**什么是爬虫**

**爬虫技术**

* 网络爬虫（Web crawler），是一种按照一定的规则，自动地抓取万维网信息的程序或者脚本。
* 爬虫一般分为数据采集，处理，储存三个部分。
* 传统爬虫从一个或若干初始网页的URL开始，获得初始网页上的URL，在抓取网页的过程中，不断从当前页面上抽取新的URL放入队列,直到满足系统的一定停止条件。
* 爬虫脚本一般属于垂直爬虫领域。

**Python爬虫工具与关键包**

* 基础包：
  + requests:一个方便、简洁、高效且人性化的HTTP请求库
  + BeautifulSoup:HTML解析库
  + pymongo:MongoDB的Python封装模块
  + selenium:一个Web自动化测试框架，用于模拟登录和获取JS动态数据
  + pytesseract:一个OCR识别模块，用于验证码识别
  + Pillow:Python图像处理模块
  + python-redis.cli
* 分布式爬虫框架
  + Scrapy
* 分布式缓存数据库
  + Redis
  + MongoDB

**网页数据爬取**

**数据API技术**

**APP数据获取**

**数据隐私**

* 大数据环境中用户的数据隐私即秘密数据，是不想被他人获知的信息。从隐私所有者的角度，可将隐私数据分为个人隐私数据和共同隐私数据，个人隐私数据包括可用来识别或定位个人的信息(如电话号码、地址、信用卡号、认证信息等)和敏感的信息(如个人的健康状况、财务信息、历史访问记录、公司的重要文件等)

# 模块http

http模块实际上是一个package，包含如下模块：

|- http.client：HTTP协议的一个底层封装；

|- http.server： 使用socketserver实现的一个基本HTTP服务器类；

|- http.cookies： 实现状态管理的一个Cookies工具；

|- http.cookiejar：提供Cookie的持久化管理；

http模块提供一个HTTPStatus类

## http模块API结构

### http模块

In [ ]:

**import** **http**

dir(http)

*# dir(http.IntEnum.\_\_members\_\_)*

*# dir(http.HTTPStatus)*

*# dir(http.HTTPStatus.\_\_members\_\_)*

*# 可以使用help获取更加详细的帮助信息*

### HTTPStatus枚举类型使用

1. 类型查看

In [ ]:

**import** **http**

*# 枚举类型*

print(http.HTTPStatus)

1. 枚举类型的值

In [ ]:

**import** **http**

*# 枚举类型访问*

print(http.HTTPStatus.OK)

*# 类型*

print(type(http.HTTPStatus.OK))

*# 枚举类型的帮助*

print(dir(http.HTTPStatus.OK ))

*# 枚举类型的值*

print(http.HTTPStatus.OK.value)

1. 枚举类型值的判定

In [ ]:

**import** **http**

*# 判定枚举类型*

**if** http.HTTPStatus.OK == 200:

print('200响应码')

1. 枚举类型转换为列表使用

In [ ]:

**import** **http**

*# 把枚举类型转换成list (list的每个元素还是枚举类型)*

list\_status = list(http.HTTPStatus)

print(list\_status)

*# 把枚举转换成list使用*

**if** 200 **in** list(http.HTTPStatus):

print('200存在')

1. HTTPStatus的**members**

In [ ]:

**import** **http**

*# 把枚举类型当成字典使用*

*# print(http.HTTPStatus.\_\_members\_\_)*

*# 遍历字典*

*# dir(http.HTTPStatus.\_\_members\_\_)*

*# http.HTTPStatus.\_\_members\_\_可以当成字典使用*

print(http.HTTPStatus.\_\_members\_\_['OK']) *# 使用字段访问枚举类型*

print(http.HTTPStatus.\_\_members\_\_['OK'].value) *# 取枚举类型的值*

*# 典型的字典访问*

*# for k,v in http.HTTPStatus.\_\_members\_\_.items():*

*# print(k,":",v)*

*# 字典的keys访问*

*# print('键访问：', http.HTTPStatus.\_\_members\_\_.keys())*

*# 字典的values访问*

*# print('值访问：', http.HTTPStatus.\_\_members\_\_.values())*

### http.client模块

#### http.client的API结构

In [ ]:

**import** **http.client**

*# help在控制台下查看，效果更加好些。*

*# help 显示整个API的使用*

*# help(http.client)*

*# dir显示成员*

print(dir(http.client))

*# http.client也引入了http.HTTPStatus的枚举类型*

print(type(http.client.ACCEPTED))

1. http.client的核心模块就两个：
2. |- HTTPConnection 或者 HTTPSConnection：表示请求发起（包含网络连接基础工作），请求发起包含请求协议。
3. |- HTTPResponse：表示请求成功后返回的HTTP响应
4. |- 其中响应头使用一个类封装 HTTPMessage

其他基本上是异常结构设计（这里也可以看出异常结构设计是多么重要）

1. HTTPConnection API
2. 继承结构
3. builtins.object
4. |- HTTPConnection

|- HTTPSConnection

In [ ]:

**import** **http.client**

*# help(http.client.HTTPConnection)*

print(dir(http.client.HTTPConnection))

1. HTTPResponse API
2. 继承结构
3. io.BufferedIOBase(\_io.\_BufferedIOBase, io.IOBase)

|- HTTPResponse

In [ ]:

**import** **http.client**

*# help(http.client.HTTPConnection)*

print(dir(http.client.HTTPResponse))

1. 异常结构
2. |- builtins.ConnectionResetError(builtins.ConnectionError)
3. | |- RemoteDisconnected(builtins.ConnectionResetError, BadStatusLine)
4. |
5. |
6. |- builtins.Exception(builtins.BaseException)
7. |- HTTPException
8. |- BadStatusLine
9. | |- RemoteDisconnected(builtins.ConnectionResetError, BadStatusLine)
10. |- ImproperConnectionState
11. | |- CannotSendHeader
12. | |- CannotSendRequest
13. | |- ResponseNotReady
14. |- IncompleteRead
15. |- InvalidURL
16. |- LineTooLong
17. |- NotConnected
18. |- UnimplementedFileMode
19. |- UnknownProtocol

|- UnknownTransferEncoding

#### HTTPConnection的调用模式

1. 构造HTTPConnection()对象；

2. 使用connect()连接到服务器；

3. 使用putrequest()发起一个请求；

4. 使用putheader()设置请求头；

5. 使用endheaders()结束请求头设置，并设置请求体（如果有的话）；

6. 使用HTTPConnection.send(data)设置请求数据体；

7. 使用response = getresponse()获取服务器响应对象HTTPResponse；

8. 使用HTTPResponse对象获取响应信息；

如果使用快捷方式：

HTTPConnection.request()完成使用putrequest()，putheader()，endheaders()，send(data)的功能。

其他的功能默认下，在上述功能中会自动完成。

HTTPConnection.connect()

关闭连接：

HTTPConnection.close()

设置连接服务器参数：

HTTPConnection.set\_tunnel(host, port=None, headers=None)

1. HTTPConnection模式一

In [ ]:

**import** **http.client**

host = 'www.baidu.com'

connection = http.client.HTTPConnection(host, port=80)

connection.request("GET", "/", body='', headers = {'Accept-Encoding':'identity'})

response = connection.getresponse()

print(response)

print(response.status, response.reason)

print(response.getheaders())

content = response.read()

*# print(content.decode()) # 内容太多，需要运行，再取消注释运行*

connection.close()

1. HTTPConnection模式二

In [ ]:

**import** **http.client**

host = 'www.baidu.com'

connection = http.client.HTTPConnection(host, port=80)

connection.connect()

connection.putrequest("GET", "/")

*# connection.putheader('Accept-Encoding','gzip, deflate') # http的模块不会自动处理协议，比如这里的压缩情况*

connection.putheader('Accept-Encoding','identity')

connection.endheaders()

connection.send(b'name=Louis')

response = connection.getresponse()

print(response)

print(response.status, response.reason)

content = response.read()

*# print(content.decode())*

connection.close()

注意： connect在request前调用，在putrequest后会重置连接，但在putheader后调用重置连接会导致数据丢失异常。 注意：其中send的data与request中的body参数对GET方法通常是不使用的。

1. 抓取http.client发起的数据请求协议

In [ ]:

**import** **http.client**

host = '127.0.0.1'

connection = http.client.HTTPConnection(host, port=9999)

connection.request("GET", "/", body='name=Louis', headers = {'Accept-Encoding':'identity'})

connection.close()

结果是： GET / HTTP/1.1 Host: 127.0.0.1:9999 Content-Length: 10 Accept-Encoding: identity

name=Louis

In [ ]:

**import** **http.client**

host = '127.0.0.1'

connection = http.client.HTTPConnection(host, port=9999)

connection.connect()

connection.putrequest("GET", "/")

*# connection.putheader('Accept-Encoding','gzip, deflate') # http的模块不会自动处理协议，比如这里的压缩情况*

*# connection.putheader('Accept-Encoding','identity')*

*#msg\_body = bytearray('pass=tom123','utf-8')*

*#connection.endheaders(message\_body=msg\_body)*

connection.endheaders(message\_body=b'pass=tom1453')

*# connection.send(b'name=Louis')*

connection.close()

注意：

connection.endheaders(）使用了message\_body，就不在使用send函数发送数据体了。

使用了putheader，而不使用endheaders，请求头不会发出去。

#### 处理请求响应

1. 响应数据读取
2. HTTPResponse.read([amt])

HTTPResponse.readinto(b)

1. 响应头
2. HTTPResponse.getheader(name, default=None)
3. HTTPResponse.getheaders()
4. HTTPResponse.msg

|- 其中返回的http.client.HTTPMessage本质是一个字典。按照字典方式获取响应头。

1. 其他信息获取
2. HTTPResponse.fileno()
3. HTTPResponse.status
4. HTTPResponse.reason

HTTPResponse.closed

下面是使用例子：

In [ ]:

**import** **http.client**

host = 'www.baidu.com'

connection = http.client.HTTPConnection(host, port=80)

connection.connect()

connection.putrequest("GET", "/")

*# connection.putheader('Accept-Encoding','gzip, deflate') # http的模块不会自动处理协议，比如这里的压缩情况*

connection.putheader('Accept-Encoding','identity')

connection.endheaders()

connection.send(b'name=Louis')

response = connection.getresponse()

print(response)

print(response.status, response.reason)

content = response.read()

print(response.getheaders())

print(response.msg)

**for** item **in** response.msg.items():

print('::',item)

connection.close()

print(response.closed)

### http.server模块

#### http.server的API结构

http.server模块与http.client是Web编程的两个方面：

|- http.client负责客户端底层的封装；

|- http.server负责服务器端底层封装；

如果具备了Socket的TCP编程模式，实际http.server的理解比较容易。因为毕竟就是那么几个步骤。

#### HTTPServer API说明

In [ ]:

**import** **http.server**

*# help(http.server)*

print(dir(http.server))

http.server核心就两个类：

|- socketserver.TCPServer(socketserver.BaseServer)

|- HTTPServer

|- socketserver.StreamRequestHandler(socketserver.BaseRequestHandler)

|- BaseHTTPRequestHandler

|- SimpleHTTPRequestHandler

|- CGIHTTPRequestHandler

1. HTTPServer API介绍
2. HTTPServer主要负责网络层面的处理，对客户端的请求，分发给另外一个类负责BaseHTTPRequestHandler。
3. BaseHTTPRequestHandler类是抽象的，没有任何实现，仅仅是一个接口规范，需要自己手工实现。如果想省事，可以直接调用SimpleHTTPRequestHandler实现基本的客户端请求处理。如果想实现简单的CGI处理，可以直接使用CGIHTTPRequestHandler类。

这个类支持多客户端的同步处理。

In [ ]:

**import** **http.server**

help(http.server.HTTPServer)

1. HTTPServer主要调用方法
2. | - \_\_init\_\_(server\_address, RequestHandlerClass, bind\_and\_activate=True)
3. | |- 创建一个HTTPServer对象
4. |
5. | - serve\_forever(poll\_interval=0.5)
6. | |- 接受客户连接与客户数据
7. |
8. | - shutdown()
9. | |- 关闭客户端连接
10. |
11. | - handle\_request() # if you don't use serve\_forever()
12. | |- 处理用户请求
13. |
14. | - fileno() -> int # for selector

| |- socket的文件描述符

1. 可以重载覆盖的方法
2. 一般不会去覆盖如下方法
3. | - server\_bind()
4. | - 对应socket.bind()函数
5. | - server\_activate()
6. | - 对应socket.listene()函数
7. | - get\_request() -> request, client\_address
8. | - handle\_timeout()
9. | - verify\_request(request, client\_address)
10. | - process\_request(request, client\_address)
11. | - shutdown\_request(request)
12. | - close\_request(request)

| - handle\_error()

1. 类变量
2. | - timeout # 延时常量
3. | - address\_family # socket地址族 AF\_INET
4. | - socket\_type # socket类型 SOCK\_STREAM
5. | - request\_queue\_size (only for stream sockets) # 请求队列大小（缓冲）

| - allow\_reuse\_address # 地址（包含端口）是否可以重用（默认是不能重用的）。

1. 实例变量
2. | - server\_address # 服务器地址（含端口）
3. | - RequestHandlerClass # 处理请求的类

| - socket # 服务器socket对象

#### BaseHTTPRequestHandler API 说明

BaseHTTPRequestHandler的继承结构如下：

|- socketserver.StreamRequestHandler

|- socketserver.BaseRequestHandler

In [ ]:

**import** **http.server**

**import** **socketserver**

*# help(http.server.BaseHTTPRequestHandler)*

*# dir(http.server.BaseHTTPRequestHandler)*

dir(http.server.SimpleHTTPRequestHandler)

*# dir(http.server.CGIHTTPRequestHandler)*

*# help(socketserver.BaseRequestHandler)*

*# help(socketserver.StreamRequestHandler)*

1. 三个核心数据
2. |- \_\_init\_\_(self, request, client\_address, server)
3. |- self.request
4. 对应每个客户的请求
5. |- self.client\_address
6. 对应每个客户的IP地址
7. |- self.server

对应服务server\_socket

1. 核心函数
2. | handle(self)
3. | |- 请求处理接口，该函数在BaseHTTPRequestHandler中提供基本的处理。
4. |
5. | setup(self)
6. | |- 初始化，该方法完成socket选项设置，在StreamRequestHandler类中提供实现，一般用户没有特殊情况，就不需要覆盖实现。
7. |
8. | finish(self)
9. | |- 关闭，该方法完成socket关闭与释放处理，在StreamRequestHandler类中提供实现，一般用户没有特殊情况，就不需要覆盖实现。
10. 下面三个类都是围绕上面三个函数提供更加细节化的实现。
11. |- http.server.BaseHTTPRequestHandler
12. |- 基本上没有提供任何实现
13. |- http.server.SimpleHTTPRequestHandler
14. |- 提供HEAD，POST与GET三种基本的请求实现
15. |- http.server.CGIHTTPRequestHandler
16. |- 提供CGI细节的实现，但具体CGI业务没有实现，需要用户

BaseHTTPRequestHandler

#### 使用代理类BaseHTTPRequestHandler默认处理请求

该代理类基本上只处理了基本的请求，所以运行的时候，会报错误：

|- 127.0.0.1 - - [21/Jan/2019 10:54:53] code 501, message Unsupported method ('GET')

In [ ]:

*# coding = utf-8*

**import** **http.server**

print('启动服务器')

server = http.server.HTTPServer(

server\_address=('', 11111),

RequestHandlerClass=http.server.BaseHTTPRequestHandler,

bind\_and\_activate=**True**)

print('接受用户请求')

server.serve\_forever()

print('服务器退出')

#### 使用代理类BaseHTTPRequestHandler处理定制

在对应的请求方法中，实现独自的业务处理，其他的HTTP细节由BaseHTTPRequestHandler处理完毕。

包含请求的方法解析，请求头的解析处理。

BaseHTTPRequestHandler已经提供的处理包含：

|- 请求方法 （来自请求行）

|- 请求的资源 （来自请求行）

|- 请求头 （来自请求头）

其中请求的数据体：通过rwfile读取

响应的处理：

|- 响应行：send\_response

|- 响应头：send\_header与end\_headers

|- 响应体：wfile.write

In [ ]:

*# coding = utf-8*

**import** **http.server**

**import** **os**

**class** **MyHTTPHandler**(http.server.BaseHTTPRequestHandler):

**def** do\_GET(self):

print('用户请求！')

print('客户地址：', self.client\_address)

print('请求头：', self.headers)

print('请求方法：', self.command)

print('请求资源：', self.path)

print('服务器socket：', self.server)

print('HTTP版本：', self.server\_version)

self.send\_response(200) *# 响应行*

self.send\_header('Content-type', 'text/html;charset=utf-8') *# 响应头*

self.end\_headers() *# 响应头结束*

self.wfile.write('<h1>这是来自服务器的请求</h1>'.encode('utf-8')) *# 响应数据体*

print('方便kill进程的PID：',os.getpid())

print('启动服务器')

server = http.server.HTTPServer(

server\_address=('', 11111),

RequestHandlerClass=MyHTTPHandler,

bind\_and\_activate=**True**)

print('接受用户请求')

server.serve\_forever()

print('服务器退出')

#### BaseHTTPRequestHandler的handle函数的理解

1. 重载handle方法
2. 重载handle函数后，下面程序肯定出错，因为self.headers，self.command，self.path都是在BaseHTTPRequestHandler原来的handle中处理的，被覆盖后，这些成员属性就没有任何效果。
3. 甚至包括self.send\_response等所有用到request解析结果的函数都会出错或者失效。

如果重载handle函数的话，需要自己处理请求解析。

In [ ]:

*# coding = utf-8*

**import** **http.server**

**import** **os**

**class** **MyHTTPHandler**(http.server.BaseHTTPRequestHandler):

**def** handle(self):

print('用户请求！')

print('客户地址：', self.client\_address)

print('请求头：', self.headers)

print('请求方法：', self.command)

print('请求资源：', self.path)

print('服务器socket：', self.server)

print('HTTP版本：', self.server\_version)

self.send\_response(200) *# 响应行*

self.send\_header('Content-type', 'text/html;charset=utf-8') *# 响应头*

self.end\_headers() *# 响应头结束*

self.wfile.write('<h1>这是来自服务器的请求</h1>'.encode('utf-8')) *# 响应数据体*

print('方便kill进程的PID：',os.getpid())

print('启动服务器')

server = http.server.HTTPServer(

server\_address=('', 11111),

RequestHandlerClass=MyHTTPHandler,

bind\_and\_activate=**True**)

print('接受用户请求')

server.serve\_forever()

print('服务器退出')

#### 重载handle函数定制处理请求

在BaseHTTPRequestHandler中，实际上有三个对象本质是一样的：

|- self.rfile

|- self.wfile

|- self.request

这rfile与wfile对象都是从request衍生的对象，一个读，一个写。其实从request都可以完成

分开的原因很简单，读是阻塞的。而且可以用来方便使用多路复用。

更加重要的，分开变成IO的文件描述符，使用缓冲功能。

1. 使用self.request读取请求

In [ ]:

*# coding = utf-8*

**import** **http.server**

**import** **os**

**class** **MyHTTPHandler**(http.server.BaseHTTPRequestHandler):

**def** handle(self):

print('request:', self.request)

**while** **True**:

buffer = self.request.recv(1024\*4, 0)

**if** **not** buffer:

**break**;

print(buffer.decode())

print('方便kill进程的PID：',os.getpid())

print('启动服务器')

server = http.server.HTTPServer(

server\_address=('', 11111),

RequestHandlerClass=MyHTTPHandler,

bind\_and\_activate=**True**)

print('接受用户请求')

server.serve\_forever()

print('服务器退出')

2 .使用self.rfile读取请求

In [ ]:

*# coding = utf-8*

**import** **http.server**

**import** **os**

**class** **MyHTTPHandler**(http.server.BaseHTTPRequestHandler):

**def** handle(self):

print('rfile:', self.rfile)

**while** **True**:

buffer = self.rfile.read(1024\*4)

**if** **not** buffer:

**break**;

print(buffer.decode())

print('方便kill进程的PID：',os.getpid())

print('启动服务器')

server = http.server.HTTPServer(

server\_address=('', 11111),

RequestHandlerClass=MyHTTPHandler,

bind\_and\_activate=**True**)

print('接受用户请求')

server.serve\_forever()

print('服务器退出')

1. 使用request与wfile输出响应
2. 明白上面的原理后，输出响应就比较简单了。

代码略

In [ ]:

*#*

#### SimpleHTTPRequestHandler 使用

由于SimpleHTTPRequestHandler类实现了简答的GET，POST与HEAD，其他与BaseHTTPRequestHandler一样。

不过因为BaseHTTPRequestHandler实现了基本的HTTP请求方法，所以可以直接使用，当然需要重载也是可以的。

In [ ]:

*# coding = utf-8*

**import** **http.server**

print('启动服务器')

server = http.server.HTTPServer(

server\_address=('', 11111),

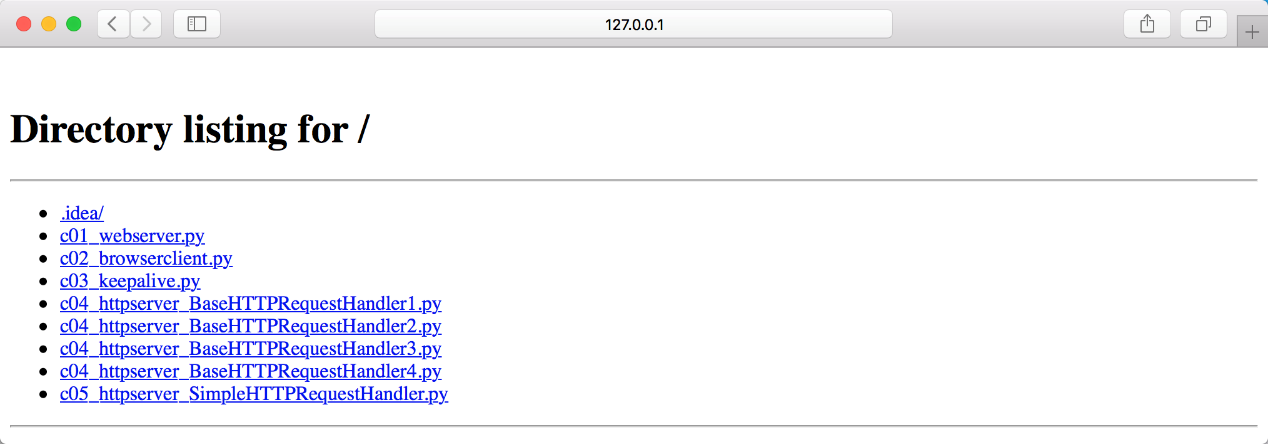
RequestHandlerClass=http.server.SimpleHTTPRequestHandler,

bind\_and\_activate=**True**)

print('接受用户请求')

server.serve\_forever()

print('服务器退出')

浏览器访问结果如下（直接罗列服务器本地文件目录）：

### http.cookies模块

#### http.cookies API结构

提供了两个类来生成响应Cookie，添加值，会自动按照Set-Cookie格式构建响应Cookie：

builtins.dict(builtins.object)

|- BaseCookie

|- SimpleCookie

SimpleCookie与BaseCookie都是一个字典。只是提供了基于Cookie数据的解析功能。

|- BaseCookie提供Cookie处理的基本规范与解析

|- SimpleCookie与BaseCookie基本一样，只是重载了两个函数value\_decode与value\_encode

从下面例子可以看得出来，SimpleCookie与BaseCookie都是用来构建响应Cookie的，就是构造：

Set-Cookie头。

BaseCookie的每个数据都是Morsel类型。

In [ ]:

**import** **http.cookies**

dir(http.cookies)

#### BaseCookie在HTTP请求中使用的例子

In [ ]:

*# coding = utf-8*

**import** **http.server**

**import** **http.cookies**

**import** **os**

**class** **MyHTTPHandler**(http.server.BaseHTTPRequestHandler):

**def** do\_GET(self):

*# super().handle()*

*# 处理Cookie*

cookies = **None**

print(self.headers)

**if** 'Cookie' **in** self.headers:

cookies = http.cookies.BaseCookie(self.headers['Cookie'])

print(cookies)

print(cookies.keys())

**for** k, v **in** cookies.items():

print(k,":", v)

self.send\_response(200) *# 响应行*

self.send\_header('Content-type', 'text/html;charset=utf-8') *# 响应头*

self.end\_headers() *# 响应头结束*

self.wfile.write('<h1>这是来自服务器的请求</h1>'.encode('utf-8')) *# 响应数据体*

print('方便kill进程的PID：',os.getpid())

print('启动服务器')

server = http.server.HTTPServer(

server\_address=('', 11111),

RequestHandlerClass=MyHTTPHandler,

bind\_and\_activate=**True**)

print('接受用户请求')

server.serve\_forever()

print('服务器退出')

输出结果：

Set-Cookie: \_xsrf=2|fab4a780|f1b7a8abb150de48ee31253159a0effe|1548039281

dict\_keys(['\_xsrf'])

\_xsrf : Set-Cookie: \_xsrf=2|fab4a780|f1b7a8abb150de48ee31253159a0effe|1548039281

#### BasicCookie与SimpleCookie区别的例子

1. BasicCookie使用

In [ ]:

*# coding = utf-8*

**import** **http.cookies**

cookies = http.cookies.BaseCookie('name=louis')

print(cookies)

print(type(cookies))

print(isinstance(cookies, dict))

re = cookies.value\_decode('name=louis')

print(re)

re=cookies.value\_encode('name=louis')

print(re)

1. SimpleCookie

In [ ]:

*# coding = utf-8*

**import** **http.cookies**

cookies = http.cookies.SimpleCookie('name=louis')

print(cookies)

print(type(cookies))

print(isinstance(cookies, dict))

re = cookies.value\_decode('name=louis')

print(re)

re=cookies.value\_encode('name=louis')

print(re)

注意：多了两个引号。 在SimpleCookie中value\_decode添加等价字符串，value\_encode添加两个字符串表示字符。

#### SimpleCookie添加Cookie的几种方式

In [ ]:

*# coding = utf-8*

**import** **http.cookies**

cookies = http.cookies.SimpleCookie('name=louis')

cookies['pass'] = 'tom123'

cookies.load('work=swim')

print(cookies.output())

out = cookies.output()

print(type(out))

print(out.encode('utf-8'))

注意：自动添加\r\n符号

### http.cookiejar模块

http.cookiejar模块提供Web客户端Cookie的解析与处理。

#### http.cookiejar模块API结构

In [ ]:

**import** **http.cookiejar**

help(http.cookiejar)

*# dir (http.cookiejar)*

1. 核心的类
2. builtins.object
3. |- Cookie
4. Cookie 数据
5. |- CookieJar
6. 加载管理Cookie文件
7. | |- FileCookieJar
8. | |- LWPCookieJar libwww-perl格式
9. | |- MozillaCookieJar Mozilla格式
10. | |- MSIECookieJar 微软格式
11. |- CookiePolicy
12. 管理Cookie策略

|- DefaultCookiePolicy

1. 每个类的API说明

直接使用Help得到更加详细的说明

#### Cookie说明

Cookie类的构造器：

def \_\_init\_\_(self, version, name, value,

port, port\_specified,

domain, domain\_specified, domain\_initial\_dot,

path, path\_specified,

secure,

expires,

discard,

comment,

comment\_url,

rest,

rfc2109=False,)

Cookie类是一个实体类，主要用来表示Cookie数据，该数据主要主要来自：

| - 服务器响应（也可以写入请求）

| - 浏览器本地文件加载（写一个保存到本地文件）

#### CookieJar说明与使用

CookieJar类主要用来处理Cookie，处理功能主要提现在如下两个方法：

|- add\_cookie\_header(request)

|- 把CookieJar中数据写入请求头。

|- extract\_cookies(response, request)

|- 从response抽取Cookie数据。

注意：

|- 上面的request一般指urllib.request.Request

|- 上面的response与一般需要有info()函数，该response一般是urllib.request.urlopen()返回的response。

CookieJar中内容访问：

|- CookieJar支持迭代器(iterator)协议，迭代器元素是：Cookie迭代对象。

1. 获取服务器响应的Cookie

In [ ]:

*# coding = utf-8*

**import** **http.cookiejar**

**import** **urllib.request**

**import** **http.client**

host = 'http://www.baidu.com'

request = urllib.request.Request(url=host)

response = urllib.request.urlopen(request)

cookie = http.cookiejar.CookieJar()

*# result = cookie.make\_cookies(response, request)*

cookie.extract\_cookies(response,request)

**for** item **in** cookie:

*# print(item)*

print(item.name, ':', item.value)

#### 使用LWPCookieJar保存与加载Cookie数据

1. 保存Cookie数据

In [ ]:

*# coding = utf-8*

**import** **http.cookiejar**

**import** **urllib.request**

**import** **http.client**

host = 'http://www.baidu.com'

request = urllib.request.Request(url=host)

response = urllib.request.urlopen(request)

cookie = http.cookiejar.LWPCookieJar('a.txt')

*# result = cookie.make\_cookies(response, request)*

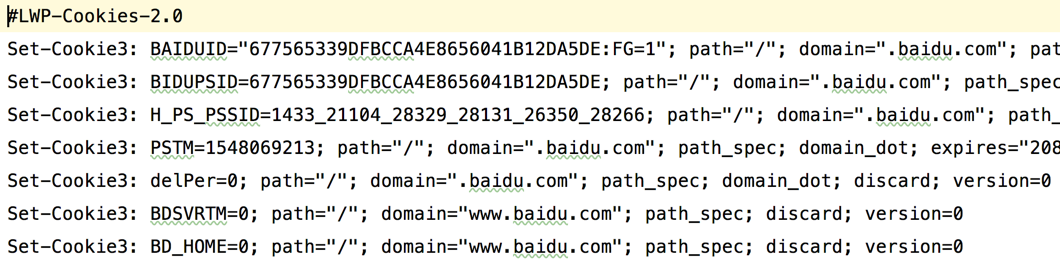
cookie.extract\_cookies(response, request)

cookie.save(ignore\_discard=**True**, ignore\_expires=**True**)

**for** item **in** cookie:

*# print(item)*

print(item.name, ':', item.value)

保存的文件是下面格式（LWP指的是libwww-perl 兼容的Cookie格式）：

1. 加载Cookie数据

In [ ]:

*# coding = utf-8*

**import** **http.cookiejar**

cookie = http.cookiejar.LWPCookieJar('a.txt', delayload=**False**)

cookie.load(ignore\_discard=**True**, ignore\_expires=**True**)

**for** item **in** cookie:

*# print(item)*

print(item.name, ':', item.value)

BAIDUID : 677565339DFBCCA4E8656041B12DA5DE:FG=1

BIDUPSID : 677565339DFBCCA4E8656041B12DA5DE

H\_PS\_PSSID : 1433\_21104\_28329\_28131\_26350\_28266

PSTM : 1548069213

delPer : 0

BDSVRTM : 0

BD\_HOME : 0

#### 使用MozillaCookieJar保存与加载Cookie

1. 保存Cookie

In [ ]:

*# coding = utf-8*

**import** **http.cookiejar**

**import** **urllib.request**

**import** **http.client**

cookie = http.cookiejar.MozillaCookieJar('a.txt', delayload=**False**)

host = 'http://www.baidu.com'

request = urllib.request.Request(url=host)

response = urllib.request.urlopen(request)

cookie.extract\_cookies(response, request)

cookie.save(ignore\_discard=**True**, ignore\_expires=**True**)

**for** item **in** cookie:

*# print(item)*

print(item.name, ':', item.value)

In [ ]:

*# 显示上面的问价*

**with** open('a.txt') **as** fd:

lines = fd.read()

print(lines)

1. 加载Cookie

In [ ]:

*# coding = utf-8*

**import** **http.cookiejar**

**import** **urllib.request**

**import** **http.client**

cookie = http.cookiejar.MozillaCookieJar('a.txt', delayload=**False**)

cookie.load(ignore\_discard=**True**,ignore\_expires=**True**)

**for** item **in** cookie:

*# print(item)*

print(item.name, ':', item.value)

#### 备注说明

实际上现在不同的浏览器都提供Cookie的存放格式：

|- FireFox与Safari使用的是Sqlite存储方式

|- Chrome使用的是二进制存储方式

大家可以使用我们上面介绍的Cookie存放位置，找到不同浏览器的Cookie存储文件，并常识自己读取加载。

# 理解SSL过程与机制

## 理解SSL的加密过程与机制

SSL加密机制实际是一种信任机制，这个在现实有很多例子：比如印章就是一种信任机制。常用加密技术不完全类似印章的原理，但本质都差不多。

SSL中数据通信都采用加密，其加密的机制使用的是CA签名加密机制。

### 对称加密与非对称加密

加密算法分为两种：对称加密和非对称加密。

|- 对称加密就是发送双发使用相同的密钥对消息进行加解密，常见的对称加密为DES、3DES,AES等。

|- 非对称加密是发送双方各自拥有一对公钥私钥，其中公钥是公开的，私钥是保密的。当发送方向接收方发送消息时，发送方利用接收方的公钥对消息进行加密，接收方收到消息后，利用自己的私钥解密就能得到消息的明文。其中非对称加密方法有RSA、Elgamal、ECC等。

CA加密机制就是一种非对称加密机制。

### 数字签名

数字签名在网络通讯中的过程如下：

|- 当通信A端向B端发送信息时，会将报文生成报文摘要，同时对报文摘要进行hash计算，得到hash值，然后对hash值进行加密，然后将加密的hash值放置在报文后面，这个加密后的hash值就称为签名。

|- 通信A端将报文、签名和数字证书一同发送给B端。B端收到这些信息后，会首先验证签名，利用签名算法对签名进行解密，得到报文摘要的hash值，然后将得到的报文生成报文摘要并利用签名hash算法生成新的hash值，通过对比这两个hash值是否一致，就能判断信息是否完整，是否是由真正的通信A端发送的。

数字签名有两个作用：

|- 确认消息发送方可靠

|- 确认消息完整准确。

### SSL加密过程

1. 使用openssl工具生成CA证书

产生一个证书文件与私钥。

1. 使用openssl工具生成需要签名的服务器（或者客户端）证书与私钥。

产生一个待签名证书与私钥。

1. 使用openssl工具，用CA证书对服务器证书签名。

产生一个签名证书

## 使用openssl工具产生CA签名证书实现

### 产生CA根证书与私钥

openssl req -newkey rsa:2048 -passout pass:123456 -keyout ca\_rsa\_private.pem -x509 -days 365 -out ca.crt -subj "/C=CN/ST=SC/L=CD/O=MAGE/OU=TEACHING/CN=NO/emailAddress=1234567@qq.com"

使用的是RSA非对称加密，两个重要参数说明：

-keyout ca\_rsa\_private.pem ：指定生成的密钥文件；

-out ca.crt：指定生成证书的文件。

localhost:certs yangqiang$ openssl req -newkey rsa:2048 -passout pass:123456 -keyout ca\_rsa\_private.pem -x509 -days 365 -out ca.crt -subj "/C=CN/ST=SC/L=CD/O=MEGE/OU=TEACHING/CN=NO/emailAddress=1234567@qq.com"

Generating a 2048 bit RSA private key

....................+++

...............+++

writing new private key to 'ca\_rsa\_private.pem'

-----



### 产生服务器（或者客户端）待签名证书与私钥

openssl req -newkey rsa:2048 -passout pass:server -keyout server\_rsa\_private.pem -out server.csr -subj "/C=CN/ST=BJ/L=CP/O=MAGEZongbu/OU=MNGT/CN=SERVER/emailAddress=38395870@qq.com"

其中两个关注的参数说明：

|- -out server.csr 指定生成的待签名证书。

|- -keyout server\_rsa\_private.pem 指定生成的服务器（或者客户端）私钥。

|- -passout pass:server 密码（这个在服务器加载整数与私钥的时候，需要每次都输入，如果不想输入，可以生成没有密码的私钥。）

localhost:certs yangqiang$ openssl req -newkey rsa:2048 -passout pass:server -keyout server\_rsa\_private.pem -out server.csr -subj "/C=CN/ST=BJ/L=CP/O=MAGEZongbu/OU=MNGT/CN=SERVER/emailAddress=38395870@qq.com"

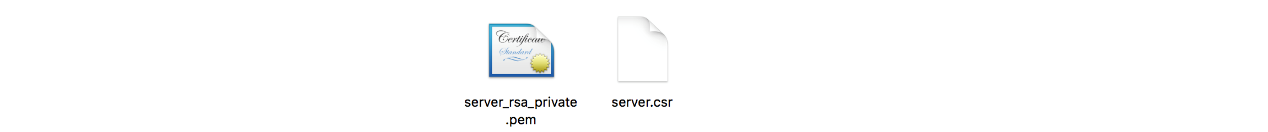
Generating a 2048 bit RSA private key

......................................................................+++

................................+++

writing new private key to 'server\_rsa\_private.pem'

-----



### 对服务器证书进行签名

openssl x509 -req -days 365 -in server.csr -CA ca.crt -CAkey ca\_rsa\_private.pem -passin pass:123456 -CAcreateserial -out server.crt

重要参数说明：

|- -in server.csr 需要签名的整数。

|- -CA ca.crt 根证书

|- -CAkey ca\_rsa\_private.pem 根证书私钥

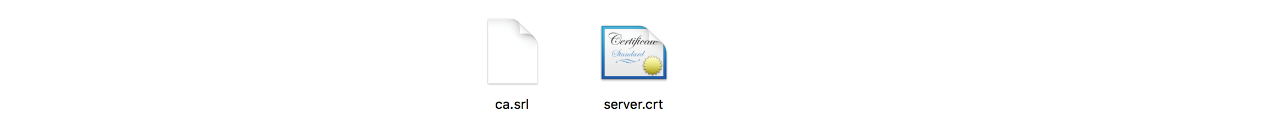
|- -out server.crt 生成的签名证书（可以使用的）

localhost:certs yangqiang$ openssl x509 -req -days 365 -in server.csr -CA ca.crt -CAkey ca\_rsa\_private.pem -passin pass:123456 -CAcreateserial -out server.crt

Signature ok

subject=/C=CN/ST=BJ/L=CP/O=MAGEZongbu/OU=MNGT/CN=SERVER/emailAddress=38395870@qq.com

Getting CA Private Key



## 使用python实现SSL服务器认证加密

也可以实现客户认证加密，也可以实现双向认证加密。

### ssl模块的API结构

使用help与dir可以仔细阅ssl模块的帮助与API结构，实际上核心的类就两个：

|- SSLContext

|- SSLSocket

在help中可以看到其继承结构如下：

|- socket.socket(\_socket.socket)

|- SSLSocket

|- 包含socket中方法，也包含SSL特色的网络通信操作函数。

|- \_ssl.\_SSLContext(builtins.object)

|- SSLContext

|- \_\_init\_\_(self, protocol=<\_SSLMethod.PROTOCOL\_TLS: 2>) ：指定版本

|- load\_cert\_chain(self, /, certfile, keyfile=None, password=None) ：加载证书与私钥

|- wrap\_socket(self, sock, server\_side=False, do\_handshake\_on\_connect=True, suppress\_ragged\_eofs=True, server\_hostname=None, session=None)：对socket通信进行签名加密处理。

In [ ]:

**import** **ssl**

*# print(dir(ssl))*

*# print(dir(ssl.SSLContext))*

*# help(ssl.SSLContext)*

*# help(ssl.SSLSocket)*

*# print(dir(ssl.SSLSocket))*

### SSL socket服务器端实现

In [ ]:

*# coding = utf-8*

**import** **ssl**

**import** **socket**

*# 1. 创建SSL上下文，需要指定版本,缺省的是PROTOCOL\_TLS，我们指定服务器签名PROTOCOL\_TLS\_SERVER*

*# 这一步的关键是选择合适的版本，每个不同的版本之间是不互融的。*

ctx = ssl.SSLContext(ssl.PROTOCOL\_TLS\_SERVER)

*# 2. 加载证书与服务器私钥*

ctx.load\_cert\_chain(certfile='./certs/server.crt', *# 服务器已经签名的证书*

keyfile='./certs/server\_rsa\_private.pem', *# 服务器私钥*

password='server') *# 生成服务器时输入的密码（这里不指定，就需要输入）*

*# 3. 创建socket*

server\_socket = socket.socket(

socket.AF\_INET,

socket.SOCK\_STREAM,

socket.IPPROTO\_TCP)

*# 4. 使用ctx创建一个包装器socket，该socket具有签名加密功能。*

ssl\_server\_socket = ctx.wrap\_socket(

sock=server\_socket, *# 被包装的socket*

server\_side=**True**) *# 指定是服务器测签名加密*

*# 5. 正常的socket通信操作*

*# 绑定服务器地址*

ssl\_server\_socket.bind(('', 8443)) *# 443时SSL通用端口,注意使用元组表示地址格式*

*# 监听*

ssl\_server\_socket.listen(2)

**while** **True**:

*# 接受客户连接*

print('等待客户连接！')

client\_socket, client\_address = ssl\_server\_socket.accept()

print('有客户连接：', client\_address)

*# 接受客户数据*

info = client\_socket.recv(1024).decode("utf-8")

print(f"从客户**{**client\_address**}**接受的信息**{**info**}**")

*# 给客户发送一个信息*

client\_socket.send('来自服务器的信息！收到没有？'.encode('utf-8'))

client\_socket.close()

**break**

ssl\_server\_socket.close()

### SSL socket客户端实现

In [ ]:

*# coding = utf-8*

**import** **ssl**

**import** **socket**

*# 1.创建SSL通信上下文*

ctx = ssl.SSLContext(ssl.PROTOCOL\_TLS\_CLIENT)

*# 2.加载根证书*

ctx.load\_verify\_locations('./certs/ca.crt')

*# 3.创建一个客户端socket*

client\_socket = socket.socket(

socket.AF\_INET,

socket.SOCK\_STREAM,

socket.IPPROTO\_TCP)

*# 4.创建socket包装器，实现SSL通信*

ssl\_client\_socket = ctx.wrap\_socket(

sock=client\_socket,

server\_side=**False**,

server\_hostname='SERVER') *# 这个值是识别创建服务器证书的产品名，就是CN指定的名字*

*# 5.通用的socket通信模式*

*# 连接到服务器*

ssl\_client\_socket.connect(('127.0.0.1',8443))

*# 发送数据*

ssl\_client\_socket.send('来自客户的数据！'.encode('utf-8'))

*# 接受数据*

info = ssl\_client\_socket.recv(1024).decode("utf-8")

print(f"从服务器接受的信息：**{**info**}**")

ssl\_client\_socket.close()

这里只是通过证书说明SSL中的签名加密过程，在这个基础上，可以实现客户端签名认证与双向签名认证。

**注意：**上面两个程序不能同时在ipython中执行，因为ipython是单线程，服务器运行起来，客户端没有办法执行。

# 可理解HTTPS

## HTTPS就是HTTP+SSL的协议。

SSL包含两层含义： | - 加密 | - 信任

一般我们再访问HTTPS站点的时候，都使用证书，可以在点击地址栏，查看证书信息。



## 浏览器证书安装与下载

通过 SSL 加密的 HTTPS 连接访问网站时，需要安装并配置一个受信任的 CA 根证书（Trusted CA Root Certificate）。

通常访问一些加密网站（https）之所以不需要自己安装证书，是因为系统或浏览器已经提前安装了一些受信任机构颁发的证书。

## 怎么保障服务器的身份

  正如上面我们自己也可以颁发证书，怎么保证这个证书是信任的呢？也就是服务器怎么是可信的？

  正规CA的证书是有比较靠谱的第三方机构颁发。     CA证书是收费的（从每年三百块RMB到一万五不等）。最贵的证书是VeriSign签发的。

  12306的证书原来是自己颁发的，现在是DigiCert Inc（成都的一家中国公司）颁发（ 可以访问官网：<https://www.sslcertificateshop.com> ）

  中国铁路总公司采用自己的SSL证书的原因是：CA机构也有不靠谱的。  
    
  荷兰一家CA，叫DigiNotar。这CA被黑客攻破，导致这家CA办法给一些用户的证书的私钥失效（私钥是在SSL加密环节中非常重要的东西），这就使得以这家CA的名义伪造证书成了可能。黑客可以通过这家公司的名义伪造证书给一些非法网站，客户一看这是加密过的还是大型CA签出来的证书便很容易信任。因为DigiNotar名气很大，并且很多大公司都使用它的证书，微软等操作系统厂商在这事情发生之后开始忙不迭的发布更新补丁来宣布DigiNotar的证书失效。

# HTTP协议理解

## 网络协议结构与HTTP协议

### 网络OSI模型

网络OSI（Open System Interconnection）协议规范分成7层：

|-物理层

|-链路层

|-网络层

|-传输层

|-会话层

|-表示层

|-应用层

一般从开发协议来说分成3层（链路层与网络层看成一起，也是常见的高层，中层与低层）或者4层。

|-链路层（ETHERNET协议：ARP）

|-网络层（IP等）

|-传输层（TCP，UDP等）

|-应用层（HTTP，FTP，SMTP，TELNET等）

### HTTP协议层

HTTP协议是应用协议，他下面依赖的是TCP传输协议，TCP依赖的是IP协议，IP协议依赖的是链路层ARP（Address Resolution Protocol）协议。

HTTP

|

TCP

|

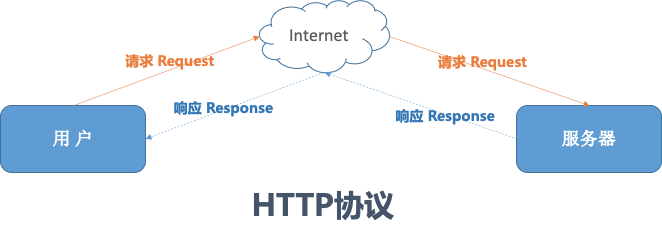
IP

|

ARP

1. HTTP协议特点：
2. HTTP协议是基于TCP协议的应用协议，是属于应用层的面向对象的协议，浏览器实现的就是HTTP协议。它于1990年提出，目前使用的是HTTP/1.0与1.1，HTTP-NG(Next Generation of HTTP)的建议已经提出。
3. HTTP协议的主要特点：
4. 1.基于TCP协议。
5. 2.简单快速，协议是基于文本的：
6. 客户向服务器请求服务时，只需传送请求方法和路径。由于HTTP协议简单，使得HTTP服务器的程序规模小，因而通信速度很快。
7. 3.灵活：
8. HTTP允许传输任意类型的数据对象。正在传输的类型由Content-Type加以标记。
9. 4.无连接：
10. 服务器处理完客户的请求，并收到客户的应答后，即断开连接。
11. 5.无状态：

HTTP协议是无状态协议。不识别连接者的状态。



## HTTP协议格式

HTTP协议是Web浏览器与Web服务器之间的对话协议，HTTP协议可以根据分工分成请求协议与响应协议两个部分：

|- 请求协议：Web浏览器向Web服务器发起请求使用的协议；

|- 响应协议：Web服务器对Web浏览器的请求进行响应的协议；

1. HTTP协议的，主要有如下三个部分构成。
2. |-
3. |- HTTP URL
4. |- HTTP请求

|- HTTP响应

1. HTTP/1.0与1.1
2. |- HTTP/1.0
3. 这是第一个在通讯中指定版本号的 HTTP 协议版本，至今仍被广泛采用，特别是在代理服务器中。
4. |- HTTP/1.1
5. 当前版本。持久连接被默认采用，并能很好地配合代理服务器工作。还支持以管道方式在同时发送多个请求，以便降低线路负载，提高传输速度。
6. 1.1对比1.0的主要区别特点
7. |- 1 缓存处理
8. |- 2 带宽优化及网络连接的使用
9. |- 3 错误通知的管理
10. |- 4 消息在网络中的发送
11. |- 5 互联网地址的维护

|- 6 安全性及完整性

1. HTTP URL介绍
2. 也称：通用资源定位符
3. 格式组成：协议，主机，资源路径
4. |- http://host[":"port] [abs\_path]
5. |- 协议：常见的协议：HTTP，HTTPS，FTP，本地文件，RMI等。
6. |- 主机：包含IP或者域名，端口
7. |-资源路径：
8. 使用 / 分隔

资源路径可以后缀?开始的字符串，称为querystring（查询字符串，类似SQL语句作用）。Querystring可以使用&合并多个值对。

例子：一般浏览器都支持

|- WEB：http://baidu.com/index.html

|- WEB：http:// 220.181.111.85:80/index.html?name=louis&favor=football。

|- FTP：ftp:// 220.181.111.85/publish/a.txt

|- FILE：file:///root/codes

### HTTP协议请求格式

下面使用socket编程来实现一个程序模拟web服务器，并抓取浏览器发送请求数据，通过这些请求数据来认知HTTP协议的请求格式。

下面例子使用的技术是socket通信编程技术，因为HTTP是建立在TCP之上，所以下面在用TCP编程模式，其中对多用户处理，采用多路并发的编处理技术。

#### 接收显示浏览器请求数据的程序实现

In [ ]:

*# coding =utf-8*

**import** **socket** *# socket编程模块*

**import** **select** *# 多路复用模块*

**import** **sys** *# 系统调用*

**import** **signal**

**import** **os**

*# 方便发送信号灭掉这个进程*

print('本进程ID：**%d**，可以使用kill灭之，如果在ipython中不方便结束的话！' % os.getpid())

*# 这里处理ctrl+c的信号（方便ctrl+c退出）*

**def** handle\_int(signum, handler):

print('程序中断退出，信号：**%d**' % signum, '处理器是：**{}**'.format(handler))

sys.exit(0)

*# 绑定对ctrl+c信号的处理*

signal.signal(signalnum=signal.SIGINT, handler=handle\_int)

*# 下面代码为了清晰思路，没有做任何代码的异常处理*

*# 多路复用数据*

io\_inputs = []

io\_outputs = []

io\_error = []

*# 创建socket*

server\_socket = socket.socket(

socket.AF\_INET, *# 网络地址族：常用的是internet网地址格式（IP地址）*

socket.SOCK\_STREAM, *# 网络通信方式：流与报文两种*

socket.IPPROTO\_TCP) *# 通信协议：数据包的格式*

*# 绑定地址*

server\_address = ('', 9999) *# 地址包含IP地址与端口地址*

server\_socket.bind(server\_address)

*# 监听*

server\_socket.listen(2)

*# 把server\_socket加入多路复用IO中*

io\_inputs.append(server\_socket)

*# 开始监控多路复用异步IO（包含server\_socket的连接，新连接的也加入，短线的删除）*

**while** **True**:

print('开始多路复用IO监控......')

ready\_inputs, ready\_outputs, ready\_error = select.select(

io\_inputs, io\_outputs, io\_error, **None**)

*# 检查返回值，并相应的处理,这里我们只接收，不发送，所以不处理输出IO*

**for** fd **in** ready\_inputs:

*# 服务器IO与每个客户的IO分开处理*

**if** fd == server\_socket: *# 服务器IO*

*# 对服务器IO的护处理：接收客户连接*

client\_socket, client\_address = fd.accept() *# 这里需要处理异常，就是服务器挂掉，退出应用*

print('客户连接：IP=**%s**,PORT=**%d**' % client\_address)

*# 把新连接的客户加入多路复用监控处理*

io\_inputs.append(client\_socket)

**else**: *# 每个客户的IO*

*# 对客户IO是接收数据请求：请求都是HTTP协议的请求协议*

**while** **True**:

buffer = fd.recv(1024\*4, 0) *# 接收缓冲大小与接收标记*

**if** **not** buffer:

fd.close() *# 关闭*

io\_inputs.remove(fd) *# 客户退出*

print('客户连退出')

**break**

**else**:

print(buffer.decode('UTF-8'))

*# 上述代码可以粘贴到pycharm等IDE环境中运行，也可以在这儿运行*

#### HTTP请求协议的格式说明

GET / HTTP/1.1  
Host: 127.0.0.1:9999  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,/;q=0.8  
Upgrade-Insecure-Requests: 1  
Cookie: \_xsrf=2|f97c87ab|aa4e0381b08224325486cb739b6d5c6a|1547777015  
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/12.0.2 Safari/605.1.15  
Accept-Language: zh-cn  
Accept-Encoding: gzip, deflate  
Connection: keep-alive  
空行  
请求体

HTTP请求协议由4各部分构成：

|- 请求行：GET / HTTP/1.1

|- 请求头：Host: 127.0.0.1:9999

|- 空行： 表示请求头结束

|- 响应体： 表示发送给服务器的数据

实际上，上面每一行结束都是使用两个非可视化字符表示结束：'\r\n'。这是属于HTTP协议的一部分。

1. 请求行
2. GET / HTTP/1.1
3. 由三个部分构成
4. |- GET： 请求方法
5. |- /： 请求资源
6. |- HTTP/1.1： 请求协议与版本
7. 浏览器默认发起的请求方法都是GET。

用户可以根据自身情况，指定请求资源：

1. 请求头：
2. Host: 127.0.0.1:9999
3. Accept: text/html,application/xhtml+xml,application/xml;q=0.9,/;q=0.8
4. Upgrade-Insecure-Requests: 1
5. Cookie: xsrf=2|f97c87ab|aa4e0381b08224325486cb739b6d5c6a|1547777015
6. User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 1012\_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/12.0.2 Safari/605.1.15
7. Accept-Language: zh-cn
8. Accept-Encoding: gzip, deflate
9. Connection: keep-alive
10. 由多行构成，每行由两个部分构成：
11. |- HEADER键：表示数据的用途
12. |- HEADER值：发起请求的数据

所有的HEADER的名称由HTTP协议规定，HEADER值的格式由协议规定。

1. 空行

请求头后面紧跟一个空行，表示请求数据描述完毕；空行后面紧跟的是请求的用户数据。

1. 用户数据
2. 用户数据紧跟在空行后面，比如用户登录的用户名与密码等。
3. 用户数据在某些请求方法是没有的，比如GET请求方法的请求协议就没有数据行。因为GET主要目的是GET数据，不是发送数据。GET方法传递数据使用QueryString格式。

POST方法的数据行就存在，因为POST请求方式就是上传数据，比如用户登录信息，上传文件等。POST方法的目的就是POST数据。

**作业：**

使用HTML写一个页面，使用GET，POST，PUT，TRACE等方法发起请求，并观察HTTP请求协议的格式。

#### HTTP请求方法说明

下面介绍的请求方法，其中GET，POST与HEAD是必须实现的，其他的可选实现。

1. GET方法

GET：请求指定的页面信息，并返回实体主体。

1. POST方法

POST： 请求服务器接受所指定的文档作为对所标识的URI的新的从属实体。

1. HEAD方法

HEAD： 只请求页面的首部。

1. PUT方法

PUT： 从客户端向服务器传送的数据取代指定的文档的内容。

1. DELETE方法：

DELETE： 请求服务器删除指定的页面。

1. OPTIONS方法：

OPTIONS： 允许客户端查看服务器的性能。

1. TRACE方法：

TRACE： 请求服务器在响应中的实体主体部分返回客户端发送的数据。 （回显数据，便于跟踪与测试）

1. 其他方法：
2. PATCH： 实体中包含一个表，表中说明与该URI所表示的原内容的区别。
3. MOVE： 请求服务器将指定的页面移至另一个网络地址。
4. COPY： 请求服务器将指定的页面拷贝至另一个网络地址。
5. LINK： 请求服务器建立链接关系。
6. UNLINK： 断开链接关系。
7. WRAPPED： 允许客户端发送经过封装的请求。

Extension-mothed：在不改动协议的前提下，可增加另外的方法。

### HTTP协议响应格式

#### 模拟浏览器发起HTTP请求，实现响应协议数据获取的代码实现

In [ ]:

*# coding = utf-8*

**import** **socket**

**import** **signal**

**import** **sys**

**import** **re**

*# 这里处理ctrl+c的信号（方便ctrl+c退出）*

**def** handle\_int(signum, handler):

print('程序中断退出，信号：**%d**' % signum, '处理器是：**{}**'.format(handler))

client\_socket.close()

sys.exit(0)

*# 绑定对ctrl+c信号的处理*

signal.signal(signalnum=signal.SIGINT, handler=handle\_int)

server\_name = 'www.huanqiu.com'

*# 获取IP地址（这个过程不是必须的）*

list\_info = socket.getaddrinfo(

host=server\_name,

port=80,

family=socket.AF\_INET,

type=socket.SOCK\_STREAM,

proto=socket.IPPROTO\_TCP,

flags=socket.AI\_ALL) *# socket.AI\_\*\*\*参数的一部分*

**for** \_, \_, \_, \_, address **in** list\_info:

print('IP：**%s**，PORT：**%d**' % address)

*# 创建socket*

client\_socket = socket.socket(

socket.AF\_INET, *# 网络地址族：常用的是internet网地址格式（IP地址）*

socket.SOCK\_STREAM, *# 网络通信方式：流与报文两种*

socket.IPPROTO\_TCP) *# 通信协议：数据包的格式*

*# 连接到到服务器（取上面取出的第一个INET地址）*

print(list\_info[0][-1])

client\_socket.connect(list\_info[0][-1]) *# 没有返回值，需要进行异常处理，这里容易被信号中断*

*# 发送数据*

request\_string = ''

request\_string += 'GET / HTTP/1.1**\r\n**'

request\_string += 'Host: **%s**:9999**\r\n**' % server\_name *# 可以替换成IP地址*

request\_string += 'Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8**\r\n**'

request\_string += 'Upgrade-Insecure-Requests: 1**\r\n**'

request\_string += 'Cookie: \_xsrf=2|f877d065|146c6a9838e67ba203776913fae34f45|1547796259**\r\n**'

request\_string += 'User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/12.0.2 Safari/605.1.15**\r\n**'

request\_string += 'Accept-Language: zh-cn**\r\n**'

*# request\_string += 'Accept-Encoding: gzip, deflate\r\n'*

request\_string += 'Connection: keep-alive**\r\n**'

request\_string += '**\r\n**'

request\_string += '**\r\n**'

bytes\_num = client\_socket.send(request\_string.encode('UTF-8'))

print('发送成功的字节数：**%d**' % bytes\_num)

*# 接收服务器响应的协议数据*

header\_buffers = b''

**while** **True**:

*# 一个子节一个子节读取*

buffer = client\_socket.recv(1, 0) *# MSG\_\*\*等*

header\_buffers += buffer

*# 判定最后四个子节是否是'\r\n\r\n'：表示响应头读完*

last\_four\_bytes = header\_buffers[-4:] *# 取最后四个子节*

**if** last\_four\_bytes == b'**\r\n\r\n**':

**break** *# header结束*

*# 把子节转换为字符串*

header\_string=header\_buffers.decode('utf-8')

print('响应协议头：**\n**', header\_string)

*# 获取请求体的长度*

regex = r'Content-Length: (\d\*?)\r\n'

len\_content = re.findall(regex, header\_string,re.MULTILINE)

len\_content = int(len\_content[0])

print('响应正文长度为：**{}**'.format(len\_content))

*# 下面读取响应正文*

body\_buffers = b''

**while** **True**:

buffer = client\_socket.recv(1024\*10, 0) *# MSG\_\*\*等*

body\_buffers += buffer

**if** len(body\_buffers) == len\_content:

**break**

print('响应体读取完毕！')

body\_string = body\_buffers.decode('utf-8')

*# print(body\_string)*

#### HTTP响应协议的格式说明

HTTP/1.1 200 OK Server: NWS\_TCloud\_S1 Connection: keep-alive Date: Sat, 19 Jan 2019 14:41:09 GMT Cache-Control: max-age=600 Expires: Sat, 19 Jan 2019 14:51:09 GMT Last-Modified: Sat, 19 Jan 2019 14:30:00 GMT Content-Type: text/html Content-Length: 145971 X-NWS-LOG-UUID: 1950543609143247235 07102197e13b4c6b15335e94859e887e X-Cache-Lookup: Hit From Disktank3 X-Daa-Tunnel: hop\_count=1 X-Cache-Lookup: Hit From Inner Cluster

HTTP响应协议也是由4各部分构成：

|- 响应行：HTTP/1.1 200 OK

|- 响应头：Server: NWS\_TCloud\_S1

|- 空行： 表示响应头结束

|- 响应体

与请求协议格式一样，每行结束都是使用'\r\n'表示。

1. 响应行
2. 响应行由三部分构成：
3. |- HTTP协议版本：HTTP/1.1
4. |- 状态码：200
5. |- 状态描述：OK

其中状态码也是HTTP协议规范的一部分。

1. 响应头
2. 响应头由多行构成，直到空行表示结束。每行由两个部分构成：
3. |- 响应头Key： 响应值说明
4. |- 响应头Value：响应值

响应头也是HTTP协议的一部分。

1. 空行

空行表示响应头结束。加上最后一个响应头末尾的'\r\n'，一般这个地方有两个'\r\n'，一般作为识别响应头结束的位置或者标记。

1. 响应体
2. 响应体就是响应的正文内容，一般是HTML页面，CSS，JS等，也可以是图片等。
3. 响应头的长度什么时候结束，一般都在响应头中说明，响应正文的长度使用Content\_Length说明：
4. |- Content-Length: 145971
5. 一般服务器为了传输的速度，都会响应体进行压缩，为了防止压缩，可以在请求的时候，使用请求头中说明，或者取消压缩相关的请求头说明，比如我们可以取消如下请求头，服务器就不会发送压缩数据。
6. |- Accept-Encoding: gzip, deflate # 在请求头就是表示支持gzip压缩或者deflate压缩格式，可以不指定这个请求，服务器就不在发送压缩数据。
7. 注意：

有的服务器因为数据太大，会采用分包的方式，分包格式采用独有的分包标记来实现。这里不详细介绍。大家今后碰见了稍微注意下。

#### HTTP响应码说明

HTTP响应码使用3位数表示

HTTP响应码分成5类：

|- 1xx ：报告的 -请求被接收到，继续处理

|- 2xx ：成功 - 被成功地接收（received），理解（understood），接受（accepted）的动作 。

|- 3xx ：重发 - 为了完成请求必须采取进一步的动作。

|- 4xx ：客户端出错 - 请求包括错的语法或不能被满足。

|- 5xx ：服务器出错 - 服务器无法完成显然有效的请求。

1. 1XX响应码
2. 100 继续

101 转换协议

1. 2XX响应码
2. 200 OK
3. 201 已创建
4. 202 接受
5. 203 非权威信息
6. 204 无内容
7. 205 重置内容

206 部分内容

1. 3XX响应码
2. 300 多个选择
3. 301 永久移动
4. 302 发现
5. 303 见其它
6. 304 没有被改变
7. 305 使用代理

307 临时重发

1. 4XX响应码
2. 400 坏请求
3. 401 未授权的
4. 402 必要的支付
5. 403 禁用
6. 404 没有找到
7. 405 方式不被允许
8. 406 不接受的
9. 407 需要代理验证
10. 408 请求超时
11. 409 冲突
12. 410 不存在
13. 411 长度必需
14. 412 先决条件失败
15. 413 请求实体太大
16. 414 请求URI太大
17. 415 不被支持的媒体类型
18. 416 请求的范围不满足

417 期望失败

1. 5XX响应码
2. 500 服务器内部错误
3. 501 不能实现
4. 502 坏网关
5. 503 服务不能获得
6. 504 网关超时

505 HTTP版本不支持

## HTTP常见头说明

### HTTP请求头说明

1. Accept

Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9, text/plain;q=0.8, video/x-mng,image/png,image/jpeg,image/gif;q=0.2, text/css,/;q=0.1 s

Accept消息报头：告诉服务器，请求者可以接受的内容类型与格式。

|- text/xml表示请求内容是text文本，子类型是xml格式。子类型使用+指定多个混合， \*/\*表示其他任意其他内容类型与格式， 每个类型之间使用逗号分隔。

|- 在内容类型与格式后可以使用分号隔开指定内容类型提供的优先级别q=优先级别，优先级别在0到1之间的数。字母q表示quality 品质。

其中类型的支持由浏览器决定。但是类型的定义由MIMETYPE定义有兴趣可以扩展MIME-TYPE的知识点。

MIME-TYPE的知识扩展可以参考：

|- http://www.w3school.com.cn/media/media\_mimeref.asp

不过常用的MIME-TYPE有：

|- Text

|- Image

|- Audio

|- Video

|- Application

1. Accept-Charset
2. 指定请求数据的字符编码

Accept-Charset: ISO-8859-1, utf-8;q=0.66, \*;q=0.66

对我们来说常见的字符编码：

|- GB2312：标准汉字编码规范

|- GBK：扩展汉字编码规范

|- UTF-8：是一种针对Unicode的可变长度字符编码，又称万国码

|- ISO-8859-1：西文编码

1. Accept-Encoding
2. 指定请求的数据压缩格式

Accept-Charset: ISO-8859-1, utf-8;q=0.66, \*;q=0.66

用于优化数据传输的大小。

可以指定的压缩编码格式：

identity：默认值，不采用压缩

gzip：用于UNIX系统的文件压缩，扩展名.gz

deflate： 同时使用了LZ77算法与哈夫曼编码的一个无损数据压缩算法

compress： 古老的 unix 档案压缩指令，压缩后的档案会加上 .Z扩展名。

1. Accept-Language
2. 告诉服务器，请求端可以使用的语言参考

Accept-Language: zh-CN

语言的格式：

语言-地区

对我们来说比较常用的语言：

en ：英文 en-US ：美国英语

zh ：汉语 zh-CN ：大陆地区汉语

1. Authorization/Proxy-Authorization

传递授权信息给服务器/代理服务器。

Authorization: Basic bXluYW1lOm15cGFzcw==

Basic表示授权方式，授权信息的加密算法是Base64，后面紧跟加密的信息。加密信息为name：password格式，使用冒号分隔授权名与授权密码。

常见加密算法：

|- Basic授权方式

|- Digest授权方式（容易暴露用户名）

Authorization: Digest username=“myuser”, //用户名

realm=“HTTP Developer‘s Handbook”, //提示领域

uri=“/”, //请求资源

nonce=“a4b8c8d7e0f6a7b2c3d2e4f5a4b7c5d2e7f”, //用来产生digest消息

response=“47d5aaf1b20e5b3483901267a3944737” //加密信息

1. Cookie

传递非用户的数据，用来管理HTTP状态

Cookie: fname=chris; lname=shiflett

Cookie的值使用分号分隔，每个值是name：value的值对

1. Expect

用来告诉服务器，是否需要继续传递消息主体

Expect:100-Continue

目前Expect的值只能是100-continue，或者其他扩展。这个选项用来提升访问性能，用于客户端判定服务器是否愿意接受客户端发来的消息主体，（在有些情况下，如果服务器拒绝查看消息主体，这时客户端发送消息主体是不合适的或会降低效率。 ）

注意：

如果服务器不能满足任何expectation值，服务器必须以417（期望失败）状态码响应，或者如果服务器满足请求时遇到其它问题，服务器必须发送4xx状态码。

该HEAD在有的代理服务器没有实现，会导致显示不正常，所有选择WEB服务器的一个很重要的测试，就是测试服务器是否支持Expect头。

1. From
2. 用来转发一个电子邮件地址：

From: qiang.yang@oracle.com

现在很少有WEB服务器支持该HEAD。

1. Host
2. 用来提供多寻址方式

Host: www.google.com：80

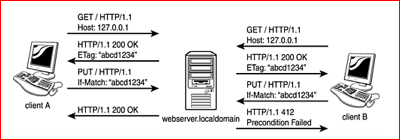
拥有一个IP地址对应有多个主机名服务器，允许源服务器或网关去区分有内在歧义的URLS。

端口使用冒号分隔，默认端口80可以省略。

1. If-Match/ If-None-Match
2. 发送一个实体与服务器当前实体比较

If-Match: "xyzzy"

设置条件请求。如果一个客户端已经从一个资源获得一个或多个实体（entity），那么它可以通过在If-Match头域里包含相应的实体标签（entity tag）来验证实体是否就是服务器当前实体。与响应的Etag有关。



A客户端先发送IF-match，服务器匹配后修改实体资源，B客户端发If-match就无法匹配，导致412响应

1. If-Modified-Since /If-Unmodified-Since
2. 指定最后修改的时间

If-Modified-Since: Fri, 12 May 2013 18:53:33 GMT

由客户端往服务器发送的头，可以看到，再次请求本地存在的 cache 页面时，客户端会通过 If-Modified-Since 头将先前服务器端发过来的 Last-Modified 最后修改时间戳发送回去，这是为了让服务器端进行验证，通过这个时间戳判断客户端的页面是否是最新的，如果不是最新的，则返回新的内容，如果是最新的，则 返回 304 告诉客户端其本地 cache 的页面是最新的，于是客户端就可以直接从本地加载页面了，这样在网络上传输的数据就会大大减少，同时也减轻了服务器的负担。

1. If-Range
2. 条件请求部分数据重传，条件是实体匹配或者时间匹配。
3. If-Range: "df6b0-b4a-3be1b5e1"

If-Range: Tue, 21 May 2002 12:34:56 GMT

IF-Range头部需配合Range，如果没有Range参数，则If-Range会被视为无效。

如果If-Range匹配上，那么客户端已经存在的部分是有效的，服务器将返回缺失部分，也就是Range里指定的，然后返回206（Partial content)，否则证明客户端的部分已无效（可能已经更改），那么服务器将整个实体内容全部返回给客户端，同时返回200OK

1. Range ：
2. 用来指定部分数据请求

Range: bytes 0-499, 1000-1499，1500-

其中bytes表示单位，目前只能是bytes。

其他例子说明：假设请求10000个字节数据。

-- 第一个500字节: bytes=0-499

-- 第二个500字节: bytes=500-999

-- 最后500字节: bytes=-500 或 bytes=9500-

-- 仅仅第一个和最后一个字节: bytes=0-0,-1

1. Max-Forwards ：
2. 为TRACE和OPTIONS提供一种机制去限制转发请求的代理或网关的数量。

用来解决代理服务器阻止返回的响应问题。

Max-Forwards: 0

1. Referer :
2. 告诉服务器我是从哪个页面链接过来的，服务器籍此可以获得一些信息用于处理。

指定请求URI来源的资源URI允许服务器为了个人兴趣，记录日志，优化缓存等来产生回退链接列表。它照样允许服务器为维护而跟踪过时或写错的链接。

Referer: http://www.w3.org/hypertext/DataSources/Overview.html

1. TE：

使用语法与Accept-Encoding与Content-Encoding一样，但该HEADER用于传输的编码指定。

1. User-Agent：

用来指定用户的代理请求工具。服务器可以根据这个特性来提供更适合客户的响应。

User-Agent: Mozilla/5.0 (X11; U; Linux i686; zh-CN; rv:1.9.2.13)

### HTTP请求与响应都使用的头说明

1. Connection：
2. 用于服务器对客户端连接的管理
3. Connect常用的值
4. |- Keep-Alive：保持连接。
5. |- Close：在传输完成后关闭连接。

|- Upgrade：用于代理服务器更新响应。

Connection: Keep-Alive

Connection: Close

1. Keep-Alive：
2. 这个HEAD可能今后不被推荐使用。用于说明保持连接的时间，单位是秒数。

这个HEAD的值没有明确定义，有的服务器用来表示TCP连接打开的最大数量。

Keep-Alive：300

1. Cache-Control

用来管理服务器的缓冲，包含代理服务器。

Cache-Control: max-age=600, no-cache="Set-Cookie"

其中包含的设置项有：

|- no-cache：不是用来禁止缓冲，而是用来重新校验缓冲的回送给客户的数据。如果no-cache缓存控制指令没有指定一个field-name，那么一个缓存不能在没有通过源服务器对它进行成功重验证的情况下，利用此响应去满足后续的请求。这允许源服务器去防止响应被缓存保存，即使此缓存已被设置可以返回陈旧响应给客户端。

|- no-store：no-store缓存控制指令的目的在于防止不经意地释放或保留敏感信息

|- max-age：值为秒数，表示请求时间长度不超过max-age的缓存。

|- max-stale=600 ：指定缓存的过期时间。

|- min-fresh=600：表示请求时间长度超过min-fresh的缓存。

|- no-transform：告知服务器不要修改响应的缓存数据。

|- only-if-cached：告知服务器如果缓存存在，则直接使用缓存。

|- cache-extension：缓存扩展。代理服务器不支持会直接丢弃该HEAD。

1. Date
2. 指定消息产生的时间。

注意：时间格式必须按照Tue, 21 May 2002 12:34:56 GMT 指定。

Date: Tue, 21 May 2002 12:34:56 GMT

1. Pragma

在HTTP/1.0用来控制缓冲，在HTTP1.1中使用Cache-Control。

Pragma: no-cache

1. Trailer

用来确定在消息正文后，添加HEAD的值。

Trailer: Date

产生的效果如下，必须与Transfer-Encoding：chunked使用。

HTTP/1.1 200 OK

Content-Type: text/html

Transfer-Encoding: chunked

Trailer: Date

7f

<html>

</html>

0

Date: Tue, 21 May 2002 12:34:56 GMT

1. Transfer-Encoding：
2. Transfer-Encoding: chunked 表示输出的内容长度不能确定，普通的静态页面、图片之类的基本上都用不到这个。但动态页面就有可能会用到。

使用这个HEAD表示数据的开始行不是数据，而是表示数据开始，数据结束使用行内容为0的单独行。

Transfer-Encoding: chunked

1. Upgrade：

用来告知服务器与客户端协议或者协议版本的变化。响应会发送101 Switching Protocols 响应状态。

Upgrade: HTTP/2.0, HTTPS/1.3, IRC/6.9, RTA/x11, websocket

1. Via：

被代理服务器使用，用来标识自己的信息，与User-Agent一样。

Via: 1.0 fred, 1.1 example.com (Apache/1.1)

1. Warning

实体可能会发生的问题的通用警告

Warning: 199 Miscellaneous warning

### HTTP非标准请求头

1. X-Requested-With

标识Ajax请求，大部分js框架发送请求时都会设置它为XMLHttpRequest

X-Requested-With: XMLHttpRequest

1. DNT

请求web应用禁用用户追踪

DNT: 1 (Do Not Track Enabled)

DNT: 0 (Do Not Track Disabled)

1. X-Forwarded-For

一个事实标准，用来标识客户端通过HTTP代理或者负载均衡器连接的web服务器的原始IP地址

X-Forwarded-For: client1, proxy1, proxy2

X-Forwarded-For: 129.78.138.66, 129.78.64.103

1. X-Forwarded-Host

一个事实标准，用来标识客户端在HTTP请求头中请求的原始host,因为主机名或者反向代理的端口可能与处理请求的原始服务器不同

X-Forwarded-Host: en.wikipedia.org:8080

X-Forwarded-Host: en.wikipedia.org

1. X-Forwarded-Proto

一个事实标准，用来标识HTTP原始协议，因为反向代理或者负载均衡器和web服务器可能使用http,但是请求到反向代理使用的是https

X-Forwarded-Proto: https

1. Front-End-Https

微软应用程序和负载均衡器使用的非标准header字段 Front-End-Https: on

1. X-Http-Method-Override

请求web应用时，使用header字段中给定的方法（通常是put或者delete）覆盖请求中指定的方法（通常是post）,如果用户代理或者防火墙不支持直接使用put或者delete方法发送请求时，可以使用这个字段

X-HTTP-Method-Override: DELETE

1. X-ATT-DeviceId

允许更简单的解析用户代理在AT&T设备上的MakeModel/Firmware

X-Att-Deviceid: GT-P7320/P7320XXLPG

1. X-Wap-Profile

设置描述当前连接设备的详细信息的xml文件在网络中的位置

x-wap-profile: http://wap.samsungmobile.com/uaprof/SGH-I777.xml

1. Proxy-Connection

早起HTTP版本中的一个误称，现在使用标准的connection字段

Proxy-Connection: keep-alive

11.X-UIDH

服务端深度包检测插入的一个唯一ID标识Verizon Wireless的客户

X-UIDH: ...

1. X-Csrf-Token,X-CSRFToken,X-XSRF-TOKEN

防止跨站请求伪造

X-Csrf-Token: i8XNjC4b8KVok4uw5RftR38Wgp2BFwql

1. X-Request-ID,X-Correlation-ID

标识客户端和服务端的HTTP请求

X-Request-ID: f058ebd6-02f7-4d3f-942e-904344e8cde5

### HTTP通用响应头说明

1. Accept-Ranges
2. 说明服务器是否支持指定范围的数据重发或者发送。
3. HEAD的值说明：
4. |- bytes表示支持

|- none表示不支持。

Accept-Ranges: bytes

Accept-Ranges: none

1. Age
2. 说明用户请求资源的存在年龄。

值的单位为秒，使用4字节整数表示，注意不要溢出。

Age：2500

1. Authentication-Info

包含用户的授权信息

HTTP/1.1 200 OK

Authentication-Info: qop="auth-int", rspauth="5913ebca817739aebd2655bcfb952d52",

cnonce="f5e2d7c0b6a7f2e3d2c4b5a4f7e4d8c8b7a", nc="00000001"

1. Content-Disposition：
2. Content-disposition 是 MIME 协议的扩展，MIME 协议指示 MIME 用户代理如何显示附加的文件。Content-disposition其实可以控制用户请求所得的内容存为一个文件的时候提供一个默认的文件名，文件直接在浏览器上显示或者在访问时弹出文件下载对话框。
3. 可以解决如下问题：
4. |- 希望某类或者某已知MIME 类型的文件（比如：\*.gif;\*.txt;\*.htm）能够在访问时弹出“文件下载”对话框。
5. |- 希望客户端下载时以指定文件名显示。

|- 希望某文件直接在浏览器上显示而不是弹出文件下载对话框。

Content-Disposition: attachment; filename="example.pdf"

1. Etag
2. 提供一个值唯一指定请求资源的版本(分块发送的资源边界标识)

客户一般用来重发给服务器，用来请求对缓存数据的状态管理。

ETag: "1cdb-4efedbb8"

1. Location：
2. 告知浏览器重新请求的位置。
3. 用于为了完成请求或识别一个新资源，使接收者能重定向于Location指明的URI而不是请求URI。

其中必须使用绝对位置。

Location: http://httphandbook.org/

1. Proxy-Authenticate/WWW-Authenticate
2. 要求客户提供授权信息
3. |- Basic是加密算法

|- Realm指定提示域信息。

WWW-Authenticate: Basic realm="testrealm@keakon.cn"

1. Refresh
2. 告知浏览器刷新的参数
3. 等价于

<meta http-equiv="refresh" content="3; url=http://httphandbook.org/">

Refresh: 3; url=http://httphandbook.org/

1. Retry-After

指定重试时间间隔或者时间

Retry-After: Tue, 21 May 2002 12:34:56 GMT

Retry-After: 600

1. Server

指定服务器相关信息

Server: Apache

1. Set-Cookie
2. 用来指示客户端浏览器存储数据到cookie。
3. Cookie格式说明：
4. fname=chris ：存储文件名
5. domain=httphandbook.org ：存储的域
6. path=/ ：存储路径
7. expires=Tue, 21 May 2002 12:34:56 GMT：失效时间

Secure ：表明使用安全连接发送，比如SSL。

Set-Cookie: fname=chris; domain=.httphandbook.org; path=/; expires=Tue, 21 May 2002 12:34:56 GMT; secure

1. Vary：
2. vary的意义在于告诉代理服务器缓存，如何判断请求是否一样，vary中的组合就是服务器判断的依据，

比如Vary中有User-Agent，那么即使相同的请求，如果用户使用IE打开了一个页面，再用Firefox打开这个页面的时候，CDN/代理会认为是不同的页面，如果Vary中没有User-Agent，那么CDN/代理会认为是相同的页面，直接给用户返回缓存的页面，而不会再去web服务器请求相应的页面。

Vary: Accept-Language, User-Agent

### HTTP用于响应体的头说明

1. Allow：
2. 对请求而言，是告诉服务器只能使用指定的方法请求。

对响应而言，是告诉浏览器，服务只能支持的方法。

Allow: GET, HEAD, POST

1. Content-Encoding
2. 指定实体内容的压缩编码格式：
3. 一般编码格式是：
4. |- gzip
5. |- compress

|- deflate

Content-Encoding：gzip,deflate,compress

1. Content-Language

实体内容支持的语言。

1. Content-Length

实体内容的长度

1. Content-Location

指定资源的位置

1. Content-MD5

指定实体内容的MD5信息

Content-MD5: ZTFmZDA5MDYyYTMzZGQzMDMxMmIxMjc4YThhNTMyM2I=

1. Content-Range

服务器用来返回请求资源的一部分。

Content-Range: 600-900/1234

说明返回总的1234字节的600-900字节。

1. Content-Type

指定内容实体的文件格式/子格式。

Content-Type: text/html

1. Expires

指定内容实体过期的时间

Expires: Tue, 21 May 2002 12:34:56 GMT

1. Last-Modified

资源最后修改的时间，往往用来计算资源的Age。

Last-Modified: Tue, 21 May 2002 12:34:56 GMT

### HTTP非标准响应头

1. X-XSS-Protection

过滤跨站脚本

X-XSS-Protection: 1; mode=block

1. Content-Security-Policy, X-Content-Security-Policy,X-WebKit-CSP
2. 定义内容安全策略

X-WebKit-CSP: default-src 'self'

1. X-Content-Type-Options

唯一的取值是"",阻止IE在响应中嗅探定义的内容格式以外的其他MIME格式

X-Content-Type-Options: nosniff

1. X-Powered-By
2. 指定支持web应用的技术

X-Powered-By: PHP/5.4.0

1. X-UA-Compatible
2. 推荐首选的渲染引擎来展示内容，通常向后兼容，也用于激活IE中内嵌chrome框架插件

<meta http-equiv="X-UA-Compatible" content="chrome=1" />

X-UA-Compatible: IE=EmulateIE7

X-UA-Compatible: IE=edge

X-UA-Compatible: Chrome=1

1. X-Content-Duration

提供音视频的持续时间，单位是秒，只有Gecko内核浏览器支持

X-Content-Duration: 42.666

1. Upgrade-Insecure-Requests

标识服务器是否可以处理HTTPS协议

Upgrade-Insecure-Requests: 1

1. X-Request-ID,X-Correlation-ID

标识一个客户端和服务端的请求

X-Request-ID: f058ebd6-02f7-4d3f-942e-904344e8cde5

### 响应头的使用例子：

#### 要求浏览器使用basic登录的例子

这里使用响应头，发送一个用户需要登录的协议沟通：

"HTTP/1.1 401 Unauthorized\r\n"

"WWW-Authenticate: Basic realm=\"Louis Young Examples for Response!\"\r\n"

"Connection: Keep-Alive\r\n"

"Keep-Alive: 115\r\n"

"\r\n\r\n");

代码略

#### 获取浏览器的发送的cookie

代码略

#### 获取服务器发送的set-cookie

代码略

## HTTP Cookie与Session

### Cookie的格式

注意： 在HTTP协议中，一般Cookie支持大小最多4K所以使用Cookie处理数据存在局限性。

一般Cookie采用明文信息，所以极度不安全。一般浏览器中可以设置是否存储Cookie。

#### 几个浏览器发送的cookie例子

有的站点在访问时，浏览器不会发起cookie。

1. 自己写的程序抓取到的cookie

Cookie: \_xsrf=2|f877d065|146c6a9838e67ba203776913fae34f45|1547796259

1. 使用浏览器向百度翻译请求时发送的cookie

Cookie: Hm\_lpvt\_64ecd82404c51e03dc91cb9e8c025574=1547803048; Hm\_lvt\_64ecd82404c51e03dc91cb9e8c025574=1547545982,1547720682,1547775577,1547803048; from\_lang\_often=%5B%7B%22value%22%3A%22fra%22%2C%22text%22%3A%22%u6CD5%u8BED%22%7D%2C%7B%22value%22%3A%22zh%22%2C%22text%22%3A%22%u4E2D%u6587%22%7D%2C%7B%22value%22%3A%22en%22%2C%22text%22%3A%22%u82F1%u8BED%22%7D%5D; to\_lang\_often=%5B%7B%22value%22%3A%22en%22%2C%22text%22%3A%22%u82F1%u8BED%22%7D%2C%7B%22value%22%3A%22zh%22%2C%22text%22%3A%22%u4E2D%u6587%22%7D%5D; FANYI\_WORD\_SWITCH=1; HISTORY\_SWITCH=1; REALTIME\_TRANS\_SWITCH=1; SOUND\_PREFER\_SWITCH=1; SOUND\_SPD\_SWITCH=1; locale=zh; BDORZ=B490B5EBF6F3CD402E515D22BCDA1598; H\_PS\_PSSID=1430\_21119\_28329\_28132\_26350\_28267\_27245\_22158; PSINO=1; delPer=0; pgv\_pvi=5448934400; pgv\_si=s1851237376; BDUSS=hsVWV6czh0a1hOQ3BaYkhTM0FrOXhNYnBCUWFsMlY0clhlYkNvTkRKdENDUGxiQUFBQUFBJCQAAAAAAAAAAAEAAAAFyMhmAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAEJ70VtCe9FbY0; BIDUPSID=C53590ACE23DAC88DBE0C3D65AEBAA30; PSTM=1539535646; BAIDUID=DB00283B42FBC875B67496A00F47ABAB:FG=1

使用浏览器解析出来是一个表格：

| **cookie名称** | **Cookie值** |
| --- | --- |
| BAIDUID | DB00283B42FBC875B67496A00F47ABAB:FG=1 |
| BDORZ | B490B5EBF6F3CD402E515D22BCDA1598 |
| BDUSS | hsVWV6czh0a1hOQ3BaYkhTM0FrOXhNYnBCUWFsMlY0clhlYkNvTkRKdENDUGxiQUFBQUFBJCQAAAAAAAAAAAEAAAAFyMhmAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAEJ70VtCe9FbY0 |
| BIDUPSID | C53590ACE23DAC88DBE0C3D65AEBAA30 |
| delPer | 0 |
| FANYI\_WORD\_SWITCH | 1 |
| from\_lang\_often | %5B%7B%22value%22%3A%22fra%22%2C%22text%22%3A%22%u6CD5%u8BED%22%7D%2C%7B%22value%22%3A%22zh%22%2C%22text%22%3A%22%u4E2D%u6587%22%7D%2C%7B%22value%22%3A%22en%22%2C%22text%22%3A%22%u82F1%u8BED%22%7D%5D |
| H\_PS\_PSSID | 1430\_21119\_28329\_28132\_26350\_28267\_27245\_22158 |
| HISTORY\_SWITCH | 1 |
| Hm\_lpvt\_64ecd82404c51e03dc91cb9e8c025574 | 1547972748 |
| Hm\_lvt\_64ecd82404c51e03dc91cb9e8c025574 | 1547545982,1547720682,1547775577,1547803048 |
| locale | zh |
| pgv\_pvi | 5448934400 |
| pgv\_si | s1851237376 |
| PSINO | 1 |
| PSTM | 1539535646 |
| REALTIME\_TRANS\_SWITCH | 1 |
| SOUND\_PREFER\_SWITCH | 1 |
| SOUND\_SPD\_SWITCH | 1 |
| to\_lang\_often | %5B%7B%22value%22%3A%22en%22%2C%22text%22%3A%22%u82F1%u8BED%22%7D%2C%7B%22value%22%3A%22zh%22%2C%22text%22%3A%22%u4E2D%u6587%22%7D%5D |

1. 抓取到百度首页响应的Cookie如下：

HTTP/1.1 200 OK

Content-Type: text/html;charset=utf-8

Set-Cookie: BD\_HOME=1; path=/

Set-Cookie: BDSVRTM=321; path=/

Set-Cookie: H\_PS\_PSSID=1430\_21119\_28206\_28132\_26350\_28267\_27245\_22158; path=/; domain=.baidu.com

Content-Encoding: gzip

Expires: Sun, 20 Jan 2019 08:36:16 GMT

Transfer-Encoding: Identity

Cache-Control: private

Date: Sun, 20 Jan 2019 08:36:16 GMT

Connection: Keep-Alive

Server: BWS/1.1

Bdqid: 0x8878399100134524

Bdpagetype: 2

Strict-Transport-Security: max-age=172800

X-Ua-Compatible: IE=Edge,chrome=1

其中Cookie是：

Set-Cookie: BD\_HOME=1; path=/

Set-Cookie: BDSVRTM=321; path=/

Set-Cookie: H\_PS\_PSSID=1430\_21119\_28206\_28132\_26350\_28267\_27245\_22158; path=/; domain=.baidu.com

#### Cookie的格式

Cookie就是使用Key:Value格式构成的值对。

其中的Key有通用约定用途的Key，也可以用户自己定义的Key，因为Cookie既用于浏览器与Web服务器之间数据约定，也用于用户的数据约定。

Cookie约定的Key有：

Set-Cookie: <name>=<value>[; <name>=<value>]...

[; expires=<date>][; domain=<domain\_name>]

[; path=<some\_path>][; secure][; httponly]

下面是百度的Cookie的例子：

Set-Cookie: BAIDUID=801BAD1D5CB256E7448139623D0AE089:FG=1; expires=Thu, 31-Dec-37 23:55:55 GMT; max-age=2147483647; path=/; domain=.baidu.com Set-Cookie: BIDUPSID=801BAD1D5CB256E7448139623D0AE089; expires=Thu, 31-Dec-37 23:55:55 GMT; max-age=2147483647; path=/; domain=.baidu.com Set-Cookie: PSTM=1547976171; expires=Thu, 31-Dec-37 23:55:55 GMT; max-age=2147483647; path=/; domain=.baidu.com Set-Cookie: BD\_LAST\_QID=15551781544627683897; path=/; Max-Age=1

1. name=value

指定cookie的名称和值，名称大小写不敏感，值必须经过URL编码。

1. domain=
2. 域，表示当前cookie所属于哪个域或子域下面。
3. 对于服务器返回的Set-Cookie中，如果没有指定Domain的值，那么其Domain的值是默认为当前所提交的http的请求所对应的主域名的。
4. 比如访问 http://www.baidu.com，返回一个cookie，没有指名domain值，那么其为值为默认的www.baidu.com。
5. 注：
6. |- 临时cookie(没有expires参数的cookie)不能带有domain选项。
7. |- 当客户端发送一个http请求时，会将有效的cookie一起发送给服务器。

|- 如果一个cookie的domain和path参数和URL匹配，那么这个cookie就是有效的。一个URL中包含有domain和path

1. path=

表示cookie的存储的路径。

1. secure

表示该cookie只能用https传输。一般用于包含认证信息的cookie，要求传输此cookie的时候，必须用https传输。

1. httponly :
2. 表示cookie不能被客户端脚本获取到。

表示此cookie必须用于http或https传输。这意味着，浏览器脚本，比如javascript中，是不允许访问操作此cookie的。

6.expires=:

如果cookie超过date所表示的日期时，cookie将失效。

如果没有设置这个选项，那么cookie将在浏览器关闭时失效。

注意：date是格林威治时间(GMT)，使用如下格式表示：

DAY, DD MMM YYYY HH:MM:SS GMT

|- DAY：星期 (Sun, Mon, Tue, Wed, Thu, Fri, Sat)

|- DD：每个月的天数（从 01 到每个月最大天数)

|- MMM：月份（Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec)

|- YYYY：年（2019）

|- HH：小时（24小时制：22 表示 10:00 P.M.)

|- MM：分钟（00-59）

|- SS：秒：（00-59）

1. max-age=

用max-age指定当前cookie是在多长时间之后而失效（单位是秒数，从请求开始计时）

### 发送Cookie

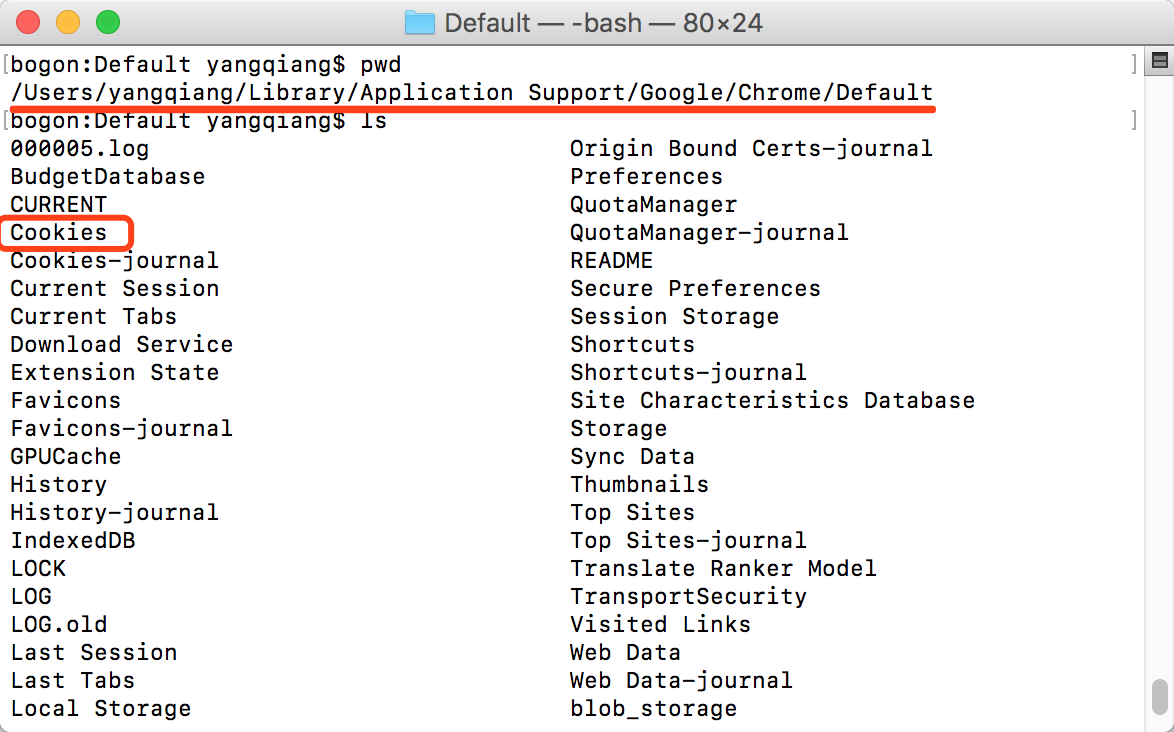
### 获取Cookie

### 浏览器中Cookie的存放

1. Chrome的存放位置
2. Mac OS X: ~/Library/Application Support/Google/Chrome/Default
3. Windows XP: Documents and SettingsusernameLocal SettingsApplication DataGoogleChromeUser DataDefault
4. Windows 7: C:\Users\XXXX\AppData\Local\Google\Chrome\User Data\Default\

Linux: ~/.config/google-chrome/Default

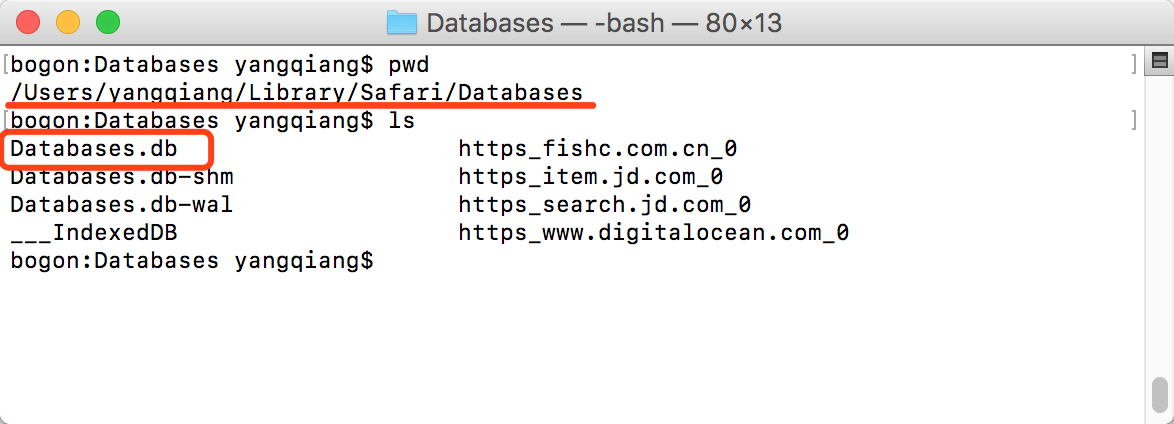
在Mac下的截图：



1. Safari的存放位置
2. Mac : /Users/yangqiang/Library/Safari/Databases

注意：Safari的Cookie保存采用的是Sqlite数据保存。

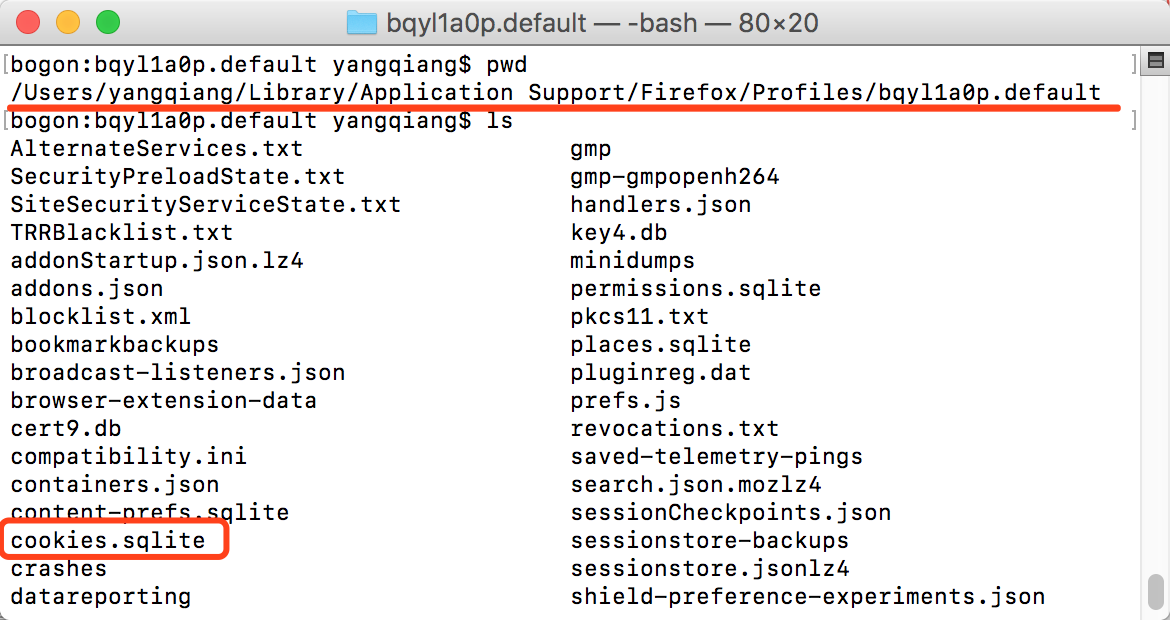
在Mac下截图



1. Firefox的存放位置

/Users/yangqiang/Library/Application Support/Firefox/Profiles/bqyl1a0p.default

在Mac下截图



### Session机制

HTTP协议本身是无状态的，客户端的每一次请求之间都是独立的，因为无论是客户端还是服务器都没有必要纪录彼此过去的行为。

然而随着互联网技术的发展，很快这种无状态协议满足不了应用需求：

|- DHTML技术开始出现（Javascript + DOM ）

|- 服务器端则出现了CGI规范以响应客户端的动态请求

|- cookie的出现解决HTTP协议无状态的缺陷。

|- session出现提供客户端与服务器之间保持状态的最终解决方案。

#### Session的作用

Session是用来识别用户状态的一种机制或者解决方案。

Session包含两个部分：

|- 状态存储

|- 状态数据

#### Sesson状态存储

Session机制是一种服务器端的机制，在服务器就是一个类似Hash的结构，这种类似散列表的结构来保存用户ID信息。

#### Session状态数据

|

|- 当客户端第一次请求的时候，服务器没有该客户状态，则为此客户端生成一个Session ID（该ID与客户端唯一绑定），并且创建一条存储纪录；

|

|- 当客户端再次访问，服务器首先检查这个客户端的请求里是否已包含了一个Session ID，从而识别用户状态（比如用户是否登录等），用户状态就是绑定在ID上的数据（这种数据通常使用Key-Value的方式存放，就是Hash表结构）

# 模块urllib 一个基于HTTP访问请求的Python模块库

### Urllib模块库是python内置的HTTP请求库，无需通过pip 网络安装下载，仅需使用前使用 import 导入即可。

1. urllib — URL处理模块

2. urllib.request — 请求模块

3. urllib.response — 响应模块

4. urllib.parse — URL解析模块（处理URL编码）

5. urllib.error — 定义urllib.request产生的异常

6. urllib.robotparser — 解析网站的robots.txt文件，这个文件中的URL一般不建议爬取

由于有上面的基础，这个部分的模块直接按照功能结构解释说明

In [ ]:

**import** **urllib**

print(dir(urllib))

## URL解析

URL解析的功能由urllib.parse模块完成

## URL请求

### urllib.request API结构

整个模块核心是完成一次HTTP请求：

|-请求发起

|-请求处理中数据处理

|- 包含Cookie的处理。

请求的发起

urllib.request.urlopen函数

返回一个响应。

请求数据表示采用两种方式：

|- URL字符串

|- Request类封装的数据

urllib.request模块的核心是Request类以及各种类型的请求处理器：

'AbstractBasicAuthHandler',

'AbstractDigestAuthHandler',

'AbstractHTTPHandler',

'BaseHandler',

'CacheFTPHandler',

'DataHandler',

'FTPHandler',

'FileHandler',

'HTTPBasicAuthHandler',

'HTTPDefaultErrorHandler',

'HTTPDigestAuthHandler',

'HTTPHandler',

'HTTPPasswordMgr',

'HTTPRedirectHandler',

'HTTPSHandler',

'ProxyBasicAuthHandler',

'ProxyDigestAuthHandler',

'ProxyHandler',

'UnknownHandler',

上述的处理器提供两个函数，把处理器与请求过程绑定在一起：

install\_opener

build\_opener

同时提供两个处理器：

'HTTPCookieProcessor',：处理Cookie

'HTTPErrorProcessor',：处理错误

URL打开与定向

'URLopener',

'OpenerDirector',

### urllib.response API结构

urllib.response模块中定义的类都在urllib.request中使用。主要类包含：

|- addbase

|- addclosehook

|- addinfo

|- addinfourl

其中提供info()返回headers，提供geturl()返回url

### urllib.request与urllib.response常见应用

#### 使用urlopen函数实现HTTP请求

使用URL字符串实现请求

In [ ]:

*# 内置模块*

**import** **urllib.request**

*# url字符串*

url = 'http://www.baidu.com'

*# 默认的都是GET方法，所以不需要data参数，对一般的请求，也不需要使用证书。*

response = urllib.request.urlopen(url=url)

*# print(type(response))*

*# 返回的是http.client.HTTPResponse对象。*

*# 头*

*# print(response.headers)*

*# 额外扩展的函数*

*# print(response.info())*

print(response.geturl())

*# 响应行*

print(response.status)

print(response.reason)

*# 读取内容*

print(response.read())

#### 使用Request类实现HTTP请求

使用Request类构造一个请求

In [ ]:

**import** **urllib.request**

url = 'http://www.baidu.com'

request = urllib.request.Request(url=url, data='hello'.encode('utf-8'), method='GET')

*# 默认的都是GET方法，所以不需要data参数，对一般的请求，也不需要使用证书。*

response = urllib.request.urlopen(url=request,data='world'.encode('utf-8'))

print(type(response))

*# 返回的是http.client.HTTPResponse对象。*

*# 头*

*# print(response.headers)*

*# # 响应行*

print(response.status)

print(response.reason)

*# 读取内容*

print(response.read())

*# 注意：urlopen中的data会覆盖Request中定义的data*

# 模块requests

\* requests是第三方模块。

\* Requests 是用Python语言编写，基于 urllib，采用 Apache2 Licensed 开源协议的 HTTP 库。

\* 比urllib方便，快速，完全满足htpp访问的需求

\* 完美支持Python3.X

* Beautiful is better than ugly.(美丽优于丑陋)
* Explicit is better than implicit.(清楚优于含糊)
* Simple is better than complex.(简单优于复杂)
* Complex is better than complicated.(复杂优于繁琐)
* Readability counts.(重要的是可读性)

pip instal requests import request

In [ ]:

**import** **requests**

print(dir(requests))

## requests模块API结构

requests模块提供三个核心类负责HTTP请求与响应处理：

| - Request封装请求数据

| - Response封装响应数据

| - Session管理完整的请求会话 ,发起请求，返回响应

还有其他异常结构设计，以及Timeout延时处理。

同时提供一组快捷调用函数。

### Request类

Request类封装请求数据。

构造器如下：

|- def \_\_init\_\_(self, method=None, url=None, headers=None, files=None, data=None,

params=None, auth=None, cookies=None, hooks=None, json=None):

参数的说明如下：（来自help输出的帮助）

:param method: HTTP method to use.

:param url: URL to send.

:param headers: dictionary of headers to send.

:param files: dictionary of {filename: fileobject} files to multipart upload.

:param data: the body to attach to the request. If a dictionary is provided, form-encoding will take place.

:param json: json for the body to attach to the request (if files or data is not specified).

:param params: dictionary of URL parameters to append to the URL.

:param auth: Auth handler or (user, pass) tuple.

:param cookies: dictionary or CookieJar of cookies to attach to this request.

:param hooks: dictionary of callback hooks, for internal usage.

其中files，data，json是三种数据方式。

### Response类

返回请求的响应，主要有三个数据是比较关注的：

|

| - 响应体数据：

|

| - 响应行

|

| - 响应头

响应体数据获取（以下成员都可以获取）：

| - content

| - text

| - iter\_content

| - iter\_lines

| - json

响应行

| - status\_code属性

响应头

| - headers

其他：

| - url （访问的URL）

| - history （访问历史）

| - ok （状态码是否是200）

| - next （是否需要重定向）

| - links （头中连接字符串）

### Session类

提供Cookie的持久，连接池与配置处理；其核心方法只有一个，发起请求：

|-request(self, method, url, params=None, data=None, headers=None, cookies=None,

files=None, auth=None, timeout=None, allow\_redirects=True,

proxies=None, hooks=None, stream=None, verify=None, cert=None, json=None)

争对不同的请求方法提供单独的函数提交请求：

|- delete

|- get

|- head

|- options

|- patch

|- post

|- put

request的实现实际是两个过程完成：

|- prepare\_request(self, request) 构造一个PreparedRequest

|- send(self, request, \*\*kwargs) 发送一个PreparedRequest

其中的PreparedRequest类与Request类负责请求信息

Request提供prepare(self)函数把Request转换为PreparedRequest。

PreparedRequest是发送到服务器的请求

In [ ]:

*# coding = utf-8*

**import** **requests**

session = requests.Session()

response = session.request(

method='GET',

url='http://www.baidu.com',

params={'name': 'louis','age': 20})

*# print(response.content.decode('utf-8'))*

print(response.status\_code)

print(response.headers)

print(response.history)

# GET方法

In [ ]:

res = requests.get('https://api.github.com/events')

In [ ]:

res.text

In [ ]:

res.content

In [ ]:

res.url

In [ ]:

res.encoding

In [ ]:

res.json()

### 带参数的GET

In [ ]:

payload = {'key1': 'value1', 'key2': 'value2'}

res = requests.get("http://httpbin.org/get", params=payload)

res.url

### 定制请求头

如果你想为请求添加 HTTP 头部，只要简单地传递一个 dict 给 headers 参数就可以了。 例如，在前一个示例中我们没有指定 content-type:

In [ ]:

url = 'https://api.github.com/some/endpoint'

headers = {'user-agent': 'my-app/0.0.1'}

r = requests.get(url,headers = headers)

# POST方法

In [ ]:

r = requests.post('http://httpbin.org/post', data = {'key':'value'})

r.url

In [ ]:

# bs4模块

* bs4是在爬取网页，解析页面使用比较多的技术之一。
* 使用这个模块，需要额外安装模块
  + 安装指令：'pip install bs4'
* bs4的帮助
  + 在python帮助环境下，使用import与help可以得到全部文档。

Last login: Fri Mar 15 16:09:43 on ttys000

localhost:~ yangqiang$ python

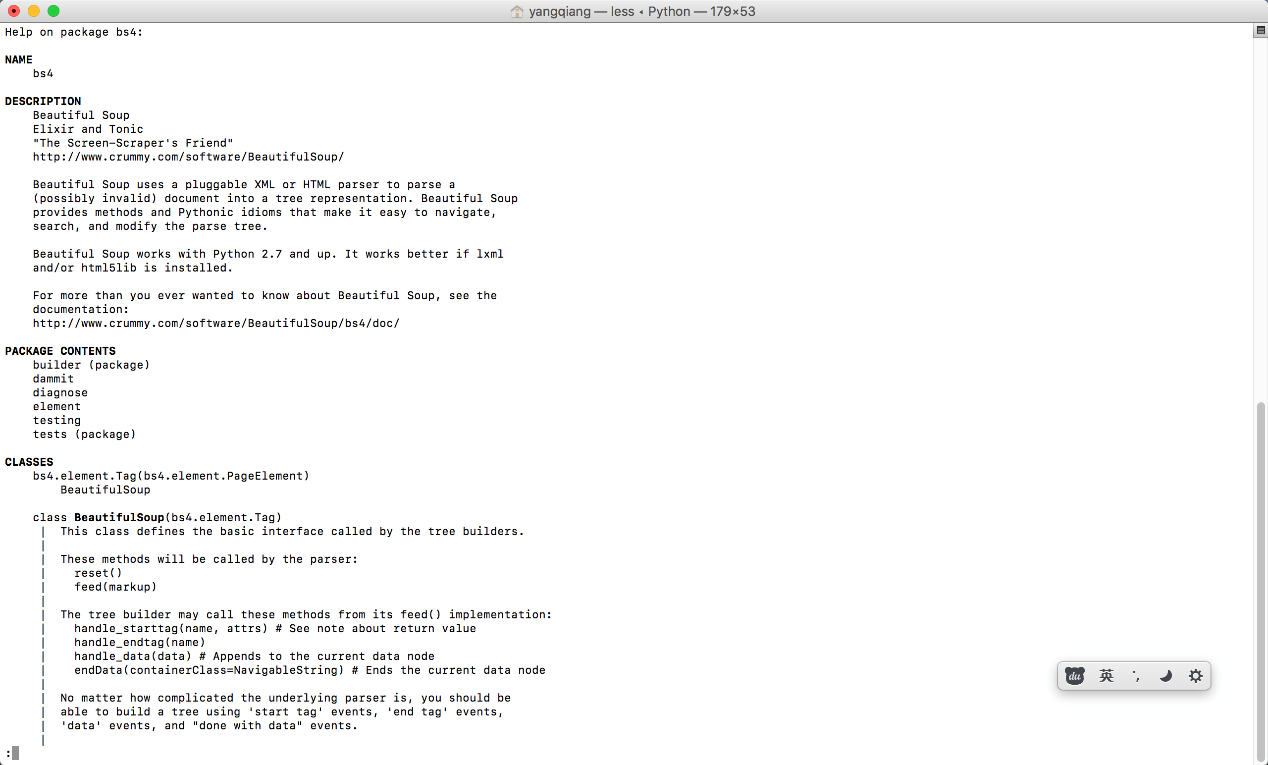
Python 3.6.6 (v3.6.6:4cf1f54eb7, Jun 26 2018, 19:50:54)

[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin

Type "help", "copyright", "credits" or "license" **for** more information.

>>> import bs4

>>> help(bs4)

* 帮助截图

## BeautifulSoup对象构建

### BeautifulSoup构造器说明

\_\_init\_\_(self, markup='', features=**None**, builder=**None**, parse\_only=**None**, from\_encoding=**None**, exclude\_encodings=**None**, \*\*kwargs)

|- markup：标记字符串，或者类似文件的对象，一般我们使用下载的页面字符串。

|- features：解析的方式

|- "lxml"

|- "lxml-xml"

|- "html.parser" *# 一般我们使用这种解析HTML*

|- "html5lib"

|- "html"

|- "html5"

|- "xml")

### 构建对象

**import** **requests**

**import** **bs4**

**def** crawl\_one\_page(page\_no):

url = 'https://ke.qq.com/course/list?mt=1001&st=2002&tt=3019&price\_min=0&price\_max=0&page=**%d**' % page\_no

session = requests.Session()

response = session.get(url)

**if** **not** response:

**return**

content\_html = response.content.decode('utf-8')

bs\_content = bs4.BeautifulSoup(content\_html, "html.parser") *# <---对象创建*

## BeautifulSoup的节点管理属性

1. BeautifulSoup类型说明：
   * BeautifulSoup 就从Tag继承的，上面介绍的方式在Tag都有用。
     + | BeautifulSoup
     + | bs4.element.Tag
     + | bs4.element.PageElement
     + | builtins.object

* BeautifulSoup的节点数据管理，常见属性都是与HTML或者XML的DOM有关系。
  + children # 所有的子节点
  + descendants # 所有后代节点
  + next # 下一个节点
  + nextSibling # 下一个节点
  + next\_elements # 下一个元素
  + next\_siblings # 后面兄弟节点
  + parents # 父节点
  + previous # 前一个节点
  + previousSibling # 前一个兄弟节点
  + previous\_elements # 前一个元素节点
  + previous\_siblings # 前面的兄弟节点
  + isSelfClosing # 单元素节点
  + is\_empty\_element # 同上
  + parserClass # 解析类
  + string # 字符串格式的数据
  + stripped\_strings # 去空的字符串数据
  + text # 节点值
* 属性使用

**for** item\_ **in** bs\_content.children:

print(type(item\_))

* 输出的主要数据类型：

<**class** '**bs4**.element.NavigableString'>

<**class** '**bs4**.element.Doctype'>

<**class** '**bs4**.element.NavigableString'>

<**class** '**bs4**.element.Tag'>

<**class** '**bs4**.element.NavigableString'>

<**class** '**bs4**.element.NavigableString'>

<**class** '**bs4**.element.Tag'>

<**class** '**bs4**.element.Comment'>

## BeautifulSoup四大数据类型

### Tag数据类型

#### Tag的属性

* Tag 表示HTML或者XML的标签，Tag有两个重要的属性：
  + name
  + attrs
* 其他重要属性，可以从构造器得知
  + builder
  + parser
  + namespace
  + prefix
  + parent
  + previous
  + is\_xml
* 从源代码知道，节点的内容属性
  + contents

\_\_init\_\_(self, parser=**None**, builder=**None**, name=**None**, namespace=**None**, prefix=**None**, attrs=**None**, parent=**None**, previous=**None**, is\_xml=**None**)

| Basic constructor.

2 . Tag属性使用例子

print(bs\_content.name)

print(bs\_content.attrs)

print(bs\_content.builder)

print(bs\_content.namespace)

print(bs\_content.prefix)

print(bs\_content.parent)

print(bs\_content.previous)

print(bs\_content.is\_xml)

print(bs\_content.contents)

#### tag的访问

* tag的访问方式很多种
  + 使用节点管理属性遍历得到，比如使用children属性访问。（比较麻烦，不推荐）
    - 使用find函数得到（后面介绍）
    - 使用select函数得到（后面介绍）
  + 成员方式，使用节点name作为属性访问。
* tag的属性访问方式： Tag.name

print(bs\_content.html.body.section.div.div)

* 上面html是bs\_content的节点，body是html的节点，依次类推。

#### tag的属性访问

* tag的属性访问方式有：
  + 使用attrs属性访问；
  + 使用get方法访问；
  + 使用下标[ ] 运算符

print(bs\_content.html['lang'])

print(bs\_content.html.attrs)

print(bs\_content.html.get('lang'))

* 上述的下标运算，还可以修改属性，释放属性。

### NavigableString数据类型

* NavigableString类型主要就是标签内部的文本内容；
* NavigableString访问方式
  + 使用string属性

print('节点:', bs\_content.html.body.section.div.h1)

print('string:', bs\_content.html.body.section.div.h1.string)

print('text:', bs\_content.html.body.section.div.h1.text)

print('节点:', bs\_content.html.body.section.div.div.a)

print('string:', bs\_content.html.body.section.div.div.a.string)

print('text:', bs\_content.html.body.section.div.div.a.text)

* 运行结果：

节点: <h1 class="hidden-clip">Python在线培训视频教程\_编程语言\_IT·互联网</h1>

string: Python在线培训视频教程\_编程语言\_IT·互联网

text: Python在线培训视频教程\_编程语言\_IT·互联网

节点: <a class="mod-breadcrumbs\_\_nav" href="/course/list?mt=1001" report-tdw="module=hotword&amp;action=filter&amp;platform=1&amp;ver1=1001" title="IT·互联网">

IT·互联网 <i></i>

</a>

string: **None**

text:

IT·互联网

* 说明：
  + string得到NavigableString；当节点的内容包含其他子节点，string返回None，但text总是返回其中的文本。
  + text返回的string类型；

### Comment数据类型

* Comment表示注释，访问Comment使用string得到，但string不包含注释符号，这样容易与NavigableString混淆，可以判定类型来区分。使用string得到值。
  + | - name属性
  + | - string属性

**if** type(节点.string)==bs4.element.Comment:

* 一个例子
* **for** item\_ **in** bs\_content.children:
* **if** type(item\_) == bs4.element.Comment:
* print(dir(item\_))
* print(item\_.name, item\_.string)
* **break**

### Doctype数据类型

* 一个HTML最多一个Doctype类型；也是特殊的Doctype，使用不多，一般包含如下几个属性：
  + name,
  + string

**for** item\_ **in** bs\_content.children:

**if** type(item\_) == bs4.element.Doctype:

print(item\_.name, item\_.string)

## 节点查询-find与find\_all

### find与find\_all函数说明

1. find:
   * 返回找到的第一个元素

find(self, name=**None**, attrs={}, recursive=**True**, text=**None**, \*\*kwargs)

|- find的参数都是查询的条件。

|- name ： 节点名

|- attrs={} ： 节点的属性

|- recursive：是否递归，还是只对当前节点子阶段查找

|- text ： 节点的文本值

1. findAll：
   * 返回所有找到的元素：

find\_all(self, name=**None**, attrs={}, recursive=**True**, text=**None**, limit=**None**, \*\*kwargs)

参数与find一样，因为find\_all返回所有结果，为了效率，提供limit参数限制查询的数量，当查询结果满足数量，终止查询。

1. find与find\_all的返回类型
   * find\_all返回的类型：bs4.element.ResultSet
   * find返回的类型是：bs4.element.Tag
2. find与find\_all的函数别名：
   * findChild = find(self, name=None, attrs={}, recursive=True, text=None, \*\*kwargs)
   * findChildren = find\_all(self, name=None, attrs={}, recursive=True, text=None, limit=None, \*\*kwargs)

### find与find\_all 的查询条件

* 因为find与find\_all仅仅返回结果不同，功能行为完全相同，下面使用find\_all来作为例子说明, 并且使用腾讯课堂的url作为例子：
  + url：https://ke.qq.com/course/list?mt=1001&st=2002&tt=3019&price\_min=0&price\_max=0&page=1

1. 全部查找

list\_course = bs\_content.find\_all()

1. 根据name查找

list\_course = bs\_content.find\_all(name='div')

1. 根据attr查找

list\_course = bs\_content.find\_all(attrs={'class':"main autoM clearfix"})

1. 根据text查找

list\_course = bs\_content.find\_all(text='教育')

* 注意：有的文本不是很规范，两边存在空格换行之类的。

list\_course = bs\_content.find\_all(text='''

609人最近报名

''')

1. 根据多个条件同时查找

list\_course = bs\_content.find\_all(

name='a',

attrs={'class': "item-source-link"},

text='马哥教育'

)

### ResultSet的使用

* ResultSet的继承结构
  + | ResultSet
  + | builtins.list
  + | builtins.object
* 本质是list，list中的元素还是Tag。

### name与attrs参数的多重使用形式

* The value of a key-value pair in the 'attrs' map can be a **string**, **a list of strings**, **a regular expression object**, or **a callable** that **takes a string** and **returns whether or not the string matches for some custom definition of 'matches'**. The same is true of the tag name.
* name参数与attrs参数的值可以使用的形式：
  + 字符串（最自然的方式，前面就是这样使用的）
  + 字符串列表
  + 正则表达式对象
  + 可调用函数（参数一个，类型是字符串，返回值是逻辑值，决定是否被选择）

#### 字符串列表

1. name = ['div', 'a', ....]
   * 同时选择多个节点
2. attrs = {'节点属性': ['value1', 'value2', ...... ]}
   * 同时判定多个属性
3. 例子：

results = bs\_content.find\_all(

name=['div', 'a'],

attrs={'class': ['market-bd market-bd-6 course-list course-card-list-multi-wrap js-course-list',

'item-source-link']}

)

#### 正则表达式对象

* 必须使用正则表达式对象，不能使用正则表达式列表。

1. name = re.compile(r'正则表达式')
   * 匹配正则表达式满足的节点。
2. attrs = {'节点属性': re.compile(r'正则表达式')}
   * 匹配节点属性满足正则表达式的节点。
3. 例子：

results = bs\_content.find\_all(

name=re.compile(r'\w**{3}**'),

attrs={'class': re.compile(r'\w{10,}')}

)

print(len(results))

* 说明：
  + 上面例子选择节点名包含3隔字符，属性class至少10隔字符以上的节点。

#### 可调用函数

1. name = 函数名
   * 如果函数返回True，就返回节点；返回False，节点就被过滤掉。
2. attrs = {'节点属性': 函数名}
   * 如果函数返回True，就匹配属性；返回False，具有这个属性值得节点就被过滤掉。
3. 例子：

**def** node\_func(value):

*# print('节点:', type(value))*

**if** value.name == 'div' **or** value.name == 'a':

**return** **True**

**else**:

**return** **False**

**def** attr\_func(value):

*# print('属性:', type(value), value)*

**if** value == 'market-bd':

**return** **True**

**return** **False**

results = bs\_content.find\_all(

name=node\_func,

attrs={'class': attr\_func}

)

print(len(results))

* 说明：
  + 上面例子匹配div与a节点，并且class属性中包含market-bd值。
  + class属性函数对class的属性的每个值都会调用。

### keyword参数

1. 如果使用keyword，则默认是属性
   * class属性与关键字class冲突，使用class\_，其他冲突关键字一次类推。

results = bs\_content.find\_all(class\_='item-source-link')

print(len(results))

1. keyword参数指定属性，也可以使用字符串列表与可调用函数。

soup.find\_all(href=re.compile("\w**{3}**"), id='button')

## 选择器查找-select与select\_one

### select函数说明

1. 函数说明：

select(self, selector, \_candidate\_generator=**None**, limit=**None**)

|- selector : 样式选择器

|- \_candidate\_generator：候选生成器，用来替代默认的节点搜索(选择器只在生成器产生的节点中选择)

|- limit：限制返回的节点数

1. 使用例子代码

**def** div\_candidate\_generator(tag):

print(tag.name)

print(len(list(tag.descendants)))

**for** item\_ **in** tag.descendants:

**if** isinstance(item\_, bs4.element.Tag):

**yield** item\_

result = bs\_content.select(

selector='div > div',

\_candidate\_generator=div\_candidate\_generator,

limit=300)

print(len(result))

### select的选择器使用方式

* select使用的选择器遵循CSS选择器规则
  + 标签名
  + id选择器
  + class选择器
  + \*通配符
  + 属性选择器
  + 组合选择器
  + 伪类选择器

#### 基本选择器

1. 标签选择器
   * 表签名

* 例子代码

result = bs\_content.select('span')

print(len(result))

* 说明：
  + 返回所有span节点元素

1. id选择器
   * #id值

* 例子代码

result = bs\_content.select('#js\_main\_nav')

print(len(result))

* 说明：
  + 选择id值为js\_main\_nav的节点元素。

1. class选择器
   * .class属性值

* 例子代码

*# 实际的属性是多个值： main autoM clearfix：原始节点是：<section class="main autoM clearfix">*

result = bs\_content.select('.main')

print(len(result))

* 说明：
  + 返回class属性值有main的节点元素。

1. \*通配选择
   * 直接使用\*，表示选择所有标签，在组合选择器中也可以使用。

* 例子代码

result = bs\_content.select('\*')

print(len(result))

* 说明：
  + 选择所有节点

#### 属性选择器

* 属性选择器有如下几种形式：
  + [属性]：选择具有属性的节点。
  + [属性 = 值]：选择属性具有属性值的节点
  + [属性 = ~值]：选择属性的属性值包含指定值的节点
  + [属性 = | 值]：选择属性的属性值以指定值开头的节点
* 还有其他扩展的属性值，可以参考更加全的CSS选择器。
* 例子代码

*# 所有具有type属性的节点*

result = bs\_content.select('[type]')

print(len(result))

*# 选择属性type=text/javascript的节点*

result = bs\_content.select('[type=text/javascript]')

print(len(result))

*# 属性class中含有main单词的节点*

result = bs\_content.select('[class~=main]')

print(len(result))

*# 选择type属性以text开头所有节点*

result = bs\_content.select('[type|=text]')

#### 组合选择器

* 组合选择器有多种
  + 节点与节点组合
  + 节点与属性组合
  + 其他组合方式

1. 节点与节点组合
   * element1, element2 ：选择节点element1与element2
   * element1 element2 ：选择element1内部的element2节点
   * element1 > element2 ：选择element1的所有element2子节点
   * element1 + element2 ：选择element1后面的element2节点（紧邻）

* 例子代码

*# 选择节点section与div*

result = bs\_content.select('section,div')

print(len(result))

*# 选择节点section内的所有div节点*

result = bs\_content.select('section div')

print(len(result))

*# 选择节点section的div子节点*

result = bs\_content.select('section > div')

print(len(result))

*# 选择节点section后的div节点*

result = bs\_content.select('header + section')

print(len(result))

1. 节点与属性组合
   * element[属性]
   * element[属性 = 值]
   * element[属性 ~= 值]
   * element[属性 |= 值]

* 例子代码：

*# 所有具有type属性的script节点*

result = bs\_content.select('script[type]')

print(len(result))

*# 选择属性type=text/javascript的script节点*

result = bs\_content.select('script[type=text/javascript]')

print(len(result))

*# 属性class中含有main单词的section节点*

result = bs\_content.select('section[class~=main]')

print(len(result))

*# 选择class属性以topic开头所有section节点*

result = bs\_content.select('section[class|="topic"]')

print(len(result))

1. 混合组合方式
   * 上面的所有选择器都可以作为节点使用

result = bs\_content.select('section > div > div[class~="course-list"]')

print(len(result))

**for** item\_ **in** result:

print(item\_.attrs)

result = bs\_content.select('section > div > [class~="course-list"]')

print(len(result))

**for** item\_ **in** result:

print(item\_.attrs)

result = bs\_content.select('section > div > .market-bd-6')

print(len(result))

**for** item\_ **in** result:

print(item\_.attrs)

* 运行结果

1

{'class': ['market-bd', 'market-bd-6', 'course-list', 'course-card-list-multi-wrap', 'js-course-list'], 'data-report-module': 'middle-course'}

1

{'class': ['market-bd', 'market-bd-6', 'course-list', 'course-card-list-multi-wrap', 'js-course-list'], 'data-report-module': 'middle-course'}

1

{'class': ['market-bd', 'market-bd-6', 'course-list', 'course-card-list-multi-wrap', 'js-course-list'], 'data-report-module': 'middle-course'}

#### 伪类选择器

* 目前bs4没有实现伪类选择器，大家在使用诶时候，一定需要明确。

result = bs\_content.select('body > section > div > div:nth-child(2)')

print(len(result))

* 说明：
  + 上面代码抛出异常：NotImplementedError: Only the following pseudo-classes are implemented: nth-of-type.

# Flask-SQLAlchemy

* 安装MySQL
* 安装MariaDB

## 安装

* pip install flask-sqlalchemy
* pip install pymysql

## 配置[¶](file:///C:\Users\11863\Desktop\05%E6%95%B0%E6%8D%AE%E5%BA%93%E6%93%8D%E4%BD%9C.html#配置)

mysql+pymysql://账号:密码@ip地址/数据库名字

In [ ]:

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'mysql+pymysql://root:123qweASD@localhost/garbage'

db = SQLAlchemy(app)

## 数据模型

构建对象 继承db.Model

In [ ]:

**class** **Garbage**(db.Model):

\_\_tablename\_\_ = 'garbage'

id = db.Column(db.Integer, primary\_key=**True**)

name = db.Column(db.String(255), unique=**True**)

type = db.Column(db.Integer, unique=**True**)

**def** \_\_init\_\_(self, name, type):

self.name = name

self.type = type

**def** \_\_repr\_\_(self):

**return** '<Garbage **%r**>' % self.name

## 模型接口

数据可查询方法

In [ ]:

**class** **GarbageInfo**():

**def** \_\_init\_\_(self):

self.\_\_fields\_\_ = ['id','name', 'type']

**def** findAll(self):

**return** Garbage.query.all()

**def** find\_by\_type(self,find\_type):

**return** Garbage.query.filter\_by(type = find\_type).all()

**def** find\_by\_name(self,find\_name):

**return** Garbage.query.filter(Garbage.name.contains(find\_name))

**def** find\_exam\_by\_rand(self):

r = random.randint(1,30)

**return** Garbage.query.slice(r,r+10).all()

## 对接路由使用

In [ ]:

@app.route('/garbage',methods=['GET'])

**def** get\_garbage():

type = request.args.get('type')

garbageInfo = GarbageInfo()

ret\_data=garbageInfo.find\_by\_type(type)

ret\_data = Class\_To\_Data(ret\_data,garbageInfo.\_\_fields\_\_)

**return** Response(json.dumps({'status':200,'msg':'ok','data':ret\_data},ensure\_ascii=**False**,))

## 格式化JSON返回

In [ ]:

**def** Class\_To\_Data(data\_list,fields,type=0):

**if** **not** type:

user\_list = []

**for** u **in** data\_list:

temp = {}

**for** f **in** fields:

**if** f **in** ['create\_time','login\_time']:

temp[f] = datetime.datetime.strftime(getattr(u,f), "%Y-%m-**%d** %H:%M:%S ")

**else**:

temp[f] = getattr(u,f)

user\_list.append(temp)

**else**:

user\_list = {}

**for** f **in** fields:

**if** f **in** ['create\_time', 'login\_time']:

d = getattr(data\_list, f)

**if** d:

user\_list[f] = datetime.datetime.strftime(d, "%Y-%m-**%d** %H:%M:%S ")

**else**:

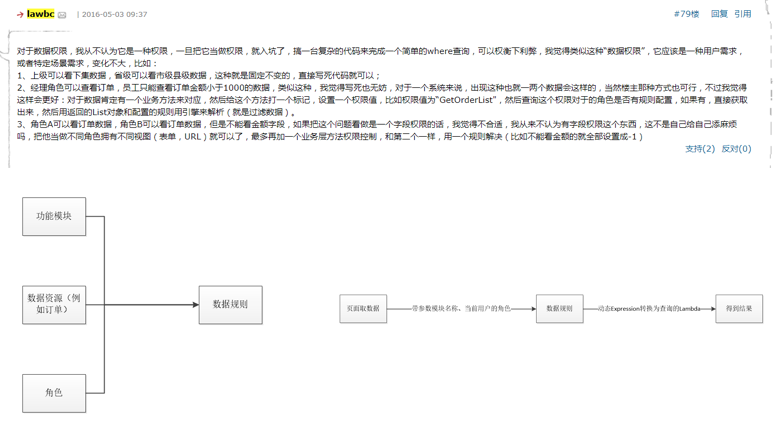
user\_list[f] = getattr(data\_list, f)

**return** user\_list

# 数据接口设计

* 跨平台性
* 良好的响应速度
* 接口要为移动客户端考虑
* 考虑移动端的网络情况和耗电量
* 通用的数据交换格式
* 接口统计功能
* 客户端与服务端的肥瘦平衡
* 隐式用户与显式用户
* 安全问题
* 良好的接口说明文档和测试程序
* 版本的维护

# 数据权限问题



# 部署

apache nginx

# anaconda环境

* jupyterLab
* notebook