**引言**

**Django的部署方式**

以Django为代表的python web应用部署时采用wsgi协议与服务器对接（被服务器托管），而这类服务器通常都是基于**多线程**的，也就是说**每一个网络请求服务器都会有一个对应的线程来用web应用（如Django）进行处理**。

**考虑两类应用场景**

1. **用户量大，高并发**

如秒杀抢购、双十一某宝购物、春节抢火车票

1. **大量的HTTP持久连接**

使用同一个TCP连接来发送和接收多个HTTP请求/应答，而不是为每一个新的请求/应答打开新的连接的方法。

对于HTTP 1.0，可以在请求的包头（Header）中添加**Connection: Keep-Alive**。

对于HTTP 1.1，所有的连接默认都是持久连接。

对于这两种场景，通常基于多线程的服务器很难应对。

**C10K问题**

对于前文提出的这种高并发问题，我们通常用C10K这一概念来描述。C10K—— **C**oncurrently handling **ten thousand** connections，即并发10000个连接。对于单台服务器而言，根本无法承担，而采用多台服务器分布式又意味着高昂的成本。如何解决C10K问题？

**Tornado**

Tornado在设计之初就考虑到了性能因素，旨在解决C10K问题，这样的设计使得其成为一个拥有非常高性能的**解决方案**（服务器与框架的集合体）。

**1 关于Tornado**

**知识点**

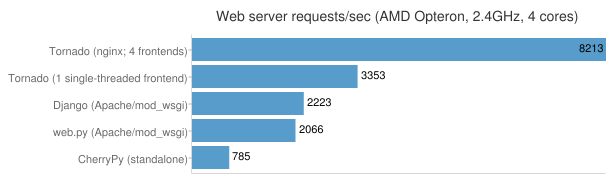
* 了解什么是Tornado框架
* 了解Tornado与Django的区别

**1.1 Tornado是为何物**

Tornado全称Tornado Web Server，是一个用Python语言写成的Web服务器兼Web应用框架，由FriendFeed公司在自己的网站FriendFeed中使用，被Facebook收购以后框架在2009年9月以开源软件形式开放给大众。

特点：

* 作为Web框架，是一个轻量级的Web框架，类似于另一个Python web框架Web.py，其拥有异步非阻塞IO的处理方式。
* 作为Web服务器，Tornado有较为出色的抗负载能力，官方用nginx反向代理的方式部署Tornado和其它Python web应用框架进行对比，结果最大浏览量超过第二名近40%。



性能： Tornado有着优异的性能。它试图解决C10k问题，即处理大于或等于一万的并发，下表是和一些其他Web框架与服务器的对比:

Tornado框架和服务器一起组成一个WSGI的全栈替代品。单独在WSGI容器中使用tornado网络框架或者tornaod http服务器，有一定的局限性，为了最大化的利用tornado的性能，推荐同时使用tornaod的网络框架和HTTP服务器。

## 1.2 Tornado与Django

### Django

Django是走**大而全**的方向，注重的是**高效开发**，它最出名的是其全自动化的管理后台：只需要使用起ORM，做简单的对象定义，它就能自动生成数据库结构、以及全功能的管理后台。

Django提供的方便，也意味着Django内置的ORM跟框架内的其他模块耦合程度高，应用程序必须使用Django内置的ORM，否则就不能享受到框架内提供的种种基于其ORM的便利。

* session功能
* 后台管理
* ORM

## Tornado

Tornado走的是**少而精**的方向，注重的是**性能优越**，它最出名的是异步非阻塞的设计方式。

* HTTP服务器
* 异步编程
* WebSockets 长连接

**2 初识Tornado**

**知识点**

* Tornado的安装
* 了解Tornado的原理
* 掌握Tornado的基本写法
* 掌握Tornado的基本模块
  + tornado.web
  + tornado.ioloop
  + tornado.httpserver
  + tornado.options

**2.1 安装**

**自动安装**

1. 查看自己当前的环境是否已安装
2. $ pip list
3. 安装
4. $ pip install tornado

**手动安装**

1. 下载安装包tornado-4.3.tar.gz（<https://pypi.python.org/packages/source/t/tornado/tornado-4.3.tar.gz>）
2. $ tar xvzf tornado-4.3.tar.gz
3. $ cd tornado-4.3
4. $ python setup.py build
5. $ sudo python setup.py install

**关于使用平台的说明**

Tornado should run on any Unix-like platform, although for the best performance and scalability only Linux (with epoll) and BSD (with kqueue) are recommended for production deployment (even though Mac OS X is derived from BSD and supports kqueue, its networking performance is generally poor so it is recommended only for development use). Tornado will also run on Windows, although this configuration is not officially supported and is recommended only for development use.

Tornado应该运行在**类Unix平台**，在线上部署时为了最佳的性能和扩展性，仅推荐**Linux**和**BSD**（因为充分利用Linux的epoll工具和BSD的kqueue工具，是Tornado不依靠多进程/多线程而达到高性能的原因）。

对于Mac OS X，虽然也是衍生自BSD并且支持kqueue，但是其网络性能通常不太给力，因此仅推荐用于开发。

对于Windows，Tornado官方没有提供配置支持，但是也可以运行起来，不过仅推荐在开发中使用。

# 2.2 Hello Qianfeng

## 上代码

新建文件hello.py，代码如下：

# coding:utf-8

import tornado.web

import tornado.ioloop

class IndexHandler(tornado.web.RequestHandler):

"""主路由处理类"""

def get(self):

"""对应http的get请求方式"""

self.write("Hello Qianfeng!")

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

app = tornado.web.Application([

(r"/", IndexHandler),

])

app.listen(8000)

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

执行如下命令，开启tornado：

$ python hello.py

打开浏览器，输入网址127.0.0.1:8000（或localhost:8000），查看效果：

## 代码讲解

#### 1. tornado.web

tornado的基础web框架模块

* **RequestHandler**

封装了对应一个请求的所有信息和方法，write(响应信息)就是写响应信息的一个方法；对应每一种http请求方式（get、post等），把对应的处理逻辑写进同名的成员方法中（如对应get请求方式，就将对应的处理逻辑写在get()方法中），当没有对应请求方式的成员方法时，会返回“**405: Method Not Allowed**”错误。

我们将代码中定义的get()方法更改为post()后，再用浏览器重新访问（浏览器地址栏中输入网址访问的方式为get请求方式），演示如下：

# coding:utf-8

import tornado.web

import tornado.ioloop

class IndexHandler(tornado.web.RequestHandler):

"""主路由处理类"""

def post(self): # 我们修改了这里

"""对应http的post请求方式"""

self.write("Hello Qianfeng!")

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

app = tornado.web.Application([

(r"/", IndexHandler),

])

app.listen(8000)

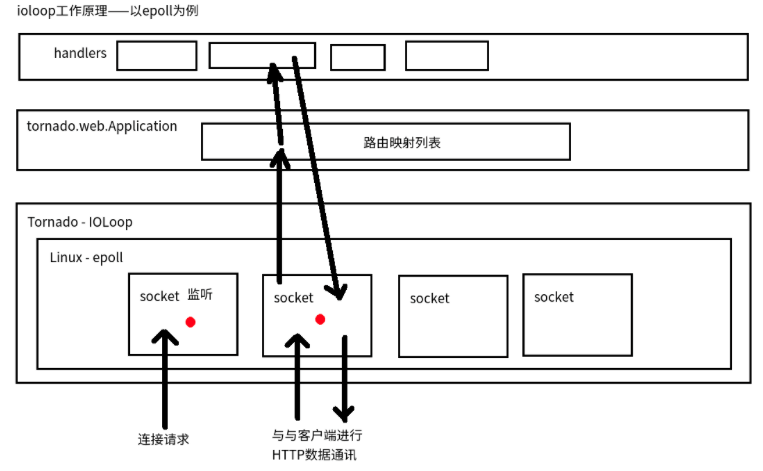
tornado.ioloop.IOLoop.current().start()

* **Application**

Tornado Web框架的核心应用类，是与服务器对接的接口，里面保存了路由信息表，其初始化接收的第一个参数就是一个路由信息映射元组的列表；其listen(端口)方法用来创建一个http服务器实例，并绑定到给定端口（**注意：此时服务器并未开启监听**）。

#### 2. tornado.ioloop

tornado的核心io循环模块，封装了Linux的epoll和BSD的kqueue，tornado高性能的基石。 以Linux的epoll为例，其原理如下图：



* **IOLoop.current()**

返回当前线程的IOLoop实例。

* **IOLoop.start()**

启动IOLoop实例的I/O循环,同时服务器监听被打开。

## 总结Tornado Web程序编写思路

1. 创建web应用实例对象，第一个初始化参数为路由映射列表。
2. 定义实现路由映射列表中的handler类。
3. 创建服务器实例，绑定服务器端口。
4. 启动当前线程的IOLoop。

# 2.3 httpserver

上一节我们说在tornado.web.Application.listen()（示例代码中的app.listen(8000)）的方法中，创建了一个http服务器示例并绑定到给定端口，我们能不能自己动手来实现这一部分功能呢？

现在我们修改上一示例代码如下：

# coding:utf-8

import tornado.web

import tornado.ioloop

import tornado.httpserver # 新引入httpserver模块

class IndexHandler(tornado.web.RequestHandler):

"""主路由处理类"""

def get(self):

"""对应http的get请求方式"""

self.write("Hello Qianfeng!")

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

app = tornado.web.Application([

(r"/", IndexHandler),

])

# ------------------------------

# 我们修改这个部分

# app.listen(8000)

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

http\_server.listen(8000)

# ------------------------------

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

在这一修改版本中，我们引入了tornado.httpserver模块，顾名思义，它就是tornado的HTTP服务器实现。

我们创建了一个HTTP服务器实例http\_server，因为服务器要服务于我们刚刚建立的web应用，将接收到的客户端请求通过web应用中的路由映射表引导到对应的handler中，所以在构建http\_server对象的时候需要传出web应用对象app。http\_server.listen(8000)将服务器绑定到8000端口。

实际上一版代码中app.listen(8000)正是对这一过程的简写。

## 单进程与多进程

我们刚刚实现的都是**单进程**，我们也可以**一次启动多个进程**，修改上面的代码如下：

# coding:utf-8

import tornado.web

import tornado.ioloop

import tornado.httpserver

class IndexHandler(tornado.web.RequestHandler):

"""主路由处理类"""

def get(self):

"""对应http的get请求方式"""

self.write("Hello Qianfeng!")

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

app = tornado.web.Application([

(r"/", IndexHandler),

])

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

# -----------修改----------------

http\_server.bind(8000)

http\_server.start(0)

# ------------------------------

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

http\_server.bind(port)方法是将服务器绑定到指定端口。

http\_server.start(num\_processes=1)方法指定开启几个进程，参数num\_processes默认值为1，即默认仅开启一个进程；如果num\_processes为None或者<=0，则自动根据机器硬件的cpu核芯数创建同等数目的子进程；如果num\_processes>0，则创建num\_processes个子进程。

本例中，我们使用http\_server.start(0)，而我的虚拟机设定cpu核数为2，演示结果：

我们在前面写的http\_server.listen(8000)实际上就等同于：

http\_server.bind(8000)

http\_server.start(1)

## 说明

#### 1.关于app.listen()

**app.listen()这个方法只能在单进程模式中使用。**

对于app.listen()与手动创建HTTPServer实例

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

http\_server.listen(8000)

这两种方式，建议大家先使用后者即创建HTTPServer实例的方式，因为其对于理解tornado web应用工作流程的完整性有帮助，便于大家记忆tornado开发的模块组成和程序结构；在熟练使用后，可以改为简写。

#### 2.关于多进程

虽然tornado给我们提供了一次开启多个进程的方法，但是由于：

* 每个子进程都会从父进程中复制一份IOLoop实例，如过在创建子进程前我们的代码动了IOLoop实例，那么会影响到每一个子进程，势必会干扰到子进程IOLoop的工作；
* 所有进程是由一个命令一次开启的，也就无法做到在不停服务的情况下更新代码；
* 所有进程共享同一个端口，想要分别单独监控每一个进程就很困难。

不建议使用这种多进程的方式，而是**手动开启多个进程，并且绑定不同的端口**。

# 2.4 options

在前面的示例中我们都是将服务端口的参数写死在程序中，很不灵活。

tornado为我们提供了一个便捷的工具，**tornado.options模块——全局参数定义、存储、转换**。

## tornado.options.define()

用来定义options选项变量的方法，定义的变量可以在全局的tornado.options.options中获取使用，传入参数：

* **name** 选项变量名，**须保证全局唯一性**，否则会报“Option 'xxx' already defined in ...”的错误；
* **default**　选项变量的默认值，如不传默认为None；
* **type** 选项变量的类型，从命令行或配置文件导入参数的时候tornado会根据这个类型转换输入的值，转换不成功时会报错，可以是str、float、int、datetime、timedelta中的某个，若未设置则根据default的值自动推断，若default也未设置，那么不再进行转换。**可以通过利用设置type类型字段来过滤不正确的输入。**
* **multiple** 选项变量的值是否可以为多个，布尔类型，默认值为False，如果multiple为True，那么设置选项变量时值与值之间用英文逗号分隔，而选项变量则是一个list列表（若默认值和输入均未设置，则为空列表[]）。
* **help** 选项变量的帮助提示信息，在命令行启动tornado时，通过加入命令行参数 --help　可以查看所有选项变量的信息（注意，代码中需要加入tornado.options.parse\_command\_line()）。

## tornado.options.options

全局的options对象，所有定义的选项变量都会作为该对象的属性。

## tornado.options.parse\_command\_line()

转换命令行参数，并将转换后的值对应的设置到全局options对象相关属性上。追加命令行参数的方式是--myoption=myvalue

新建opt.py，我们用代码来看一下如何使用：

# coding:utf-8

import tornado.web

import tornado.ioloop

import tornado.httpserver

import tornado.options # 新导入的options模块

tornado.options.define("port", default=8000, type=int, help="run server on the given port.") # 定义服务器监听端口选项

tornado.options.define("qianfeng", default=[], type=str, multiple=True, help="qianfeng subjects.") # 无意义，演示多值情况

class IndexHandler(tornado.web.RequestHandler):

"""主路由处理类"""

def get(self):

"""对应http的get请求方式"""

self.write("Hello Qianfeng!")

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

tornado.options.parse\_command\_line()

print tornado.options.options.qianfeng # 输出多值选项

app = tornado.web.Application([

(r"/", IndexHandler),

])

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

http\_server.listen(tornado.options.options.port)

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

执行如下命令开启程序：

$ python opt.py --port=9000 --qianfeng=python,c++,java,php,ios

效果如下：

## tornado.options.parse\_config\_file(path)

从配置文件导入option，配置文件中的选项格式如下：

myoption = "myvalue"

myotheroption = "myothervalue"

我们用代码来看一下如何使用，新建配置文件config，注意字符串和列表按照python的语法格式：

port = 8000

qianfeng = ["python","c++","java","php","ios"]

修改opt.py文件：

# coding:utf-8

import tornado.web

import tornado.ioloop

import tornado.httpserver

import tornado.options # 新导入的options模块

tornado.options.define("port", default=8000, type=int, help="run server on the given port.") # 定义服务器监听端口选项

tornado.options.define("qianfeng", default=[], type=str, multiple=True, help="qianfeng subjects.") # 无意义，演示多值情况

class IndexHandler(tornado.web.RequestHandler):

"""主路由处理类"""

def get(self):

"""对应http的get请求方式"""

self.write("Hello Qianfeng!")

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

tornado.options.parse\_config\_file("./config") # 仅仅修改了此处

print tornado.options.options.qianfeng # 输出多值选项

app = tornado.web.Application([

(r"/", IndexHandler),

])

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

http\_server.listen(tornado.options.options.port)

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

## 说明

#### 1. 日志

当我们在代码中调用parse\_command\_line()或者parse\_config\_file()的方法时，tornado会默认为我们配置标准logging模块，即默认开启了日志功能，并向标准输出（屏幕）打印日志信息。

如果想关闭tornado默认的日志功能，可以在命令行中添加--logging=none 或者在代码中执行如下操作:

from tornado.options import options, parse\_command\_line

options.logging = None

parse\_command\_line()

#### 2. 配置文件

我们看到在使用prase\_config\_file()的时候，配置文件的书写格式仍需要按照python的语法要求，其优势是可以直接将配置文件的参数转换设置到全局对象tornado.options.options中；然而，其不方便的地方在于需要在代码中调用tornado.options.define()来定义选项，而且不支持字典类型，故而在实际应用中大都不使用这种方法。

在使用配置文件的时候，通常会新建一个python文件（如config.py），然后在里面直接定义python类型的变量（可以是字典类型）；在需要配置文件参数的地方，将config.py作为模块导入，并使用其中的变量参数。

如config.py文件：

# conding:utf-8

# Tornado app配置

settings = {

'template\_path': os.path.join(os.path.dirname(\_\_file\_\_), 'templates'),

'static\_path': os.path.join(os.path.dirname(\_\_file\_\_), 'statics'),

'cookie\_secret':'0Q1AKOKTQHqaa+N80XhYW7KCGskOUE2snCW06UIxXgI=',

'xsrf\_cookies':False,

'login\_url':'/login',

'debug':True,

}

# 日志

log\_path = os.path.join(os.path.dirname(\_\_file\_\_), 'logs/log')

使用config.py的模块中导入config，如下：

# conding:utf-8

import tornado.web

import config

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

app = tornado.web.Application([], \*\*config.settings)

...

# 2.5 练习

1. 尝试解释清Tornado利用epoll机制实现支持高并发的原因。
2. 能够不参考课件熟练默写出Tornado的基本代码案例。

**3 深入Tornado**

**知识点**

* Application设置
* debug模式
* 路由设置扩展
* RequestHandler的使用
  + 输入方法
  + 输出方法
  + 可重写接口

**3.1 Application**

**settings**

前面的学习中，我们在创建tornado.web.Application的对象时，传入了第一个参数——路由映射列表。实际上Application类的构造函数还接收很多关于tornado web应用的配置参数，在后面的学习中我们用到的地方会为大家介绍。

我们先来学习一个参数：

**debug**，设置tornado是否工作在调试模式，默认为False即工作在生产模式。当设置debug=True 后，tornado会工作在调试/开发模式，在此种模式下，tornado为方便我们开发而提供了几种特性：

* **自动重启**，tornado应用会监控我们的源代码文件，当有改动保存后便会重启程序，这可以减少我们手动重启程序的次数。需要注意的是，一旦我们保存的更改有错误，自动重启会导致程序报错而退出，从而需要我们保存修正错误后手动启动程序。这一特性也可单独通过autoreload=True设置；
* **取消缓存编译的模板**，可以单独通过compiled\_template\_cache=False来设置；
* **取消缓存静态文件hash值**，可以单独通过static\_hash\_cache=False来设置；
* **提供追踪信息**，当RequestHandler或者其子类抛出一个异常而未被捕获后，会生成一个包含追踪信息的页面，可以单独通过serve\_traceback=True来设置。

使用debug参数的方法：

import tornado.web

app = tornado.web.Application([], debug=True)

**路由映射**

先前我们在构建路由映射列表的时候，使用的是二元元组，如：

[(r"/", IndexHandler),]

对于这个映射列表中的路由，实际上还可以传入多个信息，如：

[

(r"/", Indexhandler),

(r"/cpp", QianfengHandler, {"subject":"c++"}),

url(r"/python", QianfengHandler, {"subject":"python"}, name="python\_url")

]

对于路由中的字典，会传入到对应的RequestHandler的initialize()方法中：

from tornado.web import RequestHandler

class QianfengHandler(RequestHandler):

def initialize(self, subject):

self.subject = subject

def get(self):

self.write(self.subject)

对于路由中的name字段，注意此时不能再使用元组，而应使用tornado.web.url来构建。name是给该路由起一个名字，可以通过调用RequestHandler.reverse\_url(name)来获取该名子对应的url。

# coding:utf-8

import tornado.web

import tornado.ioloop

import tornado.httpserver

import tornado.options

from tornado.options import options, define

from tornado.web import url, RequestHandler

define("port", default=8000, type=int, help="run server on the given port.")

class IndexHandler(RequestHandler):

def get(self):

python\_url = self.reverse\_url("python\_url")

self.write('<a href="%s">qianfeng</a>' %

python\_url)

class QianfengHandler(RequestHandler):

def initialize(self, subject):

self.subject = subject

def get(self):

self.write(self.subject)

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

tornado.options.parse\_command\_line()

app = tornado.web.Application([

(r"/", Indexhandler),

(r"/cpp", QianfengHandler, {"subject":"c++"}),

url(r"/python", QianfengHandler, {"subject":"python"}, name="python\_url")

],

debug = True)

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

http\_server.listen(options.port)

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

# 3.2 输入

下面几节主要讲解tornado.web.RequestHandler。

回想一下，利用HTTP协议向服务器传参有几种途径？

* 查询字符串（query string)，形如key1=value1&key2=value2；
* 请求体（body）中发送的数据，比如表单数据、json；
* 提取uri的特定部分，如/blogs/2016/09/0001，可以在服务器端的路由中用正则表达式截取；
* 在http报文的头（header）中增加自定义字段，如X-XSRFToken=qianfeng。

我们现在来看下tornado中为我们提供了哪些方法来获取请求的信息。

## 1. 获取查询字符串参数

#### get\_query\_argument(name, default=\_ARG\_DEFAULT, strip=True)

从请求的查询字符串中返回指定参数name的值，如果出现多个同名参数，则返回最后一个的值。

default为设值未传name参数时返回的默认值，如若default也未设置，则会抛出tornado.web.MissingArgumentError异常。

strip表示是否过滤掉左右两边的空白字符，默认为过滤。

#### get\_query\_arguments(name, strip=True)

从请求的查询字符串中返回指定参数name的值，注意返回的是list列表（即使对应name参数只有一个值）。若未找到name参数，则返回空列表[]。

strip同前，不再赘述。

## 2. 获取请求体参数

#### get\_body\_argument(name, default=\_ARG\_DEFAULT, strip=True)

从请求体中返回指定参数name的值，如果出现多个同名参数，则返回最后一个的值。

default与strip同前，不再赘述。

#### get\_body\_arguments(name, strip=True)

从请求体中返回指定参数name的值，注意返回的是list列表（即使对应name参数只有一个值）。若未找到name参数，则返回空列表[]。

strip同前，不再赘述。

#### 说明

对于请求体中的数据要求为字符串，且格式为表单编码格式（与url中的请求字符串格式相同），即key1=value1&key2=value2，HTTP报文头Header中的"Content-Type"为application/x-www-form-urlencoded 或 multipart/form-data。

## 3. 前两类方法的整合

#### get\_argument(name, default=\_ARG\_DEFAULT, strip=True)

从请求体和查询字符串中返回指定参数name的值，如果出现多个同名参数，则返回最后一个的值。

default与strip同前，不再赘述。

#### get\_arguments(name, strip=True)

从请求体和查询字符串中返回指定参数name的值，注意返回的是list列表（即使对应name参数只有一个值）。若未找到name参数，则返回空列表[]。

strip同前，不再赘述。

#### 说明

对于请求体中数据的要求同前。 **这两个方法最常用**。

用代码来看上述六中方法的使用：

# coding:utf-8

import tornado.web

import tornado.ioloop

import tornado.httpserver

import tornado.options

from tornado.options import options, define

from tornado.web import RequestHandler, MissingArgumentError

define("port", default=8000, type=int, help="run server on the given port.")

class IndexHandler(RequestHandler):

def post(self):

query\_arg = self.get\_query\_argument("a")

query\_args = self.get\_query\_arguments("a")

body\_arg = self.get\_body\_argument("a")

body\_args = self.get\_body\_arguments("a", strip=False)

arg = self.get\_argument("a")

args = self.get\_arguments("a")

default\_arg = self.get\_argument("b", "qianfeng")

default\_args = self.get\_arguments("b")

try:

missing\_arg = self.get\_argument("c")

except MissingArgumentError as e:

missing\_arg = "We catched the MissingArgumentError!"

print e

missing\_args = self.get\_arguments("c")

rep = "query\_arg:%s<br/>" % query\_arg

rep += "query\_args:%s<br/>" % query\_args

rep += "body\_arg:%s<br/>" % body\_arg

rep += "body\_args:%s<br/>" % body\_args

rep += "arg:%s<br/>" % arg

rep += "args:%s<br/>" % args

rep += "default\_arg:%s<br/>" % default\_arg

rep += "default\_args:%s<br/>" % default\_args

rep += "missing\_arg:%s<br/>" % missing\_arg

rep += "missing\_args:%s<br/>" % missing\_args

self.write(rep)

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

tornado.options.parse\_command\_line()

app = tornado.web.Application([

(r"/", IndexHandler),

])

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

http\_server.listen(options.port)

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

#### 注意：以上方法返回的都是unicode字符串

## 思考

1. 什么时候设置default，什么时候不设置default？
2. 什么时候使用strip，亦即什么时候要截断空白字符，什么时候不需要？

## 4. 关于请求的其他信息

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：
* {
* "form\_filename1":[<tornado.httputil.HTTPFile>, <tornado.httputil.HTTPFile>],
* "form\_filename2":[<tornado.httputil.HTTPFile>,],
* ...
* }

tornado.httputil.HTTPFile是接收到的文件对象，它有三个属性：

* + filename 文件的实际名字，与form\_filename1不同，字典中的键名代表的是表单对应项的名字；
  + body 文件的数据实体；
  + content\_type 文件的类型。 这三个对象属性可以像字典一样支持关键字索引，如request.files["form\_filename1"][0]["body"]。

我们来实现一个上传文件并保存在服务器本地的小程序upload.py：

# coding:utf-8

import tornado.web

import tornado.ioloop

import tornado.httpserver

import tornado.options

from tornado.options import options, define

from tornado.web import RequestHandler

define("port", default=8000, type=int, help="run server on the given port.")

class IndexHandler(RequestHandler):

def get(self):

self.write("hello qianfeng.")

class UploadHandler(RequestHandler):

def post(self):

files = self.request.files

img\_files = files.get('img')

if img\_files:

img\_file = img\_files[0]["body"]

file = open("./qianfeng", 'w+')

file.write(img\_file)

file.close()

self.write("OK")

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

tornado.options.parse\_command\_line()

app = tornado.web.Application([

(r"/", IndexHandler),

(r"/upload", UploadHandler),

])

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

http\_server.listen(options.port)

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

## 5. 正则提取uri

tornado中对于路由映射也支持正则提取uri，提取出来的参数会作为RequestHandler中对应请求方式的成员方法参数。若在正则表达式中定义了名字，则参数按名传递；若未定义名字，则参数按顺序传递。提取出来的参数会作为对应请求方式的成员方法的参数。

# coding:utf-8

import tornado.web

import tornado.ioloop

import tornado.httpserver

import tornado.options

from tornado.options import options, define

from tornado.web import RequestHandler

define("port", default=8000, type=int, help="run server on the given port.")

class IndexHandler(RequestHandler):

def get(self):

self.write("hello qianfeng.")

class SubjectCityHandler(RequestHandler):

def get(self, subject, city):

self.write(("Subject: %s<br/>City: %s" % (subject, city)))

class SubjectDateHandler(RequestHandler):

def get(self, date, subject):

self.write(("Date: %s<br/>Subject: %s" % (date, subject)))

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

tornado.options.parse\_command\_line()

app = tornado.web.Application([

(r"/", IndexHandler),

(r"/sub-city/(.+)/([a-z]+)", SubjectCityHandler), # 无名方式

(r"/sub-date/(?P<subject>.+)/(?P<date>\d+)", SubjectDateHandler), #　命名方式

])

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

http\_server.listen(options.port)

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

**建议：提取多个值时最好用命名方式。**

# 3.3 输出

## 1. write(chunk)

将chunk数据写到输出缓冲区。如我们在之前的示例代码中写的：

class IndexHandler(RequestHandler):

def get(self):

self.write("hello qianfeng!")

#### 想一想，可不可以在同一个处理方法中多次使用write方法？

**下面的代码会出现什么效果？**

class IndexHandler(RequestHandler):

def get(self):

self.write("hello qianfeng 1!")

self.write("hello qianfeng 2!")

self.write("hello qianfeng 3!")

write方法是写到缓冲区的，我们可以像写文件一样多次使用write方法不断追加响应内容，最终所有写到缓冲区的内容一起作为本次请求的响应输出。

#### 想一想，如何利用write方法写json数据？

import json

class IndexHandler(RequestHandler):

def get(self):

stu = {

"name":"zhangsan",

"age":24,

"gender":1,

}

stu\_json = json.dumps(stu)

self.write(stu\_json)

**实际上，我们可以不用自己手动去做json序列化，当write方法检测到我们传入的chunk参数是字典类型后，会自动帮我们转换为json字符串。**

class IndexHandler(RequestHandler):

def get(self):

stu = {

"name":"zhangsan",

"age":24,

"gender":1,

}

self.write(stu)

#### 两种方式有什么差异？

对比一下两种方式的响应头header中Content-Type字段，自己手动序列化时为Content-Type:text/html; charset=UTF-8，而采用write方法时为Content-Type:application/json; charset=UTF-8。

**write方法除了帮我们将字典转换为json字符串之外，还帮我们将Content-Type设置为application/json; charset=UTF-8。**

## 2. set\_header(name, value)

利用set\_header(name, value)方法，可以手动设置一个名为name、值为value的响应头header字段。

用set\_header方法来完成上面write所做的工作。

import json

class IndexHandler(RequestHandler):

def get(self):

stu = {

"name":"zhangsan",

"age":24,

"gender":1,

}

stu\_json = json.dumps(stu)

self.write(stu\_json)

self.set\_header("Content-Type", "application/json; charset=UTF-8")

## 3. set\_default\_headers()

该方法会在进入HTTP处理方法前先被调用，可以重写此方法来预先设置默认的headers。**注意：在HTTP处理方法中使用set\_header()方法会覆盖掉在set\_default\_headers()方法中设置的同名header。**

class IndexHandler(RequestHandler):

def set\_default\_headers(self):

print "执行了set\_default\_headers()"

# 设置get与post方式的默认响应体格式为json

self.set\_header("Content-Type", "application/json; charset=UTF-8")

# 设置一个名为qianfeng、值为python的header

self.set\_header("qianfeng", "python")

def get(self):

print "执行了get()"

stu = {

"name":"zhangsan",

"age":24,

"gender":1,

}

stu\_json = json.dumps(stu)

self.write(stu\_json)

self.set\_header("qianfeng", "i love python") # 注意此处重写了header中的qianfeng字段

def post(self):

print "执行了post()"

stu = {

"name":"zhangsan",

"age":24,

"gender":1,

}

stu\_json = json.dumps(stu)

self.write(stu\_json)

终端中打印出的执行顺序：

get请求方式的响应header:

post请求方式的响应header:

## 4. set\_status(status\_code, reason=None)

为响应设置状态码。

参数说明：

* status\_code int类型，状态码，若reason为None，则状态码必须为下表中的。
* reason string类型，描述状态码的词组，若为None，则会被自动填充为下表中的内容。

| **Code** | **Enum Name** | **Details** |
| --- | --- | --- |
| 100 | CONTINUE | HTTP/1.1 RFC 7231, Section 6.2.1 |
| 101 | SWITCHING\_PROTOCOLS | HTTP/1.1 RFC 7231, Section 6.2.2 |
| 102 | PROCESSING | WebDAV RFC 2518, Section 10.1 |
| 200 | OK | HTTP/1.1 RFC 7231, Section 6.3.1 |
| 201 | CREATED | HTTP/1.1 RFC 7231, Section 6.3.2 |
| 202 | ACCEPTED | HTTP/1.1 RFC 7231, Section 6.3.3 |
| 203 | NON\_AUTHORITATIVE\_INFORMATION | HTTP/1.1 RFC 7231, Section 6.3.4 |
| 204 | NO\_CONTENT | HTTP/1.1 RFC 7231, Section 6.3.5 |
| 205 | RESET\_CONTENT | HTTP/1.1 RFC 7231, Section 6.3.6 |
| 206 | PARTIAL\_CONTENT | HTTP/1.1 RFC 7233, Section 4.1 |
| 207 | MULTI\_STATUS | WebDAV RFC 4918, Section 11.1 |
| 208 | ALREADY\_REPORTED | WebDAV Binding Extensions RFC 5842, Section 7.1 (Experimental) |
| 226 | IM\_USED | Delta Encoding in HTTP RFC 3229, Section 10.4.1 |
| 300 | MULTIPLE\_CHOICES | HTTP/1.1 RFC 7231, Section 6.4.1 |
| 301 | MOVED\_PERMANENTLY | HTTP/1.1 RFC 7231, Section 6.4.2 |
| 302 | FOUND | HTTP/1.1 RFC 7231, Section 6.4.3 |
| 303 | SEE\_OTHER | HTTP/1.1 RFC 7231, Section 6.4.4 |
| 304 | NOT\_MODIFIED | HTTP/1.1 RFC 7232, Section 4.1 |
| 305 | USE\_PROXY | HTTP/1.1 RFC 7231, Section 6.4.5 |
| 307 | TEMPORARY\_REDIRECT | HTTP/1.1 RFC 7231, Section 6.4.7 |
| 308 | PERMANENT\_REDIRECT | Permanent Redirect RFC 7238, Section 3 (Experimental) |
| 400 | BAD\_REQUEST | HTTP/1.1 RFC 7231, Section 6.5.1 |
| 401 | UNAUTHORIZED | HTTP/1.1 Authentication RFC 7235, Section 3.1 |
| 402 | PAYMENT\_REQUIRED | HTTP/1.1 RFC 7231, Section 6.5.2 |
| 403 | FORBIDDEN | HTTP/1.1 RFC 7231, Section 6.5.3 |
| 404 | NOT\_FOUND | HTTP/1.1 RFC 7231, Section 6.5.4 |
| 405 | METHOD\_NOT\_ALLOWED | HTTP/1.1 RFC 7231, Section 6.5.5 |
| 406 | NOT\_ACCEPTABLE | HTTP/1.1 RFC 7231, Section 6.5.6 |
| 407 | PROXY\_AUTHENTICATION\_REQUIRED | HTTP/1.1 Authentication RFC 7235, Section 3.2 |
| 408 | REQUEST\_TIMEOUT | HTTP/1.1 RFC 7231, Section 6.5.7 |
| 409 | CONFLICT | HTTP/1.1 RFC 7231, Section 6.5.8 |
| 410 | GONE | HTTP/1.1 RFC 7231, Section 6.5.9 |
| 411 | LENGTH\_REQUIRED | HTTP/1.1 RFC 7231, Section 6.5.10 |
| 412 | PRECONDITION\_FAILED | HTTP/1.1 RFC 7232, Section 4.2 |
| 413 | REQUEST\_ENTITY\_TOO\_LARGE | HTTP/1.1 RFC 7231, Section 6.5.11 |
| 414 | REQUEST\_URI\_TOO\_LONG | HTTP/1.1 RFC 7231, Section 6.5.12 |
| 415 | UNSUPPORTED\_MEDIA\_TYPE | HTTP/1.1 RFC 7231, Section 6.5.13 |
| 416 | REQUEST\_RANGE\_NOT\_SATISFIABLE | HTTP/1.1 Range Requests RFC 7233, Section 4.4 |
| 417 | EXPECTATION\_FAILED | HTTP/1.1 RFC 7231, Section 6.5.14 |
| 422 | UNPROCESSABLE\_ENTITY | WebDAV RFC 4918, Section 11.2 |
| 423 | LOCKED | WebDAV RFC 4918, Section 11.3 |
| 424 | FAILED\_DEPENDENCY | WebDAV RFC 4918, Section 11.4 |
| 426 | UPGRADE\_REQUIRED | HTTP/1.1 RFC 7231, Section 6.5.15 |
| 428 | PRECONDITION\_REQUIRED | Additional HTTP Status Codes RFC 6585 |
| 429 | TOO\_MANY\_REQUESTS | Additional HTTP Status Codes RFC 6585 |
| 431 | REQUEST\_HEADER\_FIELDS\_TOO\_LARGE Additional | HTTP Status Codes RFC 6585 |
| 500 | INTERNAL\_SERVER\_ERROR | HTTP/1.1 RFC 7231, Section 6.6.1 |
| 501 | NOT\_IMPLEMENTED | HTTP/1.1 RFC 7231, Section 6.6.2 |
| 502 | BAD\_GATEWAY | HTTP/1.1 RFC 7231, Section 6.6.3 |
| 503 | SERVICE\_UNAVAILABLE | HTTP/1.1 RFC 7231, Section 6.6.4 |
| 504 | GATEWAY\_TIMEOUT | HTTP/1.1 RFC 7231, Section 6.6.5 |
| 505 | HTTP\_VERSION\_NOT\_SUPPORTED | HTTP/1.1 RFC 7231, Section 6.6.6 |
| 506 | VARIANT\_ALSO\_NEGOTIATES | Transparent Content Negotiation in HTTP RFC 2295, Section 8.1 (Experimental) |
| 507 | INSUFFICIENT\_STORAGE | WebDAV RFC 4918, Section 11.5 |
| 508 | LOOP\_DETECTED | WebDAV Binding Extensions RFC 5842, Section 7.2 (Experimental) |
| 510 | NOT\_EXTENDED | An HTTP Extension Framework RFC 2774, Section 7 (Experimental) |
| 511 | NETWORK\_AUTHENTICATION\_REQUIRED | Additional HTTP Status Codes RFC 6585, Section 6 |

class Err404Handler(RequestHandler):

"""对应/err/404"""

def get(self):

self.write("hello qianfeng")

self.set\_status(404) # 标准状态码，不用设置reason

class Err210Handler(RequestHandler):

"""对应/err/210"""

def get(self):

self.write("hello qianfeng")

self.set\_status(210, "qianfeng error") # 非标准状态码，设置了reason

class Err211Handler(RequestHandler):

"""对应/err/211"""

def get(self):

self.write("hello qianfeng")

self.set\_status(211) # 非标准状态码，未设置reason，错误

## 5. redirect(url)

告知浏览器跳转到url。

class IndexHandler(RequestHandler):

"""对应/"""

def get(self):

self.write("主页")

class LoginHandler(RequestHandler):

"""对应/login"""

def get(self):

self.write('<form method="post"><input type="submit" value="登陆"></form>')

def post(self):

self.redirect("/")

## 6. send\_error(status\_code=500, \*\*kwargs)

抛出HTTP错误状态码status\_code，默认为500，kwargs为可变命名参数。使用send\_error抛出错误后tornado会调用write\_error()方法进行处理，并返回给浏览器处理后的错误页面。

class IndexHandler(RequestHandler):

def get(self):

self.write("主页")

self.send\_error(404, content="出现404错误")

**注意：默认的write\\_error()方法不会处理send\\_error抛出的kwargs参数，即上面的代码中content="出现404错误"是没有意义的。**

#### 尝试下面的代码会出现什么问题？

class IndexHandler(RequestHandler):

def get(self):

self.write("主页")

self.send\_error(404, content="出现404错误")

self.write("结束") # 我们在send\_error再次向输出缓冲区写内容

**注意：使用send\_error()方法后就不要再向输出缓冲区写内容了！**

## 7. write\_error(status\_code, \*\*kwargs)

用来处理send\_error抛出的错误信息并返回给浏览器错误信息页面。可以重写此方法来定制自己的错误显示页面。

class IndexHandler(RequestHandler):

def get(self):

err\_code = self.get\_argument("code", None) # 注意返回的是unicode字符串，下同

err\_title = self.get\_argument("title", "")

err\_content = self.get\_argument("content", "")

if err\_code:

self.send\_error(err\_code, title=err\_title, content=err\_content)

else:

self.write("主页")

def write\_error(self, status\_code, \*\*kwargs):

self.write(u"<h1>出错了，程序员GG正在赶过来！</h1>")

self.write(u"<p>错误名：%s</p>" % kwargs["title"])

self.write(u"<p>错误详情：%s</p>" % kwargs["content"])

**3.4 接口与调用顺序**

下面的接口方法是由tornado框架进行调用的，我们可以选择性的重写这些方法。

**1. initialize()**

对应每个请求的处理类Handler在构造一个实例后首先执行initialize()方法。在讲输入时提到，**路由映射中的第三个字典型参数会作为该方法的命名参数传递，**如：

class ProfileHandler(RequestHandler):

def initialize(self, database):

self.database = database

def get(self):

...

app = Application([

(r'/user/(.\*)', ProfileHandler, dict(database=database)),

])

**此方法通常用来初始化参数（对象属性），很少使用。**

**2. prepare()**

预处理，即在执行对应请求方式的HTTP方法（如get、post等）前先执行，**注意：不论以何种HTTP方式请求，都会执行prepare()方法**。

以预处理请求体中的json数据为例：

import json

class IndexHandler(RequestHandler):

def prepare(self):

if self.request.headers.get("Content-Type").startswith("application/json"):

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

else:

self.json\_dict = None

def post(self):

if self.json\_dict:

for key, value in self.json\_dict.items():

self.write("<h3>%s</h3><p>%s</p>" % (key, value))

def put(self):

if self.json\_dict:

for key, value in self.json\_dict.items():

self.write("<h3>%s</h3><p>%s</p>" % (key, value))

用post方式发送json数据时：

用put方式发送json数据时：

**3. HTTP方法**

| **方法** | **描述** |
| --- | --- |
| get | 请求指定的页面信息，并返回实体主体。 |
| head | 类似于get请求，只不过返回的响应中没有具体的内容，用于获取报头 |
| post | 向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改。 |
| delete | 请求服务器删除指定的内容。 |
| patch | 请求修改局部数据。 |
| put | 从客户端向服务器传送的数据取代指定的文档的内容。 |
| options | 返回给定URL支持的所有HTTP方法。 |

**4. on\_finish()**

在请求处理结束后调用，即在调用HTTP方法后调用。通常该方法用来进行资源清理释放或处理日志等。**注意：请尽量不要在此方法中进行响应输出。**

**5. set\_default\_headers()**

**6. write\_error()**

**7. 调用顺序**

我们通过一段程序来看上面这些接口的调用顺序。

class IndexHandler(RequestHandler):

def initialize(self):

print "调用了initialize()"

def prepare(self):

print "调用了prepare()"

def set\_default\_headers(self):

print "调用了set\_default\_headers()"

def write\_error(self, status\_code, \*\*kwargs):

print "调用了write\_error()"

def get(self):

print "调用了get()"

def post(self):

print "调用了post()"

self.send\_error(200) # 注意此出抛出了错误

def on\_finish(self):

print "调用了on\_finish()"

**在正常情况未抛出错误时，调用顺序为：**

1. set\_defautl\_headers()
2. initialize()
3. prepare()
4. HTTP方法
5. on\_finish()

**在有错误抛出时，调用顺序为：**

1. set\_default\_headers()
2. initialize()
3. prepare()
4. HTTP方法
5. **set\_default\_headers()**
6. write\_error()
7. on\_finish()

# 3.5 练习

1. 将Application的设置参数（目前只学习了debug）抽离为一个字典类型变量settings，并在构造Application对象时使用settings。
2. 熟练使用RequestHandler的各种输入输出方法。
3. 尝试抽象出BaseHandler基类，继承自RequestHandler，并在此基类中实现prepare（解析json数据）、write\_error两个接口。

**4 模板**

**知识点**

* 静态文件配置
  + static\_path
  + StaticFileHandler
* 模板使用
  + 变量与表达式
  + 控制语句
  + 函数
  + 块

# 4.1 静态文件

现在有一个预先写好的静态页面文件 （[下载静态文件资源](file:///C:\Users\Tina\Desktop\tornado%E7%92%87%E5%8F%A5%E6%AC%A2\c04\statics\template_statics.tar.gz)）， 我们来看下如何用tornado提供静态文件。

## static\_path

我们可以通过向web.Application类的构造函数传递一个名为**static\_path**的参数来告诉Tornado从文件系统的一个特定位置提供静态文件，如：

app = tornado.web.Application(

[(r'/', IndexHandler)],

static\_path=os.path.join(os.path.dirname(\_\_file\_\_), "statics"),

)

在这里，我们设置了一个当前应用目录下名为statics的子目录作为static\_path的参数。现在应用将以读取statics目录下的filename.ext来响应诸如/static/filename.ext的请求，并在响应的主体中返回。

#### 对于静态文件目录的命名，为了便于部署，建议使用static

对于我们提供的静态文件资源，可以通过http://127.0.0.1/static/html/index.html来访问。而且在index.html中引用的静态资源文件，我们给定的路径也符合/static/...的格式，故页面可以正常浏览。

<link href="/static/plugins/bootstrap/css/bootstrap.min.css" rel="stylesheet">

<link href="/static/plugins/font-awesome/css/font-awesome.min.css" rel="stylesheet">

<link href="/static/css/reset.css" rel="stylesheet">

<link href="/static/css/main.css" rel="stylesheet">

<link href="/static/css/index.css" rel="stylesheet">

<script src="/static/js/jquery.min.js"></script>

<script src="/static/plugins/bootstrap/js/bootstrap.min.js"></script>

<script src="/static/js/index.js"></script>

## StaticFileHandler

我们再看刚刚访问页面时使用的路径http://127.0.0.1/static/html/index.html，这中url显然对用户是不友好的，访问很不方便。我们可以通过**tornado.web.StaticFileHandler**来自由映射静态文件与其访问路径url。

tornado.web.StaticFileHandler是tornado预置的用来提供静态资源文件的handler。

import os

current\_path = os.path.dirname(\_\_file\_\_)

app = tornado.web.Application(

[

(r'^/()$', StaticFileHandler, {"path":os.path.join(current\_path, "statics/html"), "default\_filename":"index.html"}),

(r'^/view/(.\*)$', StaticFileHandler, {"path":os.path.join(current\_path, "statics/html")}),

],

static\_path=os.path.join(current\_path, "statics"),

)

* **path** 用来指明提供静态文件的根路径，并在此目录中寻找在路由中用正则表达式提取的文件名。
* **default\_filename** 用来指定访问路由中未指明文件名时，默认提供的文件。

现在，对于静态文件statics/html/index.html，可以通过三种方式进行访问：

1. <http://127.0.0.1/static/html/index.html>
2. <http://127.0.0.1/>
3. <http://127.0.0.1/view/index.html>

# 4.2 使用模板；

## 1. 路径与渲染

使用模板，需要仿照静态文件路径设置一样，向web.Application类的构造函数传递一个名为**template\_path**的参数来告诉Tornado从文件系统的一个特定位置提供模板文件，如：

app = tornado.web.Application(

[(r'/', IndexHandler)],

static\_path=os.path.join(os.path.dirname(\_\_file\_\_), "statics"),

template\_path=os.path.join(os.path.dirname(\_\_file\_\_), "templates"),

)

在这里，我们设置了一个当前应用目录下名为templates的子目录作为template\_path的参数。在handler中使用的模板将在此目录中寻找。

现在我们将静态文件目录statics/html中的index.html复制一份到templates目录中，此时文件目录结构为：

.

├── statics

│ ├── css

│ │ ├── index.css

│ │ ├── main.css

│ │ └── reset.css

│ ├── html

│ │ └── index.html

│ ├── images

│ │ ├── home01.jpg

│ │ ├── home02.jpg

│ │ ├── home03.jpg

│ │ └── landlord01.jpg

│ ├── js

│ │ ├── index.js

│ │ └── jquery.min.js

│ └── plugins

│ ├── bootstrap

│ │ └─...

│ └── font-awesome

│ └─...

├── templates

│ └── index.html

└── test.py

**在handler中使用render()方法来渲染模板并返回给客户端**。

class IndexHandler(RequestHandler):

def get(self):

self.render("index.html") # 渲染主页模板，并返回给客户端。

current\_path = os.path.dirname(\_\_file\_\_)

app = tornado.web.Application(

[

(r'^/$', IndexHandler),

(r'^/view/(.\*)$', StaticFileHandler, {"path":os.path.join(current\_path, "statics/html")}),

],

static\_path=os.path.join(current\_path, "statics"),

template\_path=os.path.join(os.path.dirname(\_\_file\_\_), "templates"),

)

## 2. 模板语法

### 2-1 变量与表达式

在tornado的模板中使用{{}}作为变量或表达式的占位符，使用render渲染后占位符{{}}会被替换为相应的结果值。

我们将index.html中的一条房源信息记录

<li class="house-item">

<a href=""><img src="/static/images/home01.jpg"></a>

<div class="house-desc">

<div class="landlord-pic"><img src="/static/images/landlord01.jpg"></div>

<div class="house-price">￥<span>398</span>/晚</div>

<div class="house-intro">

<span class="house-title">宽窄巷子+160平大空间+文化保护区双地铁</span>

<em>整套出租 - 5分/6点评 - 北京市丰台区六里桥地铁</em>

</div>

</div>

</li>

改为模板：

<li class="house-item">

<a href=""><img src="/static/images/home01.jpg"></a>

<div class="house-desc">

<div class="landlord-pic"><img src="/static/images/landlord01.jpg"></div>

<div class="house-price">￥<span>{{price}}</span>/晚</div>

<div class="house-intro">

<span class="house-title">{{title}}</span>

<em>整套出租 - {{score}}分/{{comments}}点评 - {{position}}</em>

</div>

</div>

</li>

渲染方式如下：

class IndexHandler(RequestHandler):

def get(self):

house\_info = {

"price": 398,

"title": "宽窄巷子+160平大空间+文化保护区双地铁",

"score": 5,

"comments": 6,

"position": "北京市丰台区六里桥地铁"

}

self.render("index.html", \*\*house\_info)

{{}}不仅可以包含变量，还可以是表达式，如：

<li class="house-item">

<a href=""><img src="/static/images/home01.jpg"></a>

<div class="house-desc">

<div class="landlord-pic"><img src="/static/images/landlord01.jpg"></div>

<div class="house-price">￥<span>{{p1 + p2}}</span>/晚</div>

<div class="house-intro">

<span class="house-title">{{"+".join(titles)}}</span>

<em>整套出租 - {{score}}分/{{comments}}点评 - {{position}}</em>

</div>

</div>

</li>

class IndexHandler(RequestHandler):

def get(self):

house\_info = {

"p1": 198,

"p2": 200,

"titles": ["宽窄巷子", "160平大空间", "文化保护区双地铁"],

"score": 5,

"comments": 6,

"position": "北京市丰台区六里桥地铁"

}

self.render("index.html", \*\*house\_info)

### 2-2 控制语句

可以在Tornado模板中使用Python条件和循环语句。控制语句以{\%和\%}包围，并以类似下面的形式被使用：

{% if page is None %}

或

{% if len(entries) == 3 %}

控制语句的大部分就像对应的Python语句一样工作，支持if、for、while，**注意end**:

{% if ... %} ... {% elif ... %} ... {% else ... %} ... {% end %}

{% for ... in ... %} ... {% end %}

{% while ... %} ... {% end %}

再次修改index.html:

<ul class="house-list">

{% if len(houses) > 0 %}

{% for house in houses %}

<li class="house-item">

<a href=""><img src="/static/images/home01.jpg"></a>

<div class="house-desc">

<div class="landlord-pic"><img src="/static/images/landlord01.jpg"></div>

<div class="house-price">￥<span>{{house["price"]}}</span>/晚</div>

<div class="house-intro">

<span class="house-title">{{house["title"]}}</span>

<em>整套出租 - {{house["score"]}}分/{{house["comments"]}}点评 - {{house["position"]}}</em>

</div>

</div>

</li>

{% end %}

{% else %}

对不起，暂时没有房源。

{% end %}

</ul>

python中渲染语句为：

class IndexHandler(RequestHandler):

def get(self):

houses = [

{

"price": 398,

"title": "宽窄巷子+160平大空间+文化保护区双地铁",

"score": 5,

"comments": 6,

"position": "北京市丰台区六里桥地铁"

},

{

"price": 398,

"title": "宽窄巷子+160平大空间+文化保护区双地铁",

"score": 5,

"comments": 6,

"position": "北京市丰台区六里桥地铁"

},

{

"price": 398,

"title": "宽窄巷子+160平大空间+文化保护区双地铁",

"score": 5,

"comments": 6,

"position": "北京市丰台区六里桥地铁"

},

{

"price": 398,

"title": "宽窄巷子+160平大空间+文化保护区双地铁",

"score": 5,

"comments": 6,

"position": "北京市丰台区六里桥地铁"

},

{

"price": 398,

"title": "宽窄巷子+160平大空间+文化保护区双地铁",

"score": 5,

"comments": 6,

"position": "北京市丰台区六里桥地铁"

}]

self.render("index.html", houses=houses)

### 2-3 函数

#### static\_url()

Tornado模板模块提供了一个叫作static\_url的函数来生成静态文件目录下文件的URL。如下面的示例代码：

<link rel="stylesheet" href="{{ static\_url("style.css") }}">

这个对static\_url的调用生成了URL的值，并渲染输出类似下面的代码：

<link rel="stylesheet" href="/static/style.css?v=ab12">

**优点：**

* static\_url函数创建了一个基于文件内容的hash值，并将其添加到URL末尾（查询字符串的参数v）。这个hash值确保浏览器总是加载一个文件的最新版而不是之前的缓存版本。无论是在你应用的开发阶段，还是在部署到生产环境使用时，都非常有用，因为你的用户不必再为了看到你的静态内容而清除浏览器缓存了。
* 另一个好处是你可以改变你应用URL的结构，而不需要改变模板中的代码。例如，可以通过设置**static\_url\_prefix**来更改Tornado的默认静态路径前缀/static。如果使用static\_url而不是硬编码的话，代码不需要改变。

#### 转义

我们新建一个表单页面new.html

<!DOCTYPE html>

<html>

<head>

<title>新建房源</title>

</head>

<body>

<form method="post">

<textarea name="text"></textarea>

<input type="submit" value="提交">

</form>

{{text}}

</body>

</html>

对应的handler为：

class NewHandler(RequestHandler):

def get(self):

self.render("new.html", text="")

def post(self):

text = self.get\_argument("text", "")

print text

self.render("new.html", text=text)

当我们在表单中填入如下内容时：

<script>alert("hello!");</script>

写入的js程序并没有运行，而是显示出来了：

我们查看页面源代码，发现<、>、"等被转换为对应的html字符。

&lt;script&gt;alert(&quot;hello!&quot;);&lt;/script&gt;

这是因为tornado中默认开启了模板自动转义功能，防止网站受到恶意攻击。

我们可以通过raw语句来输出不被转义的原始格式，如：

{% raw text %}

注意：在Firefox浏览器中会直接弹出alert窗口，而在Chrome浏览器中，需要set\_header("X-XSS-Protection", 0)

若要关闭自动转义，一种方法是在Application构造函数中传递**autoescape=None**，另一种方法是在每页模板中修改自动转义行为，添加如下语句：

{% autoescape None %}

**escape()**

关闭自动转义后，可以使用escape()函数来对特定变量进行转义，如：

{{ escape(text) }}

#### 自定义函数

在模板中还可以使用一个自己编写的函数，只需要将函数名作为模板的参数传递即可，就像其他变量一样。

我们修改后端如下：

def house\_title\_join(titles):

return "+".join(titles)

class IndexHandler(RequestHandler):

def get(self):

house\_list = [

{

"price": 398,

"titles": ["宽窄巷子", "160平大空间", "文化保护区双地铁"],

"score": 5,

"comments": 6,

"position": "北京市丰台区六里桥地铁"

},

{

"price": 398,

"titles": ["宽窄巷子", "160平大空间", "文化保护区双地铁"],

"score": 5,

"comments": 6,

"position": "北京市丰台区六里桥地铁"

}]

self.render("index.html", houses=house\_list, title\_join = house\_title\_join)

前段模板我们修改为：

<ul class="house-list">

{% if len(houses) > 0 %}

{% for house in houses %}

<li class="house-item">

<a href=""><img src="/static/images/home01.jpg"></a>

<div class="house-desc">

<div class="landlord-pic"><img src="/static/images/landlord01.jpg"></div>

<div class="house-price">￥<span>{{house["price"]}}</span>/晚</div>

<div class="house-intro">

<span class="house-title">{{title\_join(house["titles"])}}</span>

<em>整套出租 - {{house["score"]}}分/{{house["comments"]}}点评 - {{house["position"]}}</em>

</div>

</div>

</li>

{% end %}

{% else %}

对不起，暂时没有房源。

{% end %}

</ul>

### 2-4 块

我们可以使用块来复用模板，块语法如下：

{% block block\_name %} {% end %}

现在，我们对模板index.html进行抽象，抽离出父模板base.html如下：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, user-scalable=no">

{% block page\_title %}{% end %}

<link href="{{static\_url('plugins/bootstrap/css/bootstrap.min.css')}}" rel="stylesheet">

<link href="{{static\_url('plugins/font-awesome/css/font-awesome.min.css')}}" rel="stylesheet">

<link href="{{static\_url('css/reset.css')}}" rel="stylesheet">

<link href="{{static\_url('css/main.css')}}" rel="stylesheet">

{% block css\_files %}{% end %}

</head>

<body>

<div class="container">

<div class="top-bar">

{% block header %}{% end %}

</div>

{% block body %}{% end %}

<div class="footer">

{% block footer %}{% end %}

</div>

</div>

<script src="{{static\_url('js/jquery.min.js')}}"></script>

<script src="{{static\_url('plugins/bootstrap/js/bootstrap.min.js')}}"></script>

{% block js\_files %}{% end %}

</body>

</html>

而子模板index.html使用extends来使用父模板base.html，如下：

{% extends "base.html" %}

{% block page\_title %}

<title>爱家-房源</title>

{% end %}

{% block css\_files %}

<link href="{{static\_url('css/index.css')}}" rel="stylesheet">

{% end %}

{% block js\_files %}

<script src="{{static\_url('js/index.js')}}"></script>

{% end %}

{% block header %}

<div class="nav-bar">

<h3 class="page-title">房 源</h3>

</div>

{% end %}

{% block body %}

<ul class="house-list">

{% if len(houses) > 0 %}

{% for house in houses %}

<li class="house-item">

<a href=""><img src="/static/images/home01.jpg"></a>

<div class="house-desc">

<div class="landlord-pic"><img src="/static/images/landlord01.jpg"></div>

<div class="house-price">￥<span>{{house["price"]}}</span>/晚</div>

<div class="house-intro">

<span class="house-title">{{title\_join(house["titles"])}}</span>

<em>整套出租 - {{house["score"]}}分/{{house["comments"]}}点评 - {{house["position"]}}</em>

</div>

</div>

</li>

{% end %}

{% else %}

对不起，暂时没有房源。

{% end %}

</ul>

{% end %}

{% block footer %}

<p><span><i class="fa fa-copyright"></i></span>爱家租房&nbsp;&nbsp;享受家的温馨</p>

{% end %}

# 4.3 练习

1. 对比Django模板与Tornado模板的异同。
2. 练习使用Tornado模板的语法。

**5 数据库**

**知识点**

* torndb安装
* 连接初始化
* 执行语句
  + execute
  + execute\_rowcount
* 查询语句
  + get
  + query

# 5.1 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。

## torndb安装

pip install torndb

## 连接初始化

我们需要在应用启动时创建一个数据库连接实例，供各个RequestHandler使用。我们可以在构造Application的时候创建一个数据库实例并作为其属性，而RequestHandler可以通过self.application获取其属性，进而操作数据库实例。

import torndb

class Application(tornado.web.Application):

def \_\_init\_\_(self):

handlers = [

(r"/", IndexHandler),

]

settings = dict(

template\_path=os.path.join(os.path.dirname(\_\_file\_\_), "templates"),

static\_path=os.path.join(os.path.dirname(\_\_file\_\_), "statics"),

debug=True,

)

super(Application, self).\_\_init\_\_(handlers, \*\*settings)

# 创建一个全局mysql连接实例供handler使用

self.db = torndb.Connection(

host="127.0.0.1",

database="qianfeng",

user="root",

password="mysql"

)

## 使用数据库

新建数据库与表：

create database `qianfeng` default character set utf8;

use qianfeng;

create table houses (

id bigint(20) unsigned not null auto\_increment comment '房屋编号',

title varchar(64) not null default '' comment '标题',

position varchar(32) not null default '' comment '位置',

price int not null default 0,

score int not null default 5,

comments int not null default 0,

primary key(id)

)ENGINE=InnoDB default charset=utf8 comment='房屋信息表';

### 1. 执行语句（增删改）

* execute(query, parameters, \*kwparameters) 返回影响的最后一条自增字段值
* execute\_rowcount(query, parameters, \*kwparameters) 返回影响的行数

query为要执行的sql语句，parameters与kwparameters为要绑定的参数，如：

db.execute("insert into houses(title, position, price, score, comments) values(%s, %s, %s, %s, %s)", "独立装修小别墅", "紧邻文津街", 280, 5, 128)

或

db.execute("insert into houses(title, position, price, score, comments) values(%(title)s, %(position)s, %(price)s, %(score)s, %(comments)s)", title="独立装修小别墅", position="紧邻文津街", price=280, score=5, comments=128)

执行语句主要用来执行非查询语句。

class InsertHandler(RequestHandler):

def post(self):

title = self.get\_argument("title")

position = self.get\_argument("position")

price = self.get\_argument("price")

score = self.get\_argument("score")

comments = self.get\_argument("comments")

try:

ret = self.application.db.execute("insert into houses(title, position, price, score, comments) values(%s, %s, %s, %s, %s)", title, position, price, score, comments)

except Exception as e:

self.write("DB error:%s" % e)

else:

self.write("OK %d" % ret)

### 2. 查询语句

* get(query, parameters, \*kwparameters) 返回单行结果或None，若出现多行则报错。返回值为torndb.Row类型，是一个类字典的对象，即同时支持字典的关键字索引和对象的属相访问。
* query(query, parameters, \*kwparameters) 返回多行结果，torndb.Row的列表。

以上一章节模板中的案例来演示，先修改一下index.html模板，将

<span class="house-title">{{title\_join(house["titles"])}}</span>

改为

<span class="house-title">{{house["title"]}}</span>

添加两个新的handler：

class GetHandler(RequestHandler):

def get(self):

"""访问方式为http://127.0.0.1/get?id=111"""

hid = self.get\_argument("id")

try:

ret = self.application.db.get("select title,position,price,score,comments from houses where id=%s", hid)

except Exception as e:

self.write("DB error:%s" % e)

else:

print type(ret)

print ret

print ret.title

print ret['title']

self.render("index.html", houses=[ret])

class QueryHandler(RequestHandler):

def get(self):

"""访问方式为http://127.0.0.1/query"""

try:

ret = self.application.db.query("select title,position,price,score,comments from houses limit 10")

except Exception as e:

self.write("DB error:%s" % e)

else:

self.render("index.html", houses=ret)

# 5.2 练习

1. 复习MySQL的使用与sql语法。
2. 练习在Tornado中使用torndb操作数据库。

**6 安全应用**

**知识点**

* Cookie操作
* 安全Cookie
* 跨站请求伪造原理
* XSRF保护
  + 模板
  + 请求体
  + HTTP报文头
* 用户验证
  + authenticated装饰器
  + get\_current\_user()方法
  + login\_url设置

# 6.1 Cookie

对于RequestHandler，除了在第二章中讲到的之外，还提供了操作cookie的方法。

## 设置

**set\_cookie(name, value, domain=None, expires=None, path='/', expires\_days=None)**

参数说明：

| **参数名** | **说明** |
| --- | --- |
| name | cookie名 |
| value | cookie值 |
| domain | 提交cookie时匹配的域名 |
| path | 提交cookie时匹配的路径 |
| expires | cookie的有效期，可以是时间戳整数、时间元组或者datetime类型，为**UTC时间** |
| expires\_days | cookie的有效期，天数，优先级低于expires |

import datetime

class IndexHandler(RequestHandler):

def get(self):

self.set\_cookie("n1", "v1")

self.set\_cookie("n2", "v2", path="/new", expires=time.strptime("2016-11-11 23:59:59","%Y-%m-%d %H:%M:%S"))

self.set\_cookie("n3", "v3", expires\_days=20)

# 利用time.mktime将本地时间转换为UTC标准时间

self.set\_cookie("n4", "v4", expires=time.mktime(time.strptime("2016-11-11 23:59:59","%Y-%m-%d %H:%M:%S")))

self.write("OK")

#### 原理

设置cookie实际就是通过设置header的**Set-Cookie**来实现的。

class IndexHandler(RequestHandler):

def get(self):

self.set\_header("Set-Cookie", "n5=v5; expires=Fri, 11 Nov 2016 15:59:59 GMT; Path=/")

self.write("OK")

## 获取

**get\_cookie(name, default=None)**

获取名为name的cookie，可以设置默认值。

class IndexHandler(RequestHandler):

def get(self):

n3 = self.get\_cookie("n3")

self.write(n3)

## 清除

**clear\_cookie(name, path='/', domain=None)**

删除名为name，并同时匹配domain和path的cookie。

**clear\_all\_cookies(path='/', domain=None)**

删除同时匹配domain和path的所有cookie。

class ClearOneCookieHandler(RequestHandler):

def get(self):

self.clear\_cookie("n3")

self.write("OK")

class ClearAllCookieHandler(RequestHandler):

def get(self):

self.clear\_all\_cookies()

self.write("OK")

**注意：执行清除cookie操作后，并不是立即删除了浏览器中的cookie，而是给cookie值置空，并改变其有效期使其失效。真正的删除cookie是由浏览器去清理的。**

## 安全Cookie

Cookie是存储在客户端浏览器中的，很容易被篡改。Tornado提供了一种对Cookie进行简易加密签名的方法来防止Cookie被恶意篡改。

使用安全Cookie需要为应用配置一个用来给Cookie进行混淆的秘钥cookie\_secret，将其传递给Application的构造函数。我们可以使用如下方法来生成一个随机字符串作为cookie\_secret的值。

>>> import base64, uuid

>>> base64.b64encode(uuid.uuid4().bytes + uuid.uuid4().bytes)

'2hcicVu+TqShDpfsjMWQLZ0Mkq5NPEWSk9fi0zsSt3A='

Base64是一种基于64个可打印字符来表示二进制数据的表示方法。由于2的6次方等于64，所以每6个比特为一个单元，对应某个可打印字符。三个字节有24个比特，对应于4个Base64单元，即3个字节需要用4个可打印字符来表示。

uuid, 通用唯一识别码（英语：Universally Unique Identifier，简称UUID），是由一组32个16进制数字所构成（两个16进制数是一个字节，总共16字节），因此UUID理论上的总数为16^32=2^128，约等于3.4 x 10^38。也就是说若每纳秒产生1兆个UUID，要花100亿年才会将所有UUID用完。

uuid模块的uuid4()函数可以随机产生一个uuid码，bytes属性将此uuid码作为16字节字符串。

将生成的cookie\_secret传入Application构造函数：

app = tornado.web.Application(

[(r"/", IndexHandler),],

cookie\_secret = "2hcicVu+TqShDpfsjMWQLZ0Mkq5NPEWSk9fi0zsSt3A="

)

### 获取和设置

#### set\_secure\_cookie(name, value, expires\_days=30)

设置一个带签名和时间戳的cookie，防止cookie被伪造。

#### get\_secure\_cookie(name, value=None, max\_age\_days=31)

如果cookie存在且验证通过，返回cookie的值，否则返回None。max\_age\_day不同于expires\_days，expires\_days是设置浏览器中cookie的有效期，而max\_age\_day是过滤安全cookie的时间戳。

class IndexHandler(RequestHandler):

def get(self):

cookie = self.get\_secure\_cookie("count")

count = int(cookie) + 1 if cookie else 1

self.set\_secure\_cookie("count", str(count))

self.write(

'<html><head><title>Cookie计数器</title></head>'

'<body><h1>您已访问本页%d次。</h1>' % count +

'</body></html>'

)

我们看签名后的cookie值：

"2|1:0|10:1476412069|5:count|4:NQ==|cb5fc1d4434971de6abf87270ac33381c686e4ec8c6f7e62130a0f8cbe5b7609"

字段说明：

1. 安全cookie的版本，默认使用版本2，不带长度说明前缀
2. 默认为0
3. 时间戳
4. cookie名
5. base64编码的cookie值
6. 签名值，不带长度说明前缀

**注意：Tornado的安全cookie只是一定程度的安全，仅仅是增加了恶意修改的难度。Tornado的安全cookies仍然容易被窃听，而cookie值是签名不是加密，攻击者能够读取已存储的cookie值，并且可以传输他们的数据到任意服务器，或者通过发送没有修改的数据给应用伪造请求。因此，避免在浏览器cookie中存储敏感的用户数据是非常重要的。**

# 6.2 XSRF

## 跨站请求伪造

先建立一个网站127.0.0.1:8000，使用上一节中的Cookie计数器：

class IndexHandler(RequestHandler):

def get(self):

cookie = self.get\_secure\_cookie("count")

count = int(cookie) + 1 if cookie else 1

self.set\_secure\_cookie("count", str(count))

self.write(

'<html><head><title>Cookie计数器</title></head>'

'<body><h1>您已访问本页%d次。</h1>' % count +

'</body></html>'

)

再建立一个网站127.0.0.1:9000，

class IndexHandler(RequestHandler):

def get(self):

self.write('<html><head><title>被攻击的网站</title></head>'

'<body><h1>此网站的图片链接被修改了</h1>'

'<img alt="这应该是图片" src="http://127.0.0.1:8000/?f=9000/">'

'</body></html>'

)

在9000网站我们模拟攻击者修改了我们的图片源地址为8000网站的Cookie计数器页面网址。当我们访问9000网站的时候，在我们不知道、未授权的情况下8000网站的Cookie被使用了，以至于让8000网址认为是我们自己调用了8000网站的逻辑。这就是CSRF（Cross-site request forgery）跨站请求伪造（跨站攻击或跨域攻击的一种），通常缩写为CSRF或者XSRF。

我们刚刚使用的是GET方式模拟的攻击，为了防范这种方式的攻击，任何会产生副作用的HTTP请求，比如点击购买按钮、编辑账户设置、改变密码或删除文档，都应该使用HTTP POST方法（或PUT、DELETE）。但是，这并不足够：一个恶意站点可能会通过其他手段来模拟发送POST请求，保护POST请求需要额外的策略。

## XSRF保护

浏览器有一个很重要的概念——**同源策略**(Same-Origin Policy)。 所谓同源是指，域名，协议，端口相同。 不同源的客户端脚本(javascript、ActionScript)在没明确授权的情况下，不能读写对方的资源。

由于第三方站点没有访问cookie数据的权限（同源策略），所以我们可以要求每个请求包括一个特定的参数值作为令牌来匹配存储在cookie中的对应值，如果两者匹配，我们的应用认定请求有效。而第三方站点无法在请求中包含令牌cookie值，这就有效地防止了不可信网站发送未授权的请求。

### 开启XSRF保护

要开启XSRF保护，需要在Application的构造函数中添加xsrf\_cookies参数：

app = tornado.web.Application(

[(r"/", IndexHandler),],

cookie\_secret = "2hcicVu+TqShDpfsjMWQLZ0Mkq5NPEWSk9fi0zsSt3A=",

xsrf\_cookies = True

)

当这个参数被设置时，Tornado将拒绝请求参数中不包含正确的\_xsrf值的POST、PUT和DELETE请求。

class IndexHandler(RequestHandler):

def post(self):

self.write("hello qianfeng")

用不带\_xsrf的post请求时，报出了HTTP 403: Forbidden ('\_xsrf' argument missing from POST)的错误。

### 模板应用

在模板中使用XSRF保护，只需在模板中添加

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

如新建一个模板index.html

<!DOCTYPE html>

<html>

<head>

<title>测试XSRF</title>

</head>

<body>

<form method="post">

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

<input type="text" name="message"/>

<input type="submit" value="Post"/>

</form>

</body>

</html>

后端

class IndexHandler(RequestHandler):

def get(self):

self.render("index.html")

def post(self):

self.write("hello qianfeng")

模板中添加的语句帮我们做了两件事：

* 为浏览器设置了\_xsrf的Cookie（注意此Cookie浏览器关闭时就会失效）
* 为模板的表单中添加了一个隐藏的输入名为\_xsrf，其值为\_xsrf的Cookie值

渲染后的页面原码如下：

<!DOCTYPE html>

<html>

<head>

<title>测试XSRF</title>

</head>

<body>

<form method="post">

<input type="hidden" name="\_xsrf" value="2|543c2206|a056ff9e49df23eaffde0a694cde2b02|1476443353"/>

<input type="text" name="message"/>

<input type="submit" value="Post"/>

</form>

</body>

</html>

# 6.3 用户验证

用户验证是指在收到用户请求后进行处理前先判断用户的认证状态（如登陆状态），若通过验证则正常处理，否则强制用户跳转至认证页面（如登陆页面）。

## authenticated装饰器

为了使用Tornado的认证功能，我们需要对登录用户标记具体的处理函数。我们可以使用@tornado.web.authenticated装饰器完成它。当我们使用这个装饰器包裹一个处理方法时，Tornado将确保这个方法的主体只有在合法的用户被发现时才会调用。

class ProfileHandler(RequestHandler):

@tornado.web.authenticated

def get(self):

self.write("这是我的个人主页。")

## get\_current\_user()方法

装饰器@tornado.web.authenticated的判断执行依赖于请求处理类中的self.current\_user属性，如果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应用设置中login\_url指定的URL，而非法用户的POST请求将返回一个带有403（Forbidden）状态的HTTP响应。

在获取self.current\_user属性的时候，tornado会调用get\_current\_user()方法来返回current\_user的值。也就是说，**我们验证用户的逻辑应写在get\_current\_user()方法中，若该方法返回非假值则验证通过，否则验证失败。**

class ProfileHandler(RequestHandler):

def get\_current\_user(self):

"""在此完成用户的认证逻辑"""

user\_name = self.get\_argument("name", None)

return user\_name

@tornado.web.authenticated

def get(self):

self.write("这是我的个人主页。")

## login\_url设置

当用户验证失败时，将用户重定向到login\_url上，所以我们还需要在Application中配置login\_url。

class LoginHandler(RequestHandler):

def get(self):

"""在此返回登陆页面"""

self.write("登陆页面")

app = tornado.web.Application(

[

(r"/", IndexHandler),

(r"/profile", ProfileHandler),

(r"/login", LoginHandler),

],

"login\_url":"/login"

)

#### 想一想，完成登陆操作后应该进入哪个页面？

在login\_url后面补充的next参数就是记录的跳转至登录页面前的所在位置，所以我们可以使用next参数来完成登陆后的跳转。

修改登陆逻辑：

class LoginHandler(RequestHandler):

def get(self):

"""登陆处理，完成登陆后跳转回前一页面"""

next = self.get\_argument("next", "/")

self.redirect(next+"?name=logined")

用户访问的网址顺序：

# 6.4 练习

请用本章所学知识，在开启XSRF保护的前提下利用Cookie分别用模板和Javascript AJAX两种方式实现用户登陆验证系统。

**7 异步与WebSockets**

**知识点**

* 理解同步与异步执行过程
* 理解异步代码的回调写法与yield写法
* Tornado异步
  + 异步Web客户端AsyncHTTPClient
  + tornado.web.asynchronous
  + tornado.gen.coroutine
  + 并行协程用法
* WebSocket的使用

# 7.1 认识异步

## 1. 同步

我们用两个函数来模拟两个客户端请求，并依次进行处理：

# coding:utf-8

def req\_a():

"""模拟请求a"""

print '开始处理请求req\_a'

print '完成处理请求req\_a'

def req\_b():

"""模拟请求b"""

print '开始处理请求req\_b'

print '完成处理请求req\_b'

def main():

"""模拟tornado框架，处理两个请求"""

req\_a()

req\_b()

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

main()

执行结果：

开始处理请求req\_a

完成处理请求req\_a

开始处理请求req\_b

完成处理请求req\_b

**同步是按部就班的依次执行，始终按照同一个步调执行，上一个步骤未执行完不会执行下一步。**

**想一想**，如果在处理请求req\_a时需要执行一个耗时的工作（如IO），其执行过程如何？

# coding:utf-8

import time

def long\_io():

"""模拟耗时IO操作"""

print "开始执行IO操作"

time.sleep(5)

print "完成IO操作"

return "io result"

def req\_a():

print "开始处理请求req\_a"

ret = long\_io()

print "ret: %s" % ret

print "完成处理请求req\_a"

def req\_b():

print "开始处理请求req\_b"

print "完成处理请求req\_b"

def main():

req\_a()

req\_b()

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

main()

执行过程：

开始处理请求req\_a

开始执行IO操作

完成IO操作

完成处理请求req\_a

开始处理请求req\_b

完成处理请求req\_b

在上面的测试中，我们看到耗时的操作会将代码执行阻塞住，即req\_a未处理完req\_b是无法执行的。

#### 我们怎么解决耗时操作阻塞代码执行？

## 2. 异步

对于耗时的过程，我们将其交给别人（如其另外一个线程）去执行，而我们继续往下处理，当别人执行完耗时操作后再将结果反馈给我们，这就是我们所说的异步。

我们用容易理解的线程机制来实现异步。

### 2.1 回调写法实现原理

# coding:utf-8

import time

import thread

def long\_io(callback):

"""将耗时的操作交给另一线程来处理"""

def fun(cb): # 回调函数作为参数

"""耗时操作"""

print "开始执行IO操作"

time.sleep(5)

print "完成IO操作，并执行回调函数"

cb("io result") # 执行回调函数

thread.start\_new\_thread(fun, (callback,)) # 开启线程执行耗时操作

def on\_finish(ret):

"""回调函数"""

print "开始执行回调函数on\_finish"

print "ret: %s" % ret

print "完成执行回调函数on\_finish"

def req\_a():

print "开始处理请求req\_a"

long\_io(on\_finish)

print "离开处理请求req\_a"

def req\_b():

print "开始处理请求req\_b"

time.sleep(2) # 添加此句来突出显示程序执行的过程

print "完成处理请求req\_b"

def main():

req\_a()

req\_b()

while 1: # 添加此句防止程序退出，保证线程可以执行完

pass

if \_\_name\_\_ == '\_\_main\_\_':

main()

执行过程:

开始处理请求req\_a

离开处理请求req\_a

开始处理请求req\_b

开始执行IO操作

完成处理请求req\_b

完成IO操作，并执行回调函数

开始执行回调函数on\_finish

ret: io result

完成执行回调函数on\_finish

**异步的特点是程序存在多个步调，即本属于同一个过程的代码可能在不同的步调上同时执行。**

### 2.2 协程写法实现原理

在使用回调函数写异步程序时，需将本属于一个执行逻辑（处理请求a）的代码拆分成两个函数req\_a和on\_finish，这与同步程序的写法相差很大。而同步程序更便于理解业务逻辑，所以我们能否用同步代码的写法来编写异步程序？

#### 回想yield关键字的作用？

#### 初始版本

# coding:utf-8

import time

import thread

gen = None # 全局生成器，供long\_io使用

def long\_io():

def fun():

print "开始执行IO操作"

global gen

time.sleep(5)

try:

print "完成IO操作，并send结果唤醒挂起程序继续执行"

gen.send("io result") # 使用send返回结果并唤醒程序继续执行

except StopIteration: # 捕获生成器完成迭代，防止程序退出

pass

thread.start\_new\_thread(fun, ())

def req\_a():

print "开始处理请求req\_a"

ret = yield long\_io()

print "ret: %s" % ret

print "完成处理请求req\_a"

def req\_b():

print "开始处理请求req\_b"

time.sleep(2)

print "完成处理请求req\_b"

def main():

global gen

gen = req\_a()

gen.next() # 开启生成器req\_a的执行

req\_b()

while 1:

pass

if \_\_name\_\_ == '\_\_main\_\_':

main()

执行过程：

开始处理请求req\_a

开始处理请求req\_b

开始执行IO操作

完成处理请求req\_b

完成IO操作，并send结果唤醒挂起程序继续执行

ret: io result

完成处理请求req\_a

#### 升级版本

我们在上面编写出的版本虽然req\_a的编写方式很类似与同步代码，但是在main中调用req\_a的时候却不能将其简单的视为普通函数，而是需要作为生成器对待。

现在，我们试图尝试修改，让req\_a与main的编写都类似与同步代码。

# coding:utf-8

import time

import thread

gen = None # 全局生成器，供long\_io使用

def gen\_coroutine(f):

def wrapper(\*args, \*\*kwargs):

global gen

gen = f()

gen.next()

return wrapper

def long\_io():

def fun():

print "开始执行IO操作"

global gen

time.sleep(5)

try:

print "完成IO操作，并send结果唤醒挂起程序继续执行"

gen.send("io result") # 使用send返回结果并唤醒程序继续执行

except StopIteration: # 捕获生成器完成迭代，防止程序退出

pass

thread.start\_new\_thread(fun, ())

@gen\_coroutine

def req\_a():

print "开始处理请求req\_a"

ret = yield long\_io()

print "ret: %s" % ret

print "完成处理请求req\_a"

def req\_b():

print "开始处理请求req\_b"

time.sleep(2)

print "完成处理请求req\_b"

def main():

req\_a()

req\_b()

while 1:

pass

if \_\_name\_\_ == '\_\_main\_\_':

main()

执行过程：

开始处理请求req\_a

开始处理请求req\_b

开始执行IO操作

完成处理请求req\_b

完成IO操作，并send结果唤醒挂起程序继续执行

ret: io result

完成处理请求req\_a

### 思考

1. Tornado里的异步就是协程，这句话对吗？
2. Tornado中出现yield就是异步，这句话对吗？
3. 怎么理解yield将程序挂起？在Tornado中又如何理解yield挂起程序实现异步？

# 7.2 Tornado异步

因为epoll主要是用来解决网络IO的并发问题，所以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。

## 1. tornado.httpclient.AsyncHTTPClient

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）

## 2. 测试接口

新浪IP地址库

接口说明

1.请求接口（GET）：

[http://int.dpool.sina.com.cn/iplookup/iplookup.php?format=json&ip=[ip地址字串](http://int.dpool.sina.com.cn/iplookup/iplookup.php?format=json&ip=%5bip地址字串)]

2.响应信息：

（json格式的）国家 、省（自治区或直辖市）、市（县）、运营商

3.返回数据格式：

{"ret":1,"start":-1,"end":-1,"country":"\u4e2d\u56fd","province":"\u5317\u4eac","city":"\u5317\u4eac","district":"","isp":"","type":"","desc":""}

## 3. 回调异步

class IndexHandler(tornado.web.RequestHandler):

@tornado.web.asynchronous # 不关闭连接，也不发送响应

def get(self):

http = tornado.httpclient.AsyncHTTPClient()

http.fetch("http://int.dpool.sina.com.cn/iplookup/iplookup.php?format=json&ip=14.130.112.24",

callback=self.on\_response)

def on\_response(self, response):

if response.error:

self.send\_error(500)

else:

data = json.loads(response.body)

if 1 == data["ret"]:

self.write(u"国家：%s 省份: %s 城市: %s" % (data["country"], data["province"], data["city"]))

else:

self.write("查询IP信息错误")

self.finish() # 发送响应信息，结束请求处理

### tornado.web.asynchronous

此装饰器用于回调形式的异步方法，并且应该仅用于HTTP的方法上（如get、post等）。

此装饰器不会让被装饰的方法变为异步，而只是告诉框架被装饰的方法是异步的，当方法返回时响应尚未完成。只有在request handler调用了finish方法后，才会结束本次请求处理，发送响应。

不带此装饰器的请求在get、post等方法返回时自动完成结束请求处理。

## 4. 协程异步

在上一节中我们自己封装的装饰器get\_coroutine在Tornado中对应的是tornado.gen.coroutine。

class IndexHandler(tornado.web.RequestHandler):

@tornado.gen.coroutine

def get(self):

http = tornado.httpclient.AsyncHTTPClient()

response = yield http.fetch("http://int.dpool.sina.com.cn/iplookup/iplookup.php?format=json&ip=14.130.112.24")

if response.error:

self.send\_error(500)

else:

data = json.loads(response.body)

if 1 == data["ret"]:

self.write(u"国家：%s 省份: %s 城市: %s" % (data["country"], data["province"], data["city"]))

else:

self.write("查询IP信息错误")

也可以将异步Web请求单独出来：

class IndexHandler(tornado.web.RequestHandler):

@tornado.gen.coroutine

def get(self):

rep = yield self.get\_ip\_info("14.130.112.24")

if 1 == rep["ret"]:

self.write(u"国家：%s 省份: %s 城市: %s" % (rep["country"], rep["province"], rep["city"]))

else:

self.write("查询IP信息错误")

@tornado.gen.coroutine

def get\_ip\_info(self, ip):

http = tornado.httpclient.AsyncHTTPClient()

response = yield http.fetch("http://int.dpool.sina.com.cn/iplookup/iplookup.php?format=json&ip=" + ip)

if response.error:

rep = {"ret:0"}

else:

rep = json.loads(response.body)

raise tornado.gen.Return(rep) # 此处需要注意

**代码中我们需要注意的地方是get\_ip\_info返回值的方式，在python 2中，使用了yield的生成器可以使用不返回任何值的return，但不能return value，因此Tornado为我们封装了用于在生成器中返回值的特殊异常tornado.gen.Return，并用raise来返回此返回值。**

### 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：

class IndexHandler(tornado.web.RequestHandler):

@tornado.gen.coroutine

def get(self):

ips = ["14.130.112.24",

"15.130.112.24",

"16.130.112.24",

"17.130.112.24"]

rep1, rep2 = yield [self.get\_ip\_info(ips[0]), self.get\_ip\_info(ips[1])]

rep34\_dict = yield dict(rep3=self.get\_ip\_info(ips[2]), rep4=self.get\_ip\_info(ips[3]))

self.write\_response(ips[0], rep1)

self.write\_response(ips[1], rep2)

self.write\_response(ips[2], rep34\_dict['rep3'])

self.write\_response(ips[3], rep34\_dict['rep4'])

def write\_response(self, ip, response):

self.write(ip)

self.write(":<br/>")

if 1 == response["ret"]:

self.write(u"国家：%s 省份: %s 城市: %s<br/>" % (response["country"], response["province"], response["city"]))

else:

self.write("查询IP信息错误<br/>")

@tornado.gen.coroutine

def get\_ip\_info(self, ip):

http = tornado.httpclient.AsyncHTTPClient()

response = yield http.fetch("http://int.dpool.sina.com.cn/iplookup/iplookup.php?format=json&ip=" + ip)

if response.error:

rep = {"ret:1"}

else:

rep = json.loads(response.body)

raise tornado.gen.Return(rep)

# 7.3 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。

目前常见的浏览器如 Chrome、IE、Firefox、Safari、Opera 等都支持 WebSocket，同时需要服务端程序支持 WebSocket。

## 1. Tornado的WebSocket模块

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。

#### WebSocketHandler.open()

当一个WebSocket连接建立后被调用。

#### WebSocketHandler.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**。

#### WebSocketHandler.on\_close()

当WebSocket连接关闭后被调用。

#### WebSocketHandler.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。

#### WebSocketHandler.close()

关闭WebSocket连接。

#### WebSocketHandler.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）。

## 2. 前端JavaScript编写

在前端JS中使用WebSocket与服务器通讯的常用方法如下：

var ws = new WebSocket("ws://127.0.0.1:8888/websocket"); // 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

alert(evt.data); // 接收的消息内容在事件参数evt的data属性中

};

## 3. 在线聊天室的小Demo

#### 后端代码 server.py

# coding:utf-8

import tornado.web

import tornado.ioloop

import tornado.httpserver

import tornado.options

import os

import datetime

from tornado.web import RequestHandler

from tornado.options import define, options

from tornado.websocket import WebSocketHandler

define("port", default=8000, type=int)

class IndexHandler(RequestHandler):

def get(self):

self.render("index.html")

class ChatHandler(WebSocketHandler):

users = set() # 用来存放在线用户的容器

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(u"[%s]-[%s]-进入聊天室" % (self.request.remote\_ip, datetime.datetime.now().strftime("%Y-%m-%d %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(u"[%s]-[%s]-说：%s" % (self.request.remote\_ip, datetime.datetime.now().strftime("%Y-%m-%d %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(u"[%s]-[%s]-离开聊天室" % (self.request.remote\_ip, datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")))

def check\_origin(self, origin):

return True # 允许WebSocket的跨域请求

if \_\_name\_\_ == '\_\_main\_\_':

tornado.options.parse\_command\_line()

app = tornado.web.Application([

(r"/", IndexHandler),

(r"/chat", ChatHandler),

],

static\_path = os.path.join(os.path.dirname(\_\_file\_\_), "static"),

template\_path = os.path.join(os.path.dirname(\_\_file\_\_), "template"),

debug = True

)

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

http\_server.listen(options.port)

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

#### 前端代码index.html

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>聊天室</title>

</head>

<body>

<div id="contents" style="height:500px;overflow:auto;"></div>

<div>

<textarea id="msg"></textarea>

<a href="javascript:;" onclick="sendMsg()">发送</a>

</div>

<script src="{{static\_url('js/jquery.min.js')}}"></script>

<script type="text/javascript">

var ws = new WebSocket("ws://192.168.114.177:8000/chat");

ws.onmessage = function(e) {

$("#contents").append("<p>" + e.data + "</p>");

}

function sendMsg() {

var msg = $("#msg").val();

ws.send(msg);

$("#msg").val("");

}

</script>

</body>

</html>

# 7.4 练习

1. 请解释清同步、异步、yield、协程几个概念和Tornado实现异步的原理。
2. 练习使用Tornado异步Web客户端。
3. 练习使用WebSocket。
4. 修改WebSocket案例中的在线聊天代码，将获取到的用户IP利用异步客户端查询归属地，并将消息显示格式为

[城市网友]-[IP]-[时间]：消息

如

[北京网友]-[14.130.112.24]-[2016-08-29 00:00:00]: 你好，Python

**注意，局域网的内网IP没有归属地，只需写出代码即可，不用测试。**