链接: https://www.cnblogs.com/wupeiqi/p/9078770.html

<https://mp.weixin.qq.com/s?__biz=MzA5ODUzOTA0OQ==&mid=2651689264&idx=1&sn=11fbc3076ad08dc22bf80eb975c8ee14&chksm=8b693276bc1ebb60cdf08d14a5856050edee89406b9f94679c0b78765921cd04b3702cae6381&mpshare=1&scene=1&srcid=0601iRcbOSz3LQTWArSeIWqS#rd>



# 前端和框架

## 谈谈你对http协议的认识。

1.HTTP协议初识

1.1什么是HTTP协议

http协议： 对浏览器客户端 和 服务器端 之间数据传输的格式规范（简单理解）。

1.2查看http协议头的工具

1）使用火狐的firebug插件（右键->firebug->网络）

2）使用谷歌的“审查元素”

3）使用系统自带的telnet工具（远程访问工具）

a）telnet localhost 8080 访问tomcat服务器

b）ctrl+] 回车 可以看到回显

c）输入请求内容

d）回车，即可查看到服务器响应信息。

1.3 http协议内容

请求（浏览器-》服务器）

GET /hello HTTP/1.1

Host: localhost:8080

User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:35.0)

Gecko/20100101 Firefox/35.0

Accept: text/html,application/xhtml+xml,applicatio/xml;q=0.9,\*/\*;q=0.8

Accept-Language: zh-cn,en-us;q=0.8,zh;q=0.5,en;q=0.3

Accept-Encoding: gzip, deflate

响应（服务器-》浏览器）

HTTP/1.1 200 OK

Server: Apache-Coyote/1.1

Content-Length: 24

Date: Fri, 30 Jan 2015 01:54:57 GMT

this is hello servlet!!!

2.Http请求

2.1请求行

①GET ②/hello ③HTTP/1.1

①请求方式

常见的请求方式： GET 、 POST、 HEAD、 TRACE、 PUT、 CONNECT 、DELETE

GET vs POST 区别

GET：/testMethod.html?name=eric&password=123456 HTTP/1.1

-地址栏（URI）会跟上参数数据。以？开头，多个参数之间以&分割。

-GET提交参数数据有限制，不超过1KB。

-GET方式不适合提交敏感密码。

-注意： 浏览器直接访问的请求，默认提交方式是GET方式

POST：

-参数不会跟着URI后面。参数而是跟在请求的实体内容中。没有？开头，多个参数之间以&分割。

-POST提交的参数数据没有限制。

-POST方式适合提交敏感数据。

②请求资源

URL: 统一资源定位符。http://localhost:8080/testImg.html。只能定位互联网资源。是URI的子集。

URI： 统一资源标记符。/hello。用于标记任何资源。可以是本地文件系统，局域网的资源（//192.168.14.10/myweb/index.html），可以是互联网

③http协议版本

http1.0：当前浏览器客户端与服务器端建立连接之后，只能发送一次请求，一次请求之后连接关闭。

http1.1：当前浏览器客户端与服务器端建立连接之后，可以在一次连接中发送多次请求。（基本都使用1.1）

2.2请求头

Accept: text/html,image/\* – 浏览器接受的数据类型

Accept-Charset: ISO-8859-1 – 浏览器接受的编码格式

Accept-Encoding: gzip,compress –浏览器接受的数据压缩格式

Accept-Language: en-us,zh- –浏览器接受的语言

Host: www.it315.org:80 –（必须的）当前请求访问的目标地址（主机:端口）

If-Modified-Since: Tue, 11 Jul 2000 18:23:51 GMT –浏览器最后的缓存时间

Referer: http://www.it315.org/index.jsp – 当前请求来自于哪里

User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0) –浏览器类型

Cookie:name=eric – 浏览器保存的cookie信息

Connection: close/Keep-Alive – 浏览器跟服务器连接状态。close: 连接关闭 keep-alive：保存连接。

Date: Tue, 11 Jul 2000 18:23:51 GMT – 请求发出的时间

2.3 实体内容

只有POST提交的参数会放到实体内容中

2.4 HttpServletRequest对象

HttpServletRequest对象作用是用于获取请求数据。

核心的API：

请求行：

request.getMethod(); 请求方式

request.getRequetURI() / request.getRequetURL() 请求资源

request.getProtocol() 请求http协议版本

请求头：

request.getHeader("名称") 根据请求头获取请求值

request.getHeaderNames() 获取所有的请求头名称

实体内容:

request.getInputStream() 获取实体内容数据

2.5 service 和 doGET、doPOST方法区别

HttpSevlet类的源码：

protected void service(HttpServletRequest req, HttpServletResponse resp)

throws ServletException, IOException {

//得到请求方式

String method = req.getMethod();

if (method.equals(METHOD\_GET)) {

long lastModified = getLastModified(req);

if (lastModified == -1) {

doGet(req, resp);

} else {

long ifModifiedSince;

try {

ifModifiedSince = req.getDateHeader(HEADER\_IFMODSINCE);

} catch (IllegalArgumentException iae) {

ifModifiedSince = -1;

}

if (ifModifiedSince < (lastModified / 1000 \* 1000)) {

maybeSetLastModified(resp, lastModified);

doGet(req, resp);

} else {

resp.setStatus(HttpServletResponse.SC\_NOT\_MODIFIED);

}

}

} else if (method.equals(METHOD\_HEAD)) {

long lastModified = getLastModified(req);

maybeSetLastModified(resp, lastModified);

doHead(req, resp);

} else if (method.equals(METHOD\_POST)) {

doPost(req, resp);

} else if (method.equals(METHOD\_PUT)) {

doPut(req, resp);

} else if (method.equals(METHOD\_DELETE)) {

doDelete(req, resp);

} else if (method.equals(METHOD\_OPTIONS)) {

doOptions(req,resp);

} else if (method.equals(METHOD\_TRACE)) {

doTrace(req,resp);

} else {

String errMsg = lStrings.getString("http.method\_not\_implemented");

Object[] errArgs = new Object[1];

errArgs[0] = method;

errMsg = MessageFormat.format(errMsg, errArgs);

resp.sendError(HttpServletResponse.SC\_NOT\_IMPLEMENTED, errMsg);

}

}

2.6 传递的请求参数如何获取

GET方式： 参数放在URI后面

POST方式： 参数放在实体内容中

获取GET方式参数：

request.getQueryString();

获取POST方式参数：

request.getInputStream();

问题：但是以上两种不通用，而且获取到的参数还需要进一步地解析。

所以可以使用统一方便的获取参数的方式：

核心的API：

request.getParameter("参数名"); 根据参数名获取参数值（注意，只能获取一个值的参数）

request.getParameterValue("参数名“)；根据参数名获取参数值（可以获取多个值的参数）

request.getParameterNames(); 获取所有参数名称列表

2.7 请求参数编码问题

修改POST方式参数编码：

request.setCharacterEncoding("utf-8");

修改GET方式参数编码：

手动解码：String name = new String(name.getBytes("iso-8859-1"),"utf-8");

3 Http响应

HTTP/1.1 ①200 OK –响应行

Server: Apache-Coyote/1.1 –响应头（key-vaule）

Content-Length: 24

Date: Fri, 30 Jan 2015 01:54:57 GMT

–一个空行

this is hello servlet!!! –实体内容

①状态码: 服务器处理请求的结果（状态）

常见的状态：

200 ： 表示请求处理完成并完美返回

302： 表示请求需要进一步细化。

404： 表示客户访问的资源找不到。

500： 表示服务器的资源发送错误。（服务器内部错误）

3.2常见的响应头

Location: http://www.it315.org/index.jsp -表示重定向的地址，该头和302的状态码一起使用。

Server:apache tomcat —表示服务器的类型

Content-Encoding: gzip – 表示服务器发送给浏览器的数据压缩类型

Content-Length: 80 –表示服务器发送给浏览器的数据长度

Content-Language: zh-cn –表示服务器支持的语言

Content-Type: text/html; charset=GB2312 –表示服务器发送给浏览器的数据类型及内容编码

Last-Modified: Tue, 11 Jul 2000 18:23:51 GMT –表示服务器资源的最后修改时间

Refresh: 1;url=http://www.it315.org –表示定时刷新

Content-Disposition: attachment; filename=aaa.zip –表示告诉浏览器以下载方式打开资源（下载文件时用到）

Transfer-Encoding: chunked

Set-Cookie:SS=Q0=5Lb\_nQ; path=/search –表示服务器发送给浏览器的cookie信息（会话管理用到）

Expires: -1 –表示通知浏览器不进行缓存

Cache-Control: no-cache

Pragma: no-cache

Connection: close/Keep-Alive –表示服务器和浏览器的连接状态。close：关闭连接 keep-alive:保存连接

3.3 HttpServletResponse对象

响应行：

response.setStatus() 设置状态码

响应头：

response.setHeader(“name”,”value”) 设置响应头

实体内容：

response.getWriter().writer(); 发送字符实体内容

response.getOutputStream().writer() 发送字节实体内容

## 谈谈你对websocket协议的认识。

　　一、Websocket协议简介

　　Websocket是一个持久化的协议，相对于HTTP这种非持久的协议来说；

　　HTTP协议：

　　HTTP的生命周期通过 Request 来界定，也就是一个 Request 一个 Response ，那么在 HTTP1.0 中，这次HTTP请求就结束了。

　　在HTTP1.1中进行了改进，使得有一个keep-alive，也就是说，在一个HTTP连接中，可以发送多个Request，接收多个Response。但是请记住 Request = Response ， 在HTTP中永远是这样，也就是说一个request只能有一个response。而且这个response也是被动的，不能主动发起。

　　Websocket协议：

　　WebSocket是一种双向通信协议。在建立连接后，WebSocket服务器端和客户端都能主动向对方发送或接收数据，就像Socket一样；

　　WebSocket需要像TCP一样，先建立连接，连接成功后才能相互通信。

　　WebSocket只需要经过一次HTTP请求，就可以做到源源不断的信息传送了（在程序设计中，这种设计叫做回调，即：你有信息了再来通知我，而不是我傻乎乎的每次跑来问你）

## 3、什么是magic string ？

## 4、如何创建响应式布局？

## 5、你曾经使用过哪些前端框架？

## 6、什么是ajax请求？并使用jQuery和XMLHttpRequest对象实现一个ajax请求。

## 7、如何在前端实现轮训？

## 8、如何在前端实现长轮训？

## 9、vuex的作用？

## 10、vue中的路由的拦截器的作用？

## 11、axios的作用？

## 12、列举vue的常见指令。

## 13、简述jsonp及实现原理？

## 14、是什么cors ？

## 16、列举Http请求中的状态码？

## 15、列举Http请求中常见的请求方式？

## 17、列举Http请求中常见的请求头？

## 24、django、flask、tornado框架的比较？

[*Django*](https://link.zhihu.com/?target=https%3A//www.djangoproject.com/)

Django 是一个高级的 Python Web 框架，支持快速开发，简洁、实用的设计。如果你正在建一个和电子商务网站相似的应用，那你应该选择用 Django 框架。它能使你快速完成工作，也不必担心太多的技术选择。它能提供从模版引擎到 ORM 所需的一切东西。用 Django 构建你的app 的时候，你必须要遵循 Django 的方式，这点像极了 Ruby on Rails 的 Rails 框架。有些人会觉得这样有点不爽，但在我看来这是极好的，毕竟我坚信：“约定优于机制”。相对于所有其他的库，Django 框架有最强的社区，这意味着可以轻松获得帮助。

[*Flask*](https://link.zhihu.com/?target=http%3A//flask.pocoo.org/)

Flask 是基于 Werkzeug，Jinja 2 的 Python 轻量级框架（microframework）。注意——“microframework” 里的 “micro” 可能会产生误解。轻量级不意味着 Flask 是一个不成熟、不靠谱的库。它表示 Flask 的核心就是非常非常简单的。不像 Django 框架，它不会给你带来技术选择上的问题，你可以自由选择你喜欢的任何模版引擎或 ORM。即使它默认配备了 Jinja 模板引擎，你也随时可以自由选择。 在我看来，用 Flask 来编写 API 服务（RESTful rervices）是再好不过的。

[*Twisted*](https://link.zhihu.com/?target=https%3A//twistedmatrix.com/trac/)

Twisted 是用 Python 实现的基于事件驱动的网络引擎框架。它是一个高性能的引擎，其快速的主要原因是一个被称为 deferred 的 object，Twisted 是建立在 deferred object 之上。对于不了解 deferred object 的人来说，deferred object 是通过异步架构实现的机制。Twisted 是很快速的，但是不适合编写常规的 WebApps。如果你想做一些底层网络的东西，Twisted 是你的好帮手。

[*Tornado*](https://link.zhihu.com/?target=http%3A//www.tornadoweb.org/en/stable/)

Tornado 是一个 Python Web 框架和异步网络库，最初是由 FriendFeed 开发的。Tornado 采用非阻塞网络 I / O 模型，可以处理数以千计的网络连接，这意味着对于 long polling 、WebSockets 和其他需要长时间实时连接的 apps 来说，Tornado 是一个理想的 Web 框架。Tornado 介于 Django 和 Flask 之间。如果你想要用 Django 或 Flask 写一些东西，但你想要一个更好的性能，你应该选择用 Tornado 框架。配合上合理的架构，它能很好的处理 C10K 问题。

[*Cyclone*](https://link.zhihu.com/?target=http%3A//cyclone.io/)

Cyclone 是用 Python 编写的一款异步非阻塞的轻量级 Web Server 框架。它实现了 [Tornado](https://link.zhihu.com/?target=http%3A//www.oschina.net/p/tornado) 的 API，底层实现是基于 Twisted Protocol 的。现在，如果你想要 Twisted 的性能和易于编写常规的 webapps，那么请选择 Cyclone。相对于 Tornado 框架，我更喜欢 Cyclone。它有一个非常类似于 Tornado 的 API，实际上，它是 Tornado 的一个 fork 分支。但是问题就是它拥有的社区相对较小。Alexandre Fiori 是主要代码贡献者。

[*Pyramid*](https://link.zhihu.com/?target=http%3A//docs.pylonsproject.org/en/latest/docs/pyramid.html)

Pyramid 是一个通用的，开源的 Python web 应用开发框架。其主要目标就是让 Python 开发人员更轻松的开发 web 应用程序。我没有用过 Pyramid 框架，但是我看过它的文档。据我了解，Pyramind 和 Flask 很相似，我认为可以用 Flask 框架的地方也可以用 Pyramid 框架，反之亦然。

## 25、什么是wsgi？

博客：https://www.jianshu.com/p/679dee0a4193

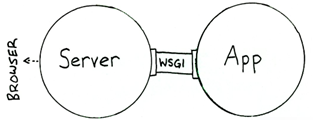
### 25.1 WSGI协议

首先弄清下面几个概念：  
**WSGI：**全称是Web Server Gateway Interface，WSGI不是服务器，python模块，框架，API或者任何软件，只是一种规范，描述web server如何与web application通信的规范。server和application的规范在[PEP 3333](https://link.jianshu.com?t=https:/www.python.org/dev/peps/pep-3333/)中有具体描述。要实现WSGI协议，必须同时实现web server和web application，当前运行在WSGI协议之上的web框架有Bottle, Flask, Django。  
**uwsgi：**与WSGI一样是一种通信协议，是uWSGI服务器的独占协议，用于定义传输信息的类型(type of information)，每一个uwsgi packet前4byte为传输信息类型的描述，与WSGI协议是两种东西，据说该协议是fcgi协议的10倍快。  
**uWSGI：**是一个web服务器，实现了WSGI协议、uwsgi协议、http协议等。

WSGI协议主要包括server和application两部分：

* WSGI server负责从客户端接收请求，将request转发给application，将application返回的response返回给客户端；
* WSGI application接收由server转发的request，处理请求，并将处理结果返回给server。application中可以包括多个栈式的中间件(middlewares)，这些中间件需要同时实现server与application，因此可以在WSGI服务器与WSGI应用之间起调节作用：对服务器来说，中间件扮演应用程序，对应用程序来说，中间件扮演服务器。

WSGI协议其实是定义了一种server与application解耦的规范，即可以有多个实现WSGI server的服务器，也可以有多个实现WSGI application的框架，那么就可以选择任意的server和application组合实现自己的web应用。例如uWSGI和Gunicorn都是实现了WSGI server协议的服务器，Django，Flask是实现了WSGI application协议的web框架，可以根据项目实际情况搭配使用。



wsgi.png-22.9kB

像Django，Flask框架都有自己实现的简单的WSGI server，一般用于服务器调试，生产环境下建议用其他WSGI server。

### 25.2 WSGI协议的实现

以Django为例，分析一下WSGI协议的具体实现过程。

### 25.3 django WSGI application

WSGI application应该实现为一个可调用对象，例如函数、方法、类(包含`**call**`方法)。需要接收两个参数：

* 一个字典，该字典可以包含了客户端请求的信息以及其他信息，可以认为是请求上下文，一般叫做environment（编码中多简写为environ、env）
* 一个用于发送HTTP响应状态（HTTP status ）、响应头（HTTP headers）的回调函数

通过回调函数将响应状态和响应头返回给server，同时返回响应正文(response body)，响应正文是可迭代的、并包含了多个字符串。下面是Django中application的具体实现部分：

class WSGIHandler(base.BaseHandler):

initLock = Lock()

request\_class = WSGIRequest

def \_\_call\_\_(self, environ, start\_response):

# 加载中间件

if self.\_request\_middleware is None:

with self.initLock:

try:

# Check that middleware is still uninitialized.

if self.\_request\_middleware is None:

self.load\_middleware()

except:

# Unload whatever middleware we got

self.\_request\_middleware = None

raise

set\_script\_prefix(get\_script\_name(environ))

# 请求处理之前发送信号

signals.request\_started.send(sender=self.\_\_class\_\_, environ=environ)

try:

request = self.request\_class(environ)

except UnicodeDecodeError:

logger.warning('Bad Request (UnicodeDecodeError)',

exc\_info=sys.exc\_info(),

extra={'status\_code': 400,})

response = http.HttpResponseBadRequest()

else:

response = self.get\_response(request)

response.\_handler\_class = self.\_\_class\_\_

status = '%s %s' % (response.status\_code, response.reason\_phrase)

response\_headers = [(str(k), str(v)) for k, v in response.items()]

for c in response.cookies.values():

response\_headers.append((str('Set-Cookie'), str(c.output(header=''))))

# server提供的回调方法，将响应的header和status返回给server

start\_response(force\_str(status), response\_headers)

if getattr(response, 'file\_to\_stream', None) is not None and environ.get('wsgi.file\_wrapper'):

response = environ['wsgi.file\_wrapper'](response.file\_to\_stream)

return response

可以看出application的流程包括:

* 加载所有中间件，以及执行框架相关的操作，设置当前线程脚本前缀，发送请求开始信号；
* 处理请求，调用get\_response()方法处理当前请求，该方法的的主要逻辑是通过urlconf找到对应的view和callback，按顺序执行各种middleware和callback。
* 调用由server传入的start\_response()方法将响应header与status返回给server。
* 返回响应正文

### 25.4 django WSGI Server

负责获取http请求，将请求传递给WSGI application，由application处理请求后返回response。以Django内建server为例看一下具体实现。  
通过runserver运行django项目，在启动时都会调用下面的run方法，创建一个WSGIServer的实例，之后再调用其serve\_forever()方法启动服务。

def run(addr, port, wsgi\_handler, ipv6=False, threading=False):

server\_address = (addr, port)

if threading:

httpd\_cls = type(str('WSGIServer'), (socketserver.ThreadingMixIn, WSGIServer), {})

else:

httpd\_cls = WSGIServer

# 这里的wsgi\_handler就是WSGIApplication

httpd = httpd\_cls(server\_address, WSGIRequestHandler, ipv6=ipv6)

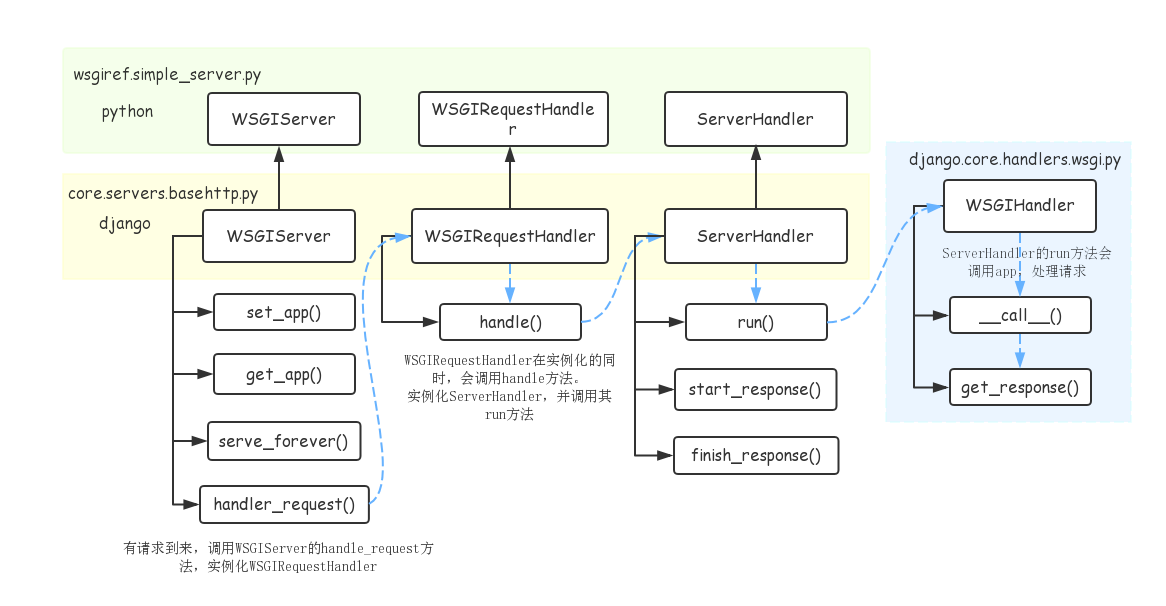
if threading:

httpd.daemon\_threads = True

httpd.set\_app(wsgi\_handler)

httpd.serve\_forever()

下面表示WSGI server服务器处理流程中关键的类和方法。



* **WSGIServer**  
  run()方法会创建WSGIServer实例，主要作用是接收客户端请求，将请求传递给application，然后将application返回的response返回给客户端。
  + 创建实例时会指定HTTP请求的handler：WSGIRequestHandler类
  + 通过set\_app和get\_app方法设置和获取WSGIApplication实例wsgi\_handler
  + 处理http请求时，调用handler\_request方法，会创建WSGIRequestHandler实例处理http请求。
  + WSGIServer中get\_request方法通过socket接受请求数据
* **WSGIRequestHandler**
  + 由WSGIServer在调用handle\_request时创建实例，传入request、cient\_address、WSGIServer三个参数，\_\_init\_\_方法在实例化同时还会调用自身的handle方法
  + handle方法会创建ServerHandler实例，然后调用其run方法处理请求
* **ServerHandler**
  + WSGIRequestHandler在其handle方法中调用run方法，传入self.server.get\_app()参数，获取WSGIApplication，然后调用实例(\_\_call\_\_)，获取response，其中会传入start\_response回调，用来处理返回的header和status。
  + 通过application获取response以后，通过finish\_response返回response
* **WSGIHandler**
  + WSGI协议中的application，接收两个参数，environ字典包含了客户端请求的信息以及其他信息，可以认为是请求上下文，start\_response用于发送返回status和header的回调函数

虽然上面一个WSGI server涉及到多个类实现以及相互引用，但其实原理还是调用WSGIHandler，传入请求参数以及回调方法start\_response()，并将响应返回给客户端。

### 25.5 django simple\_server

django的simple\_server.py模块实现了一个简单的HTTP服务器，并给出了一个简单的demo，可以直接运行，运行结果会将请求中涉及到的环境变量在浏览器中展示出来。  
其中包括上述描述的整个http请求的所有组件:  
ServerHandler, WSGIServer, WSGIRequestHandler，以及demo\_app表示的简易版的WSGIApplication。  
可以看一下整个流程：

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

# 通过make\_server方法创建WSGIServer实例

# 传入建议application，demo\_app

httpd = make\_server('', 8000, demo\_app)

sa = httpd.socket.getsockname()

print("Serving HTTP on", sa[0], "port", sa[1], "...")

import webbrowser

webbrowser.open('http://localhost:8000/xyz?abc')

# 调用WSGIServer的handle\_request方法处理http请求

httpd.handle\_request() # serve one request, then exit

httpd.server\_close()

def make\_server(

host, port, app, server\_class=WSGIServer, handler\_class=WSGIRequestHandler

):

"""Create a new WSGI server listening on `host` and `port` for `app`"""

server = server\_class((host, port), handler\_class)

server.set\_app(app)

return server

# demo\_app可调用对象，接受请求输出结果

def demo\_app(environ,start\_response):

from io import StringIO

stdout = StringIO()

print("Hello world!", file=stdout)

print(file=stdout)

h = sorted(environ.items())

for k,v in h:

print(k,'=',repr(v), file=stdout)

start\_response("200 OK", [('Content-Type','text/plain; charset=utf-8')])

return [stdout.getvalue().encode("utf-8")]

demo\_app()表示一个简单的WSGI application实现，通过make\_server()方法创建一个WSGIServer实例，调用其handle\_request()方法，该方法会调用demo\_app()处理请求，并最终返回响应。

### 25.6 uWSGI

uWSGI旨在为部署分布式集群的网络应用开发一套完整的解决方案。主要面向web及其标准服务。由于其可扩展性，能够被无限制的扩展用来支持更多平台和语言。uWSGI是一个web服务器，实现了WSGI协议，uwsgi协议，http协议等。  
uWSGI的主要特点是：

* 超快的性能
* 低内存占用
* 多app管理
* 详尽的日志功能（可以用来分析app的性能和瓶颈）
* 高度可定制（内存大小限制，服务一定次数后重启等）

uWSGI服务器自己实现了基于uwsgi协议的server部分，我们只需要在uwsgi的配置文件中指定application的地址，uWSGI就能直接和应用框架中的WSGI application通信。

## 26、django请求的生命周期？

转载自<http://www.cnblogs.com/renpingsheng/p/7534897.html>

### 26.1 Django中请求的生命周期

<div class="postBody">

<div id="cnblogs\_post\_body" class="cnblogs-markdown"><h2 id="概述">1. 概述</h2>

首先我们知道HTTP请求及服务端响应中传输的所有数据都是字符串.

在Django中,当我们访问一个的url时,会通过路由匹配进入相应的html网页中.

Django的请求生命周期是指当用户在浏览器上输入url到用户看到网页的这个时间段内,Django后台所发生的事情

而Django的生命周期内到底发生了什么呢??

1. 当用户在浏览器中输入url时,浏览器会生成请求头和请求体发给服务端

请求头和请求体中会包含浏览器的动作(action),这个动作通常为get或者post,体现在url之中.

2. url经过Django中的wsgi,再经过Django的中间件,最后url到过路由映射表,在路由中一条一条进行匹配,

一旦其中一条匹配成功就执行对应的视图函数,后面的路由就不再继续匹配了.

3. 视图函数根据客户端的请求查询相应的数据.返回给Django,然后Django把客户端想要的数据做为一个字符串返回给客户端.

4. 客户端浏览器接收到返回的数据,经过渲染后显示给用户.

视图函数根据客户端的请求查询相应的数据后.如果同时有多个客户端同时发送不同的url到服务端请求数据

服务端查询到数据后,怎么知道要把哪些数据返回给哪个客户端呢??

因此客户端发到服务端的url中还必须要包含所要请求的数据信息等内容.

例如,http://www.aaa.com/index/?nid=user这个url中,  
客户端通过get请求向服务端发送的nid=user的请求,服务端可以通过request.GET.get("nid")的方式取得nid数据

客户端还可以通过post的方式向服务端请求数据.

当客户端以post的方式向服务端请求数据的时候,请求的数据包含在请求体里,这时服务端就使用request.POST的方式取得客户端想要取得的数据

需要注意的是,request.POST是把请求体的数据转换一个字典,请求体中的数据默认是以字符串的形式存在的.

2. FBV模式和CBV模式

一个url对应一个视图函数,这个模式叫做FBV(Function Base Views)

除了FBV之处,Django中还有另外一种模式叫做CBV(Class Base views),即一个url对应一个类

例子:使用cbv模式来请求网页

路由信息:

urlpatterns = [

url(r'^fbv/',views.fbv),

url(r'^cbv/',views.CBV.as\_view()),

]

视图函数配置:

from django.views import View

class CBV(View):

def get(self,request):

return render(request, "cbv.html")

def post(self,request):

return HttpResponse("cbv.get")

cbv.html网页的内容:

<body>

<form method="post" action="/cbv/">

{% csrf\_token %}

<input type="text">

<input type="submit">

</form>

</body>

启动项目,在浏览器中输入http://127.0.0.1:8000/cbv/,回车,得到的网页如下:



在input框中输入”hello”,后回车,得到的网页如下:



使用fbv的模式,在url匹配成功之后,会直接执行对应的视图函数.

而如果使用cbv模式,在url匹配成功之后,会找到视图函数中对应的类,然后这个类回到请求头中找到对应的Request Method.

如果是客户端以post的方式提交请求,就执行类中的post方法;

如果是客户端以get的方式提交请求,就执行类中的get方法

然后查找用户发过来的url,然后在类中执行对应的方法查询生成用户需要的数据.

### 26.2 fbv方式请求的过程

用户发送url请求,Django会依次遍历路由映射表中的所有记录,一旦路由映射表其中的一条匹配成功了,  
就执行视图函数中对应的函数名,这是fbv的执行流程

### 26.3 cbv方式请求的过程

当服务端使用cbv模式的时候,用户发给服务端的请求包含url和method,这两个信息都是字符串类型

服务端通过路由映射表匹配成功后会自动去找dispatch方法,然后Django会通过dispatch反射的方式找到类中对应的方法并执行

类中的方法执行完毕之后,会把客户端想要的数据返回给dispatch方法,由dispatch方法把数据返回经客户端

例子,把上面的例子中的视图函数修改成如下:

from django.views import View

class CBV(View):

def dispatch(self, request, \*args, \*\*kwargs):

print("dispatch......")

res=super(CBV,self).dispatch(request,\*args,\*\*kwargs)

return res

def get(self,request):

return render(request, "cbv.html")

def post(self,request):

return HttpResponse("cbv.get")

打印结果:

<HttpResponse status\_code=200, "text/html; charset=utf-8">

dispatch......

<HttpResponse status\_code=200, "text/html; charset=utf-8">

* 1
* 2
* 3

需要注意的是:

以get方式请求数据时,请求头里有信息,请求体里没有数据

以post请求数据时,请求头和请求体里都有数据.

* 1

### 26.4 Django请求生命周期之响应内容

http提交数据的方式有"post","get","put","patch","delete","head","options","trace".

提交数据的时候,服务端依据method的不同会触发不同的视图函数.

对于from表单来说,提交数据只有get和post两种方法

* 1

另外的方法可以通过Ajax方法来提交

服务端根据个人请求信息的不同来操作数据库,可以使用原生的SQL语句,也可以使用Django的ORM语句.

Django从数据库中查询处理完用户想要的数据,将结果返回给用户.

从Django中返回的响应内容包含响应头和响应体

在Django中,有的时候一个视图函数,执行完成后会使用HttpResponse来返回一个字符串给客户端.  
这个字符串只是响应体的部分,返回给客户端的响应头的部分应该怎么设置呢???

为返回给客户端的信息加一个响应头:

修改上面例子的视图函数为如下:

from django.views import View

class CBV(View):

def dispatch(self, request, \*args, \*\*kwargs):

print("dispatch......")

res=super(CBV,self).dispatch(request,\*args,\*\*kwargs)

print(res)

return res

def get(self,request):

return render(request, "cbv.html")

def post(self,request):

res=HttpResponse("cbv.post")

res.set\_cookie("k2","v2")

res.set\_cookie("k4","v4")

print("res:",res)

print("request.cookie:",request.COOKIES)

return res

打印的信息:

res: <HttpResponse status\_code=200, "text/html; charset=utf-8">

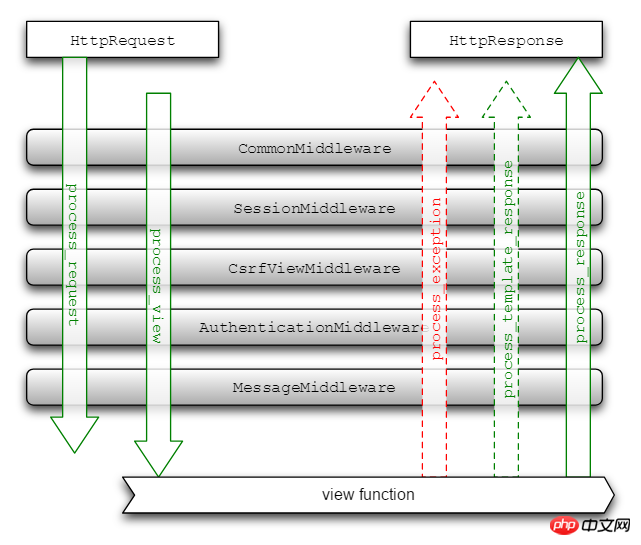
request.cookie: {'csrftoken': 'jmX9H1455MYzDRQs8cQLrA23K0aCGoHpINL50GnMVxhUjamI8wgm

## 27、列举django的内置组件？

## 28、列举django中间件的5个方法？以及django中间件的应用场景？

一、Django中间件的请求周期

我们从浏览器发出一个请求 Request，得到一个响应后的内容 HttpResponse ，这个请求传递到 Django的过程如下：



也就是说，每一个请求都是先通过中间件中的 process\_request 函数，这个函数返回 None 或者 HttpResponse 对象，如果返回前者，继续处理其它中间件，如果返回一个 HttpResponse，就处理中止，返回到网页上。

## 29、简述什么是FBV和CBV？

博客：<http://www.cnblogs.com/yangxt90/articles/8745088.html>

**Django的FBV和CBV**

**FBV**

**FBV（function base views）** 就是在视图里使用函数处理请求。

在之前django的学习中，我们一直使用的是这种方式，所以不再赘述。

**CBV**

**CBV（class base views）** 就是在视图里使用类处理请求。

Python是一个面向对象的编程语言，如果只用函数来开发，有很多面向对象的优点就错失了（继承、封装、多态）。所以Django在后来加入了Class-Based-View。可以让我们用类写View。这样做的优点主要下面两种：

1. 提高了代码的复用性，可以使用面向对象的技术，比如Mixin（多继承）
2. 可以用不同的函数针对不同的HTTP方法处理，而不是通过很多if判断，提高代码可读性

**使用class-based views**

如果我们要写一个处理GET方法的view，用函数写的话是下面这样

from django.http import HttpResponse

def my\_view(request):

if request.method == 'GET':

return HttpResponse('OK')

如果用class-based view写的话，就是下面这样

[复制代码](javascript:void(0);)

from django.http import HttpResponse

from django.views import View

class MyView(View):

def get(self, request):

return HttpResponse('OK')

[复制代码](javascript:void(0);)

Django的url是将一个请求分配给可调用的函数的，而不是一个class。针对这个问题，class-based view提供了一个as\_view()静态方法（也就是类方法），调用这个方法，会创建一个类的实例，然后通过实例调用dispatch()方法，dispatch()方法会根据request的method的不同调用相应的方法来处理request（如get() , post()等）。到这里，这些方法和function-based view差不多了，要接收request，得到一个response返回。如果方法没有定义，会抛出HttpResponseNotAllowed异常。

在url中，就这么写：

[复制代码](javascript:void(0);)

# urls.py

from django.conf.urls import url

from myapp.views import MyView

urlpatterns = [

url(r'^index/$', MyView.as\_view()),

]

[复制代码](javascript:void(0);)

我们可以看看as\_view这个方法的源码

[复制代码](javascript:void(0);)

@classonlymethod

def as\_view(cls, \*\*initkwargs):

"""

Main entry point for a request-response process.

"""

for key in initkwargs:

if key in cls.http\_method\_names:

raise TypeError("You tried to pass in the %s method name as a "

"keyword argument to %s(). Don't do that."

% (key, cls.\_\_name\_\_))

if not hasattr(cls, key):

raise TypeError("%s() received an invalid keyword %r. as\_view "

"only accepts arguments that are already "

"attributes of the class." % (cls.\_\_name\_\_, key))

def view(request, \*args, \*\*kwargs):

self = cls(\*\*initkwargs)

if hasattr(self, 'get') and not hasattr(self, 'head'):

self.head = self.get

self.request = request

self.args = args

self.kwargs = kwargs

return self.dispatch(request, \*args, \*\*kwargs)

view.view\_class = cls

view.view\_initkwargs = initkwargs

# take name and docstring from class

update\_wrapper(view, cls, updated=())

# and possible attributes set by decorators

# like csrf\_exempt from dispatch

update\_wrapper(view, cls.dispatch, assigned=())

return view

[复制代码](javascript:void(0);)

可以看到as\_view最终的执行结果就是返回了一个view函数，而在url中其实我们就是在调用这个函数，这个函数先是实例化出了一个View类的对象，最后返回的是这个对象的dispatch方法的执行结果

那么这个dispatche方法又干了什么呢

[复制代码](javascript:void(0);)

def dispatch(self, request, \*args, \*\*kwargs):

# Try to dispatch to the right method; if a method doesn't exist,

# defer to the error handler. Also defer to the error handler if the

# request method isn't on the approved list.

if request.method.lower() in self.http\_method\_names:

handler = getattr(self, request.method.lower(), self.http\_method\_not\_allowed)

else:

handler = self.http\_method\_not\_allowed

return handler(request, \*args, \*\*kwargs)

[复制代码](javascript:void(0);)

这个方法其实就是判断我们的请求方法，并根据请求方法执行相应的方法对应的函数

类的属性可以通过两种方法设置，第一种是常见的Python的方法，可以被子类覆盖

[复制代码](javascript:void(0);)

from django.http import HttpResponse

from django.views import View

class GreetingView(View):

name = "yuan"

def get(self, request):

return HttpResponse(self.name)

# You can override that in a subclass

class MorningGreetingView(GreetingView):

name= "alex"

[复制代码](javascript:void(0);)

第二种方法，你也可以在url中指定类的属性：

在url中设置类的属性Python

urlpatterns = [

url(r'^index/$', GreetingView.as\_view(name="egon")),

]

**使用Mixin**

我觉得要理解django的class-based-view（以下简称cbv），首先要明白django引入cbv的目的是什么。在django1.3之前，generic view也就是所谓的通用视图，使用的是function-based-view（fbv），亦即基于函数的视图。有人认为fbv比cbv更pythonic，窃以为不然。python的一大重要的特性就是面向对象。而cbv更能体现python的面向对象。cbv是通过class的方式来实现视图方法的。class相对于function，更能利用多态的特定，因此更容易从宏观层面上将项目内的比较通用的功能抽象出来。关于多态，不多解释，有兴趣的同学自己Google。总之可以理解为一个东西具有多种形态（的特性）。cbv的实现原理通过看django的源码就很容易明白，大体就是由url路由到这个cbv之后，通过cbv内部的dispatch方法进行分发，将get请求分发给cbv.get方法处理，将post请求分发给cbv.post方法处理，其他方法类似。怎么利用多态呢？cbv里引入了mixin的概念。Mixin就是写好了的一些基础类，然后通过不同的Mixin组合成为最终想要的类。

所以，理解cbv的基础是，理解Mixin。Django中使用Mixin来重用代码，一个View Class可以继承多个Mixin，但是只能继承一个View（包括View的子类），推荐把View写在最右边，多个Mixin写在左边。

**关于csrf\_token的装饰器**

我们知道当我们向django发送post请求时，有一个中间件会检验csrf\_token，如果我们不想使用它可以将它注释，同样我们也可以通过装饰器来避免发送POST请求时被服务器拒绝

[复制代码](javascript:void(0);)

from django.shortcuts import render, HttpResponse

from django.views import View

# Create your views here.

from django.views.decorators.csrf import csrf\_exempt, csrf\_protect

from django.utils.decorators import method\_decorator

@csrf\_exempt # 避免csrf验证

def foo(request):

return HttpResponse("foo")

# 方式1

# @method\_decorator(csrf\_exempt, name="dispatch")

class IndexView(View):

# 方式2

@method\_decorator(csrf\_exempt)

def dispatch(self, request, \*args, \*\*kwargs):

print("hello world")

# 执行父类的dispatch方法

res = super(IndexView, self).dispatch(request, \*args, \*\*kwargs)

return res

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

return HttpResponse("index")

def post(self, request, \*args, \*\*kwargs):

return HttpResponse("post index")

def delete(self, request):

return HttpResponse("delete index")

[复制代码](javascript:void(0);)

可以看到FBV和CBV的形式都可以通过装饰器的形式来实现，还有一个csrf\_protect是可以在中间件被注释时也可以进行验证

## 30、django的request对象是在什么时候创建的？

Request

　　我们知道当URLconf文件匹配到用户输入的路径后，会调用对应的view函数，并将  **HttpRequest对象**  作为第一个参数传入该函数。

　　我们来看一看这个HttpRequest对象有哪些属性或者方法：

**属性：**

1  HttpRequest.scheme 　     请求的协议，一般为http或者https，字符串格式(以下属性中若无特殊指明，均为字符串格式)

2  HttpRequest.body  　　    http请求的主体，二进制格式。

3  HttpRequest.path             所请求页面的完整路径(但不包括协议以及域名)，也就是相对于网站根目录的路径。

4  HttpRequest.path\_info     获取具有 URL 扩展名的资源的附加路径信息。相对于HttpRequest.path，使用该方法便于移植。

if the WSGIScriptAlias for your application is set to "/minfo", then path might be "/minfo/music/bands/the\_beatles/" and path\_info would be "/music/bands/the\_beatles/".

5  HttpRequest.method               获取该请求的方法，比如： GET   POST .........

6  HttpRequest.encoding             获取请求中表单提交数据的编码。

7  HttpRequest.content\_type      获取请求的MIME类型(从CONTENT\_TYPE头部中获取)，django1.10的新特性。

8  HttpRequest.content\_params  获取CONTENT\_TYPE中的键值对参数，并以字典的方式表示，django1.10的新特性。

9  HttpRequest.GET                    返回一个 **querydict**对象(类似于字典，本文最后有querydict的介绍)，该对象包含了所有的HTTP GET参数

10  HttpRequest.POST                返回一个 **querydict**，该对象包含了所有的HTTP POST参数，通过表单上传的所有  **字符**  都会保存在该属性中。

11  HttpRequest.COOKIES  　     返回一个包含了所有cookies的**字典**。

12  HttpRequest.FILES  　　       返回一个包含了所有的上传文件的  **querydict**  对象。通过表单所上传的所有  **文件**  都会保存在该属性中。

　　                                             key的值是input标签中name属性的值，value的值是一个UploadedFile对象

13  HttpRequest.META                返回一个包含了所有http头部信息的字典

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

14  HttpRequest.session       中间件属性

15  HttpRequest.site　　      中间件属性

16  HttpRequest.user　　     中间件属性，表示当前登录的用户。

　　 HttpRequest.user实际上是由一个定义在django.contrib.auth.models 中的  **user model**  类  所创建的对象。

　　 该类有许多字段，属性和方法。列举几个常用的：        获取更详细信息-->[官方文档](https://docs.djangoproject.com/en/1.10/ref/contrib/auth/#django.contrib.auth.models.User.is_authenticated)。

　　　　1  字段：

**username**    用户名

　　　　　　first\_name  

　　　　　　last\_name 

　　　　　　email

　　　　　　password   

　　　　　　groups

　　　　　　user\_permissions,

　　　　　　is\_staff     布尔值，标明用户是否可以访问admin页面

　　　　　　is\_superuser

**last\_login**  上一次登陆时间

**date\_joined**     用户创建时间

　　　　2  属性  

　　　　　　is\_authenticated   布尔值，标志着用户是否已认证。在django1.10之前，没有该属性，但有与该属性同名的方法。

　　　　3  方法

　　　　　　1  HttpRequest.user.get\_username()  注意：方法的圆括号在templates标签中必需省略！！

　　　　　　　　　获取username。尽量使用该方法来代替使用username字段

　　　　　　2  HttpRequest.user.get\_full\_name()  注意：方法的圆括号在templates标签中必需省略！！

　　　　　　　　　获取first\_name和last\_name

　　　　　　3  HttpRequest.user.short\_name()  注意：方法的圆括号在templates标签中必需省略！！

　　　　　　　　　获取first\_name

　　　　　　4  HttpRequest.user.set\_password(raw\_password)  注意：该方法无法在template标签中使用！！

　　　　　　　　　设置密码

　　　　　　5  HttpRequest.user.check\_password(raw\_password)  注意：该方法无法在template标签中使用！！

　　　　　　　　　如果raw\_password与用户密码相等，则返回True

**方法：**

1  HttpRequest.get\_host()            返回请求的源主机。example:  127.0.0.1:8000

2  HttpRequest.get\_port()            django1.9的新特性。

3  HttpRequest.get\_full\_path()     返回完整路径，并包括附加的查询信息。example:  "/music/bands/the\_beatles/?print=true"

4  HttpRequest.bulid\_absolute\_uri(location)      返回location的绝对uri，location默认为request.get\_full\_path()。

　　  Example: "https://example.com/music/bands/the\_beatles/?print=true"

*QueryDict*

是一个类似于Python中字典的一种对象，他是Python中字典的子类，所以继承了字典的所有方法，

　　当然QueryDict对字典的某些方法进行了加工，并补充了一些独特的方法。这里列出部分方法。详情请看： [官方文档](https://docs.djangoproject.com/en/1.10/ref/request-response/)*。*

1  QueryDict.get(key,default=None)   返回key所对应的value，若key不存在，则返回default的值

2  QueryDict.update(other\_dict)   更新

3  QueryDict.values()   列出所有的值

4  QueryDict.items()   列出所有的键值对,若一个key有多个值，只显示最后一个值。

5  QueryDict.pop(key)   删除某个键值对

6  QueryDict.getlist(key)   根据输入的key返回一个Python中的list

7  QueryDict.dict()   返回QueryDict的字典的表现形式

## 31、如何给CBV的程序添加装饰器？

## 32、列举django orm 中所有的方法（QuerySet对象的所有方法）

## 33、only和defer的区别？

## 34、select\_related和prefetch\_related的区别？

## 35、filter和exclude的区别？

## 36、列举django orm中三种能写sql语句的方法。

## 37、django orm 中如何设置读写分离？

## 38、F和Q的作用?

## 39、values和values\_list的区别？

## 40、如何使用django orm批量创建数据？

## 41、django的Form和ModeForm的作用？

## 42、django的Form组件中，如果字段中包含choices参数，请使用两种方式实现数据源实时更新。

## 43、django的Model中的ForeignKey字段中的on\_delete参数有什么作用？

## 44、django中csrf的实现机制？

## 45、django如何实现websocket？

http://www.cnblogs.com/MnCu8261/tag/python/default.html?page=2

## 46、基于django使用ajax发送post请求时，都可以使用哪种方法携带csrf token？

## 47、django中如何实现orm表中添加数据时创建一条日志记录。

## 48、django缓存如何设置？

## 49、django的缓存能使用redis吗？如果可以的话，如何配置？

## 50、django路由系统中name的作用？

## 51、django的模板中filter和simple\_tag的区别？

## 52、django-debug-toolbar的作用？

## 53、django中如何实现单元测试？

## 54、解释orm中 db first 和 code first的含义？

## 55、django中如何根据数据库表生成model中的类？

## 56、使用orm和原生sql的优缺点？

## 57、简述MVC和MTV

## 58、django的contenttype组件的作用？

## 59、谈谈你对restfull 规范的认识？

## 60、接口的幂等性是什么意思？

## 61、什么是RPC？

## 62、Http和Https的区别？

## 63、为什么要使用django rest framework框架？

## 64、django rest framework框架中都有那些组件？

## 65、django rest framework框架中的视图都可以继承哪些类？

## 66、简述 django rest framework框架的认证流程。

## 67、django rest framework如何实现的用户访问频率控制？

## 68、Flask框架的优势？

## 69、Flask框架依赖组件？

## 70、Flask蓝图的作用？

## 71、列举使用过的Flask第三方组件？

## 72、简述Flask上下文管理流程?

## 73、Flask中的g的作用？

## 74、Flask中上下文管理主要涉及到了那些相关的类？并描述类主要作用？

## 75、为什么要Flask把Local对象中的的值stack 维护成一个列表？

## 76、Flask中多app应用是怎么完成？

## 77、在Flask中实现WebSocket需要什么组件？

## 78、wtforms组件的作用？

## 79、Flask框架默认session处理机制？

## 80、解释Flask框架中的Local对象和threading.local对象的区别？

## 81、Flask中 blinker 是什么？

## 82、SQLAlchemy中的 session和scoped\_session 的区别？

## 83、SQLAlchemy如何执行原生SQL？

## 84、ORM的实现原理？

## 85、DBUtils模块的作用？

## 86、以下SQLAlchemy的字段是否正确？如果不正确请更正：

## 87、SQLAchemy中如何为表设置引擎和字符编码？

## 88、SQLAchemy中如何设置联合唯一索引？

## 89、简述Tornado框架的特点。

## 90、简述Tornado框架中Future对象的作用？

## 91、Tornado框架中如何编写WebSocket程序？

## 92、Tornado中静态文件是如何处理的？如： <link href="{{static\_url("commons.css")}}" rel="stylesheet" />

## 93、Tornado操作MySQL使用的模块？

## 94、Tornado操作redis使用的模块？

## 95、简述Tornado框架的适用场景？

## 96、git常见命令作用：

## 97、简述以下git中stash命令作用以及相关其他命令。

## 98、git 中 merge 和 rebase命令 的区别。

## 99、公司如何基于git做的协同开发？

## 100、如何基于git实现代码review？

## 101、git如何实现v1.0 、v2.0 等版本的管理？

## 102、什么是gitlab？

## 103、github和gitlab的区别？

https://blog.csdn.net/admin\_root1/article/details/78911732

## 104、如何为github上牛逼的开源项目贡献代码？

## 105、git中 .gitignore文件的作用?

在Studio里使用Git管理代码的过程中，可以修改.gitignore文件中的标示的方法来忽略开发者想忽略掉的文件或目录，如果没有.gitignore文件，可以自己手工创建。在.gitignore文件中的每一行保存一个匹配的规则例如：

## 106、什么是敏捷开发？

## 107、简述 jenkins 工具的作用?

Jenkins ，之前叫做Hudson，是基于Java开发的一种开源持续集成工具，可以监控秩序重复的工作，提供了大量的plugin满足各式的集成需求，由于我们以前每天频繁手动部署各种项目到各台windows机器上比较浪费时间，现在的话，有了jenkins，你可以一键部署，很快，很方便···

——-好记性不如烂笔头——

## 108、公司如何实现代码发布？

## 109、简述 RabbitMQ、Kafka、ZeroMQ的区别？

## 110、RabbitMQ如何在消费者获取任务后未处理完前就挂掉时，保证数据不丢失？

## 111、RabbitMQ如何对消息做持久化？

## 112、RabbitMQ如何控制消息被消费的顺序？

## 113、以下RabbitMQ的exchange type分别代表什么意思？如：fanout、direct、topic。

## 114、简述 celery 是什么以及应用场景？

## 115、简述celery运行机制。

## 116、celery如何实现定时任务？

## 117、简述 celery多任务结构目录？

## 118、celery中装饰器 @app.task 和 @shared\_task的区别？

## 119、简述 requests模块的作用及基本使用？

## 120、简述 beautifulsoup模块的作用及基本使用？

## 121、简述 seleninu模块的作用及基本使用?

## 122、scrapy框架中各组件的工作流程？

## 123、在scrapy框架中如何设置代理（两种方法）？

## 124、scrapy框架中如何实现大文件的下载？

## 125、scrapy中如何实现限速？

## 126、scrapy中如何实现暂定爬虫？

## 127、scrapy中如何进行自定制命令？

## 128、scrapy中如何实现的记录爬虫的深度？

## 129、scrapy中的pipelines工作原理？

## 130、scrapy的pipelines如何丢弃一个item对象？

## 131、简述scrapy中爬虫中间件和下载中间件的作用？

## 132、scrapy-redis组件的作用？

## 133、scrapy-redis组件中如何实现的任务的去重？

## 134、scrapy-redis的调度器如何实现任务的深度优先和广度优先？

## 135、简述 vitualenv 及应用场景?

## 136、简述 pipreqs 及应用场景？

## 137、在Python中使用过什么代码检查工具？

## 138、简述 saltstack、ansible、fabric、puppet工具的作用？

## 139、B Tree和B+ Tree的区别？

## 140、请列举常见排序并通过代码实现任意三种。

## 141、请列举常见查找并通过代码实现任意三种。

## 142、请列举你熟悉的设计模式？

## 143、有没有刷过leetcode？

## 144、列举熟悉的的Linux命令。

## 145、公司线上服务器是什么系统？

## 146、解释 PV、UV 的含义？

## 147、解释 QPS的含义？

## 148、uwsgi和wsgi的区别？

## 149、supervisor的作用？

## 150、什么是反向代理？

## 151、简述SSH的整个过程。

## 152、有问题都去那些找解决方案？

## 153、是否有关注什么技术类的公众号？

## 154、最近在研究什么新技术？

## 155、是否了解过领域驱动模型？

统计：80 + 34 + 46 + 155 = 315题