特别说明

摘要

网络爬虫是一种自动搜集互联网信息的程序。通过网络爬虫不仅能够为搜索引擎采集网络信息，而且可以作为定向信息采集器，定向采集某些网站下的特定信息，如招聘信息，租房信息等。

本文通过 JAVA 实现了一个基于广度优先算法的多线程爬虫程序。本论文阐述了网络爬虫实现中一些主要问题：为何使用广度优先的爬行策略，以及如何实现广度优先爬行；为何要使用多线程，以及如何实现多线程；系统实现过程中的数据存储；网页信息解析等。

通过实现这一爬虫程序，可以搜集某一站点的 URLs，并将搜集到的 URLs 存入数据库。

【关键字】网络爬虫；JAVA；广度优先；多线程。

ABSTRACT

SPIDER is a program which can auto collect informations from internet. SPIDER can collect data for search engines, also can be a Directional information collector, collects specifically informations from some web sites, such as HR informations, house rent informations.

In this paper, use JAVA implements a breadth-first algorithm multi-thread SPDIER. This paper expatiates some major problems of SPIDER: why to use breadth-first crawling strategy, and how to implement breadth-first crawling; why to use multi-threading, and how to implement multi-thread; data structure; HTML code parse. etc.

This SPIDER can collect URLs from one web site, and store URLs into database.

【KEY WORD】SPIDER; JAVA; Breadth First Search; multi-threads.

目录

第一章 引言

随着互联网的飞速发展，网络上的信息呈爆炸式增长。这使得人们在网上找到所需的信息越来越困难，这种情况下搜索引擎应运而生。搜索引擎搜集互联网上数以亿计的网页，并为每个词建立索引。在建立搜索引擎的过程中，搜集网页是非常重要的一个环节。爬虫程序就是用来搜集网页的程序。以何种策略偏历互联网上的网页，也成了爬虫程序主要的研究方向。现在比较流行的搜索引擎，比如 google，百度，它们爬虫程序的技术内幕一般都不公开。目前几种比较常用的爬虫实现策略：广度优先的爬虫程序，Repetitive 爬虫程序，定义爬行爬虫程序，深层次爬行爬虫程序。此外，还有根据概率论进行可用 Web 页的数量估算，用于评估互联网 Web 规模的抽样爬虫程序；采用爬行深度、页面导入链接量分析等方法，限制从程序下载不相关的 Web 页的选择性爬行程序等等。

爬虫程序是一个自动获取网页的程序。它为搜索引擎从互联网上下载网页，是搜索引擎的重要组成部分。爬虫程序的实现策略，运行效率直接影响搜索引擎的搜索结果。不同的搜索引擎，会根据对搜索结果的不同需求，选择最合适的爬行策略来搜集互联网上的信息。高效，优秀的爬虫程序可以使人们在互联网上寻找到更及时，更准确的信息。

实现网络爬虫的重点和难点有：多线程的实现；对临界资源的分配；遍历 web 图的遍历策略选择和实现；存储数据结构的选择和实现。

本文通过 JAVA 语言实现一个基于广度优先偏历算法的多线程爬虫程序。通过实现此爬虫程序可以定点搜集某一站点的 URLs，如果需要搜集其他信息，可以在解析 URLs 的同时，解析获取相应信息。

第二章 相关技术介绍

2.1 JAVA 线程

2.1.1 线程概述

几乎每种操作系统都支持线程的概念—进程就是在某种程度上相互隔离的，独立运行的程序。一般来说，这些操作系统都支持多进程操作。所谓多进程，就是让系统（好像）同时运行多个程序。比如，我在 Microsoft Word 编写本论文的时候，我还打开了一个 mp3 播放器来播放音乐，偶尔的，我还会再编辑 Word 的同时让我的机器执行一个打印任务，而且我还喜欢通过 IE 从网上下载一个 Flash 动画。对于我来说，这些操作都是同步进行的，我不需要等一首歌曲放完了再来编辑我的论文。看起来，它们都同时在我的机器上给我工作。

事实的真相是，对于一个 CPU 而言，它在某一个时间点上，只能执行一个程序。CPU 不断的在这些程序之间“跳跃”执行。那么，为什么我们看不出任何的中断现象呢？这是因为，相对于我们的感觉，它的速度实在太快了。我们人的感知时间可能以秒来计算。而对于 CPU 而言，它的时间是以毫秒来计算的，从我们肉眼看来，它们就是一个连续的动作。

因此，虽然我们看到的都是一些同步的操作，但实际上，对于计算机而言，它在某个时间点上只能执行一个程序，除非你的计算机是多 CPU 的。

多线程（Multi-Thread）扩展了多进程（multi-Process）操作的概念，将任务的划分下降到了程序级别，使得各个程序似乎可以在同一个时间内执行多个任务。每个任务称为一个线程，能够同时运行多个线程的程序称为多线程程序。

多线程和多进程有什么区别呢？对于进程来说，每个进程都有自己的一组完整的变量，而线程则共享相同的数据。

2.1.2 JAVA 线程模型

我们知道，计算机程序得以执行的三个要素是：CPU，程序代码，可存取的数据。在 JAVA 语言中，多线程的机制是通过虚拟 CPU 来实现的。可以形象的理解为,在一个 JAVA 程序内部虚拟了多台计算机,每台计算机对应一个线程，有自己的 CPU，可以获取所需的代码和数据，因此能独立执行任务，相互间还可以共用代码和数据。JAVA 的线程是通过 java.lang.Thread 类来实现的，它内部实现了虚拟 CPU 的功能，能够接收和处理传递给它的代码和数据，并提供了

独立的运行控制功能。

我们知道，每个 JAVA 应用程序都至少有一个线程，这就是所谓的主线程。它由 JVM 创建并调用 JAVA 应用程序的 main（）方法。

JVM 还通常会创建一些其他的线程，不过，这些线程对我们而言通常都是不可见的。比如，用于自动垃圾收集的线程，对象终止或者其他的 JVM 处理任务相关的线程。

2.1.3 创建线程

2.1.3.1 创建线程方式一

在 JAVA 中创建线程的一种方式是通过 Thread 来实现的。Thread 有很多个构造器来创建一个线程（Thread）实例：

Thread();创建一个线程。

Thread(Runnable target);创建一个线程，并指定一个目标。

Thread(Runnable target,String name);创建一个名为 name 的目标为 target 的线程。

Thread(String name);创建一个名为 name 的线程。

Thread(ThreadGroup group,Runnable target);创建一个隶属于 group 线程组，目标为 target 的线程。

通常，我们可以将一个类继承 Thread，然后，覆盖 Thread 中的 run()方法，这样让这个类本身也就成了线程。

每个线程都是通过某个特定 Thread 对象所对应的方法 run()来完成其操作的，方法 run()称为线程体。

使用 start()方法，线程进入 Runnable 状态，它将线程调度器注册这个线程。调用 start()方法并不一定马上会执行这个线程，正如上面所说，它只是进入 Runnble 而不是 Running。

2.1.3.2 创建线程方式二

通过实现 Runnable 接口并实现接口中定义的唯一方法 run()，可以创建一个线程。在使用 Runnable 接口时，不能直接创建所需类的对象并运行它，而是必须从 Thread 类的一个实例内部运行它。

从上面两种创建线程的方法可以看出，如果继承 Thread 类，则这个类本身

可以调用 start 方法，也就是说将这个继承了 Thread 的类当作目标对象；而如果实现 Runnable 接口，则这个类必须被当作其他线程的目标对象。

2.1.4 JAVA 中的线程的生命周期

JAVA 的线程从产生到消失,可分为 5 种状态：新建（New），可运行（Runnable），运行（Running），阻塞（Blocked）以及死亡（Dead）。其中，Running 状态并非属于 JAVA 规范中定义的线程状态，也就是说，在 JAVA 规范中，并没有将运行（Running）状态真正的设置为一个状态，它属于可运行状态的一种。

当使用 new 来新建一个线程时，它处于 New 状态，这个时候，线程并未进行任何操作。

然后，调用线程的 start()方法，来向线程调度程序（通常是 JVM 或操作系统）注册一个线程，这个时候，这个线程一切就绪，就等待 CPU 时间了。

线程调度程序根据调度策略来调度不同的线程，调用线程的 run 方法给已经注册的各个线程以执行的机会，被调度执行的线程进入运行（Running）状态。当线程的 run 方法运行完毕，线程将被抛弃，进入死亡状态。你不能调用 restart 方法来重新开始一个处于死亡状态的线程，但是，你可以调用处于死亡状态的线程对象的各个方法。

如果线程在运行（Running）状态中因为 I/O 阻塞，等待键盘键入，调用了线程的 sleep 方法，调用了对象的 wait()方法等，则线程将进入阻塞状态，直到这些阻塞原因被解除，如：IO 完成，键盘输入了数据，调用 sleep 方法后的睡眠时间到以及其他线程调用了对象的 notify 或 notifyAll 方法来唤醒这个因为等待而阻塞的线程等，线程将返回到 Runnable 状态重新等待调度程序调度，注意，被阻塞的线程不会直接返回到 Running 状态，而是重新回到 Runnable 状态等待线程调度程序的调用。

线程调度程序会根据调度情况，将正在运行中的线程设置为 Runnable 状态，例如，有一个比当前运行状态线程更高运行等级的线程进入 Runnable 状态，就可能将当前运行的线程从 Running 状态 “踢出”,让它回到 Runnable 状态。

2.1.5 JAVA 线程的结束方式

线程会以以下三种方式之一结束：

线程到达其 run()方法的末尾；

线程抛出一个未捕获到的 Exception 或 Error；

另一个线程调用一个 Deprecated 的 stop()方法。注意，因为这个方法会引起线程的安全问题，已经被不推荐使用了，所以，不要再程序调用这个方法。

2.1.6 多线程同步

当同时运行的相互独立的线程需要共享数据并且需要考虑其他线程的状态时，就需要使用一套机制使得这些线程同步，避免在争用资源时发生冲突，甚至发生死锁。JAVA 提供了多种机制以实现线程同步。多数 JAVA 同步是以对象锁定为中心的。JAVA 中从 Object 对象继承来的每个对象都有一个单独的锁。由于 JAVA 中的每个对象都是从 Object 继承来的。所以 JAVA 中的每个对象都有自己的锁。这样使它在共享的线程之间可以相互协调。在 JAVA 中实现线程同步的另一个方法是通过使用 synchronized 关键字。JAVA 使用 synchronized 关键字来定义程序中要求线程同步的部分。synchronized 关键字实现的基本操作是把每个需要线程同步的部分定义为一个临界区，在临界区中同一时刻只有一个线程被执行。

2.2 URL 消重

2.2.1 URL 消重的意义

在 SPIDER 系统实际运行的过程中，每秒下载的 10 个页面中，分析的 URL 大多数是重复的，实际上新的 URL 才几个。在持续下载的过程中，新的 URL 非常少，还是以新浪网举例，1 天 24 小时中总共出现的新 URL 也就是 10000 左右。这种情况非常类似于操作系统中虚拟储存器管理。所谓的虚拟储存器，是指具有请求调入和置换功能，能从逻辑上对内存容量加以扩充的一种储存器系统。其关键在于允许一个作业只装入部分的页或段就可以启动运行，当作业运行的时候在内存中找不到所需要的页或段的时候，就会发生请求调入，而从外存中找到的页或段将会置换内存中暂时不运行的页面到外存。

URL 消重工作量是非常巨大的。以下在新浪新闻页面为例，新浪一个新闻页面大小为 50～60k，每个页面有 90～100 个 URL，如果每秒下载 10 个页面，就会产生 900～1000 次的 URL 排重操作，每次排重操作都要在几百万至几千万的 URL 库中去查询。这种操作对数据库系统是一个灾难，理论上任何需要产生磁盘 I/O 动作的存储系统都无法满足这种查询的需求。

2.2.2 网络爬虫 URL 去重储存库设计

在爬虫启动工作的过程中，我们不希望同一个网页被多次下载，因为重复下载不仅会浪费 CPU 机时，还会为搜索引擎系统增加负荷。而想要控制这种重复性下载问题，就要考虑下载所依据的超链接，只要能够控制待下载的 URL 不重复，基本可以解决同一个网页重复下载的问题。

非常容易想到，在搜索引擎系统中建立一个全局的专门用来检测，是否某一个 URL 对应的网页文件曾经被下载过的 URL 存储库，这就是方案。接着要考虑的就是如何能够更加高效地让爬虫工作，确切地说，让去重工作更加高效。如果实现去重，一定是建立一个 URL 存储库，并且已经下载完成的 URL 在进行检测时候，要加载到内存中，在内存中进行检测一定会比直接从磁盘上读取速度快很多。我们先从最简单的情况说起，然后逐步优化，最终得到一个非常不错的解决方案。

2.2.2.1 基于磁盘的顺序存储

这里，就是指把每个已经下载过的 URL 进行顺序存储。你可以把全部已经下载完成的 URL 存放到磁盘记事本文件中。每次有一个爬虫线程得到一个任务 URL 开始下载之前，通过到磁盘上的该文件中检索，如果没有出现过，则将这个新的 URL 写入记事本的最后一行，否则就放弃该 URL 的下载。

这种方式几乎没有人考虑使用了，但是这种检查的思想是非常直观的。试想，如果已经下载了 100 亿网页，那么对应着 100 亿个链接，也就是这个检查 URL 是否重复的记事本文件就要存储这 100 亿 URL，况且，很多 URL 字符串的长度也不小，占用存储空间不说，查找效率超级低下，这种方案肯定放弃。

2.2.2.2 基于 Hash 算法的存储

对每一个给定的 URL，都是用一个已经建立好的 Hash 函数，映射到某个物理地址上。当需要进行检测 URL 是否重复的时候，只需要将这个 URL 进行 Hash 映射，如果得到的地址已经存在，说明已经被下载过，放弃下载，否则，将该 URL 及其 Hash 地址作为键值对存放到 Hash 表中。

这样，URL 去重存储库就是要维护一个 Hash 表，如果 Hash 函数设计的不好，在进行映射的时候，发生碰撞的几率很大，则再进行碰撞的处理也非常复杂。而且，这里使用的是 URL 作为键，URL 字符串也占用了很大的存储空间。

2.2.2.3 基于 MD5 压缩映射的存储

MD5 算法是一种加密算法，同时它也是基于 Hash 的算法。这样就可以对 URL 字符串进行压缩，得到一个压缩字符串，同时可以直接得到一个 Hash 地址。另外，MD5 算法能够将任何字符串压缩为 128 位整数，并映射为物理地址，

而且 MD5 进行 Hash 映射碰撞的几率非常小，这点非常好。从另一个方面来说，非常少的碰撞，对于搜索引擎的爬虫是可以容忍的。况且，在爬虫进行检测的过程中，可以通过记录日志来保存在进行 MD5 时发生碰撞的 URL，通过单独对该 URL 进行处理也是可行的。

在 Java 中有一个 Map 类非常好，你可以将压缩后的 URL 串作为 Key，而将 Boolean 作为 Value 进行存储，然后将工作中的 Map 在爬虫停止工作后序列化到本地磁盘上；当下一次启动新的爬虫任务的时候，再将这个 Map 反序列化到内存中，供爬虫进行 URL 去重检测。

2.2.2.4 基于嵌入式 Berkeley DB 的存储

Berkeley DB 的特点就是只存储键值对类型数据，这和 URL 去重有很大关系。去重，可以考虑对某个键，存在一个值，这个值就是那个键的状态。使用了 Berkeley DB，你就不需要考虑进行磁盘 IO 操作的性能损失了，这个数据库在设计的时候很好地考虑了这些问题，并且该数据库支持高并发，支持记录的顺序存储和随机存储，是一个不错的选择。

URL 去重存储库使用 Berkeley DB，压缩后的 URL 字符串作为 Key，或者直接使用压缩后的 URL 字节数组作为 Key，对于 Value 可以使用 Boolean，一个字节，或者使用字节数组，实际 Value 只是一个状态标识，减少 Value 存储占用存储空间。

2.2.2.5 基于布隆过滤器（Bloom Filter）的存储

使用布隆过滤器，设计多个 Hash 函数，也就是对每个字符串进行映射是经过多个 Hash 函数进行映射，映射到一个二进制向量上，这种方式充分利用了比特位。不过，我没有用过这种方式，有机会可以尝试一下。可以参考 Google 的 http://www.googlechinablog.com/2007/07/bloom-filter.html。

2.2.3 LRU 算法实现 URL 消重

用双向链表来实现大容量 cache 的 LRU 算法。原理是：cache 的所有位置都用双向链表连接起来，当一个位置被命中后，就将通过调整链表的指向将该位置调整到链表的头位置，新加入的内容直接放在链表的头上。这样，在进行过多次查找操作后，最近被命中过的内容就像链表的头移动，而没有命中过的内容就向链表的后面移动。当需要替换时，链表最后的位置就是最近最少被命中位置，我们只需要将新的内容放在链表前面，淘汰链表最后的位置就实现了 LRU 算法。

2.3 URL 类访问网络

JAVA 提供了许多支 Internet 连接的类，URL 类就是其中之一。在使用 URL 类之前，必须创建一个 URL 对象，创建的方法是使用其构造函数，通过向其指定一个 URL 地址，就能实例化该类。如：URL url=new URL（http://www.sina.com.cn）；

如果传递无效的 URL 给 URL 对象，该对象会抛出 MalformedURLException 异常。当成功创建一个 URL 对象后，我们调用 openConnection 函数建立与 URL 的通信，此时，我们就获得了一个 URLConnection 对象的引用，URLConnection 类包含了许多与网络上的 URL 通信的函数。在下载网页前，我们需要判断目标网页是否存在，这时调用 URLConnection 类的 getHeaderField()方法，获得服务器返回给 SPIDER 程序的响应码，如果响应码包含”20*”字样，表示目标网页存在，下一步就下载网页，否则就不下载。getHeaderField()方法仅仅获得服务器返回的头标志，其通信开销是最小的，因此在下载网页前进行此测试，不仅能减小网络流量，而且能提高程序效率。当目标网页存在时 2 调用 URLConnection 类 getInputStream()函数明确打开到 URL 的连接，获取输入流，再用 java.io 包中的 InputStreamReader 类读取该输入流，将网页下载下来。

2.4　爬行策略浅析

2.4.1 宽度或深度优先搜索策略

搜索引擎所用的第一代网络爬虫主要是基于传统的图算法，如宽度优先或深度优先算法来索引整个 Web，一个核心的 U RL 集被用来作为一个种子集合，这种算法递归的跟踪超链接到其它页面，而通常不管页面的内容，因为最终的目标是这种跟踪能覆盖整个 Web. 这种策略通常用在通用搜索引擎中,因为通用搜索引擎获得的网页越多越好，没有特定的要求.

2.4.1.1　宽度优先搜索算法

宽度优先搜索算法(又称广度优先搜索) 是最简便的图的搜索算法之一，这一算法也是很多重要的图的算法的原型. Dijkstra 单源最短路径算法和 Prim 最小生成树算法都采用了和宽度优先搜索类似的思想.宽度优先搜索算法是沿着树的宽度遍历树的节点，如果发现目标，则算法中止. 该算法的设计和实现相对简单，属于盲目搜索. 在目前为覆盖尽可能多的网页，一般使用宽度优先搜

索方法. 也有很多研究将宽度优先搜索策略应用于聚焦爬虫中. 其基本思想是认为与初始 U RL 在一定链接距离内的网页具有主题相关性的概率很大. 另外一种方法是将宽度优先搜索与网页过滤技术结合使用, 先用广度优先策略抓取网页, 再将其中无关的网页过滤掉. 这些方法的缺点在于, 随着抓取网页的增多, 大量的无关网页将被下载并过滤, 算法的效率将变低.

2.4.1.2 深度优先搜索

深度优先搜索所遵循的搜索策略是尽可能“深”地搜索图. 在深度优先搜索中, 对于最新发现的顶点, 如果它还有以此为起点而未探测到的边, 就沿此边继续汉下去. 当结点 v 的所有边都已被探寻过, 搜索将回溯到发现结点 v 有那条边的始结点. 这一过程一直进行到已发现从源结点可达的所有结点为止. 如果还存在未被发现的结点, 则选择其中一个作为源结点并重复以上过程, 整个进程反复进行直到所有结点都被发现为止. 深度优先在很多情况下会导致爬虫的陷入(trapped) 问题, 所以它既不是完备的, 也不是最优的.

2.4.2 聚焦搜索策略

基于第一代网络爬虫的搜索引擎抓取的网页一般少于 1 000 000 个网页, 极少重新搜集网页并去刷新索引. 而且其检索速度非常慢, 一般都要等待 10 s 甚至更长的时间. 随着网页页信息的指数级增长及动态变化, 这些通用搜索引擎的局限性越来越大, 随着科学技术的发展, 定向抓取相关网页资源的聚焦爬虫便应运而生.聚焦爬虫的爬行策略只挑出某一个特定主题的页面, 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, 主要通过内容和 Web 的链接结构来指导进一步的页面抓取[2].

聚焦爬虫会给它所下载下来的页面分配一个评价分, 然后根据得分排序, 最后插入到一个队列中. 最好的下一个搜索将通过对弹出队列中的第一个页面进行分析而执行, 这种策略保证爬虫能优先跟踪那些最有可能链接到目标页面的页面. 决定网络爬虫搜索策略的关键是如何评价链接价值, 即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, 表现出的链接的“重要程度”也不同, 从而决定了不同的搜索策略. 由于链接包含于页面之中,而通常具有较高价值的页面包含的链接也具有较高的价值, 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, 因为这种搜索引擎只关心某一特定主题的页面.

2.4.3 基于内容评价的搜索策略

基于内容评价的搜索策略[3,4], 主要是根据主题(如关键词、主题相关文档)

与链接文本的相似度来评价链接价值的高低, 并以此决定其搜索策略: 链接文本是指链接周围的说明文字和链接 U RL 上的文字信息, 相似度的评价通常采用以下公式:

$$sim(d\,i, d\,j) = \Sigma\ mk=1 w\,ik \times w\,jk / (\Sigma\ mk=1 w\,2ik)(\Sigma\ mk=1 w\,2jk)$$

其中, di 为新文本的特征向量, d j 为第 j 类的中心向量,m 为特征向量的维数,wk 为向量的第 K 维.由于 Web 页面不同于传统的文本, 它是一种半结构化的文档, 包含许多结构信息 Web 页面不是单独存在的, 页面中的链接指示了页面之间的相互关系, 因而有些学者提出了基于链接结构评价链接价值的方法.

2.4.4 基于链接结构评价的搜索策略

基于链接结构评价的搜索策略, 是通过对 Web 页面之间相互引用关系的分析来确定链接的重要性, 进而决定链接访问顺序的方法. 通常认为有较多入链或出链的页面具有较高的价值. PageRank 和 Hits 是其中具有代表性的算法.

2.4.4.1 PageRank 算法

基于链接评价的搜索引擎的优秀代表是 Google (http://www.Google.com) , 它独创的“链接评价体系”(PageRank 算法) 是基于这样一种认识, 一个网页的重要性取决于它被其它网页链接的数量, 特别是一些已经被认定是“重要”的网页的链接数量. PageRank 算法最初用于 Google 搜索引擎信息检索中对查询结果的排序过程[5] , 近年来被应用于网络爬虫对链接重要性的评价, PageRank 算法中, 页面的价值通常用页面的 PageRank 值表示, 若设页面 p 的 PageRank 值为 PR (p) , 则 PR (p) 采用如下迭代公式计算:

$$PR(p) = C\,1T + (1 - C)\,\Sigma\,C \in in(p) PR(p) \hat{u} out(C)\,\hat{u}$$

其中 T 为计算中的页面总量, C< 1 是阻尼常数因子, in (p) 为所有指向 p 的页面的集合, out (C) 为页面 C 出链的集合. 基于 PageRank 算法的网络爬虫在搜索过程中, 通过计算每个已访问页面的 PageRank 值来确定页面的价值, 并优先选择 PageRank 值大的页面中的链接进行访问.

2.4.4.2 H ITS 算法

HITS 方法定义了两个重要概念: Authority 和 Hub. Authority 表示一个权威页面被其它页面引用的数量, 即该权威页面的入度值. 网页被引用的数量越大, 则该网页的 Authority 值越大; Hub 表示一个 Web 页面指向其它页面的数量, 即该页面的出度值. 网页的出度值越大, 其 Hub 值越高. 由于 Hub 值高的页面通常都提供了指向权威页面的链接, 因而起到了隐含说明某主题页面权威性的

作用.HITS (Hyperlink- Induced Top ic Search) 算法是利用 HuböAuthority 方法的搜索方法,Authority 表示一个页面被其它页面引用的数量, 即该页面的入度值. Hub 表示一个 Web 页面指向其它页面的数量, 即该页面的出度值. 算法如下: 将查询 q 提交给传统的基于关键字匹配的搜索引擎. 搜索引擎返回很多网页, 从中取前 n 个网页作为根集, 用 S 表示.通过向 S 中加入被 S 引用的网页和引用 S 的网页将 S 扩展成一个更大的集合 T.以 T 中的 Hub 网页为顶点集 V l, 以权威网页为顶点集 V 2,V1 中的网页到 V_2 中的网页的超链接为边集 E , 形成一个二分有向图 $S G = (V 1, V 2, E)$.对 V 1 中的任一个顶点 v , 用 $H(v)$ 表示网页 v 的 Hub 值, 对 V 2 中的顶点 u, 用 $A(u)$ 表示网页的 Authority 值. 开始时 $H(v) = A(u) = 1$, 对 u 执行公式(1) 来修改它的 $A(u)$, 对 v 执行公式(2) 来修改它的 $H(v)$, 然后规范化 $A(u)$, $H(v)$, 如此不断的重复计算上述运算, 直到 $A(u)$, $H(v)$ 收敛.

$$A(u) = \Sigma v: (v, u) \in EH(v) \quad (1)$$

$$H(v) = \Sigma v: (v, u) \in EA(v) \quad (2)$$

式(1) 反映了若一个网页由很多好的 Hub 指向, 则其权威值会相应增加(即权威值增加为所有指向它的网页的现有 Hub 值之和). 式(2) 反映了若一个网页指向许多好的权威页, 则 Hub 值也会相应增加(即 Hub 值增加为该网页链接的所有网页的权威值之和).虽然基于链接结构价的搜索考虑了链接的结构和页面之间的引用关系, 但忽略了页面与主题的相关性, 在某些情况下, 会出现搜索偏离主题的问题. 另外, 搜索过程中需要重复计算 PageRank 值或 Authority 以及 Hub 权重, 计算复杂度随页面和链接数量的增长呈指数级增长[6].

2.4.5 基于巩固学习的聚焦搜索

近年来对 Web 信息资源分布的研究表明很多类型相同的网站在构建方式上, 主题相同的网页在组织方式上都存在着一定的相似性, 有的学者就考虑将巩固学习引入网络爬虫的训练过程中, 从这些相似性获取一些“经验”, 而这些经验信息在搜索距相关页面集较远的地方往往能获得较好的回报, 而前两种策略在这种情况下容易迷失方向.在巩固学习模型中, 把网络爬虫经过若干无关页面的访问之后才能获得的主题相关页面称为未来回报, 对未来回报的预测值称为未来回报价值, 用 Q 价值表示. 这种方法的核心就是学习如何计算链接的 Q 价值, 根据未来回报价值确定正确的搜索方向. 目前这类搜索策略不足之处在于学习效率低的问题, 而且在训练过程中增加了用户的负担.

2.4.6 基于语境图的聚焦搜索

基于巩固学习的网络爬虫通过计算链接的 Q 价值可以确定搜索方向, 但它却无法估计距离目标页面的远近. 为此, Diligent 等提出了基于“语境图”的搜索策略, 它通过构建典型页面的 web“语境图”来估计离目标页面的距离, 距离较近的页面较早得到访问[7].基于“语境图”的搜索策略需要借助已有的通用搜索引擎构建“语境图”, 而搜索引擎的检索结果并非一定代表真实的 web 结构, 因而这种方式也具有局限性.

第三章 系统需求分析及模块设计

3.1 系统需求分析

SPIDER 要获取的对象是存在于网络上数以亿计的网页，这些网页以超链接形式互相联系在一起，每一网页对应一个超链接，也称统一资源定位符（URL）。我们可以把网络看做一个图 M(V,E)，网络中的网页构成节点集 V，他们之间的链接构成边集 E，SPIDER 正是从某一节点开始，沿着边，遍历图 M，每访问到图中一个节点 Vi，就进行一定的处理。

为了达到上述目的，一个 SPIDER 必须被设计成多线程的，A 个线程并发地在网络上协同工作，才有可能在尽可能短的时间内遍历完网络中的网页。但网页数目是如此之大，如果任 SPIDER 程序无穷地搜索下去，那么程序几乎不能终止。所以我们限制 SPIDER 每次工作只访问一个站点。一个再大型的站点，其中的网页数目也是有限的，因此 SPIDER 程序能在有限的时间内结束。

当 SPIDER 程序访问到一个网页，必须进行以下几项基本处理：抽取网页中包含的文本；抽取网页中包含的 URL，并将其区分为网站内 URL 或网站外 URL。

3.2 SPIDER 体系结构

此爬虫程序主要分为三个部分：任务执行端，任务调度端，数据服务端。

每一个 SPIDER 任务执行端关联一个站点,一个线程下载一个基于 URL 链接的页面, 并进行 Web 页面解析, 得到站内 URL 和发现新站点 URL 另外,将 URL 队列持久化到数据库, 因此在 SPIDER 任务执行端以外 Down 掉后, 能够断点续传.

SPIDER 客户端线程间的协调通信采用 Java 的线程同步技术 synchronized, 在数据服务端中对 URL 进行缓存提高了系统处理速度. SPIDER 的任务执行和任务调度端都需要维持一个 URL 队列: 任务执行端的 URL 队列中存储了站内 URL; 任务调度端则是站点的 URL. 在这些 URL 队列上有大量的操作, 包括 URL 查找、 URL 插入、URL 状态更新等. 如果 SPIDER 以 300 页 ö 秒的速度下载 Web 页面, 平均将会产生 2000 多个 URL [12] , 因此简单的采用内存数据结构存储这些 URL 队列有一定的问题, 系统没有足够的内存空间;而采用直接持久化到数据库, 则需要大量的数据库连接、查询等操作, 系统效率会明显下降. 如果采用 URL 压缩的办法,尽管在一定程度上可以平衡空间和时间的矛盾, 但仍然不适用于大规模数据采集的 SPIDER.

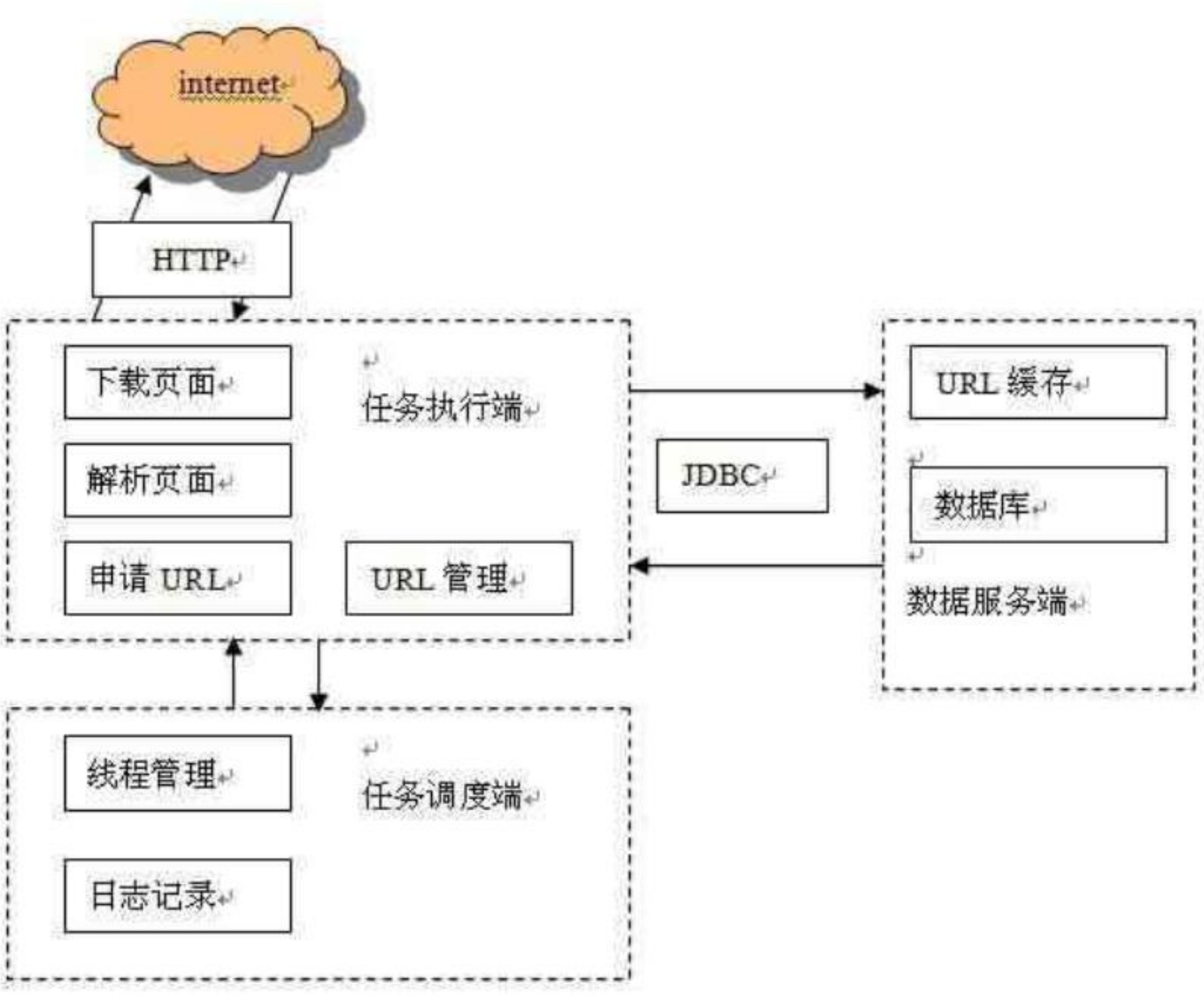


图 3.2 SPIDER 体系结

3.3 各主要功能模块（类）设计

SPIDERWorker 类：该类继承自线程类，请求任务 URL，根据得到的 URL 下载相应的 HTML 代码，利用 HTML 代码调用其他模块完成相关处理。

SPIDERManager 类：该类用于控制 SPIDERWorker 线程。

UrlQueueManager 类：该类用于维护 URL 等待队列，分配任务 URL，URL 消重模块。

UrlParse 类：用于解析 HTML，获取并过滤 URL。

DateBaseConnect 类：用于提供数据库连接。

3.4 SPIDER 工作过程

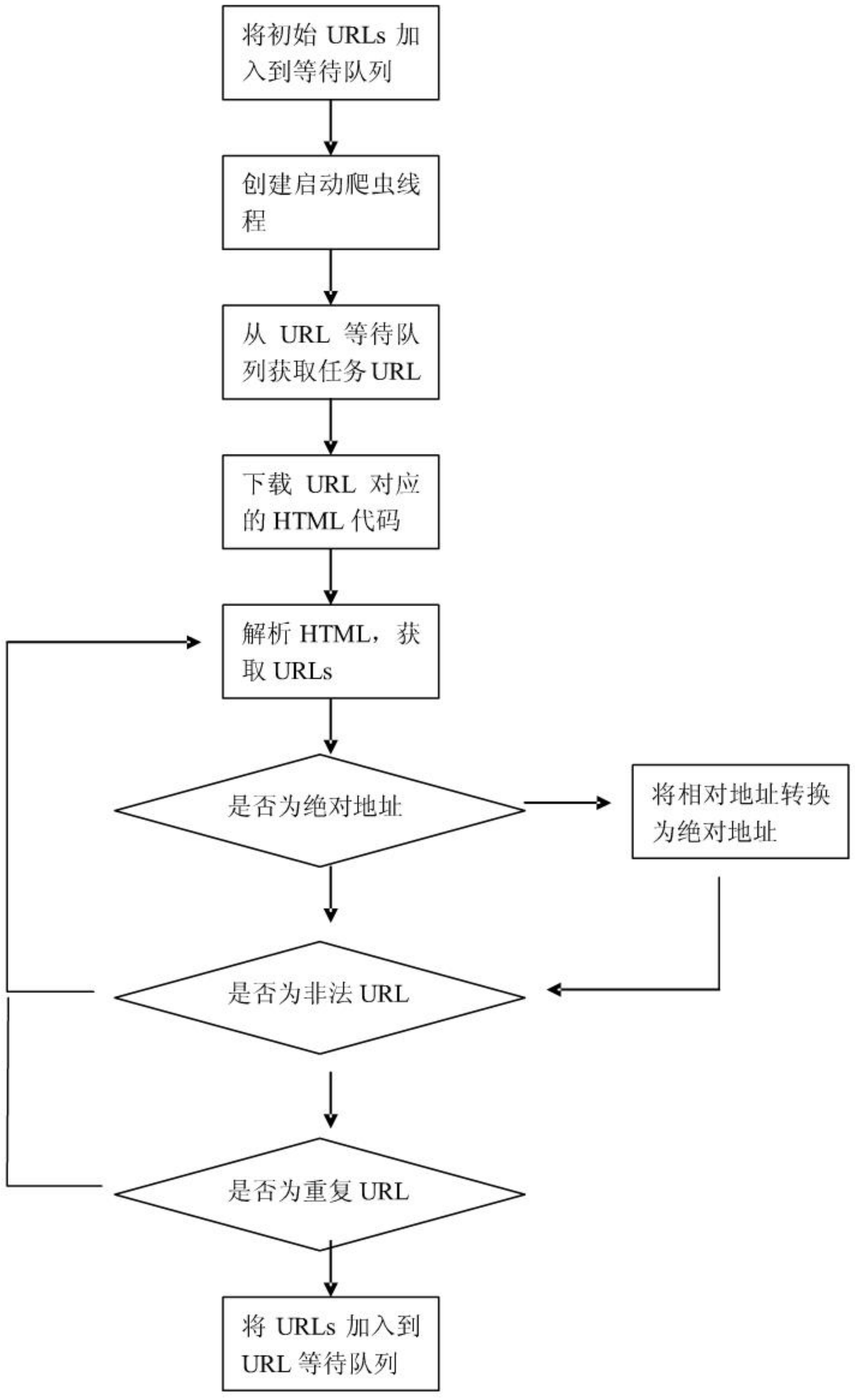
①将给定的初始 URL 加入到 URL 等待队列。

②创建爬虫线程，启动爬虫线程

③每个爬虫线程从 URL 等待队列中取得任务 URL。然后根据 URL 下载网

页，然后解析网页，获取超链接 URLs。如果获取到的 URL 为相对地址，需要转换为绝对地址，然后淘汰站外 URLs，错误 URLs 或者不能解析的 URL 地址。再判断这些 URL 是否已经被下载到，如果没有则加入到 URL 等待队列。

④继续执行步骤③，直到结束条件停止。



第四章 系统分析与设计

4.1 SPIDER 构造分析

构造 SPIDER 程序有两种方式：（1）把 SPIDER 程序设计为递归的程序；（2）编写一个非递归的程序，它要维护一个要访问的网页列表。考虑使用哪一种方式的前提是，构造的 SPIDER 程序必须能够访问非常大的 Web 站点。本系统中使用了非递归的程序设计方法。这是因为，当一个递归程序运行时要把每次递归压入堆栈，但在本系统设计中使用的是多线程，它允许一次运行多个任务，但是，多线程与递归是不兼容的。因为在这一过程中每一个线程都有自己的堆栈，而当一个方法调用它自身时，它们需要使用同一个堆栈。这就意味着递归的 SPIDER 程序不能使用多线程。

每个SPIDER线程都会独立的去完成获取URLs的任务，并将获取到的URLs加入一个公共的 URL 等待队列中。

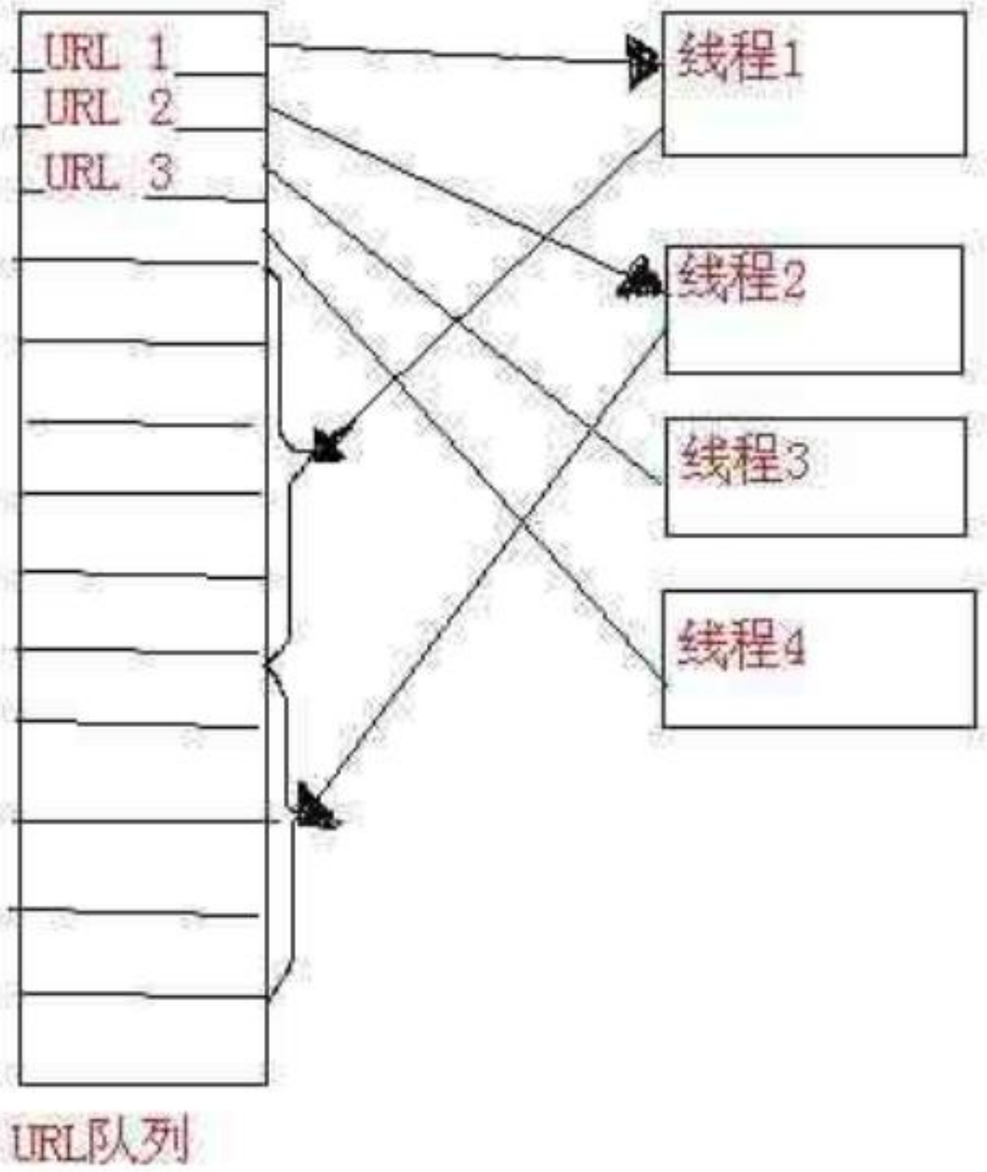


图 4.1

图 4.1 表示了该系统爬虫线程的实现策略。假设线程 1 从 URL 队列中获取一条任务 URL 1，然后它会下载对应的 HTML，解析出里面包含 URLs，然后

再将这些 URLs 加入到 URL 队列中去。然后线程 1 会再从 URL 队列中获取新的 URL，下载 HTML 代码，并解析出 URLs，再加入到 URL 队列中去。而线程 2 同时也会下载它获取到的 URL 2 对应的 HTML 代码，解析出 URLs 加入到等待队列中。以此类推，多个线程并发地去完成爬虫工作。

4.2 爬行策略分析

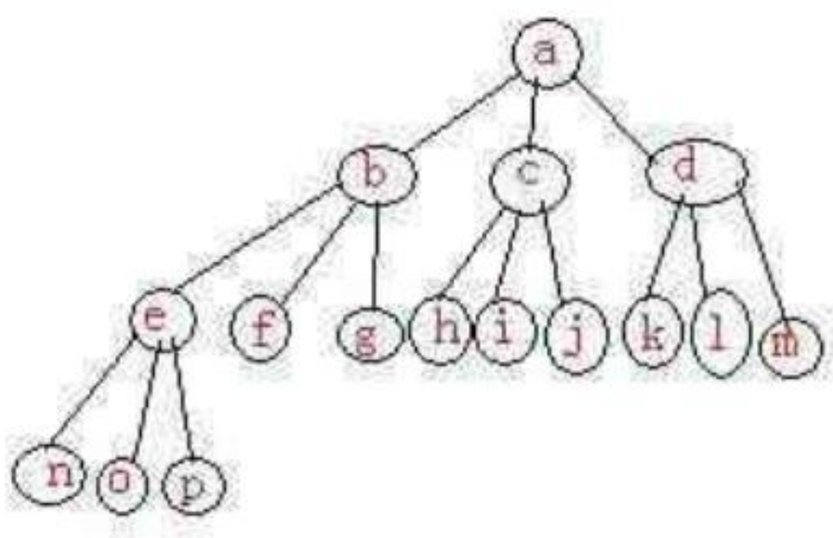


图 4.2.1

因为本论文实现的爬虫程序的初衷是尽可能遍历某一站点所有的页面。广度优先算法的实行理论是覆盖更多的节点，所以此爬虫程序选择了广度优先算法。实现的策略是：先获取初始 URL 对应 HTML 代码里所有的 URLs。然后依次获取这些 URLs 对应的 HTML 里的 URLs，当这一层所有的 URLs 都下载解析完后，在获取下一层的信息。通过这种循环的获取方式实现广度优先爬行。

如图 4.2.1，假如 a 代表初始 URL，bcd 为以 a 获取的 3 个 URLs，efg 为以 b 获取的 URLs，以此类推。那么这些 URLs 获取的顺序就是 abcdefghijklmnop 这样一个顺序。当获取到 b 的 URLs 之后，并不会马上去解析这些 URLs，而是先解析同 b 在同一层中的 cd 对应的 URLs。当这一层 URLs 全部解析完后，再开始下一层 URLs。

广度优先算法的等待队列设计如图 4.2.2 所示。

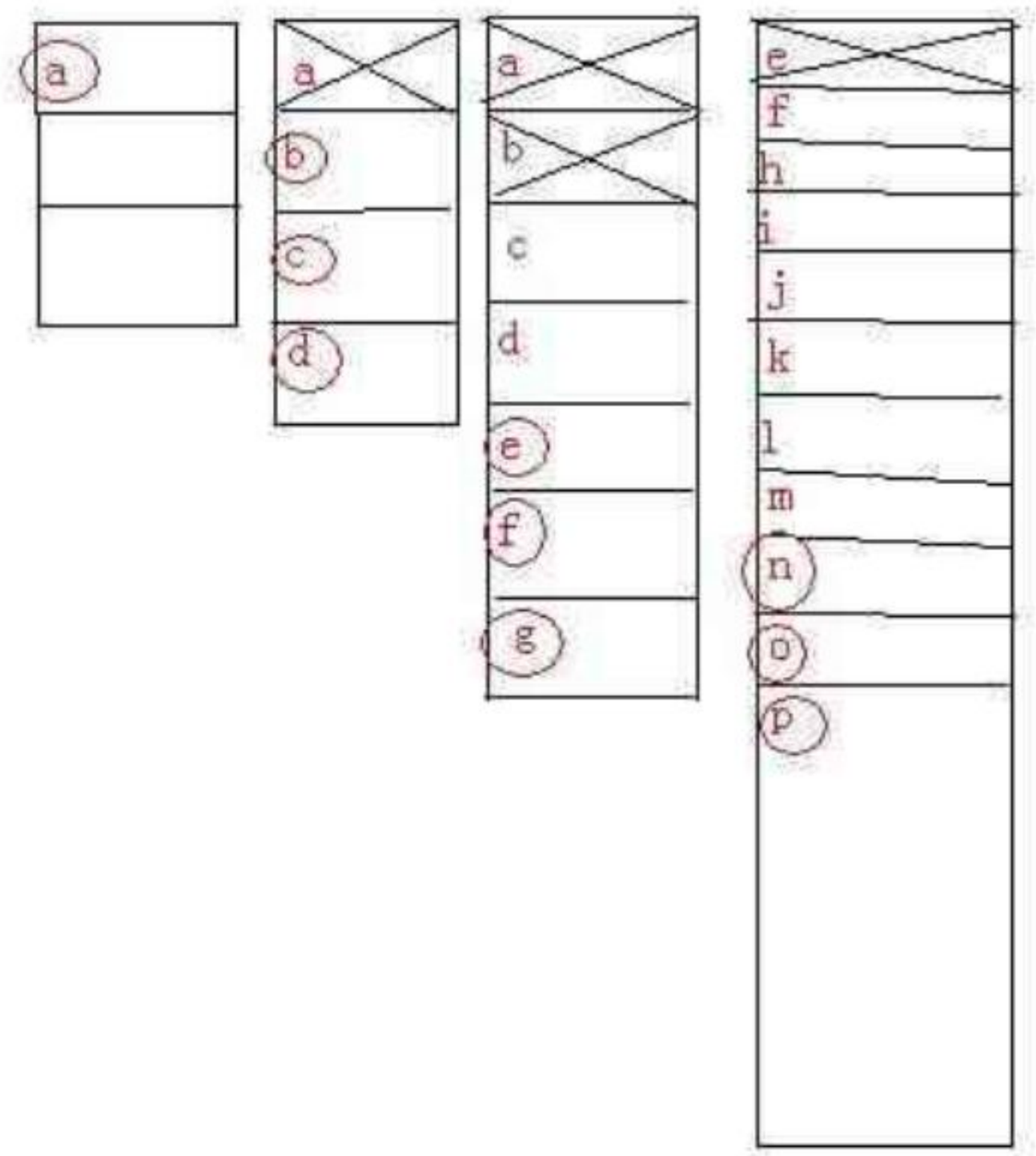


图 4.2.2

图 4.2.2 列举了不同时间段时，URL 等待队列的存储状态。第一个方框是将初始 URL：a 加入到等待队列。第二个方框为，解析 a 对应 HTML 获取 URLs：bcd，同时删除 a。第三个方框为，解析 b 对应 HTML 获取 URLs：efg，同时删除 URL：b。第四个方框为，解析 e 对应 HTML 获取 URLs：nop，并删除 e。通过这样的存储方法实现如图 4.2.1 的广度爬行算法。

4.3 URL 抽取，解析和保存

4.3.1 URL 抽取

通过观察研究 HTML 代码，我们可以知道。HTML 代码中，页面之间的跳转，关联是通过 href 标签来实现的。我们需要获取 HTML 代码中的 URLs，就可以通过寻找 href 标签来达到目的。

 我 猜 [20090531]5-31

 火影忍者[331，页面上 303 既是]

通过观察得知，一般 href 标签是以 href=这样的形式出现的。但是不同的网站 href=后面的内容有所不同。比如 href=”url”这样情况，我们就可以通过截取双引号之间的内容来获取 URL；如果是 href='url'这种情况，我们就需要截取单引号之间的内容来获取 URL；或者有些是 href=url，我们需要以等号为开始标记，而这种情况通常结尾是空格或者>符号。

通过这种方法，我们获取网页中大部分的 URLs。但是有些 URLs 是通过提交表单，或者通过 javascript 来跳转的。这些情况就需要更细致的考虑，才能获取。

4.3.2 URL 解析

截取出来的字符串，可能为相对地址或者绝对地址。所以需要判断 URL 为绝对地址，还是相对地址。相对地址需要先转化为绝对地址，再进行过滤。因为解析出来的 URL 地址可能是一些文件的地址，或者为 javascript 文件或者 css 文件。这些格式的 URL 是无法获取 HTML 代码的，也就不可能进行 URL 解析。所以我们需要过滤掉这些 URLs。然后再进行 URL 消重处理，最后加入到 URL 等待队列。

为了把爬行限制在同一站点内需要截断指向站外的链接,保证 SPIDER 总在站内执行,即准确地根据超链 URL 判断超链是否指向站外.由 RFC 对 URL 的定义可知,URL 的格式为[protocol://host:port/path?query],一般情况下,同一网站内所有页面对应 URL 的 host 是相同的,所以可以使用 host 匹配作为判断超链是否指向站外的标准. 进一步研究发现,很多大型网站中一个分类目录对应一个主机,所以前面的判断标准必须改进.研究 host 的组成可知, host 的格式一般为[站内分类.站点标志串.站点类型各异的串].站点类型串只有[com |edu|gov|net| 国家域名]几种类型,所以我们取站点类型各异串前面的串,即站点标志串作匹配,超链 URL 的 host 中是否包含此串,为超链是否站内的判断标准.

4.3.3 URL 保存

因为等待 URLs 的数目非常多，如果全部采用 List 来存储非常的占用内存空间。所以将等待 URLs 存入数据库中，并设计 2 个缓存区，用于向队列中加入和取得 URLs。URL 等待队列设计成三段式：第一段为一个 List,用来加入新得到的 URL。当这个 List 中的数目过多时，则将 List 中的内容加入到数据库，并清空该 List，以便加入新的 URLs；第二段为数据库，当第一段数据过多时，将第一段内的数据存入数据库；第三段也为一个 List，从这里面分配任务 URL，当该 List 内 URL 不足时，将数据库里的数据再转存入。

图 4.3.3 表示了 URL 等待队列的结构。

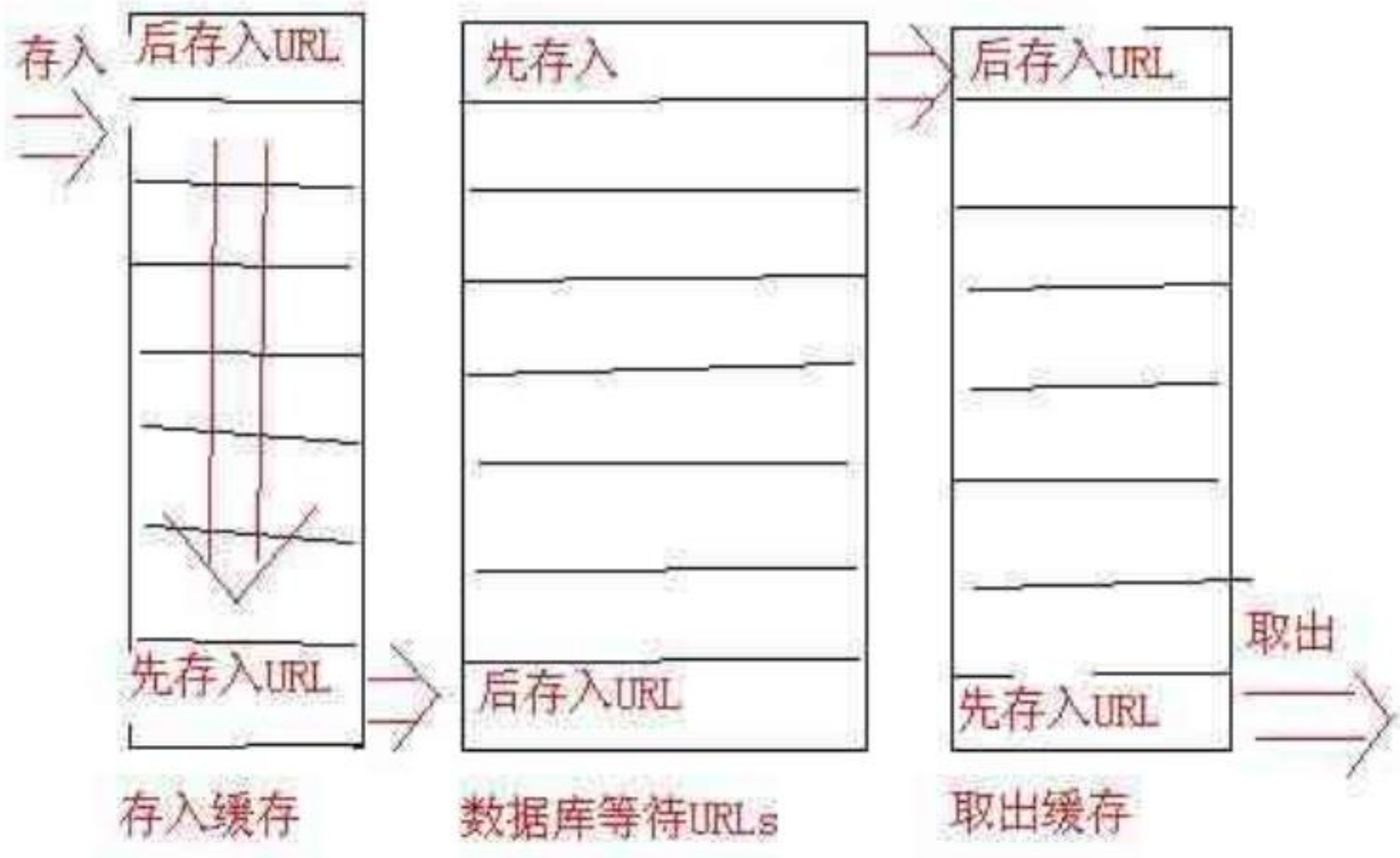


图 4.3.3

第五章 系统实现

5.1 实现工具

操作系统是 winXP;JAVA 程序的编写工具是 eclipse-SDK-3.2.1-win32；数据库是 MYSQL 5 5.0.51a。

5.2 爬虫工作

这个类继承自线程类，实现线程在 java 中有两种方法：一种是直接继承 Thread 类；一种是实现 Runnable 接口。我采用了第二种方法：

public class SpiderWorker implements Runnable {

在这个类中必须要实现重写 run()这个方法。我在这个方法里定义了一个循环，这个线程会重复地执行爬虫动作。

在这个循环里，首先会向 URL 等待队列里请求一个 URL。因为 URL 队列会出现为空或者被其他线程占用的情况。所以我在这里写了一个循环：

```
s = null;
while (s == null) {
    s = urlQueueManager.getWaitQueue();

}
```

如果没有得到 URL 就继续向 URL 等待队列申请。

当得到任务 URL 以后，会通过这个 URL 得到对应的 HTML 代码。具体方法是调用 getHtml(String sourse_url)这个方法：

```
HttpURLConnection connection = null;
InputStreamReader in = null;
BufferedReader br = null;
URL url = null;
    url = new URL(sourse_url);
    connection = (HttpURLConnection)
url.openConnection();
    connection.connect();

    // 打开的连接读取的输入流。
    in = new InputStreamReader(url.openStream());
    br = new BufferedReader(in);
    String c;
```

```
while ((c = br.readLine()) != null) {
    html.append(c);
}
return html.toString();
```

这个方法是通过调用 JAVA 里面的 URL 这个类，可以用给定的 URL 构造这个类的一个实例，然后通过 openStream()这个方法得到 HTML 代码的数据流，然后再一行一行地把数据流转换成 String 字符串，再用 StringBuffer 将这些字符串拼接成一个完整的 HTML 代码。

当得到 HTML 代码以后，程序就会调用 Url_Parse 这个类里面的方法来解析 HTML。

5.3 URL 解析

从 HTML 代码中提取 URLs，主要是通过检索字符串中的 href 字符串来实现的。对于一个 HTML 代码，我寻找其中的 href=字符串，然后记录它的下表 i。然后判断下表 i+1 位置上的字符是双引号，单引号或者两者皆不是，然后选择对应的字符作为截取 URL 的终止标记。截取过后的 href 标记就剔除它与它前面的部分，以便而后的操作可以继续检索 href 标记，直到正个 HTML 代码中所有的 href 标记都被解析过后，操作终止。

 首 页 动 作 片 恐 怖 片 爱情片

例如上面那段 HTML 代码。我们先检索 href=标记，然后判断出第 i+1 位为一个双引号，所以我们可以截取 i+1 位到第 2 个双引号的位置。之间的这段字符串即为 URL。当完成这一步操作后，原字符串被截取从“ class=”开始。我们继续检索 href=标签，判断它的第 i+1 位为一个单引号，所以我们又截取 i+1 位到第 2 个单引号的位置。这步以后原字符串又被截取为“ target=”开始，我们可以继续检索 href=标签。这个地方 href=没有接续任何符号，所以当我们没有发现单引号或双引号的时候，可以判断为这种情况。我们就去检索空格和< 标签，以下标较小的字符作为截取 URL 的结束标记。

下面是 href=后面接续双引号情况的 JAVA 代码，其他情况的代码方式相同。

```
public void getHref_UrlsList(String html_text, String fromURL,
        UrlQueueManager urlQueueManager, int biaoji) {
```

```
// 站内URL队列
List<String> siteInsideUrls = new ArrayList<String>();
String url = "";

// HTML中是否还含有href标签
boolean haveHref = html_text.contains("href");

while (haveHref) {
        html_text = html_text.substring(html_text.indexOf("href=")
+ 5);

        // 当 href= 后以 " 开头的情况
        if ('\"' == html_text.charAt(0)) {
            html_text = html_text.substring(1);
            url = html_text.substring(0, html_text.indexOf("\""));
            url = addURLhost(fromURL, url);
            if (isSiteInsideUrl(url, urlQueueManager)) {
                if (!urlQueueManager.isContainUrl(url)) {
                    siteInsideUrls.add(url);
                }
            }
            haveHref = html_text.contains("href");
        }
        urlQueueManager.waitQueueAdd(siteInsideUrls);
    }
```

在每个 URL 被截取出来之后，需要判断这些 URL 是相对地址，还是绝对地址。

动作片

TOM365

例如上面的 HTML 代码，如果截取出来的 URL 为../mlist/1_1.html 这种形式，即为相对地址。我们需要将其转化为绝对地址。假如这个相对地址的父 URL 为 http://www.tom365.com/movie_2004/html/6664.html。根据相对地址的概念，../为返回上一层，所以可以得到这个相对地址的绝对地址为

http://www.tom365.com/mlist/1_1.html。比如像上面的第 2 种 URL，它包含完整的协议信息，域名地址。可以判断它为绝对地址。

当得到这些完整的 URL 地址以后，我们需要对其进行过滤。很多 URL 它们指向的文件不是 HTML 文件，而是一些 CSS 文件，或者 RAR 包文件，或者

只是接续“#”符号，代表只是调用一段 javascript 代码。像这种情况我们就直接抛弃这些 URLs。

下面是一段实行代码。代码通过检索 URL 字符串中是否包含".css"，".rar"，".zip"这些后缀来进行判断。

```
// 如果url中包含以下 字符串，则不加入队列
if (url.toLowerCase().contains(".css")
        || url.toLowerCase().contains(".rar") ||
url.contains("#")
        || url.contains(".zip") || url.contains("javascript"))
{

    return false;
}
```

过滤完后的 URLs，再判断它为站内 URL 或者为站外 URL。一般情况下同一网站内的 URL 的 host 名因该是一致的。所以我们可以通过判断 URLs 中是否包含站点 host 就可以了。

下面的代码是 host 为.com 的情况。其他情况的代码可以类推。

```
// host域名为.com
if (urlQueueManager.initURL.contains(".com")) {
    String str = urlQueueManager.initURL.substring(0,
            urlQueueManager.initURL.indexOf(".com"));
    if (url.contains(str.substring(str.lastIndexOf('.')) + ".com")) {

        return true;
    }
}
```

如果为站内URL则加入到缓存队列。

5.4 URL 队列管理

5.4.1 URL 消重处理

URL 消重处理，我是用 LRU 算法加上 MD5 压缩算法来实现的。因为 URLs 的数量非常巨大，为了节省内存空间。我先通过 MD5 压缩来得到每个 URL 对于的一个 hash 码。这个 hash 码的长度相对较小，可以节省内存开销。而且产生碰撞的几率非常小，可以忽略不计。

然后这个 URL 消重队列的维护是同时 LRULinkedHashMap 来实现的。这个 Map 非常好，它总是淘汰重复次数最少的 URL。这样就算 URL 数过大，也可

以尽量避免重复下载 URL。它的具体构造如下：

```
public class LRULinkedHashMap<K, V> extends LinkedHashMap<K, V> {

    private static final long serialVersionUID = 1L;

    private final int maxCapacity;

    private static final float DEFAULT_LOAD_FACTOR = 0.75f;

    private final Lock lock = new ReentrantLock();

    public LRULinkedHashMap(int maxCapacity) {
        super(maxCapacity, DEFAULT_LOAD_FACTOR, true);
        this.maxCapacity = maxCapacity;
    }

    @Override
    protected boolean removeEldestEntry(java.util.Map.Entry<K, V>
eldest) {
        return size() > maxCapacity;
    }

    @Override
    public V get(Object key) {
        try {
            lock.lock();
            return super.get(key);
        } finally {
            lock.unlock();
        }
    }

    @Override
    public V put(K key, V value) {
        try {
            lock.lock();
            return super.put(key, value);
        } finally {
            lock.unlock();
        }
    }
}
```

有了这个 map 以后，我就会用 URL 到里面去碰撞。因为有些网站的 URL

写法不固定。也许是同一个 URL，但是有些在最后一位接续"/"符号，而有些则没有接续。这样当进行 URL 去重处理时，会认为它们不是一个 URL。所以我们必须先除去后面的"/"符号，再进行 URL 去重判断。

```
    public synchronized boolean isContainUrl(String url) {

        if (url.endsWith("/")) {
            url = url.substring(0, url.length() - 1);
        }

        boolean b =
lRULinkedHashMap.containsKey(Url_Md5.MD5Encode(url));
        lRULinkedHashMap.put(Url_Md5.MD5Encode(url), true);

        return b;

    }
```

5.4.2 URL 等待队列维护

对 URL 等待队列的操作主要有 2 个：从里面取出 URLs;往里面加入 URLs。

但是因为 URL 等待队列会非常巨大，所以我将 URL 等待队列设计成 3 段式。2 段基于内存的 URL 缓存区，和一个基于数据库的储存区。

所以这里就会有 2 个方法来完成数据直接的交接。当加入 URL 缓存太长时，调用下面的方法，将缓存区的内容加入到数据库。具体的实现方法是，当存入缓存超过一定数目的时候。调用 waitQueueDecrease()这个函数，将存入缓存里的数据转移到数据库。方法是取出下标为 0 的元素，将其加入到数据库，然后删除下标为 0 的元素。不断重复这个操作，直到存入缓存被清空。

下面是具体实现的 JAVA 代码。

```
public synchronized void waitQueueDecrease() {
        try {
            Statement stmt = null;
            while (waitQueueFron.size() > 0) {
                try {
                    stmt = DateBaseConnect.conn().createStatement();
                    while (waitQueueFron.get(0).size() > 0) {
                        String url = (String)waitQueueFron.get(0).get(0);
                        waitQueueFron.get(0).remove(0);
                        stmt.execute("insert into middlequeue (url) values('"
                                + url + "');");
                        stmt.execute("insert into databasequeue (url) values('"
```

```
                + url + "');");
        }
```

当取出缓存区空以后，需要将数据库的内容加入到缓存区。通过调用waitQueueAdd()这个函数来实现。具体的实现方法是：从数据库里搜索前 25 条数据，因为数据库添加数据时是顺序往下压入的。对于 MYSQL 数据库，可以使用 LIMIT 这个关键字。检索存入数据库最上端的 25 条数据，然后依次将其加入到取出缓存区。然后删除数据库中这 25 条数据。

下面是具体实现的 JAVA 代码：

```
public synchronized void waitQueueAdd() {

        String sql = "SELECT * FROM middlequeue LIMIT 25;";
        Statement stmt = null;
        ResultSet res = null;
        try {
            stmt = DateBaseConnect.conn().createStatement();
            res = stmt.executeQuery(sql);
            while (res.next()) {
                waitQueueBack.add(res.getString("url"));
            }
            stmt.execute("delete from middlequeue limit 25");
```

5.4.3 数据库设计

对于 MYSQL 数据库的设计。我建立了 2 个表，分别为 middlequeue 和 databasequeue。middlequeue 表和 databasequeue 表都只有一个字段 URL，同时它作为主键，因为存入的 URL 不可能重复。Middlequeue 用于 URL 等待队列的主存储部分。而 databasequeue 表记录爬虫程序运行得到的所有 URLs。

JAVA 程序和数据之间的通信是通过 JDBC 实现的。下面是 JAVA 程序连接 MYSQL 数据库的代码。

```
public static Connection conn() {
    Connection conn = null;
    try {

        // 加载Connector/J驱动
        Class.forName("com.mysql.jdbc.Driver").newInstance();
        // 建立到MySQL的连接
```

```
            conn = DriverManager.getConnection(
                    "jdbc:mysql://localhost:3306/mysql", "root",
"root");
        } catch (Exception ex) {
            System.out.println("Error : " + ex.toString());
        }

        return conn;
    }
```

第六章 系统测试

我以 http://www.csdn.net/为初始 URL。然后用爬虫程序去运行。5 分钟内总共爬行出了 2201 个 URL。下面是从数据库里截图的 2 段爬行结果。

图 6.1 部分运行结果

图 6.1 为最先爬虫出来的 URLs 结果。我分析 http://www.csdn.net/的 HTML 代码。

最先检索出来 href 如下：

```
<base target="_blank" />
<link href="images/favicon.ico" rel="SHORTCUT ICON" />
```

这是一个相对地址，对应图 6.1 的爬虫结果。程序将其转化为了，绝对地址。但分析这个 URL 可以得知。这个 URL 为 ico 文件，是不可能被当作 HTML 解析的。所以这种 URL 因该过滤掉。这是程序设计上的不足。

然后被检索出来的 href 标签是：

```
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7" />
<link rel="stylesheet" href="http://csdnimg.cn/www/css/main.css" type="text/css"
```

这是一段完整 URL，但是其 host 名同初始 URL 不一致，同时它的后缀为 css，表明它指向的文件为 css 文件。所以因该被过滤掉。对应图 6.1 的爬虫结果，可以看到这段 URL 的确被过滤了。

第三个被检索到的 href 标签为：

```
<cite id="login"><a href="http://passport.csdn.net/CSDNUserRegister.aspx"
```

这是一段很标准的 URL,它的 host 为 csdn.net 并且可以得到 HTML 代码。所以这个 URL 被加入 URL 队列。根据测试结果可以得知爬虫程序能够正确解析出页面 URLs。

我在 SpiderWorker 的 run 方法写入这样一段代码：

```
System.out.println("线程"+ biaoji+"运行");
```

图 6.2

图 6.2 为控制台打印出来的信息。根据显示结果可以看出，不同的线程的确是在交替完成爬行任务。

第七章 结论

从课题着手到现在论文完成，经历了 3 个月的时间。在这个 3 个月里，我不断学习，探索，从对网络爬虫一无所知，到能成功编写出网络爬虫程序。对网络爬虫中比较主流的技术都有了更深的理解。网络爬虫是一个自动搜集互联网信息的工具，实现了网络爬虫，就可以在互联网这样一个巨大的信息海洋里遨游。

这篇论文实现了一个基于广度优先策略的多线程爬虫程序，可以搜集站内 URLs。但是在功能细节上还有很多不足，比如系统不够优化，功能不够强大，没有解析网页信息。对于网络爬虫这个庞大的知识体系来说，这篇论文实现的功能只能算一些皮毛。要深刻地理解爬虫程序，在爬虫程序设计中有所作为，还需要长达几年，或者更长时间的积累。所以在以后的时间，我将继续研究网络爬虫技术。分析设计一些比较复杂的爬行策略，优化爬虫程序。希望在这一课题上达到另一个高度。

参考文献

1.刘世涛等，简析搜索引擎中网络爬虫的搜索策略[N]，阜阳师范学院学报(自然科学版)，2006(09)：P60～63.

2.吴小竹等，基于 JAVA 的多线程 SPIDER 的设计与实现[J]，福建电脑，2004(06)：P62～63.

3.胡宏涛等，基于网络的信息获取技术浅析[J]，福建电脑，2006(04)：P60～61.

4.李学勇等，网络蜘蛛搜索策略比较研究[A]，计算机工程与应用，2004(04)：P131.

5.高克宁等，支持 Web 信息分类的高性能蜘蛛程序[J]，小型微型计算机系统，2006（07）：P1309～1312.

6.吴强等，JAVA 线程[A]，JAVASE 平台程序设计和实战，2004（05）：P103～135.

7.荣传湘等，搜索引擎中数据获取的设计与实现[J] 沈阳：小型微型计算机系统，1999.

8.Michelangelo Diligenti, Frans Coetzee, Steve Lawrence, etal. Focused Crawling using Context Graph s[J], Intemat ional Conference on Very Large Databases. 2002, (26) : P 527～534.

9.叶允明等. 分布式 Web Crawler 的研究: 结构、算法和策略[J]. 电子学报, 2002, 30 (12A).

[

致谢

我要感谢毕业论文指导老师，朱大勇老师，感谢他在论文方向和论文实施计划上给予我指导。感谢李广镇同学，戴国强同学，潘秀银同学在程序具体实施过程中给我提供了宝贵的意见和提议。

外文资料原文

Efficient URL Caching for World Wide Web Crawling

Andrei Z. Broder

IBM TJ Watson Research Center

19 Skyline Dr

Hawthorne, NY 10532

abroder@us.ibm.com

Marc Najork

Microsoft Research

1065 La Avenida

Mountain View, CA 94043

najork@microsoft.com

Janet L. Wiener

Hewlett Packard Labs

1501 Page Mill Road

Palo Alto, CA 94304

janet.wiener@hp.com

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.

译文

基于网络爬虫的有效 URL 缓存

概要：

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。

5. 模拟结果

我们论据 caching 的效率用关于 2 个 URLs 的流

1.一个踪迹所以的解析出来的 URLs 分配给一个特殊的机器。我们叫这个完全的踪迹。

2. 一个踪迹所以的解析出来的 URLs 分配给一个特殊的机器，然后送到另外一个机器里被执行。我们叫这个为交叉的子踪迹，因为他是完全踪迹的一个子集。

发现这两个选择的原因是，依赖其他构造的决定，他将 cache 只有那些被送

到其他机器的 URLs 或者用一个单独的 cache。

我们执行上面提到的每个 caching 算法，设定了一个很宽范围的 cache 大小。我们完成了大概 1800 个这样的实验。我们先描述我们的算法实现，再展示我们的模拟结果。

5.1 算法实现

每个算法的实现都是直截了当的。我们用一个 hash table 来找出 cache 中的每个项目。我们同时也保留一个 cache 项目的独立的数据结构，所以我们可以选择一个来淘汰。

对于 RANDOM，这个数据结构就是一个 list.对于 CLOCK，是一个 list 和一个 handle，这些项目同样包含 标记 bits。关于 LRU，是一个堆，用最后的进入时间来组织。STATIC

不需要格外的数据结构，因为它重来不淘汰项目。MIN 比较复杂，因为对于 cache 中的每个项目，MIN 需要知道它会不会是下一个请求。所以我们更详细地描述一下 MIN。

A 作为请求的踪迹或者顺序，也就是，At 是那个在时间 t 时被请求的项目。我们再用一个序列包含 A 中 At 下一个出现的时间。如果在时间 t 之后 At 没有进一步的请求，Nt=∞，

$$N_t = \begin{cases} \min\{s \mid s > t \wedge A_s = A_t\}, & \text{if } \exists s > t \text{ s.t. } A_s = A_t \\ \infty, & \text{otherwise.} \end{cases}$$

为了发现 Nt 序列，我们逆向读了踪迹 A，从 t max 到 0，用一个 hash table 键 At 和 值 t。对每个项目 At，我们探明那个 hash table。如果没有发现，Nt=∞，然后储存（At，t）到 table 里。如果被发现了，我们找（At，t’）， Nt=t’，然后替换（At，t’）成(At,t)。

给定 Nt，执行 MIN 就简单了：我们同时读 At 和 Nt，然后从此对于每个请求的项目，我们知道什么时候它将会被请求。我们用它将会被请求的时间标记每个项目，如果可能，淘汰那些下一次氢气有很高值的项目，用一个堆来识别它很快。

5.2 结论

我们只介绍一个爬虫主机的结论。其他 3 个主机的结论可以以此类推。

Figure2 显示了丢失率在一个完全的踪迹（这个是，cache 中整个请求中丢失的百分比）。我们观察 cache 大小从 k=（2 的零次方）到 k=（2 的 25 次方）。Figure3 中我们展现了同样的数据关于 MIN 的丢失率。Figure4 和 5 描述了一样的模拟关于 cross—trace。

对于踪迹，LRU 和 CLOCK 运行几乎一样，只是稍微比理想的 MIN 要差一些，除了下面那些临界的区域。RANDOM 只是略微的劣于 CLOCK 和 LRU，当然 STATIC 通常要更差些。

对于非常大的 cache，STATIC 似乎要好于 MIN。但是，这只是一个人造物品对于我们的计数策略：我们只是主管丢失而且 STATICS 没有主管 cache 中的初始装载。

6. 总结和未来发展趋势

进过运行大约 1800 次模拟进过一个踪迹包括 268.6 亿的 URLs，我们的主要结论是 URL caching 非常地有效-在我们的机制里，一个包含大约 5000 个入口的 cache 可以达到大约 80%的撞击率。有趣的是，这个大小是一个临界点，就是说，一个很小的 cache 效果很差，一个更大的 cache 不能增加性能。对实际目的我们的调查已经完成：在 5.2 部分中我们的讨论，我们分给每个爬虫现场一个 cache 大小在 100 至 500 个入口。所有的 caching 策略几乎一样；我们推荐用 CLOCK 或者 RANDOM，通过一个环形链的分散 table 来实现。因此，对于 500 个爬虫线程，这个 cache 将会是一个大约 2MB 的完全的可忽略地比喻成爬虫程序需要的其他的数据结构。如果绝对只是去减少分布式爬虫之间的交叉机器通信，那么就可以用一个稍微小一点 cache。对于任何一种情况，目标是丢失率小于 20%。

但是，有很多容易讨论的问题，值得进一步研究。第一个容易解决的问题，什么程度爬虫次序策论影响 caching 的实现。各种策略被提议，但是有迹象从爬虫开始后一段时间，一般的策略不关键。因此，我们相信 caching 实现很相像，对于任何的爬行策略。我们可以尝试去实现我们自己的不同的策略，但是理想的我们将实现独立的爬虫。不幸的，在网络上爬行不是一件简单的努力，这个不像我们获得商业搜索引起的日志文件。

由于观察到的现象，达到最好的性能的 cache 的大小由线程数决定，第二个问题是是否每个线程的 cache 都有意义。通常，但不是总是，一个总的 cache 比几个分开的 cache 的集合要好，因为公共的项目只需要被储存一次。但是这个认为，需要在 URL caching 上下文中检测。

第三个容易被讨论的问题，关于我们的目的在第 5 部分关于 host 中给定的 URL 的范围。如果我们的模式正确，那么它将确定意味关于适合的模式对于 web 图，一个话题被不同的科学家关心：数学家，物理学家，计算机科学家。我们希望我们的论文可以激发对不同模式的 cache 性能的研究。那些执行的好的模式通过一个给定主机里的相关链接更接近于现实。我们让我们的 URL 踪迹更适合于这个研究通过赠送它给网络存档。