网络爬虫的设计与实现

摘　要

本课题的主要目的是介绍网络爬虫的历史和发展历程，介绍了一些网络爬虫中常见算法，根据算法实现简单的网络爬虫，并对网络爬虫的动态抓取策略进行探究。

网络爬虫出现的不是很晚，但它是在最近结合大数据时代得以实现迅猛的发展。爬虫的种类很多，但不同种类爬虫的区别实际上就是对资源的取舍问题。基本网络爬虫通过Python语言中的BeautifulSoup对网站资源进行分析，获得相应URL；利用Set集合的无重复性对url进行去重。随着互联网的发展，以AJAX为代表的动态刷新技术越来越受到网站开发者的喜爱，所以原本的抓取方法无法抓取动态的页面。为了适应不同需求，编制网络爬虫之前我们需要根据预先设定的目标对资源所存在的网站进行细致的分析。我分别采用了两者不同的方式来实现对动态目标进行抓取。

关键词： 网络爬虫、遍历算法、数据抓取

#### The design and implementation of a web crawler

#### Abstract

The main purpose of this project is to introduce the history and development of Web crawler, introduces some common web crawler algorithm, according to the algorithm to achieve a simple web crawler, and to exexplore dynamic web crawler crawling strategy.

Web crawler appears not very late, but it is in conjunction with the recent era of Big Data to achieve rapid development. Many species of reptiles, but differentiating between different types of reptiles actually means trade-offs of resources. The basic web crawler uses the BeautifulSoup in the Python language to analyze the site's resources to get the corresponding URL;，Use the non-repeatability of the Set to load the url. With the development of the Internet to AJAX as the representative of the dynamic refresh technology more and more by the site developers love, so the original crawling method can not crawl the dynamic page. In order to meet the different needs, the preparation of network crawlers before we need to set the target according to the existence of the resources of the site for detailed analysis. I used the two different ways to achieve the dynamic target to crawl.

**Key words:** Web crawler, calendar calculation method, the data capture

目　　录

摘要 1

Abstract 1

1. [绪论](#_第一章_绪论)

第一节 [课题背景](#_第一节_课题背景)

第二节 [课题意义](#_第二节_课题意义)

第三节 [网络爬虫的历史](#_第三节_网络爬虫的历史)

第四节 [网络爬虫的分类](#_第四节_网络爬虫的分类)

1. [相关技术背景](#_第二章_相关技术背景)

第一节 [网络爬虫的定义](#_第一节_网络爬虫的定义)

第二节 [搜索策略剖析](#_第二节_搜索策略剖析)

第三节 [开发方法概述](#_第三节_开发方法概述)

1. [基本网络爬虫模型的分析和设计](#_第三章_基本网络爬虫模型的分析和设计)

第一节 [基本流程分析](#_第一节_基本流程分析：)

第二节 [百度百科词条爬虫设计](#_第二节_百度百科词条爬虫设计)

1. [需求确定](#_一、需求确定：)
2. [分析目标](#_二、分析目标：_首先观察词条页面的URL的特点)
3. [系统设计](#_三、系统设计：)
4. [代码编写](#_四、代码编写：)
5. [爬虫运行](#_五、运行爬虫)

第三节 [动态加载技术分析](#_第三节_动态加载技术分析：)

第四节 [花瓣网图片抓取爬虫设计](#_第四节_花瓣网图片抓取爬虫设计：)

1. [需求确定](#_一、需求确定：_1)
2. [分析目标](#_二、分析目标：)
3. [系统设计](#_三、系统设计)
4. [代码编写](#_四、代码编写)
5. [爬虫运行](#_五、爬虫运行)

第五节 [QQ空间说说的抓取](#_第五节_QQ空间说说的抓取)

1. [需求确定](#_一_需求确定)
2. [分析目标](#_二、分析目标)
3. [系统设计](#_三、_系统设计)
4. [代码编写](#_四、代码编写_1)
5. [爬虫运行](#_五、运行爬虫_1)
6. [总结与展望](#_第四章_总结与展望)

致谢 53

参考文献 54

# 第一章 绪论

## 第一节 课题背景

网络爬虫（web crawler），是一种按照一定的规则，自动的抓取万维网信息的程序或者脚本。。

网络检索功能与万维网的发展同步进行。网络内容的爆炸性发展同时也带来了对人们对特定内容检索的需求。这就是搜索引擎诞生的原因。而搜索引擎的核心就是爬虫。随着信息不断的丰富,人们的需求也在不断的提高,通过网络进行信息的搜索已经成为人们每天都要进行的内容.如何使搜索引擎能时刻满足人们的需求。最初的检索功能通过索引站的方式实现，而有了网络爬虫这个技术之后，网络爬虫的时代便开始一发不可收拾了。

## 第二节 课题意义

互联网是一个庞大的非结构化的数据库，将数据有效的检索并组织呈现出来有着巨大的应用前景。然而随着互联网规模的上升，在资源丰富的同时也带来了冗余的问题。搜索引擎是一个很好的搜索工具。但作为通用的软件，常规搜索引擎为了增加自己的通用性而不得不减少了一些专业性，这就是常规搜索引擎的的局限性。不同领域、不同背景的用户往往具有不同的检索目的和需求，通用搜索引擎所返回的结果包含大量用户不关心的网页。为了设计更好的完成搜索任务，精确搜索内容，一个灵活的爬虫有着无可替代的重要意义。

## 第三节 网络爬虫的历史

在互联网发展初期，网站相对较少，信息查找比较容易。然而伴随互联网爆炸性的发展，普通网络用户想找到所需的资料简直如同大海捞针，这时为满足大众信息检索需求的专业搜索网站便应运而生了。

现代意义上的搜索引擎的祖先，是1990年由蒙特利尔大学学生Alan Emtage发明的Archie。因为当时距离万维网的创建还有四年，但在网络对文件传输还是有很大的需求的，所以ftp（文件传输协议）软件成为了共享文件的主要工具。而且由于大量的文件散布在各个分散的FTP主机中，查询起来非常不便，因此Alan Archie工作原理与现在的搜索引擎已经很接近，Archie恰恰可以自动索引Internet上匿名的免费FTP文件信息，并提供一种根据文件名称查询文件所在FTP地址的方法。因此，Archie被称为现代搜索引擎的祖先。然而，客观的讲，它和一个真正的网络爬虫有着一些区别。原因有两个：一是它只能搜索FTP文件资源，并不能获取诸如网页等其他类型的文件资源，因此它其实是世界上第一个FTP爬虫。二是它没有机器人（Robot）程序，不能象今天的搜索引擎那样快速有效的抓取Internet上的网页文章内容，相反它使用的是一个基于脚本的文件名称收集器，并通过正则表达式来匹配用户查询与文件名称来实现查询，并通过文件列表的方式提供信息查询结果。

当时，“机器人”一词在产业界中十分流行。作为新兴的技术，爬虫自然被人们赋予了机器人的名字和期盼，“机器人”（Computer Robot）是指某个能以人类无法达到的速度不间断地执行某项任务的软件程序。由于专门用于检索的“机器人”程序象蜘蛛一样在网络间爬来爬去，寻找一个个的“猎物”—也就是信息。因此， 搜索引擎的“机器人”程序就被称为“蜘蛛”程序。世界上第一个可以被称之为网络爬虫的程序是Matthew Gray开发的World wide Web Wanderer。它的功能很简单，是用来度量互联网大小的，同时也可以用来统计互联网上的服务器数量，后来则发展为能够检索网站域名。Wanderer不仅可以说是一个真正意义上的网络爬虫，更可以说是第一个为万维网设计的爬虫，即万维网搜索引擎。而与Wanderer相对应，Martin Koster于1993年10月创建了ALIWEB，它是Archie的HTTP版本。ALIWEB不使用“机器人”程序，而是靠网站主动提交信息来建立 自己的链接索引，类似于现在我们熟知的Yahoo，作为一个爬虫来说，这种抓取信息的方式太过于被动，很多用户认为像网站提交信息就是浪费，这是ALIWEB不成功的地方，但对于整个网络爬虫的发展是很重要的一步，其开发者Martin通过这次失败预见性的思考到了不能任由程序设计者随意抓取数据，其参与设计的机器人拒绝协议也成为了当今爬虫的设计规范。

随着互联网的迅猛发展，使得直接遍历所有新出现的网页变得越来越困难，因此，在Matthew Gray的Wanderer基础上，一些有先见之明的编程者将传统的“蜘蛛”程序工作原理作了些改进。其设想是，类似于蜘蛛的行为，不进行无意义的闲逛，而是通过一些可能保存信息的通路，慢慢找到目标信息。如果按照计算机网络的说法，就是既然每个网页都可能有连向不同相关性的网站的链接，那么从一个标准的页面开始搜索，就有可能检索整个互联网。到1993年底，一些基于此原理的搜索引擎开始纷纷涌现，其中以JumpStation、The World Wide Web Worm（Goto的前身，也就是今天Overture），和Repository-Based Software Engineering(RBSE) spider最负盛名。

JumpStation的Web bot从网页上收集了关于标题和标题的信息，并为其Web界面使用了一个非常简单的搜索和检索系统。系统线性搜索数据库，匹配关键字。不用说，随着网络越来越大，JumpStation变得越来越慢，终于停止了。WWW蠕虫仅索引其访问的页面的标题和URL。它使用正则表达式来搜索索引。JumpStation和蠕虫的结果按搜索顺序排列，意味着结果的顺序完全不相关。RSBE蜘蛛是通过实现基于与关键字字符串相关性的排名系统，因此说RBSE是第一个在搜索结果排列中引入关键字串匹配程度概念的引擎 最早现代意义上的搜索引擎出现于1994年7月。当时Michael Mauldin将John Leavitt的蜘蛛程序接入到其索引程序中，创建了大家现在熟知的Lycos。同年4月，斯坦福（Stanford）大学的两名博士生，David Filo和美籍华人杨致远（Gerry Yang）共同创办了超级目录索引Yahoo，在开始时，他们称自己的网站为雅虎收集界面，但随着链接数量的增加，他们的网页每天开始获得数千次点击，团队创造了更好地组织数据的方法。为了帮助数据检索，雅虎（www.yahoo.com）成为可搜索的目录。搜索功能是一个简单的数据库搜索引擎。由于Yahoo!条目已被输入并手动分类，所以Yahoo!并未被真正归类为搜索引擎。相反，它通常被认为是可搜索的目录。雅虎自动化了收集和分类过程的一些方面，模糊了引擎和目录之间的区别。并成功地使搜索引擎的概念深入人心。从此搜索引擎进入了高速发展时期。目前，互联网上有名有姓的搜索引擎已达数百家，其检索的信息量也与从前不可同日而语。比如在国外拥有统治地位的Google，其数据库中存放的网页已达30亿之巨。

网络爬虫是搜索引擎的核心，窥一斑而知全豹，从搜索引擎的发展过程中，我们也可以看出来网络爬虫的发展步伐。一部搜索引擎史也就是网络爬虫史。从开始的简单抓取到现在的各种大规模的分布式、高度集群化等等。虽然距爬虫的出现才不带三十年的历史，但这样的发展速度是其他各行业都无法比拟的。这也凸显学习爬虫相关技术的重要性。

## 第四节 网络爬虫的分类

根据不同的应用，爬虫系统在许多方面存在差异，大体而言，可以将爬虫划分为如下三种类型：

　　一、批量型爬虫(Batch Crawler)：批量型爬虫是一种具有明确目标的爬虫，当爬虫完成任务的程度达到某个设定的目标后，即停止抓取过程。至于具体目标可能各异，根据情况而定。也许是设定抓取一定数量的网页即可，也许是设定抓取消耗的时间等。比如当下某些流行的头条类网站，其基础的爬虫架构就是批量型爬虫，爬取一定时间内的信息。总而言之，其区别于增量型爬虫的最大特点就是有限制。

　　二、增量型爬虫(Incremental Crawler)：谈到增量型爬虫，就不得不提待累积型爬虫，其实两者很相似，核心目标都是要维持一个庞大的网页的“仓库”。它们与批量型爬虫不同，会保持持续不断的抓取，对于抓取到的网页，要定期更新。累积型爬虫主要完成的是遍历存储全部的所能够访问的网页，而增量型是指在具有一定量规模的网络页面集合的基础上，采用更新网页的方法，在生成的网页“仓库”中通过一定的算法选择出过时网页进行重新抓取，用较小的代价来保证整个数据库中的网页与真实[网络数据](https://www.baidu.com/s?wd=%E7%BD%91%E7%BB%9C%E6%95%B0%E6%8D%AE&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9PvfkuHN-n16LnH0dujwB0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPH0LnHRdn1n1)足够接近。当然，可以进行增量式抓取的前提是系统已经抓取了足够数量的网络页面，并具有这些页面被抓取的时间信息。因为互联网的网页处于不断变化中，新增网页、网页被删除或者网页内容更改都很常见，而一个合格的爬虫需要保证的就是数据的时效性和可靠性。所以增量型处于持续不断的抓取过程中，不是在抓取新网页，就是在更新已有网页，通常情况下既包括累积式抓取，也包括增量式抓取的策略。累积式抓取一般用于网页集群的整体建立或大规模更新阶段；而增量式抓取则主要针对数据集合的日常维护中。通用的商业搜索引擎如百度、谷歌等爬虫基本都属此类。

　　三、垂直型爬虫(Focused Crawter)：以上两种谈到的爬虫，无论是批量型还是垂直型爬虫，它们聚焦的地方都是网页的数量选择，对于网页的内容不做太多的关心。而垂直型爬虫和两者切入的角度正好不同，垂直型爬虫多关注特定主题内容或者属于特定行业的网页，比如对于跑步社区网站来说，只需要从互联网页而里找到与跑步相关的页面内容即可，其他行业的内容不在考虑范围。垂直型爬虫一个最大的特点和难点就是：如何识别找到的URL所指向的网页内容是否属于指定行业或者主题。从节省系统资源的角度来说，不太可能把所有互联网页面下载下来之后再去筛选，这样做和增量型爬虫无异，设计出垂直型爬虫就是为了减少系统资源的浪费。因此，爬虫的设计者往往需要在抓取阶段就能够动态识别某个网址是否与主题相关，并尽量不去抓取无关页面，以达到节省资源的目的。很多专题网站或学科搜索引擎都是使用的这类型的爬虫。

# 第二章 相关技术背景

## 第一节 网络爬虫的定义

一、网络爬虫是一个自动下载网页的程序，它可以从Web上下载网页，是搜索引擎的核心组成部分。通用网络爬虫从一个或若干给定的标准网页的URL开始，获得初始网页上的URL列表；在抓取网页的过程中，不断从已下载页面上抽取新的URL放入待爬行队列，直到满足系统的停止条件。  
　　二、主题网络爬虫是基本网络爬虫的升级，它根据一定的网页分析算法将从网页中获取的全部URL剔除掉与主题无关的链接，保留主题相关的链接并将其放入待抓取的URL队列中；然后根据一定的策略从队列中选择提取要抓取的网页URL，并重复上述过程，直到满足系统的停止条件。所有被网络爬虫抓取的网页将会被系统存储到数据库中，进行一定的分析、清洗，并根据主题建立相应的索引，对于主题网络爬虫来说，这一过程所得到的分析结果还可能对后续的抓取过程进行反馈和指导。

## 第二节 搜索策略剖析

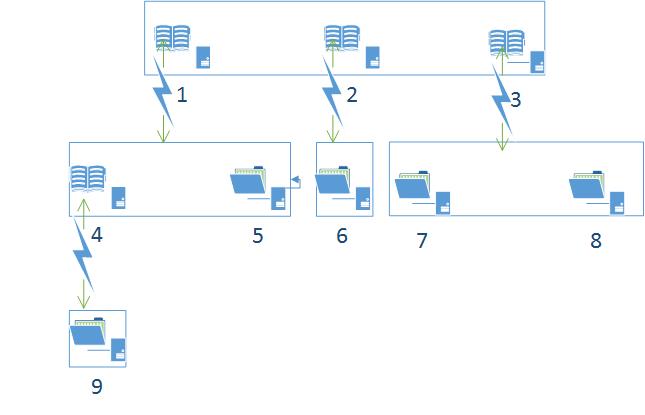
从上面的网络爬虫定义可以看出，网络爬虫的核心其实就是如何遍历所有需求的URL。通过一个给定的超链接，我们可以明确经由一个网站到另一个网站。这其实是就是有向图的遍历方法，接下来，我们就以这两种遍历方法作为切入点，剖析一下常见的搜索策略。

## 深度优先搜索策略:深度优先搜索是在开发爬虫早期使用较多的方法。在数据结构中，它的定义是假设给定图G，在G中任选一顶点v为初始点(源点)，首先访问出发点v，并将其标记为已访问过；然后依次从v出发搜索v的每个可以访问点w。若w未曾访问过，则以w为新的出发点继续进行深度优先遍历，直至图中所有和源点v有路径相通的顶点(亦称为从源点可达的顶点)均已被访问为止。若此时图中仍有未访问的顶点，则另选一个尚未访问的顶点作为新的源点重复上述过程，直至图中所有顶点均已被访问为止。这是数据结构中的定义，而在爬虫中，每个定点其实就是每个URL，它的目的是要达到被搜索结构的叶结点，也就是不包含任何URL的HTML文件）。在一个HTML文件中，当一个超链接被选择后，被链接的HTML文件将执行深度优先搜索，即在搜索其余的超链接结果之前必须先完整地搜索单独的一条链，形成一条从起始链接到最终节点的一条通路。深度优先搜索沿着HTML文件上的超链接走到不能再深入为止，然后返回到某一个HTML文件，再继续选择该HTML文件中的其他超链接。当不再有其他超链接可选择时，说明搜索已经结束。其优点是能遍历一个Web站点或深层嵌套的文档集合。缺点是因为Web结构相当深，有可能造成一旦进去再也出不来的情况发生。所以在实际的设计之中往往要根据所要求的数据对深入的层数进行限制，防止出现上面的情况。

二、宽度优先搜索策略:在数据结构中，有向图的广度优先遍历算法是一个层次遍历的过程，它根据所有节点与起始节点的距离的长短划分成一个个的层次。这和树的层序遍历算法十分类似，它也需要一个队列以保持遍历过的顶点顺序，以便按出队的顺序再去访问这些顶点的邻接顶点。而在爬虫的宽度优先搜索中，先搜索完起始页面中所有的超链接，这些超链接因为在一个页面上，距离当前访问页面的距离相同，所以可以看成是同层页面。然后再继续搜索下一层，直到底层为止。例如，一个HTML 文件中有3个超链接，选择其中之一，处理相应的HTML文件，然后不再选择第二个HTML文件中的任何超链接，而是返回，选择第一个页面里的第二个超链接，处理其中包含的全部的URL，再返回，选择第三个超链接，并处理其中的URL信息。至此，第一层的可以说是全部处理完毕，可以处理第一个链接中所包含的URL了。也就是说，一旦一层上的所有超链接都被选择过，就可以开始在刚才处理过的HIML文件中搜索其余的超链接。这就保证了对浅层的首先处理。这样做有两个很明显的好处，首先当程序遇到一个无穷尽的深层分支时，不会导致爬虫陷进WWW的深层结构中导致出不来的情况发生。宽度优先搜索策略还有一个优点，它能在两个HTML文件之间找到最短路径。宽度优先搜索策略通常是实现爬虫的最佳策略，因为它容易实现，而且具备基本完备的功能。但是如果要遍历一个指定的站点或者深层嵌套的HTML文件集等类似这种具有一定“线性特征的数据结构时”，用宽度优先搜索策略则需要花费较长时间才能到达深层的HTML文件。

三、聚焦搜索策略：前面两个提到的都是基于数据结构中图的遍历而相应产生的算法。而在实际应用中不会只是使用一种方法，而是两者兼而有之。对于一个具体问题来说，需要爬取的范围都是确定的。所以，如何将这种选择以代码形式显现就很重要了。聚焦爬虫就是一种多方面考察的爬行策略，它不只考虑页面与初始的拓扑结构，而是根据需求确定出某个特定主题的页面，根据“最好优先原则”进行访问，快速、有效地获得更多的与主题相关的页面，最后再通过内容与该页面上的链接结构指导进一步的页面抓取。简而言之，聚焦爬虫的核心是会给它所下载的页面一个评价分，根据得分排序插入到待抓取队列中。完成对当前页面的搜索后，根据评分的高低次序从队列中弹出下一个页面进行分析，这种策略保证爬虫能优先抓取那些最有可能成为高“价值”的页面。聚焦网络爬虫搜索策略的关键是如何评价链接价值，即人们为其设计的链接价值的计算方法，不同的价值评价方法计算出的链接的价值不同，表现在程序当中的就是链接的被访问的次序不同，从而决定了不同的搜索路径。由于链接是页面内容的一部分，因此通常具有较高价值的页面包含的链接也具有较高价值，因而对链接价值的评价有时也转换为对页面价值的评价。这种策略通常运用在专业搜索引擎中，因为这种网站具有一定的专业性，关注的是某些特定话题，对信息在广度和宽度上都有一定的要求，所以和聚焦搜索策略十分类似。

这样分析下来，我们为其实前两种搜索策略是聚焦搜索策略的一种拓展，广度优先搜索就是认为和网页处于同层的网页最具有价值。所以表现在遍历顺序上就是这样。深度优先搜索遍历同理。所以我认为爬虫的一个设计重点就是如何将需求中的价值以代码的形式表示出来。



通过上面文字的描述，让我们对三种搜索方式有了一个概念性的理解。下面我们结合一个具体的网络结构图来简单说明一下。左图（给图起名字）是一个简化了的网络结构图。每一个图标都表示这一个URL，其中样子为书状的表示这是一个只想网页的超链接，而样子为文件夹状的表示这是一个指向文件的链接（比如FTP链接）每个方框表示一个页面。闪电连线表示链接。从图上可以看出，1、2、3这三个超链接在同在初始网页上，4、5在一1所指向的网页上，6在2所指向的网页上，7、8在3所指向的网页上，9在4所指向的网页上。

第一个是深度优先搜索策略，我们就联想数据结构中的深度优先搜索遍历，在爬虫抓取到1后，立刻将1放到待抓取URL列表中。然后爬虫程序根据1找到所指向的页面。在1中有4和5两个URL,还是重复上面的步骤，在检索到4后立刻将其放到待抓取URL中，继续对4所指向的网页进行抓取，在9中有超链接，但是指向的不是一个新的网页，而是一个具体的文件。在爬虫完成对4所指向的网页的抓取后，因为并不包含其他的新的URL，所以爬虫程序回溯到上一层，对5所指向的网页进行抓取。就这样，爬虫逐渐的完成对整个网络的抓取。最后对各个链接的遍历顺序应该是:

1->4->9->5->2->6->7->8（修改公式格式）

接下来是广度优先搜索策略，我们也结合广度优先搜索遍历算法来看。与上面的方法不同，在检索到初始网页上的第一个链接1后，继续完成对初始网页的搜索。直到完成对初始网页的抓取，并将1、2、3三个URL全部放入待抓取URL队列中。然后从队列中取出第一个元素，也就是超链接1，对其所指向的网页进行搜索，得到四五后分别将其按检索到的顺序入队。检索完1所指向的网页后，继续从队头取出元素。根据爬虫所遍历的顺序，这时取出来的应该是2。如此不断执行，也可以完成对整个网络的抓取。最后对各个链接的抓取顺序应该是：

1->2->3->4->5->6->7->8->9(修改公式格式)

最后就是有关聚焦搜索策略的相关内容了，这是最贴合实际的情况。我们假设要抓取所有html文件和其时间，对其他的内容不做关心。也就是衡量一个超链接的价值的是这个超链接所指向内容的类型，对于需求来说，这个价值很好实现，如果指向的是html网页，就置其价值为1（将其放入待抓取URL链接中）；如果超链接指向的是其他类型的文件。就置其价值为0（不放入待抓取URL队列中。那么，如何判断这个URL指向的是html还是其他类型的文件呢？这就要具体问题具体来看了，在一个维护良好的网站系统中，不同类型的文件在其URL可以有明确的判断方式，如果在不访问的前提下就可以判断类型是最优的。当然，如果判断不出来可以进行试探访问以判断其类型，但这就没有前一种高效了。把如何衡量价值的问题解决后，抓取起来就很简单了，开始时类似于广度优先搜索遍历，在初始网页中抓取到1、2、3这三个超链接后分别对其进行判断。结果检查，都是网页，遂将它们全部置入待抓取URL队列中。然后从队头取出1，对其所指向的网页进行遍历，检索到了4、5两个URL。经过检查，发现5指向的并不是HTML文件，遂将其抛弃，仅将4加入待抓取URL队列中。就这样，一边抓取，一边对所抓取到的URL进行评估。直到爬虫完成对整个网络的检索。最后对各个链接的抓取顺序应该是：（5、6、7、8、9因为指向的是非html文件因而被抛弃。）

1->2->3->4(修改公式格式)

## 第三节 开发方法概述

### 所使用的语言：

（一）：在本次的课程设计中，我主要使用Python作为自己的开发语言。作为脚本语言的一种，Python有着其他翻译型语言不能相比的优势。Python非常简单，非常适合人类阅读。阅读一个结构良好的Python程序就感觉像是在读正常的文章一样，只不过这篇文章有着自己的特殊的语法结构，Python的这种易读的特性是它最大的优点之一。它使你能够专注于解决问题而不是把时间浪费在语法的纠结上。Python虽然是用c语言写的，但是它摈弃了c中让人头痛的指针，简化了Python的语法，使得无论是有没有编程经验的人来说，Python语言学习都十分简单。Python是FLOSS（自由/开放源码软件）之一。简单地说，你可以自由地发布使用Python语言构建的项目、阅读其他的项目的源代码、对它做你认为需要的改造。开源是计算机发展的趋势，因为只有让更多的人可以自由的参与进来，这种语言才有发展的趋势。Python也是如此，它很年轻，也希望看到能有更加优秀的人来不断的创新。作为一门出生在20世纪末的语言，在目睹了着从面向过程向面向对象的变迁后，Python语言既支持传统的面向过程的函数编程也支持着新兴面向对象的抽象编程。在面向过程的语言中，程序是由过程或仅仅是一个个孤独的函数构建起来的，重用性不高。在面向对象的语言中，程序是由很多相互之间有联系的、包括着各种数据和方法对象所搭建起来的。与其他主要的语言如C++和Java相比，Python以一种简单而又有力的方式实现面向对象编程。⑻丰富的库。Python标准库确实很庞大。Python有可定义的第三方库可以使用。它可以帮助你处理各种工作，包括正则表达式匹配、单元测试、线程、数据库、网页浏览器、FTP、XML、XML-RPC、HTML、GUI（图形用户界面）、以及其他多种的功能。记住，只要安装了Python，所有这些功能都是可用的。除了标准库以外，还有许多其他高质量的库，如BeautifulSoup、Requests和matplotlib等等。通过这些强大的第三方库，可以很容易的实现有关爬虫的相关操作。最后一点，Python语言写成的代码都十分的规范。这是因为Python采用强制缩进的方式，这种在语法上的强制约束使得代码具有极佳的可读性，不会因为个人的习惯导致代码的可读性变差。

（二）、Java（根据后续开发情况确定）

（三）、R，进行数据分析（根据后续开发情况确定）

### 二、使用到的关键类库：

（一）BeautifulSoup： 首先BeautifulSoup提供了一些简单的方法和Pythonic成语，用于导航，搜索和修改解析树；其次用于剖析文档并提取所需内容的工具包。编写应用程序不需要太多代码BeautifulSoup会自动将传入的文件转换为Unicode和传出的文档到UTF-8。您不必考虑编码，除非文档没有指定编码，而BeautifulSoup不能检测到。那么你只需要指定原始编码；最后BeautifulSoup是流行的Python解析器，如[lxml](http://lxml.de/)和[html5lib](http://code.google.com/p/html5lib/)，可以让您尝试不同的解析策略或交易速度来实现灵活性。相比Java中提供的文档解析工具httpparser，dom4j来说。我认为BeautifulSoup的功能更强大、也更灵活。符合Python语言自己的特点，通过匹配不同的parser，可以完成包括html、xml、html5等不同文件的解析，它可以把各种标签直接分析出来，而不像dom4j等需要一层层的查询。这样对于复杂网站的处理来说，大大加快了速度，提高了运行效率。而且对于非英语网站来说，BeautifulSoup的自动处理编码也大大简化了程序员为了处理乱码问题所需要的工作。

（二）Requests： Requests库是用[Python](http://blog.csdn.net/shanzhizi)语言编写的，基于 urllib（一个Python的原生类库），采用 Apache2 Licensed 开源协议的 HTTP 库。它比 urllib操作起来更加方便，可以节约我们大量的工作，完全满足程序员对HTTP的各种 需求。重要的一点是它支持 Python3 ，所以使用起来比其他的有关http库更加规范，安全性也更强。在我的设计中，涉及到了有关动态网页内容的抓取，BeautifulSoup因为只能保存转换成html的相关文件，因此就帮不上忙了。这时就需要Requests来帮忙，它里面包含的内容可以处理各种请求，这就大大方便了我们程序员进行操作。

### （三）re： re是Regular expression operations的缩写。该模块提供正则表达式匹配操作。正则表达式有能力去指定一系列与所给表达式在形式上匹配的字符串。Re模块是Python的内置模块，此模块包含的各类函数可以让程序员以多种方式来检查所检查的一些类字符串是否匹配给定的正则表达式或者是给定的正则表达式去匹配特定的字符串。这虽然看上去是正反两个不同的过程，但是其实可以归结为相同的事物。Python通过re模块提供对正则表达式的支持，使用re模块一般是先将正则表达式的字符串形式编译成Pattern对象，然后用Pattern对象来处理文本得到一个匹配的结果，也就是一个Match或Search对象，最后通过group方法得到所匹配的结果并进行操作。Re库的强大之处不仅在于是正则式的匹配，更可以在生成Pattern对象时通过指定各种匹配模式，来实现更强大的功能。

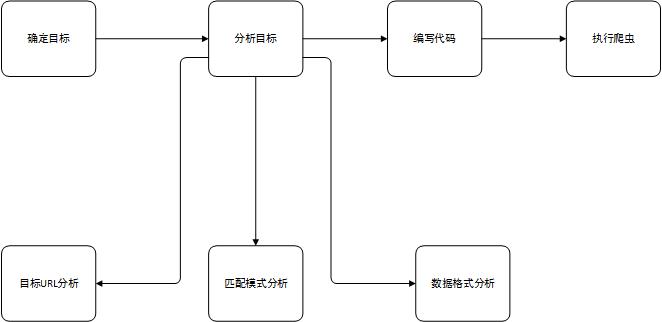
#### （四）httpclient（Java的类库，依据后面的程序情况书写）

#### （五）matplotlib（Python的绘图类库，依据后面的程序情况书写）

三、使用的数据库（暂时没有使用数据库，以后在程序完善后继续添加，现在想的是使用Mysql存放爬取的数据）

# 第三章 基本网络爬虫模型的分析和设计

## 第一节 基本流程分析：



爬虫设计流程图（待格式化）

这是一个普通的爬虫从开始到工作的流程。首先根据需求，明确所要抓取的具体目标。第二步就是分析目标。在明确具体的需求后，我们应该明确待抓取对象的URL的格式，以明确要抓取的范围。如果这个URL没有能够明确的指定的话，那我们的爬虫程序就会抓取很多与本任务无关的互联网上的其他网页，造成资源的浪费。其次，在明确了目标URL的特点后，就是进行匹配模式的分析了。这是我们就会用到正则表达式，这是一个很有效的工具，通过对正则表达式的合理创建，我们可以高效的从所有备选URL中筛选出我们所需要的那一部分。最后一项就是待抓取数据的格式。爬虫当然不会只抓取URL，链接只是一个路径，更重要的是抓取网页上的数据，所以待抓取数据的分析也很重要，不仅要考虑数据的格式，大小等基本信息。对于中文的抓取，更要考虑到处理编码的问题。第三步就是代码段编制了，再完成了有关分析的相关操作后，就是要根据上一步的分析得到的策略来编写代码了。从这些流程可以看出，这些步骤中最重要的就是分析这一步了。应予以多加留意。

## 第二节 百度百科词条爬虫设计

在了解完爬虫的相关知识后，我们以一个具体的实例来研究一个基本的爬虫是怎样诞生的。

### 一、需求确定：

百度百科是百度公司推出的一部内容开放、自由的网络百科全书平台。截至2017年4月，百度百科已经收录了超过1432万的词条，几乎涵盖了所有已知的知识领域。百度百科是一个涵盖各领域知识的中文信息收集平台。所以，我们想以百度百科中太原理工大学的页面<http://baike.baidu.com/item/太原理工大学>作为爬虫的入口地址，爬取太原理工大学词条页面以及与之相关的1000个百度百科的词条网页，并将这些网页的URL、词条名称以及简介都保存下来。

### 二、分析目标： 首先观察词条页面的URL的特点



太原理工大学百度百科的词条页面（待格式化）



211工程百度百科的词条页面（待格式化）

从上面的几个URL的格式可以看出，词条页面的URL是有一定特点的。都是以http://baike.baidu.com/item/为开头，具体所指向的页面在其后加上网页的主题即可。结构十分清楚明了。

然后是观察网页中所包含的URL的相关信息。通过浏览器所带的检查工具，我们可以观察页面上面每个部件的特点。

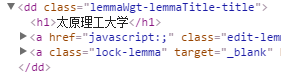


“211工程”的链接格式

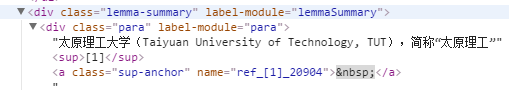
通过观察，我们可以看出。网页上所包含的链接都是不完整的。要想访问这些链接，我们需要在截取后手动的向前面添加baike.baidu.com来构成一个完整的URL。通过这两部我们就明确了目标URL的具体格式，完成了目标URL的分析。

第二步就是匹配模式的分析了，我们从第一步了解到，网页上所有指向词条的页面的都是以/item开头的后面接着一系列的内容。所以匹配模式也很简单，就是从网页上面的所有链接中筛选出以/item开头的链接就可以了。

最后是数据源的分析，我们还是通过检查元素来观察整个待抓取信息的结构。



审查元素-太原理工大学标题（待格式化）



审查元素-太原理工大学简介（待格式化）

从上面的审查元素可以看出，词条的标题是处在一名字叫dd的标签之中，class的名字叫做lemmaWgt-lemmaTitle-title。而简介的内容处在一个div中，其对应的class的名字是lemma-summary。在我们获取数据的时候，就可以通过类名来直接找到这两个部分的内容了。



页面编码（待格式化）

最后，我们通过审查元素再确认一下页面的编码方式，为UTF-8，所以在接下来的接收过程中程序要以以UTF-8的形式解码再处理，这样就避免了乱码问题的发生。

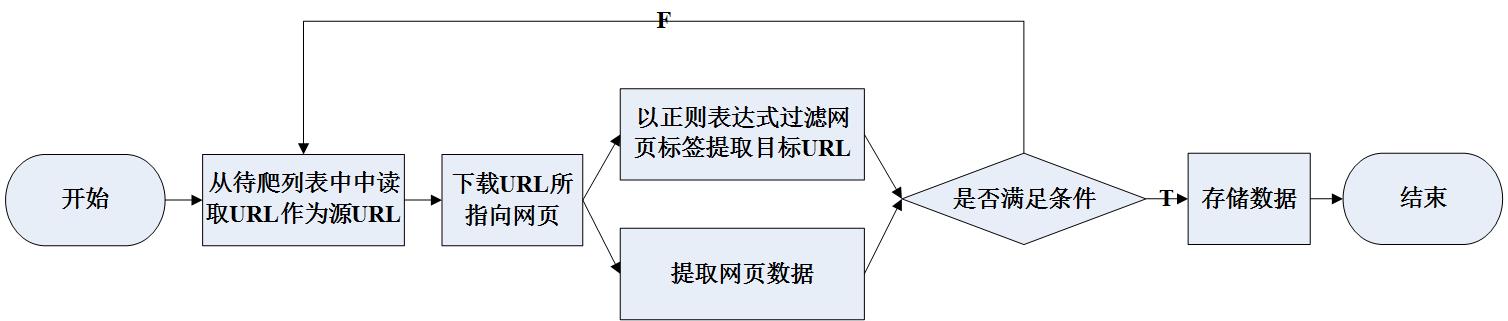
至此，我们对百度百科的网站的抓取策略已基本构建完成。

|  |  |
| --- | --- |
| 目标 | 百度百科太原理工大学相关词条页面-标题和简介 |
| 入口页 | <http://baike.baidu.com/item>太原理工大学 |
| URL格式 | http://baike.baidu.com/item/XXXX |
| 数据格式 | 标题：<dd class="lemmaWgt-lemmaTitle-title"> |
| 简介：<div class="lemma-summary" label-module="lemmaSummary"> |
| 页面编码 | UTF-8 |

抓取策略表（待格式化）

### 三、系统设计：

首先，应确定整个程序的工作流程



因为本程序抓取的网页数量较少，所以没有使用数据库存储待抓取URL列表，直接存储在set集合中，也就是内存之中，这样也可以增加程序的运行速度；网页上的数据也保存在一个对象之中（内存）。最后的数据存储也是准备通过一个方法调用Python的文件输出流，创建一个html文件并将目标数据写到所新建的html文件中，只要用户有浏览器就可以查看信息。（后期考虑将数据同时存放在数据库之中）

根据本网络爬虫的设计，本网络爬虫是一个自动提取网页的程序，根据从待爬队列中抽取的网页进行下载，再根据一定的策略将其中的符合要求的URL和页面上的数据同时保存下来，如此循环往复直到满足用户的需求或者网页抓取完毕。

（后期考虑增加一个有关的图）

### 四、代码编写：

首先是编写主函数，内容很简单，就是通过创建一个调度类的对象，开始网页的抓取。

然后就是调度类（spidermain）的编写，因为他是调度类，所以在本类的构造函数中先要创建分别创建URL管理器、HTML下载器、HTML解析器和HTML输出器的对象。创

创建好四个对象后，就要开始爬虫主程序的编写。在上一步已经完成了对流程的分析，所以直接根据流程图撰写代码，就可以完成有关爬虫调度程序的编写。

核心代码如下：

通过URL管理器获得一个新的连接

new\_url =*self*.urls.get\_new\_url();

将刚刚获得的地址以参数的形式传入HTML下载器，下载URL所指向的界面

html\_cont=*self*.downloader.download(new\_url)

通过调用HTML解析器解析所下载网页中所包含的URL和数据

new\_urls,new\_data=*self*.parser.parse(new\_url,html\_cont)

存储URL和数据

*self*.urls.add\_new\_urls(new\_urls)

*self*.outputer.collect\_data(new\_data)

打印数据到HTML文件中

self.outputer.output\_html()

这样调度类的编写就完成了，调度类没有什么业务逻辑，但是指导着整个爬虫程序的进行，十分关键。

接下来是URL管理器（UrlManager）。这个方法的内容很简单，主要由4个方法构成add\_new\_url(self,url)，向管理器中新增加一个URL， add\_new\_urls(self,urls):向URL管理器中增加一系列的URL， has\_new\_url(self):判断URL管理器中是否还有待爬取的URL，get\_new\_url(self)获取一个新的URL。这四个方法没有什么复杂的地方，就是注意一点，由于互联网上各个界面可能存在着指向对方的链接，所以在遍历完一个网页后，不能简单的删除掉，而是应该在整个程序中维持两个集合。一个是待访问集合，而另一个是已访问集合。

核心代码：url not in self.new\_urls and url not in self.old\_urls:

当调度程序调用URL管理器向程序中新增已爬取的网页时，要判断这个链接是不是已经在待爬取集合内或是不是已经访问过一次了（在已访问集合中）。在访问完网页后，不仅将这个URL从待爬取集合中删除，更要将其添加到已爬取集合中，这样才能保证数据的完整性。多设置一个队列可能会对多占用一些内存空间，但带来的好处不仅仅只有减少因重复访问同一网页带来的资源浪费，更在一定程度上保证了程序的稳定性，防止程序调入循环抓取的陷阱中。

当URL管理器给出了一个待抓取的URL后，就轮到HTML下载器工作了。下载器负责的是将网页的内容下载下载，就和我们平常使用浏览器浏览网页一样。只不过我们只是将对应的html文件下载下来，不进行解释。这只需要调用urllib库下的request对象的urlopen方法，再讲其中的内容通过调用read（）返回给调度程序。但值得注意的是，因为在先前的设计当中，我们了解到百度百科的全部网页的编码方式都是UTF-8，所以在回传下载好的HTML文件时，要注意通过提供的decode方法将解码方式设置为使用UTF-8解码。这样做可以减少乱码问题发生的可能性。

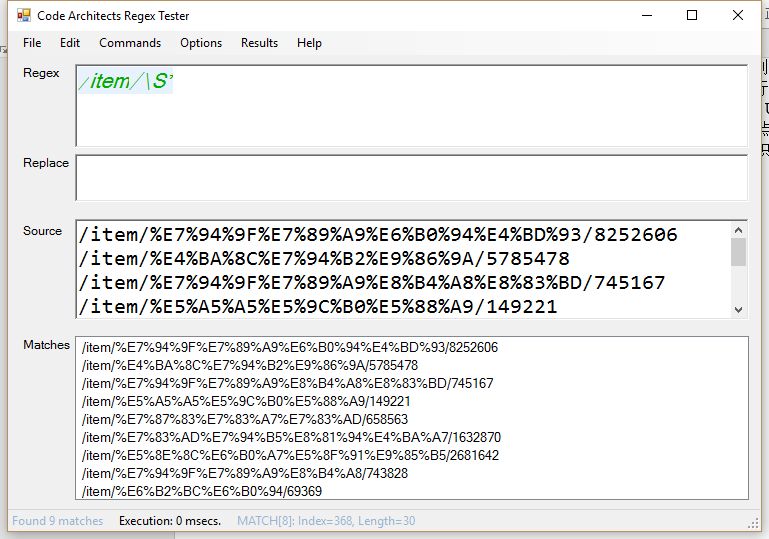
在得到网页的文件后，调度程序就要利用HTML解析器来解析下载好的文件了。HTML解析器的编写与我们之前分析所制定的策略息息相关。

HTML有一个函数是调度函数直接调用的，就是parser函数。调度程序要依据这个函数，输入下载好的HTML文件，得到解析后的数据。我们先利用BeautifulSoup提供的方法将传入的HTML文件转化成BeautifulSoup对象。然后我们可以将这个函数具体实现拆成两个小函数来分别实现，第一部分负责从HTML文件中解析符合条件的超链接。第二部分负责从HTML文件中解析出目标数据即百科的标题和简介。这两部的执行没有关联，所以可以分开进行。有条件还可以将程序做成多线程以进一步提高效率。

第一部分是抓取所有符合条件的URL，也就是那些依旧指向百度百科词条的超链接。

links = soup.find\_all('a', href=re.compile(r"/item/"))

这句话通过BeautifulSoup所提供的的find\_all方法，可以轻松找到网页中的所有超链接。但咱们所抓取的只是百度百科的页面，再find——all方法中不行要对标签的类型进行限制，还要对超链接标签的href属性进行约束。通过上面的目标URL分析的结果中我们知道，百度百科词条页面上的指向其他词条页面的超链接的特点：href属性都是以“/item/”开头的，结尾是内容而定。所以利用正则表达式的相关知识，同时结合目标字符串的特点，写出了/item/\S\*的正则表达式。



正则表达式检验（待格式化）

通过正则表达式检测工具，证明表达式书写争取，可以正确匹配到目标网页。

获取到所有百科词条的URL后，因为这是不完整的，不可以直接存储，否则会造成无法访问的问题。所以接下来程序要对生成的links集合进行遍历，修改每一个链接，让他们变成完整的链接。

new\_full\_url = parse.urljoin(page\_url, new\_url)

这里所使用的不是简单的字符串的拼接，而是使用了一个urljoin的函数，这个函数有两个参数，返回值是将第二个参数所代表的字符串根据第一个参数的格式进行转化后的类型。其实可以理解成相对路径和绝对路径的转化。第二个是相对路径，第一个是绝对路径。这个函数完成的就是从相对路径到绝对路径的转化。

通过匹配、补全这两步。HTML解析器完成了第一部分的任务-将HTML文件中所包含的指向其他界面的URL解析出来。

第二部分是从HTML文件中解析出词条的标题和简介。

因为前面分析工作做得很好，所以这部分解决起来也很容易。

title\_node =soup.find('dd',class\_ = "lemmaWgt-lemmaTitle-title").find("h1")

res\_data['title']=title\_node.get\_text()

summary\_node =soup.find('div',class\_ = "lemma-summary")

res\_data['summary']=summary\_node.get\_text()

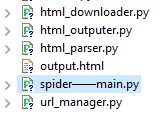
还是通过调用BeautifulSoup提供的find方法，同时指定标签名和类名两个属性来获取对应的节点。再得到节点后直接使用get\_text()方法来获取节点的内容。因为这两者都已经是根节点了，所以可以使用get\_text()方法来获取值，这里调用.string的属性值也可以达到相同的效果。

通过BeautifulSoup提供的方法，我们很快的完成了第二部分的任务-将HTML文件中所包含的词条标题内容和简介内容解析出来，并存放到一个字典类型的数据中。而且为了日后可以编写的根据这个标题和简介打开自己的目标网站，程序将词条页面的URL作为一项数据和词条标题、词条简介共同保存了起来。

因为爬虫要不断进行循环抓取，所以在上一步获得的数据需要立刻保存起来。因此在完成了解析，返回了具体的URL数据和网页数据后。URL管理器和HTML输出器分别将URL和网页数据保存起来。URL管理器的增加方法没有值得注意的地方，批量增加新的URL就是对每个URL调用前面所编写的增加一个URL的方法即可，还增加了重用性。就是有关HTML输出器，在循环爬取的过程中，因为程序还没有结束，所以它首先要保证每次从HTML解析器中获得的字典类型的数据都保存下来。所以程序应该在HTML输出器中定义一个集合对象，每次爬取完一个网页后，就将从这个网页中所提取出的数据保存起来。直到整个爬虫停止后，再一并输出。

当经过了网页获取，网页下载、网页解析和数据转存后。爬虫的一次爬取工作就算完成了，接下来就是要根据用户的需求和网页中所包含的链接数再进行不断的重复爬取了。当爬虫停止后，就该调用HTML输出器的output\_html()方法，将前面每次爬虫所收集的数据打印输出到HTML文件中。这里没有什么问题，唯一要注意的就是注意编码，在声明HTML时要制定好编码方式为UTF-8，这样才与Python保持一致，否则将会产生中文乱码问题。

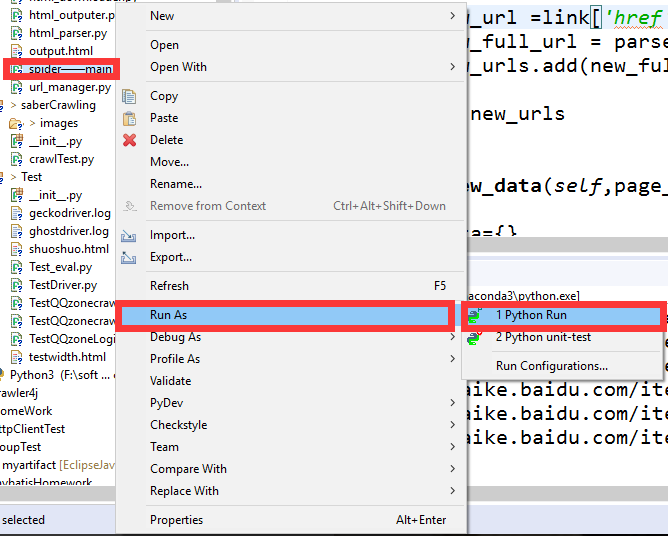
### 五、运行爬虫



百度百科爬的文件结构。

其中spider——main.py为程序的入口地址，里面包括着主函数以及爬虫的调度类。url\_manager.py为URL管理器、html\_downloader.py为HTML下载器、html\_parser.py为HTML解析器、html\_outputer.py为HTML输出器；output.html为程序最后的输出文件。

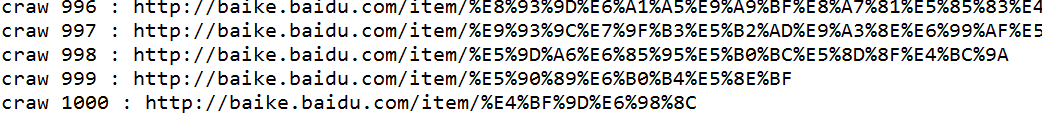
运行步骤:



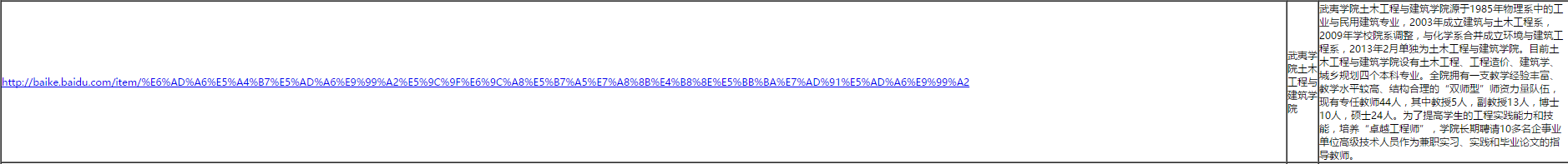
程序执行步骤（待格式化）

右击main函数，选择Python Run。程序即开始执行。

在经过一段时间后，程序运行结束。



程序运行结果（控制台）（待格式化）



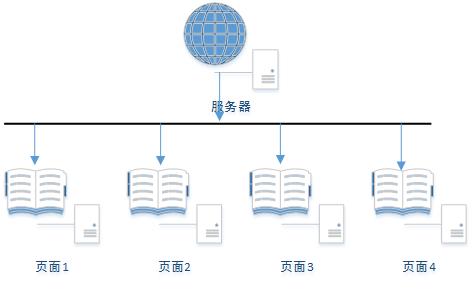
程序运行结果（HTML文件）（待格式化）

最终的结果表明我们就完成了一个基本爬虫的构建。从需求到最后的实现，核心就是分析阶段。无论是设计什么软件，重要的地方都是有关于设计。只要分析好了目标，设计好了结构，后面的编码都是轻松解决的。

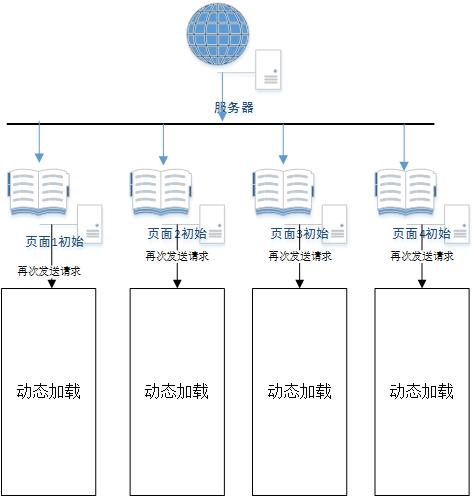
这个百度百科的抓取程序只是对静态页面的抓取，下面我们分析两种关于动态页面的抓取方法。

## 第三节 动态加载技术分析：

自从推特首创的下拉刷新这一模式后，越来越多的网站特别是社区类网站采用这一模式。在浏览网站上的内容时，它们不仅将内容分页，更将每页包含的内容分成几个部分，当客户请求的时候再加载，提高了交互性。这样做，不仅可以让用户更自由的选择浏览的数量，更可以通过这种将大任务分成几部分的小任务，从而可以更好的均衡服务器的负载水平。



如图所示的就是传统的网站的结构图，百度百科就属于这种结构。从图中我们可以看出，服务器中包含着每个网页自己的文件服务器。当浏览器发出请求后，返回的就是整个页面。适合内容较少的页面，如百科类这种以文字作为主要内容的网页。

如图所示的就是现在大部分的社区类网站的结构图，如花瓣网、QQ空间等。从图中可以看出，当浏览器向服务器发出一个请求后，服务器向用户返回一个网页，但这个网页不是全部显示的，而是先显示一小部分，而本页的其他内容是动态的根据用户的需求加载。这种架构适合于社区网站。因为作为社区类网站，网页的内容不仅仅包含文字，还可能包含着很多的图片甚至视频，如果一次将一个网页的内容全部加载下来再提供给用户来看的话，会极大的增加用户的等待时间，影响用户的体验，所以可以通过这种动态加载的方式，让用户边浏览，边加载，提高用户的体验，同时也增加了用户的访问速度，达到均衡服务器的目的。

这是两种网页的架构模式，只有弄清楚所要抓取的网站属于哪种模式，在设计爬虫时根据相应的类型设计，才能达到高效便捷的目的。像百度百科属于第一种方式，所以在设计抓取方式时，直接将页面下载再分析就可以得到网页上的内容了。但如果要抓取动态网页就要根据不同网站的加载的方式来进行具体的分析了。

## 第四节 花瓣网图片抓取爬虫设计：

### 一、需求确定：

花瓣网是一家图片分享网站，是一家基于兴趣的社交分享网站，网站为用户提供了一个简单地采集工具，帮助用户将自己喜欢图片重新组织和收藏。花瓣网的模式是采用瀑布流的形式展现图片内容，无需用户翻页，新的图片不断自动加载在页面底端，让用户不断的[发现](http://baike.baidu.com/item/%E5%8F%91%E7%8E%B0)新的图片。我们的目标就是根据某一个选定的主题，自动的下载指定数量的相关图片。这次就以家装作为主题，我们去花瓣上下载100张相关图片。

### 二、分析目标：

首先先观察目标页面的URL：







几个搜索结果页面URL

http://huaban.com/search/?q=\*\*\*是搜索结果的结构，前半部分是固定的，改变的只是最后的等号后的词，对应着搜索的内容，也就是说，如果要抓取某个方面的主题，就相应的将最后的关键词替换成相应的词就可以了。

在了解到了目标网页的URL后，接下来的就是如何将网页上的图片下载到本地上的问题了。平常我们的下载都是利用浏览器自带的功能，获取到图片对应的URL后，下载到本地。网络爬虫也是同理，我们需要先明确图片的格式，然后获取100张图片的链接。最后根据这些链接将图片下载下来。

为了保证下载的图片的质量，我们选择下载原始图片而不是图片的缩略图。为了了解原始图片的格式，我们同时查看几张图片对应的缩略图和原始图的URL，用以明确原始图片的URL构成：







原始图片的URL  

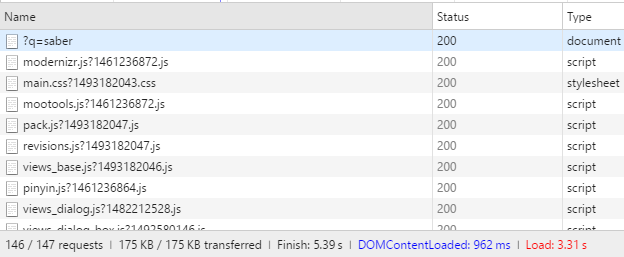





缩略图的URL

通过这两者的比较，我们可以清楚的看出来原始图片的URL构成，前半部分是固定的花瓣网的图片标识：<http://img.hb.aicdn.com；URL>最后是固定的代表原始图片的表示：\_fw658。而中间的一部分是代表着具体图片的标识（key）。所以，现在问题的关键就转向如何获得具体图片的标识了(key)。

和百度百科相比，花瓣网的网页内容较多，构成复杂，直接观察网页的源码难以看出，我们可以利用浏览器中自带的检查工具来观察加载网页时加载的内容，以此来寻找图片key值的获取方式。

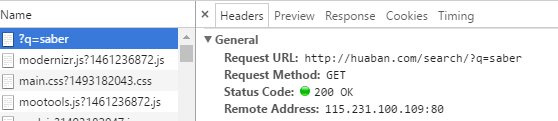


初次加载网页所加载的内容

可以看出，初次加载网页时接受的对象有很多，其中大部分是脚本，这是因为开始请求的网页，除了图片外，还有网站本身的框架、广告、插件等内容，所以有很多的脚本。但是第一项名为?=saber的内容很值得人注意。它的名称正好就是搜索网址最后一部分的内容，而且等号后面的内容也正是搜索的关键词。

所以我们点开这个document对象来仔细分析一遍。

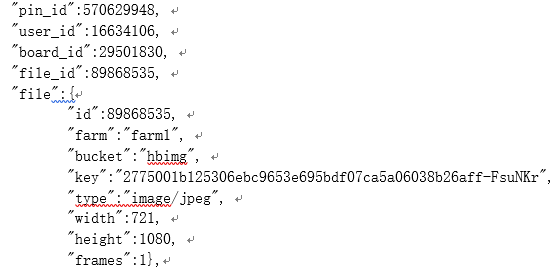
首先分析请求头部。



请求头部

通过图片可以看出，这个请求的URL正是搜索网页的URL，这也进一步证明了我们的猜想。这个document对象中包含的内容和图片的信息有很大的关系。那为了获取到图片的具体信息，我们需要进一步去翻看通过这个请求响应回来的具体内容。因为返回的也是一个html文件，为了查看分析方便，我们把这些内容复制到一个文本编辑工具里面查看比较方便。

通过对内容的大概浏览，我们发现这些变量名称的定义都很规范，通过他们的名称就可以大概理解这些变量的具体作用，类似于setting、scheme、count等等。可以看出这些都是有关网页格式的限定，而非图片信息。经过仔细寻找，我们发现了一个名为app.page["pins"]的列表，里面保存着多个字典，在这些字典中有着很多的名称很醒目的键值对，如pin\_id、user\_id和key等等，这些名字中带有id的属性应该就是唯一标示图片的具体信息。我们可以利用获得的这些信息来反过来验证猜测。



一个图片的相关信息（部分）

通过检查元素审查网页，再结合我们在上面响应中的得到的信息，利用搜索的方法，我们可以了解到:pin\_id是一个图片的具体的标识，而其中file字典中的key正是我们需要找的那个决定文件存储路径的key值。这两个属性是我们所必须的，有了图片的id和其对应的的key后，我们就可以自动构建图片的下载链接了。

据此，我们可以总结获得图片URL的基本步骤：

1. 向http://huaban.com/search/?q=（主题词）发送请求
2. 接受到相关请求后解析html，从中截取出名为app.page["pins"]
3. 遍历app.page["pins"]中每个元素，获得每个元素的key值。
4. 组合成图片的URL

在解决了如何获得一个图片的下载地址后，然后要解决的就是本爬虫最关键的部分了。作为图片分享网站，花瓣网对于一个搜索主题并不是直接显示全部的搜索结果，而是先显示一部分，当用户下拉网页到当前所加载内容的最后部分时，网页再加载一部分图片。所以，当我们向http://huaban.com/search/?q=saber发送了请求后，其中的app.page["pins"]所包含的仅有部分图片，也就是第一次加载的图片。所以，现在的关键就成了如何模仿用户的下拉操作，继续向服务器请求图片。

我们继续使用检查工具来完成这个任务。我们可以利用检查工具中的网络分析部分，观察在下拉的时候向网站发送了哪些请求、收到了哪些响应以及响应中包含了那些内容。



下拉时收到的响应

从图上可以看出，在下拉网页时，浏览器收到了一个xhr对象，该对象的名字和第一次加载网页时浏览器所收到的那个document对象类似。由此推断两者的内容也应该类似，但第二个对象的获取是在网站的基本设置已经加载完成后，所以内容中有关的网页的设置应该会减少，而主要是关于图片的相关信息。这些就是根据该对象的名称所进行的推测，具体的详情还应该要根据具体的相应内容来判断，但依据自己学习到的内容进行一定程度上的推测，可以让我们在筛选内容时更加具有目的性。

第二个值得注意的地方就是XHR对象，和第一次的document对象不同，第二次接收的不在是静态的文本对象，经过上网查阅相关资料，XHR是AJAX功能实现所依赖的一个对象，该对象的全称是XMLHttpRequest。XMLHttpRequest 对象提供了对 HTTP 协议的完全的访问，包括做出普通的get请求的能力以及各种高级的请求。XMLHttpRequest 可以自由的选择返回 Web 服务器的响应的方式，包括同步和异步，并且能够以文本或者一个 DOM 文档的形式返回内容。简而言之，这个对象返回的也是一个文本文件，但是属于动态生成的。这也正是AJAX的特点，在不更新网页整体的条件下，对局部进行内容进行更新，从而减少网络资源的消耗，提高用户的体验。避免了类似于那种一页页翻页的网站出现的资源重复加载所带来的浪费问题。既然XHR对象返回的内容属于文本，那就很简单了。首先，需要找到xhr类型的url地址，然后根据地址发送相应的AJAX请求，最后解析响应中的内容。

那首先就是确定请求的URL了，和上面类似，我也进行多次的加载来观察请求URL的特点：









几次请求的URL

通过观察我们可以发现，获得XHR对象的请求URL的特点很明确。改版的就只有page参数。改变方式为第一次为2，随着每次刷新逐次的加1。从这里也可以看出来每次加载的图片是20张，据此我们可以确定20张图片是一组，在确定好下载的数量后，就可以相应的确定出需要下载的组数。我们还可以看出来这里发送请求所使用的方式是GET方法。为了确定自己的结论，我又更换了自己所查询的主题词来观察不同主题词下动态加载的URL是否有所区别。



通过比较可以看出，刚才的结论没有问题。在每次的请求时注意根据请求的次数来修改page的值。

在解决了获取URL的问题后，接下来的就是有关解析具体内容的问题了。这是观察请求对应的response：

这个response对象中包含的内容也是一个列表，在这个列表中包含着两个字典：第一个是名为query的字典，保存这有关jQuery的相关信息,AJAX作为动态刷新技术，需要配合JQuery来从数据库抓取数据。另一个就是名为pins的字典了，既然属性名和第一次加载网页时包含图片信息的那个字典的名字一样，那它的内容也不言而喻了，两者的结构完全相同，故可以采用相同的解析方法。

至此，对所有链接信息的分析已经基本全部完成了，接下来就是匹配模式分析了。现在的需求是从相应中提取所有图片信息，也就是获得名为app.page["pins"]的字典。但是现在有两种不同的响应，一种是网页初始加载时的静态响应，还有一部分是属于根据需求图片数量而进行的动态请求所获得的响应。这两部都是获取信息的操作，应该整合在一部分，也就是说我们应该寻找一种同时可以符合两种情况的匹配模式，从而可以更好的整合相同部分的功能。

因此，我们先对两种请求的内容使用Python中的requests库内的get方法进行请求，来具体观察特点。结果测试，发现无论是哪种请求所获得的内容在结构上都是相同的。所有的图片信息都包含在名为app.page["pins"]的字典中。这与上面使用浏览器自带的检查工具所抓取的结果不一致，但结果查阅相关资料，表明这两者是一致的，JQuery通过其底层的AJAX对象的get方法返回一个XHR对象，前面使用拦截器发现的XHR对象就是这样生成的，但对于JQuery还没有完成，最后还是要把这对象中的内容添加到网页中，也就是初始加载的网页中。所以两者有着相同的属性名”pins”。

进行完匹配模式的分析后，我们最后对目标格式进行分析。我们本次要抓取的对象是花瓣网上的图片。这就存在两个问题。第一个是网站的问题：随着网站的建设，现在的网站上不仅存在着图片，还包括着视频，故第一个问题是甄别出目标的文件类型。第二个是图片的问题，仅仅对于图片来讲，就包含着很多种格式如JPG、png等等，故第二个问题就是确定所下载图片的具体格式。通过对app.page["pins"]的分析，我发现在其中存在着名为type的属性。



这个属性以下划线分成两部分，第一部分是指定的文件类型，而第二部分则是指定的此文件的文件名。故我们可以抽取出来每个图片信息中的type属性，对下划线两侧的内容进行分别处理，可以同时解决这两个问题。这样数据格式分析的任务也就完成了。

最后，我们通过审查元素再确认一下页面的编码方式，为UTF-8，所以在接下来的接收过程中程序要以UTF-8的形式解码再处理，这样就避免了乱码问题的发生。

至此，对于目标的分析就基本完成了。

|  |  |
| --- | --- |
| 目标 | 花瓣网图片爬虫 |
| 入口页 | <http://huaban.com/search/?q>=关键词 |
| 动态请求格式 | http://huaban.com/search/?q=saber&j1zt9zn0&page=20&per\_page=20&wfl=1 |
| 数据格式 | Image/jpeg 图片/图片类型 |
| 页面编码 | UTF-8 |

### 三、系统设计

经过上面对目标的分析，我们也大致了解任务目标和任务所要采集的内容。如基本爬虫一样，总体的步骤还是收集URL、分析URL、最后下载相关信息。但与百度百科爬虫不一样的是。这里收集URL分为两部分，第一部分是收集主页信息，第二部分是收集动态请求。收集主页信息的方法很简单，因为已经给出了网站的初始地址，利用requests库所提供的get方法直接就可以获得网页的内容。有关动态请求的获取就需要动态构建，即根据请求的次数来动态生成相应的请求URL。

分析URL就是将URL所对应的HTML文件中的有关图片信息下载下来。这里的HTML文件有两个部分，一部分是从初始页中下载的，另一部分是根据网页动态加载的。虽然HTML文件被分为了两部分，但通过上面对于目标的分析我们可以了解到这两个HTML文件的结构是相同的，所找寻找的图片信息在相同名称的一个字典之中。因此，为了提高程序可可重用性，我们可以将对于数据的分析只完成一个模块，然后传入不同的HTML文件来进行解析，将来自不同HTML文件的图片信息存储到一个相同位置。

下载相关信息对于本任务来讲就是将获得的图片的URL下载下来。利用Python所提供的的文件操作可以直接将一个请求的内容直接下载下来，需要注意的就是做好文件操作中可能出现的异常捕获。



爬虫总工作流程

### 四、代码编写

进行完总体的设计过后，就该进行代码的编写了。

首先是主函数的编写，主函数的编写很简单，就是声明一个对象，再调用相关方法。

然后就是具体类的编写了，我们声明了一个名为HuabanCrawler的类，由这个类完成抓取图片的所有操作。

第一部分是接收初始网页和确定下载图片的数量。我们通过类的初始化来完成操作。



其中，homeurl定义了起始网页，也就是搜索详情页的URL。images列表保存所有的图片信息。最后，因为需要将保存好的图片信息下载下来，所以在当前文件夹下建立一个目录来保存图片。

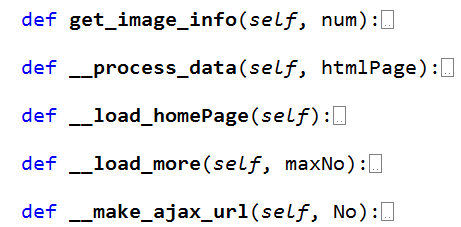
第二部分是获取图片信息。这一部分是整个爬虫的核心，通过这个模块，爬虫需要将全部待下载的图片的信息获取到。我们可以对这部分进行一个分解：



接收图片信息的具体流程

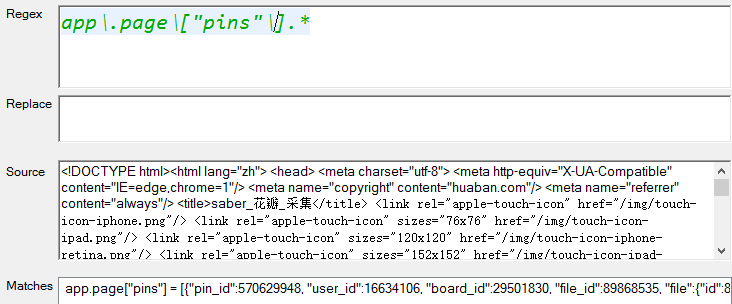
整个的获取图片信息可以分成四步来进行。第一步是将初始页上的图片信息下载下来；第二步是根据所要下载的数量，计算需要请求的次数；第三步是根据计算出来的需要请求的次数，动态的发出请求来获取响应中的网页信息。最后就是根据第一步和第三步中获得的网页信息，解析出其中有关图片的信息保存下来。

为了完成这些功能，我们在类中一共定义了5个函数：



get\_image\_info函数是完成获取图片信息的主要函数，负责完成各个函数的调度，完成这部分的事务逻辑部分。从之前的分析，我们了解到一共有两部分的图片信息。第一部分是初始页面上的，还有一部分信息是在后来动态加载的页面上。所以，这个函数的编写也是分为两个部分，一部分是获取初始页面的的信息。就通过调用函数实现，也不需要传入参数，直接使用初始化时定义的变量就可以。第二部分是就需要进行分析了，在分析目标时，我们就清楚了每个页面的特点。首先，每个页面有20张图片，如果一共需要下载100张图片的话，那么抛去初始页上的20张，还需要动态加载的次数就是（100-20）/20=4次。这样分析后，那整个函数的结构就很明确了，一共分两部分：第一部分调用函数获得初始页上的信息保存起来。第二部分先计算需要下载的次数，然后通过for循环来重复调用函数以获得动态网页上的内容，需要注意的是，根据我们前面的分析，动态请求每次的URL都有不同，page属性是从2开始随调用次数的增加而逐渐增加的，所以我们在调用函数时应该让传入的参数匹配当前进行的请求中的page参数，以免发生请求的URL为不存在的问题。

\_\_process\_data函数是这部分的核心函数，传入的参数是一个网页的源文件，功能是从传入的网页源文件中截取出有关图片信息的部分，即选择出名为app.page[“pins”]的列表，将其内容保存起来，并将其中有关于图片的信息保存下载，最后完成第二部分的业务逻辑功能。首先要完成的，先将网页中保存着的有关app.Page[“pins”]的列表匹配出来，这里用到的就是正则表达式。通过前面的目标分析，我明知道这里的待匹配项目很简单，就是一个固定名称为app.page["pins"]的列表，因此我们只要匹配这个名字加上器后面的内容就可以了。所以我们可以直接写出此表达式为r'app\.page\["pins"\].\*'。我们可以利用正则表达式测试工具来对自己输入的内容进行测试。观察室否可以成功匹配到网页上的相关内容:



模拟正则表达式匹配的结果

从图上我们可以看出，我们设计的这个匹配式可以顺利的完成关键字符串的匹配。但是在匹配之前要注意在匹配前先将页面的解码方式转换成UTF-8，否则会发生编码错误。



防止产生编码错误

在获得了图片信息的相关字符串后，为了可以分类获得各项属性的值，我们需要将这个字符串的序列转换成一个字典。我们可以直接利用python提供的eval函数来直接完成这个操作。eval（str）可以将字符串str当成有效的表达式来求值并返回计算结果。此函数可以自己判断内部字符串符合的是哪种格式，然后进行相应的求值。在上面的测试中，我们可以看出图片信息是符合列表的格式，因此，当调用eval函数后，上面求得的有关图片信息的字符串会自动转换成列表。

但需要注意的是，首先，在Python中，大部分的常量关键字都是以首字母大写的形式存在的，但是在图片信息的列表中存在着好多类似false、true这种纯小写的单词，同时也存在着null这类在Python中没有定义的变量。所以在转换之前，我们需要将一些变量进行提前赋值，将在Python中存在的但非大写的转换成在大写，而在Python中不存在的转换成近义词，防止在转换过程中出现错误。



Python中以None取代了其他语言中的null

在解决完Python中不存在的变量后，还有一个问题需要注意，在前面的分析目标中，我们知道图片信息是存在于一个列表中，该列表中保存着好多各个图片信息的字典。但是我们在转换的时候不需要前面的列表名称，在转换的时候，我们只需要将整个列表的内容截取出来就可以了，这时我们就用到了Python中的切片功能，利用这个功能，可以简单的从字符串中截取一部分出来。该功能类似于C中的字符数组，一个字符串中的各个字符都看成一个数组中的各个元素，利用数字索引来访问。



从上图我们可以看出，需要的内容就是从第一个pin\_id所在的字典开始，也就是从第20个字符开始计算。因为元素的索引从0开始，所以切面因该从19开始，到最后结束，因此我们可以将切片写成[19:-1]。



字符串转换为列表

当通过正则表达式匹配出的字符串成功转换为包含图片信息的列表后，接下来就是将需要的信息从列表中个各个字典中取出了。这里我们用到的就是循环遍历了。



在前面的分析中，我们一共需要从一个图片信息中获取三部分的内容，分别是图片的ID，图片的key值以及图片的类型。这三部分内容都在字典中每个元素中，所以可以直接通过元素的名字来访问。id属性比较简单，没有需要进行操作的内容，直接取出内容存储就可以。第二部分是图片的key值，但在前面的分析中我们知道，首先这个key值在图片字典中的file子字典中，所以需要使用二级索引来访问。其次，在通过索引获得了对应的值后，也不能直接存储，因为key值只是图片下载地址的一部分。所以我们需要手动进行字符串拼接，也就是在key值前面加上搜索详情页的前缀，在key值后面加上完整图片的完整后缀\_fw658。最后一部分的文件类型的判断。我们知道，类型的属性值通过一个下划线分为两部分，前一部分是文件的类型，后一部分是文件的具体类型。所以，对于图片类型的判断需要分成两部分来进行。显示截取出前一部分，判断搜索到的这个文件是不是图片，如果是图片，我们就取出下划线后半部分保存成文件最后的后缀名。如果检查出文件类型非图片，我们就设其类型为默认值。最后，再获得全部一个图片的三部分信息后，将信息存储在预设的变量中，供后面下载模块进行下载。

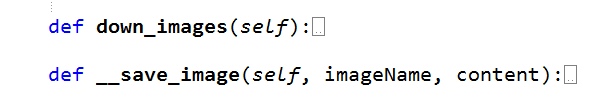
\_\_load\_homePage函数完成的功能很简单，就是将初始页面的内容返回，因此没有额外的参数。

\_\_load\_more函数负责动态加载页面，传入的参数就是当前访问的page的数值，该值的计算由调用它的函数get\_image\_info来完成。该函数和\_\_load\_homepage类似，整个函数中仅有一条语句，也是返回语句，通过调用requests库中的get请求来返回页面的具体内容。但与\_\_load\_homepage不同的是，在get的参数中，填的不是具体的地址，而是一个函数。我们向这个函数中传入具体的页数，然后通过此函数来返回一个对应页数的URL供get方法来请求。这就属于函数式编程，以函数作为参数来传递，优化了流程，让逻辑更加紧密。

\_\_make\_ajax\_url函数的功能就是根据传入的参数，构建相应的URL字符串并返回。在前面的分析中我们已经清楚了动态请求URL的结构，所以该函数完成的就是根据传入的参数和URL的结构，完成相应的拼接操作。

这样，第二部分获取图片信息就编写完了。通过这个模块，我们就可以将网页上存储着的有关图片的信息保存到本地，接下来的就是利用下载下来的信息将位于服务器中的图片信息下载到本地。

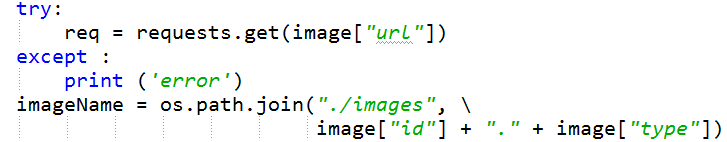
第三部分是下载图片，一共写了两个函数：



下载模块所包括的函数

其中：

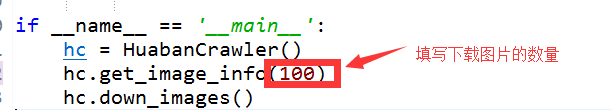
down\_images是下载模块的主函数，负责将从上个模块中获得的图片信息中提取出来，然后按照里面的内容将图片下载下来。在上一步的获取图片信息中，针对每一个图片，我们分别保存了三项信息，分别是该图片的id、通过key值组合成的文件下载地址以及图片的具体类型。所以在这个函数中，我们就将这三种信息从保存信息的列表中取出进行下载。首先利用前一步组合成的图片的具体URL，利用requests库的get方法，从相应的地址中获得对应图片的字节流。这样就获得了文件的具体内容。然后我们就利用图片id和图片的具体类型为相应的图片生成名字，然后再名字前面加入文件的存贮路径，将这个完整的路径保存下载作为后来打开文件操作符的参数。在获得图片的字节流和存储路径后，我们将这些当做参数来调用专门的下载函数来下载图片。



\_\_save\_image函数是负责具体的文件读写，负责将文件下载到本地。传入的参数有一个字节流和一个字符串。该函数利用传入的字符串作为路径参数打开一个文件操作符，然后向这个文件操作符中写入相应的字节流。最后将图片下载到相应的目录中。

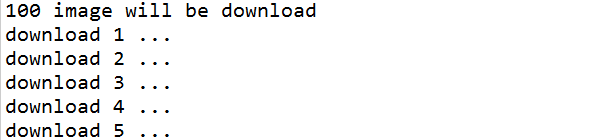


### 五、爬虫运行



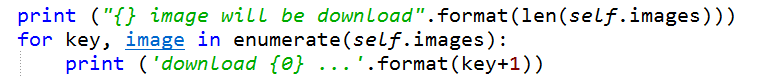


在主函数中填入要下载的图片数量、再确认在类的初始化中确认了搜索的初始网页后，就可以运行爬虫了。

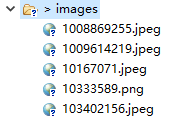
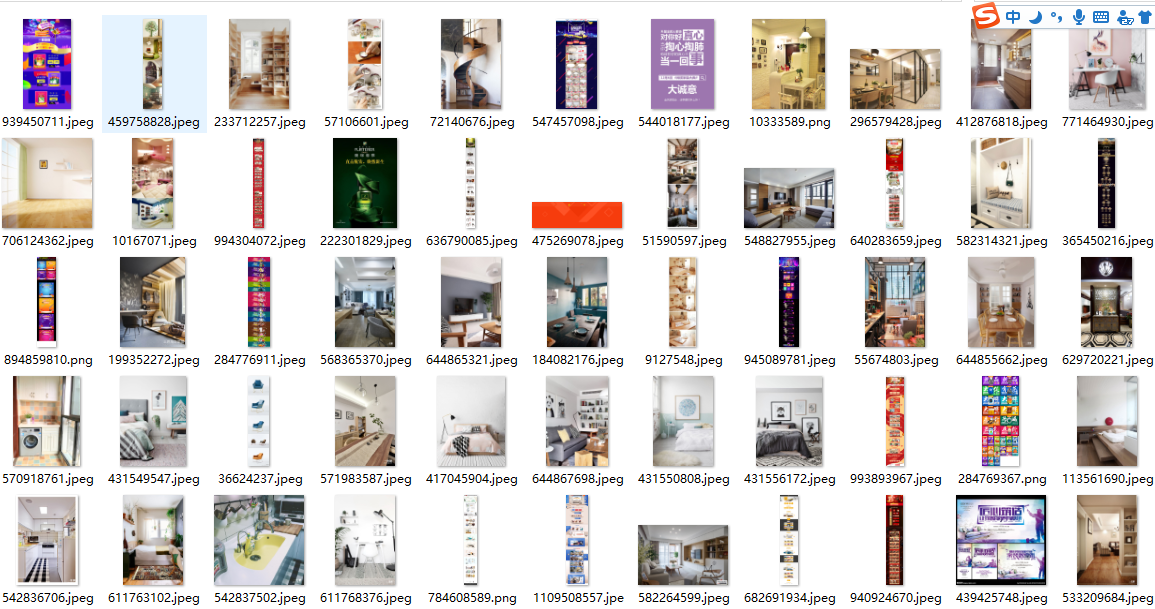


在控制台中的输出信息

在控制台中我们可以看到打印出来的相关信息，这是在down\_images函数中设置的用来检测图片下载情况的语句所运行出来的结果。



然后再项目底部刷新，发现出现新的文件夹，内部保存着下载好的图片。打开资源管理器查看，可以看到已成功下载指定主题的100张图片，与网站比较，内容一致。说明爬虫成功运行。

分别在项目管理目录中和资源管理器中查看的结果

## 第五节 QQ空间说说的抓取

### 一 需求确定

QQ空间（Qzone）是腾讯公司于2005年开发出来的一个具有个性空间，具有博客（blog）的功能，自问世以来受到众多人的喜爱。在QQ空间上可以书写日志、写说说，上传用户个人的图片，听音乐，写心情，.通过多种方式展现自己。也就是说，空间中的内容可以反映出一个人的某些想法与行为特征，因此，我们可以通过抓取某个qq号的全部说说来具体分析某个用户的特点。与花瓣网类似，qq空间中说说的加载也是采用的动态加载技术，但与花瓣网不同的地方是在说说的详情页中，每页的条数有最大的限制，当显示的条数带到当页可显示的最大条数后。在页面下方就会显示对应的翻页按钮，在点击后就会进入新的一页。

总的来说，qq空间中说说的抓取也是一项动态抓取任务。这次，我们使用另一种工具来实现这个操作。Selenium  是Web应用程序测试的工具的一种。Selenium测试直接运行在浏览器中，就像真正的用户在操作一样。我们可以通过调用这个框架下的webdriver来直接实例化一个真正的浏览器，让程序员真切的观察到自己的编写的代码是如何真正的执行的。Selenium作为web测试工具，有着广泛的应用场景。它支持的浏览器有很多，包括IE（7, 8, 9, 10, 11），Mozilla Firefox，Safari，Google Chrome，Opera等。这个工具的主要功能包括：测试与浏览器的兼容性——测试你的应用程序看是否能够很好得工作在不同浏览器和操作系统之上，这也是它之所以要支持多种浏览器的原因。测试系统功能——创建回归测试检验软件功能和用户需求。虽然Selenium是一个主流的测试工具，但我们也可以用它来进行开发活动，虽然可能在运行的时候速度会比其他方式慢，但它带来的好处就是程序在运行的时候非常直观，当程序运行出现错误时我们也可以及时的看到出错的位置而不用再向代码中添加测试语句了。

通过上面的分析，我们已经对Selenium的工作方式有一个基本的认识了，那么在这个爬虫中，我们就通过使用这个框架，完成对一个具体用户的所有说说的抓取。

### 二、分析目标

因为Selenium可以模拟出一个浏览器的具体实现，所以我们可以手动浏览一次某个人的空间来确定爬虫的具体执行步骤。



QQ空间主界面

在进入某用户的主界面后，首先点击某用户主界面的相应按钮，进入该用户的说说界面。

在进入相应页面后，可以看到所有的说说信息都已经出现在下方的框架中。通过向下浏览，我们可以发现每当浏览到当前可显示的最下方之后，网页会自动加载剩下的内容，但浏览到一定程度后，网页就不会继续加载未读的说说记录了，而是在页面的最后显示出下一页的按钮。



页面最下方的按钮

在点击下一页后，我们就重复上面的操作来浏览不同的页面内容。知道浏览完该用户的全部说说内容。当浏览到最后的页面内容后，最下方的翻页处仍有“下一页”按钮的存在，但此时此按钮已变成不可点击的了。



最后一页中最下面的按钮。

通过对一个用户的空间访问的手动模拟，我们也基本明确了程序将要进行的操作。首先是访问一个用户的qq空间的说说主界面，然后就是通过下拉滚动条来获取这个页面上的全部信息。然后就是对当前页面上的全部说说信息进行抓取。再获取完当前用户在该页面上的全部信息后，就通过页面上的下一页按钮来进入下一待抓取页面中，重复前面的下拉-抓取操作直到到达最后一页。

既然总体的流程已经基本确定下来，按照顺序，我们先对初始界面进行分析。我们先打开几个不同用户的说说主界面，来分析一下说说界面的URL特点，因为初始界面将作为日后爬虫程序的入口地址。





几个说说主页的地址

从这些地址可以看出相关的特点，空间主页的地址都是以user.qzone.qq.com开头、以/311作为结尾，中间以下划线隔开的是该用户的qq号。作为入口页的说说主页地址还是很明确的，所以留待确认的就只是目标用户的qq号。当确定了目标用户的qq号后，自然目标用户的说说主页的地址也就可以确定下来了。

当确定了爬虫的入口地址后，接下来就是抓取用户页面的内容了。通过上面的手动模拟过程，我们知道说说的内容也是属于动态加载的，所以也不能直接全部获取到。与花瓣网不同的是，说说页面上的内容不会无限加载，而是有一定的限度。因此我们可以采用与抓取图片不同的方式来获取动态加载的内容。因为Selenium模拟出来的是一个真正的浏览器，所以我们可以使用与手动相同的方式-通过滚动条滚动到页面的最底部来实现将页面的全部内容加载下来的目的。

在滚动完后，就是对当前页面中的内容的获取了。我们要获取的是说说的记录，所以我们选取了说说的发表日期，使用的设备以及发表的内容这三项数据作为我们收集的目标。

首先，通过对说说页面进行审查元素，我们可以发现网页是由多个frame和iframe组成的，因为直接查看网页中的内容比较繁琐，因此为了获得了解到各个frame中的内容，我们可以通过程序将各个框架中所包含的内容输出出来，来确定各个框架中的内容。通过对框架中的内容进行遍历，我们了解到一个叫做app\_canvas\_frame的iframe中包含了所有的说说内容，也就是说这个框架是我们后面爬虫程序主要抓取的位置。



app\_canvas\_framr所处的位置

从app\_canvas\_framre所处的位置我们也可以大致推断出其所包含的内容，因为它处在layout-body->pageContent->pageApp中，里面包含的自然就是有关页面中的主要内容即每条说说的相关信息了。

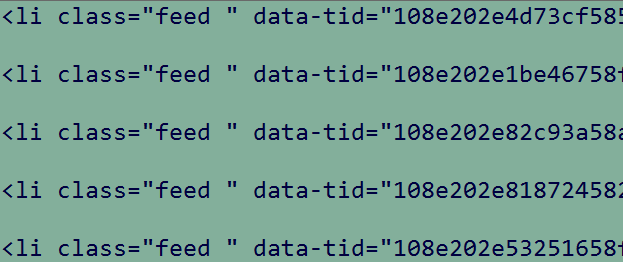
此外，在这个源文件的开始，我们发现了一条xhtml属性的定义。



xhtml属性

通过查阅相关资料，我了解到了这是一条定义命名空间的语句，浏览器会将此命名空间用于该属性所在元素内的所有内容。

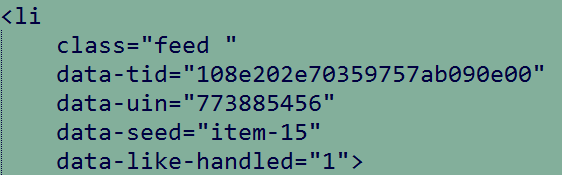
在明确了具体的框架目标后，接下来我们所要关注的就是说说信息在框架代码中的存在形式了。通过对一条说说信息的搜索，我们发现了所有说说的存在位置。每条说说的相关内容都被保存在一个ol的有序列表中，在其中有各种的信息。通过对每条信息的仔细查找，我们发现里面也包含着我们需要的内容，例如说说内容，说说发表时间已经所使用的设备。为了更好的了解将要获取的内容，下面我们对说说的代码进行一个简单的分析。



说说列表

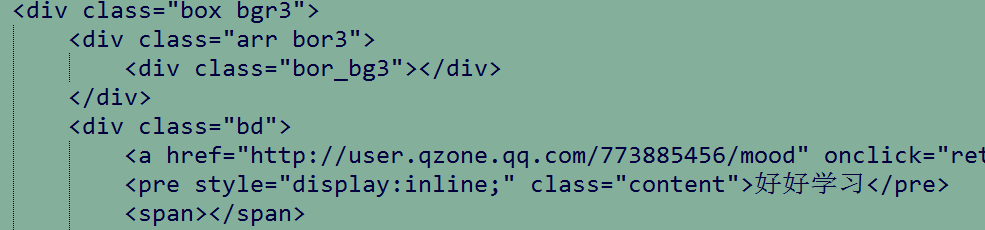
所有说说都被保存在一个有序列表的每个列表项中，每条记录都代表着一条说说的所有相关内容。

平常我们在空间中看到的记录的样子很简单，但是实际到了代码中，想要表示一条记录则需要的代码量可不少，因为一条说说记录包含着很多内容。



说说的具体信息

这些信息是标识一条说说的具体标识，包括class类型、说说的id、说说拥有着的qq号等等。



说说内容的具体位置

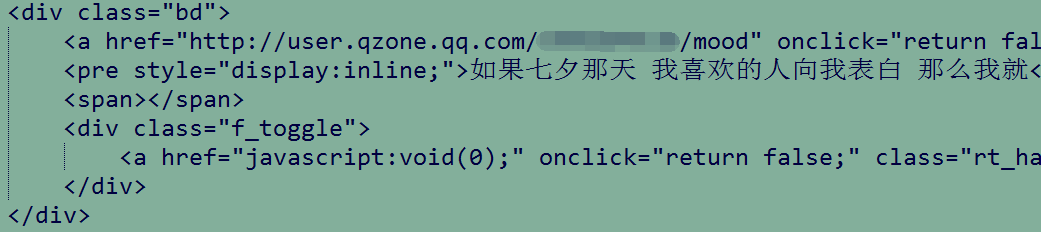
通过比较说说的内容，我们找到了说说内容在说说中的所处位置，如图所示，一条说说的内容在多层div的嵌套之下的一个名为pre的标签中，其类的名称为content，这也与这个标签所包含的内容相符合，表示说说的内容。

我们知道，说说大体可以分为两种：一种是自己发的即原创型说说，还有一种则是转发好友的说说即转载型说说。如果一条说说是转载他人的，那么在记录中肯定会有所体现，这也就说明我们有必要搞清两者之间的不同，从而更好的将二者的内容剥离开来，从而更好的、更具有目的性的进行抓取。



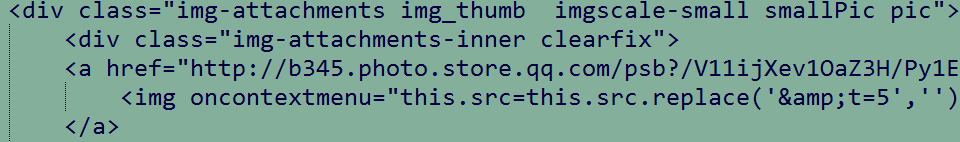
转载内容的div

通过对说说记录的梳理，我们发现一个类名为quote bor2的div就是说说转载前的相关信息。通过其类名中所包含的quote也可以进行确认，一条转载的说说对于转载的用户来说就是被引用的部分。既然被引用的内容也是一条说说，那其中肯定也包含着说说的相关信息：

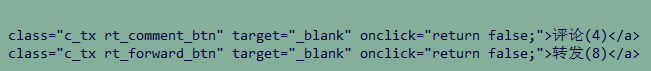


原说说的文字内容

第一部分是一个类名为bd的div，这个div里面主要包括的内容就是被转载说说中的文字内容以及相关操作。与上面用户所发表的内容类似，文字的主要内容都在名为pre的标签之中，但不同的是没有具体的类名标识，这就提示我们在抓取时，不仅要确定标签的名称，同时也需要将类名作为约束放入查找中，这样查找出的内容才真正的是用户所写的内容，而不是所转载的说说中的内容。



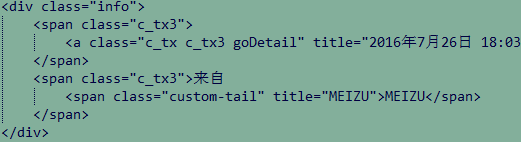
第二部分值得关注的是类名为img-attachments img\_thumb imgscale-small smallPic pic的div和其子div以及其中包括的a标签、img标签，经过与原说说内容的比对，我们发现这个标签下的img标签都是对应于原说说内容中的图片。所以，如果我们想要抓取说说的图片的话，可以以从类名为img-attachments-inner clearfix的div下的全部a标签中将href属性的值提取出来，这样就得到了图片的URL，再利用python的文件流将获得的内容写回本地即可，具体的操作已在上一个爬虫中演示完成，这里就不再赘述了，只是注意一点，这里的图片还都是被转载说说中所包含的图片，而并非用户所发表内容中所包含的图片，真正抓取时应根据自己的需求下载相应的内容。



评论与转发信息

上图显示的是被转发信说说的评论与转发的相关情况，正如类名所指的一个是comment也就是评论的相关内容，而另一个forward则是表示的转发的相关情况。这两个都是被包含在a标签中，其中a标签的href属性则指向的相应的评论、转发页面。如果对这两个的操作所执行的数量关心的话，则可以抓取这两个标签中的text内容中的数字来实现，如果对具体的评论内容、评论人员或者是转发人员关心的话，则可以通过a标签中对应的href属性的值去访问新的页面再进行抓取。

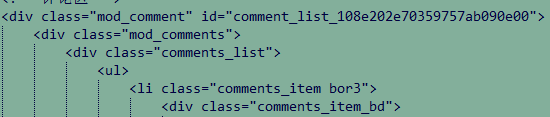
以上分析的三部分的内容都是被转发说说所包含的信息。虽然不一定没个说说都包含着被转发的内容，但是通过分析被转发部分的组成，我们可以了解到一个说说的基本组成。麻雀虽小，五脏俱全。通过这部分的分析，我们基本清楚了一个说说的具体组成部分，从而为接下来的对用户内容的抓取做好了准备。在分析被转发的内容前，我们已经得到了说说的主题内容，接下来我们要分析的就是这条说说内容所包含着的其他信息了。



“info”div的内容

在标识被转发说说的quote bor2的div结束后，接下来的就是说说本体的相关信息了。第一个值得注意的是info的div。在这个div中有着两个十分关键的span，这两个span本身的类名相同，但其中的内容则有很大的区别。第一个span从其title中可以看出其所保存内容为时间，和原说说一对比，发现其包含的正是我们需要的说说的发布时间。所以，在后面的程序中如果需要抓取时间，我们就可以抓取类名为\_tx c\_tx3 goDetail的a标签中的内容，这样就可以获得说说发布的时间了。但还有需要注意的地方是，在这个a标签中有两处包括着时间，一处是在a标签的标签头的title属性中包括着一个时间，另一处实在a标签的内容中。二者都是时间，title属性中的时间具体到了小时和分钟数，而内容中的时间是一个传统的日期格式。在后面的抓取中我们要根据自己的需要相应的获取所关心的数据。

第二个span的内容也很明确，它的类名叫做custom-tail，内容显示为Meizu，表明其内容正是用户发表说说说使用的设备。所以，在后面的程序中如果需要抓取用户发表所使用的设别，我们就可以抓取类名为custom-tail的a标签中的内容，这样就可以获得说说发布的所使用的设备了。与时间相同的是，在获取设备的a标签中也存在两处包含用户设备的位置，一处是title属性而另一处是内容。但与获取时间不同，这两处的设备名称完全一样，所以在获取时不需要进行分辨，选择容易实现的方式就可以满足需求。

在分析完了说说的基本信息后，继续向下审阅，下一部分值得注意的是评论区。

评论区初始结构

从图上可以看出，评论区有一个类名叫做comments\_list的div，里面包含着一个无序列表，其中的每一条记录都是一组评论。之所以是一组评论是因为qq空间将回复评论的评论归为一组，如果我们要统计总评论数的话就不可以只统计无序列表中li标签的个数，这样统计出来的就只是第一评论的数量。

在明确了评论部分的总体结构后，我们继续向下分析评论区的具体结构。总的来说，每条评论都分成了两个大部分，第一部分是用户的首次评论。第二部分是用户对此评论的后续评论。分别在两个div中：



两个div

第一个类名为comments\_item\_bd的div保存的内容是原始评论。第二个类名为mod\_comments\_sub的div保存的是其他用户对原始评论所附加的评论。下面我们对这两部分的内容进行分析来获得有价值的信息。



原始评论的总体结构

原始评论也分为两个部分。一部分是评论者的用户信息，而另一部分则是评论的具体内容。

评论者的一些信息保存在类名为ui-avatar的div中，这一部分是以一个超链接的形式存在的，超链接的主体是一个图片，通过对这个超链接进行搜索，发现其正是该用户的在空间中的头像。而在超链接的属性中则包含着该用户的其他信息。href属性中的超链接指向的是该用户空间的主页。title属性中包含的则是该用户的昵称以及qq号。最后的data-uin属性的属性值也是用户的qq号。这些就是评论者的用户信息的具体内容。而其中包括的target属性则指定了从浏览器的新一页打开相关页面。从这些属性看下来我们可以通过抓取data-uin属性来获得该用户的唯一标识qq号来进行其他的分析操作。

二聚体的评论相关信息被保存在类名为comments\_content的div中。在这个div中，最值得关注的就是其中的span标签以及所包含的内容了，这个标签内部保存的内容正是用户的原始评论内容。这是一个可以抓取的重点。除了评论的内容外，作为一条评论，自然还包括着评论者的信息，这是以一个超链接的形式来呈现的，这点与上面的显示方式类似。但不同的地方是，这次显示的用户的昵称而非空间的头像，而此超链接所指向的位置也并非该用户的qq空间，而是具体的指向了该用户的说说主页，超链接的打开方式也从在新窗口打开转换到了在当前页打开。除此之外，在这个模块中还包含着另一个div，其中有一个span和一个超链接。Span标签则是保存着这条评论的发表时间，而超链接则是通过点击来调用JavaScript函数来完成相应的回复操作的。

至此，四部分的内容组合起来就构成了一条完整的用户评论。



完整的用户评论

对比上用户评论，我们可以在总的div中找到一一对应的部分。其中头像是独立的在一个div中，其他的三部分在另一个div中。因此在获取时我们可以根据需要来对内容进行获取。

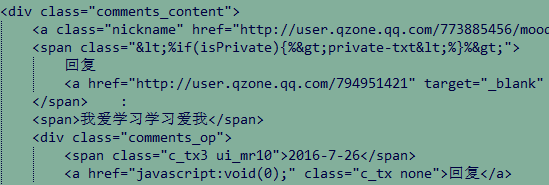
|  |  |
| --- | --- |
| 评论中 | 代码中 |
| 头像 | a标签中的img标签 |
| 昵称 | a标签中的文字 class="nickname" |
| 内容 | Span标签 |
| 时间和相应操作 | div class="comments\_op" |

从这里我们也可以看出，指定用户的空间的说说主页的URL，除了前面的在空间的主页后加上/311以外，还可以将311替换成mood.这样也可以达到相同的效果，两者都可以作为用户空间中说说主页的入口地址。



后续评论的结构

前面我们分析了原始评论的结构。现在分析一下后续评论的结构。与原始评论区别最大的一点就是后续评论可能不止一条，很有可能是由多人针对不同情况回复的多条内容所组成。根据前面的经验，可以合理推断出这部分的总体结构也是一个列表。其中每一条记录都是一条评论。通过查看代码，发现其结构正如预测，主题结构是一个无序的列表，其中每一条记录都是一条评论，这里的类名与上方原始评论的类名一致，也表明了其中包含的内容与结构应于原始评论中的内容类似。通过对两者进行比较，发现两者的第一部分的结构完全一致。都是显示着发表评论用户的头像以及指向相应用户的空间主页。第二部分是两者存在区别的地方，说是存在区别，也只是评论内容的部分存在一定的区别，即span标签的表示有所区别。在原始评论中，评论是默认针对说说主体进行的，所以直接将内容显示出来。但后续评论则不同，不仅要将内容表示出来，而且还需要指出具体回复的对象。所以，在后续评论中有关评论内容的div中，不仅要包含回复内容的div，还需要包含一个标签用来指定回复的对象。



后续评论中的评论内容的结构

通过审查代码，我们可以看出这里是通过增加一个span标签来实现关联被回复用户。我们可以看到，这个增加的span标签里面包含着一个a标签，这个a标签所采用的的格式与前面指定用户的方式类似，都是链接中包含着用户的qq空间主页、打开方式和用户昵称等内容。这样就达到了指明被回复对象的目的。

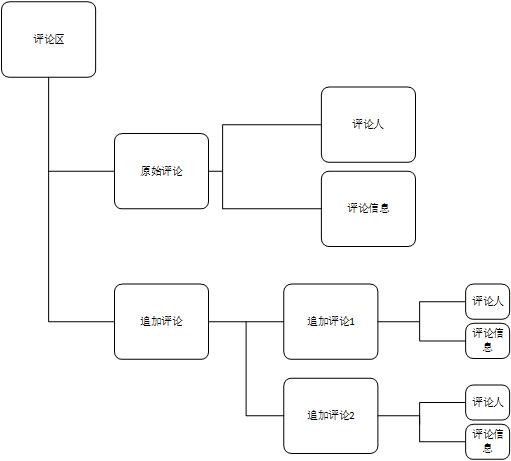
通过分析，一条后续评论的结构也就明确了。总体结构和原始评论类似。只是在内容处新增了待回复的部分，让整个评论的结构更加清楚。



完整的后续评论

从实际的效果截图中可以看出来，和原始评论相比。后续评论追加的部分就是回复 和 相应的名字。这部分正是通过在div中新增加的span标签实现的。而其他的部分和初始评论部分相同，如果要抓取信息可以依据前面的结构来对目标信息进行抓取。我们分析的内容是一条后续评论的结构，也就是无序列表中的一条记录。我们知道，后续评论可能不止一条。所以，每一条评论可以作一条记录存在。

这样，结合两部分的内容，我们就分析完成了评论区的结构分析。



综上所述，我们可以看出来评论部分的特点。评论的内容都是存储在span标签中，如果要抓取评论，可以选择span标签进行抓取。如果要抓取评论人的信息，则可以选择从a标签的data-uin中提取用户的qq号。

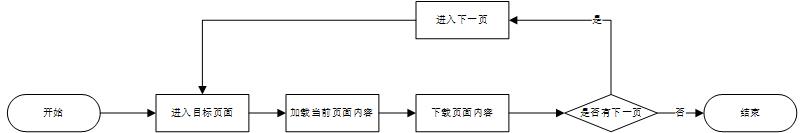
这样，我们就基本上完成了对qq空间上说说面板的信息的分析。下面要构建qq空间的基本抓取策略。

|  |  |
| --- | --- |
| 目标 | 百度百科太原理工大学相关词条页面-标题和简介 |
| 入口页 | user.qzone.qq.com/目标用户的qq号/mood（mood可以替换成311） |
| qq号 | data-uin="794951421" |
| 说说内容 | pre style="display:inline;" class="content" |
| 说说信息 | 时间：<a class="c\_tx c\_tx3 goDetail"> |
| 设备：<span class="custom-tail" title="MEIZU"> |
| 被转载说说 | <div class="quote bor2"> |
| 评论数 | class="c\_tx comment\_btn" |
| 转载数 | class="c\_tx forward\_btn" |
| 评论列表 | div class="comments\_list" |
| 原始评论 | div |
| 追加评论 | <div class="comments\_content"> |
| 页面编码 | UTF-8 |

### 三、 系统设计

经过上面对目标的分析，我们也基本明确了qq空间中说说主页上面的信息。从上面最后的表格中我们可以看出来对于每条说说来讲包含的信息都很多。我们选择了获取说说的内容、说说的时间以及说说发表所使用的设备这三种信息作为我们要抓取的目标。

在确定了基本的抓取目标后，就是抓取流程的确定了。首先是进入相应用户的界面，然后是拉取整个界面的内容。在确认整个页面都加载完毕后，就是将整个页面的内容下载下来。待下载完成后，将读取的内容写到文件中，最后点击下一页。如此往复循环直到将所有页面中的的说说内容下载下来，当不存在下一页按钮后即表明完成说说的爬取。



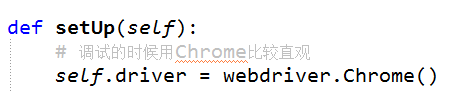
### 四、代码编写

在完成对目标的分析和对系统的分析后，接下来要进行的就是系统的编写了。

首先应该完成的是主函数的编写，但这次我们编写程序时使用的是测试用的框架，所以我们在程序的开头调用了unittest模块来进行单元测试的相关工作。在程序中，我通过unittest.main()来启动单元测试的测试模块。这也就是主函数中的全部内容。

因为我们采用的是单元测试的方法来进行程序的编写。所以在函数中分为两部分，一部分是测试部分，另一部分是待测部分。测试部分包含着负责初始化工作的setUp函数和负责退出清理的tearDown函数。而待测部分就是qq空间说说的爬取函数。

初始化函数和清理函数都很简单，因为这个函数没有涉及到很多复杂的操作。所以在setUp函数中有一句话，就是将利用导入的webdriver为函数本身生成一个浏览器程序，为了提高兼容性，这次我们选择的是生成一个谷歌浏览器对象。



因为在初始化函数中没有申请任何资源，所以在负责结束清理的函数tearDown函数中不需要释放任何资源。但是为了可以更好监控程序的整体运行状态，我们可以在这个函数中加入一条语句向控制台打印内容，来表示程序的结束。

接下来就是被测程序，也就是爬虫主要程序的编写了。根据流程图所示，首先是访问相应的说说主界面。因为在初始化程序中已经利用导入的webdriver生成的对应的driver对象。所以我们可以直接利用self来获取一个driver对象来进行程序需要的全部操作。首先是访问指定页面，这个操作是通过driver对象调用get方法来实现的。但在实际运行的过程中这一步出现了问题。不是因为程序逻辑的问题，而是因为腾讯为了加强用户信息的安全，在访问空间的说说信息时需要验证用户的身份，所以无论是访问空间的任何一个页面，如果是在未登录的情况下，页面都会自动地重定向到登录页面。所以，为了可以正常的获取说说信息，我们必须模拟登陆操作来达到自动抓取的目的。即我们需要编写一个自动登陆的部分。

这里需要先分析一下程序进行的流程。首先进入登陆页面后是这样的情况：



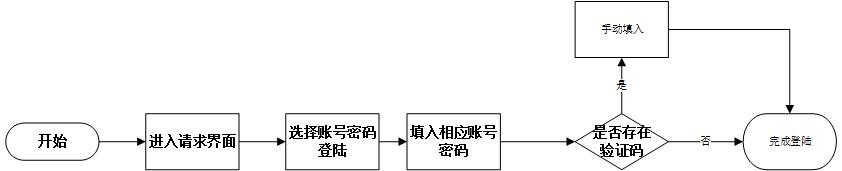
自动跳转界面

这是自动跳转的界面，这个界面显示的是要求用户使用快捷登陆的界面。显然，我们无法模拟这部分的操作，所以我们观察下方的选项，发现有一个账号密码登陆的选项，我们可以通过这一部分完成自动登陆的操作。



账号密码登陆界面

当点击账号密码登陆后，页面继续跳转到我们熟悉的页面，在这个页面中有三部分内容。分别是账号输入框、密码输入框和登陆按钮，在确定了登录界面的具体情况后候我们就可以针对性的设计自动登陆的流程了。



如若想实现自动登陆的操作，必须获得各个组件的名称，这样我们才可以通过webdriver中提供的方法来实现对各部分的访问。那么首先我们要做的就是使用工具来确定目标的名称。



登陆框架的名称

在前面的分析目标中，我们就了解到了qq空间是由多个frame和iframe组成的，所以，我们需要账号密码登陆超链接所在的框架名称，这样才能继续下面的选择操作。



账号密码登陆超链接的属性

然后，我们通过审查元素的方法获取到了账号密码登陆超链接的id属性，这样我们就可以通过id的唯一标识来访问这个超链接了。点击此超链接后，登陆框架自动的跳转到账号密码登陆的界面。

因为我们要自动登陆，也就是要完成自动填写账号密码的操作。这时我们就需要获得账号框、密码框和登录按钮三者的id属性。同理，我们也可以利用审查元素来获得相应的内容。



账号文本框的属性



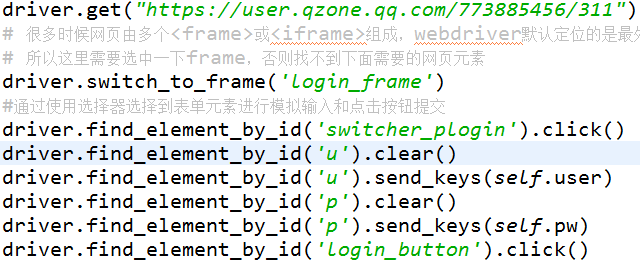
密码文本框的属性



登陆按钮的属性

在相应的属性值全部获取完成后，我们就可以完成自动登陆部分的代码编写了。在前面我们已经完成了这部分的流程图，具体来说就是首先访问目标用户的说说主页，这时系统会自动跳转到登陆的界面。然后我们先选中登陆的框体，点选其中的账号密码登陆超链接，这样这部分的框架就会动态的变换到账号密码登陆的界面。接下来我们利用前面获得的两个文本框的id值，自动填入相应的账号密码。最后在完成相应账号信息的填写后，点选登陆按钮。如若保存的账号密码无误，则系统会自动跳转到登陆之前的页面也就是用户的说说主页。

核心代码如下：



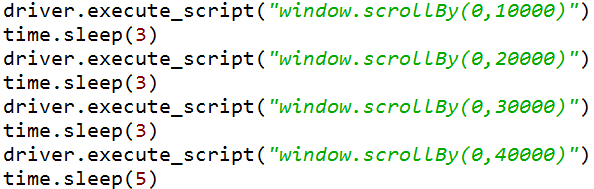
自动登陆部分的核心代码

但是qq空间作为一个比较成熟的社区系统，对于网络机器人还是有一定的防止措施的。最简单的是在自动登陆的时候可能会要求输入验证码，经过我的多次尝试。发现每当第一次登陆的时候必定需要输入验证码，如果之前已经进行过登陆。在短时间内重复登陆的话就不需要了。目前还没有解决自动识别验证码并登陆的操作，目前的解决方法是如果遇到自动登陆失败的情况，则需要用户手动进行登陆。

在可以成功进入目标用户的说说主页后，依据设计，接下来需要完成的就是循环抓取目标用户的说说内容了。循环的第一步就是加载每个页面的全部内容。

因为我们这次模拟的是一个真正的浏览器操作，所以加载内容的方法也和平常我们的操作方式相同，利用滚动条的下滑来完成加载当前页面的全部内容。drive中也存在着执行相应方法的操作。开始时我尝试了一次向下滚动一个极大的距离来尝试满足每页的最大加载数，但是失败了。结果只加载了一次后窗体就不在运行了。而后经过自己的手工测试，发现空间的动态加载有两部分的特点。一是每次当滚动条滚动到最下方时，页面都会有一个很短的但是存在的加载时间，这一点当网速较慢时很明显。第二点是每一页的说说不止需要一次就可以加载完成，经过我的多次测试证实了一共需三到四次的滚动才能最终加载完一整个的页面。

所以经过优化的加载页面操作应该是这样的：一共分4次加载，而且在每次的加载都应比上一次加载长一些的距离以保证完全的加载。最重要的是在每次加载的间隙中应该有一定的停止时间来供网页加载说说内容，否则如果是连续下滑的话，网页没有加载的时间，在网页正在加载时的下滑操作就全部无效化了。



滚动操作的代码

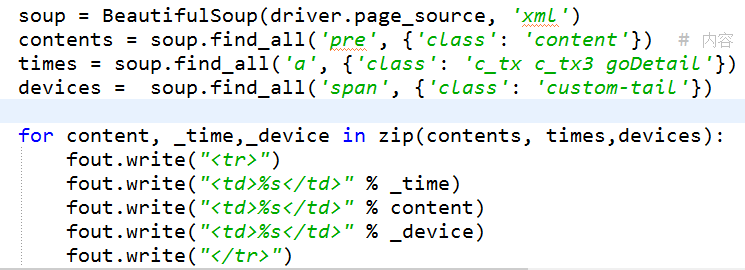
滚动操作的完成就标志一个页面的内容已全部加载完成了。下面就是将当前页面上的说说信息全部抓取下来的操作了。这部分是整个爬虫的核心。不过有了之前的分析，这部分反而显得是最简单的一部分，就是利用之前我们在前面的爬虫中使用过的find的一系列的方法，根据目标的标签类型和类名来获取响应对象的值。

如若需要获取说说的内容，则需要先将浏览器的焦点转换到说说所在的框架上。因此利用检查工具先获得框架的名称。



说说内容所在框架

在将框体的焦点转换到说说详情所在的框架后，接下来的操作就和前面的爬取基本相同了。首先是获取网页的源文件，将其转换成soup对象。然后就是利用find\_all方法获取页面上的全部说说内容、相应的发表时间以及所使用的设备。之后就是利用for循环遍历得到的结果，将全部的结果利用文件的输出写回本地的文件中，完成数据的持久化操作。



抓取部分的核心代码

在抓取部分也成功完成后，接下来的内容就是完成循环的最后一步，向下翻页的操作了。首先我们需要了解下一页的超链接的特点。



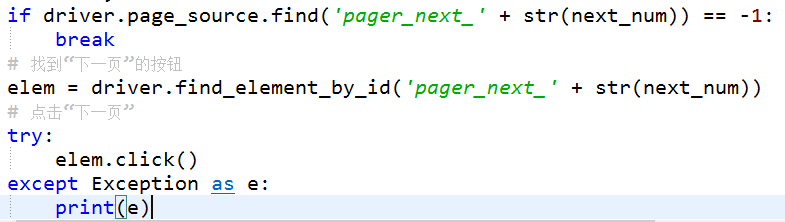
几个“下一页”超链接的属性

通过比较几个下一页超链接的属性，我们明确了下一页超链接id值的特点：以pager\_next\_作为开始，后面跟随一个数字。这个数字从0开始，以1为步长逐次增加。前面这些都是存在可点击的下一页超链接的页面。为了结束循环，我们还需要最后一页的情况。



最后一页的“下一页”

从图上我们可以看出，这个最后一页仍会显示下一页的文字，但是已经不存在相应的超链接了，而且从span所处的父span的类名也可以看出来，这部分没有任何功能。总结上面两部分的内容后，我们可以确定下来翻页的流程了，首先在爬虫中需要保存一个逐渐递增的变量来生成超链接所在的id，然后使用find方法寻找此超链接的存在性。如果存在对应id值的超链接，则利用driver选择此超链接并进行跳转，循环继续进行。如果在当前的页面中没有找到对应的超链接，说明此时爬虫已经到达目标用户说说内容的最后一页，应该跳出循环，程序结束。



跳转部分的核心代码

至此，一个qq空间说说的爬取程序就已全部构建完成。从进入页面、加载信息、获取信息知道下载信息，一个清楚的爬虫已经出现在我们的眼前，接下来就是爬虫的实例演示了。

### 五、运行爬虫

经过上面的几步后，一个爬虫程序就可以正式运行了，在运行前。还需要注意几个问题。首先是配置初始变量，一共有两个部分的变量需要配置。第一部分是需要一个qq账号信息。第二部分是需要一个目标qq空间说说主页地址。而且，这两部分的初始化信息需要保持一致，即提供的qq号有权限去访问相应用户的空间。否则会发生错误。因为爬虫也只是将原本手动的行为自动化，并不能超越原来的用户权限。

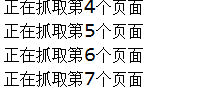


测试用qq账号信息



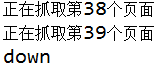
目标qq空间的说说主页地址

填写完这两部分信息后，接下来就是启动爬虫抓取信息了。



控制台中打印的信息

为了更好的检测爬虫的运行，我们在代码中加入了相应的输出语句来监控爬虫的运行状态。网络爬虫是运行在网络上的自动化程序，因此对网络的有一定的要求，在测试过程中也出现过无法定位的异常导致爬虫终止的情况。重复运行后程序正常运行，据此我推断程序可能是因为网络的原因没有将后续的内容加载下来，导致爬虫的下载部分未找到目标，致使整个程序通知运行。成功运行后，程序会再最后打印出down标识标志程序结束。



程序运行结束



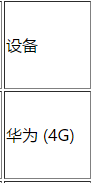
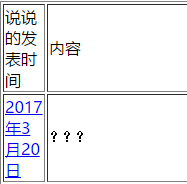
用户的说说页数

通过与用户的页面比较，证实了爬虫已经遍历完毕用户的全部说说列表。



根据说说内容生成的html文件

下面看本地的情况，爬虫根据从用户处获取的数据在本地成功生成了记录了相应内容的文件。

爬虫记录下的内容（左侧）、实际的内容（右侧）

如图所示，爬虫程序成功将说说的内容，发表时间以及所使用的的设备记录了下来。与实际中的说说相比，说明爬虫已将说说中的信息成功抓取下来。

# 第四章 总结与展望

2017年3月，我开始进行我的毕业设计1任务，时至今日，论文基本完成。从最初的不知所措，然后是大量的查阅的资料，再到对思路逐渐的清晰，整个写作过程不知道怎么用语言完整的来表达出来。历经了几个月的奋战，紧张而又充实的毕业设计终于落下了帷幕。回想这段日子的经历和感受，我感慨万千，在这次毕业设计的过程中，我拥有了无数难忘的回忆和收获。   
 3月初，在与基地指导老师和学校的指导老师的交流讨论中渐渐有了选题的思路，自己也上网查阅了一些有关资料，再结合学校所给的一些选题。最后我将自己的毕业设计的题目确定为网络爬虫的设计实现。选择这个题目也是有多方面的考虑。首先网络一直是我比较感兴趣的地方，加上通过和老师、同学的交流，我也了解到了爬虫也是有一定的发展前景的，特别是在当前数据爆炸的时代。所以我选择了网络爬虫作为自己毕业设计的题目，即符合我的兴趣，还可以开拓自己的视野，增长自己的学识。当选题结束定下来最后的题目后，我便立刻着手开始进行资料的收集工作，当时面对纷繁复杂的的参考资料真是有些茫然，不知如何下手。我将这一困难告诉了指导老师，指导导师通过他自己的经验，为困惑中的我推荐了一些入门的书籍和材料。在阅读过老师精心的选择的材料后，使我对自己的工作方向和方法有了一个基本的认识。   
　　为了可以全方位的掌握有关爬虫的相关信息。为了克服离学校远的缺点，我利用网络网络远程接入校园网，利用学校提供的知网、万方等数据库广泛的查阅论文资料、于此同时我还在网上查找各类相关内容。除此之外，我还拜托我的同学在外校的数据库中寻找相关信息，尽量使我的资料完整、精确、数量多，这有利于论文的撰写。然后我将收集到的资料仔细整理分类，及时拿给导师进行沟通。   
　　4月初，资料已经查找完毕了，我开始着手程序和论文的框架搭建。在构建过程中遇到了任何的困难我就会及时的和导师联系，并和同学互相交流，请教有关方面的专业课老师。在大家的帮助下，困难一个一个解决掉，毕业设计的框架也基本完成。

在基本确定了整个毕业设计的结构后，我首先将前面收集到的资料细化，选取符合自己的部分仔细研读，来完成论文前面的部分。

　　4月中旬，论文前半部分有关爬虫的理论部分已经完成。接下来要进行的部分就是爬虫的代码编写工作。为了完成满意的程序，选择了python作为了我所使用的开发语言。Python中包含着很多官方的和第三方的库，专是为爬虫开发所设计的，使用python不仅可以简化开发难度，更可以让我把注意力集中在分析设计上。于是我认真学习了python语言的相关知识。作为学习过多种编程语言的我来说，python的语法很简单，对于其中的难点也就是一些python独有的特性，多联系几次后也可以基本掌握。学习爬虫另一项重要的内容就是有关正则表达式的学习。如何高效、正确的的使用正则表达式对于一个爬虫的效率是很关键的。很有可能就因为正则表达式书写的区别导致整个程序在效率方面有了天壤之别。在之前的课程中我们也学习过与正则表达式类似的内容如EL表达式，在有些课程中也接触过正则表达式的相关内容如离散数学。在整个的过程中，我学到了新知识，增长了见识。在今后的日子里，我仍然要不断地充实自己，扩充自己的知识面以契合当前飞速发展的社会。

这次毕设给我的最深的体会就是在大多情况下编码都是最简单的任务。重点都在于对目标的分析上。看上去短短百行的代码好像再很短时间就可以编写完成。但在前面对网站结构的分析上就需要对网站上万行的代码进行仔细的分析与测试。只有分析好了目标，在编写代码时才会轻松。脚踏实地，认真严谨，实事求是的学习态度，不怕困难、坚持不懈、吃苦耐劳的精神是我在这次设计中最大的收益。我想这是一次意志的磨练，是对我实际能力的一次提升，也会对我未来的学习和工作有很大的帮助。

参考文献（五号黑体）

1　陈日耀.金属切削原理.北京：机械工业出版社，1985.33~36（五号宋体）

2　Trent E M .Metal Cutting. 2nd Ed. London: BUTTERWORTHS, 1984.120~125

致　　谢（小四号黑体）

感谢方昀老师、李楠高级软件工程师等在毕业设计工作中给予的帮助。

感谢我的学友和朋友对我的关心和帮助（五号宋体）

外文资料

ABSTRACT  
Crawling the web is deceptively simple: the basic algorithm is (a)Fetch a page (b) Parse it to extract all linked URLs (c) For all the URLs not seen before, repeat (a)–(c). However, the size of the web (estimated at over 4 billion pages) and its rate of change (estimated at 7% per week) move this plan from a trivial programming exercise to a serious algorithmic and system design challenge. Indeed, these two factors alone imply that for a reasonably fresh and complete crawl of the web, step (a) must be executed about a thousand times per second, and thus the membership test (c) must be done well over ten thousand times per second against a set too large to store in main memory. This requires a distributed architecture, which further complicates the membership test.  
 A crucial way to speed up the test is to cache, that is, to store in main memory a (dynamic) subset of the “seen” URLs. The main goal of this paper is to carefully investigate several URL caching techniques for web crawling. We consider both practical algorithms: random replacement, static cache, LRU, and CLOCK, and theoretical limits: clairvoyant caching and infinite cache. We performed about 1,800 simulations using these algorithms with various cache sizes, using actual log data extracted from a massive 33 day web crawl that issued over one billion HTTP requests. Our main conclusion is that caching is very effective – in our setup, a cache of roughly 50,000 entries can achieve a hit rate of almost 80%. Interestingly, this cache size falls at a critical point: a substantially smaller cache is much less effective while a substantially larger cache brings little additional benefit. We conjecture that such critical points are inherent to our problem and venture an explanation for this phenomenon.  
1. INTRODUCTION  
A recent Pew Foundation study [31] states that “Search engines have become an indispensable utility for Internet users” and estimates that as of mid-2002, slightly over 50% of all Americans have used web search to find information. Hence, the technology that powers web search is of enormous practical interest. In this paper, we concentrate on one aspect of the search technology, namely the process of collecting web pages that eventually constitute the search engine corpus.  
 Search engines collect pages in many ways, among them direct URL submission, paid inclusion, and URL extraction from nonweb sources, but the bulk of the corpus is obtained by recursively exploring the web, a process known as crawling or SPIDERing. The basic algorithm is  
 (a) Fetch a page  
 (b) Parse it to extract all linked URLs  
 (c) For all the URLs not seen before, repeat (a)–(c)  
 Crawling typically starts from a set of seed URLs, made up of URLs obtained by other means as described above and/or made up of URLs collected during previous crawls. Sometimes crawls are started from a single well connected page, or a directory such as yahoo.com, but in this case a relatively large portion of the web (estimated at over 20%) is never reached. See [9] for a discussion of the graph structure of the web that leads to this phenomenon.  
 If we view web pages as nodes in a graph, and hyperlinks as directed edges among these nodes, then crawling becomes a process known in mathematical circles as graph traversal. Various strategies for graph traversal differ in their choice of which node among the nodes not yet explored to explore next. Two standard strategies for graph traversal are Depth First Search (DFS) and Breadth First Search (BFS) – they are easy to implement and taught in many introductory algorithms classes. (See for instance [34]).  
 However, crawling the web is not a trivial programming exercise but a serious algorithmic and system design challenge because of the following two factors.  
 1. The web is very large. Currently, Google [20] claims to have indexed over 3 billion pages. Various studies [3, 27, 28] have indicated that, historically, the web has doubled every 9-12 months.  
 2. Web pages are changing rapidly. If “change” means “any change”, then about 40% of all web pages change weekly [12]. Even if we consider only pages that change by a third or more, about 7% of all web pages change weekly [17].  
 These two factors imply that to obtain a reasonably fresh and 679 complete snapshot of the web, a search engine must crawl at least 100 million pages per day. Therefore, step (a) must be executed about 1,000 times per second, and the membership test in step (c) must be done well over ten thousand times per second, against a set of URLs that is too large to store in main memory. In addition, crawlers typically use a distributed architecture to crawl more pages in parallel, which further complicates the membership test: it is possible that the membership question can only be answered by a peer node, not locally.  
 A crucial way to speed up the membership test is to cache a (dynamic) subset of the “seen” URLs in main memory. The main goal of this paper is to investigate in depth several URL caching techniques for web crawling. We examined four practical techniques: random replacement, static cache, LRU, and CLOCK, and compared them against two theoretical limits: clairvoyant caching and infinite cache when run against a trace of a web crawl that issued over one billion HTTP requests. We found that simple caching techniques are extremely effective even at relatively small cache sizes such as 50,000 entries and show how these caches can be implemented very efficiently.  
 The paper is organized as follows: Section 2 discusses the various crawling solutions proposed in the literature and how caching fits in their model. Section 3 presents an introduction to caching techniques and describes several theoretical and practical algorithms for caching. We implemented these algorithms under the experimental setup described in Section 4. The results of our simulations are depicted and discussed in Section 5, and our recommendations for practical algorithms and data structures for URL caching are presented in Section 6. Section 7 contains our conclusions and directions for further research.  
2. CRAWLING  
Web crawlers are almost as old as the web itself, and numerous crawling systems have been described in the literature. In this section, we present a brief survey of these crawlers (in historical order) and then discuss why most of these crawlers could benefit from URL caching.  
 The crawler used by the Internet Archive [10] employs multiple crawling processes, each of which performs an exhaustive crawl of 64 hosts at a time. The crawling processes save non-local URLs to disk; at the end of a crawl, a batch job adds these URLs to the per-host seed sets of the next crawl.  
 The original Google crawler, described in [7], implements the different crawler components as different processes. A single URL server process maintains the set of URLs to download; crawling processes fetch pages; indexing processes extract words and links; and URL resolver processes convert relative into absolute URLs, which are then fed to the URL Server. The various processes communicate via the file system.  
 For the experiments described in this paper, we used the Mercator web crawler [22, 29]. Mercator uses a set of independent, communicating web crawler processes. Each crawler process is responsible for a subset of all web servers; the assignment of URLs to crawler processes is based on a hash of the URL’s host component. A crawler that discovers an URL for which it is not responsible sends this URL via TCP to the crawler that is responsible for it, batching URLs together to minimize TCP overhead. We describe Mercator in more detail in Section 4.  
 Cho and Garcia-Molina’s crawler [13] is similar to Mercator. The system is composed of multiple independent, communicating web crawler processes (called “C-procs”). Cho and Garcia-Molina consider different schemes for partitioning the URL space, including URL-based (assigning an URL to a C-proc based on a hash of the entire URL), site-based (assigning an URL to a C-proc based on a hash of the URL’s host part), and hierarchical (assigning an URL to a C-proc based on some property of the URL, such as its top-level domain).  
 The WebFountain crawler [16] is also composed of a set of independent, communicating crawling processes (the “ants”). An ant that discovers an URL for which it is not responsible, sends this URL to a dedicated process (the “controller”), which forwards the URL to the appropriate ant.  
 UbiCrawler (formerly known as Trovatore) [4, 5] is again composed of multiple independent, communicating web crawler processes. It also employs a controller process which oversees the crawling processes, detects process failures, and initiates fail-over to other crawling processes.  
 Shkapenyuk and Suel’s crawler [35] is similar to Google’s; the different crawler components are implemented as different processes. A “crawling application” maintains the set of URLs to be downloaded, and schedules the order in which to download them. It sends download requests to a “crawl manager”, which forwards them to a pool of “downloader” processes. The downloader processes fetch the pages and save them to an NFS-mounted file system. The crawling application reads those saved pages, extracts any links contained within them, and adds them to the set of URLs to be downloaded.  
 Any web crawler must maintain a collection of URLs that are to be downloaded. Moreover, since it would be unacceptable to download the same URL over and over, it must have a way to avoid adding URLs to the collection more than once. Typically, avoidance is achieved by maintaining a set of discovered URLs, covering the URLs in the frontier as well as those that have already been downloaded. If this set is too large to fit in memory (which it often is, given that there are billions of valid URLs), it is stored on disk and caching popular URLs in memory is a win: Caching allows the crawler to discard a large fraction of the URLs without having to consult the disk-based set.  
 Many of the distributed web crawlers described above, namely Mercator [29], WebFountain [16], UbiCrawler[4], and Cho and Molina’s crawler [13], are comprised of cooperating crawling processes, each of which downloads web pages, extracts their links, and sends these links to the peer crawling process responsible for it. However, there is no need to send a URL to a peer crawling process more than once. Maintaining a cache of URLs and consulting that cache before sending a URL to a peer crawler goes a long way toward reducing transmissions to peer crawlers, as we show in the remainder of this paper.  
3. CACHING  
In most computer systems, memory is hierarchical, that is, there exist two or more levels of memory, representing different tradeoffs between size and speed. For instance, in a typical workstation there is a very small but very fast on-chip memory, a larger but slower RAM memory, and a very large and much slower disk memory. In a network environment, the hierarchy continues with network accessible storage and so on. Caching is the idea of storing frequently used items from a slower memory in a faster memory. In the right circumstances, caching greatly improves the performance of the overall system and hence it is a fundamental technique in the design of operating systems, discussed at length in any standard textbook [21, 37]. In the web context, caching is often mentioned  
in the context of a web proxy caching web pages [26, Chapter 11]. In our web crawler context, since the number of visited URLs becomes too large to store in main memory, we store the collection of visited URLs on disk, and cache a small portion in main memory.  
 Caching terminology is as follows: the cache is memory used to store equal sized atomic items. A cache has size k if it can store at most k items.1 At each unit of time, the cache receives a request for an item. If the requested item is in the cache, the situation is called a hit and no further action is needed. Otherwise, the situation is called a miss or a fault. If the cache has fewer than k items, the missed item is added to the cache. Otherwise, the algorithm must choose either to evict an item from the cache to make room for the missed item, or not to add the missed item. The caching policy or caching algorithm decides which item to evict. The goal of the caching algorithm is to minimize the number of misses.  
 Clearly, the larger the cache, the easier it is to avoid misses. Therefore, the performance of a caching algorithm is characterized by the miss ratio for a given size cache. In general, caching is successful for two reasons:  
 \_ Non-uniformity of requests. Some requests are much more popular than others. In our context, for instance, a link to yahoo.com is a much more common occurrence than a link to the authors’ home pages.  
 \_ Temporal correlation or locality of reference. Current requests are more likely to duplicate requests made in the recent past than requests made long ago. The latter terminology comes from the computer memory model – data needed now is likely to be close in the address space to data recently needed. In our context, temporal correlation occurs first because links tend to be repeated on the same page – we found that on average about 30% are duplicates, cf. Section 4.2, and second, because pages on a given host tend to be explored sequentially and they tend to share many links. For example, many pages on a Computer Science department server are likely to share links to other Computer Science departments in the world, notorious papers, etc.  
 Because of these two factors, a cache that contains popular requests and recent requests is likely to perform better than an arbitrary cache. Caching algorithms try to capture this intuition in various ways.  
 We now describe some standard caching algorithms, whose performance we evaluate in Section 5.  
3.1 Infinite cache (INFINITE)  
 This is a theoretical algorithm that assumes that the size of the cache is larger than the number of distinct requests.  
3.2 Clairvoyant caching (MIN)  
 More than 35 years ago, L´aszl´o Belady [2] showed that if the entire sequence of requests is known in advance (in other words, the algorithm is clairvoyant), then the best strategy is to evict the item whose next request is farthest away in time. This theoretical algorithm is denoted MIN because it achieves the minimum number of misses on any sequence and thus it provides a tight bound on performance.  
3.3 Least recently used (LRU)  
 The LRU algorithm evicts the item in the cache that has not been requested for the longest time. The intuition for LRU is that an item that has not been needed for a long time in the past will likely not be needed for a long time in the future, and therefore the number of misses will be minimized in the spirit of Belady’s algorithm.  
 Despite the admonition that “past performance is no guarantee of future results”, sadly verified by the current state of the stock markets, in practice, LRU is generally very effective. However, it requires maintaining a priority queue of requests. This queue has a processing time cost and a memory cost. The latter is usually ignored in caching situations where the items are large.  
3.4 CLOCK  
 CLOCK is a popular approximation of LRU, invented in the late sixties [15]. An array of mark bits M0;M1; : : : ;Mk corresponds to the items currently in the cache of size k. The array is viewed as a circle, that is, the first location follows the last. A clock handle points to one item in the cache. When a request X arrives, if the item X is in the cache, then its mark bit is turned on. Otherwise, the handle moves sequentially through the array, turning the mark bits off, until an unmarked location is found. The cache item corresponding to the unmarked location is evicted and replaced by X.  
3.5 Random replacement (RANDOM)  
 Random replacement (RANDOM) completely ignores the past. If the item requested is not in the cache, then a random item from the cache is evicted and replaced.  
 In most practical situations, random replacement performs worse than CLOCK but not much worse. Our results exhibit a similar pattern, as we show in Section 5. RANDOM can be implemented without any extra space cost; see Section 6.  
3.6 Static caching (STATIC)  
 If we assume that each item has a certain fixed probability of being requested, independently of the previous history of requests, then at any point in time the probability of a hit in a cache of size k is maximized if the cache contains the k items that have the highest probability of being requested.  
 There are two issues with this approach: the first is that in general these probabilities are not known in advance; the second is that the independence of requests, although mathematically appealing, is antithetical to the locality of reference present in most practical situations.  
 In our case, the first issue can be finessed: we might assume that the most popular k URLs discovered in a previous crawl are pretty much the k most popular URLs in the current crawl. (There are also efficient techniques for discovering the most popular items in a stream of data [18, 1, 11]. Therefore, an on-line approach might work as well.) Of course, for simulation purposes we can do a first pass over our input to determine the k most popular URLs, and then preload the cache with these URLs, which is what we did in our experiments.  
 The second issue above is the very reason we decided to test STATIC: if STATIC performs well, then the conclusion is that there is little locality of reference. If STATIC performs relatively poorly, then we can conclude that our data manifests substantial locality of reference, that is, successive requests are highly correlated.  
4. EXPERIMENTAL SETUP  
 We now describe the experiment we conducted to generate the crawl trace fed into our tests of the various algorithms. We conducted a large web crawl using an instrumented version of the Mercator web crawler [29]. We first describe the Mercator crawler architecture, and then report on our crawl.  
4.1 Mercator crawler architecture  
 A Mercator crawling system consists of a number of crawling processes, usually  
running on separate machines. Each crawling process is responsible for a subset of all web servers, and consists of a number of worker threads (typically 500) responsible for downloading and processing pages from these servers.  
 Each worker thread repeatedly performs the following operations: it obtains a URL from the URL Frontier, which is a diskbased data structure maintaining the set of URLs to be downloaded; downloads the corresponding page using HTTP into a buffer (called a RewindInputStream or RIS for short); and, if the page is an HTML page, extracts all links from the page. The stream of extracted links is converted into absolute URLs and run through the URL Filter, which discards some URLs based on syntactic properties. For example, it discards all URLs belonging to web servers that contacted us and asked not be crawled.  
 The URL stream then flows into the Host Splitter, which assigns URLs to crawling processes using a hash of the URL’s host name. Since most links are relative, most of the URLs (81.5% in our experiment) will be assigned to the local crawling process; the others are sent in batches via TCP to the appropriate peer crawling processes. Both the stream of local URLs and the stream of URLs received from peer crawlers flow into the Duplicate URL Eliminator (DUE). The DUE discards URLs that have been discovered previously. The new URLs are forwarded to the URL Frontier for future download. In order to eliminate duplicate URLs, the DUE must maintain the set of all URLs discovered so far. Given that today’s web contains several billion valid URLs, the memory requirements to maintain such a set are significant. Mercator can be configured to maintain this set as a distributed in-memory hash table (where each crawling process maintains the subset of URLs assigned to it); however, this DUE implementation (which reduces URLs to 8-byte checksums, and uses the first 3 bytes of the checksum to index into the hash table) requires about 5.2 bytes per URL, meaning that it takes over 5 GB of RAM per crawling machine to maintain a set of 1 billion URLs per machine. These memory requirements are too steep in many settings, and in fact, they exceeded the hardware available to us for this experiment. Therefore, we used an alternative DUE implementation that buffers incoming URLs in memory, but keeps the bulk of URLs (or rather, their 8-byte checksums) in sorted order on disk. Whenever the in-memory buffer fills up, it is merged into the disk file (which is a very expensive operation due to disk latency) and newly discovered URLs are passed on to the Frontier.  
 Both the disk-based DUE and the Host Splitter benefit from URL caching. Adding a cache to the disk-based DUE makes it possible to discard incoming URLs that hit in the cache (and thus are duplicates) instead of adding them to the in-memory buffer. As a result, the in-memory buffer fills more slowly and is merged less frequently into the disk file, thereby reducing the penalty imposed by disk latency. Adding a cache to the Host Splitter makes it possible to discard incoming duplicate URLs instead of sending them to the peer node, thereby reducing the amount of network traffic. This reduction is particularly important in a scenario where the individual crawling machines are not connected via a high-speed LAN (as they were in our experiment), but are instead globally distributed. In such a setting, each crawler would be responsible for web servers “close to it”.  
 Mercator performs an approximation of a breadth-first search traversal of the web graph. Each of the (typically 500) threads in each process operates in parallel, which introduces a certain amount of non-determinism to the traversal. More importantly, the scheduling of downloads is moderated by Mercator’s politeness policy, which limits the load placed by the crawler on any particular web server. Mercator’s politeness policy guarantees that no server ever receives multiple requests from Mercator in parallel; in addition, it guarantees that the next request to a server will only be issued after a multiple (typically 10\_) of the time it took to answer the previous request has passed. Such a politeness policy is essential to any large-scale web crawler; otherwise the crawler’s operator becomes inundated with complaints.  
4.2 Our web crawl  
 Our crawling hardware consisted of four Compaq XP1000 workstations, each one equipped with a 667 MHz Alpha processor, 1.5 GB of RAM, 144 GB of disk2, and a 100 Mbit/sec Ethernet connection. The machines were located at the Palo Alto Internet Exchange, quite close to the Internet’s backbone.  
 The crawl ran from July 12 until September 3, 2002, although it was actively crawling only for 33 days: the downtimes were due to various hardware and network failures. During the crawl, the four machines performed 1.04 billion download attempts, 784 million of which resulted in successful downloads. 429 million of the successfully downloaded documents were HTML pages. These pages contained about 26.83 billion links, equivalent to an average of 62.55 links per page; however, the median number of links per page was only 23, suggesting that the average is inflated by some pages with a very high number of links. Earlier studies reported only an average of 8 links [9] or 17 links per page [33]. We offer three explanations as to why we found more links per page. First, we configured Mercator to not limit itself to URLs found in anchor tags, but rather to extract URLs from all tags that may contain them (e.g. image tags). This configuration increases both the mean and the median number of links per page. Second, we configured it to download pages up to 16 MB in size (a setting that is significantly higher than usual), making it possible to encounter pages with tens of thousands of links. Third, most studies report the number of unique links per page. The numbers above include duplicate copies of a link on a page. If we only consider unique links3 per page, then the average number of links is 42.74 and the median is 17.  
 The links extracted from these HTML pages, plus about 38 million HTTP redirections that were encountered during the crawl, flowed into the Host Splitter. In order to test the effectiveness of various caching algorithms, we instrumented Mercator’s Host Splitter component to log all incoming URLs to disk. The Host Splitters on the four crawlers received and logged a total of 26.86 billion URLs.  
 After completion of the crawl, we condensed the Host Splitter logs. We hashed each URL to a 64-bit fingerprint [32, 8]. Fingerprinting is a probabilistic technique; there is a small chance that two URLs have the same fingerprint. We made sure there were no such unintentional collisions by sorting the original URL logs and counting the number of unique URLs. We then compared this number to the number of unique fingerprints, which we determined using an in-memory hash table on a very-large-memory machine. This data reduction step left us with four condensed host splitter logs (one per crawling machine), ranging from 51 GB to 57 GB in size and containing between 6.4 and 7.1 billion URLs.  
 In order to explore the effectiveness of caching with respect to inter-process communication in a distributed crawler, we also extracted a sub-trace of the Host Splitter logs that contained only those URLs that were sent to peer crawlers. These logs contained 4.92 billion URLs, or about 19.5% of all URLs. We condensed the sub-trace logs in the same fashion. We then used the condensed logs for our simulations.  
5. SIMULATION RESULTS  
 We studied the effects of caching with respect to two streams of URLs:  
 1. A trace of all URLs extracted from the pages assigned to a particular machine. We refer to this as the full trace.  
 2. A trace of all URLs extracted from the pages assigned to a particular machine that were sent to one of the other machines for processing. We refer to this trace as the cross subtrace, since it is a subset of the full trace.  
 The reason for exploring both these choices is that, depending on other architectural decisions, it might make sense to cache only the URLs to be sent to other machines or to use a separate cache just for this purpose.  
 We fed each trace into implementations of each of the caching algorithms described above, configured with a wide range of cache sizes. We performed about 1,800 such experiments. We first describe the algorithm implementations, and then present our simulation results.  
5.1 Algorithm implementations  
 The implementation of each algorithm is straightforward. We use a hash table to find each item in the cache. We also keep a separate data structure of the cache items, so that we can choose one for eviction. For RANDOM, this data structure is simply a list. For CLOCK, it is a list and a clock handle, and the items also contain “mark” bits. For LRU, it is a heap, organized by last access time. STATIC needs no extra data structure, since it never evicts items. MIN is more complicated since for each item in the cache, MIN needs to know when the next request for that item will be. We therefore describe MIN in more detail. Let A be the trace or sequence of requests, that is, At is the item requested at time t. We create a second sequence Nt containing the time when At next appears in A. If there is no further request for At after time t, we set Nt = 1. Formally,  
 To generate the sequence Nt, we read the trace A backwards, that is, from tmax down to 0, and use a hash table with key At and value t. For each item At, we probe the hash table. If it is not found, we set Nt = 1and store (At; t) in the table. If it is found, we retrieve (At; t0), set Nt = t0, and replace (At; t0) by (At; t) in the hash table. Given Nt, implementing MIN is easy: we read At and Nt in parallel, and hence for each item requested, we know when it will be requested next. We tag each item in the cache with the time when it will be requested next, and if necessary, evict the item with the highest value for its next request, using a heap to identify itquickly.  
5.2 Results  
 We present the results for only one crawling host. The results for the other three hosts are quasi-identical. Figure 2 shows the miss rate over the entire trace (that is, the percentage of misses out of all requests to the cache) as a function of the size of the cache. We look at cache sizes from k = 20 to k = 225. In Figure 3 we present the same data relative to the miss-rate of MIN, the optimum off-line algorithm. The same simulations for the cross-trace are depicted in Figures 4 and 5.  
 For both traces, LRU and CLOCK perform almost identically and only slightly worse than the ideal MIN, except in the critical region discussed below. RANDOM is only slightly inferior to CLOCK and LRU, while STATIC is generally much worse. Therefore, we conclude that there is considerable locality of reference in the trace, as explained in Section 3.6. For very large caches, STATIC appears to do better than MIN. However, this is just an artifact of our accounting scheme: we only charge for misses and STATIC is not charged for the initial loading of the cache. If STATIC were instead charged k misses for the initial loading of its cache, then its miss rate would be of course worse than MIN’s.  
6. CONCLUSIONS AND FUTURE DIRECTIONS  
 After running about 1,800 simulations over a trace containing 26.86 billion URLs, our main conclusion is that URL caching is very effective – in our setup, a cache of roughly 50,000 entries can achieve a hit rate of almost 80%. Interestingly, this size is a critical point, that is, a substantially smaller cache is ineffectual while a substantially larger cache brings little additional benefit. For practical purposes our investigation is complete: In view of our discussion in Section 5.2, we recommend a cache size of between 100 to 500 entries per crawling thread. All caching strategies perform roughly the same; we recommend using either CLOCK or RANDOM, implemented using a scatter table with circular chains. Thus, for 500 crawling threads, this cache will be about 2MB – completely insignificant compared to other data structures needed in a crawler. If the intent is only to reduce cross machine traffic in a distributed crawler, then a slightly smaller cache could be used. In either case, the goal should be to have a miss rate lower than 20%.  
 However, there are some open questions, worthy of further research. The first open problem is to what extent the crawl order strategy (graph traversal method) affects the caching performance. Various strategies have been proposed [14], but there are indications [30] that after a short period from the beginning of the crawl the general strategy does not matter much. Hence, we believe that caching performance will be very similar for any alternative crawling strategy. We can try to implement other strategies ourselves, but ideally we would use independent crawls. Unfortunately, crawling on web scale is not a simple endeavor, and it is unlikely that we can obtain crawl logs from commercial search engines.  
 In view of the observed fact that the size of the cache needed to achieve top performance depends on the number of threads, the second question is whether having a per-thread cache makes sense. In general, but not always, a global cache performs better than a collection of separate caches, because common items need to be stored only once. However, this assertion needs to be verified in the URL caching context.  
 The third open question concerns the explanation we propose in Section 5 regarding the scope of the links encountered on a given host. If our model is correct then it has certain implications regarding the appropriate model for the web graph, a topic of considerable interest among a wide variety of scientists: mathematicians, physicists, and computer scientists. We hope that our paper will stimulate research to estimate the cache performance under various models. Models where caching performs well due to correlation of links on a given host are probably closer to reality. We are making our URL traces available for this research by donating them to the Internet Archive.

中文译文

要在网络上爬行非常简单：基本的算法是：（a）取得一个网页（b）解析它提取所有的链接URLs（c）对于所有没有见过的URLs重复执行 （a）-（c）。但是，网络的大小（估计有超过40亿的网页）和他们变化的频率（估计每周有7%的变化）使这个计划由一个微不足道的设计习题变成一个非常 严峻的算法和系统设计挑战。实际上，光是这两个要素就意味着如果要进行及时地，完全地爬行网络，步骤（a）必须每秒钟执行大约1000次，因此，成员检测 （c）必须每秒钟执行超过10000次，并有非常大的数据储存到主内存中。这个要求有一个分布式构造，使得成员检测更加复杂。  
 一个非常重要的方法加速这个检测就是用cache（高速缓存），这个是把见过的URLs存入主内存中的一个（动态）子集中。这个论文最主要的成果就是仔细 的研究了几种关于网络爬虫的URL缓存技术。我们考虑所有实际的算法：随机置换，静态cache，LRU，和CLOCK，和理论极限：透视cache和极 大的cache。我们执行了大约1800次模拟，用不同的cache大小执行这些算法，用真实的log日志数据，获取自一个非常大的33天的网络爬行，大 约执行了超过10亿次的http请求。  
 我们的主要的结论是 cache是非常高效的-在我们的机制里，一个有大约50000个入口的cache可以完成80%的速率。有趣的是，这cache的大小下降到一个临界 点：一个足够的小一点的cache更有效当一个足够的大一点的cache只能带来很小的额外好处。我们推测这个临界点是固有的并且冒昧的解释一下这个现 象。  
1.介绍  
皮尤基金会最新的研究指出：“搜索引擎已经成为互联网用户不可或缺的工具”，估计在2002年中期，初略有超过1半的美国人用网络搜索获取信息。因此，一 个强大的搜索引擎技术有巨大的实际利益，在这个论文中，我们集中于一方面的搜索技术，也就是搜集网页的过程，最终组成一个搜索引擎的文集。  
 搜索引擎搜集网页通过很多途径，他们中，直接提交URL，回馈内含物，然后从非web源文件中提取URL，但是大量的文集包含一个进程叫 crawling 或者 SPIDERing，他们递归的探索互联网。基本的算法是：  
Fetch a page  
Parse it to extract all linked URLs  
For all the URLs not seen before，repeat（a）-(c)  
网络怕从一般开始于一些 种子URLs。有些时候网络爬虫开始于一个正确连接的页面，或者一个目录就像：yahoo.com，但是因为这个原因相关的巨大的部分网络资源无法被访问到。（估计有超过20%）  
如果把网页看作图中的节点，把超链接看作定向的移动在这些节点之间，那么网络爬虫就变成了一个进程就像数学中的图的遍历一样。不同的遍历策略决定 着先不访问哪个节点，下一个访问哪个节点。2种标准的策略是深度优先算法和广度优先算法-他们容易被实现所以在很多入门的算法课中都有教。  
但是，在网络上爬行并不是一个微不足道的设计习题，而是一个非常严峻的算法和系统设计挑战因为以下2点原因：  
网络非常的庞大。现在，Google需要索引超过30亿的网页。很多研究都指出，在历史上，网络每9-12个月都会增长一倍。  
网络的页面改变很频繁。如果这个改变指的是任何改变，那么有40%的网页每周会改变。如果我们认为页面改变三分之一或者更多，那么有大约7%的页面每周会变。  
这2个要素意味着，要获得及时的，完全的网页快照，一个搜索引擎必须访问1亿个网页每天。因此，步骤（a）必须执行大约每秒1000次，成员检测 的步骤（c）必须每秒执行超过10000次，并有非常大的数据储存到主内存中。另外，网络爬虫一般使用一个分布式的构造来平行地爬行更多的网页，这使成员 检测更为复杂：这是可能的成员问题只能回答了一个同行节点，而不是当地。  
 一个非常重要的方法加速这个检测就是用cache（高速缓存），这个是把见过的URLs存入主内存中的一个（动态）子集中。这个论文最主要的成果就是仔细 的研究了几种关于网络爬虫的URL缓存技术。我们考虑所有实际的算法：随机置换，静态cache，LRU，和CLOCK，和理论极限：透视cache和极 大的cache。我们执行了大约1800次模拟，用不同的cache大小执行这些算法，用真实的log日志数据，获取自一个非常大的33天的网络爬行，大 约执行了超过10亿次的http请求。  
 这个论文像这样组织的：第2部分讨论在文学著作中几种不同的爬行解决方案和什么样的cache最适合他们。第3部分介绍关于一些cache的技术和介绍了 关于cache几种理论和实际算法。第4部分我们实现这些算法，在实验机制中。第5部分描述和讨论模拟的结果。第6部分是我们推荐的实际算法和数据结构关 于URLcache。第7部分是结论和指导关于促进研究。  
2.CRAWLING  
 网络爬虫的出现几乎和网络同期，而且有很多的文献描述了网络爬虫。在这个部分，我们呈现一个摘要关于这些爬虫程序，并讨论问什么大多数的网络爬虫会受益于URL cache。  
 网络爬虫用网络存档雇员多个爬行进程，每个一次性完成一个彻底的爬行对于64个hosts 。爬虫进程储存非本地的URLs到磁盘；在爬行的最后，一批工作将这些URLs加入到下个爬虫的每个host的种子sets中。  
 最初的google 爬虫，实现不同的爬虫组件通过不同的进程。一个单独的URL服务器进行维护需要下载的URL的集合；爬虫程序获取的网页；索引进程提取关键字和超链接；URL解决进程将相对路径转换给绝对路径。这些不同的进程通过文件系统通信。  
 这个论文的中实验我们使用的meractor网络爬虫。Mercator使用了一个独立的集合，通信网络爬虫进程。每个爬虫进程都是一个有效的web服务 器子集；URLs的分配基于URLs主机组件。没有责任通过TCP传送这个URL给网络爬虫，有责任把这些URLs绑在一起减少TCP开销。我们描述 mercator很多的细节在第4部分。  
 任何网络爬虫必须维护一个集合，装那些需要被下载的URLs。此外，不能重复地下载同一个URL，必须要个方法避免加入URLs到集合中超过一次。一般的，达到避免可以用维护一个发现URLs的集合。如果数据太多，可以存入磁盘，或者储存经常被访问的URLs。  
3.CACHING  
 在大多数的计算机系统里面，内存是分等级的，意思是，存在2级或更多级的内存，表现出不同的空间和速度。举个例，在一个典型的工作站里，有一个非常小但是 非常快的内存，一个大，但是比较慢的RAM内存，一个非常大胆是很慢的disk内存。在一个网络环境中，也是分层的。Caching就是一种想法储存经常 用到的项目从慢速内存到快速内存。  
 Caching术语就像下面：cache是内存用来储存同等大小的元素。一个cache有k的大小，那么可以储存k个项目.在每个时间段,cache接受 到来自一个项目的请求.如果这个请求项目在这个cache中，这种情况将会引发一个碰撞并且不需要进一步的动作。另一方面，这种情况叫做 丢失或者失败。如果cache没有k个项目，那个丢失的项目被加入cache。另一方面，算法必须选择驱逐一个项目来空出空间来存放那个丢失的项目，或者 不加入那个丢失的项目。Caching算法的目标是最小化丢失的个数。  
 清楚的，cache越大，越容易避免丢失。因此，一个caching算法的性能要在看在一个给定大小的cache中的丢失率。  
 一般的，caching成功有2个原因：  
 不一致的请求。一些请求比其他一些请求多。  
时间相关性或地方的职权范围。  
3.1 无限cache(INFINITE)  
 这是一个理论的算法，假想这个cache的大小要大于明显的请求数。  
3.2 透视cache（MIN）  
 超过35年以前，L?aszl?o Belady表示如果能提前知道完整的请求序列，就能剔除下一个请求最远的项目。这个理论的算法叫MIN，因为他达到了最小的数量关于丢失在任何序列中，而且能带来一个飞跃性的性能提升。  
3.3 最近被用到（LRU）  
 LRU算法剔除最长时间没用被用到的项目。LRU的直觉是一个项目如果很久都没被用过，那么在将来它也会在很长时间里不被用到。  
 尽管有警告“过去的执行不能保证未来的结果”，实际上，LRU一般是非常有效的。但是，他需要维护一个关于请求的优先权队列。这个队列将会有一个时间浪费和空间浪费。  
3.4 CLOCK  
 CLOCK是一个非常流行的接近于LRU，被发明与20世纪60年代末。一个排列标记着M0，M1，….Mk对应那些项目在一个大小为k的cache中。 这个排列可以看作一个圈，第一个位置跟着最后一个位置。CLOCK控制指针对一个项目在cache中。当一个请求X到达，如果项目X在cache中，然后 他的标志打开。否则，关闭标记，知道一个未标记的位置被剔除然后用X置换。  
3.5 随机置换（RANDOM）  
 随机置换完全忽视过去。如果一个项目请求没在cache中，然后一个随机的项目将被从cache中剔除然后置换.  
 在大多数实际的情况下，随机替换比CLOCK要差，但并不是差很多。  
3.6 静态caching（STATIC）  
 如果我们假设每个项目有一个确定的固定的可能性被请求，独立的先前的访问历史，然后在任何时间一个撞击在大小为k的cache里的概率最大，如果一个cache中包含那k个项目有非常大的概率被请求。  
 有2个问题关于这个步骤：第一，一般这个概率不能被提前知道；第二，独立的请求，虽然理论上有吸引力，是对立的地方参考目前在大多数实际情况。  
 在我们的情况中，第一种情况可以被解决：我们可以猜想上次爬行发现的最常用的k个URLs适合于这次的爬行的最常用的k个URLs。（也有有效的技术可以 发现最常用的项目在一个流数据中。因此，一个在线的步骤可以运行的很好）当然，为了达到模拟的目的，我们可以首先忽略我们的输入，去确定那个k个最常用 URLs，然后预装这些URLs到我们做实验的cache中。  
 第二个情况是我们决定测试STATIC的很大的原因：如果STATIC运行的很好，Sname结论是这里有很少的位置被涉及。如果STATIC运行的相对差，那么我们可以推断我们的数据显然是真实被提及的位置，连续的请求是密切相关的。  
4 实验机制  
4.1 Meractor 爬虫体系  
 一个Meractor爬虫体系有一组爬虫进程组成，一般在不同的机器上运行。每个爬虫进程都是总网络服务器的子集，然后由一些工作线程组成（一般有500个），他们负责下载和处理网页从这些服务器。  
 举一个例子，每个工作现场在一个系统里用4个爬行进程。  
 每个工作现场重复地完成以下的操作：它获得一个URL从URL边境里，一个磁盘数据结构维护被下载的URL集合；用HTTP协议下载对应的网页到一个缓冲 区中；如果这个网页是HTML，提取所有的超链接。把提取出来的超链接流转换为完全链接然后运行通过URL过滤器，丢弃一些基于syntactic properties的链接。比如，它丢弃那些基于服务器联系我们，不能被爬行的链接。  
 URL流被送进主机Splitter里面，主机splitter用来分配URL给爬虫进程通过URL的主机名。直到大多数的超链接被关联，大部分的URL（在我们的实验中是81。5%）将被分配给本地爬虫进程；剩下的传说通过TCP给适当的爬虫进程。  
 本地URLs流和从爬虫中得到的URLs流都要送到复制URL消除器中（DUE）。DUE会除去那些被访问过的URLs。新的URLs会被送到URL边境中去以供以后下载。  
 为了避免重复URLs，DUE必须维护发现的URLs的集合。假设今天的网络包括几十亿有效的URLs，内存就需要维护这个集合是非常重要的。 Mercator可以被认为可以维护这个集合通过一个分布式的内存中的hash table（这个地方是每个爬虫进程维护URLs的子集时分配给它）；但是DUE执行（这个强制URLs成8-byte的checksums，而且用前3 —bytes来用作hash table的索引）需要大约5.2bytes 每个URl，意思就是它会用5GB的RAM每个爬虫机器来维护一个10亿个URLs的集合每台机器。这个内存需求非常不合理在很多的设置里，而且实际上， 它对于我们超过了硬件的适用性在这个实验里。因此，我们用一个选择性的DUE来执行那个缓冲器引入URLs到内存中，但是保存大多的URLs（或者更好， 他们的8-bytes checksum）到一个排序好的队列在磁盘中。  
基于磁盘的DUE和主机Splitter都受益于URL caching。给基于磁盘的DUE加一个cache可以使它丢弃引入的URLs，发生碰撞在cache中，替代加入他们到内存缓存区中。而且有个结果 是，内存缓存区要慢些，而且不频繁地和磁盘文件链接。将cache加入到一个主机Splitter中可以丢弃引入的重复的URLs代替将它们传入每个节 点，这样可以减少总的网络通信。这个通信的减少非常重要在爬虫进程没有通过高速LAN连接的时候，但是被替代成球形分布式。在这样一个装置中，每个爬虫负 责让web servers关掉它。 Mercator执行一个遍历通过广度优先算法在网络图上。每个爬虫进程中的线程同时执行。更重要的是，下载的行程被Mercator的 politeness policy调节，它限制每个爬虫的负载咋一些特殊的网络服务器中。Mercator的politeness policy保证没有服务器不断同时收到多个请求；而且，它还保证下一个请求会在它的上一个请求几倍的时间内完成（通常是10倍）。这样一个 politeness policy基本在任何一个大量搜索的网络爬虫中，否则爬虫将会陷入繁重的处理中。  
4.2 我们的网络爬虫  
 我们的网络爬虫由4个Compaq XP1000工作站组成，每个装备一个667MHz的Alpha processor，1.5GB的RAM，144GB的磁盘，和一个100Mbit/sec的以太网连接。每个机器定位于Palo Alto Internet Exchange,十分接近于Internet的backbone。  
 这个爬虫运行从7月12到2002年的9月3日，虽然它活跃地爬行只有33天。下载时由于不同的硬件和网络故障。爬行过程中，那4个机器完成了10.4亿 的下载尝试，7.84亿成功下载。4.29亿的成功下载文档是HTML页面。这些页面包含了大约268.3亿个超链接，相当于每个页面有62.55个超链 接；但是，中间的数值每个超链接只有24个，暗示平均的超链接数被一些包含很多链接的页面扩大了。早期的论文报道每个页面平均只有8个超链或者17个超链 接。我们提供了3个解释关于为什么我们每个页面找到了更过的超链接。首先，我们认为Mercator并没有限制发现URLs在anchor tags，但是更好的是提取所有的tags在可能包含他们的地方。第二，我们认为他下载页面一直16MB的大小（一个设置显著地大于平常），让它可能遇到 上万个的超链接页面.第三，大部分的论文报道那些每个页面中唯一的超链接。如果我们只考虑每个页面中唯一的超链接，那么平均值是47.74，而中间值为 17.  
 那些超链接从这些HTML中提取出来，加上大约3800万的HTTP跳转，在这个爬行中，流入到Host Splitter中。为了去测试不同caching算法的效率，我们通过Mercator的Host Splitter组件将所有的引入URLs打日志到磁盘中.四个爬虫中的Host Splitter接收并日志记录了总共268.6亿个URLs。  
 完成爬行后，我们浓缩了Host Splitter日志文件。我们把每个URL hash化为一个64—bit的识别码。我们确信没有故意的碰撞在排序最初的URL日志文档，而且计算了唯一的URLs个数。然后我们把这些唯一的URL 数和唯一的识别码数比较，我们决定用一个内存hash table在一个内存很大的机器里。数据减少的过程，大小距离51GB到57GB，而且包含64亿和71亿个URLs。  
 为了发现caching的效率，在一个分布式的爬虫程序里的交互的进程通信，我们也获得了一个日志文件，记录那些URLs被传给每个爬虫。这个日志文件包 含49.2亿个URLs，大约相当于全部URLs的19.5%。我们也浓缩了这个日志文件用同样的方式。然后我们会用这个浓缩的日志文件到我们的模拟中。