------**

获取url / form 表单传来的值，如有相同的变量名，**获取最后一个**

self.get\_argument(name, default=””, strip=True)

self.get\_arguments(name, strip=True)

## 2、上传

**发送请求的格式：**<multipart/form-data>

**接收方法：**

self.request.files # 这是一个列表

self.request.files[‘key’][0][‘body’] # 这是上传的文件

**通过 get\_argument也可获取**

## 3、json

**发送请求的格式：**<application/json>

**服务器端，判断格式：**

self.request.headers.get(“content-type”)

.startswith(“application/json”) #判断格式为json

json\_data = json.loads(self.request.body) #json序列化加接收

## 4、RequestHandler.request

对象存储了关于请求的相关信息，具体属性有：

**method：**HTTP的请求方式，如GET或POST;

**host**：被请求的主机名；

**uri：**请求的完整资源标示，包括路径和查询字符串；

**path：**请求的路径部分；

**query:**请求的查询字符串部分；

**version**:使用的HTTP版本；

**headers:**请求的协议头，是类字典型的对象，支持关键字索引的方式获取特定协议头信息， 例如：request.headers["Content-Type"]

**body:**请求体数据；

**remote\_ip:**客户端的IP地址；

**files:**用户上传的文件，为字典类型，型如：

## 5、返回数据

self.writ()

self.set\_status()

# 配置静态文件-响应图片

1. **获取静态文件的路径**

path = os.path.join(os.path.dirname(\_\_file\_\_), ”../static”)

1. **设置静态文件路径 : config.py-setting**

“static\_path”: path

“static\_url\_prefix”:”/static” # 隐藏真实url，与url中路径配置相同

**3、初始化 ： server.py**

from config import setting

tornado.web.Application(\*\*settings)

**4、配置路由： url.py**

from tornado.web import StaticFileHandler

url = [

(r”/static/(.\*)”, StaticFileHandler)#静态文件访问

]

# 模板配置

## 1、基础配置

### （1）设置模板

**/setting.py**

import os

setting = {

“template\_path”:os.path.join( # 模板路径，相对路径

os.path.dirname(\_\_FILE\_\_),”template”),

}

**/server.py**

tornado.web.application(\*\*setting)

### （2）使用变量标注位置

**## 需要配置静态文件路径 ##**

**/static/index.html**

{{static\_url(“css/index”)}}/xxx/js

### （3）设置render

**/controller/indexHandler.py**

indexHandler(self):

self.render(“index.html”) #加载模板

### （4）非必要设置

* **使用变量**

**/controller/indexHandler.py**

index(self):

# a,b是关键key

self.render(“index.html”,a=11,b=22...)

**/controller/index.html**

<h1>{{a}}</h1>{{b}}

* **index.html中使用python代码**

{% python代码 %}

## 模板重载/继承

### （1）新建base.html

{% block body %} 这里写默认显示内容 {% end %}

### 继承父页面的子页面

# 替换掉base.html页面中body段

{% extends ‘base.html’ %}

{% block body %}

html代码

{% end %}

## UImodel

### 自定义UIModel

Class OrderModel(tornado.web.UIModule):

def render(self, order, \*args, \*\*kargs):

self.render\_string(“uimode/origin.html”)

def one(self):

#这里时uimode/origin.html中调用的方法

### 修改配置文件

/setting.py

# 引入model

setting = {

“ui\_modules”:OriginModel #设置到全局

}

### 静态文件添加

{% module OriginModel() %}

# 允许跨域

先发送一个OPTIONS请求，游览器会返回允许的header，

set\_headers

继承这个handler，会默认网客户端写cookie

class BaseHandler(RequestHandler):

def set\_default\_headers(self):

#任何站点

self.set\_header(“Access-Control-Allow-Orign”,”\*”)

#任何头

self.set\_header(“Access-Control-Allow-Handers”,”\*”)

self.set\_header(“Access-Control-Max-Age”, 1000)

self.set\_header(“Content-type”,”application/json”)

self.set\_header(“Access-Control-Allow-Methods”,”POST,GET,OPTIONS”)

self.set\_header('Access-Control-Allow-Headers',

'Content-Type, **tsessionid**, Access-Control-Allow-Origin, Access-Control-Allow-Headers, X-Requested-By, Access-Control-Allow-Methods')

因为跨域发送的请求是OPTIONS

**Request Method:** OPTIONS

def options(self):

pass

# 验证码

描述：生成图片验证码

Url

method：get

@param codeID int 验证码编号

前端发送验证码编号，后端接收，生成验证码，保存编号，与文本

编号放在前端，是防止刷新二维码

import logging

from .BaseHandler import BaseHandler

class ImageCodeHandler(Basehandler):

def get(self):

code\_id = self.get\_argument(“codeID”)

pre\_code\_id = self.get\_argument(“pcodeID”)

if pre\_code\_id:

try:

self.redis.delect(“code\_id”)

except Exception as e:

logging.error(e) # redis错误

// 调用验证码类

// 存放redis中缓存

// 返回前端

# websocket

## 以前的方式

前端轮询（有无数据，服务器都响应）、长轮询（有数据服务器响应）、websocket

## 原理

-->开始使用http报文(使用101状态码)

-->升级为websocket

## 使用方法

websocket没有get、post等请求

websocket中有request有http的request属性

self.request.remte\_ip # 客户端IP

1. **引入/继承**

from tornado.websocket import webSocketHandler

class IndexHandler(websocketHandler)

1. **创建类属性**

ws\_opt = [] //类属性，保存连接用户，全局

1. **创建连接属性**

def open( self ): # 方法名，固定，class Indexhandler

for user in self.ws\_opt:

ws\_opt.write\_message(“这是广播”)

# 将当前对象放入，因继承 websocketHandler，所有websocket方法 # 都可使用

self.ws\_opt.append( self )

1. **发送信息**

def on\_message(self, msg): #固定方法 class Indexhandler

1. **关闭**

def on\_close(self): # class Indexhandler

self.ws\_opt.remove(self) # 用户关闭连接后从容器中移除用户

1. **跨域名/跨源**

def check\_origin(self) # class Indexhandler

return true # 允许跨源

# SESSION

通过操作redis实现session方法，

## 1、导入

import uuid

import loggin

import json

import config

## 2、类

class Session (object):

def \_\_init\_\_(self, request\_handler):

self.request\_handler = request\_handler

self.session\_id = self.request\_handler.

get\_secure\_cookie(“session”)

if not self.session\_id:

# 用户第一次访问，生成一个ID

self.sessin\_id = uuid.uuid4().get\_hex() //唯一

self.data = {}

else:

# 获取session

try:

data = self.redis.get(“sess\_%s” % session\_id)

except Exception e:

loggin.error(e)

self.data = {}

if not data:

self.data = {}

else:

self.data = json.loads(data)

## 3、保存

def save(self):

json\_data = json.dump(self.data) # 序列化

try:

self.redis.setex(“sess\_%s” % json\_data, config.session\_expires, json\_data)

except Exception e:

logging.error(e)

raise Exception(“save session failed”)

## 4、删除

def clear(self):

#删除客户端的cookie

self.request\_handler.clear\_cookie(“session\_id”)

try:

self.redis.delete(“sess\_%s” % self.session\_id)

except Exception as e:

logging.error(e)

## 5、使用

def get(self):

self.session = session(self)

# authenticated用户登录验证

## 同步方法实现：

**/controller/index.py**

1. **添加装饰**

@tornado.web.authenticated

index(self):

**2、定义get\_current\_usr方法**

def get\_current\_usr(self):

< @tornado.web.authenticated会自动调用本方法，返回真，验证成功，返回假验证失败>

**3、定义登陆失败后的路由**

setting.py中：

login\_url=”/login”

<url.py> #路由中设置

(r“/login”, loginHandler) # 自定义跳转控制器

## 异步装饰器

**查看authenticated\_async方法**

# 跨站请求伪造(xsrf)

## 使用cookie

### 使用模板

**（1）配置setting.py**

cookie\_secret = “uuid生成的字符串”

xsrf\_cookie = true # 开启

**（2）/static/index.html**

{%module xsrf\_form\_html() %}

### ajax / 不使用模板

set\_secure\_cookie

get\_secure\_cookie **（1）配置setting.py**

cookie\_secret = “uuid生成的字符串”

xsrf\_cookie = true # 开启

**（2）/controller/indexHandler.py**

def get(self):

self.xsrf\_token

**（3）/application**

class staticFileHandler(tornado.web.StaticFileHandler):

def \_\_init\_\_(self, \*args, \*\*kargs):

super(staticFileHandler.self).\_\_init\_\_(\*args, \*\*kargs)

self.xsrf.token

## jwt验证

因前后端分离，所有的登录都是通过ajax发送，在get获取页面（无登陆）的时候，没有设置cookie字段

**\*\*\*\* 首次请求时就应把jwt的code写入前端页面或cookie中 \*\*\*\***

### 生成-服务器端

1. **安装库，引用库**

pip install pyjwt

import jwt

**2、handler获取参数**

param = self.request.body.decode（”utf-8“）

param = json.loads(param)

**3、验证**

使用wtform验证

**4、组成参数**

payload = {

“id”:user.id, # 可设置其他使用参数

**“exp”:datetime.utcnow() #过期时间**

}

# 在setting文件下设置secret\_key字段

secret\_key = “密钥（随机）”

token=jwt.encode(**payload**,**secret\_key**,**algorithm**=”HS256”)

**5、返回**

**# 使用url访问可通过cookie设置，**

**# 通过ajax获取的不能设置cookie，要手动设置**

self.write(token)

### 服务器端解析jwt

import jwt

jwt.decode(

tokenid, **# 通过get或其他方式返回到前端的tokenid**

self.settings["secret\_key"], # setting.py中设置的secret\_key

**leeway**=self.settings["jwt\_expire"], # setting.py中设置的过期时间

**options**={"verify\_exp": True} # 开启验证过期  
)

## 前端配合

**1、使用axios**

let that = this

axios.post(“/login”,{

“id”:111, // 携带的参数

“tokenid”: getCookie(“\_xsrf”) // 获取cookie

}).then((res)=>{

that.$cookies.set(“token”, res.token) // 设置cookie

location.href = “./index.html” // 设置跳转

}).catch(function(err){

console.log(err)

}

1. **使用jquery**

**/index.html 前端**

$.ajax({

contentType:”application/json”,

headers:{ // 请求头一定要携带

“X-XSRFTOKEN”:getCookie(“\_xsrf”) }

})

在用户第一次登录首页时，先判断登录状态ajax--check\_login

# 异步

* tornado的异步方法提高了并发效果，但每一个用户的访问速度并未提升
* 在操作数据库中必须使用异步类才能实现非阻塞（aiomysql)

## 1、异步客户端

* **异步发送请求 （POST）**

# 要想发挥异步请求的全部效能，**client.fetch**请求的方法必须是**另一个**项目，不能是# 当前的tornado，如果要请求当前的项目，client.fetch一样会**同步堵塞**

**import urllib**

**from tornado.httpclient import HTTPRequest**

async def get(self):

data\_send = urllib.parse.urlencode(data) #编码转换

#定义 请求对象

req = tornado.HTTPCline.HTTPRequest(

**url**=url,**method**=”POST”,**body**=data\_send,**header**={}

);

client = AsyncHTTPClient() #异步客户端

response = await client.fetch(req) #异步请求方法;a

* **异步发送请求（GET）**

from tornado.httpclient import AsyncHTTPClient

async def asynchronous\_fetch(url):

http\_client = **AsyncHTTPClient()**

response = await http\_client.**fetch**(“www.baidu.com”)

return response.body

## 2、aiomysql 异步mysql

异步读存数据库操作

**/setting.py**

db = dick(

“host”=127.0.0.1,

“user”=”root”,

“password”=””,

“name”=”miaomu”,

)

## 3、peewee-async

### （1）安装

安装 --pre 安装0.6.0以上版本

pip install peewee-async

### （2）配置

**（1）/config/models.py**

import peewee\_async

**# 连接数据库**

**datebase** = peewee\_async.MySQLDatabase(

host,database,user,port,password )

**（2）/config/application.py**

**# 导入**

from .models import database

**# 设置不允许异步**

database.set\_allow\_sync(False)

**# 将database放入application中**

self.db = peewee\_async.Manager(**database**)

**（3）/apps/index/model.py**

**#默认使用建立baseModel, database选项可建立在baseModel**

**#不继承BaseMode，必须每一个model都设置database项，指定数据库**

**from config.models import baseModel**

class GOODS(**baseModel**): # 需要建立baseModel

....

class Meta:

database = **database #指定数据库**

table\_name = “GOOD” #表名，0.6以前版本使用db\_name

**（4）/apps/index/indeHandler.py**

from .model import **GOODS** #导入model

async def handler(): #创建

# 创建数据

# await self.db.create(**GOODS**, supplier\_id=8,name=”11”)

# 执行命令

await self.db.execute( peewee的方法 )

### 执行

#### 注意：

**Order.select(Goods.id) # select()中不能填写 \***

#### insert一条命令

data\_source = ("1",2,0)# 数据集合

# 添加的字段名

field = (Order.name, Order.num, Order.status)

await self.db.**execute**( # 执行的man命令

Order.**insert**(data\_source, field) # orm命令

)

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

# 返回值

成功1，失败0

#### insert多条命令

data\_source = [ # 数据集合

("1",2,0),("2",2,0),

]

# 添加的字段名

field = (Order.name, Order.num, Order.status)

await self.db.**execute**( # 执行的man命令

Order.**insert\_many**(data\_source, field) # orm命令

)

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

# 返回值

成功1，失败0

#### select查询一条信息

**info** = await self.db.**get**(

Classify.**select()**.**where**(Classify.parent\_id == 1)

)

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

# 获取变量值

print(**info**.name)

#### select查询多条数据

info = await self.db.**execute**(

Classify.**select().where**(Classify.parent\_id == 0)

)

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

for item in info: **# 解析**

print(item.name)

print(item.id)

print(item.parent\_id)

#### UPDATE

await self.db.execute(

Classify.update(name=self.json\_obj['info'])

.where(Classify.id == self.json\_obj['id'])

)

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

# 返回值

成功1，失败0

#### delete

await self.db.execute(

Classify.delete().where(...)

)

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

# 返回值

成功1，失败0

#### 排序

await self.db.execute(

Classify.select().where().**order\_by(Goods.id)**

)

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

# 返回值

成功1，失败0

#### 分页

await self.db.execute(

# 第一页，一页2个

Classify.select().where().**paginate(1, 2)**

)

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

# 返回值

成功1，失败0

# peewee-orm

除了get\_by\_id可以执行，其他的都不能直接执行，需要for循环，迭代效应，execute执行

get\_by\_id如果没有对应的id，会报错

model\_to\_dict

## 新建model

django orm,sqlalchemy

1. 安装peewee,底层依赖pymysql

pip install peewee

1. from peewee import \*

db = MySQLDatabase(“数据库名称”,host=”11”,port=”3306”,user=”root”,passwd)

class Base(Model):

# 这样 就不用每个表都添加datbase字段了，一下表继承这个表

class Meta:

database= db

# name是表名

# 没有设置主键，系统会默认添加id字段，并设为主键

class Name(Model): #这个model是peewee中的model

name = CharFiled(max\_length=255,verbose\_name=”注释”,index=True)#索引

class Mate:

database= db

table\_name = “自定义”

if \_\_name\_\_ ==”\_\_main\_\_”:

db.create\_tables([Name]) #生成表结构，如果有就不会新建

## 保存

from appmodel import Superline # 导入model

def save\_mode ():

superline = Superline()

superline.name = “hangzhou”

superline.save() #保存数据

return superline.id #数据库生成的id，保存后返回的id值

superline = Superline(\*\*data)

# 如何一次插入多条数据

def save\_mode\_manye ():

for data in da:

superline

## 查询

1. 获取一条数据

def get\_model ()：

# Superline.id是model中定义的字段

good = Superline(Superline.id == 1)

good = Superline.get\_by\_id(1) # 通过id获取

1. 获取所有数据

def get\_model ():

# 类似生成器，在for循环时才调用，迭代协议

good = Superline.select() # select \* from Superline，不执行，调用时才执行

# 查询指定字段

good = Superline.select(Superline.name)

# 根据条件获取

good = Superline.select().where(Superline.name == “11”)

#多个条件

good = Superline.select().where(（Superline.name=“11”）&（superline.name!=”aa”）)

#模糊查询

# select \* from Superline where name like “%飞天”

# 查询官网query Examples

Superline.select().where(Superline.name.**contains**(“飞天”))

1. 排序

Superline.select().order\_by(Superline.id.desc()降序) .asc() //升序

1. 分页

2代表第3条数据 ， 4代表取几条数据

Superline.select().order\_by(Superline.id).paginate(2,4)

## 更新

1. 查询后跟新

# 会直接更新之前查询到的数据 ，youid更新没有进行插入

good = Good.get\_by\_id(1)

good.name = “22”

good.save()

1. 使用update

# update click\_num=100 where id=1

Good.update(click\_num=100).where(Good.id==1).execute() #必须执行execute才能执行update语句，update生成的是一个update对象

Good.update(click\_num=Good.click\_num+1).where(Good.id==1).execute()

生成的都是一个Modelupdate对象

## 删除

Goods.delete().where(...).execute()

# wtform

**1、编写form类**

from wtforms import Form

class MessageForm(Form):

name = StringFiled(“姓名”)

**2、使用--index.py**

import MessageForm

message = MessageForm(self,request.arguments)

**3、验证**

message.validate():

# 上传图片、头像

**使用aiofile库**，github上有手册

1. **引入类**

import aiofiles

import os

**2、配置路径--setting.py**

"MEDIA\_ROOT": "./statics" # 以server.py为基准

**3、获取文件**

# image 文件名称

files\_meta = self.request.files.get("image", None)

**4、写入文件**

# 因为可能上传多个文件

for meta in files\_meta:

# file\_path 保存文件的路径

async with aiofiles.open(file\_path, 'wb') as f:

# meta[‘body’]数据

await f.write(meta['body'])

# 富文本

ueditor