Tornado

文档：https://tornado-zh-cn.readthedocs.io/zh_CN/latest/

github：https://github.com/tornadoweb/tornado

介绍

Tornado是使用Python开发的全栈式（full-stack）Web框架和异步网络库，最早由4名Google前软件工程师（布雷特·泰勒）2007创办的Friendfeed(一个社交聚合网站)开发而来的。通过使用非阻塞IO，Tornado可以处理数以万计的开放连接，是long polling、WebSockets和其他需要为用户维护长连接应用的理想选择。

目前最新版本6.1, 我们实际项目开发是使用的不可能是最新版本，所以在此我们在tornado基础阶段所学所用的版本为6.0.

特点

- 开源的轻量级全栈式Web框架，提供了一整套完善的异步编码方案。
- 高性能

  基于协程，底层就是基于asyio来实现的完整的协程调度

  采用异步非阻塞IO处理方式，不依赖多进程或多线程
  采用单进程单线程异步IO的网络模式，其高性能源于Tornado基于Linux的Epoll（UNIX为kqueue）的异步网络IO，具有出色的抗负载能力

  Tornado为了实现高并发和高性能，使用了一个`IOLoop`事件循环来处理`socket`的读写事件
- WSGI全栈替代产品，Tornado把应用（Application）和服务器（Server）结合起来，既是WSGI应用也可以是WSGI服务，通俗来讲就是说，Tornado既是web服务器也是web框架，甚至可以通过Tornado替代nginx来运行Flask或者django框架

安装

```
pip install tornado==6.0.4
```

入门

运行项目

server.py

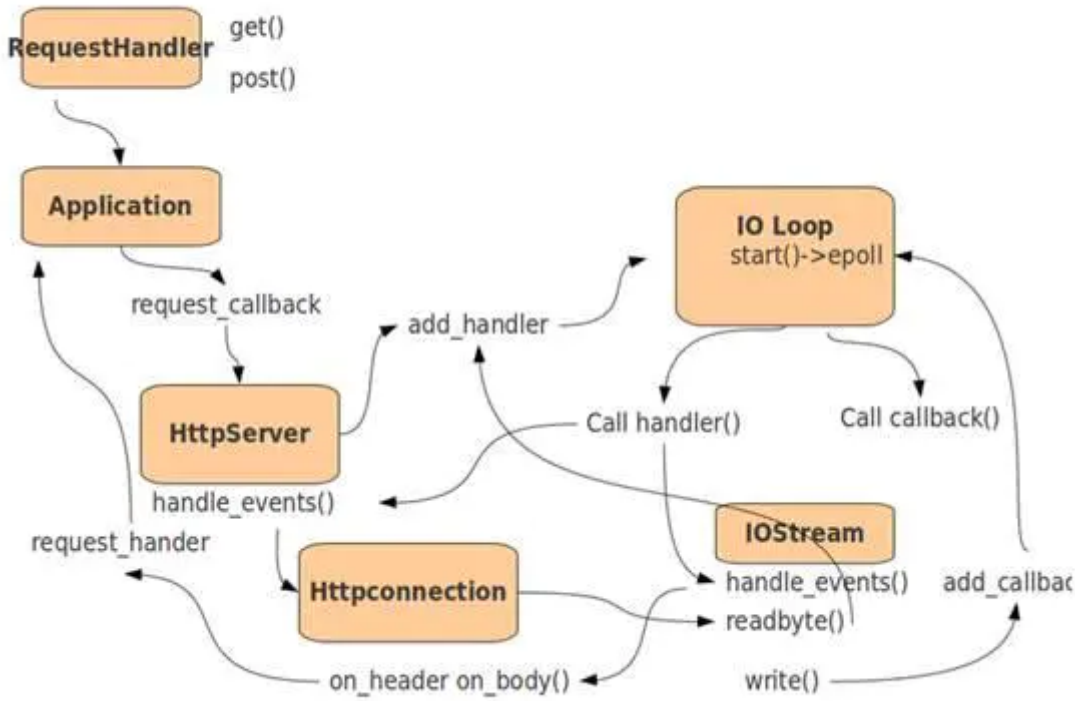
```
from tornado import ioloop
from tornado import web

class Home(web.RequestHandler):
    def get(self):
        # self.write 响应数据
        self.write("hello!")

def make_app():
    # Application是tornado web框架的核心应用类，是与服务器对应的接口，里面保存了路由映射表
    # handlers 设置路由列表
```

```
    return web.Application(handlers=[
        (r"/", Home),
    ])

if __name__ == "__main__":
    # 创建应用实例对象
    app = make_app()
    # 设置监听的端口和地址
    app.listen(8888)
    # ioloop，全局的tornado事件循环，是服务器的引擎核心，start表示创建IO事件循环,等待客户端连接
    ioloop.IOLoop.current().start()
```



终端运行

server.py，代码：

```
from tornado import ioloop
from tornado import web
from tornado.options import define,options,parse_command_line
define("port", default=8888, type=int,help="设置监听端口号，默认为8888")
class Home(web.RequestHandler):
    def get(self):
        # self.write 响应数据
        self.write("hello!")

def make_app():
    # handlers 设置路由列表
    return web.Application(handlers=[
        (r"/", Home),
    ])

if __name__ == "__main__":
```

```
    # 解析终端启动命令，格式：python server.py --port=端口号
    parse_command_line()
    # 创建应用实例对象
    app = make_app()
    # 设置监听的端口和地址
    app.listen(options.port) # options.port 接收参数
    # ioloop，全局的tornado事件循环，是服务器的引擎核心，start表示创建IO事件循环
    ioloop.IOLoop.current().start()
```

调试模式

开启自动加载和调试模式，代码：

```
from tornado import ioloop
from tornado import web
from tornado import autoreload
from tornado.options import define,options,parse_command_line

# 配置信息
settings = {
    'debug' : True,
}

define("port", default=8888, type=int,help="设置监听端口号，默认为8888")

# 类视图
class Home(web.RequestHandler):
    # 视图方法
    def get(self):
        # self.write 响应数据
        self.write("hello!")

def make_app():
    # handlers 设置路由列表
    return web.Application(handlers=[
        (r"/", Home),
    ],**settings) # 加载配置

if __name__ == "__main__":
    # 创建应用实例对象
    parse_command_line()
    app = make_app()
    # 设置监听的端口和地址
    app.listen(options.port)
    # ioloop，全局的tornado事件循环，是服务器的引擎核心，start表示创建IO事件循环
    ioloop.IOLoop.current().start()
```

路由拆分

代码：

```
from tornado import ioloop
from tornado import web
from tornado import autoreload
from tornado.options import define,options,parse_command_line
```

```
settings = {
    'debug' : True,
}

define("port", default=8888, type=int,help="设置监听端口号，默认为8888")

class Home(web.RequestHandler):
    def get(self):
        # self.write 响应数据
        self.write("hello!")

# 路由列表
urls = [
    (r"/", Home),
]

if __name__ == "__main__":
    # 创建应用实例对象
    parse_command_line()
    app = web.Application(urls,**settings)
    # 设置监听的端口和地址
    app.listen(options.port)
    # ioloop，全局的tornado事件循环，是服务器的引擎核心，start表示创建IO事件循环
    ioloop.IOLoop.current().start()
```

视图编写

```
from tornado import ioloop
from tornado import web
from tornado import autoreload
from tornado.options import define,options,parse_command_line

settings = {
    'debug' : True,
}

define("port", default=8888, type=int,help="设置监听端口号，默认为8888")

class Home(web.RequestHandler):
    def get(self):
        # self.write 响应数据
        self.write("hello!get")

    def post(self):
        self.write("hello!post")

    def put(self):
        self.write("hello!put")

    def patch(self):
        self.write("hello!patch")

    def delete(self):
        self.write("hello!delete")
```

```
# 路由列表
urls = [
    (r"/", Home),
]

if __name__ == "__main__":
    # 创建应用实例对象
    parse_command_line()
    app = web.Application(urls,**settings)
    # 设置监听的端口和地址
    app.listen(options.port)
    # ioloop，全局的tornado事件循环，是服务器的引擎核心，start表示创建IO事件循环
    ioloop.IOLoop.current().start()
```

多进程模式

```
from tornado import ioloop
from tornado import web,httpserver
from tornado import autoreload
from tornado.options import define,options,parse_command_line

settings = {
    'debug' : False,
}

define("port", default=8888, type=int,help="设置监听端口号，默认为8888")

class Home(web.RequestHandler):
    def get(self):
        # self.write 响应数据
        self.write("hello!get")

    def post(self):
        self.write("hello!post")

    def put(self):
        self.write("hello!put")

    def patch(self):
        self.write("hello!patch")

    def delete(self):
        self.write("hello!delete")

# 路由列表
urls = [
    (r"/", Home),
]

if __name__ == "__main__":
    # 创建应用实例对象
    app = web.Application(urls,**settings)
    parse_command_line()
    server = httpserver.HTTPServer(app)
    # 设置监听的端口和地址
```

```
    server.bind(options.port)
    server.start(0) # 0表示进程=CPU核数+1
    # ioloop，全局的tornado事件循环，是服务器的引擎核心，start表示创建IO事件循环
    ioloop.IOLoop.current().start()
```

请求与响应

请求

`tornado.httputil.HTTPServerRequest`

server.py，代码：

```
from tornado import ioloop
from tornado import web
from tornado import autoreload

from tornado.options import define,options,parse_command_line

settings = {
    'debug' : True,
}

define("port", default=8888, type=int,help="设置监听端口号，默认为8888")
class Home(web.RequestHandler):
    def get(self):
        print(self.request) # 请求处理对象
        # HTTPServerRequest(protocol='http', host='127.0.0.1:8888',
method='GET', uri='/?name=xiaoming', version='HTTP/1.1', remote_ip='127.0.0.1')
        # print(self.request.protocol) # 协议
        # print(self.request.method) # Http请求方法
        # print(self.request.uri)    # uri地址
        # print(self.request.full_url())    # 完整url地址
        # print(self.request.version) # HTTP协议版本
        # print(self.request.headers) # 请求头 HTTPHeaders
        # print(self.request.body) # 请求体[原始数据]
        # print(self.request.host) # 地址端口
        # print(self.request.files) # 上传文件
        # print(self.request.cookies) # cookie信息
        # print(self.request.remote_ip) # 客户端IP地址
        print(self.request.query_arguments) # 地址参数列表
        print(self.request.body_arguments) # 请求体参数列表
        print(self.request.request_time()) # 请求处理时间
        self.write("hello world")

# 设置路由列表
urls = [
    (r"/", Home),
]

if __name__ == "__main__":
    # 创建应用实例对象
    parse_command_line()
    app = web.Application(urls,**settings)
    # 设置监听的端口和地址
```

```
    app.listen(options.port)
    # ioloop，全局的tornado事件循环，是服务器的引擎核心，start表示创建IO事件循环
    ioloop.IOLoop.current().start()
```

接收查询字符串

server.py，代码：

```
from tornado import ioloop
from tornado import web
from tornado import autoreload
from tornado.options import define,options,parse_command_line

settings = {
    'debug' : True,
}

define("port", default=8888, type=int,help="设置监听端口号，默认为8888")
class Home(web.RequestHandler):
    def get(self):
        # print(self.request.arguments["name"][0].decode())
        # name = self.get_argument("name") # self.get_query_argument("name")
        # print(name) # xiaoming
        names = self.get_arguments("name") # # self.get_query_arguments("name")
        print(names) # ['xiaoming', '123']
        # self.write 响应数据
        # self.write("hello!")
        self.write("hello world")

# 设置路由列表
urls = [
    (r"/", Home),
]

if __name__ == "__main__":
    # 创建应用实例对象
    parse_command_line()
    app = web.Application(urls,**settings)
    # 设置监听的端口和地址
    app.listen(options.port)
    # ioloop，全局的tornado事件循环，是服务器的引擎核心，start表示创建IO事件循环
    ioloop.IOLoop.current().start()
```

浏览器：`http://127.0.0.1:8888/?name=xiaoming&name=xiaohong`

接收请求体

```
from tornado import ioloop
from tornado import web
from tornado import autoreload
from tornado.options import define,options,parse_command_line

settings = {
    'debug' : True,
}
```

```
define("port", default=8888, type=int,help="设置监听端口号，默认为8888")
class Home(web.RequestHandler):
    def get(self):
        # print(self.request.arguments["name"][0].decode())
        # name = self.get_argument("name") # self.get_query_argument("name")
        # print(name) # xiaoming
        names = self.get_arguments("name") # # self.get_query_arguments("name")
        print(names) # ['xiaoming', '123']
        self.write("hello!get")

    def post(self):
        print(self.request.arguments) # {'name': [b'xiaoming', b'xiaohong']}
        print(self.request.body_arguments) # {'name': [b'xiaohong']}
        print(self.get_argument("name")) # xiaohong
        print(self.get_body_argument("name")) # xiaohong
        print(self.get_arguments("name")) # ['xiaoming', 'xiaohong']
        print(self.get_body_arguments("name")) # ['xiaohong']
        self.write("hello!post")

# 设置路由列表
urls = [
    (r"/", Home),
]

if __name__ == "__main__":
    # 创建应用实例对象
    parse_command_line()
    app = web.Application(urls,**settings)
    # 设置监听的端口和地址
    app.listen(options.port)
    # ioloop，全局的tornado事件循环，是服务器的引擎核心，start表示创建IO事件循环
    ioloop.IOLoop.current().start()
```

接收路由参数

```
from tornado import ioloop
from tornado import web
from tornado import autoreload
from tornado.options import define,options,parse_command_line

settings = {
    'debug' : True,
}

define("port", default=8888, type=int,help="设置监听端口号，默认为8888")

class Home(web.RequestHandler):
    def get(self,name):
        print(name)
        self.write("home!get")

class Index(web.RequestHandler):
    def get(self,name):
        print(name)
```

```
        self.write("index!get")

# 路由列表
urls = [
    (r"/home/(.+)", Home), # 不绑定传参
    (r"/index/(?P<name>.+)", Index), # 绑定传参
]

if __name__ == "__main__":
    # 创建应用实例对象
    parse_command_line()
    app = web.Application(urls,**settings)
    # 设置监听的端口和地址
    app.listen(options.port)
    # ioloop，全局的tornado事件循环，是服务器的引擎核心，start表示创建IO事件循环
    ioloop.IOLoop.current().start()
```

响应

```
from tornado import ioloop
from tornado import web
from tornado import autoreload
from tornado.options import define, options, parse_command_line

settings = {
    'debug': True,
}

define("port", default=8888, type=int, help="设置监听端口号，默认为8888")

from datetime import datetime
class Home(web.RequestHandler):
    def set_default_headers(self):
        self.set_header("time", int(datetime.now().timestamp()))

    def get(self):
        # self.write("<h1>hello</h1>") # 响应html文本信息
        self.write({"message":"hello get"}) # 响应json数据
        self.set_header("Content-Type","text/json; charset=gbk")
        self.add_header("Company","OldboyEdu") # 自定义响应头
        self.set_cookie("name","xiaohui") # 设置cookie

    def post(self):
        self.write({"message": "hello post"})  # 响应json数据

    def put(self):
        self.clear_header("time")
        # self.set_status(404,"Not Found")
        # self.send_error(500,reason="服务器炸了！")
        self.send_error(404, msg="服务器炸了！", info="快报警")

    def write_error(self, status_code, **kwargs):
        self.write("<h1>完蛋啦...</h1>")
        self.write("<p>错误信息:%s</p>" % kwargs["msg"])
        self.write("<p>错误描述:%s</p>" % kwargs["info"])
```

```
    def patch(self):
        # 页面跳转
        self.redirect("http://www.baidu.com")

# 设置路由列表
urls = [
    (r"/", Home),
]

if __name__ == "__main__":
    # 创建应用实例对象
    parse_command_line()
    app = web.Application(urls, **settings)
    # 设置监听的端口和地址
    app.listen(options.port)
    # ioloop，全局的tornado事件循环，是服务器的引擎核心，start表示创建IO事件循环
    ioloop.IOLoop.current().start()
```

cookie

基本使用

> 设置cookie　self.set_cookie(name, value)
>
> 获取cookie　self.get_cookie(name)

server.py,代码:

```
from tornado import web
from tornado import ioloop
settings = {
    "debug": True,
}

from datetime import datetime
class Home(web.RequestHandler):
    def get(self):
        # 设置cookie

 self.set_cookie("uname","xiaoming",expires=int(datetime.now().timestamp())+10)
        self.write("set cookie")

class Index(web.RequestHandler):
    def get(self):
        # 获取cookie
        uname = self.get_cookie("uname","")
        self.write("uname=%s" % uname)

urls = [
    (r"/cookie/set", Home),
    (r"/cookie/get", Index),
]

if __name__ == '__main__':
    app = web.Application(urls,**settings)
```

```
    app.listen(port=8888)
    ioloop.IOLoop.current().start()
```

加密使用

> 设置cookie　self.set_secure_cookie(name,value)
>
> 获取cookie　self.get_secure_cookie(name)
>
> 删除cookie　self.clear_cookie(name)
>
> 清空cookie　self.clear_all_cookie()

```
from tornado import web
from tornado import ioloop
settings = {
    "debug": True,
    # import base64, uuid
    # base64.b64encode(uuid.uuid4().bytes + uuid.uuid4().bytes)
    "cookie_secret": "WO+JNAJ3QZyOe4SMVXZpXAt3uG9hoU0UokoCBeYn1Y4="
}

from datetime import datetime
class Home(web.RequestHandler):
    def get(self):

 self.set_secure_cookie("name","xiaoming",expires=int(datetime.now().timestamp()
)+30)

 self.set_secure_cookie("age","16",expires=int(datetime.now().timestamp())+30)
        self.write("set cookie")

class Index(web.RequestHandler):
    def get(self):
        # 获取cookie[加密]
        age = self.get_secure_cookie("age")
        name = self.get_secure_cookie("name")
        if age is not None:
            age  = age.decode()
        if name is not None:
            name = name.decode()
        self.write("age=%s,name=%s" % (age,name))

class Page(web.RequestHandler):
    def get(self):
        """删除cookie"""
        # self.clear_cookie("age") # 删除指定名称的cookie[不管是否有加密]
        self.clear_all_cookies() # 删除所有cookie[慎用]
        self.write("del cookie")

urls = [
    (r"/cookie/set", Home),
    (r"/cookie/get", Index),
    (r"/cookie/del", Page),
]

if __name__ == '__main__':
    app = web.Application(urls,**settings)
```

```
    app.listen(port=8888)
    ioloop.IOLoop.current().start()
```

> 注意:
>
> 1. tornado没有提供session操作，如果需要使用到session可以自己实现或者引入第三方模块。
> 2. cookie的删除, 是不管是否加密的.

静态文件

```
from tornado import web
from tornado import ioloop
import os
settings = {
    'debug': True,
    # 静态文件保存路径
    "static_path": os.path.join(os.path.dirname(__file__), 'static'),
    # 静态文件url地址前缀
    "static_url_prefix":"/static/", # 必须前后有斜杠
    # 提供静态文件访问支持的视图类
    "static_handler_class": web.StaticFileHandler,
}

class Home(web.RequestHandler):
    def get(self):
        # 项目中使用join拼凑路径时，必须注意第二个参数，必能以斜杠开头，会出现路径穿越(路径穿透)问题
        path = os.path.join(os.path.dirname(__file__),"/static")
        self.write("path=%s" % path)

urls = [
    (r"/", Home),
    # 上面settings中关于静态文件的配置，主要是提供给Application应用对象进行初始化生成下面路由时候使用到的。
    # (r"/static/(.*)", web.StaticFileHandler, {"path": os.path.join(os.path.dirname(__file__), 'static')}),
]

if __name__ == '__main__':
    app = web.Application(urls,**settings)
    app.listen(port=8888)
    ioloop.IOLoop.current().start()
```

页面响应

加载template模板

```
from tornado import web
from tornado import ioloop
import os
settings = {
    'debug': True,
    "template_path": os.path.join(os.path.dirname(__file__), 'templates'),
```

```
}

class Home(web.RequestHandler):
    def get(self):
        self.render("index.html",data={"message":"hello world"})

urls = [
    (r"/", Home),
]

if __name__ == '__main__':
    app = web.Application(urls,**settings)
    app.listen(port=8888)
    ioloop.IOLoop.current().start()
```

templates/index.html，代码：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <p>{{data["message"]}}</p>

    tornado默认内置了一套非常强大的模板引擎<br>
    这套模板引擎是基于jinja2模板引擎的基础上进行了改造而成的。<br>
    当然jinja2是基于django的DTL模板引擎基础上改造而成的。<br>
    所以flask和tornado进行比较的时候，从来不提tornado抄袭模板引擎这个事，反而会和django去比较模板引擎的问题。
</body>
</html>
```

路由进阶

路由语法和参数

在路由列表的路由成员中，我们一共可以设置4个参数

> url(r"/uri路径", 视图类, {"参数名":"参数值"}, name="路由别名")

server.py

```
from tornado import web
from tornado import ioloop
settings = {
    'debug': True,
}

class Home(web.RequestHandler):
    def initialize(self,company)-> str:
        # initialize 初始化方法[钩子方法]
        self.company = company
```

```
    def get(self):
        print(self.company)
        print("uri路径：%s" % self.reverse_url("home") ) # 对路由别名进行 反解析
        self.write("hello,get")
    def post(self):
        print(self.company)

from tornado.web import url
urls = [
    # (r"/", Home), # 这个格式的路由其实是简写模式， 在tornaodo.web中内部中最终由
_ApplicationRouter 的 Rule来进行封装和匹配路由和视图的关系
    # url(pattern=路由uri地址, handler=视图类,kwargs=提供给视图类的公共参数,name="路由别
名，用于反解析"),
    url(pattern=r"/abc", handler=Home,kwargs=
{"company":"OldBoyEdu"},name="home"),
]

if __name__ == '__main__':
    app = web.Application(urls,**settings)
    app.listen(port=8888)
    ioloop.IOLoop.current().start()
```

视图进阶

视图中内置的钩子方法

在tornado提供的视图类中，我们除了可以编写客户端http请求对应名称的视图方法和初始化方法**initialize**以外，还提供了一个预处理方法**prepare**和**on_finish**，prepare方法会在http请求方法执行之前先执行,**on_finish**会在http响应完成时进行。

server.py，代码：

```
from tornado import ioloop
from tornado import web
from tornado.httpserver import HTTPServer
from tornado.options import define, options, parse_command_line
from tornado.web import url
settings = {
    'debug': True,
}

define("port", default=8888, type=int, help="设置监听端口号，默认为8888")

class Home(web.RequestHandler):
    def prepare(self):
        print(self.request.headers.get("Content-Type"))
        if self.request.headers.get("Content-
Type","").startswith("application/json"):
            import json
            self.json_data = json.loads(self.request.body)
            self.is_json = True
        else:
            self.is_json = False
            self.json_data = None
```

```
    def post(self):
        if self.is_json:
            print(self.json_data)
        self.write("hello,post")

    def on_finish(self):
        print("请求处理结束!可以在此进行收尾工作，例如：记录日志等")

# 设置路由列表
urls = [
    (r"/", Home),
]

if __name__ == "__main__":
    # 创建应用实例对象
    parse_command_line()
    app = web.Application(urls, **settings)
    server = HTTPServer(app)
    # 设置监听的端口和地址
    server.listen(options.port)
    server.start(1)
    ioloop.IOLoop.current().start()
```

视图方法调用顺序

```
from tornado import ioloop
from tornado import web
from tornado.httpserver import HTTPServer
from tornado.options import define, options, parse_command_line
from tornado.web import url
settings = {
    'debug': True,
}

define("port", default=8888, type=int, help="设置监听端口号，默认为8888")

class Home(web.RequestHandler):
    def initialize(self):
        print("initialize执行了")

    def prepare(self):
        print("prepare执行了")


    def set_default_headers(self):
        print("set_default_headers执行了")

    def get(self):
        self.write("hello,get")
        print("视图http方法执行了")
        # self.send_error(200,msg="注意：丢炸弹了")  # 此处抛出错误


    def write_error(self, status_code, **info):
        print("write_error执行了,msg=%s" % info["msg"])
```

```
    def on_finish(self):
        print("on_finish执行了")

# 设置路由列表
urls = [
    (r"/", Home),
]

if __name__ == "__main__":
    # 创建应用实例对象
    parse_command_line()
    app = web.Application(urls, **settings)
    server = HTTPServer(app)
    # 设置监听的端口和地址
    server.listen(options.port)
    server.start(1)
    ioloop.IOLoop.current().start()
```

当视图中没有任何异常时，执行顺序：

```
set_defautl_headers()
initialize()
prepare()
视图http方法()
on_finish()
```

当视图中抛出异常错误时，执行顺序：

```
set_default_headers()
initialize()
prepare()
视图http方法()
set_default_headers()
write_error()
on_finish()
```

冲刷缓存

在前面的学习中，我们使用了`self.write()`来完成数据的响应。

事实上，在tornado提供的视图操作中，视图中提供了一个 _write_buffer列表用于暂时缓存提供给客户端的数据, 这个 _write_buffer就是输出缓冲区

`self.write()`本质上来说是将chunk数据块写到输出缓冲区中。所以才出现在视图中多次调用`self.write()`输出数据的情况,因为self.write根本没有输出数据,而是把数据写入到了输出缓冲区里面. 如果没有其他操作干预的情况下，则视图方法处理完成以后，会讲输出缓冲区中所有的数据冲刷出来响应给客户端。

除了`self.write()`方法以外，tornado还提供了2个方法用于在视图中冲刷缓存数据到客户端的。

`self.flush()` 立刻把数据从输出缓冲区冲刷出去。

`self.finish()`立刻把数据从输出缓冲区冲刷出去。但是与self.flush()不同的是, self.finish()执行了以后, 后面的所有输出调用都不在支持.也就不能返回给客户端。

server.py，代码：

```
from tornado import ioloop
from tornado import web
from tornado.httpserver import HTTPServer
from tornado.options import define, options, parse_command_line
settings = {
    'debug': True,
}

define("port", default=8888, type=int, help="设置监听端口号，默认为8888")

class Home(web.RequestHandler):
    def get(self):
        import time
        self.write("hello,get1")
        self.flush()
        time.sleep(3)
        self.write("hello,get2")
        self.flush()
        time.sleep(3)
        self.write("hello,get3")
        self.flush()
        self.finish("这里一般不写任何内容，表示视图处理结束")
        self.write("hello,get4")

# 设置路由列表
urls = [
    (r"/", Home),
]

if __name__ == "__main__":
    # 创建应用实例对象
    parse_command_line()
    app = web.Application(urls, **settings)
    server = HTTPServer(app)
    # 设置监听的端口和地址
    server.listen(options.port)
    server.start(1)
    ioloop.IOLoop.current().start()
```

用户认证

tornado提供了装饰器tornado.web.authenticated与视图内置方法`get_current_user`允许我们轻松的实现用户认证功能。

装饰器authenticated依赖于请求处理类中的self.current_user属性来进行判断用户是否通过认证，如果self.current_user值为假（None、False、0、""等），任何GET或HEAD请求都将把访客重定向到settings配置中login_url设置的URL，而非法用户的POST请求将返回HTTPError(403)异常, Forbidden。

server.py，代码：

```
from tornado import web
from tornado import ioloop
settings = {
```

```
    'debug': True,
    # 登录页面的url地址
    "login_url": r"/login"
}

from tornado.web import authenticated

class HttpRequest(web.RequestHandler):
    def get_current_user(self):
        username = self.get_argument("username", "")
        password = self.get_argument("password", "")

        if username == "root" and password == "123":
            return username

class Home(HttpRequest):
    @authenticated
    def get(self):
        self.write("hello,用户个人中心")

    @authenticated
    def post(self):
        self.write("hello,用户中心")

class UserLogin(web.RequestHandler):
    def get(self):
        self.write("登录页面")

urls = [
    (r"/", Home),
    (settings["login_url"], UserLogin),
]

if __name__ == '__main__':
    app = web.Application(urls,**settings)
    app.listen(port=8888)
    ioloop.IOLoop.current().start()
```

> 先访问：http://127.0.0.1:8888/?username=root&password=123
>
> 再访问：http://127.0.0.1:8888/

模板语法

基本语法

变量、表达式与自定义函数

server.py，代码：

```
from tornado import ioloop
from tornado import web
from tornado.options import define, options, parse_command_line
import os
settings = {
```

```
    'debug': True,
    # 静态文件保存路径
    "static_path": os.path.join(os.path.dirname(__file__), 'static'),
    # 静态文件url地址前缀
    "static_url_prefix":"/static/", # 必须前后有斜杠
    "template_path": os.path.join(os.path.dirname(__file__), 'templates'),
}

define("port", default=8888, type=int, help="设置监听端口号，默认为8888")

def money_format(data):
    return "%.2f" % data

import time
class Home(web.RequestHandler):
    def get(self):
        data = {
            "name":"xiaoming",
            "money": 100,
            "address":["北京市","昌平区","白沙路地铁站"]
        }
        info = {"name":"xiaoming"}

 self.render("index.html",time=time,info=info,money_format=money_format,**data)

# 设置路由列表
from tornado.web import url
urls = [
    url(r"/", Home,{}, name="home"),
]

if __name__ == "__main__":
    # 创建应用实例对象
    parse_command_line()
    app = web.Application(urls, **settings)

    # 设置监听的端口和地址
    app.listen(options.port)
    # ioloop，全局的tornado事件循环，是服务器的引擎核心，start表示创建IO事件循环
    ioloop.IOLoop.current().start()
```

templates/index.html，代码:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    {# 模板注释 #}
    <p>name={{name}}</p>
    <p>name={{info['name']}}</p>
    <p>{{"-".join(address)}}</p>
    <p>{{money_format(money)}}</p>
</body>
</html>
```

控制语句

```
# 判断
{% if ... %}
{% elif ... %}
{% else ... %}
{% end %}

# 遍历
{% for ... in ... %}
{% end %}

# 循环
{% while ... %}
{% end %}
```

内置标签或函数

内置标签

```
# 导包
{% from ... import ... %}
{% import ... %}
# 加载其他模板
{% include ... %}
# 输出原始数据
{% raw ... %}

# 语句/局部变量
{% set 变量名=变量值 %}

# 异常处理
{% try %}...{% except %}...{% else %}...{% finally %}...{% end %}
# 模板继承
{% extends *filename* %}
{% block 模板块名称} {% end %}
```

内置函数

```
# 输出转义数据，tornado在配置中允许通过autoescape=None设置全局转义
{{ escape(text) }}
# 静态文件存储路径
{{ static_url("style.css") }}
# 路由反解析
reverse_url("路由别名")

# CSRF防范机制，CSRF也叫XSRF
# tornado开启csrf必须在配置中进行设置 xsrf_cookies = True
# 补充：
# 在前后端分离项目中，客户端可以通过cookie来读取XSRFToken，cookie名称为_xsrf，请求头必须名称：X-XSRFToken
# 在视图方法中可以通过 self.xsrf_token 来获取 XSRFTokentoken
{% module xsrf_form_html() %}
```

内置变量

```
# 客户端请求对象
request
```

数据库

与Django框架相比，Tornado没有自带ORM，对于数据库需要自己去适配。我们使用MySQL数据库。

在Tornado3.0版本以前提供tornado.database模块用来操作MySQL数据库，而从3.0版本开始，此模块就被独立出来，作为torndb包单独提供。torndb只是对MySQLdb的简单封装，不支持Python 3。所以如果在当前版本中使用torndb进行数据库操作，需要修改源代码，所以在此，我们使用pymysql。

> 项目中如果要使用ORM，可以使用SQLAlchemy，但开发中，很少有人这么使用.
>
> 同时，tornado强大的地方在于其异步非阻塞，所以我们后面关于数据库操作，不管是mysql，mongodb还是redis基本都是异步读写操作。

MySQL

```
pip install pymysql
```

mysql.py，代码:

```
import pymysql
class MySQL(object):
    def __init__(self, host, user, pwd, name):
        self.host = host
        self.user = user
        self.pwd = pwd
        self.name = name
        self.data = None
        self.last_sql = None

    def connect(self):
```

```
        self.db = pymysql.Connect(host=self.host, user=self.user,
passwd=self.pwd, db=self.name)
        self.cursor = self.db.cursor()

    def close(self):
        self.cursor.close()
        self.db.close()

    def get_one(self, sql):
        try:
            self.connect()
            self.cursor.execute(sql)
            res = self.cursor.fetchone()
            self.data = res
            self.last_sql = sql
            self.close()
        except:
            print("fail to select")

        return self
    def get_all(self, sql):
        try:
            self.connect()
            self.cursor.execute(sql)
            res = self.cursor.fetchall()
            self.last_sql = sql
            self.data = res
            self.close()
        except:
            print("fail to select")
        return self

    def get_all_obj(self, sql):
        resList = []
        fieldList = []
        arr = sql.lower().split(" ")
        while "" in arr:
            arr.remove("")
        tableName = arr[arr.index("from")+1]
        fieldSql = "select COLUMN_NAME from information_schema.COLUMNS where
table_name='%s' and table_schema='%s'" % (
            tableName, self.name
        )
        fields = self.get_all(fieldSql).data
        for item in fields:
            fieldList.append(item[0])

        res = self.get_all(sql).data
        for item in res:
            obj = {}
            count = 0
            for x in item:
                obj[fieldList[count]] = x
                count += 1
            resList.append(obj)
        self.data = resList
        return self
```

```
    def insert(self, sql):
        return self.execute(sql)

    def update(self, sql):
        return self.execute(sql)

    def delete(self, sql):
        return self.execute(sql)

    def execute(self, sql):
        count = 0
        try:
            self.connect()
            count = self.cursor.execute(sql)
            self.db.commit()
            self.data = self.db.rowcount()
            self.last_sql = sql
            self.close()
        except:
            print("fail execute sql")
            self.db.rollback()
        return count
```

server.py，代码：

```
from tornado import ioloop
from tornado import web
from mysql import MySQL


mysql = {
    "host": "127.0.0.1",
    "user": "root",
    "pwd": "123",
    "db": "students"
}

db = MySQL(mysql["host"], mysql["user"], mysql["pwd"], mysql["db"])

settings = {
    'debug': True,
}

class Request(web.RequestHandler):
    def initialize(self,db):
        self.db = db

    def write(self, chunk: web.Union[str, bytes, dict, list]) -> None:
        if self._finished:
            raise RuntimeError("Cannot write() after finish()")
        if not isinstance(chunk, (bytes, web.unicode_type, dict, list)):
            message = "write() only accepts bytes, unicode, and dict objects"
            # if isinstance(chunk, list):
            #     message += (
            #         ". Lists not accepted for security reasons; see "
```

```
            #         +
"http://www.tornadoweb.org/en/stable/web.html#tornado.web.RequestHandler.write"
# noqa: E501
            #     )
            raise TypeError(message)
        if isinstance(chunk, (dict,list)):
            chunk = web.escape.json_encode(chunk)
            self.set_header("Content-Type", "application/json; charset=UTF-8")
        chunk = web.utf8(chunk)
        self._write_buffer.append(chunk)

class Home(Request):
    def get1(self):
        """查询一条数据"""
        data = self.db.get_one("select * from tb_student").data
        print(data)
        self.write("hello,get")

    def get2(self):
        """查询多条数据[返回元祖]"""
        data = self.db.get_all("select * from tb_student").data
        print(data)
        self.write("hello,get")

    def get3(self):
        """查询多条数据[返回元祖]"""
        data = self.db.get_all("select * from tb_student").data
        print(data)
        """查询多条数据[返回列表]"""
        data = self.db.get_all_json("select * from tb_student").data
        print(data)
        self.write("hello,get")

    def get(self):
        """添加/删除/修改"""
        # """添加一条"""
        # name = "xiaobai"
        # avatar = "2.png"
        # age = 16
        # sex = True
        # money = 100
        # sql = "INSERT INTO tb_student (name,avatar,age,sex,money) VALUES
('%s','%s',%s,%s,%s)" % (name,avatar,age,sex,money)
        # data = self.db.insert(sql).data
        # print(data)

        """添加多条"""
        # student_list = [
        #
{"name":"xiaohui1","avatar":"1.png","age":16,"sex":True,"money":1000},
        #
{"name":"xiaohui2","avatar":"2.png","age":16,"sex":False,"money":1000},
        #
{"name":"xiaohui3","avatar":"3.png","age":16,"sex":True,"money":1000},
        #
{"name":"xiaohui4","avatar":"4.png","age":16,"sex":False,"money":1000},
        # ]
        # table = "tb_student"
```

```
        #
        # fields = ",".join( student_list[0].keys() )
        # sql = "INSERT INTO %s (%s) VALUES " % (table,fields)
        #
        # for student in student_list:
        #     sql += "('%s','%s',%s,%s,%s)," %
(student["name"],student["avatar"],student["age"],student["sex"],student["money"
])
        # sql = sql[:-1]
        # data = self.db.insert(sql).data
        # print(data)

        """更新"""
        # name = "小辉"
        # sql = "UPDATE tb_student set name='%s' WHERE id = 3" % (name)
        # data = self.db.update(sql).data
        # print(data)

        """删除"""
        # sql = "DELETE FROM tb_student WHERE id = 16"
        # data = self.db.delete(sql).data
        # print(data)

        self.write("hello,get")

# 设置路由列表
urls = [
    (r"/", Home,{"db":db}),
]

if __name__ == "__main__":
    # 创建应用实例对象
    app = web.Application(urls, **settings)
    # 设置监听的端口和地址
    app.listen(8888)
    # ioloop，全局的tornado事件循环，是服务器的引擎核心，start表示创建IO事件循环
    ioloop.IOLoop.current().start()
```

同步和异步

概念

同步是指代在程序执行多个任务时，按部就班的依次执行，必须上一个任务执行完有了结果以后，才会执行下一个任务。

异步是指代在程序执行多个任务时，没有先后依序，可以同时执行，所以在执行上一个任务时不会等待结果，直接执行下一个任务。一般最终在下一个任务中通过状态的改变或者通知、同调的方式来获取上一个任务的执行结果。

同步

server.py，代码：

```
def client_A():
    """模拟客户端A"""
    print('开始处理请求1-1')
```

```
    # time.sleep(5)
    print('完成处理请求1-2')

def client_B():
    """模拟客户端B"""
    print('开始处理请求2-1')
    print('完成处理请求2-2')

def tornado():
    """模拟tornado框架"""
    client_A()
    client_B()

if __name__ == "__main__":
    main()
```

异步

server.py，代码：

```
from threading import Thread
from time import sleep

def async(func):
    def wrapper(*args, **kwargs):
        thread = Thread(target=func, args=args, kwargs=kwargs)
        thread.start()
        print(threading.activeCount())
    return wrapper

@async
def funcA():
    sleep(5)

    print("funcA执行了")

def funcB():
    print("funcB执行了")

if __name__ == "__main__":
    funcA()
    funcB()
```

协程

要理解什么是协程（Coroutine），必须先清晰**迭代器**和**生成器**的概念。

迭代器

迭代器就是一个对象，一个可迭代的对象，是可以被for循环遍历输出的对象。当然专业的说，就是实现了迭代器协议的对象。

任何一个对象，只要类中实现了`__iter__()`就是一个可迭代对象（iterable）。

任何一个对象，只要类中实现了`__iter__()`和`__next__()`就是一个迭代器（iterator）。

迭代器一定是可迭代对象，可迭代对象不一定是迭代器。

要了解迭代器，我们先编写一个代码来看看python提供的可迭代对象。常见的有：str，list，tuple，dic，set，文件对象。

迭代器是惰性执行的，可以节省内存，不能反复，只能向下取值。

server.py，代码：

```
# 可迭代对象
arr = [4,5,6,7]
print(dir(arr))

for item in arr:
    print(item)

# 不可迭代对象
num = 123
print(dir(num))

for item in num:
    print(item) # TypeError: 'int' object is not iterable # 类型错误：'int'对象是不可迭代的

# 可迭代对象
class Colors(object):
    def __init__(self):
        self.data = ["红色", "橙色", "紫色", "黄色"]

    def __iter__(self):
        return iter(self.data)


colors = Colors()
for color in colors:
    print(color)
```

查看一个对象是否是可迭代对象或迭代器：

```
from collections import Iterable
from collections import Iterator
print(isinstance([1,2,3,4],Iterable)) # True        # 查看是不是可迭代对象
print(isinstance([1,2,3,4],Iterator)) # False       # 查看是不是迭代器
print(isinstance([1,2,3,4].__iter__(),Iterator))    # True，
# 所有的迭代对象都有一个__iter__方法，该方法的作用就是返回一个迭代器对象
```

接下来，动手编写一个迭代器。

server.py，代码：

```
class Num(object):
    def __init__(self):
        self.current = 0

    def __next__(self):
        if self.current >= 5:
            raise StopIteration
```

```
        self.current += 1
        return self.current

    def __iter__(self):
        return self

num = Num()
for item in num:
    print(item)

num = Num()
itera = iter(num)
print(next(itera))
print(next(itera))
print(next(itera))
print(next(itera))
print(itera.__next__())
# print(next(itera))
```

> `__iter__()` 方法返回一个特殊的迭代器对象， 这个迭代器对象实现了 `__next__()` 方法并通过 StopIteration 异常标识迭代的完成。
>
> `__next__()` 方法返回下一个迭代器对象。
>
> **StopIteration** 异常用于标识迭代的完成，防止出现无限循环，在 `__next__()` 方法中可以设置在完成**指定循环次数**后触发 **StopIteration** 异常来结束迭代。

生成器

在 Python 中，使用了 **yield** 的函数被称为生成器函数。

生成器函数执行以后的返回结果就是**生成器**（**generator**），是一种特殊的迭代器。生成器只能用于迭代操作。

yield 是一个python内置的关键字，它的作用有一部分类似**return**，可以返回函数的执行结果。但是不同的是，return 会终止函数的执行，yield 不会终止生成器函数的执行。两者都会返回一个结果，但return只能一次给函数的调用处返回值，而yield是可以多次给next()方法返回值，而且yield还可以接受外界send()方法的传值。所以，更准确的来说，**yield是暂停程序的执行，交出程序的执行权，并记录了程序当前的运行状态。**

server.py，代码：

```
def func():
    for item in [4,5,6]:
        return item

def gen1():
    for item in [4,5,6]:
        yield item

def gen2():
    key = 0
    print(">>>>> 嘟嘟，开车了")
    while True:
        food = yield "第%s次" % key
```

```
        print('接收了，%s'% food)
        key +=1

f = func()
print(f)
g1 = gen1()
print(g1)
for item in g1:
    print(item)

g2 = gen1()
print(g2)
print(next(g2))
print(next(g2))
print(g2.__next__())
# print(next(g2))

g3 = gen2()
g3.send(None) # g3.__next__() 预激活生成器,让生成器内部执行到第一个yield位置，否则无法通过send传递数据给内部的yield
for item in ["苹果","芒果"]:
    print(g3.send(item))
```

> 使用生成器可以让代码量更少，内存使用更加高效节约。
>
> 所以在工作中针对海量数据查询，大文件的读取加载，都可以考虑使用生成器来完成。因为一次性读取大文件或海量数据必然需要存放内容，而往往读取的内容大于内存则可能导致内存不足，而使用生成器可以像挤牙膏一样，一次读取一部分数据通过yield方法，每次yield返回的数据都是保存在同一块内存中的，所以比较起来肯定比一次性读取大文件内容来说，内存的占用更少。

yield 和 yield from

server.py，代码：

```
def gen1():
    a = 0
    while True:
        # print("+++++++")
        a = yield a**2

def gen2(gen):
    # yield from gen
    a = 0
    b = 1
    gen.send(None)
    while True:
        # print("-------")
        b = yield a
        a = gen.send(b)


if __name__ == '__main__':
    g2 = gen2(gen1())
    g2.send(None)
    for i in range(5):
        # print(">>>> %s" % i)
        print(g2.send(i))
```

基于生成器来实现协程异步

这也是协程的实现原理，任务交替切换执行（遇到IO操作时进行任务切换才有使用价值）。

server.py，代码：

```
import time
def gen1():
    while True:
        print("--1")
        yield
        print("--2")
        time.sleep(1)

def gen2():
    while True:
        print("--3")
        yield
        print("--4")
        time.sleep(1)

if __name__ == "__main__":
    g1 = gen1()
    g2 = gen2()
    for i in range(3):
        next(g1)
        print("主程序!")
        next(g2)
```

Tornado的协程

Tornado的异步编程也主要体现在网络IO的异步上，即异步Web请求。

异步Web请求客户端

Tornado提供了一个异步Web请求客户端tornado.httpclient.AsyncHTTPClient用来进行异步Web请求。

fetch(request, callback=None)

用于执行一个web请求request，并异步返回一个tornado.httpclient.HTTPResponse响应。

request可以是一个url，也可以是一个tornado.httpclient.HTTPRequest对象。如果是url，fetch会自己构造一个HTTPRequest对象。

HTTPRequest

HTTP请求类，HTTPRequest的构造函数可以接收众多构造参数，最常用的如下：

- **url** (string) – 要访问的url，此参数必传，除此之外均为可选参数
- **method** (string) – HTTP访问方式，如“GET”或“POST”，默认为GET方式
- **headers** (HTTPHeaders or dict) – 附加的HTTP协议头
- **body** – HTTP请求的请求体

HTTPResponse

HTTP响应类，其常用属性如下：

- **code**: HTTP状态码，如 200 或 404
- **reason**: 状态码描述信息
- **body**: 响应体字符串
- **error**: 异常（可有可无）

基于gen.coroutine的协程异步

```
from tornado import web,httpclient,gen,ioloop
import json
class Home(web.RequestHandler):
    @gen.coroutine
    def get(self):
        http = httpclient.AsyncHTTPClient()
        response = yield http.fetch("http://ip-api.com/json/123.112.18.111?
lang=zh-CN")
        if response.error:
            self.send_error(500)
        else:
            data = json.loads(response.body)
            if 'success' == data["status"]:
                self.write("国家：%s 省份：%s 城市：%s" % (data["country"],
data["regionName"], data["city"]))
            else:
                self.write("查询IP信息错误")

# 设置路由列表
urls = [
    (r"/", Home),
]

if __name__ == "__main__":
    # 创建应用实例对象
    app = web.Application(urls, debug=True)
    # 设置监听的端口和地址
    app.listen(port=8888)
    # ioloop，全局的tornado事件循环，是服务器的引擎核心，start表示创建IO事件循环
    ioloop.IOLoop.current().start()
```

将异步Web请求单独抽取出来

```
from tornado import web,httpclient,gen,ioloop
import json
class Home(web.RequestHandler):
    @gen.coroutine
    def get(self):
        http = httpclient.AsyncHTTPClient()
        rep = yield self.get_ip_info("123.112.18.111")
        if 'success' == rep["status"]:
            self.write(u"国家：%s 省份：%s 城市：%s" % (rep["country"],
rep["regionName"], rep["city"]))
        else:
            self.write("查询IP信息错误")
```

```
    @gen.coroutine
    def get_ip_info(self, ip):
        http = httpclient.AsyncHTTPClient()
        response = yield http.fetch("http://ip-api.com/json/%s?lang=zh-CN" % ip)
        if response.error:
            rep = {"status":"fail"}
        else:
            rep = json.loads(response.body)
        raise gen.Return(rep)  # 此处需要注意

# 设置路由列表
urls = [
    (r"/", Home),
]

if __name__ == "__main__":
    # 创建应用实例对象
    app = web.Application(urls, debug=True)
    # 设置监听的端口和地址
    app.listen(port=8888)
    # ioloop，全局的tornado事件循环，是服务器的引擎核心，start表示创建IO事件循环
    ioloop.IOLoop.current().start()
```

并行协程

Tornado可以同时执行多个异步，并发的异步可以使用列表或字典，如下：

```
from tornado import web,httpclient,gen,ioloop
import json
class Home(web.RequestHandler):
    @gen.coroutine
    def get(self):
        ips = ["123.112.18.111",
               "112.112.233.89",
               "119.112.23.3",
               "120.223.70.76"]
        rep1, rep2 = yield [self.get_ip_info(ips[0]), self.get_ip_info(ips[1])]
        self.write_response(ips[0], rep1)
        self.write_response(ips[1], rep2)
        rep_dict = yield dict(rep3=self.get_ip_info(ips[2]),
rep4=self.get_ip_info(ips[3]))
        self.write_response(ips[2], rep_dict['rep3'])
        self.write_response(ips[3], rep_dict['rep4'])

    def write_response(self,ip, rep):
        if 'success' == rep["status"]:
            self.write("IP:%s 国家: %s 省份: %s 城市: %s<br>" % (ip,rep["country"],
rep["regionName"], rep["city"]))
        else:
            self.write("查询IP信息错误<br>")
    @gen.coroutine
    def get_ip_info(self, ip):
        http = httpclient.AsyncHTTPClient()
        response = yield http.fetch("http://ip-api.com/json/%s?lang=zh-CN" % ip)
```

```
        if response.error:
            rep = {"status":"fail"}
        else:
            rep = json.loads(response.body)
        raise gen.Return(rep)  # 此处需要注意

# 设置路由列表
urls = [
    (r"/", Home),
]

if __name__ == "__main__":
    # 创建应用实例对象
    app = web.Application(urls, debug=True)
    # 设置监听的端口和地址
    app.listen(port=8888)
    # ioloop，全局的tornado事件循环，是服务器的引擎核心，start表示创建IO事件循环
    ioloop.IOLoop.current().start()
```

Tornado的WebSocket

WebSocket是HTML5规范中新提出的客户端-服务器通讯协议，协议本身使用新的ws://URL格式。

WebSocket 是独立的、创建在 TCP 上的协议，和 HTTP 的唯一关联是使用 HTTP 协议的101状态码进行协议切换，使用的 TCP 端口是80，可以用于绕过大多数防火墙的限制。

WebSocket 使得客户端和服务器之间的数据交换变得更加简单，允许服务端直接向客户端推送数据而不需要客户端进行请求，两者之间可以创建持久性的连接，并允许数据进行双向传送。

Tornado提供支持WebSocket的模块是tornado.websocket，其中提供了一个WebSocketHandler类用来处理通讯。

常用方法

open()

当一个WebSocket连接建立后被调用。

on_message(message)

当客户端发送消息message过来时被调用，**注意此方法必须被重写。**

on_close()

当WebSocket连接关闭后被调用。

write_message(message, binary=False)

向客户端发送消息messagea，message可以是字符串或字典（字典会被转为json字符串）。若binary为False，则message以utf8编码发送；二进制模式（binary=True）时，可发送任何字节码。

close()

关闭WebSocket连接。

check_origin(origin)

判断源origin，对于符合条件（返回判断结果为True）的请求源origin允许其连接，否则返回403。可以重写此方法来解决WebSocket的跨域请求（如始终return True）。

快速使用

server.py，代码：

```
from tornado import web,ioloop
from tornado.websocket import WebSocketHandler

class Index(web.RequestHandler):
    def get(self):
        self.render("templates/index.html")

class Home(WebSocketHandler):

    def open(self):
        self.write_message("欢迎来到socket.html")

    def on_message(self, message):
        print("接收数据：%s" % message)

    def on_close(self):
        print("socket连接断开")

    def check_origin(self, origin):
        return True  # 允许WebSocket的跨域请求

# 设置路由列表
urls = [
    (r"/", Index),
    (r"/home", Home),
]

if __name__ == "__main__":
    # 创建应用实例对象
    app = web.Application(urls, debug=True)
    # 设置监听的端口和地址
    app.listen(port=8888)
    # ioloop，全局的tornado事件循环，是服务器的引擎核心，start表示创建IO事件循环
    ioloop.IOLoop.current().start()
```

tempales/index.html，代码：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title></title>
</head>
<body>
```

```
<div id="content"></div>
<script>
    var ws = new WebSocket("ws://127.0.0.1:8888/home"); // 新建一个ws连接
    ws.onopen = function() {  // 连接建立好后的回调
       ws.send("Hello, world");  // 向建立的连接发送消息
    };
    ws.onmessage = function (evt) {  // 收到服务器发送的消息后执行的回调
       content.innerHTML+=evt.data+"<br>";  // 接收的消息内容在事件参数evt的data属性中
    };
</script>
</body>
</html>
```

案例：聊天室

server.py，代码：

```
from tornado import web,ioloop,httpserver,options
import datetime

from tornado.web import RequestHandler
from tornado.websocket import WebSocketHandler

class Index(RequestHandler):
    def get(self):
        self.render("templates/index.html")

class Chat(WebSocketHandler):
    users = set()  # 用来存放用户的容器

    def open(self):
        self.users.add(self)  # 建立连接后添加用户到容器中
        for u in self.users:  # 向已在线用户发送消息
            u.write_message("[%s]-[%s]-登录" % (self.request.remote_ip,
datetime.datetime.now().strftime("%H:%M:%S")))

    def on_message(self, message):
        for u in self.users:  # 向在线用户广播消息
            u.write_message("[%s]-[%s]-发送：%s" % (self.request.remote_ip,
datetime.datetime.now().strftime("%H:%M:%S"), message))

    def on_close(self):
        self.users.remove(self) # 用户关闭连接后从容器中移除用户
        for u in self.users:
            u.write_message("[%s]-[%s]-退出" % (self.request.remote_ip,
datetime.datetime.now().strftime("%H:%M:%S")))

    def check_origin(self, origin):
        return True  # 允许WebSocket的跨域请求

urls = [
    (r"/", Index),
    (r"/chat", Chat),
]


if __name__ == '__main__':
```

```
    # 创建应用实例对象
    app = web.Application(urls)
    server = httpserver.HTTPServer(app)
    # 设置监听的端口和地址
    server.listen(port=8888,address="0.0.0.0")
    server.start(1)
    # ioloop，全局的tornado事件循环，是服务器的引擎核心，start表示创建IO事件循环
    ioloop.IOLoop.current().start()
```

templates/index.html，代码：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>聊天室</title>
</head>
<body>
    <div>
        <textarea id="msg"></textarea>
        <button onclick="sendMsg()">发送</button>
    </div>
    <div id="content" style="height:500px;overflow:auto;"></div>
    <script>
        var ws = new WebSocket("ws://127.0.0.1:8888/chat");
        ws.onmessage = function(message) {
            console.log("接收数据:", message);
            content.innerHTML +="<p>" + message.data + "</p>";
        };

        function sendMsg() {
            console.log("发送数据:", msg.value);
            ws.send(msg.value);
            msg.value = "";
        }
    </script>
</body>
</html>
```