tornado-简介和原理

tornado-设计初衷

1. 追求小而精
2. epoll IO 多路复用和协程
3. 支持 WebSocket
4. 单线程程序(GIL 限制，本身某种意义上不启动多进程就是单线程程序)

Python GIL 介绍详情

tronado 应用场景

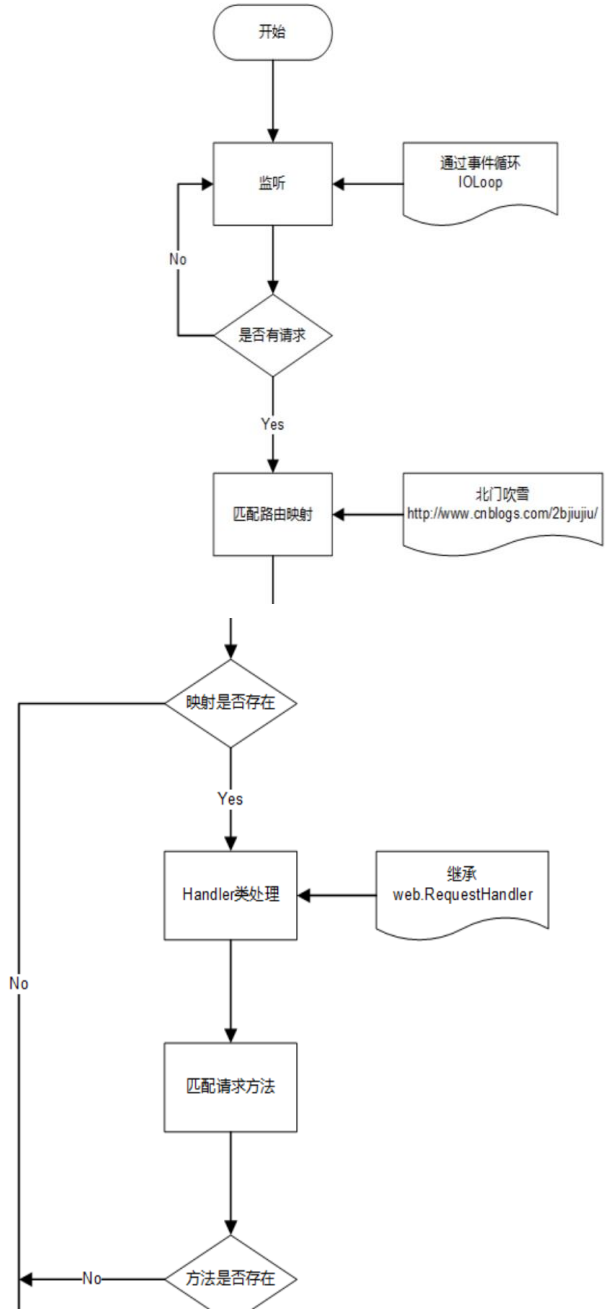
1. 大量的 http 请求连接(大量的用户请求，要求并发性和高性能)

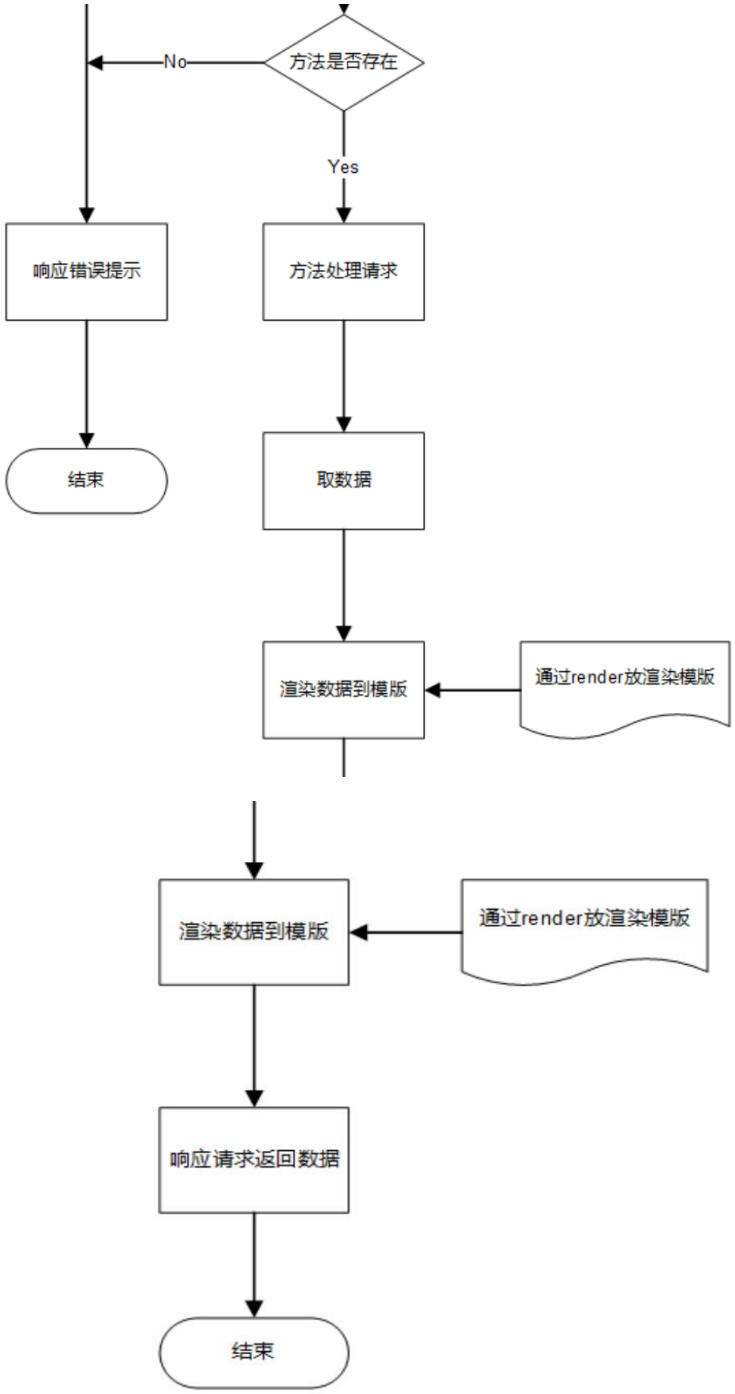
tronado-基础-Hello World-Web 架构

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

```
from  tornado  import  web, ioloop
class  HelloHandler(web.RequestHandler):
        def  get(self,  *args,  **kwargs):
                self.write("Hello 北门吹雪")
def  make_app():
        return  web.Application([
                ("/", HelloHandler),
        ])
if  __name__  ==  '__main__':
        app  =  make_app()
        app.listen(8000)
        ioloop.IOLoop.current().start()
```

tornado 请求原理图


开始
监听
通过事件循环
IOLoop
No
是否有请求
Yes
匹配路由映射
北门吹雪
http://www.cnblogs.com/2bjiujiu/
映射是否存在
Yes
Handler类处理
继承
web.RequestHandler
No
匹配请求方法
No
方法是否存在


方法是否存在
No
Yes
响应错误提示
方法处理请求
结束
取数据
渲染数据到模版
通过render放渲染模版
渲染数据到模版
通过render放渲染模版
响应请求返回数据
结束

经验：

1. tornado 高性能 Web 原理是利用 Linux epoll IO 多路模型和协程异步编程
2. tornado Web 框架核心模块是 web 和 核心事件循环模块是 IOLoop

tornado 入门

Overview

FriendFeed 是一款使用 **Python** 编写的，相对简单的 非阻塞式 Web 服务器。其应用程序使用的 Web 框架看起来有些像 web.py 或者 Google 的 webapp， 不过为了能有效利用非阻塞式服务器环境，这个 Web 框架还包含了一些相关的有用工具 和优化。

Tornado 就是我们在 FriendFeed 的 Web 服务器及其常用工具的开源版本。

Tornado 和现在的主流 Web 服务器框架（包括大多数 Python 的框架）有着明显的区别：**它是非阻塞式服务器，而且速度相当快。**得利于其 非阻塞的方式和对 epoll 的运用，Tornado 每秒可以处理数以千计的连接，因此 Tornado 是实时 Web 服务的一个 理想框架。我们开发这个 Web 服务器的主要目的就是为了处理 FriendFeed 的实时功能 ——在 FriendFeed 的应用里每一个活动用户都会保持着一个服务器连接。（关于如何扩容 服务器，以处理数以千计的客户端的连接的问题，请参阅 The C10K problem ）

以下是经典的 “Hello, world” 示例：

```
import tornado.ioloop
import tornado.web

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")

application = tornado.web.Application([
    (r"/", MainHandler),
])

if __name__ == "__main__":
    application.listen(8888)
    tornado.ioloop.IOLoop.instance().start()
```

查看下面的 Tornado 攻略以了解更多关于 tornado.web 包 的细节。

我们清理了 Tornado 的基础代码，减少了各模块之间的相互依存关系，所以理论上讲， 你可以在自己的项目中独立地使用任何模块，而不需要使用整个包。

下载和安装

自动安装： Tornado 已经列入 PyPI ，因此可以通过 pip 或者 easy_install 来安装。如果你没有安装 libcurl 的话，你需要将其单独安装到系统中。请参见下面的安装依赖一节。注意一点，使用 pip 或 easy_install 安装的 Tornado 并没有包含源代码中的 demo 程序。

手动安装： 下载 tornado-2.0.tar.gz

```
tar xvzf tornado-2.0.tar.gz
cd tornado-2.0
python setup.py build
sudo python setup.py install
```

Tornado 的代码托管在 GitHub 上面。对于 Python 2.6 以上的版本，因为标准库中已经包括了对 epoll 的支持，所以你可以不用 setup.py 编译安装，只要简单地将 tornado 的目录添加到 PYTHONPATH 就可以使用了。

安装需求

Tornado 在 Python 2.5, 2.6, 2.7 中都经过了测试。要使用 Tornado 的所有功能，你需要安装 PycURL (7.18.2 或更高版本) 以及 simplejson (仅适用于 Python 2.5，2.6 以后的版本标准库当中已经包含了对 JSON 的支持)。为方便起见，下面将列出 Mac OS X 和 Ubuntu 中的完整安装方式：

Mac OS X 10.6 (Python 2.6+)

```
sudo easy_install setuptools pycurl
```

Ubuntu Linux (Python 2.6+)

```
sudo apt-get install python-pycurl
```

Ubuntu Linux (Python 2.5)

```
sudo apt-get install python-dev python-pycurl python-simplejson
```

模块索引

最重要的一个模块是 web， 它就是包含了 Tornado 的大部分主要功能的 Web 框架。其它的模块都是工具性质的， 以便让 web 模块更加有用 后面的 Tornado 攻略 详细讲解了 web 模块的使用方法。

主要模块

- web - FriendFeed 使用的基础 Web 框架，包含了 Tornado 的大多数重要的功能
- escape - XHTML, JSON, URL 的编码/解码方法
- database - 对 MySQLdb 的简单封装，使其更容易使用
- template - 基于 Python 的 web 模板系统
- httpclient - 非阻塞式 HTTP 客户端，它被设计用来和 web 及 httpserver 协同工作
- auth - 第三方认证的实现（包括 Google OpenID/OAuth、Facebook Platform、Yahoo BBAuth、FriendFeed OpenID/OAuth、Twitter OAuth）
- locale - 针对本地化和翻译的支持
- options - 命令行和配置文件解析工具，针对服务器环境做了优化

底层模块

- httpserver - 服务于 web 模块的一个非常简单的 HTTP 服务器的实现
- iostream - 对非阻塞式的 socket 的简单封装，以方便常用读写操作
- ioloop - 核心的 I/O 循环

Tornado 攻略

请求处理程序和请求参数

Tornado 的 Web 程序会将 URL 或者 URL 范式映射到 tornado.web.RequestHandler 的子类上去。在其子类中定义了 get() 或 post() 方法，用以处理不同的 HTTP 请求。

下面的代码将 URL 根目录 / 映射到 MainHandler，还将一个 URL 范式 /story/([0-9]+) 映射到 StoryHandler。正则表达式匹配的分组会作为参数引入 的相应方法中：

```
class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("You requested the main page")

class StoryHandler(tornado.web.RequestHandler):
    def get(self, story_id):
        self.write("You requested the story " + story_id)

application = tornado.web.Application([
    (r"/", MainHandler),
    (r"/story/([0-9]+)", StoryHandler),
])
```

你可以使用 get_argument() 方法来获取查询字符串参数，以及解析 POST 的内容：

```
class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write('<html><body><form action="/" method="post">'
                   '<input type="text" name="message">'
                   '<input type="submit" value="Submit">'
                   '</form></body></html>')

    def post(self):
```

```
        self.set_header("Content-Type", "text/plain")
        self.write("You wrote " + self.get_argument("message"))
```

上传的文件可以通过 self.request.files 访问到，该对象将名称（HTML 元素 <input type="file">的 name 属性）对应到一个文件列表。每一个文件都以字典的形式 存在，其格式为 {"filename":..., "content_type":..., "body":...}。

如果你想要**返回一个错误信息给客户端**，例如"403 unauthorized"，只需要抛出一个 tornado.web.HTTPError 异常：

```
if not self.user_is_logged_in():
    raise tornado.web.HTTPError(403)
```

请求处理程序可以通过 **self.request 访问到代表当前请求的对象**。该 HTTPRequest 对象包含了一些有用的属性，包括：

- arguments - 所有的 GET 或 POST 的参数
- files - 所有通过 multipart/form-data POST 请求上传的文件
- path - 请求的路径（？之前的所有内容）
- headers - 请求的开头信息

你可以通过查看源代码 httpserver 模组中 HTTPRequest 的定义，从而了解到它的 所有属性。

重写 RequestHandler 的方法函数

除了 get()/post()等以外，RequestHandler 中的一些别的方法函数，这都是 一些空函数，它们存在的目的是在必要时在子类中重新定义其内容。对于一个请求的处理 的代码调用次序如下：

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。

下面是一个示范 initialize() 方法的例子：

```
class ProfileHandler(RequestHandler):
    def initialize(self, database):
        self.database = database

    def get(self, username):
        ...

app = Application([
    (r'/user/(.*)', ProfileHandler, dict(database=database)),
    ])
```

其它设计用来被复写的方法有：

- get_error_html(self, status_code, exception=None, **kwargs) - 以字符串的形式 返回 HTML，以供错误页面使用。
- get_current_user(self) - 查看下面的用户认证一节
- 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（默认位置 在 Application 设置中）
- get_template_path(self) - 返回模板文件的路径（默认是 Application 中的设置）

重定向(redirect)

Tornado 中的重定向有两种主要方法：self.redirect，或者使用 RedirectHandler。
你可以在使用 RequestHandler （例如 get）的方法中使用 self.redirect，将用户 重定向到别的地方。另外还有一个可选参数 permanent，你可以用它指定这次操作为永久性重定向。
该参数会激发一个 301 Moved Permanently HTTP 状态，这在某些情况下是有用的， 例如，你要将页面的原始链接重定向时，这种方式会更有利于搜索引擎优化（SEO）。
permanent 的默认值是 False，这是为了适用于常见的操作，例如用户端在成功发送 POST 请求 以后的重定向。

```
self.redirect('/some-canonical-page', permanent=True)
```

RedirectHandler 会在你初始化 Application 时自动生成。
例如本站的下载 URL，由较短的 URL 重定向到较长的 URL 的方式是这样的：

```
application = tornado.wsgi.WSGIApplication([
    (r"/([a-z]*)", ContentHandler),
    (r"/static/tornado-0.2.tar.gz", tornado.web.RedirectHandler,
     dict(url="http://github.com/downloads/facebook/tornado/tornado-0.2.tar.gz")),
], **settings)
```

RedirectHandler 的默认状态码是 301 Moved Permanently，不过如果你想使用 302 Found 状态码，你需要将 permanent 设置为 False。

```
application = tornado.wsgi.WSGIApplication([
    (r"/foo", tornado.web.RedirectHandler, {"url":"/bar", "permanent":False}),
], **settings)
```

注意，在 self.redirect 和 RedirectHandler 中，permanent 的默认值是不同的。 这样做是有一定道理的，self.redirect 通常会被用在自定义方法中，是由逻辑事件触发 的（例如环境变更、用户认证、以及表单提交）。而 RedirectHandler 是在每次匹配到请求 URL 时被触发。

模板

你可以在 Tornado 中使用任何一种 Python 支持的模板语言。但是相较于其它模板而言， Tornado 自带的模板系统速度更快，并且也更灵活。具体可以查看 template 模块的源码。
Tornado 模板其实就是 HTML 文件（也可以是任何文本格式的文件），其中包含了 Python 控制结构和表达式，这些控制结构和表达式需要放在规定的格式标记符(markup)中：

```
<html>
```

```
  <head>
     <title>{{ title }}</title>
  </head>
  <body>
    <ul>
      {% for item in items %}
        <li>{{ escape(item) }}</li>
      {% end %}
    </ul>
  </body>
</html>
```

如果你把上面的代码命名为 "template.html"，保存在 Python 代码的同一目录中，你就可以 这样来渲染它：

```
class MainHandler(tornado.web.RequestHandler):
    def get(self):
        items = ["Item 1", "Item 2", "Item 3"]
        self.render("template.html", title="My title", items=items)
```

Tornado 的模板支持"控制语句"和"表达语句"，控制语句是使用 {% 和 %} 包起来的 例如 {% if len(items) > 2 %}。表达语句是使用 {{ 和 }} 包起来的，例如 {{ items[0] }}。

控制语句和对应的 Python 语句的格式基本完全相同。我们支持 if、for、while 和 try，这些语句逻辑结束的位置需要用 {% end %} 做标记。我们还通过 extends 和 block 语句实现了模板继承。这些在 template 模块 的代码文档中有着详细的描述。

表达语句可以是包括函数调用在内的任何 Python 表述。模板中的相关代码，会在一个单独 的名字空间中被执行，这个名字空间包括了以下的一些对象和方法。（注意，下面列表中 的对象或方法在使用 RequestHandler.render 或者 render_string 时才存在的 ，如果你在 RequestHandler 外面直接使用 template 模块，则它们中的大部分是不存在的）。

- escape: tornado.escape.xhtml_escape 的别名
- xhtml_escape: tornado.escape.xhtml_escape 的别名
- url_escape: tornado.escape.url_escape 的别名
- json_encode: tornado.escape.json_encode 的别名
- squeeze: tornado.escape.squeeze 的别名
- linkify: tornado.escape.linkify 的别名
- datetime: Python 的 datetime 模组
- handler: 当前的 RequestHandler 对象
- request: handler.request 的别名
- current_user: handler.current_user 的别名

- locale: handler.locale 的别名
- _: handler.locale.translate 的别名
- static_url: for handler.static_url 的别名
- xsrf_form_html: handler.xsrf_form_html 的别名
- reverse_url: Application.reverse_url 的别名
- Application 设置中 ui_methods 和 ui_modules 下面的所有项目
- 任何传递给 render 或者 render_string 的关键字参数

当你制作一个实际应用时，你会需要用到 Tornado 模板的所有功能，尤其是 模板继承功能。所有这些功能都可以在 template 模块 的代码文档中了解到。（其中一些功能是在 web 模块中实现的，例如 UIModules）

从实现方式来讲，Tornado 的模板会被直接转成 Python 代码。模板中的语句会逐字复制到一个 代表模板的函数中去。我们不会对模板有任何限制，Tornado 模板模块的设计宗旨就是要比 其他模板系统更灵活而且限制更少。所以，当你的模板语句里发生了随机的错误，在执行模板时 你就会看到随机的 Python 错误信息。

所有的模板输出都已经通过 tornado.escape.xhtml_escape 自动转义(escape)，这种默认行为， 可以通过以下几种方式修改：将 autoescape=None 传递给 Application 或者 TemplateLoader、 在模板文件中加入 {% autoescape None %}、或者在简单表达语句 {{ ... }} 写成 {% raw ...%}。另外你可以在上述位置将 autoescape 设为一个自定义函数，而不仅仅是 None。

Cookie 和安全 Cookie

你可以使用 set_cookie 方法在用户的浏览中设置 cookie：

```
class MainHandler(tornado.web.RequestHandler):
    def get(self):
        if not self.get_cookie("mycookie"):
            self.set_cookie("mycookie", "myvalue")
            self.write("Your cookie was not set yet!")
        else:
            self.write("Your cookie was set!")
```

Cookie 很容易被恶意的客户端伪造。加入你想在 cookie 中保存当前登陆用户的 id 之类的信息，你需要对 cookie 作签名以防止伪造。Tornado 通过 set_secure_cookie 和 get_secure_cookie 方法直接支持了这种功能。 要使用这些方法，你需要在创建应用时提供一个密钥，名字为 cookie_secret。 你可以把它作为一个关键词参数传入应用的设置中：

```
application = tornado.web.Application([
    (r"/", MainHandler),
], cookie_secret="61oETzKXQAGaYdkL5gEmGeJJFuYh7EQnp2XdTP1o/Vo=")
```

签名过的 cookie 中包含了编码过的 cookie 值，另外还有一个时间戳和一个 HMAC 签名。如果 cookie 已经过期或者 签名不匹配，get_secure_cookie 将返回 None，这和没有设置 cookie 时的 返回值是一样的。上面例子的安全 cookie 版本如下：

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

```
    def get(self):
        if not self.get_secure_cookie("mycookie"):
            self.set_secure_cookie("mycookie", "myvalue")
            self.write("Your cookie was not set yet!")
        else:
            self.write("Your cookie was set!")
```

用户认证

当前已经认证的用户信息被保存在每一个请求处理器的 self.current_user 当中， 同时在模板的 current_user 中也是。默认情况下，current_user 为 None。
要在应用程序实现用户认证的功能，你需要复写请求处理中 get_current_user() 这 个方法，在其中判定当前用户的状态，比如通过 cookie。下面的例子让用户简单地使用一个 nickname 登陆应用，该登陆信息将被保存到 cookie 中：

```
class BaseHandler(tornado.web.RequestHandler):
    def get_current_user(self):
        return self.get_secure_cookie("user")

class MainHandler(BaseHandler):
    def get(self):
        if not self.current_user:
            self.redirect("/login")
            return
        name = tornado.escape.xhtml_escape(self.current_user)
        self.write("Hello, " + name)

class LoginHandler(BaseHandler):
    def get(self):
        self.write('<html><body><form action="/login" method="post">'
                   'Name: <input type="text" name="name">'
                   '<input type="submit" value="Sign in">'
                   '</form></body></html>')

    def post(self):
        self.set_secure_cookie("user", self.get_argument("name"))
        self.redirect("/")
```

```
application = tornado.web.Application([
    (r"/", MainHandler),
    (r"/login", LoginHandler),
], cookie_secret="61oETzKXQAGaYdkL5gEmGeJJFuYh7EQnp2XdTP1o/Vo=")
```

对于那些必须要求用户登陆的操作，可以使用装饰器 tornado.web.authenticated。 如果一个方法套上了这个装饰器，但是当前用户并没有登陆的话，页面会被重定向到 login_url（应用配置中的一个选项），上面的例子可以被改写成：

```
class MainHandler(BaseHandler):
    @tornado.web.authenticated
    def get(self):
        name = tornado.escape.xhtml_escape(self.current_user)
        self.write("Hello, " + name)

settings = {
    "cookie_secret": "61oETzKXQAGaYdkL5gEmGeJJFuYh7EQnp2XdTP1o/Vo=",
    "login_url": "/login",
}
application = tornado.web.Application([
    (r"/", MainHandler),
    (r"/login", LoginHandler),
], **settings)
```

如果你使用 authenticated 装饰器来装饰 post() 方法，那么在用户没有登陆的状态下， 服务器会返回 403 错误。

Tornado 内部集成了对第三方认证形式的支持，比如 Google 的 OAuth 。参阅 auth 模块 的代码文档以了解更多信息。 for more details. Checkauth 模块以了解更多的细节。在 Tornado 的源码中有一个 Blog 的例子，你也可以从那里看到 用户认证的方法（以及如何在 **MySQL** 数据库中保存用户数据）。

跨站伪造请求的防范

跨站伪造请求(Cross-site request forgery)， 简称为 XSRF，是个性化 Web 应用中常见的一个安全问题。前面的链接也详细讲述了 XSRF 攻击的实现方式。

当前防范 XSRF 的一种通用的方法，是对每一个用户都记录一个无法预知的 cookie 数据，然后要求所有提交的请求中都必须带有这个 cookie 数据。如果此数据不匹配 ，那么这个请求就可能是被伪造的。

Tornado 有内建的 XSRF 的防范机制，要使用此机制，你需要在应用配置中加上 xsrf_cookies 设定：

```
settings = {
    "cookie_secret": "61oETzKXQAGaYdkL5gEmGeJJFuYh7EQnp2XdTP1o/Vo=",
    "login_url": "/login",
    "xsrf_cookies": True,
```

```
}
application = tornado.web.Application([
    (r"/", MainHandler),
    (r"/login", LoginHandler),
], **settings)
```

如果设置了 xsrf_cookies，那么 Tornado 的 Web 应用将对所有用户设置一个 _xsrf 的 cookie 值，如果 POST PUTDELET 请求中没有这 个 cookie 值，那么这个请求会被直接拒绝。如果你开启了这个机制，那么在所有 被提交的表单中，你都需要加上一个域来提供这个值。你可以通过在模板中使用 专门的函数 xsrf_form_html() 来做到这一点：

```
<form action="/new_message" method="post">
  {{ xsrf_form_html() }}
  <input type="text" name="message"/>
  <input type="submit" value="Post"/>
</form>
```

如果你提交的是 AJAX 的 POST 请求，你还是需要在每一个请求中通过脚本添加上 _xsrf 这个值。下面是在 FriendFeed 中的 AJAX 的 POST 请求，使用了 jQuery 函数来为所有请求组东添加 _xsrf 值：

```
function getCookie(name) {
    var r = document.cookie.match("\\b" + name + "=([^;]*)\\b");
    return r ? r[1] : undefined;
}

jQuery.postJSON = function(url, args, callback) {
    args._xsrf = getCookie("_xsrf");
    $.ajax({url: url, data: $.param(args), dataType: "text", type: "POST",
        success: function(response) {
        callback(eval("(" + response + ")"));
    }});
};
```

对于 PUT 和 DELETE 请求（以及不使用将 form 内容作为参数的 POST 请求） 来说，你也可以在 HTTP 头中以 X-XSRFToken 这个参数传递 XSRF token。

如果你需要针对每一个请求处理器定制 XSRF 行为，你可以重写 RequestHandler.check_xsrf_cookie()。例如你需要使用一个不支持 cookie 的 API， 你可以通过将 check_xsrf_cookie() 函数设空来禁用 XSRF 保护机制。然而如果 你需要同时支持 cookie 和非 cookie 认证方式，那么只要当前请求是通过 cookie 进行认证的，你就应该对其使用 XSRF 保护机制，这一点至关重要。

静态文件和主动式文件缓存

你能通过在应用配置中指定 static_path 选项来提供静态文件服务：

```
settings = {
    "static_path": os.path.join(os.path.dirname(__file__), "static"),
    "cookie_secret": "61oETzKXQAGaYdkL5gEmGeJJFuYh7EQnp2XdTP1o/Vo=",
    "login_url": "/login",
    "xsrf_cookies": True,
}
application = tornado.web.Application([
    (r"/", MainHandler),
    (r"/login", LoginHandler),
    (r"/(apple-touch-icon\.png)", tornado.web.StaticFileHandler, dict(path=settings['static_path'])),
], **settings)
```

这样配置后，所有以 /static/ 开头的请求，都会直接访问到指定的静态文件目录， 比如 http://localhost:8888/static/foo.png 会从指定的静态文件目录中访问到 foo.png 这个文件。同时 /robots.txt 和/favicon.ico 也是会自动作为静态文件处理（即使它们不是以 /static/ 开头）。

在上述配置中，我们使用 StaticFileHandler 特别指定了让 Tornado 从根目录伺服 apple-touch-icon.png 这个文件，尽管它的物理位置还是在静态文件目录中。（正则表达式 的匹配分组的目的是向 StaticFileHandler 指定所请求的文件名称，抓取到的分组会以 方法参数的形式传递给处理器。）通过相同的方式，你也可以从站点的更目录伺服 sitemap.xml 文件。当然，你也可以通过在 HTML 中使用正确的 <link /> 标签来避免这样的根目录 文件伪造行为。

为了提高性能，在浏览器主动缓存静态文件是个不错的主意。这样浏览器就不需要发送 不必要的 If-Modified-Since 和 Etag 请求，从而影响页面的渲染速度。Tornado 可以通过内建的"静态内容分版(static content versioning)"来直接支持这种功能。

要使用这个功能，在模板中就不要直接使用静态文件的 URL 地址了，你需要在 HTML 中使用 static_url() 这个方法来提供 URL 地址：

```
<html>
   <head>
      <title>FriendFeed - {{ _("Home") }}</title>
   </head>
   <body>
     <div><img src="{{ static_url("images/logo.png") }}"/></div>
   </body>
 </html>
```

static_url() 函数会将相对地址转成一个类似于 /static/images/logo.png?v=aae54 的 URI，v 参数是 logo.png 文件的散列值， Tornado 服务器会把它发给浏览器，并以此为依据让浏览器对相关内容做永久缓存。

由于 v 的值是基于文件的内容计算出来的，如果你更新了文件，或者重启了服务器 ，那么就会得到一个新的 v 值，这样浏览器就会请求服务器以获取新的文件内容。 如果文件的内容没有改变，浏览器就会一直使用本地缓存的文件，这样可以显著提高页 面的渲染速度。

在生产环境下，你可能会使用 nginx 这样的更有利于静态文件 伺服的服务器，你可以将 Tornado 的文件缓存指定到任何静态文件服务器上面，下面 是 FriendFeed 使用的 nginx 的相关配置：

```
location /static/ {
```

```
    root /var/friendfeed/static;
    if ($query_string) {
        expires max;
    }
}
```

本地化

不管有没有登陆，当前用户的 locale 设置可以通过两种方式访问到：请求处理器的 self.locale 对象、以及模板中的 locale 值。Locale 的名称（如 en_US）可以通过 locale.name 这个变量访问到，你可以使用 locale.translate 来进行本地化 翻译。在模板中，有一个全局方法叫 _()，它的作用就是进行本地化的翻译。这个翻译方法有两种使用形式：

```
_("Translate this string")
```

它会基于当前 locale 设置直接进行翻译，还有一种是：

```
_("A person liked this", "%(num)d people liked this", len(people)) % {"num": len(people)}
```

这种形式会根据第三个参数来决定是使用单数或是复数的翻译。上面的例子中，如果 len(people) 是 1 的话，就使用第一种形式的翻译，否则，就使用第二种形式 的翻译。

常用的翻译形式是使用 Python 格式化字符串时的"固定占位符(placeholder)"语法，（例如上面的 %(num)d），和普通占位符比起来，固定占位符的优势是使用时没有顺序限制。

一个本地化翻译的模板例子：

```
<html>
   <head>
      <title>FriendFeed - {{ _("Sign in") }}</title>
   </head>
   <body>
     <form action="{{ request.path }}" method="post">
       <div>{{ _("Username") }} <input type="text" name="username"/></div>
       <div>{{ _("Password") }} <input type="password" name="password"/></div>
       <div><input type="submit" value="{{ _("Sign in") }}"/></div>
       {{ xsrf_form_html() }}
     </form>
   </body>
 </html>
```

默认情况下，我们通过 Accept-Language 这个头来判定用户的 locale，如果没有， 则取 en_US 这个值。如果希望用户手动设置一个 locale 偏好，可以在处理请求的类中复写 get_user_locale 方法：

```
class BaseHandler(tornado.web.RequestHandler):
    def get_current_user(self):
        user_id = self.get_secure_cookie("user")
        if not user_id: return None
        return self.backend.get_user_by_id(user_id)

    def get_user_locale(self):
        if "locale" not in self.current_user.prefs:
            # Use the Accept-Language header
            return None
        return self.current_user.prefs["locale"]
```

如果 get_user_locale 返回 None，那么就会再去取 Accept-Language header 的值。

你可以使用 tornado.locale.load_translations 方法获取应用中的所有已存在的翻 译。它会找到包含有特定名字的 CSV 文件的目录，如 es_GT.csv fr_CA.csv 这 些 csv 文件。然后从这些 CSV 文件中读取出所有的与特定语言相关的翻译内容。典型的用例 里面，我们会在 Tornado 服务器的 main() 方法中调用一次该函数：

```
def main():
    tornado.locale.load_translations(
        os.path.join(os.path.dirname(__file__), "translations"))
    start_server()
```

你可以使用 tornado.locale.get_supported_locales() 方法得到支持的 locale 列表。Tornado 会依据用户当前的 locale 设置以及已有的翻译，为用户选择 一个最佳匹配的显示语言。比如，用户的 locale 是 es_GT 而翻译中只支持了 es， 那么 self.locale 就会被设置为 es。如果找不到最接近的 locale 匹配，self.locale 就会就会取备用值 es_US。

查看 locale 模块 的代码文档以了解 CSV 文件的格式，以及其它的本地化方法函数。

UI 模块

Tornado 支持一些 UI 模块，它们可以帮你创建标准的，易被重用的应用程序级的 UI 组件。 这些 UI 模块就跟特殊的函数调用一样，可以用来渲染页面组件，而这些组件可以有自己的 CSS 和 JavaScript。

例如你正在写一个博客的应用，你希望在首页和单篇文章的页面都显示文章列表，你可以创建 一个叫做 Entry 的 UI 模块，让他在两个地方分别显示出来。首选需要为你的 UI 模块 创建一个 Python 模组文件，就叫 uimodules.py 好了：

```
class Entry(tornado.web.UIModule):
    def render(self, entry, show_comments=False):
```

```
        return self.render_string(
            "module-entry.html", entry=entry, show_comments=show_comments)
```

然后通过 ui_modules 配置项告诉 Tornado 在应用当中使用 uimodules.py：

```
class HomeHandler(tornado.web.RequestHandler):
    def get(self):
        entries = self.db.query("SELECT * FROM entries ORDER BY date DESC")
        self.render("home.html", entries=entries)

class EntryHandler(tornado.web.RequestHandler):
    def get(self, entry_id):
        entry = self.db.get("SELECT * FROM entries WHERE id = %s", entry_id)
        if not entry: raise tornado.web.HTTPError(404)
        self.render("entry.html", entry=entry)

settings = {
    "ui_modules": uimodules,
}
application = tornado.web.Application([
    (r"/", HomeHandler),
    (r"/entry/([0-9]+)", EntryHandler),
], **settings)
```

在 home.html 中，你不需要写繁复的 HTML 代码，只要引用 Entry 就可以了：

```
{% for entry in entries %}
  {% module Entry(entry) %}
{% end %}
```

在 entry.html 里面，你需要使用 show_comments 参数来引用 Entry 模块，用来 显示展开的 Entry 内容：

```
{% module Entry(entry, show_comments=True) %}
```

你可以为 UI 模型配置自己的 CSS 和 JavaScript ，只要复写 embedded_css、embedded_javascript、javascipt_files、css_files 就可以了：

```
class Entry(tornado.web.UIModule):
    def embedded_css(self):
        return ".entry { margin-bottom: 1em; }"

    def render(self, entry, show_comments=False):
        return self.render_string(
```

```
            "module-entry.html", show_comments=show_comments)
```

即使一页中有多个相同的 UI 组件，UI 组件的 CSS 和 JavaScript 部分只会被渲染一次。 CSS 是在页面的 <head> 部分，而 JavaScript 被渲染在页面结尾 </body> 之前的位 置。

在不需要额外 Python 代码的情况下，模板文件也可以当做 UI 模块直接使用。 例如前面的例子可以以下面的方式实现，只要把这几行放到 module-entry.html 中就可以了：

```
{{ set_resources(embedded_css=".entry { margin-bottom: 1em; }") }}
<!-- more template html... -->
```

这个修改过的模块式模板可以通过下面的方法调用：

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

set_resources 函数只能在 {% module Template(...) %} 调用的模板中访问到。 和 {% include ... %} 不同，模块式模板使用了和它们的上级模板不同的命名空间——它们只能访问到全局模板命名空间和它们自己的关键字参数。

非阻塞式异步请求

当一个处理请求的行为被执行之后，这个请求会自动地结束。因为 Tornado 当中使用了 一种非阻塞式的 I/O 模型，所以你可以改变这种默认的处理行为——让一个请求一直保持 连接状态，而不是马上返回，直到一个主处理行为返回。要实现这种处理方式，只需要 使用 tornado.web.asynchronous 装饰器就可以了。

使用了这个装饰器之后，你必须调用 self.finish() 以完成 HTTP 请求，否则 用户的浏览器会一直处于等待服务器响应的状态：

```
class MainHandler(tornado.web.RequestHandler):
    @tornado.web.asynchronous
    def get(self):
        self.write("Hello, world")
        self.finish()
```

下面是一个使用 Tornado 内置的异步请求 HTTP 客户端去调用 FriendFeed 的 API 的例 子：

```
class MainHandler(tornado.web.RequestHandler):
    @tornado.web.asynchronous
    def get(self):
        http = tornado.httpclient.AsyncHTTPClient()
        http.fetch("http://friendfeed-api.com/v2/feed/bret",
                   callback=self.on_response)

    def on_response(self, response):
        if response.error: raise tornado.web.HTTPError(500)
        json = tornado.escape.json_decode(response.body)
```

```
        self.write("Fetched " + str(len(json["entries"])) + " entries "
                   "from the FriendFeed API")
        self.finish()
```

例子中，**当 get() 方法返回时，请求处理还没有完成。在 HTTP 客户端执行它的回 调函数 on_response() 时，从浏览器过来的请求仍然是存在的，只有在显式调用了 self.finish() 之后，才会把响应返回到浏览器。**

关于更多异步请求的高级例子，可以参阅 demo 中的 chat 这个例子。它是一个使用 long polling 方式 的 AJAX 聊天室。如果你使用到了 long polling，你可能需要复写 on_connection_close()， 这样你可以在客户连接关闭以后做相关的清理动作。（请查看该方法的代码文档，以防误用。）

异步 HTTP 客户端

Tornado 包含了两种非阻塞式 HTTP 客户端实现：

SimpleAsyncHTTPClient 和 CurlAsyncHTTPClient。

前者是直接基于 IOLoop 实现的，因此无需外部依赖关系。

后者作为 Curl 客户端，需要安装 libcurl 和 pycurl 后才能正常工作，但是对于使用 到 HTTP 规范中一些不常用内容的站点来说，它的兼容性会更好。为防止碰到旧版本中异步界面的 bug，我们建议你安装最近的版本的 libcurl 和 pycurl。

这些客户端都有它们自己的模组(tornado.simple_httpclient 和 tornado.curl_httpclient)，你可以通过 tornado.httpclient 来指定使用哪一种 客户端，默认情况下使用的是 SimpleAsyncHTTPClient，如果要修改默认值，只要 在一开始调用 AsyncHTTPClient.configure 方法即可：

```
AsyncHTTPClient.configure('tornado.curl_httpclient.CurlAsyncHTTPClient')
```

第三方认证

Tornado 的 auth 模块实现了现在很多流行站点的用户认证方式，包括 Google/Gmail、Facebook、Twitter、Yahoo 以及 FriendFeed。这个模块可以让用户使用这些站点的账户来登陆你自己的应用，然后你就可以在授权的条件下访问原站点的一些服 务，比如下载用户的地址薄，在 Twitter 上发推等。

下面的例子使用了 Google 的账户认证，Google 账户的身份被保存到 cookie 当中，以便 以后的访问使用：

```
class GoogleHandler(tornado.web.RequestHandler, tornado.auth.GoogleMixin):
    @tornado.web.asynchronous
    def get(self):
        if self.get_argument("openid.mode", None):
            self.get_authenticated_user(self._on_auth)
            return
        self.authenticate_redirect()

    def _on_auth(self, user):
        if not user:
```

```
            self.authenticate_redirect()
            return
        # Save the user with, e.g., set_secure_cookie()
```

请查看 auth 模块的代码文档以了解更多的细节。

调试模式和自动重载

如果你将 debug=True 传递给 Application 构造器，该 app 将以调试模式 运行。在调试模式下，模板将不会被缓存，而这个 app 会监视代码文件的修改， 如果发现修改动作，这个 app 就会被重新加载。在开发过程中，这会大大减少 手动重启服务的次数。然而有些问题（例如 import 时的语法错误）还是会让服务器 下线，目前的 debug 模式还无法避免这些情况。

调试模式和 HTTPServer 的多进程模式不兼容。在调试模式下，你必须将 HTTPServer.start 的参数设为不大于 1 的数字。

调试模式下的自动重载功能可以通过独立的模块 tornado.autoreload 调用， 作为测试运行器的一个可选项目，tornado.testing.main 中也有用到它。

性能

一个 Web 应用的性能表现，主要看它的整体架构，而不仅仅是前端的表现。 和其它的 Python Web 框架相比，Tornado 的速度要快很多。

我们在一些流行的 Python Web 框架上（Django、 web.py、CherryPy）， 针对最简单的 Hello, world 例子作了一个测试。对于 Django 和 web.py，我们使用 Apache/mod_wsgi 的方式来带，CherryPy 就让它自己裸跑。这也是在生产环境中各框架常用 的部署方案。**对于我们的 Tornado，使用的部署方案为前端使用 nginx 做反向代理，带动 4 个线程模式的 Tornado，这种方案也是我们推荐的在生产环境下的 Tornado 部署方案（根据具体的硬件情况，我们推荐一个 CPU 核对应一个 Tornado 伺服实例， 我们的负载测试使用的是四核处理器）。**

我们使用 Apache Benchmark (ab)，在另外一台机器上使用了如下指令进行负载测试：

```
ab -n 100000 -c 25 http://10.0.1.x/
```

在 AMD Opteron 2.4GHz 的四核机器上，结果如下图所示：

在我们的测试当中，相较于第二快的服务器，Tornado 在数据上的表现也是它的 4 倍之 多。即使只用了一个 CPU 核的裸跑模式，Tornado 也有 33% 的优势。这个测试不见得非常科学，不过从大体上你可以看出，我们开发 Tornado 时对于性能 的注重程度。和其他的 Python Web 开发框架相比，它不会为你带来多少延时。

生产环境下的部署

在 FriendFeed 中，我们使用 nginx 做负载均衡和静态文件伺服。 我们在多台服务器上，同时部署了多个 Tornado 实例，通常，一个 CPU 内核 会对应一个 Tornado 线程。

因为我们的 Web 服务器是跑在负载均衡服务器（如 nginx）后面的，所以需要把 xheaders=True 传到 HTTPServer 的构造器当中去。这是为了让 Tornado 使用 X-Real-IP 这样的的 header 信息来获取用户的真实 IP 地址，如果使用传统 的方法，你只能得到这台负载均衡服务器的 IP 地址。

下面是 nginx 配置文件的一个示例，整体上与我们在 FriendFeed 中使用的差不多。 它假设 nginx 和 Tornado 是跑在同一台机器上的，四个 Tornado 服务跑在 8000-8003 端口上：

```
user nginx;
worker_processes 1;
error_log /var/log/nginx/error.log;
pid /var/run/nginx.pid;

events {
    worker_connections 1024;
    use epoll;
}

http {
    # Enumerate all the Tornado servers here
    upstream frontends {
        server 127.0.0.1:8000;
        server 127.0.0.1:8001;
        server 127.0.0.1:8002;
        server 127.0.0.1:8003;
    }
    include /etc/nginx/mime.types;
    default_type application/octet-stream;
    access_log /var/log/nginx/access.log;
    keepalive_timeout 65;
    proxy_read_timeout 200;
    sendfile on;
    tcp_nopush on;
```

```
tcp_nodelay on;
gzip on;
gzip_min_length 1000;
gzip_proxied any;
gzip_types text/plain text/html text/css text/xml
           application/x-javascript application/xml
           application/atom+xml text/javascript;

# Only retry if there was a communication error, not a timeout
# on the Tornado server (to avoid propagating "queries of death"
# to all frontends)
proxy_next_upstream error;
server {
    listen 80;
    # Allow file uploads
    client_max_body_size 50M;
    location ^~ /static/ {
        root /var/www;
        if ($query_string) {
            expires max;
        }
    }
    location = /favicon.ico {
        rewrite (.*) /static/favicon.ico;
    }
    location = /robots.txt {
        rewrite (.*) /static/robots.txt;
    }

    location / {
        proxy_pass_header Server;
        proxy_set_header Host $http_host;
        proxy_redirect false;
        proxy_set_header X-Real-IP $remote_addr;
```

```
            proxy_set_header X-Scheme $scheme;
            proxy_pass http://frontends;
        }
    }
}
```

WSGI 和 Google AppEngine

Tornado 对 WSGI 只提供了有限的支持，即使如此，因为 WSGI 并不支持非阻塞式的请求，所以如果你使用 WSGI 代替 Tornado 自己的 HTTP 服务的话，那么你将无法使用 Tornado 的异步非阻塞式的请求处理方式。 比如@tornado.web.asynchronous、httpclient 模块、auth 模块， 这些将都无法使用。
你可以通过 wsgi 模块中的 WSGIApplication 创建一个有效的 WSGI 应用（区别于 我们用过的 tornado.web.Application）。下面的例子展示了使用内置的 WSGI CGIHandler 来创建一个有效的 Google AppEngine 应用。

```
import tornado.web
import tornado.wsgi
import wsgiref.handlers

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")

if __name__ == "__main__":
    application = tornado.wsgi.WSGIApplication([
        (r"/", MainHandler),
    ])
    wsgiref.handlers.CGIHandler().run(application)
```

请查看 demo 中的 appengine 范例，它是一个基于 Tornado 的完整的 AppEngine 应用。

注意事项和社区支持

因为 FriendFeed 以及其他 Tornado 的主要用户在使用时都是基于 nginx 或者 Apache 代理之后的。所以现在 Tornado 的 HTTP 服务部分并不完整，它无法处理多行的 header 信息，同时对于一 些非标准的输入也无能为力。
你可以在 Tornado 开发者邮件列表 中讨论和提交 bug。

漏洞目录

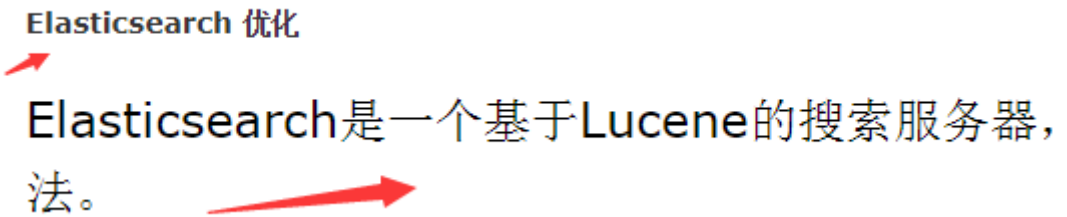
@sebug

使用 Tornado 和协程爬取博客园文章

Python3.5 后 Tornado 官方建议使用 async 和 await 的方式实现异步程序，尝试了下使用 Tornado 和协程爬取博客园的文章并使用 peewee_async 异步写入 MySQL 数据库。

一. 博客园文章抓取测试：

这里我以我自己的一篇文章详情作为测试 url，https://www.cnblogs.com/FG123/p/9934244.html ，主要是抓取文章标题、内容及作者信息：



文章标题、内容、作者用户名可通过上述的详情页 url 获取，但是作者信息需通过 http://www.cnblogs.com/mvc/blog/news.aspx?blogApp=FG123 获取，FG123 是我这篇文章的作者用户名，下面看使用 beautiful soup 抓取测试的代码及结果：

```
detail_article_html = requests.get("https://www.cnblogs.com/FG123/p/9934244.html").content
author_profile_html = requests.get("http://www.cnblogs.com/mvc/blog/news.aspx?blogApp=FG123").content
detail_soup = BeautifulSoup(detail_article_html)
title = detail_soup.find(id="cb_post_title_url").get_text()
info = detail_soup.find(id="cnblogs_post_body")
author_soup = BeautifulSoup(author_profile_html)
author = author_soup.select('div > a')
author_name = author[0].get_text()
blog_age = author[1].get_text()
fans_num = author[2].get_text()
follow_num = author[3].get_text()
print("文章标题：{}".format(title))
print("博主昵称：{}".format(author_name))
print("博主园龄：{}".format(blog_age))
print("粉丝数：{}".format(fans_num))
```

```
print("关注数：{}".format(follow_num))
print("文章内容：{}".format(info))
```

结果：

```
文章标题：Elasticsearch 优化
博主昵称：Harvard_Fly
博主园龄：3年7个月
粉丝数：11
关注数：2
文章内容：<div class="blogpost-body" id="cnblogs_post_body"><p><span style="font-size: 18pt">Elasticsearch是一个基于
在日常项目中使用它遇到的一些问题及优化解决办法。</span></p>
```

二．使用 Tornado 和协程异步抓取逻辑：

这里的抓取逻辑采用 tornado 官方文档爬虫例子的逻辑，使用 Tornado 的 Queue 实现异步生产者/消费者模式，当 Queue 满时会切换协程，首先定义协程通过解析 url 获取相关链接并去除无效的链接：

```
async def get_links_from_url(url):
    """
    通过 AsyncHTTPClient 异步 fetch url，
    通过 BeautifulSoup 提取解析内容中的所有 url
    :param url:
    :return:
    """
    response = await httpclient.AsyncHTTPClient().fetch(url)
    print('fetched %s' % url)

    html = response.body.decode("utf8", errors='ignore')
    soup = BeautifulSoup(html)
    return set([urljoin(url, remove_fragment(a.get("href")))
```

```
            for a in soup.find_all("a", href=True)])


def remove_fragment(url):
    """
    去除无效的链接
    :param url:
    :return:
    """
    pure_url, frag = urldefrag(url)
    return pure_url
```

当前 url 通过调用协程获取它包含的有效 url_list，并将非外链接的 url 放入 tornado 的 queue 中：

```
async def fetch_url(current_url):
        """
        fetching 是已爬取过的 url 集合，
        通过调用协程 get_links_from_url 获取 current_url 所有的 url，
        并将 非外链接 放入到 queue 中
        :param current_url:
        :return:
        """
        if current_url in fetching:
            return

        print('fetching %s' % current_url)
        fetching.add(current_url)
        urls = await get_links_from_url(current_url)
        fetched.add(current_url)

        for new_url in urls:
            # 非外链接
            if new_url.startswith(base_url) and new_url.endswith(".html"):
                await q.put(new_url)
```

使用 async for 的方式取出 queue 中的 url，并调用协程 fetch_url 获取它包含的 urls，调用协程 get_info_data 获取 url 页面详情数据：

```
async def worker():
```

```
        """
        使用 async for 的方式取出 q 中的 url
        并调用协程 fetch_url 获取它包含的 urls
        调用协程 get_info_data 获取 url 页面详情数据
        :return:
        """
        async for url in q:
            if url is None:
                return
            try:
                await fetch_url(url)
                await get_info_data(url)
            except Exception as e:
                print('Exception: %s %s' % (e, url))
            finally:
                q.task_done()
```

定义主协程，通过 tornado 的 gen.multi 同时初始化 concurrency 个协程，并将协程放入到事件循环中等待完成，等到队列全部为空或超时的时候放入与协程数量相同的 None 来结束协程的事件循环。

```
async def main():
    """
    主协程，通过 tornado 的 gen.multi 同时初始化 concurrency 个协程，
    并将协程放入到事件循环中等待完成，等到队列全部为空或超时
    :return:
    """
    q = queues.Queue()
    start = time.time()
    fetching, fetched = set(), set()

    # 放入初始 url 到队列
    await q.put(base_url)

    workers = gen.multi([worker() for _ in range(concurrency)])
    await q.join(timeout=timedelta(seconds=300))
    assert fetching == fetched
    print('Done in %d seconds, fetched %s URLs.' % (
```

```
        time.time() - start, len(fetched)))

    # 队列中放入 concurrency 数量的 None 结束相应协程 在 worker()中取到 None 会结束
    for _ in range(concurrency):
        await q.put(None)

    await workers
```

三. 使用 peewee_async 和 aiomysql 将爬取的数据异步写入 MySQL 数据库

使用 peewee 创建并生成 model：

```
# coding:utf-8
from peewee import *
import peewee_async

database = peewee_async.MySQLDatabase(
    'xxx', host="192.168.xx.xx",
    port=3306, user="root", password="xxxxxx"
)

objects = peewee_async.Manager(database)

database.set_allow_sync(True)
class Blogger(Model):
    article_id = CharField(max_length=50, verbose_name="文章 ID")
    title = CharField(max_length=150, verbose_name="标题")
    content = TextField(null=True, verbose_name="内容")
    author_name = CharField(max_length=50, verbose_name="博主昵称")
    blog_age = CharField(max_length=50, verbose_name="园龄")
    fans_num = IntegerField(null=True, verbose_name="粉丝数")
    follow_num = IntegerField(null=True, verbose_name="关注数")

    class Meta:
        database = database
        table_name = "blogger"
def init_table():
    database.create_tables([Blogger])
```

```
31
33 if __name__ == "__main__":
34     init_table()
```

获取博客文章的详情信息，并将信息异步写入 MySQL 数据库：

```
 1 async def get_info_data(url):
 2     """
 3     获取详情信息并异步写入 MySQL 数据库
 4     :param url:
 5     :return:
 6     """
 7     response = await httpclient.AsyncHTTPClient().fetch(url)
 8     html = response.body.decode("utf8")
 9     soup = BeautifulSoup(html)
10     title = soup.find(id="cb_post_title_url").get_text()
11     content = soup.find(id="cnblogs_post_body")
12     name = url.split("/")[3]
13     article_id = url.split("/")[-1].split(".")[0]
14     author_url = "http://www.cnblogs.com/mvc/blog/news.aspx?blogApp={}".format(name)
15     author_response = await httpclient.AsyncHTTPClient().fetch(author_url)
16     author_html = author_response.body.decode("utf8")
17     author_soup = BeautifulSoup(author_html)
18     author = author_soup.select('div > a')
19     author_name = author[0].get_text()
20     blog_age = author[1].get_text()
21     fans_num = author[2].get_text()
22     follow_num = author[3].get_text()
23     await objects.create(
24         Blogger, title=title,
25         article_id=article_id,
26         content=content,
27         author_name=author_name,
28         blog_age=blog_age,
29         fans_num=fans_num,
30         follow_num=follow_num
```

```
31          )
```

爬取结果：

article_id	title	content	author_name	blog_age	fans_num	follow_num
9955309	JQuery模拟网页中自定义鼠标右键菜单	<div class="blogpost-body" id="cnblogs_post_body"... 79K	粥里有勺糖	1个月	0	0
9951554	算法学习—动态规划之装载问题	<div class="blogpost-body cnblogs-markdown" id="c... 6K	Stars-one	1年3个月	76	3
9941685	The 2018 ACM-ICPC Asia Qingdao Regional Contest（部分题解）	<div class="blogpost-body" id="cnblogs_post_body"... 44K	Reqaw	1年7个月	13	15
9954091	Evosuite使用方法入门	<div class="blogpost-body" id="cnblogs_post_body"... 10K	野生学霸	11个月	0	1
9953845	基于GDAL库，读取.nc文件（以海洋表温数据为例）C++版	<div class="blogpost-body" id="cnblogs_post_body"... 19K	thyou	9个月	1	0
9954878	隐马尔可夫模型（HMM）及Viterbi算法	<div class="blogpost-body cnblogs-markdown" id="c... 13K	jclian91	1年2个月	28	0
9955032	函数内存分配	<div class="blogpost-body" id="cnblogs_post_body"... 11K	CompileLife	11个月	0	1
9945871	神经网络中反向传播与梯度下降的基本概念	<div class="blogpost-body cnblogs-markdown" id="c... 14K	UniversalAIPla	3年11个月	740	9
9953749	使用C#把Tensorflow训练的.pb文件用在生产环境	<div class="blogpost-body cnblogs-markdown" id="c... 6K	bbird2018	5年11个月	29	0
9954823	写个简单的chrome插件-京东商品历史价格查询	<div class="blogpost-body cnblogs-markdown" id="c... 6K	-云-	1年9个月	7	2
9954455	Shell面试题	<div class="blogpost-body cnblogs-markdown" id="c... 8K	合合合村	4个月	16	4
1574616	博客园手机版	<div class="blogpost-body" id="cnblogs_post_body"... 974B	博客园团队	10年4个月	4350	0
9953436	java单例模式实现	<div class="blogpost-body" id="cnblogs_post_body"... 34K	LearnAndGet	1年10个月	4	2
9953744	Spring基础系列-参数校验	<div class="blogpost-body cnblogs-markdown" id="c... 41K	唯一浩哥	4年10个月	205	30
9954988	python 内置函数总结（大部分）	<div class="blogpost-body" id="cnblogs_post_body"... 100K	旦复旦兮	1年10个月	145	6
9953978	CTC（Connectionist Temporal Classification）介绍	<div class="blogpost-body cnblogs-markdown" id="c... 15K	PilgrimHui	1年6个月	11	4
9944340	【温故知新】—BABYLON.js学习之路·前辈经验（二）	<div class="blogpost-body" id="cnblogs_post_body"... 81K	柳洁琼Elena	1年1个月	16	8

简单体验了下使用 Tornado 结合协程的方式爬取博客园，这里我开启了 10 个协程，已经感觉速度很快了，协程间的切换开销是非常小的，而且一个线程或进程可以拥有多个协程，经过实测相比多线程的爬虫确实要快些。

Python 面试题 Tornado 的核心是什么？

Tornado 的核心是什么？Tornado 的核心是 ioloop 和 iostream 这两个模块，前者提供了 一个高效的 I/O 事件循环，后者则封装了 一个无阻塞的 socket 。 通过向 ioloop 中添加网络 I/O 事件，利用无阻塞的 socket ，再搭配相应的回调函数，便可达到梦寐以求的高效异步执行。

一、Tornado 简介

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。

二、Tornado 特点:

作为 Web 框架，是一个轻量级的 Web 框架，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%。

三、Tornado 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。

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。

四、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

2，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。

四、Tornado 开发方向：

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

五、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。

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

Tornado 的核是什么?

Tornado 的核心是 ioloop 和 iostream 这两个模块，前者提供了一个高效的 I/O 事件循环，后者则封装了 一个无阻塞的 socket 。通过向 ioloop 中添加网络 I/O 事件，利用无阻塞的 socket ，再搭配相应的回调 函数，便可达到梦寐以求的高效异步执行。

Django 本身提供了 runserver，为什么不能用来部署?

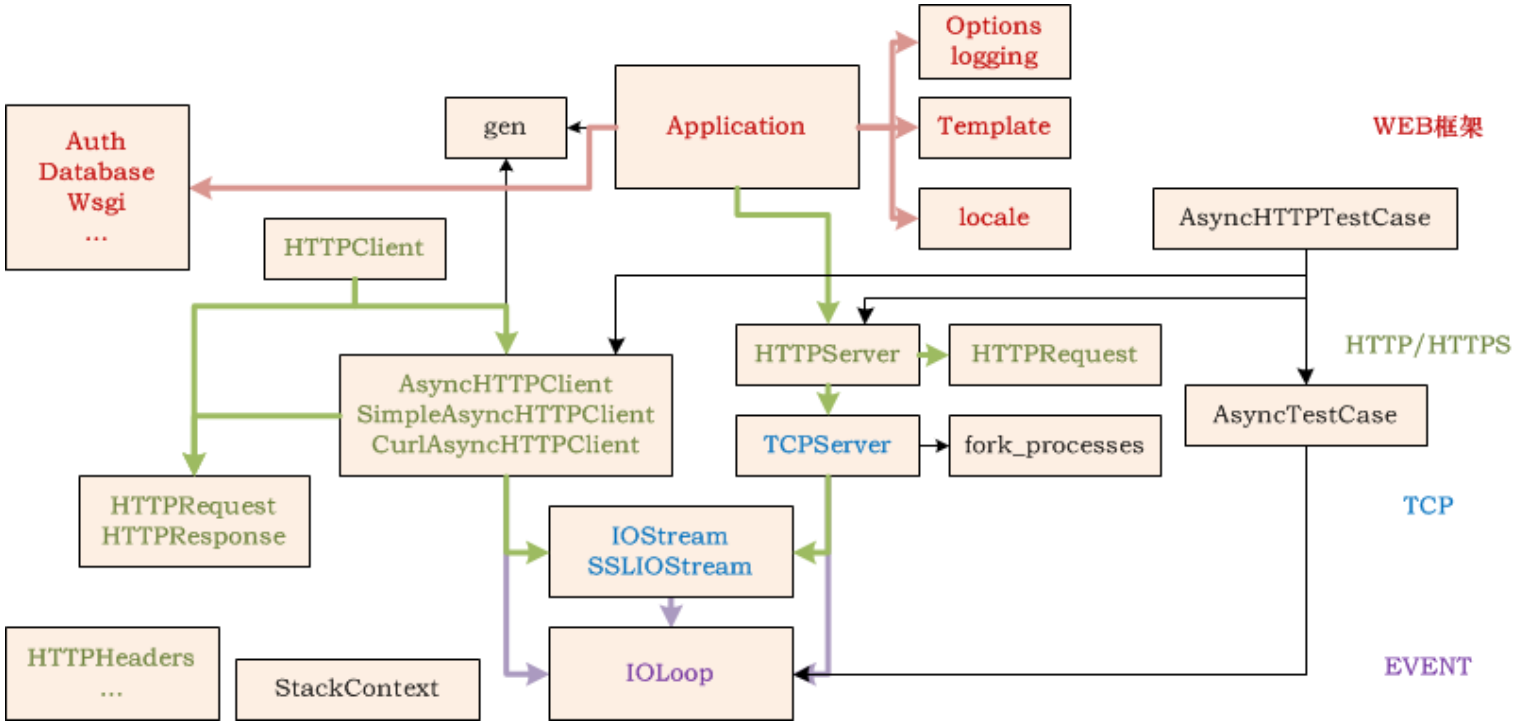
runserver 方法是调试 Django 时经常用到的运行方式，它使用 Django 自带的

WSGI Server 运行，主要在测试和开发中使用，并且 runserver 开启的方式也是单进程 。

uWSGI 是一个 Web 服务器，它实现了 WSGI 协议、uwsgi、http 等协议。注意 uwsgi 是一种通信协议，而 uWSGI 是实现 uwsgi 协议和 WSGI 协议的 Web 服务器。uWSGI 具有超快的性能、低内存占用和多 app 管理等优点，并且搭配着 Nginx

就是一个生产环境了，能够将用户访问请求与应用 app 隔离开，实现真正的部署 。相比来讲，支持的并发量更高，方便管理多进程，发挥多核的优势，提升性能。

大概了解 Tornado 框架的设计模型



从上面的图可以看出，Tornado 不仅仅是一个 WEB 框架，它还完整地实现了 HTTP 服务器和客户端，在此基础上提供 WEB 服务。它可以分为四层：

最底层的 EVENT 层处理 IO 事件；

TCP 层实现了 TCP 服务器，负责数据传输；

HTTP/HTTPS 层基于 HTTP 协议实现了 HTTP 服务器和客户端；

最上层为 WEB 框架，包含了处理器、模板、数据库连接、认证、本地化等等 WEB 框架需要具备的功能。

在 tornado 的子目录中，每个模块都应该有一个.py 文件，你可以通过检查他们来判断你是否从已经从代码仓库中完整的迁出了项目。在每个源代码的文件中，你都可以发现至少一个大段落的用来解释该模块的 doc string，doc string 中给出了一到两个关于如何使用该模块的例子。

下面首先介绍 Tornado 的模块按功能分类。

Tornado 模块分类

1. Core web framework

tornado.web — 包含 web 框架的大部分主要功能，包含 RequestHandler 和 Application 两个重要的类

tornado.httpserver — 一个无阻塞 HTTP 服务器的实现

tornado.template — 模版系统

tornado.escape — HTML,JSON,URLs 等的编码解码和一些字符串操作
tornado.locale — 国际化支持
2. Asynchronous networking 底层模块
tornado.ioloop — 核心的 I/O 循环
tornado.iostream — 对非阻塞式的 socket 的简单封装，以方便常用读写操作
tornado.httpclient — 一个无阻塞的 HTTP 服务器实现
tornado.netutil — 一些网络应用的实现，主要实现 TCPServer 类
3. Integration with other services
tornado.auth — 使用 OpenId 和 OAuth 进行第三方登录
tornado.database — 简单的 MySQL 服务端封装
tornado.platform.twisted — 在 Tornado 上运行为 Twisted 实现的代码
tornado.websocket — 实现和浏览器的双向通信
tornado.wsgi — 与其他 python 网络框架/服务器的相互操作
4. Utilities
tornado.autoreload — 生产环境中自动检查代码更新
tornado.gen — 一个基于生成器的接口，使用该模块保证代码异步运行
tornado.httputil — 分析 HTTP 请求内容
tornado.options — 解析终端参数
tornado.process — 多进程实现的封装
tornado.stack_context — 用于异步环境中对回调函数的上下文保存、异常处理
tornado.testing — 单元测试
理解 Tornado 的核心框架之后，就能便于我们后续的理解。