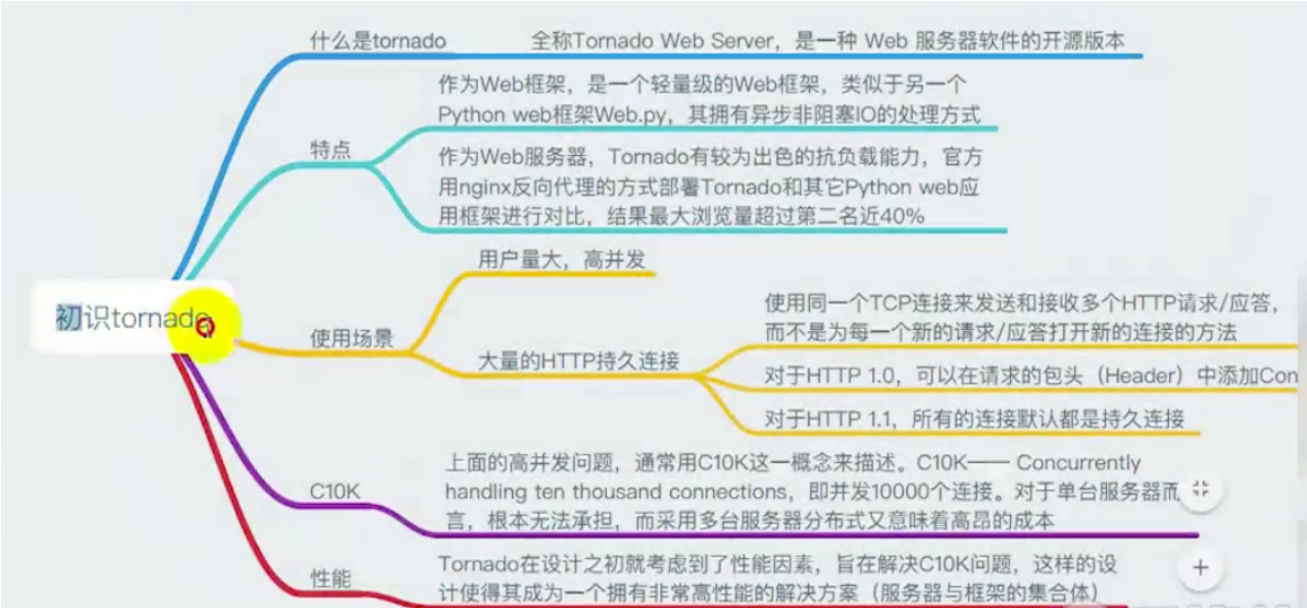
Torando的介绍

> `Tornado` 是一个 `Python web` 框架和异步网络库， 通过使用非阻塞网络 `I/O`， `Tornado` 可以支撑上万级的连接，处理 `长连接，WebSockets` ，和其他需要与每个用户保持长久连接的应用.



说明:

> Tornado 应该运行在类 Unix 的平台上， 在线上部署的时候，为了最佳的性能和扩展性，仅推荐 Linux 和 BSD平台，因为需要充分利用linux的epoll和BSD的Kqueue，这个也是Tornado不依靠多进程/多线程达到高性能的原因

ps:

> https://www.cnblogs.com/jeakeven/p/5435916.html

Torando和Django的比较

特性	Django	Torando
路由系统	有	有
视图函数	有	有
模板引擎	有	有
ORM操作	有	无
cookie	有	有
session	有	无
缓存，信号，Form，Admin	有	无

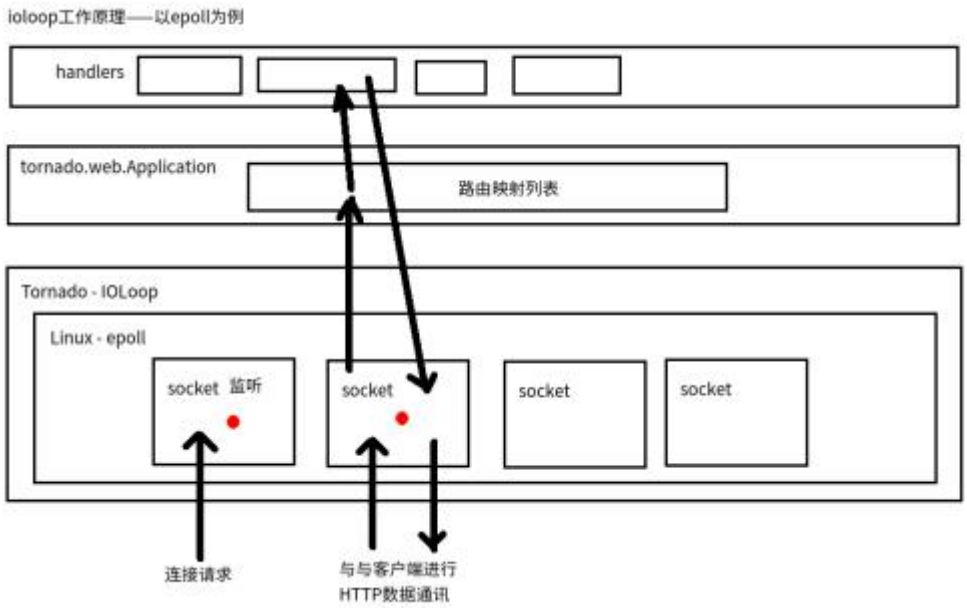
Torando入门程序

```
'''
tornado 的基础 web 框架模块
'''
import tornado.web
import tornado.options

'''
tornado 的核心 IO 循环模块，封装了 linux 的 epoll 和 BSD 的 Kqueue， 是 Tornado 高效的基础
'''
import tornado.ioloop


'''
类比 Django 中的 CBV 模式
一个业务处理的逻辑类
'''
class IndexHandler(tornado.web.RequestHandler):

    '''
    处理 get 请求的， 不能处理 post 请求
    '''
    def get(self):
        print('hello world')
        '''
        将数据返回给 web 的页面，类似于 HttpResponse
        '''
        self.write('this is web page!')

    def post(self, *args, **kwargs):
        print(*args, **kwargs)


class StoryHandler(tornado.web.RequestHandler):

    def get(self, id, *args, **kwargs):
        print(id, args, kwargs)
        self.write(id)


if __name__ == '__main__':
    '''
    实例化一个对象
    Application 是 tornado.web 的一个核心类
    里面保存了路由映射表，有一个 listen 方法， 创建了一个 socket 服务器，并绑定了一个端口 8001
    '''
    app = tornado.web.Application([
        (r'/', IndexHandler),
        (r'/story/([a-zA-Z0-9]+)/', StoryHandler)
    ])

    ## 绑定监听端口， 此时我们的服务器并没有开始监听
    app.listen(8001)
```

```
    '''
    IOLoop.current：返回当前线程的 IOLoop 实例
    IOLoop.start：启动 IOLoop 实例的 I/O 循环，同时开启了监听
    '''
    tornado.ioloop.IOLoop.current().start()
```

执行过程：

- 第一步：执行脚本，监听 `8888` 端口
- 第二步：浏览器客户端访问 `/ --> http://127.0.0.1:8888/`
- 第三步：服务器接受请求，并交由对应的类处理该请求
- 第四步：类接受到请求之后，根据请求方式（`post/get/delete...`）的不同调用并执行相应的方法
- 第五步：方法返回值的字符串内容发送浏览器



HttpServer的介绍

```
import tornado.httpserver

# 实例化一个httpserver的实例
httpServer = tornado.httpserver.HTTPServer(app)

# 绑定端口
httpServer.listen(8000)
```

但上述 tornado 默认启动的是一个进程

如何启动多个进程?

```
import tornado.httpserver

# 实例化一个httpserver的实例
httpServer = tornado.httpserver.HTTPServer(app)

# 绑定端口
<!-- httpServer.listen(8000) -->
httpServer.bind(8000)  # 将服务器绑定到指定到端口
httpServer.start(5)   # num默认开启一个进程， 如果值大于0，创建对应个数的进程
```

但是上面存在三个问题：

- 所有的子进程都是由一个命令启动的，无法做到不停止服务的情况下修改代码。如果我想修改某一个进程中的代码，就必须得停止所有的
- 所有的进程都共享一个端口号，想要分别监控非常困难

由于上面的问题， 我们不建议使用上面的方式开启多个进程， 因此我们以后建议使用 `app.listen(8000)`

options的使用

tornado为我们提供了一个便捷的 tornado.options 模块

基础方法与属性：

> tornado.options.define() : 用来定义变量的方法

```
参数:
name : 变量名， 必须保证其唯一性， 不然会报错
default: 默认值
type : 设置变量的类型， 传入值的时候， 会根据类型进行转换， 转换失败回报错， 可以是 str， int ，
float， 如果没有设置，则会根据default的值进行转换， 但如果default没有设置， 那么则不会进行转换
help : 提示信息

### 使用示例:
tornado.options.define('port', default = 8000)
tornado.options.define('list', default = [], type=str, mutiple=True)
```

> tornado.options.options : 全局的 options 对象， 所有定义的变量， 都可以作为该对象的属性

```
tornado.options.options.port
```

> tornado.options.parse_command_line() : 转换命令行参数， 并将转换的参数值保存在 options对象中

```
python server.py --port=9000  --list=good,stephen,lxx
```

> tornado.options.parse_config_file(path) : 从配置文件导入参数

```
加载在同一级的config文件
tornado.options.parse_config_file("config")
```

缺点：

- 上述的配置文件方式要求我们必须使用 Python 的语法格式去写
- 调用参数的时候，不支持字典类型

因此使用下面的方式：

创建一个 config.py 的文件

```
options = {
    "port" : 8080,
    "names" : ["stephen", 'lxxx', 'xxx']
}

import config

print("list:", config.options.port)
```

Torando快速开始

> 由一个简单的Demo来说，MVC的结构框架

MVC的形象解释 MVC的原理

基础工程

从新整理一个基础工程项目代码：

```
├── application.py      :  管理路由映射关系文件
├── config.py           :   所有的配置文件
├── models              :   数据库文件夹
├── server.py           :   启动服务文件
├── static              :   项目静态目录文件
├── template            :   项目模板文件
└── views               :   项目的视图处理文件夹
    ├── __init__.py
    └── index.py
```

各个文件的代码：

`application.py` 文件：

{% fold %}

```
#author:shangzekai

import tornado.web
from views import index
import config

class Application(tornado.web.Application):

    def __init__(self):
        path = [
            (r'/', index.IndexHandler),
        ]
        super(Application, self).__init__(path, **config.settings)
```

{% endfold %}

`server.py` 文件:

{% fold %}

```
import config

import tornado.ioloop

from application import Application


if __name__ == '__main__':
    app = Application()
    app.listen(config.options['port'])
    '''
    IOLoop.current: 返回当前线程的 IOLoop 实例
    IOLoop.start: 启动 IOLoop 实例的 I/O 循环，同时开启了监听
    '''
    tornado.ioloop.IOLoop.current().start()
```

{% endfold %}

`config.py` 文件:

{% fold %}

```
import os
BASE_DIR = os.path.dirname(__file__)

options = {
    'port':8010
}
```

```
mysql = {
    'dbhost':'localhost',
    'dbuser':'root',
    'dbpwd':'123qwe',
    'dbname':'test',
    'dbcharset':'utf8'
}

settings = {
    'debug' : True,
    'static_path' : os.path.join(BASE_DIR, 'static'),
    'template_path' : os.path.join(BASE_DIR, 'template'),
    'xsrf_cookies' : True,
    'login_url' : 'login'
}
```

{% endfold %}

Application的讲解

配置(config.py)

```
'static_path' : os.path.join(BASE_DIR, 'static'),
'template_path' : os.path.join(BASE_DIR, 'template'),
```

`debug`：设置`tornado`是否工作在调试模式下，默认为`False`即工作在生产模式下

如果debug设置为 True， 则会可有如下效果

- 自动重启

```
1. tornado应用会监控源代码文件，当有代码改动的时候便会重启服务器， 减少手动重启服务器的次数
2. 如果保存后代码有错误会导致重启失败，修改错误后需要手动重启
3. 也可以通过 autoreload = True 来单独设置自动重启
```

- 取消缓存编译的模板， 开发阶段需要关闭缓存
- 取消缓存静态文件
- 提供错误追踪信息

路由系统

> 路由系统其实就是`url`和 类 的对应关系，这里不同于其他框架，其他很多框架均是`url`对应 函数，`Tornado`中每个`url`对应的是一个类。

请求部分

```
import tornado.ioloop
import tornado.web


class MainHandler(tornado.web.RequestHandler):
    def initialize(self, name, age):
        self.name = name
```

```
        self.age = age

    def get(self):
        self.write("Hello, world")

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

application = tornado.web.Application([
    (r"/index", MainHandler),
    (r"/story/([0-9]+)", StoryHandler, {"name":'zhangsan', 'age':12}),
])


if __name__ == "__main__":
    application.listen(80)
    tornado.ioloop.IOLoop.instance().start()
```

> 反向路由解析

```
tornado.web.url(r'/xxx', index.xxHandler, name='xxx')
```

> requestHandler

- 获取get请求的参数值:

```
self.get_query_argument(name, default, strip=True) :  返回值
self.get_query_arguments(name, strip=True) : 返回一个列表
```

- 获取post请求的参数值:

```
self.get_body_argument(name, default, strip=True)
self.get_body_arguments(name,  strip=True)
```

- 既获取get的又获取post的参数:

```
self.get_argument(name, default, strip=True)
self.get_arguments(name, strip=True)
```

> request对象

存储了请求相关的信息

- method： HTTP 请求的方式
- host : 请求的主机名
- query： 请求的参数部分
- version： 请求的Http版本
- headers： 请求的头信息， 字典类型
- remote_ip : 客户端IP
- files： 用户上传的文件

> tornado.httputil.HTTPFile 对象

接收到的文件对象

- filename： 文件的实际名字
- body： 文件的数据实体
- content_type : 文件的类型

```
def post():
    filesDict = self.request.files

    for inputname in filesDict:
        filesArr = filesDict[inputname]
        for fileObj in filesArr:
            filePath = os.path.join(config.BASE_DIR, 'upfile/'+fileObj.filename)
            with open(filepath, 'wb') as fp:
                fp.write(fileObj.body)
```

响应部分

```
self.write() : 刷新缓存区

例子:
info = {
    "name" :'zhansgan',
    "age"  : 16
}

直接以json的格式返回
self.write(info)
使用json.dumps(info)变成字符串
self.write(json.dumps(info))

注意:
    自己手动序列化的，content-type的属性是text/html的， 而自动序列化的， content-type的属性是application/json的



self.finish() : 也是刷新缓存区， 但是会关闭当次请求通道， 在finish下面就不能再write了， 没有意义了

self.set_header(name, value): 手动设置一个名为name， 值为value的响应头字段

self.set_status(status_code, reason): 设置响应状态吗

self.redirect(url) : 重定向url
```

模板引擎

`Tornao`中的模板语言和`django`中类似，模板引擎将模板文件载入内存，然后将数据嵌入其中，最终获取到一个完整的字符串，再将字符串返回给请求者

`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。

- 普通变量渲染

```
{{username}}
```

- for循环

字典格式：

```
dict = {"name":'zhangsan', "age":12}

{% for key, val in dict.items() %}
     {{key}}--{{val}}
{% end %}
```

列表格式：

```
info = ['lisi', 'wangwu', 'zhaoliu']

{% for item in info %}
    {{item}}
{% end %}
```

判断语句：

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

{% if name == 'zekai' %}
    欢迎用户 {{ name }} 登录
{% else %}
    您还没有登录
{% end %}
```

** static_url生成静态文件的路径 **

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <title>XXX</title>
    <link href="{{static_url("css/common.css")}}" rel="stylesheet" />
</head>
<body>
    <script src="{{static_url("js/jquery-1.8.2.min.js")}}"></script>
</body>
</html>
```

母版

> 通过 `extends` 和 `block` 语句实现了模板继承

layout.html

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <title>XXX</title>
    <link href="{{static_url("css/common.css")}}" rel="stylesheet" />
    {% block CSS %}{% end %}
</head>
<body>

    <div class="pg-header">

    </div>

    {% block RenderBody %}{% end %}

    <script src="{{static_url("js/jquery-1.8.2.min.js")}}"></script>

    {% block JavaScript %}{% end %}
</body>
</html>
```

index.html

```
{% extends 'layout.html'%}
{% block CSS %}
    <link href="{{static_url("css/index.css")}}" rel="stylesheet" />
{% end %}

{% block RenderBody %}
```

```
    <h1>Index</h1>

    <ul>
    {%  for item in li %}
        <li>{{item}}</li>
    {% end %}
    </ul>

{% end %}

{% block JavaScript %}

{% end %}
```

导入

```
<div>
    <ul>
        <li>1024</li>
        <li>42区</li>
    </ul>
</div>
```

index.html

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <title>XXX</title>
    <link href="{{static_url("css/common.css")}}" rel="stylesheet" />
</head>
<body>

    <div class="pg-header">
        {% include 'header.html' %}
    </div>

    <script src="{{static_url("js/jquery-1.8.2.min.js")}}"></script>

</body>
</html>
```

Model的讲解

> 在`Tornado3.0`版本以前提供`tornado.database`模块用来操作`MySQL`数据库，而从`3.0`版本开始，此模块就被独立出来，作为`torndb`包单独提供。`torndb`只是对`MySQLdb`的简单封装，不支持`Python 3`

user.py

```
import pymysql
```

```
class UserModel():
    def __init__(self):
        try:
            self.db = pymysql.Connection(host='127.0.0.1', database='test',
user='root', password='123qwe', charset='utf8',cursorclass=pymysql.cursors.DictCursor)
        except Exception as e:
            return print(e)

    def getInfo(self):
        try:
            cursor = self.db.cursor()
            temp = "select * from admin"
            effect_row = cursor.execute(temp)
            result = cursor.fetchall()
            self.db.commit()
            cursor.close()
            self.db.close()

            return result
        except Exception as e:
            return print(e)
```

防止SQL注入

搭建一个登陆的环境

app.py

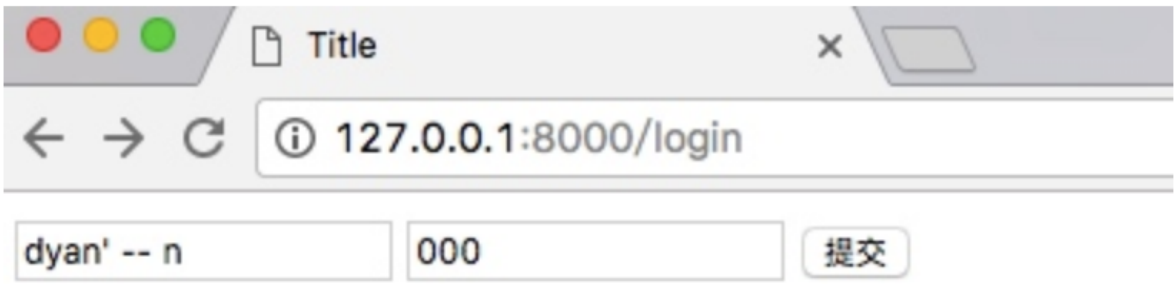
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import tornado.ioloop
import tornado.web
import pymysql

class LoginHandler(tornado.web.RequestHandler):
    def get(self):
        self.render('login.html')

    def post(self, *args, **kwargs):
        username = self.get_argument('username',None)
        pwd = self.get_argument('pwd', None)

        # 创建数据库连接
        conn = pymysql.connect(host='127.0.0.1', port=3306, user='root',
passwd='123456', db='shop')
        cursor = conn.cursor()

        # %s 要加上'' 否则会出现KeyboardInterrupt的错误
        temp = "select name from userinfo where name='%s' and password='%s'" %
(username, pwd)
        effect_row = cursor.execute(temp)
        result = cursor.fetchone()
```

```
        conn.commit()
        cursor.close()
        conn.close()

        if result:
            self.write('登录成功! ')
        else:
            self.write('登录失败! ')


settings = {
    'template_path':'template',
}

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

if __name__ == "__main__":
    application.listen(8000)
    tornado.ioloop.IOLoop.instance().start()
```

在template文件夹下，放入login.html文件：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <form method="post" action="/login">
        <input type="text" name="username" placeholder="用户名"/>
        <input type="text" name="pwd" placeholder="密码"/>
        <input type="submit" value="提交" />
    </form>
</body>
</html>
```

> 至此一个简单的登陆系统就说完了，按照正常的方式都能登陆成功，接下来我们看一下，非法用户的登陆



看一下服务端执行的SQL语句，就不难理解了，密码部分被注释掉了：

```
select name from userinfo where name='dyan' -- n' and password='000'

select name from userinfo where name='badguy' or 1=1 -- y' and password='000'
```

> 这种情况就是由于字符串拼接查询，造成注入，因此我们需要使用`pymysql`提供的参数化查询

```
 effect_row = cursor.execute("select name from userinfo where name=%s and password=%s",
(name,password))
```

这样改完之后，我们再看，会发现报错，但是不会登陆成功了，看看内部执行的语句，主要是对'符号做了转义防止注入

```
select name from userinfo where name=''dyan\' -- n'' and password=''123''
```

使用pymysql操作

- 执行sql

```
import pymysql

# 创建连接
conn = pymysql.connect(host='127.0.0.1', port=3306, user='blog', passwd='123456',
db='blog', charset='utf8')

# 创建游标，查询数据默认为元组类型
```

```
cursor = conn.cursor()


# 执行SQL，并返回收影响行数
row1 = cursor.execute("update users set password = '123'")
print(row1)
# 执行SQL，并返回受影响行数
row2 = cursor.execute("update users set password = '456' where id > %s", (1,))
print(row2)
# 执行SQL，并返回受影响行数（使用pymysql的参数化语句防止SQL注入）
row3 = cursor.executemany("insert into users(username, password, email)values(%s, %s,
%s)", [("ceshi3", '333', 'ceshi3@11.com'), ("ceshi4", '444', 'ceshi4@qq.com')])
print(row3)

# 提交，不然无法保存新建或者修改的数据
conn.commit()
```

- 获取查询数据

```
import pymysql

# 创建连接
conn = pymysql.connect(host='127.0.0.1', port=3306, user='blog', passwd='123456',
db='blog', charset='utf8')

# 创建游标，查询数据默认为元组类型
cursor = conn.cursor()
cursor.execute("select * from users")

# 获取第一行数据
row_1 = cursor.fetchone()
print(row_1)
# 获取前n行数据
row_n = cursor.fetchmany(3)
print(row_n)
```

- 获取新创建数据自增ID

```
import pymysql

# 创建连接
conn = pymysql.connect(host='127.0.0.1', port=3306, user='blog', passwd='123456',
db='blog', charset='utf8')

# 创建游标，查询数据默认为元组类型
cursor = conn.cursor()

cursor.executemany("insert into users(username, password, email)values(%s, %s, %s)",
[("ceshi3", '333', 'ceshi3@11.com'), ("ceshi4", '444', 'ceshi4@qq.com')])
new_id = cursor.lastrowid
print(new_id)
```

```
# 提交，不然无法保存新建或者修改的数据
conn.commit()
# 关闭游标
cursor.close()
conn.close
```

- fetch数据类型 **关于默认获取的数据是元组类型，如果想要或者字典类型的数据，即**

```
import pymysql

# 创建连接
conn = pymysql.connect(host='127.0.0.1', port=3306, user='blog', passwd='123456',
db='blog', charset='utf8')

# 游标设置为字典类型
cursor = conn.cursor(cursor=pymysql.cursors.DictCursor)
# 左连接查询
r = cursor.execute("select * from users as u left join articles as a on u.id =
a.user_id where a.user_id = 2")
result = cursor.fetchall()
print(result)

# 查询一个表的所有字段名
c = cursor.execute("SHOW FULL COLUMNS FROM users FROM blog")
cc = cursor.fetchall()

# 提交，不然无法保存新建或者修改的数据
conn.commit()
# 关闭游标
cursor.close()
# 关闭连接
conn.close()
```

执行结果

```
[{'user_id': 2, 'id': 2, 'password': '456', 'email': 'xinlei2017@test.com', 'a.id': 2,
'content': '成名之路', 'title': '星光大道', 'username': 'tangtang'}]
```

cookie的设置

> Tornado中可以对cookie进行操作，并且还可以对cookie进行签名以放置伪造

- 基本操作

```
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!")
```

- 清除cookie

```
self.clear_cookie(name, path='/', domain=None)
作用:  删除名为name的cookie的值
注意:  执行删除操作之后， 并不是立即清除浏览器端的cookie值， 而是给cookie的值设置为空，并将其有效期改成失效，真正删除cookie是浏览器自己决定的

self.clear_all_cookie(path='/',domain=None)
作用：删除掉path为'/'的所有的cookie的值
```

- 加密cookie（签名）

> `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：

生成秘钥的方法

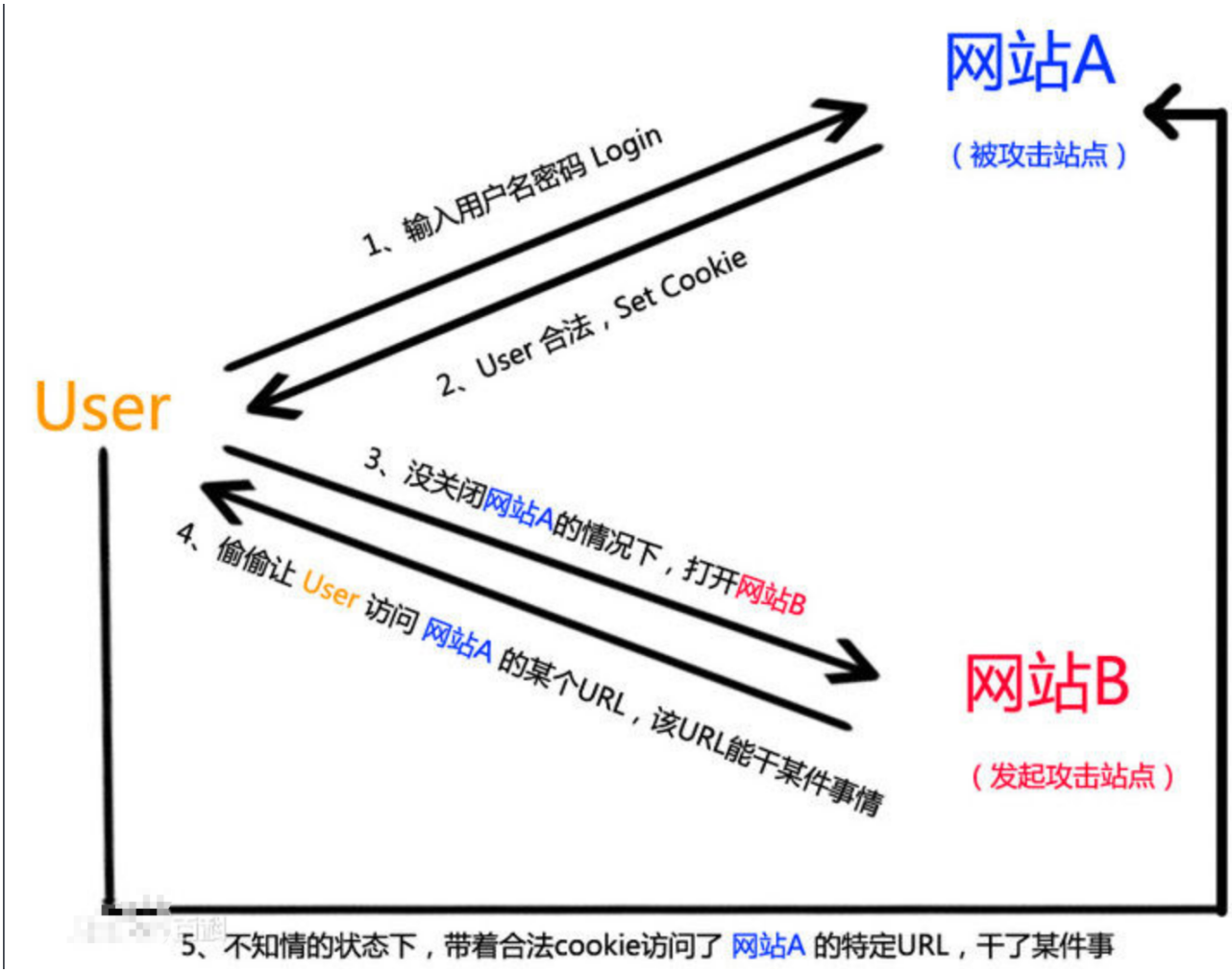
```
import base64
import uuid
base64.b64encode(uuid.uuid4().bytes + uuid.uuid4().bytes)
```

set_secure_cookie: 设置一个带有签名和时间戳的加密cookie值

```
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!")

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

XSRF

> 跨站请求伪造攻击



配置

```
settings = {
    "xsrf_cookies": True,
}

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

使用

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

XSS

> 跨站脚本攻击

XSS是指恶意攻击者利用网站没有对用户提交数据进行转义处理或者过滤不足的缺点，进而添加一些代码，嵌入到web页面中去。使别的用户访问都会执行相应的嵌入代码。

从而盗取用户资料、利用用户身份进行某种动作或者对访问者进行病毒侵害的一种攻击方式。

主要原因：过于信任客户端提交的数据！

解决办法：不信任任何客户端提交的数据，只要是客户端提交的数据就应该先进行相应的过滤处理然后方可进行下一步的操作。

进一步分析细节：

> 客户端提交的数据本来就是应用所需要的，但是恶意攻击者利用网站对客户端提交数据的信任，在数据中插入一些符号以及javascript代码，那么这些数据将会成为应用代码中的一部分了。那么攻击者就可以肆无忌惮地展开攻击啦。 因此我们绝不可以信任任何客户端提交的数据！！！

{% raw username %}

同步

> 按部就班的一步一步的执行

{% fold %}

```
#author:shangzekai
import time

###耗时操作  (操作数据库，操作文件等)
def handlerIo():
    print("耗时操作开始")
    time.sleep(5)
    print("耗时操作结束")

    return 'lxx is a good man '


def reqA():
    print('开始处理A')
    res = handlerIo()
    print('处理A完成', res)


def reqB():
    print("开始处理B")
    time.sleep(2)
    print("处理B完成")

def main():
    reqA()
    reqB()

if __name__ == '__main__':
```

```
    main()
```

{% endfold %}

```
import tornado.httpserver
import tornado.ioloop
import tornado.web
import tornado.httpclient

class MainHandler(tornado.web.RequestHandler):

    def get(self):
        print('开始')
        http_client = tornado.httpclient.HTTPClient()
        response = http_client.fetch("http://www.google.com")
        self.write('done')


def make_app():
    return tornado.web.Application([
        (r"/main", MainHandler)
    ])

if __name__ == "__main__":
    tornado.options.parse_command_line()
    app = tornado.web.Application(handlers=[
        (r"/main", MainHandler)
    ])

    app = make_app()


    http_server = tornado.httpserver.HTTPServer(app)
    http_server.listen(8002)
    tornado.ioloop.IOLoop.instance().start()
```

异步

> 对于耗时的操作， 可以交给别人（另一个线程去做）， 我们继续向下执行，当别人结束耗时操作时， 直接将结果返回即可

实现的方式：

- 回调函数的方式实现异步

```
# author:shangzekai
import time
import threading

###耗时操作 （操作数据库，操作文件等）
def handlerIo(callback):
```

```
    def run(cb):
        print("耗时操作开始")
        time.sleep(2)
        print("耗时操作结束")
        cb('lxx is a good man')
    threading.Thread(target=run, args=callback ).start()

def finish(data):
    print("开始处理回调函数")
    print("接受处理完成的数据", data)
    print("处理回调函数完成")


def reqA():
    print('开始处理A')
    handlerIo(finish)
    print('处理A完成')


def reqB():
    print("开始处理B")
    time.sleep(3)
    print("处理B完成")


def main():
    reqA()
    reqB()


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

Tornado中的异步

> 回调的异步

tornado.httpclient.AsyncHTTPClient

作用： tornado提供的异步web请求客户端，用来进行异步web请求的

方法： fetch(request,callback=None)：用于执行一个web请求，并异步响应返回一个tornado.httpclient.HttpResponse,其中request可以是一个url

还需要在请求的类上添加，装饰器 @tornado.web.asynchronous (不关闭通信的通道)

```
class StudentHandler(RequestHandler):
    def on_response(self, response):
```

```
        if response.error:
            self.send_error(500)
        else:
            data = json.loads(response.body)
            self.write(data)

        self.finish()

    @tornado.web.asynchronous
    def get(self, *args, **kwargs):

        # time.sleep(30)
        from tornado.httpclient import AsyncHTTPClient
        client = AsyncHTTPClient()

        url = "http://v.juhe.cn/weather/index?
format=2&cityname=%E8%8B%8F%E5%B7%9E&key=ac572233358b33198739dc96fbc0ffbf"
        client.fetch(url, self.on_response)
        self.write('student')
```

> 协程的异步

tornado.gen.coroutine

```
@tornado.gen.coroutine
def get(self, *args, **kwargs):

    from tornado.httpclient import AsyncHTTPClient
    url = "http://v.juhe.cn/weather/index?
format=2&cityname=%E8%8B%8F%E5%B7%9E&key=ac572233358b33198739dc96fbc0ffbf"
    client = AsyncHTTPClient()
    res = yield client.fetch(url)

    if res.error:
        self.send_error(500)
    else:
        data = json.loads(res.body)
        self.write(data)
```

Tornado的websocket

WebSocketHandler类

常见的属性和方法:

```
open() : 当一个websocket客户端链接建立后被调用

on_message() : 当客户端发送消息过来时调用

on_close() : 当客户端链接关闭后调用

write_message(): 服务端主动向客户端发送message消息


close() : 关闭socket链接
```

index.py

```
class ChatHandler(RequestHandler):

    def get(self, *args, **kwargs):

        self.render('chat.html')

    users = []
    def open(self):
        self.users.append(self)
        for user in self.users:
            user.write_message('[%s]登陆了'%(self.request.remote_ip))

    def check_origin(self, origin):
        return True

    def on_message(self, message):
        for user in self.users:
            user.write_message('[%s]说: %s'%(self.request.remote_ip, message))

    def on_close(self):
        self.users.remove(self)

        for user in self.users:
            user.write_message('[%s]推出了'%(self.request.remote_ip))
```

chat.html

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


<div id="content" style="height: 500px;width: 300px;">
```

```
</div>

<input type="text" id="msg">
<input type="button" value="发送" onclick="send_msg()">

<script
  src="http://code.jquery.com/jquery-1.12.4.min.js"
  integrity="sha256-ZosEbRLbNQzLpnKIkEdrPv7lOy9C27hHQ+Xp8a4MxAQ="
  crossorigin="anonymous"></script>

<script type="text/javascript">

    var ws = new WebSocket("ws://127.0.0.1:8110/chat");

    ws.onmessage = function (e) {
        $('#content').append("<p>"+e.data+"</p>")
    };

    function send_msg(){
        var msg = $('#msg').val();
        ws.send(msg);
        $('#msg').val("")
    }

</script>




</body>
</html>
```