

## 特点

- 非阻塞式服务器
- 每秒可处理数以千计的连接
- 适合于Web实时服务

```
In [1]: import tornado.ioloop
```

```
In [2]: import tornado.web
```

```
In [2]:
```

## 基本使用

- 程序会将URL或者URL范式映射到`tornado.web.RequestHandler`的子类上  
也就是说，被访问的内容应该放到`tornado.web.RequestHandler`的子类中
- 子类中定义`get`或`post`方法分别处理相应的HTTP请求
- URL范式  
采用正则表达式，把匹配的内容放在括号内（即形成一个“分组”）
- 子类包含的一些方法、属性
  - 用于`get`
    - `self.write()`: 输出页面的内容（HTML），参数为字符串
  - 用于`post`
    - `self.get_argument()`: 返回查询字符串，参数为标签的`name`属性
    - `self.set_header()`: 设置一项头，第一个参数为键，第二个参数为值
  - 通用
    - `self.request`: 当前请求的对象，以下为其一些属性
      - `arguments`: 所有的GET或POST参数
      - `files`: 文件列表，获取每个文件以`{"filename": ..., "content_type": ..., "body": ...}`形式的字典存在
      - `path`: 请求路径
      - `headers`: 请求的头信息
- 返回一个错误信息给客户端 直接抛出异常  
如`raise tornado.web.HTTPError(403)`抛出一个403错误
- 构建Web应用  
`tornado.web.Application()`: 返回一个Web应用，参数为一个(URL或URL范式, `RequestHandler`子类)元组形成的列表  
注意: URL或URL范式一般使用r字符串
- 监听端口  
调用`Application`的`listen`方法，参数为端口号
- 启动Web程序  
`tornado.ioloop.IOLoop.instance().start()`

```
In [3]: class MainHandler(tornado.web.RequestHandler):    #tornado.web.RequestHandler的子类
        def get(self):    #定义get方法
            self.write('<html><body><form action="/" method="post">'    #输出html
                '<input type="text" name="message">'
                '<input type="submit" value="Submit">'
                '</form></body></html>')
        def post(self):
            self.set_header("Content-Type", "text/plain")    #设置头信息
            self.write("You wrote " + self.get_argument('message'))    #获取name为message的input内容，并输出相应的内容
```

```
In [4]: application = tornado.web.Application([
        (r'/', MainHandler),
    ])    #构建Web应用
```

```
In [5]: application.listen(8080)    #监听8080端口
```

```
In [6]: tornado.ioloop.IOLoop.instance().start()
```

```
-----
KeyboardInterrupt                                Traceback (most recent call last)
<ipython-input-6-9e0ed89d4235> in <module>()
----> 1 tornado.ioloop.IOLoop.instance().start()

/usr/lib64/python2.7/site-packages/zmq/eventloop/ioloop.pyc in start(self)
    149     def start(self):
    150         try:
--> 151             super(ZMQIOLoop, self).start()
    152         except ZMQError as e:
    153             if e.errno == ETERM:

/usr/lib64/python2.7/site-packages/tornado/ioloop.pyc in start(self)
    659
    660         try:
--> 661             event_pairs = self._impl.poll(poll_timeout)
    662         except Exception as e:
    663             # Depending on python version and IOLoop implementation,

/usr/lib64/python2.7/site-packages/zmq/eventloop/ioloop.pyc in poll(self, timeout)
    120     Event masks will be IOLoop.READ/WRITE/ERROR
    121     """
--> 122     z_events = self._poller.poll(1000*timeout)
    123     return [ (fd,self._remap_events(evt)) for (fd,evt) in z_events ]
    124

/usr/lib64/python2.7/site-packages/zmq/sugar/poll.pyc in poll(self, timeout)
    99     elif isinstance(timeout, float):
    100         timeout = int(timeout)
--> 101     return zmq_poll(self.sockets, timeout=timeout)
    102
    103

/usr/lib64/python2.7/site-packages/zmq/backend/cython/_poll.so in zmq.backend.cytho
n._poll.zmq_poll (zmq/backend/cython/_poll.c:1678) ()

/usr/lib64/python2.7/site-packages/zmq/backend/cython/_poll.so in zmq.backend.cytho
n.checkrc._check_rc (zmq/backend/cython/_poll.c:1879) ()
```

KeyboardInterrupt:

In [0]:

## 重写RequestHandler的方法函数

RequestHandler类定义了一些空函数，以供必要的时候在子类中重写

- 一个请求处理的代码调用次序如下
  1. 创建一个RequestHandler对象
  2. 调用initialize()函数，其参数为Application配置中关键字参数定义
  3. 调用prepare()函数，无论是哪种HTTP请求都会调用该函数，用以输出信息  
如果函数中调用了finish()、send\_error()等函数，那么整个流程中止（也就不会调用4）
  4. 调用HTTP方法对应的函数，get(), post(), put()等
- 常见可复写的方法
  - get\_error\_html(self, status\_code, exception=None, \*\*kwargs): 以字符串形式返回html，显示错误页面
  - get\_current\_user(self): 用于用户认证
  - get\_user\_locale(self): 返回locale对象，以供当前用户使用
  - get\_login\_url(self): 返回登录网址，以供@authenticated装饰器使用
  - get\_template\_path(self): 返回模板文件路径

In [0]:

## 重定向

有两种方法，都含有permanent参数

若为True，则激发一个301 Moved Permanently的HTTP状态，用于永久性重定向

若为False，则为302 Found的普通HTTP状态

- RequestHandler中调用self.redirect()  
常用于逻辑事件触发（如环境变更、用户认证、表单提交等），默认permanent为False
- 直接使用RedirectHandler  
常用于每次匹配到对应URL时即触发，默认permanent为True

```
In [3]: #RequestHandler中调用self.redirect()
class TryRedirect(tornado.web.RequestHandler):
    def post(self):
        self.redirect('/lalalallala', permanent=True) #permanent默认为False
```

```
In [5]: #直接使用RedirectHandler
application = tornado.web.Application([
    ( r'/([a-z]*)', tornado.web.RedirectHandler, {'url' : '/lalalallala', 'permanent'
: False} ),
])

#这里的Handler对象直接使用RedurectHandler
#并且增加第三个元素，为重定向的参数，以字典的形式给出
#包含'url', 'permanent'等
#permanent默认为True
```

In [2]:

## 模板

Tornado中可以使用任何一种python支持的模板

同时，Tornado也自带了高效灵活的模块

模板文件可以是任意后缀，一般为.html

- 控制语句for, while, if, try  
用{%和%}包含起来，而且需要{% end %}作为结束标志
- 表达语句  
用{{和}}包含起来
- 渲染模板  
RequestHandle子类中的self.render()和self.render\_string()函数  
第一个参数为模板文件名  
其余参数为关键字参数，将被作为同名变量传值到模板中去

In [2]:

## Cookie和安全Cookie

- 设置Cookie 调用self.set\_cookie()方法即可  
第一、第二参数分别为键、值
- 对Cookie作签名
  1. 在创建应用时提供关键字参数cookie\_secret的密钥
  2. 通过self.set\_secure\_cookie()设置作了签名的cookie，参数同set\_cookie  
签名过的cookie包含编码过的cookie值、时间戳、HAMC签名
  3. 通过self.get\_secure\_cookie()获取作了签名的cookie，参数为键名  
如果cookie过期或不匹配或没有设置，将返回None

In [2]:

## 用户认证

- 当前已认证的用户信息会被保存到self.current\_user中，默认为None
- 要实现用户认证功能，需要重写方法函数get\_current\_user(self)，函数返回所要写入self.current\_user的用户信息  
简单的用户认证，可以使用cookie来记录用户信息
- 如果强制用户登陆，可以使用对函数使用装饰器@tornado.web.authenticated  
当用户没有登陆时，页面都重定向到login\_url
  - 如果对post方法使用该装饰器  
用户没有登陆时，服务器会返回403错误
  - login\_url作为Application的关键字参数对web应用进行配置
- 另外，Tornado也集成了Google OAuth等第三方认证支持

In [2]:

## XSRF防范

创建Web应用时加入关键值参数`xsrp_cookies=True`开启该防范机制  
开启后，程序会对所有用户设置一个`_xsrp`的cookie值

- 如果POST, PUT, DELET请求中没有`_xsrp`或不匹配，该请求会被直接拒绝
- 开启该机制后，在提交表单时需要加上一个域来提供这个值  
在`<form></form>`标签内的第一行加上表达语句`{{ xsrf_form_html() }}`即可
- 对于AJAX的POST请求  
.....
- 对于PUT和DELET请求  
.....
- 对于需要针对每一个处理器定制XSRF行为  
.....

In [2]:

## 静态文件和主动式文件缓存

- 通过`static_path`参数配置静态文件目录  
如：`static_path = os.path.join( os.path.dirname(__file__), "static" )`  
将以`/static/`为静态文件目录
- 如果需要从根目录访问静态目录下的文件  
可以进行类似如下操作

```
setting = {'static' : os.path.join( os.path.dirname(__file__), "static" )}
application = tornado.web.Application([
    ( r'/(apple-touch-icon\.png)', tornado.web.StaticFileHandler, {'path'
: setting['static']} )
])
```

- 注意URL中文件名是作为一个匹配分组  
因为它需要作为一个分组以方法参数的方式传递到处理器中
- 第三个参数时给出静态目录
- 主动式文件缓存  
在模板中使用`{{ static_url() }}`，参数为基于静态目录的地址  
并且会在尾部添加一个包含v值的查询字符串
  - v值是基于文件内容计算出来的散列值 服务器会将该值发送给目标浏览器，浏览器依据该值对相关内容作永久缓存  
当文件发生改变，或服务器重启时，该值会发生改变，此时浏览器会请求服务器作重新缓存  
否则就一直使用本地缓存
  - 对于nginx下的配置 .....

In [2]:

## 本地化翻译

.....

In [2]:

## UI模块

组件可以拥有自己的css和javascript

并且在一个页面中如果多次调用同一个模块，只加载一次css和javascript

其中css在<head>部分， javascript在</body>之前

- 每个模块为一个独立的python文件  
文件下的访问内容为**tornado.web.UIModule**的子类  
该子类应定义方法函数**render()**，并返回输出内容 该子类中可以通过重写**embedded\_css**，**embedded\_javascript**，**javascript\_files**，**css\_files**方法函数来配置css和javascript  
如：

```
def embedded_css(self):  
    return ".entry { margin-bottom : 1em; }"
```

- 配置**ui\_module**来告知Tornado调用模块  
如：`ui_modules = "uimodules"`来调用**unimodules**模块
- 模板中引用模块

```
{% module Entry(entry) %}
```

表示传值**entry**到模块文件中的**Entry**类，并生成相应的内容输出到该语句的位置

- 如果不需要额外的python语句，可以直接使用模板文件来作为模块
  - 在被调用的模板文件开头加入类似如下语句

```
{{ set_resources( embedded_css = ".entry {margin-bottom : 1em;}") }}
```

- 在调用模块的模板文件中加入类似如下控制语句调用模块

```
{% module Template("entry.html", show_comments = True) %}
```

**set\_resources()**函数只有在该语句调用下的模板中能够使用

- 该语句与**{% include ... %}**不同  
前者只能访问全局模板的命名空间和自己的关键字参数，而不能访问父级模板的命名空间

In [2]:

## 非阻塞式异步请求

.....

In [2]:

## 异步HTTP客户端

.....

In [2]:

## 第三方认证

.....

In [2]:

## 调试模式和自动重载

- 调试模式 调试模式下模板不会被缓冲，而是被程序监视，如果文件发生修改，应用会重新加载配置**debug=True**即可
- 另外，自动重载也作为一个模块独立出来可以通过**tornado.autoload**来调用它

In [2]:

## nginx部署示例

示例部署环境：

- nginx和tornado在同一台机器上
- 四个Tornado服务跑在端口8000~8003上

.....

In [2]:

## WSGI和Google AppEngine

Tornado只对WSGI作有限支持  
而不能使用非阻塞式功能

.....

In [ ]: