# 网络爬虫

又被称为网页蜘蛛，网络机器人，在FOAF社区中间，更经常的称为网页追逐者，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。

## 通用爬虫

通用网络爬虫是捜索引擎抓取系统（Baidu、Google、Yahoo等）的重要组成部分。主要目的是将互联网上的网页下载到本地，形成一个互联网内容的镜像备份。

通用**搜索引擎**（Search Engine）工作原理

通用网络爬虫，从互联网中搜集网页，采集信息，这些网页信息用于为搜索引擎建立索引从而提供支持，它决定着整个引擎系统的内容是否丰富，信息是否即时，因此其性能的优劣直接影响着搜索引擎的效果。

通用搜索引擎大多提供基于关键字的检索，难以支持根据语义信息提出的查询，无法准确理解用户的具体需求。目标是尽可能大的网络覆盖率



## 聚焦爬虫

聚焦爬虫，是"面向特定主题需求"的一种网络爬虫程序，它与通用搜索引擎爬虫的区别在于： 聚焦爬虫在实施网页抓取时会对内容进行处理筛选，尽量保证只抓取与需求相关的网页信息。

聚焦爬虫是一个自动下载网页的程序，根据既定的抓取目标，有选择的访问万维网网页和相关的链接，并不追求大的覆盖率。

## 增量爬虫

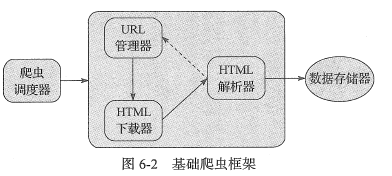
对已下载网页采取增量式更新和只爬行新产生的或者已经发生变化网页的爬虫，在一定程度上保证所爬行的页面是尽可能新的页面。和周期性爬行和刷新页面的爬虫相比，增量爬虫只会在需要的时候爬行产生或发生更新的页面，并不重新下载没有发生变化的页面。减少资源耗费，但是增加了爬行算法的复杂度。

## 深层爬虫

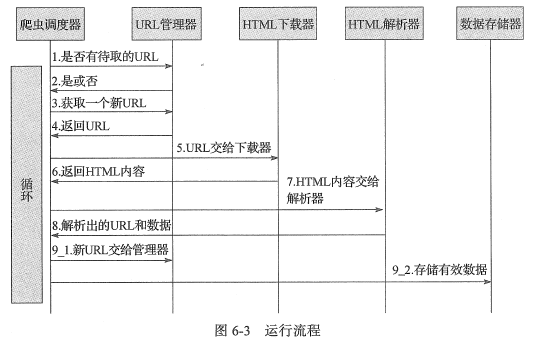
web页面存在方式分为表层和深层网页，表层是指传统搜索引擎可以搜索的页面，以超链接可以到达的静态网页为主构成的web页面。

深层网络是指大部分内容不能通过静态链接获取的，隐藏在搜索表单之后的，只有用户提交一些关键词才能获得的web页面。

## 基础爬虫架构及运行流程



* 爬虫调度器主要负责统筹其他四个模块的协调工作。
* URL管理器负责管理URL链接，维护已经爬取的URL集合和未爬取的URL集合，提供获取新URL链接的接口。
* HTML下载器用于从URL管理器中获取未爬取的URL链接并下载HTML网页。
* HTMLT解析器用于从HTML下载器中获取已经下载的HTML网页，并从中解析出新的URL链接交给URL管理器，解析出有效数据交给数据存储器。
* 数据存储器用于将HTML解析器解析出来的数据通过文件或者数据库的形式存储起来。



### URL管理器

URL管理器主要包括两个变量，一个是已爬取URL的集合，另一个是未爬取URL的集

合。采用Python中的set类型，主要是使用set的去重复功能，防止链接重复爬取，因为爬

取链接重复时容易造成死循环。

链接去重复在Python爬虫开发中是必备的功能，解决方案主要有三种:

1. 内存去重
2. 关系数据库去重
3. 缓存数据库去重。

大型成熟的爬虫基本上采用缓存数据库的去重方案，尽可能避免内存大小的限制，又比关系型数据库去重性能高很多。

爬取数量较小可以使用Python中set这个内存去重方式。

URL管理器除了具有两个URL集合，还需要提供以下接口，用于配合其他模块使用，

接口如下:

* 判断是否有待取的URL，方法定义为has\_new\_url()
* 添加新的URL到未爬取集合中，方法定义为add new\_url(url) , add new\_urls(urls)
* 获取一个未爬取的URL，方法定义为get new url()
* 获取未爬取URL集合的大小，方法定义为new url size()
* 获取已经爬取的URL集合的大小，方法定义为oldee url\_ size()

### HTML下载器

爬虫请求库中各模块，例如requests.get()方法

### HTML解析器

爬虫解析库中各模块，例如beautifulsoup

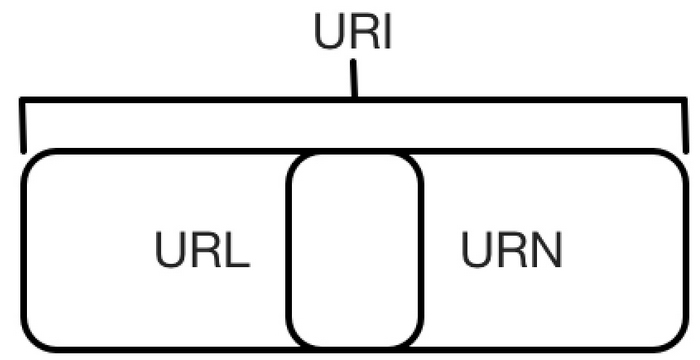
### 数据存储器

爬虫库中各形式存储。一次性写入文件容易导致系统异常，造成数据丢失

### 爬虫调度器

爬虫调度器首先要做的是初始化各个模块，然后通过cral(root\_url)方法传入入口UR L，方法内部实现按照运行流程控制各个模块的工作。

# HTTP基本原理



## URL

（Uniform / Universal Resource Locator的缩写）：统一资源定位符，是用于完整地描述Internet上网页和其他资源的地址的一种标识方法。

基本格式：scheme://host[:port#]/path/…/[?query-string][#anchor]

scheme：协议(例如：http, https, ftp)

host：服务器的IP地址或者域名

port#：服务器的端口（如果是走协议默认端口，缺省端口80）

path：访问资源的路径

query-string：参数，发送给http服务器的数据

anchor：锚（跳转到网页的指定锚点位置）

## URI

全称为 Uniform Resource Identifier，即统一资源标志符。URL 是 URI 的子集，也就是说每个 URL 都是 URI，但不是每个 URI 都是 URL

URI 还包括一个子类叫做 URN，它的全称为 Universal Resource Name，即统一资源名称。URN 只命名资源而不指定如何定位资源，如 urn:isbn:0451450523

## HTTP、HTTPS

HTTP协议（HyperText Transfer Protocol，超文本传输协议）：是一种发布和接收 HTML页面的方法。用于从网络传输超文本数据到本地浏览器的传送协议，保证传送高效而准确地传送超文本文档

HTTPS（Hypertext Transfer Protocol over Secure Socket Layer）是以安全为目标的 HTTP 通道，简单讲是HTTP的安全版，在HTTP下加入SSL层。

HTTPS 的安全基础是 SSL（Secure Sockets Layer 安全套接层），因此通过它传输的内容都是经过 SSL 加密的

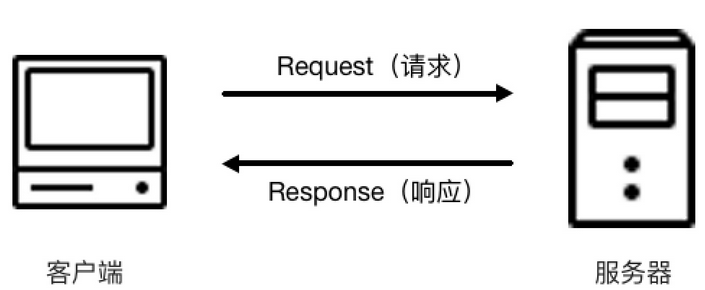
SSL主要用于Web的安全传输协议，在传输层对网络连接进行加密，保障在Internet上数据传输的安全，它的主要作用可以分为两种：

* 建立一个信息安全通道，来保证数据传输的安全。
* 确认网站的真实性，凡是使用了 https 的网站，都可以通过点击浏览器地址栏的锁头标志来查看网站认证之后的真实信息，也可以通过 CA 机构颁发的安全签章来查询。

我们如果要爬取这样的站点就需要设置忽略证书的选项，否则会提示 SSL 链接错误

HTTP的端口号为80，HTTPS的端口号为443

## HTTP请求过程



### Request

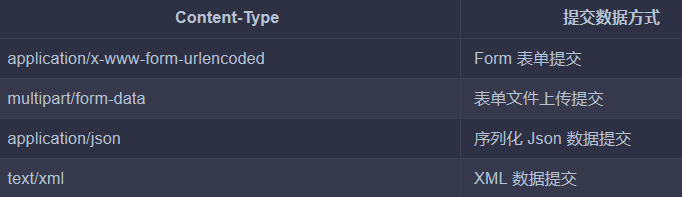
即请求，由客户端向服务端发出。可以将 Request 划分为四部分内容：Request Method、Request URL、Request Headers、Request Body，即请求方式、请求链接、请求头、请求体。

Request Method

请求方式常见的有两种类型，GET 和 POST。

在浏览器中直接输入一个 URL 并回车，这便发起了一个 GET 请求，请求的参数会直接包含到 URL 里，例如百度搜索 Python，这就是一个 GET 请求，链接为：https://www.baidu.com/s?wd=Python，URL 中包含了请求的参数信息，这里参数 wd 就是要搜寻的关键字。POST 请求大多为表单提交发起，如一个登录表单，输入用户名密码，点击登录按钮，这通常会发起一个 POST 请求，其数据通常以 Form Data 即表单的形式传输，不会体现在 URL 中。

* GET 方式请求中参数是包含在 URL 里面的，数据可以在 URL 中看到，而 POST 请求的 URL 不会包含这些数据，数据都是通过表单的形式传输，会包含在 Request Body 中。
* GET 方式请求提交的数据最多只有 1024 字节，而 POST 方式没有限制。



### Response

响应，由服务端返回给客户端。Response 可以划分为三部分，Response Status Code、Response Headers、Response Body。

Response Status Code

常见的错误代码及错误原因

200 成功 服务器已成功处理了请求。

301 永久移动 请求的网页已永久移动到新位置，即永久重定向。

302 临时移动 请求的网页暂时跳转到其他页面，即暂时重定向。

400 错误请求 服务器无法解析该请求。

401 未授权 请求没有进行身份验证或验证未通过。

403 禁止访问 服务器拒绝此请求。

404 未找到 服务器找不到请求的网页。

500 服务器内部错误 服务器遇到错误，无法完成请求。

### IP代理

**代理分类**

**(1) 根据协议区分**

根据代理的协议，代理可以分为如下类别。

* FTP代理服务器：主要用于访问FTP服务器，一般有上传、下载以及缓存功能，端口一般为21、2121等。
* HTTP代理服务器：主要用于访问网页，一般有内容过滤和缓存功能，端口一般为80、8080、3128等。
* SSL/TLS代理：主要用于访问加密网站，一般有SSL或TLS加密功能（最高支持128位加密强度），端口一般为443。
* RTSP代理：主要用于访问Real流媒体服务器，一般有缓存功能，端口一般为554。
* Telnet代理：主要用于telnet远程控制（黑客入侵计算机时常用于隐藏身份），端口一般为23。
* POP3/SMTP代理：主要用于POP3/SMTP方式收发邮件，一般有缓存功能，端口一般为110/25。
* SOCKS代理：只是单纯传递数据包，不关心具体协议和用法，所以速度快很多，一般有缓存功能，端口一般为1080。SOCKS代理协议又分为SOCKS4和SOCKS5，前者只支持TCP，而后者支持TCP和UDP，还支持各种身份验证机制、服务器端域名解析等。简单来说，SOCK4能做到的SOCKS5都可以做到，但SOCKS5能做到的SOCK4不一定能做到。

**(2) 根据匿名程度区分**

根据代理的匿名程度，代理可以分为如下类别。

* 高度匿名代理：会将数据包原封不动地转发，在服务端看来就好像真的是一个普通客户端在访问，而记录的IP是代理服务器的IP。
* 普通匿名代理：会在数据包上做一些改动，服务端上有可能发现这是个代理服务器，也有一定几率追查到客户端的真实IP。代理服务器通常会加入的HTTP头有HTTP\_VIA和HTTP\_X\_FORWARDED\_FOR。
* 透明代理：不但改动了数据包，还会告诉服务器客户端的真实IP。这种代理除了能用缓存技术提高浏览速度，能用内容过滤提高安全性之外，并无其他显著作用，最常见的例子是内网中的硬件防火墙。
* 间谍代理：指组织或个人创建的用于记录用户传输的数据，然后进行研究、监控等目的的代理服务器。

**常见代理设置**

* 使用网上的免费代理：最好使用高匿代理，另外可用的代理不多，需要在使用前筛选一下可用代理，也可以进一步维护一个代理池。
* 使用付费代理服务：互联网上存在许多代理商，可以付费使用，质量比免费代理好很多。
* ADSL拨号：拨一次号换一次IP，稳定性高，也是一种比较有效的解决方案。

## 构造合理http请求头

if \_\_name\_\_ == '\_\_main\_\_':  
 url='https://www.baidu.com/'  
 header= {'Upgrade-Insecure-Requests':'1',  
 'User-Agent':'Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36',  
 'Accept':'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8',  
 'Accept-Encoding':'gzip, deflate, sdch, br',  
 'Accept-Language':'zh-CN,zh;q=0.8',  
 }  
 s=requests.Session()  
 req=s.get(url=url,headers=header)  
 print(s.cookies)

### 静态网页和动态网页

网页的内容是 HTML 代码编写的，文字、图片等内容均是通过写好的 HTML 代码来指定的，这种页面叫做静态网页。

这种网页加载速度快，编写简单，但是存在很大的缺陷，如可维护性差，不能根据 URL 灵活多变地显示内容等，例如我们想要给这个网页的 URL 传入一个 name 参数，让其在网页中显示出来，是无法做到的。

动态网页可以动态解析 URL 中参数的变化，关联数据库并动态地呈现不同的页面内容，非常灵活多变，我们现在遇到的大多数网站都是动态网站，它们不再是一个简单的 HTML，而是可能由 JSP、PHP、Python 等语言编写的，功能相比静态网页强大和丰富太多

### 无状态HTTP

HTTP 的无状态是指 HTTP 协议对事务处理是没有记忆能力的，也就是说服务器不知道客户端是什么状态。当我们向服务器发送一个 Requset 后，服务器解析此 Request，然后返回对应的 Response，服务器负责完成这个过程，而且这个过程是完全独立的，服务器不会记录前后状态的变化，也就是缺少状态记录，这意味着如果后续需要处理需要前面的信息，则它必须要重传

保持 HTTP 连接状态的技术, Session 和 Cookies

Session 在服务端，也就是网站的服务器，用来保存用户的会话信息，Cookies 在客户端，也可以理解为浏览器端，有了 Cookies，浏览器在下次访问网页时会自动附带上它发送给服务器，服务器通过识别 Cookies 并鉴定出是哪个用户，然后再判断用户是否是登录状态，然后返回对应的 Response。

因此在爬虫中，有时候处理需要登录才能访问的页面时，我们一般会直接将登录成功后获取的 Cookies 放在 Request Headers 里面直接请求，而不必重新模拟登录。

### Session

在 Web 中 Session 对象用来存储特定用户会话所需的属性及配置信息。这样，当用户在应用程序的 Web 页之间跳转时，存储在 Session 对象中的变量将不会丢失，而是在整个用户会话中一直存在下去。当用户请求来自应用程序的 Web 页时，如果该用户还没有会话，则 Web 服务器将自动创建一个 Session 对象。当会话过期或被放弃后，服务器将终止该会话。

除非程序通知服务器删除一个 Session，否则服务器会一直保留，比如程序一般都是在我们做注销操作的时候才去删除 Session。由于关闭浏览器不会导致 Session 被删除，这就需要服务器为 Seesion 设置一个失效时间，当距离客户端上一次使用 Session 的时间超过这个失效时间时，服务器就可以认为客户端已经停止了活动，才会把 Session 删除以节省存储空间。

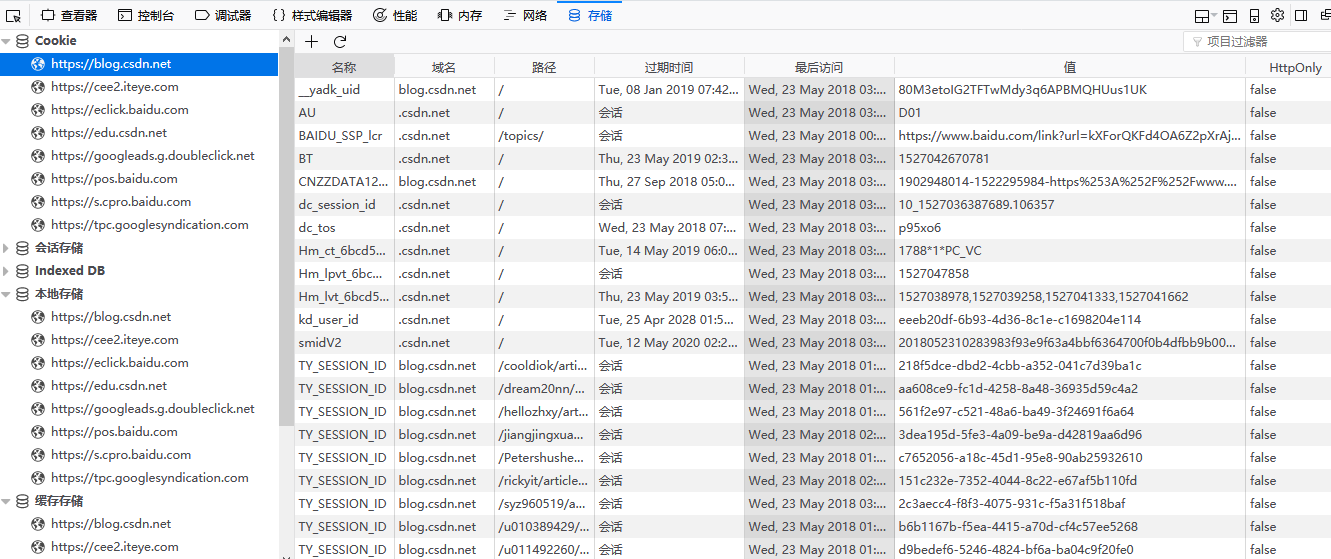
requests.Session 会话对象让你能够跨请求保持某些参数，它也会在同一个 Session 实例发出的所有请求之间保持 cookie，requests 模块不能执行 JavaScript

### Cookies

Cookie，有时也用其复数形式 Cookies，指某些网站为了辨别用户身份、进行 Session 跟踪而储存在用户本地终端上的数据。

当客户端第一次请求服务器时，服务器会返回一个 Headers 中带有 Set-Cookie 字段的 Response 给客户端，用来标记是哪一个用户，客户端浏览器会把Cookies 保存起来。当浏览器下一次再请求该网站时，浏览器会把此Cookies 放到 Request Headers 一起提交给服务器，Cookies 携带了 Session ID 信息，服务器检查该 Cookies 即可找到对应的 Session 是什么，然后再判断 Session 来以此来辨认用户状态。

Cookies 和 Session 需要配合，一个处于客户端，一个处于服务端，二者共同协作，就实现了登录会话控制。



* Name，即该 Cookie 的名称。Cookie 一旦创建，名称便不可更改
* Value，即该 Cookie 的值。如果值为 Unicode 字符，需要为字符编码。如果值为二进制数据，则需要使用 BASE64 编码。
* Max Age，即该 Cookie 失效的时间，单位秒，也常和 Expires 一起使用，通过它可以计算出其有效时间。Max Age 如果为正数，则该Cookie 在 Max Age 秒之后失效。如果为负数，则关闭浏览器时Cookie 即失效，浏览器也不会以任何形式保存该 Cookie。
* Path，即该 Cookie 的使用路径。如果设置为 /path/，则只有路径为 /path/ 的页面可以访问该 Cookie。如果设置为 /，则本域名下的所有页面都可以访问该 Cookie。
* Domain，即可以访问该 Cookie 的域名。例如如果设置为 .zhihu.com，则所有以 zhihu.com，结尾的域名都可以访问该Cookie。
* Size字段，即此 Cookie 的大小。
* Http字段，即 Cookie 的 httponly 属性。若此属性为 true，则只有在 HTTP Headers 中会带有此 Cookie 的信息，而不能通过 document.cookie 来访问此 Cookie。
* Secure，即该 Cookie 是否仅被使用安全协议传输。安全协议。安全协议有 HTTPS，SSL 等，在网络上传输数据之前先将数据加密。默认为 false。

会话Cookie、持久Cookie

会话 Cookie 就是把 Cookie 放在浏览器内存里，浏览器在关闭之后该 Cookie 即失效，持久 Cookie 则会保存到客户端的硬盘中，下次还可以继续使用，用于长久保持用户登录状态。

# 多进程和多线程

进程和线程

“多任务”就是操作系统可以同时运行多个任务。

单核CPU操作系统轮流让各个任务交替执行，任务1执行0.01秒，切换到任务2执行0.01秒反复执行。表面上看，每个任务都是交替执行的，但是，由于CPU的执行速度实在是太快了，感觉就像所有任务都在同时执行一样。真正的并行执行多任务只能在多核CPU上实现，但是，由于任务数量远远多于CPU的核心数量，所以，操作系统也会自动把很多任务轮流调度到每个核心上执行。

一个任务就是一个进程（Process），比如打开一个浏览器就是启动一个浏览器进程。

有些进程不止同时干一件事，比如Word，它可以同时进行打字、拼写检查、打印等事情。在一个进程内部，要同时干多件事，就需要同时运行多个“子任务”，我们把进程内的这些“子任务”称为线程（Thread）。一个进程至少有一个线程。

## 多进程

### fork—支持linux/unix

os模块封装了常见的系统调用，其中就包括fork

在Unix/Linux操作系统中，提供了一个fork()系统函数，它非常特殊。

编写完毕的代码，在没有运行的时候，称之为程序，正在运行着的代码，就成为进程

* 程序执行os.fork()时，操作系统会创建一个新的进程（子进程），然后复制父进程的所有信息到子进程中

普通的函数调用，调用一次，返回一次，但是fork()调用一次，返回两次，因为操作系统自动把当前进程（称为父进程）复制了一份（称为子进程），然后，分别在父进程和子进程内返回。

* 然后父进程和子进程都会从fork()函数中得到一个返回值，在子进程中这个值一定是0，而父进程中是子进程的 id号

这样做的理由是，一个父进程可以fork出很多子进程，所以，父进程要记下每个子进程的ID，而子进程只需要调用getppid()就可以拿到父进程的ID。

### multiprosessing—跨平台

multiprocessing模块是跨平台版本的多进程模块。

multiprocessing模块提供了一个Process类来代表一个进程对象。

#### 创建子进程

创建子进程时，只需要传入一个执行函数和函数的参数，创建一个Process实例，用start()方法启动。

join()方法可以等待子进程结束后再继续往下运行，通常用于进程间的同步。

def proc(name): #子进程要执行的代码

print('运行子进程 %s (%s)...' %(name, os.getpid())) #得到子进程id

p=Process(target=proc,args=('test',)) #传入函数名和参数  
print('子进程将要执行...')  
p.start()  
p.join()

print('子进程结束')

**Process([group [, target [, name [, args [, kwargs]]]]])**

* target：表示这个进程实例所调用对象；
* args：表示调用对象的位置参数元组；
* kwargs：表示调用对象的关键字参数字典；
* name：为当前进程实例的别名；
* group：大多数情况下用不到；

Process类常用方法：

* is\_alive()：判断进程实例是否还在执行；
* join([timeout])：是否等待进程实例执行结束，或等待多少秒；
* start()：启动进程实例（创建子进程）；
* run()：如果没有给定target参数，对这个对象调用start()方法时，就将执行对象中的run()方法；
* terminate()：不管任务是否完成，立即终止；

Process类常用属性：

* name：当前进程实例别名，默认为Process-N，N为从1开始递增的整数；
* pid：当前进程实例的PID值；

#### pool类

如果要启动大量的子进程，可以用进程池的方式批量创建子进程。

初始化Pool时，可以指定一个最大进程数，当有新的请求提交到Pool中时，如果池还没有满，那么就会创建一个新的进程用来执行该请求；但如果池中的进程数已经达到指定的最大值，那么该请求就会等待，直到池中有进程结束，才会创建新的进程来执行

def task(name):  
 print('运行子进程%s(%s)'%(name,os.getpid()))  
 start=time()  
 sleep(random.random()\*3)  
 end=time()  
 print('子进程%s运行 %.2f 秒'%(name,(end-start))) #子进程01213是同时进行的，某个进程运行完毕后再运行子进程4  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print('父进程%s'%os.getpid())  
 p=Pool(4) #最多同时跑4个子进程，默认大小为cpu核数  
 for i in range(5):  
 p.apply\_async(task,args=(i,))  
 p.close() #调用close之后不能添加新的进程了  
 p.join()  
 print('所有进程完毕')

调用join（）会等待所有子进程执行完毕，调用之前必须先调用close，调用close之后不能添加新的进程了

子进程

很多时候，子进程并不是自身，而是一个外部进程。我们创建了子进程后，还需要控制子进程的输入和输出。

subprocess模块启动一个子进程，控制其输入和输出。

子进程可以通过communicate()方法输入

### 进程间通信

python提供了很多进程间通信的方式，queue用于在多进程间通信，pipe用于两个进程间通信。

#### Queue

可以使用multiprocessing模块的Queue实现多进程之间的数据传递，Queue本身是一个消息列队程序，是多进程安全的队列。

* Put方法用以插入数据到队列中，它有两个可选参数:blocked和timeout。如果blocked为True(默认值)，并且timeout为正值，该方法会阻塞timeout指定的时间，直到该队列有剩余的空间。如果超时，会抛出Queue.Full异常。如果blocked为False，但该Queue已满，会立即抛出Queue.Full异常。
* Get方法可以从队列读取并且删除一个元素。同样，Get方法有两个可选参数:blocked和timeout。如果blocked为True(默认值)，并且timeout为正值，那么在等待时间内没有取到任何元素，会抛出Queue.Empty异常。如果blocked为False,分两种情况:如果Queue有一个值可用，则立即返回该值;否则，如果队列为空，则立即抛出Queue.Empty异常。

#### Pipe

Pipe常用来在两个进程间进行通信，两个进程分别位于管道的两端。

Pipe方法返回(conn1, conn2)代表一个管道的两个端。

Pipe方法有duplex参数，如果duplex参数为True(默认值)，那么这个管道是全双工模式，也就是说conn1和conn2均可收发。若duplex为False, conn1只负责接收消息，conn2只负责发送消息。

send和recv方法分别是发送和接收消息的方法。例如，在全双工模式下，可以调用connl.send发送消息，connl.recv接收消息。如果没有消息可接收，recv方法会一直阻塞。如果管道已经被关闭，那么recv方法会抛出EOFError

## 多线程（threading）

多任务可以由多进程完成，也可以由一个进程内多线程完成。

多线程运行有如下优点:

* 可以把运行时间长的任务放到后台去处理。
* 用户界面可以更加吸引人，比如用户点击了一个按钮去触发某些事件的处理，可以弹出一个进度条来显示处理的进度。
* 程序的运行速度可能加快。
* 在一些需要等待的任务实现上，如用户输入、文件读写和网络收发数据等，线程就比较有用了。在这种情况下我们可以释放一些珍贵的资源，如内存占用等。

### threading

threading模块一般通过两种方式创建多线程:

1. 把一个函数传入并创建Thread实例，然后调用start方法开始执行
2. 直接从threading. Thread继承并创建线程类，然后重写\_\_init\_\_方法和run方法

import threading,time  
def loop1():  
 print('线程 %s 运行...'%threading.current\_thread().name) #显示子线程实例名字  
 n=0  
 while n<3:  
 n=n+1  
 print('%s >>> %s'%(threading.current\_thread().name,n)) #current\_thread()函数返回当前线程实例  
 time.sleep(0.5)  
 print('线程 %s 结束...'%threading.current\_thread().name)  
  
def loop2():  
 print('线程 %s 运行...'%threading.current\_thread().name)  
 n=0  
 while n<2:  
 n+=1  
 print('%s >>> %s' %(threading.current\_thread().name,n))  
 time.sleep(0.5)  
 print('线程 %s 结束...' % threading.current\_thread().name)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print('---开始---:%s'%time.ctime(),'\n')  
 # 显示当前线程，即主线程实例名字  
 print('%s 运行'%threading.current\_thread().name)  
 # 创建子线程，子线程名字在创建时指定，loopthread为子线程的名字，不指定为默认名  
 t=threading.Thread(target=loop1,name='子线程1')  
 p=threading.Thread(target=loop2,name='子线程2')  
 #子线程可以全部start之后再join，即为同步运行，分开执行，为运行完子线程1，再运行子线程2  
 t.start()  
 t.join()  
 p.start()  
 p.join()  
 while True:  
 length=len(threading.enumerate())  
 print('当前运行线程数量：%d'%length)  
 if length<=1:  
 break  
 print('thread %s ended'%threading.current\_thread().name)

任何进程默认就会启动一个线程，该线程称为主线程，主线程又可以启动新的线程， threading模块current\_thread()函数返回当前线程的实例。

#### 线程同步—lock

多线程和多进程最大的不同在于，多进程中，同一个变量，各自有一份拷贝存在于每个进程中，互不影响，而多线程中，所有变量都由所有线程共享，所以，任何一个变量都可以被任何一个线程修改，因此，线程之间共享数据最大的危险在于多个线程同时改一个变量

t.start()  
p.start()  
t.join()  
p.join()

多线程交替运行，每个线程有自己的局部变量，导致赋予全局变量的值会打乱，确保全局变量的值为正确值，要给change\_it()上一把锁，其他线程不能同时执行change\_it()，只能等待，直到锁被释放后，获得该锁以后才能改。锁只有一个，无论多少线程，同一时刻最多只有一个线程持有该锁，所以，不会造成修改的冲突。

def change\_it(n):  
 # 先存后取，结果应该为0:  
 global balance  
 balance = balance + n  
 balance = balance - n  
  
def thread(n):  
 for i in range(1000):  
 #获取锁  
 lock.acquire()  
 try:  
 change\_it(n)  
 finally:  
 #改完释放锁  
 lock.release()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 balance=0  
 lock=threading.Lock()  
 t1 = threading.Thread(target=thread, args=(5,))  
 t2 = threading.Thread(target=thread, args=(8,))  
 t1.start()  
 t2.start()  
 t1.join()  
 t2.join()  
 print(balance)

ThreadLocal: 解决参数在一个线程中各个函数之间互相传递的问题。

local\_data = threading.local() #创建全局ThreadLocal对象

local\_data为一个ThreadLocal对象，每个thread对其可读写属性，但互不影响。可以把local\_school看成全局变量，但每个属性如local\_data.data都是线程的局部变量，可以理解为全局变量local\_data是一个dict，不但可以用local\_data.data，还可以绑定其他变量，如local\_data.teacher等等可以任意读写而互不干扰，也不用管理锁的问题，ThreadLocal内部会处理。

ThreadLocal最常用的地方就是为每个线程绑定一个数据库连接，HTTP请求，用户身份信息等，这样一个线程的所有调用到的处理函数都可以非常方便地访问这些资源。

## 协程 (coroutine )

协程又称微线程，纤程，是一种用户级的轻量级线程。

协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈。因此协程能保留上一次调用时的状态，每次过程重入时，就相当于进人上一次调用的状态。

在并发编程中，协程与线程类似，每个协程表示一个执行单元，有自己的本地数据，与其他协程共享全局数据和其他资源。

协程需要用户自己来编写调度逻辑，对于CPU来说，协程其实是单线程，所以CPU不

用去考虑怎么调度、切换上下文，这就省去了CPU的切换开销，所以协程在一定程度上又好于多线程。

Python通过yield提供了对协程的基本支持，但是不完全，第三方gevent库提供了比较完善的协程支持。gevent是一个基于协程的Python网络函数库，使用greenlet在libev事件循环顶部提供了一个有高级别并发性的API。

主要特性：

* 基于libev的快速事件循环，Linux上是epoll机制。
* 基于greenlet的轻量级执行单元。
* API复用了Python标准库里的内容。
* 支持SSL的协作式sockets.
* 可通过线程池或c-ares实现DNS查询。
* 通过monkey patching功能使得第三方模块变成协作式。

## 进程 vs 线程

多进程模式稳定性高，但是创建进程代价大，消耗内存和占用cpu多

多线程模式稳定性低，任何一个子线程挂掉会导致整个进程崩溃，因为所有线程共享进程的内存。

多任务一旦多到一个限度，就会消耗掉系统所有的资源，结果效率急剧下降，所有任务都做不好。因为线程切换需要保存当前执行的现场环境，同时将新任务的执行环境准备好。

## 异步IO

如果充分利用操作系统提供的异步IO支持，就可以用单进程单线程模型来执行多任务，这种全新的模型称为事件驱动模型

使用异步IO可以执行基于IO的多任务，基于CPU计算的多任务执行不了

## 分布式进程

在Thread和Process中，应当优选Process，因为Process更稳定，而且，Process可以分布到多台机器上，而Thread最多只能分布到同一台机器的多个CPU上。

Python的multiprocessing模块不但支持多进程，其中managers子模块还支持把多进程分布到多台机器上。一个服务进程可以作为调度者，将任务分布到其他多个进程中，依靠网络通信。

# 反爬虫

## 正常的访问速度

尽量保证一次加载页面加载且数据请求最小化，尽量每个页面访问增加时间间隔

## 隐含输入字段

（隐含字段主要阻止爬虫自动提交表单）

在 HTML 表单中，“隐含”字段可以让字段的值对浏览器可见，但是对用户不可见（除非看网页源代码），阻止网络数据采集的方式主要有两种。

第一种是表单页面上的一个字段可以用服务器生成的随机变量表示。如果提交时这个值不在表单处理页面上，服务器就有理由认为这个提交不是从原始表单页面上提交的，绕开这个问题的最佳方法就是，首先采集表单所在页面上生成的随机变量，然后再提交到表单处理页面。

## “蜜罐”（honey pot）

如果表单里包含一个具有普通名称的隐含字段（设置蜜罐圈套），比如“用户名”（username）或“邮箱地址”（email address），设计不太好的网络机器人往往不管这个字段是不是对用户可见，直接填写这个字段并向服务器提交，这样就会中服务器的蜜罐圈套。服务器会把所有隐含字段的真实值（或者与表单提交页面的默认值不同的值）都忽略，而且填写隐含字段的访问用户也可能被网站封杀。

# 验证码识别

1、图片降噪

降噪就是把不需要的信息通通去除，比如背景，干扰线，干扰像素等等，只剩下需要识别的文字，让图片变成2进制点阵最好。

对于彩色背景的验证码：每个像素都可以放在一个5维的空间里，这5个维度分别是，X,Y,R,G,B，也就是像素的坐标和颜色

2、图片切割

3、图像文本输出

# 动态渲染页面抓取

DHTML是Dynamic HTML的简称，就是动态的HTML，是相对传统的静态HTML而言的一种制作网页的概念。DHTML其实并不是一门新的语言，它只是HTML , CSS和客户端脚本的一种集成，即一个页面中包括HTML+CSS+JavaScript(或其他客户端脚本)。

DHTML不是一种技术、标准或规范，只是一种将目前已有的网页技术、语言标准整合运用，制作出能实时变换页面元素效果的网页设计概念。比如，当鼠标移至文章段落中，段落能够变成蓝色。

使用Requests访问一个网页，返回的Response内容和在浏览器上看的HTML内容不一样时就是用了动态技术

动态渲染的页面可以通过直接分析 Ajax ，借助于 Requests 或 Urllib 实现数据的抓取；

但是JavaScript 动态渲染的页面不止 Ajax 这一种，可以直接使用模拟浏览器运行的方式来实现，可见即可爬。

## Ajax数据爬取

Requests 获取的都是原始的 HTML 文档，而浏览器中的页面则是页面又经过 JavaScript 处理数据后生成的结果，这些数据的来源有多种，可能是通过 Ajax 加载的，可能是包含在了 HTML 文档中的，也可能是经过 JavaScript 经过特定算法计算后生成的。

对于第一种情况，数据的加载是一种异步加载方式，原始的页面最初不会包含某些数据，原始页面加载完后会会再向服务器请求某个接口获取数据，然后数据再被处理才呈现到网页上，这其实就是发送了一个 Ajax 请求。这样在 Web 开发上可以做到前后端分离，而且降低服务器直接渲染页面带来的压力。

### 基本原理

Ajax，全称为Asynchronous JavaScript and XML，即异步的JavaScript和XML。

它不是一门编程语言，而是利用JavaScript在保证页面不被刷新、页面链接不改变的情况下与服务器交换数据并更新部分网页的技术。

对于传统的网页，如果想更新其内容必须要刷新整个页面， Ajax可以在页面不被全部刷新的情况下更新其内容。

在这个过程中，页面实际上是在后台与服务器进行了数据交互，获取到数据之后，再利用JavaScript改变网页，这样网页内容就会更新了。

1. **发送请求**

Ajax由JavaScript实现，新建了XMLHttpRequest对象，调用onreadystatechange属性设置了监听，调用open()和send()方法向某个链接（也就是服务器）发送了请求。

var xmlhttp;  
if (window.XMLHttpRequest) {  
 // code for IE7+, Firefox, Chrome, Opera, Safari  
 xmlhttp=new XMLHttpRequest();  
} else {// code for IE6, IE5  
 xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");  
}  
xmlhttp.onreadystatechange=function() {  
 if (xmlhttp.readyState==4 && xmlhttp.status==200) {  
 document.getElementById("myDiv").innerHTML=xmlhttp.responseText;  
 }  
}  
xmlhttp.open("POST","/ajax/",true);  
xmlhttp.send();

实现请求发送之后，可以得到响应结果，但这里请求的发送变成JavaScript来完成

由于设置了监听，所以当服务器返回响应时，onreadystatechange对应的方法便会被触发，然后在这个方法里面解析响应内容即可。

1. **解析内容**

得到响应之后，onreadystatechange属性对应的方法便会被触发，利用xmlhttp的responseText属性便可取到响应内容。

返回内容可能是HTML，可能是JSON，接下来在方法中用JavaScript进一步处理。

1. **渲染网页**

通过document.getElementById().innerHTML这样的操作，便可以对某个元素内的源代码进行更改，网页显示的内容就改变了，这样的操作也被称作DOM操作，即对Document网页文档进行操作，如更改、删除等。

document.getElementById("myDiv").innerHTML=xmlhttp.responseText便将ID为myDiv的节点内部的HTML代码更改为服务器返回的内容，这样myDiv元素内部便会呈现出服务器返回的新数据，网页的部分内容看上去就更新了。

这3个步骤是由JavaScript完成的，它完成了整个请求、解析和渲染的过程。

JavaScript向服务器发送了一个Ajax请求，然后获取新的数据，将其解析，并将其渲染在网页中。

真实的数据其实都是一次次Ajax请求得到的，如果想要抓取这些数据，需要知道这些请求是怎么发送的，发往哪里，发了哪些参数。

### Ajax分析方法

Ajax有其特殊的请求类型，其 Type/触发原因 为 xhr



xhr对应详细信息中，X-Requested-With：XMLHttpRequest，标记了此请求是 Ajax 请求。

查看响应数据，用程序来模拟这些 Ajax 请求就可以了。

### Ajax结果提取

1. 分析请求

分析xhr请求的参数

1. 分析响应

观察响应内容，分析需要提取的内容

## 模拟浏览器运行

selenium