

Tornado

Python的Web框架种类繁多（比Python语言的关键字还要多），但在众多优秀的Web框架中，Tornado框架最适合用来开发需要处理长连接和应对高并发的Web应用。Tornado框架在设计之初就考虑到性能问题，通过对非阻塞I/O和epoll（Linux 2.5.44内核引入的一种多路I/O复用方式，旨在实现高性能网络服务，在BSD和macOS中是kqueue）的运用，Tornado可以处理大量的并发连接，更轻松的应对C10K（万级并发）问题，是非常理想的实时通信Web框架。

扩展：基于线程的Web服务器产品（如：Apache）会维护一个线程池来处理用户请求，当用户请求到达时就为该请求分配一个线程，如果线程池中沒有空闲线程了，那么可以通过创建新的线程来应付新的请求，但前提是系统尚有空闲的内存空间，显然这种方式很容易将服务器的空闲内存耗尽（大多数Linux发行版本中，默认线程栈大小为8M）。想象一下，如果我们要开发一个社交类应用，这类应用中，通常需要显示实时更新的消息、对象状态的变化和各种类型的通知，那也就意味着客户端需要保持请求连接来接收服务器的各种响应，在这种情况下，服务器上的工作线程很容易被耗尽，这也就意味着新的请求很有可能无法得到响应。

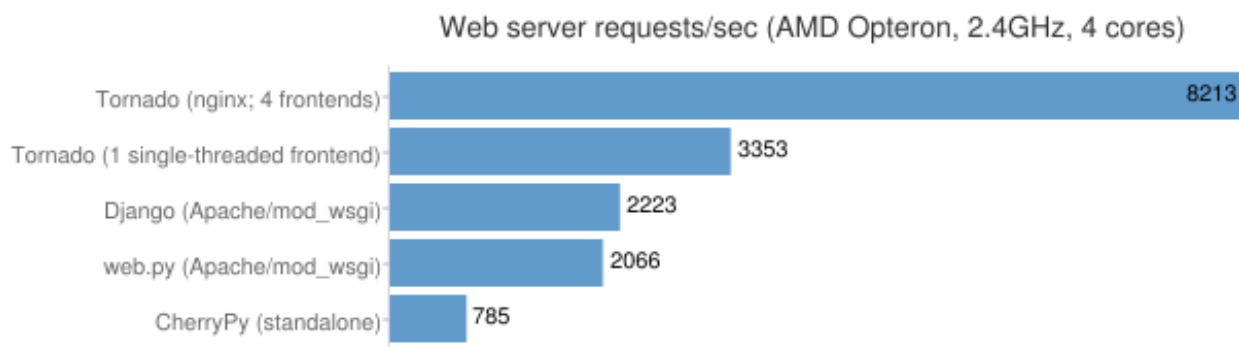
Tornado框架源于FriendFeed网站，在FriendFeed网站被Facebook收购之后得以开源，正式发布的日期是2009年9月10日。Tornado能让你能够快速开发高速的Web应用，如果你想编写一个可扩展的社交应用、实时分析引擎，或RESTful API，那么Tornado框架就是很好的选择。Tornado其实不仅仅是一个Web开发的框架，它还是一个高性能的事件驱动网络访问引擎，内置了高性能的HTTP服务器和客户端（支持同步和异步请求），同时还对WebSocket提供了完美的支持。

官方文档tornadoweb.org上面有很多不错的例子，你也可以在Github上找到Tornado的源代码和历史版本。

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tornado 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tornado的网络框架和HTTP服务器

一、Hello Tornado

1.在虚拟开发环境中执行：

```
pip install tornado
```

2.编写第一个torndao应用

```
# hello.py
import tornado.web
import tornado.ioloop

#定义处理类型
class IndexHandler(tornado.web.RequestHandler):
    #添加一个处理get请求方式的方法
    def get(self):
        #向响应中，添加数据
        self.write('好看的皮囊千篇一律，有趣的灵魂万里挑一。')

if __name__ == '__main__':
    #创建一个应用对象
    app = tornado.web.Application([(r '/', IndexHandler)])
    #绑定一个监听端口
    app.listen(8888)
```

```
#启动web程序，开始监听端口的连接
tornado.ioloop.IOLoop.current().start()
```

3.运行并访问应用

在终端执行

```
python hello.py
```

打开浏览器键入

```
http://localhost:8888
```

4.代码说明

- 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。

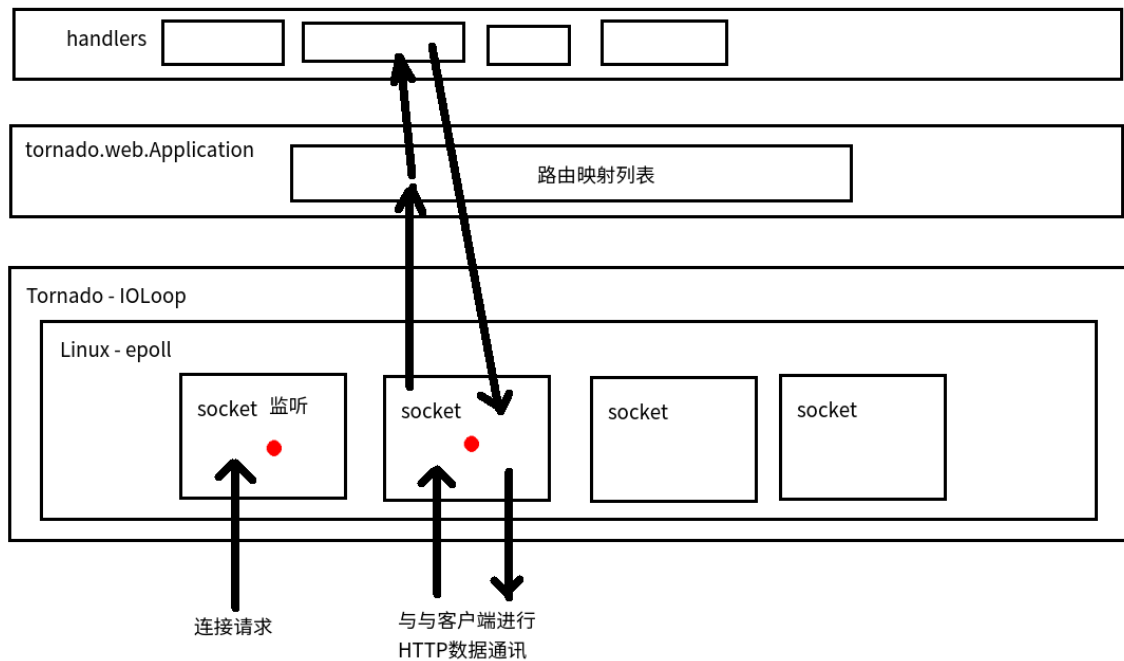
- Application

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**）。

- 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工作原理——以epoll为例



- **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。

5.httpserver

tornado.httpserver模块，顾名思义，它就是tornado的HTTP服务器实现。

我们可以创建一个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。

```
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 Tornado!")
```

```

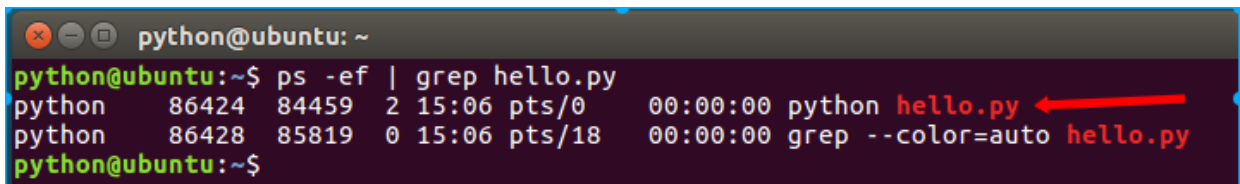
if __name__ == "__main__":
    app = tornado.web.Application([
        (r"/", IndexHandler),
    ])
    # 创建web服务器，并绑定我们自己的web应用
    http_server = tornado.httpserver.HTTPServer(app)
    http_server.listen(8000)
    tornado.ioloop.IOLoop.current().start()

```

6 多进程

我们刚刚实现的都是单进程，可以通过命令来查看：

```
$ ps -ef | grep hello.py
```



```

python@ubuntu: ~
python@ubuntu:~$ ps -ef | grep hello.py
python      86424      84459  2  15:06 pts/0    00:00:00 python hello.py
python      86428      85819  0  15:06 pts/18    00:00:00 grep --color=auto hello.py
python@ubuntu:~$

```

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 Tornado!")

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)

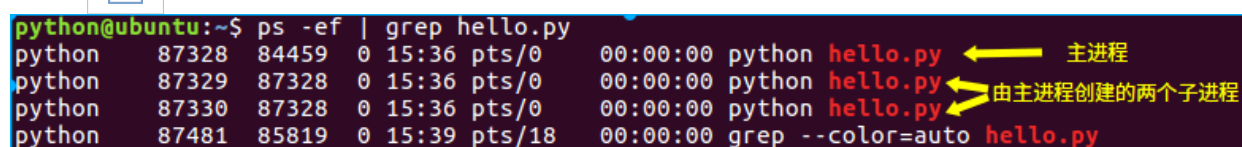
```

```
# -----  
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：



A terminal window showing the output of the command 'ps -ef | grep hello.py'. The output lists four processes: three 'python' processes and one 'grep' process. Annotations with yellow arrows point to the first three processes, labeling them as '主进程' (main process) and '由主进程创建的两个子进程' (two sub-processes created by the main process).

Process	PPID	UID	Time	State	TTY	Time	Command
python	87328	84459	0	15:36	pts/0	00:00:00	python hello.py
python	87329	87328	0	15:36	pts/0	00:00:00	python hello.py
python	87330	87328	0	15:36	pts/0	00:00:00	python hello.py
python	87481	85819	0	15:39	pts/18	00:00:00	grep --color=auto hello.py

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；

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，如果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。

```
apple:~ apple$ ab -n 100 -c 10 http://127.0.0.1:8000/
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeus.com
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient).....done

Server Software:      TornadoServer/6.0.4
Server Hostname:      127.0.0.1
Server Port:          8000

Document Path:        /
Document Length:      14 bytes

Concurrency Level:    10
Time taken for tests:  0.057 seconds
Complete requests:    100
Failed requests:       0
Total transferred:    20900 bytes
HTML transferred:     1400 bytes
Requests per second:  1757.75 [#/sec] (mean)
Time per request:     5.689 [ms] (mean)
Time per request:     0.569 [ms] (mean, across all concurrent requests)
Transfer rate:        358.76 [Kbytes/sec] received

Connection Times (ms)

```

```
apple:~ apple$ ab -n 100 -c 10 http://127.0.0.1:9000/
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeus.com
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient).....done

Server Software:      TornadoServer/6.0.4
Server Hostname:      127.0.0.1
Server Port:          9000

Document Path:        /
Document Length:      10 bytes

Concurrency Level:    10
Time taken for tests:  0.076 seconds
Complete requests:    100
Failed requests:       0
Total transferred:    20500 bytes
HTML transferred:     1000 bytes
Requests per second:  1307.50 [#/sec] (mean)
Time per request:     7.648 [ms] (mean)
Time per request:     0.765 [ms] (mean, across all concurrent requests)
Transfer rate:        261.75 [Kbytes/sec] received
```

二、应用程序配置

1. 命令行参数

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：

参数	说明
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
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.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

```
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("itcast", default=[], type=str, multiple=True,
help="itcast 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 Itcast!")
if __name__ == "__main__":
    tornado.options.parse_command_line()
    app = tornado.web.Application([
        (r"/", IndexHandler),
    ])
    print(tornado.options.options.itcast)
```



```
http_server = tornado.httpserver.HTTPServer(app)
http_server.listen(tornado.options.options.port)
tornado.ioloop.IOLoop.current().start()
```

在终端执行：

```
python opt.py --port=9000 --itcast=python,c++,java,php,ios
```

2.配置文件

应用程序对象，也可以进行配置，比如模板文件路径，静态资源路径，是否是调试模式等：

```
if __name__ == '__main__':
    #创建一个应用对象
    settings = dict(
        template_path = 'templates',
        static_path='static',
        debug = True    #调试模式
    )
    app = tornado.web.Application([(r'/',IndexHandler)], **settings)
    #绑定一个监听端口
    app.listen(8888)
    #启动web程序，开始监听端口的连接
    tornado.ioloop.IOLoop.current().start()
```

也可以单独写一个配置文件：

```
#settings.py
settings = {
    'template_path': os.path.join(os.path.dirname(__file__),
    'templates'),
    'static_path': os.path.join(os.path.dirname(__file__),
    'statics'),
    'debug':True,
}
```

使用：

```
import tornado.web
import config
.....
if __name__ == "__main__":
    app = tornado.web.Application([], **config.settings)
    .....
```

三、路由

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：

```
from tornado.web import url
[
    (r"/", Indexhandler),
    (r"/cpp", ItcastHandler, {"subject": "c++"}),
    url(r"/python", ItcastHandler, {"subject": "python"},
    name="python_url")
]
```

url方法：

指定URL和处理程序之间的映射。

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 ItcastHandler(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)来获取该名字对应的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">itcast</a>' %
                    python_url)

class ItcastHandler(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", ItcastHandler, {"subject": "c++"}),
        url(r"/python", ItcastHandler, {"subject": "python"},
name="python_url")
    ],
    debug = True)
    http_server = tornado.httpserver.HTTPServer(app)
    http_server.listen(options.port)
    tornado.ioloop.IOLoop.current().start()
```

重写 RequestHandler 的方法函数

对于一个请求的处理过程代码调用次序如下：

1. 程序为每一个请求创建一个 RequestHandler 对象；
2. 程序调用 `initialize()` 函数，这个函数的参数是 `Application` 配置中的关键字参数定义。（`initialize` 方法是 Tornado 1.1 中新添加的，旧版本中你需要重写 `__init__` 以达到同样的目的）`initialize` 方法一般只是把传入的参数存到成员变量中，而不会产生一些输出或者调用像 `send_error` 之类的方法。
3. 程序调用 `prepare()`。无论使用了哪种 HTTP 方法，`prepare` 都会被调用到，因此这个方法通常会被定义在一个基类中，然后在子类中重用。`prepare` 可以产生输出信息。如果它调用了 `finish`（或 `send_error`` 等函数），那么整个处理流程就此结束。
4. 程序调用某个 HTTP 方法：例如 `get()`、`post()`、`put()` 等。如果 URL 的正则表达式模式中有分组匹配，那么相关匹配会作为参数传入方法。

重写 `initialize()` 函数（会在创建 RequestHandler 对象后调用）：

```
class ProfileHandler(tornado.web.RequestHandler):

    def initialize(self, database):
        self.database = database

    def get(self):
        self.write("result:" + self.database)

application = tornado.web.Application([
    (r"/init", ProfileHandler, dict(database="database"))
])
```

视图函数传递参数

```
class TestHandler(RequestHandler):
    def get(self, arg1, arg2):
        self.write('arg1:{} arg2:{}'.format(arg1, arg2))

if __name__ == "__main__":
    print(tornado.options.options.itcast) # 输出多值选项
```

```
print(tornado.options.options.port) # 输出多值选项
app = tornado.web.Application([
    # 一个参数
    (r"/test/(\d+)/", TestHandler),
    # 二个参数
    (r"/test/(\d+)/(\d+)/", TestHandler),
    (r"/test/(\d+)_(\d+)/", TestHandler),
], debug=True)
```

四、输入

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、xml;
- 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=itcast。

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)`

说明：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。

参数：name参数名称

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)`

功能：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[]。

2. 获取请求体参数

- `get_body_argument(name, default=_ARG_DEFAULT, strip=True)`

功能：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。

- `get_body_arguments(name, strip=True)`

功能：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`[]`。

- 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`。对于请求体数据为`json`或`xml`的，无法通过这两个方法获取。

3. 前两类方法的整合

- `get_argument(name, default=_ARG_DEFAULT, strip=True)`

功能：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。

- `get_arguments(name, strip=True)`

功能：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`[]`。

```
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", "itcast")
default_args = self.get_arguments("b")

try:
    missing_arg = self.get_argument("c")
except MissingArgumentError as e:
    missing_arg = "We caught 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()

```

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：

参数	说明
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
headers	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： <pre>{ "form_filename1": [<tornado.httputil.HTTPFile>, <tornado.httputil.HTTPFile>], "form_filename2": [<tornado.httputil.HTTPFile>], ... }</pre> 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"]。
version	使用的HTTP版本

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

```
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 itcast.")

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("./itcast", '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。

```

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 itcast.")

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()

```

五、输出

1. write(chunk)

将chunk数据写到输出缓冲区。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。

- 如果chunk是字典，write方法可以将其序列化为字符串

```

class IndexHandler(RequestHandler):
    def get(self):
        self.write("hello itcast 1!")
        self.write("hello itcast 2!")
        self.write("hello itcast 3!")

class IndexHandler(RequestHandler):
    def get(self):

```

```
stu = {
    "name": "zhangsan",
    "age": 24,
    "gender": 1,
}
self.write(stu)
```

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。

3. **set_default_headers()**

该方法会在进入HTTP处理方法前被调用，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itcast、值为python的header
        self.set_header("itcast", "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("itcast", "i love python") # 注意此处重写了header中的itcast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)

```

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

5. **redirect(url)**

重定向

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。

- 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.write_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"])
```

六、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。将路由映射中的第三个字典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：

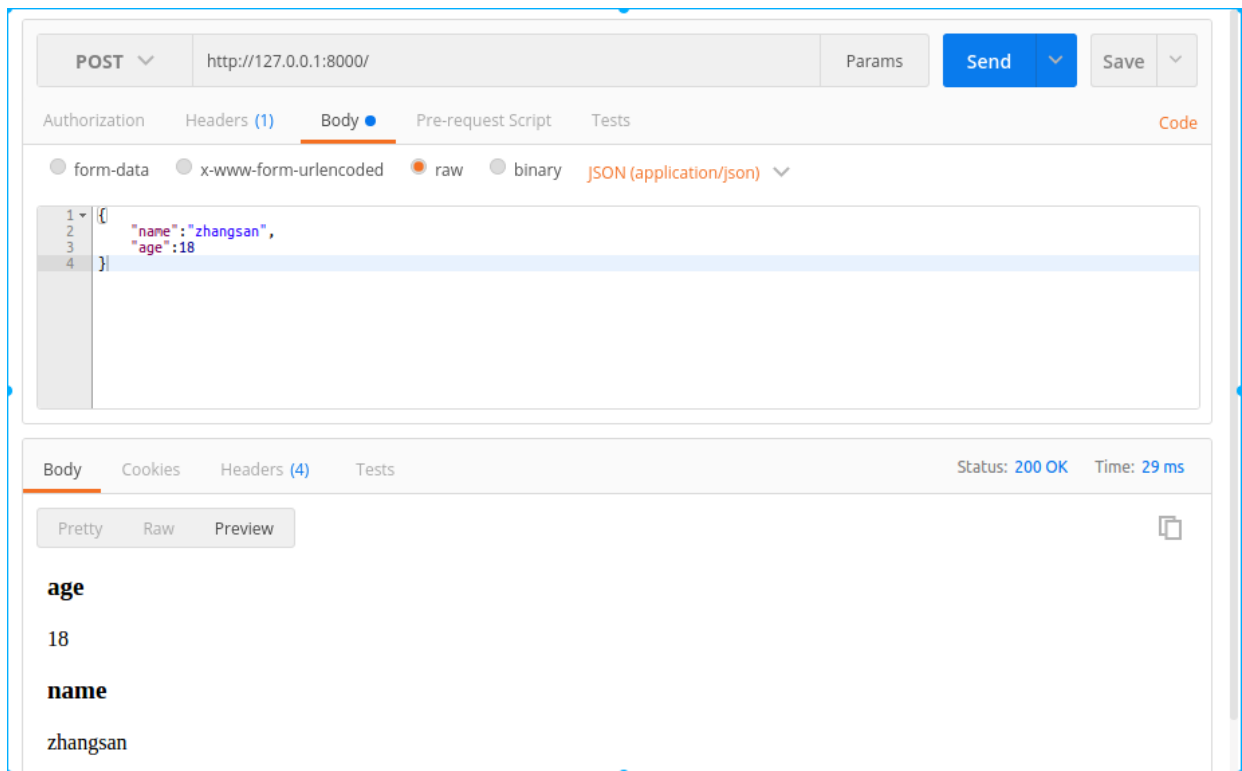
```
import json

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

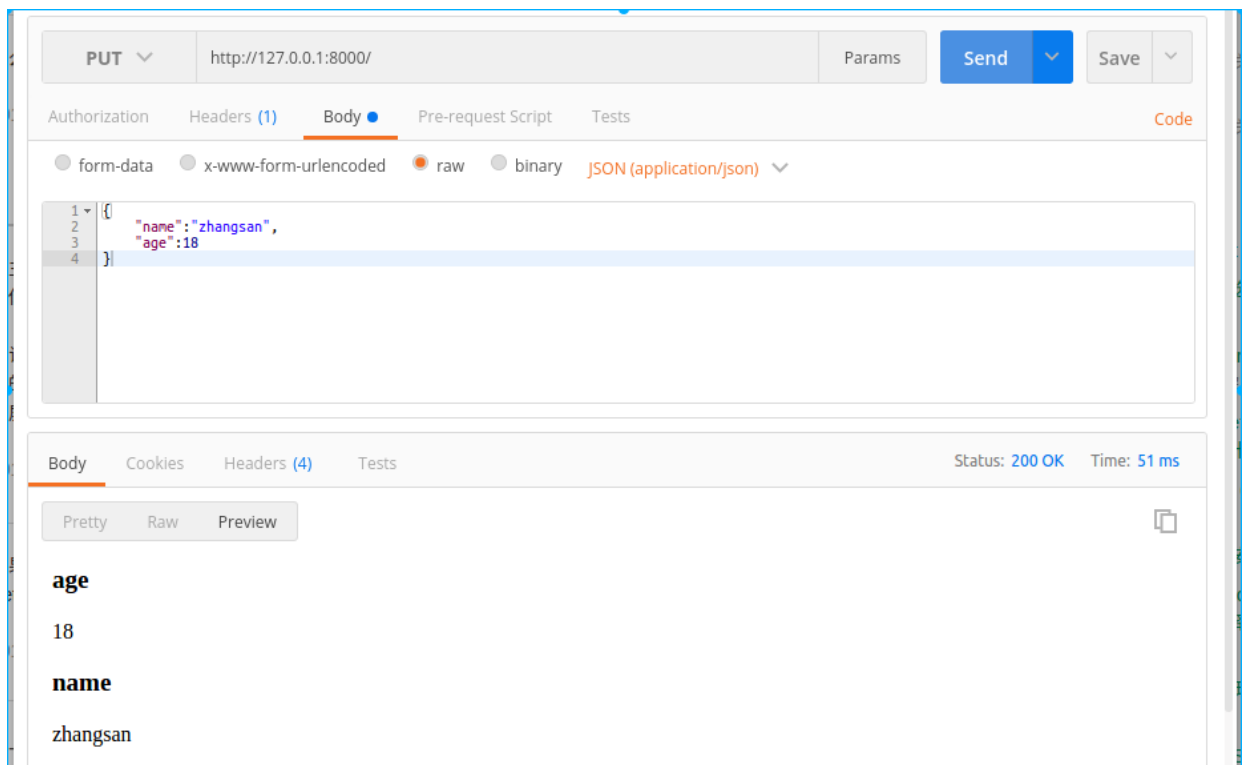
    def post(self):
        if self.json_dict:
            for key, value in self.json_dict.items():
                self.write("<h3>%s</h3><p>%s</p>" % (key, value))

    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()"
```

```
(tornado_py2) python@ubuntu:~/Documents/demo$ python test.py
调用了set_default_headers()
调用了initialize()
调用了prepare()
调用了get()
[I 161008 15:00:26 web:1946] 200 GET / (127.0.0.1) 0.91ms
调用了on_finish()

调用了set_default_headers() ←
调用了initialize()
调用了prepare()
调用了post()
调用了set_default_headers() ←
调用了write_error
[I 161008 15:00:34 web:1946] 200 POST / (127.0.0.1) 0.44ms
调用了on_finish()
```

在正常情况未抛出错误时，调用顺序为：

1. set_default_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()

七、模板

7.1 静态文件

现在有一个预先写好的静态页面文件，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>

7.2 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=""></a>
    <div class="house-desc">
        <div class="landlord-pic"></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=""></a>
  <div class="house-desc">
    <div class="landlord-pic"></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=""></a>
  <div class="house-desc">
    <div class="landlord-pic"></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=""></a>
        <div class="house-desc">
          <div class="landlord-pic"></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"
    }
]
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：

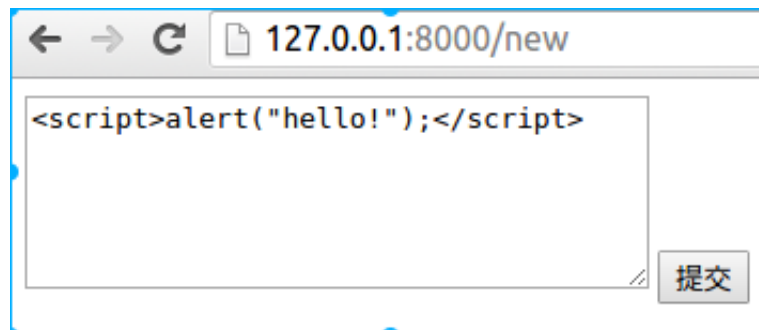
```
class NewHandler(RequestHandler):

    def get(self):
        self.render("new.html", text="")

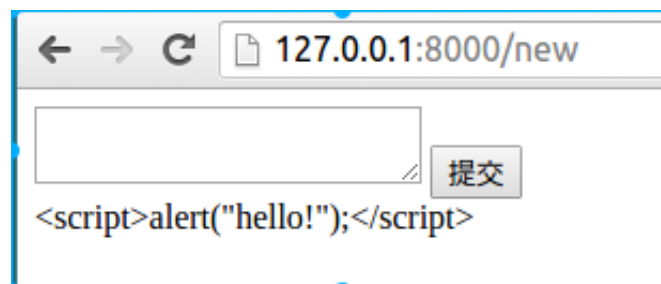
    def post(self):
        text = self.get_argument("text", "")
        print text
        self.render("new.html", text=text)
```

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=""></a>
                <div class="house-desc">
                    <div class="landlord-pic"></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 %}

```


[illegible]

```
{% end %}
```

八、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

8.1 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

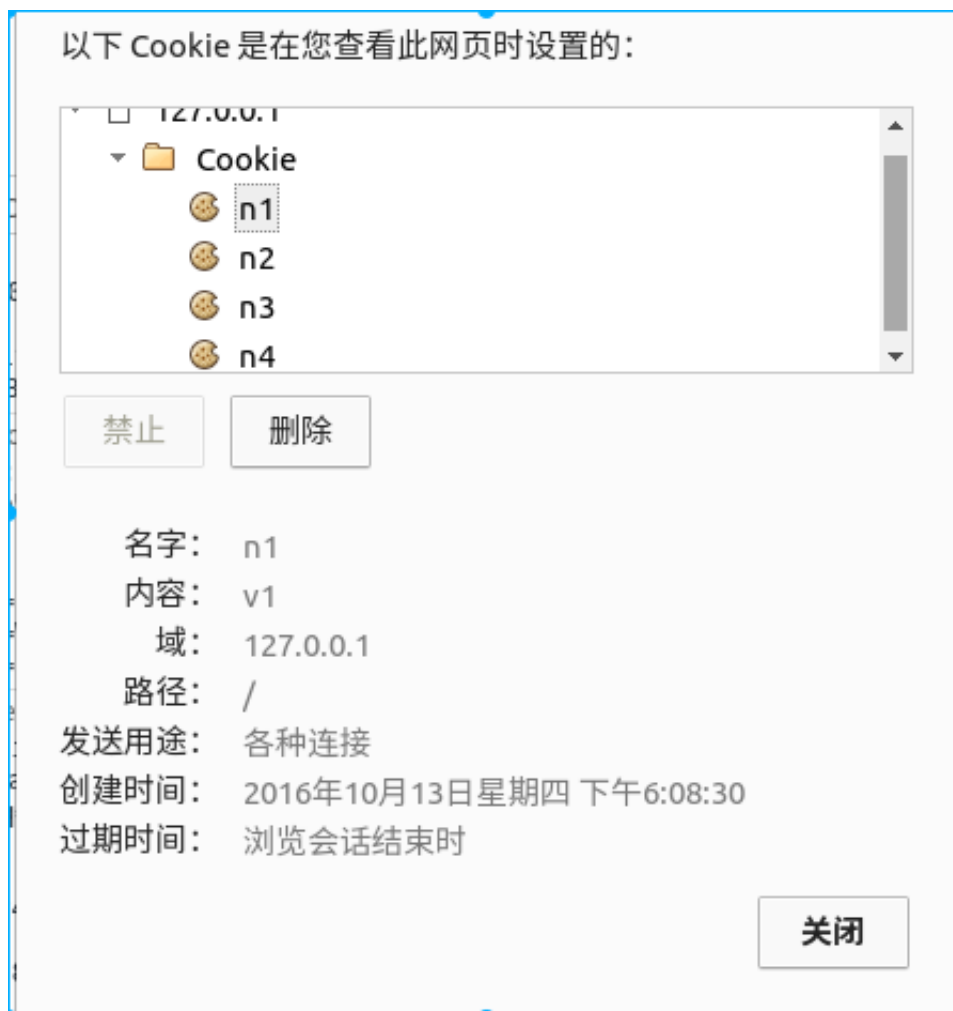
set_cookie(name, value, domain=None, expires=None, path='/', expires_days=None)

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

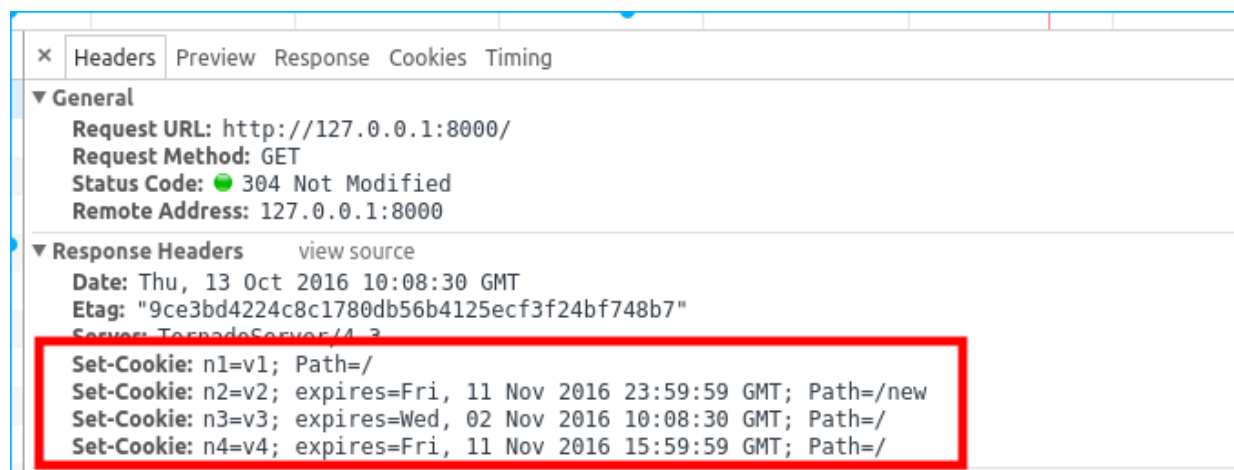
```
import datetime

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进行混淆的密钥 `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 3.4×10^{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>'
        )

```

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。

8.2 XSRF

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 itcast")
```

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 itcast")
```


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>
```

非模板应用

对于不使用模板的应用来说，首先要设置xsrf的Cookie值，可以在任意的Handler中通过获取`self.xsrf_token`的值来生成xsrf并设置Cookie。

下面两种方式都可以起到设置_xsrf Cookie的作用。

```
class XSRFTokenHandler(RequestHandler):
    """专门用来设置_xsrf Cookie的接口"""
    def get(self):
        self.xsrf_token
        self.write("Ok")

class StaticFileHandler(tornado.web.StaticFileHandler):
    """重写StaticFileHandler，构造时触发设置_xsrf Cookie"""
    def __init__(self, *args, **kwargs):
        super(StaticFileHandler, self).__init__(*args, **kwargs)
        self.xsrf_token
```

对于请求携带_xsrf参数，有两种方式：

- 若请求体是表单编码格式的，可以在请求体中添加_xsrf参数
- 若请求体是其他格式的（如json或xml等），可以通过设置HTTP头X-XSRFToken来传递_xsrf值

1. 请求体携带_xsrf参数

新建一个页面xsrf.html:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>测试XSRF</title>
</head>
<body>
  <a href="javascript:;" onclick="xsrfPost()">发送POST请求</a>
  <script src="http://cdn.bootcss.com/jquery/3.1.1/jquery.min.js">
</script>
  <script type="text/javascript">
    //获取指定Cookie的函数
    function getCookie(name) {
      var r = document.cookie.match("\\b" + name + "=([^;]*)\\b");
      return r ? r[1] : undefined;
    }
    //AJAX发送post请求，表单格式数据
    function xsrfPost() {
      var xsrf = getCookie("_xsrf");
      $.post("/new", "_xsrf="+xsrf+"&key1=value1",
function(data) {
      alert("OK");
    });
    }
  </script>
</body>
</html>
```

2. HTTP头X-XSRFToken

新建一个页面json.html:

```
<!DOCTYPE html>
```

```

<html>
<head>
  <meta charset="utf-8">
  <title>测试XSRF</title>
</head>
<body>
  <a href="javascript:;" onclick="xsrPost()">发送POST请求</a>
  <script src="http://cdn.bootcss.com/jquery/3.1.1/jquery.min.js">
</script>
  <script type="text/javascript">
    //获取指定Cookie的函数
    function getCookie(name) {
      var r = document.cookie.match("\\b" + name + "=
([^;]*)\\b");
      return r ? r[1] : undefined;
    }
    //AJAX发送post请求, json格式数据
    function xsrfPost() {
      var xsrf = getCookie("_xsrf");
      var data = {
        key1:1,
        key1:2
      };
      var json_data = JSON.stringify(data);
      $.ajax({
        url: "/new",
        method: "POST",
        headers: {
          "X-XSRFToken":xsrf,
        },
        data:json_data,
        success:function(data) {
          alert("OK");
        }
      })
    }
  </script>
</body>
</html>

```

8.3 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题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。")
```

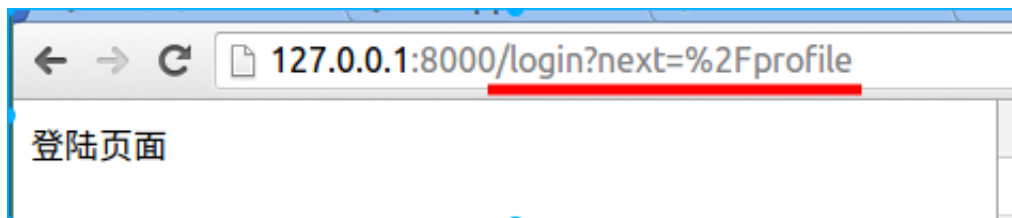
login_url设置

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=logged")
```

用户访问的网址顺序：

Name	Status	Type	Initiator	Size	Time	Timeline - Start Time	60
<input type="checkbox"/> profile	302	text/html	Other	179 B	108 ms	<div></div>	
<input type="checkbox"/> login?next=%2Fprofile	302	text/html	http://127.0.0.1:8...	178 B	38 ms	<div></div>	
<input type="checkbox"/> profile?name=logged	200	document	http://127.0.0.1:8...	220 B	57 ms	<div></div>	