第一章：WEB 搜索引擎介绍 3

1.1 搜索引擎的分类 3
1.2 搜索引擎的工作原理 4
1.3 WEB 搜索引擎的体系结构 5

第二章 WEB 搜索引擎中信息的搜集 6

2.1 搜集的时机 6
2.2 网络爬虫介绍 6
2.2.1 搜集的策略 7
2.2.2 信息指纹在爬虫中的应用 7
2.3 网页的维护与更新 7
2.4 爬虫程序设计中要注意的问题 8
2.5 网站与网络爬虫 8
2.6 两款爬虫使用说明 9
2.6.1 WebLech--http://weblech.sourceforge.net/ 9
2.6.2 Nutch 中爬虫 10

第三章 网页预处理 13

3.1. 网页噪音概述 13
3.1.1 网页噪音的概念和分类 13
3.1.2 相关研究 14
3.2. 一种实用的网页去噪方法 15
3.2.1 HTML 基本知识 15
3.2.2 HTMLParser 类 23
3.2.3 网页的分类 27
3.2.4 算法思想 27
3.2.5 算法实现依据的规则 28
3.2.6 算法流程 28
3.2.7 去噪效果演示 31
3.2.8 去噪方法的优点 32
3.2.9 去噪方法的缺点 32
3.3. 网页去噪的优点和应用 33
3.4. 本章小结 33

第四章 WEB 搜索引擎中信息的索引 34

1．概述 34

2．LUCENE 介绍 34

2．1 什么是 LUCENE？ 34
2．2 LUCENE 的应用、特点及优势 34

3．LUCENE 系统结构 35

4 LUCENE 索引 36

4．1 LUCENE 索引原理 36

4．2 LUCENE 索引文件格式 37
5 LUCENE 中索引使用 41
5．1 建立索引的一个简单例子 41
5.2 理解索引中的核心类 45
5.2.1 IndexWriter 46
5.2.2 Directory 46
5.2.3 Analyzer 46
5.2.4 Document 46
5.2.5 Field 47
5．3 LUCENE 中索引操作 48
5.3.1 理解索引过程 48
5.3.2 基本索引操作 50
5.3.2.9 为查询优化索引(index) 59
5.3.2.12 索引同步和 locking 机制 59
第五章 字符分析器 61
5.1 LUCENE 分析器——ANALYZER 62
5.1.1 Analyzer 的概述 62
5.1.2 分词器（Tokenizer）和过滤器（TokenFilter） 63
5.1.2 使用 StandardAnalyzer 进行测试 64
5.2 JAVACC 与 TOKENIZER 68
5.2.1 JavaCC 简介 68
5.2.2 通过 JavaCC 构建的 Lucene 标准分析器 69
5.3 LUCENE 分析器——ANALYZER 73
5.3.1 标准分析器——StandardAnalyzer 73
5.3.2 “停止词”分析器——StopAnalyzer 78
5.3.2 其他分析器 80
5.4 总结 82
第六章 WEB 搜索引擎中信息的查询服务 82
6.1 查询服务的系统结构及工作原理 82
6.1.1 查询服务的系统结构 82
6.1.2 查询服务系统的工作原理 83
6.1.3 查询服务系统的具体实现 85
6.2 LUCEUE 中的检索 87
6.2.1 Luceue 的检索流程 88
6.2.2 检索与结果 88
6.2.3 构建各种 Query 95
6.2.4 查询字符串的解析——QueryParser 类 102
6.3 LUCEUE 中的高级检索技巧 103
6.3.1 对检索结果的排序 103
6.3.2 多域检索和多索引检索 111
6.3.3 对检索结果的过滤 113
第七章：WEB 搜索引擎中结果排序 115

7.1 LUCENE 的评分机制 115
7.1.1 理解评分的概念 *116*
7.1.2 Lucene 中的评分机制 *116*
7.1.3 Lucene 排序算法说明 *116*
7.2 链接分析及其应用 117
7.2.1 基于随机冲浪模型的 PageRank *119*
7.2.2 HITS 算法 122
7.2.3 SASLA 算法介绍 *123*
7.3 PageRank、HITS、SALSA 算法比较分析 *124*
7. 3 链接分析在 SEWM 评测中的应用 125
7.3.1 SEWM 评测介绍 *125*
7.3.2 Hits 算法在主题提起中的应用 *126*
7.4 链接分析方法局限性及其发展前景 129

第一章：Web 搜索引擎介绍

随着 Internet 的迅速发展，Web（World Wide Web 万维网）网络给人们提供大量的信息资源，给我们的生活、学习等带来了极大的便利。但是同时各种信息在网络上的爆炸式增长，也常常使我们淹没在信息的汪洋中。那么我们在日常生活中都是如何寻找信息的呢，相信我们大家都会使用到 WEB 搜索引擎。本章将从搜索引擎的分类、基本工作原理以及体系结构三个方面做一个简单介绍。

1.1 搜索引擎的分类

获得网站网页资料，能够建立数据库并提供查询的系统，我们都可以把它叫做搜索引擎。按照工作原理的不同，可以把它们分为两个基本类别：全文搜索引擎（FullText Search Engine）和分类目录 Directory）。

全文搜索引擎的数据库是依靠一个叫“网络机器人（Spider）”或叫“网络蜘蛛（crawlers）”的软件，通过网络上的各种链接自动获取大量网页信息内容，并按以定的规则分析整理形成的。Google、百度都是比较典型的全文搜索引擎系统。

分类目录则是通过人工的方式收集整理网站资料形成数据库的，比如雅虎中国以及国内的搜狐、新浪、网易分类目录。另外，在网上的一些导航站点，也可以归属为原始的分类目录，比如“网址之家”。

全文搜索引擎和分类目录在使用上各有长短。全文搜索引擎因为依靠软件进行，所以数据库的容量非常庞大，但是，它的查询结果往往不够准确；分类目录依靠人工收集和整理网站，能够提供更为准确的查询结果，但收集的内容却非常有限。为了取长补短，现在的很多搜索引擎，都同时提供这两类查询，一般对全文搜索引擎的 查询称为搜索“所有网站”或“全部网站”，比如 Google 的全文搜索（http://www.google.com/intl/zh-CN/）；把对分类目录的查询称为搜索“分类目录”或搜索“分类网站”，比如新浪搜索和雅虎中国搜索（http://cn.search.yahoo.com/dirsrch/）。

在网上，对这两类搜索引擎进行整合，还产生了其它的搜索服务，在这里，我们权且也把它们称作搜索引擎，主要有这两类：

1.元搜索引擎(META Search Engine)。这类搜索引擎一般都没有自己网络机器人及数据库，它们的搜索结果是通过调用、控制和优化其它多个独立搜索引擎的搜索结果并以统一的格式在 同一界面集中显示。元搜索引擎虽没有“网络机器人”或“网络蜘蛛”，也无独立的索引数据库，但在检索请求提交、检索接口代理和检索结果显示等方面，均有自 己研发的特色元搜索技术。比如“metaFisher 元搜索引擎” （http://www.hsfz.net/fish/），它就调用和整合了 Google、Yahoo、AlltheWeb、百度和 OpenFind 等多家搜索引擎的数据。

2.集成搜索引擎（All－in－One Search Page）。集成搜索引擎是通过网络技术，在一个网页上链接很多个独立搜索引擎，查询时，点选或指定搜索引擎，一次输入，多个搜索引擎同时查询，搜索结果由各搜索引擎分别以不同页面显示，比如“网际瑞士军刀”（http://free.okey.net/%7Efree/search1.htm）。

1.2 搜索引擎的工作原理

全文搜索引擎的“网络机器人”或“网络爬虫”是一种网络上的软件，它遍历 Web 空间，能够扫描一定 IP 地址范围内的网站，并沿着网络上的链接从一个网页到另一个网页，从一个网站到另一个网站采集网页资料。它为保证采集的资料最新，还会回访已抓取过的网页。网络机器人或网络蜘蛛采集的网页，还要有其它程序进 行分析，根据一定的相关度算法进行大量的计算建立网页索引，才能添加到索引数据库中。我们平时看到的全文搜索引擎，实际上只是一个搜索引擎系统的检索界面，当你输入关键词进行查询时，搜索引擎会从庞大的数据库中找到符合该关键词的所有相关网页的索引，并按一定的排名规则呈现给我们。不同的搜索引擎，网页索引数据库不同，排名规则也不尽相同，所以，当我们以同一关键词用不同的搜索引擎查询时，搜索结果也就不尽相同。

和全文搜索引擎一样，分类目录的整个工作过程也同样分为收集信息、分析信息和查询信息三部分，只不过分类目录的收集、分析信息两部分主要依靠人工完成。分类目录一般都有专门的编辑人员，负责收集网站的信息。随着收录站点的增多，现在一般都是由站点管理者递交自己的网站信息给分类目录，然后由分类目录的编辑人员审核递交的网站，以决定是否收录该站点。如果该站点审核通过，分类目录的编辑人员还需要分析该站点的内容，并将该站点放在相应的类别和目录中。所有这 些收录的站点同样被存放在一个“索引数据库”中。用户在查询信息时，可以选择按照关键词搜索，也可按分类目录逐层查找。如以关键词搜索，返回的结果跟全文 搜索引擎一样，也是根据信息关联程度排列网站。需要注意的是，分类目录的关键词查询只能在网站的名称、网址、简介等内容中进行，它的查询结果也只是被收录网站首页的 URL 地址，而不是具体的页面。分类目录就像一个电话号码薄一样，按照各个网站的性质，把其网址分门别类排在一起，大类下面套着小类，一直到各 个网站的详细地址，一般还会提供各个网站的内容简介，用户不使用关键词也可进行查询，只要找到相关目录，就完全可以找到相关的网站（注意：是相关的网站，而不是这个网站上某个网页的内容，某一目录中网站的排名一般是按照标题字母的先后顺序或者收录的时间顺序决定的）。

搜索引擎并不真正搜索互联网，它搜索的实际上是预先整理好的网页索引数据。真正意义上的搜索引擎，通常指的是收集了因特网上几千万到几十亿个网页并对网页中的每一个词（即关键词）进行索引，建立索引数据库的全文搜索引擎。当用户 查找某个关键词的时候，所有在页面内容中包含了该关键词的网页都将作为搜索结果被搜出来。在经过复杂的算法进行排序后，这些结果将按照与搜索关键词的相关 度高低，依次排列。

现在的搜索引擎已普遍使用超链分析技术，除了分析索引网页本身的内容，还分析索引所有指向该网页的链接的 URL、AnchorText、甚至链接周围的文字。所以，有时候，即使某个网页 A 中并没有某个词比如“恶魔撒旦”，但如果有别的网页 B 用链接“恶魔撒旦”指向这个网页 A，那么用户搜索“恶魔撒旦”时也 能找到网页 A。而且，如果有越多网页（C、D、E、F……）用名为“恶魔撒旦”的链接指向这个网页 A，或者给出这个链接的源网页（B、C、D、E、 F……）越优秀，那么网页 A 在用户搜索“恶魔撒旦”时也会被认为更相关，排序也会越靠前。

搜索引擎的原理，可以看做三步：从互联网上抓取网页→建立索引数据库→在索引数据库中搜索排序。

从互联网上抓取网页

利用能够从互联网上自动收集网页的 Spider 系统程序，自动访问互联网，并沿着任何网页中的所有 URL 爬到其它网页，重复这过程，并把爬过的所有网页收集回来。

建立索引数据库

由分析索引系统程序对收集回来的网页进行分析，提取相关网页信息（包括网页所在 URL、编码类型、页面内容包含的关键词、关键词位置、生成时间、 大小、与其它网页的链接关系等），根据一定的相关度算法进行大量复杂计算，得到每一个网页针对页面内容中及超链中每一个关键词的相关度（或重要性），然后 用这些相关信息建立网页索引数据库。

在索引数据库中搜索排序

当用户输入关键词搜索后，由搜索系统程序从网页索引数据库中找到符合该关键词的所有相关网页。因为所有相关网页针对该关键词的相关度早已算好，所以只需按照现成的相关度数值排序，相关度越高，排名越靠前。

接受查询

用户向搜索引擎发出查询，搜索引擎接受查询并向用户返回资料。搜索引擎每时每刻都要接到来自大量用户的 几乎是同时发出的查询，它按照每个用户的要求检查自己的索引，在极短时间内找到用户需要的资料，并返回给用户。目前，搜索引擎返回主要是以网页链接的形式提供的，这些通过这些链接，用户便能到达含有自己所需资料的网页。通常搜索引擎会在这些链接下提供一小段来自这些网页的摘要信息以帮助用户判断此网页是否含有自己需要的内容

搜索引擎的 Spider 一般要定期重新访问所有网页（各搜索引擎的周期不同，可能是几天、几周或几月，也可能对不同重要性的网页有不同的更新频率），更新网页索引数据库，以反映出网页内容的更新情况，增加新的网页信息，去除死链接，并根据网页内容和链接关系的变化重新排序。这样，网页的具体内容和变化情况 就会反映到用户查询的结果中。

互联网虽然只有一个，但各搜索引擎的能力和偏好不同，所以抓取的网页各不相同，排序算法也各不相同。大型搜索引擎的数据库储存了互联网上几亿至几十亿的网页索引，数据量达到几千 G 甚至几万 G。但即使最大的搜索引擎建立超过二十亿网页的索引数据库，也只能占到互联网上普通网页的不到 30%，不同搜索引擎之间的网页数据重叠率一般在 70%以下。我们使用不同搜索引擎的重要原因，就是因为它们能分别搜索到不同的内容。而互联网上有更大量的内容，是搜索引擎无法抓 取索引的，也是我们无法用搜索引擎搜索到的。

我们心里应该有这样一个概念：搜索引擎只能搜到它网页索引数据库里储存的内容。

1.3 WEB 搜索引擎的体系结构

通过上节对搜索引擎一般工作原理的介绍，相信在我们心中可以大致勾勒出搜索的体系结构。如图 1.1 所示，

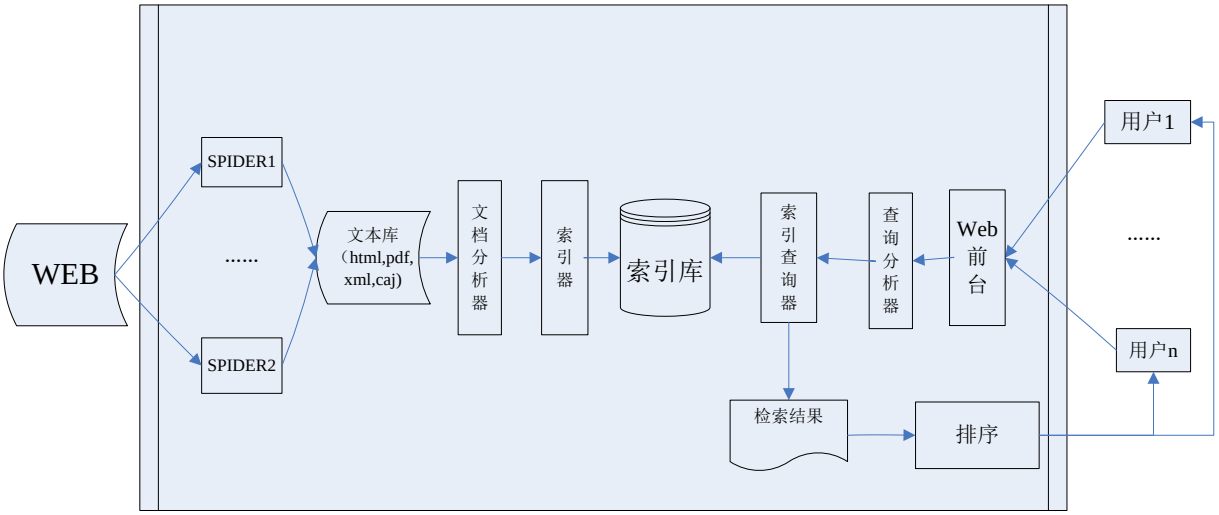


图 1.1 系统结构图

由图 1.1 我们可以看到，系统中大部分模块和前面描述的可以直接对应。但改图只描述在单机上如何搜集信息、提供服务，实际应用中一般都是多台计算机同时来做这项工作，比单机上要复杂得多。

第二章 Web 搜索引擎中信息的搜集

2.1 搜集的时机

一般来说，搜索引擎这样一个软件系统都是事先收集好网页，存储在本地硬盘上，然后再进行预处理、索引。可以想象如果等用户查询时再爬取成千上万的网页，再一个个分析处理，是不可能满足搜索引擎的响应时间要求的。[1] 一般搜索引擎都是使用一种叫网络爬虫（也叫网络爬虫和网络机器人）的程序来完成这项工作，下面我们讲介绍这样一个程序的原理以及其实现。

2.2 网络爬虫介绍

互联网其实就是一张大图，我们可以把每一个网页当作一个节点，把那些超链接（Hyperlinks)当作连接网页的弧。网页中那些蓝色的、带有下划线的文字背后其实藏着对应的网址，当你点下去的的时候，浏览器是通过这些隐含的网址转到相应的网页中的。这些隐含在文字背后的网址称为“超链接”。有了超链接，我们可以从任何一个网页出发，用图的遍历算法，自动地访问到每一个网页并把它们存起来。完成这个功能的程序叫做网络爬虫，或者在一些文献中称为"机器人"（Robot)。世界上第一个网络爬虫是由麻省理工学院 (MIT)的学生马休.格雷（Matthew Gray)在 1993 年写成的。他给他的程序起了个名字叫“互联网漫游者”("www wanderer")。以后的网络爬虫越写越复杂，但原理是一样的。

那么网络爬虫如何下载整个互联网。假定我们从一家门户网站的首页出发，先下载这个网页，然后通过分析这个网页，可以找到藏在它里面的所有超链接，也就等于知道了这家门户网站首页所直接连接的全部网页，诸如雅虎邮件、雅虎财经、雅虎新闻等等。我们接下来访问、下载并分析这家门户网站的邮件等网页，又能找到其他相连的网页。我们让计算机不停地做下去，就能下载整个的互联网。

2.2.1 搜集的策略

最长见一种方式就是所谓的“爬行”。我们在遍历互联网这张大图时，既可以从某个网页节点（或节点集）开始深度优先遍历，也可以广度优先遍历。由图论知识我们可以知道， 遍历时必须记录下已经访问过的节点（避免重复），在网络爬虫中，我们使用一个称为“哈希表”(Hash Table)的列表而不是一个记事本纪录网页是否下载过的信息。下面我们要介绍的爬虫 WebLech 就是采取这一策略。

由于不可能抓取所有的网页，有些网络爬虫对一些不太重要的网站，设置了访问的层数。这也让有些网站上一部分网页能够在搜索引擎上搜索到，另外一部分不能被搜索到。对于网站设计者来说，扁平化的网站结构设计有助于搜索引擎抓取其更多的网页。

另外一种可行的方式是使用最佳优先遍历循环的收集。首先我们应该有一个初始的种子 URL 集合，爬虫爬取这些种子 URL 后，解析出这些网页中的包括的 URL，并加入到系统维护的一个 URL 集合 S 中，然后再对 S 中的链接进行分析（去重，判断起重要程度，标记其爬行周期等），再用某种算法优先生成一个待爬行的网页集合 D，再重复这一过程，循环爬取网页。下面我们要介绍的 Nutch 爬虫就是采取这一策略。

最后还有一种方式就是让网站拥有者主动向搜索引擎提交它们的网址，系统在一定时间内定向去爬取这些网站。一般来说该种策略都是与以上两种方法配合使用，目前常见的是广度优先和最佳优先方法。

2.2.2 信息指纹在爬虫中的应用

在哈希表中以字符串的形式直接存储网址，既费内存空间，又浪费查找时间。现在的网址一般都较长，比如，如果在 Google 或者百度在查找数学之美，对应的网址长度在一百个字符以上。下面是百度的链接 http://www.baidu.com/s?ie=gb2312&bs=%CA%FD%D1%A7%D6%AE%C3%C0&sr=&z=&cl=3&f=8&wd=%CE%E2%BE%FC+%CA%FD%D1%A7%D6%AE%C3%C0&ct=0

假定网址的平均长度为一百个字符，那么存贮 200 亿个网址本身至少需要 2 TB，即两千 GB 的容量，考虑到哈希表的存储效率一般只有 50%，实际需要的内存在 4 TB 以上。即使把这些网址放到了计算机的内存中，由于网址长度不固定，以字符串的形式查找的效率会很低。因此，我们如果能够找到一个函数，将这 200 亿个网址随机地映射到 128 二进位即 16 个字节的整数空间，比如将上面那个很长的字符串对应成一个如下的随机数：89324943298439843298054545454 3，这样每个网址只需要占用 16 个字节而不是原来的一百个。这就能把存储网址的内存需求量降低到原来的 1/6。这个 16 个字节的随机数，就称做该网址的信息指纹（Fingerprint)。可以证明，只要产生随机数的算法足够好，可以保证几乎不可能有两个字符串的指纹相同，就如同不可能有两个人的指纹相同一样。由于指纹是固定的 128 位整数，因此查找的计算量比字符串比较小得多。网络爬虫在下载网页时，它将访问过的网页的网址都变成一个个信息指纹，存到哈希表中，每当遇到一个新网址时，计算机就计算出它的指纹，然后比较该指纹是否已经在哈希表中，来决定是否下载这个网页。这种整数的查找比原来字符串查找,可以快几倍到几十倍。[2]

2.3 网页的维护与更新

1)、批量搜集

批量搜集，就是每次搜集替换上一次的内容。由于每次都是重新来一次，对度大规模搜索引擎来说，搜集的时间通常会花几周。这样做的好处是系统实现比较简单，主要缺点是“时新性”不高，还有重复收集带来的带宽消耗。

2)、增量搜集

增量搜集，开始时搜集一批，往后只是：1、搜集新出现的网页；2、搜集在上次搜集后有改变的网页；3、删除上次搜集后不存在的网页

2.4 爬虫程序设计中要注意的问题

- 网站的速度
- DNS 缓存
- 不可到达的站点处理
- 网页的存储－网页的长期存储
- 搜集的效率－多进程或多线程，分布式搜集
- 避免重复收集
- 更新周期控制
- 搜集重要网页

2.5 网站与网络爬虫

网络爬虫需要抓取网页，不同于一般的访问，如果控制不好，则会引起网站服务器负担过重。2005 年 4 月，淘宝网（http://www.taobao.com）就因为雅虎搜索引擎的网络爬虫抓取其数据引起淘宝网服务器的不稳定。网站是否就无法和网络爬虫交流呢？其实不然，有多种方法可以让网站和网络爬虫进行交流。一方面让网站管理员了解网络爬虫都来自哪儿，做了些什么，另一方面也告诉网络爬虫哪些网页不应该抓取，哪些网页应该更新。

每个网络爬虫都有自 己的名字，在抓取网页的时候，都会向网站标明自己的身份。网络爬虫在抓取网页的时候会发送一个请求，这个请求中就有一个字段为 User－agent，用于 标识此网络爬虫的身份。例如 Google 网络爬虫的标识为 GoogleBot，Baidu 网络爬虫的标识为 BaiDuSpider，Yahoo 网络爬虫的 标识为 Inktomi Slurp。如果在网站上有访问日志记录，网站管理员就能知道，哪些搜索引擎的网络爬虫过来过，什么时候过来的，以及读了多少数据等等。如果网站管理员发 现某个爬虫有问题，就通过其标识来和其所有者联系。下面是博客中国(http://www.blogchina.com)2004 年 5 月 15 日的搜索引擎 访问日志：网络爬虫进入一个网站，一般会访问一个特殊的文本文件 Robots.txt，这个文件一般放在网站服务器的根目录下，如： http://www.blogchina.com/robots.txt。网站管理员可以通过 robots.txt 来定义哪些目录网络爬虫不能访问，或 者哪些目录对于某些特定的网络爬虫不能访问。例如有些网站的可执行文件目录和临时文件目录不希望被搜索引擎搜索到，那么网站管理员就可以把这些目录定义为拒绝访问目录。Robots.txt 语法很简单，例如如果对目录没有任何限制，可以用以下两行来描述：

User-agent: *

Disallow:

当然，Robots.txt 只是一个协议，如果网络爬虫的设计者不遵循这个协议，网站管理员也无法阻止网络爬虫对于某些页面的访问，但一般的网络爬虫都会遵循这些协议，而且网站管理员还可以通过其它方式来拒绝网络爬虫对某些网页的抓取。

网络爬虫在下载网页的时候，会去识别网页的 HTML 代码，在其代码的部分，会有 META 标识。通过这些标识，可以告诉网络爬虫本网页是否需要被抓取，还可以告诉网络爬虫本网页中的链接是否需要被继续跟踪。例如：表示本

网页不需要被抓取，但是网页内的链接需要被跟踪。

关于 Robots.txt 的语法和 META Tag 语法，有兴趣的读者查看文献[4] 现在一般的网站都希望搜索引擎能更全面的抓取自己网站的网页，因为这样可以让更多的访问者能通过搜索引擎找到此网站。为了让本网站的网页更全面被抓取到，网 站管理员可以建立一个网站地图，即 Site Map。许多网络爬虫会把 sitemap.htm 文件作为一个网站网页爬取的入口，网站管理员可以把网站内部所有网页的链接放在这个文件里面，那么网络蜘 蛛可以很方便的把整个网站抓取下来，避免遗漏某些网页，也会减小对网站服务器的负担。

2.6 两款爬虫使用说明

2.6.1 WebLech--http://weblech.sourceforge.net/

特点：

1)、开源，免费

2)、代码是用纯 Java 写的，可以在任何支持 Java 的平台上也行

3)、支持多线程下载网页

4)、可维持网页间的链接信息

5)、可配置性强：

深度优先或宽度优先爬行网页

可定制 URL 过滤器，这样就可以按需要爬行单个 web 服务器，单个目录或爬行整 个 WWW 网络

可设置 URL 的优先级，这样就可以优先爬行我们感兴趣或重要的网页

可记录断点时程序的状态，一边重新启动时可接着上次继续爬行。

使用方法：

1)、按需求修改配置文件 Spider.properties

saveRootDirectory = sites 设置文件的存放路径，默认为当前文件夹

mailtoLogFile = mailto.txt 设置邮件链接的存放文件

refreshHTMLs = true refreshImages = false refreshOthers = false //设置如果本地硬盘已经存在待爬取的文件，是否重新载入文件

htmlExtensions = htm,html,shtm,shtml,asp,jsp,php 设置 spider 要下载资源的扩张名，spider 也会学习新的下载类型

imageExtensions = 同上

startLocation = http://ir.dlut.edu.cn/ 设置 spider 爬行的起始页面

depthFirst = false 设置进行广度优先爬行或深度优先爬行

maxDepth = 5 爬行的最大深度（第一个页面深度为 0，其链接的深度为 1）

urlMatch ＝ 基本的 URL 过滤。下载的网页的网址中中必须包括 urlMatch 串

interestingURLs=pollBooth.pl,faq 设置感兴趣的 url

boringURLs=article.pl 设置不感学期的 url

basicAuthUser = myUser basicAuthPassword = 1234 设置需要验证的网站的用户名和密码

spiderThreads = 15 爬行的线程数

checkpointInterval = 30000 设置写断点的时间间隔（单位毫秒）

2)、运行 run.bat 开始爬行

3)、如果程序中断，运行 rusume.bat 继续爬行

2.6.2 Nutch 中爬虫

Nutch 是一款用 java 编写的开源搜索引擎软件，集爬虫、索引、检索和 Web 前台于一身，其官方下载地址为 http://lucene.apache.org/nutch/release/，在其官方 wiki 网站 http://wiki.apache.org/nutch/ 上有关于 Nutch 各方面的相关知识。在这我们下面主要 Nutch 中的爬虫部分的使用方法，配置好 Nutch 后，如果我们在其安装目录下输入命令：bin/nutch ，将会出现 Nutch 中所有的命令以及其简短说明。Nutch 中爬虫主要有两种使用方式，一种是使用 crawl 命令，该命令一般用于网内爬行，使用爬取较少 Web 服务器网页的应用；另一种是整个互联网内的爬行，适用于爬取数量较多的网页。下面我们介绍其使用方法。

2.6.2.1 网内爬行使用说明

如果我们只想在 Web 服务器上抓取百万级的网页，那么网内爬行将更合适。

网内爬行必须的配置

1、 新建一个包含种子 url 的目录。例如，如果我们想抓取大连理工大学信息检索实验室的网站，可以新建一个名为 urls/IRLab.txt 文件，该文件内容只需包括该网站的首页 url 就可以，当然了该网站的其他网页也应该能顺着该首页 url 访问到。urls/IRLab.txt 文件的内容为： http://ir.dlut.edu.cn/

2、 编辑 conf 目录下的 `crawl-urlfilter.txt`，并把该文件中的 `MY.DOMAIN.NAME` 替换为你想爬取网站的域名。例如，如果只想爬取 `ir.dlut.edu.cn` 站内，那么就应该改为： `+^http://ir.dlut.edu.cn/`
这个将包括网站 `ir.dlut.edu.cn` 下所有网页

3、 编辑 conf 目录下的 nutch-site.xml 文件，在该文件中配置以下属性适当的值。

```
<property>
  <name>http.agent.name</name>
  <value></value>
  <description>HTTP 'User-Agent' request 请求头. 不能为空 -
  请填上与你们组织相关一个词
  注: 同样也要检查其他相关属性：
        http.robots.agents
        http.agent.description
        http.agent.url
        http.agent.email
```

http.agent.version
并适当的设置它们的值
</description>
</property>
<property>
<name>http.agent.description</name>
<value></value>
<description>对我们爬虫的进一步描述--该文本用在 User-Agent request 请求头中
. 它将出现在 agent name 后面的括号中
</description>
</property>
<property>
<name>http.agent.url</name>
<value></value>
<description>刊登 User-Agent header 的一个 URL
. 它将出现在 agent name 后面的括号中
该 URL 所指定的页面用于阐明该爬虫的目的和行为。
</description>
</property>

<property>
<name>http.agent.email</name>
<value></value>
<description>放在'From' 请求头和 User-Agent 头中的邮件地址。实际应用中最好把邮件地址分开(e.g. 'info at example dot com')，以避免被发垃圾邮件
</description>
</property>

网内：运行爬虫

做好以上配置后，运行该爬虫就非常容易，只需要运行 crawl 命令就可以了。该命令的选项包括。

- -dir *dir*
- -threads *threads* 设置并行爬行的线程个数
- -depth *depth* 设置相对于根网页的爬行层次深度
- -topN *N* 设置每一层爬行的网页个数的最大值

例如，该命令的典型使用方式：

```
bin/nutch crawl urls -dir crawl -depth 3 -topN 50
```

通常在测试该命令时，一般把-depth 设置得比较小，也可以限制每层爬取的网页个数，这样可以很快看到结果。当我们对配置比较熟悉的时候，一个比较适当的深度值为 10。每层爬取的最大网页个数这个就要根据我们的资源而定。

整个 Web 域的爬取

整个 Web 域的爬取是为大规模爬取网页设计的，可能需要在多个机器几周才能完成。

相关概念

Nutch 中数据由以下几部分组成：

1、抓取数据库 crawl database,或者称 crawldb。该数据库中包含 Nutch 知道的所有 url，不管有没有被爬取过。

2、链接数据库 link database，或者称 linkdb。该数据库中包含每个 url 的链出列表，已经该 url 的链入 url 和锚文本。

3、段 *segment* 集合。每一个段 segment 都是要作为一个整体爬取的 url 集合。Segments 中有如下子目录:

4、

5、Nutch 中索引为 Lucene 格式的索引。

2.6.2.2 整个互联网 Web 爬行使用说明

Nutch 中的注射器程序 injector 可以向爬行数据库 crawldb 中添加 url。我们可以把 DMOZ 这个可开放目录中的 url 列表注入到 crawldb 中。其 url 列表地址为 http://rdf.dmoz.org/rdf/content.rdf.u8.gz （这个文件大小超过 200M，可能要花比较长的时间下载，如果只是测试，可以只下载其示例文件）。

下一步随机选择以上 url 列表的一个子集。DMOZ 的 url 列表中含有 3 百万 url，这里我们每 5000 个选一个出来，左后一共大约有 1000 个 url。其命令如下：

```
mkdir dmoz
bin/nutch org.apache.nutch.tools.DmozParser content.rdf.u8 -subset 5000 > dmoz/urls
bin/nutch inject crawl/crawldb dmoz
```

这样，我们的 url 数据库中就有了 1000 个未被爬取的 url。

爬取

在开始抓取前，需要配置 conf 目录下的 nutch-site.xml 文件，最少得插入以下属性，并对其适当赋值。参照以上网内爬行的配置。

首先，从 url 数据库中生成一个爬取列表

```
bin/nutch generate crawl/crawldb crawl/segments
```

以上命令将生成所有需要爬取的 url 列表，该爬取列表在新生成的段目录中，该段的名称为其生成的时间。我们把该段的名称保存在 shell 变量 s1 中。

```
s1=`ls -d crawl/segments/2* | tail -1`
echo $s1
```

现在我们运行 fetch 命令来爬取这个段的 url

```
bin/nutch fetch $s1
```

当以上命令结束以后，我们用爬取的结果来更新 url 数据库。

```
bin/nutch updatedb crawl/crawldb $s1
```

```
Now the database has entries for all of the pages referenced by the initial set.
```

现在我们把分数最高的 1000 个 url 生成一个新的段，以便爬取。

```
bin/nutch generate crawl/crawldb crawl/segments -topN 1000
s2=`ls -d crawl/segments/2* | tail -1`
echo $s2

bin/nutch fetch $s2
bin/nutch updatedb crawl/crawldb $s2
```

下面我们进行新的一轮的爬取。

```
bin/nutch generate crawl/crawldb crawl/segments -topN 1000
s3=`ls -d crawl/segments/2* | tail -1`
echo $s3
```

```
bin/nutch fetch $s3
bin/nutch updatedb crawl/crawldb $s3
```
循环以上步骤，我们在可以在互联网服务器上爬行到更多的网页。

第三章　网页预处理

随着 Internet 的迅猛发展，Web 网页上的信息呈现爆炸式的增长，它已然成为人们获取知识、信息的一个最重要的来源。然而，网页噪音却至少占了网页内容的一半以上。我们知道，网页噪音是任何网页都不可避免的部分，它也是影响阅读网页和进行 Web 信息处理的一个重要因素。所以网页去噪是必不可少的，同时它也是搭建搜索引擎过程中的一个重要环节。

本章首先介绍网页噪音的概念，然后着重介绍一种网页去噪的方法，通过该方法，我们可以从一个网页源文件中去除与主题无关的内容，提取出网页的一些主要内容，这些主要内容包括网页标题，网页 url，正文等信息。最后我们简要地介绍网页去噪的主要应用。

本章的主要内容有：

- 网页噪音概述
- 一种网页去噪方法
- 网页去噪的应用

3.1. 网页噪音概述

3.1.1 网页噪音的概念和分类

噪音可以理解为任何不喜欢的声音。所以，顾名思义，网页噪音就是在一个页面内与页面主题无关（浏览者不关心）的区域及项。例如：为了增强交互性和制作动态网页而加入的 script 内容；出于商业目的而加入的广告；为了使网页美观而加入的修饰内容等等。这些内容的存在，使得准确地识别并清除网页中的噪音内容成为提高 Web 处理准确性的一项重要技术。它通常分布在网页的边缘，有时也夹杂在主题内容中间。

一个页面中常见的噪音分为以下几类：

(1) 导航类：为了维持网页间的链接关系，方便浏览者对网站进行浏览而设置的链接。

(2) 修饰类：为了美化页面而采用的背景、修饰图片、动画等。如站点标志图片、广告条。

(3) 交互类：为了收集用户提交信息或提供站内搜索服务的表单等。如在线问卷调查表。

(4) 其它类：网页中声明的版权信息、创建时间、作者等描述性信息。

3.1.2 相关研究

有时候，我们可能从这些噪声内容中得到一些我们需要的内容；另一些时候，我们可能不希望这些噪声内容来干扰我们的注意力。大部分时候，噪音内容都无内容相关性，同时，它们链接的网页常常也无内容相关性。这样，网页中的噪音内容不仅给 Web 上基于网页内容的应用系统带来困难，也给基于网页超链接指向的应用系统带来困难。

在搜索引擎领域，检索结果的准确性和速度是评价一个检索系统的两个主要指标。如果不去除原始网页的噪音内容，检索系统必然对噪音内容也建立索引，若在查询过程中，查询关键词在某个网页的噪音内容中出现，那么必然导致把该网页作为结果返回，而网页的主题内容可能和这个查询关键词完全无关。这样，检索的准确度就会下降。这显然不是我们和用户希望看到的。同时，若对噪音内容也建立索引，也必然导致索引结果的规模变大，从而降低检索的速度。所以，针对噪音给搜索引擎领域带来的影响，大量的研究也已经展开。

Shian-Hua Lin and Jan-Ming Ho[1]提出首先根据 table 标签将网页分成若干内容块(content block)，然后将词作为特征抽取出来，并计算每个特征词的熵值，接着根据内容块中每个特征词的熵值进而计算每个内容块的熵值，最后通过与熵值的阈值比较来划分出主题内容块和噪音内容块。此种方法将页面看成是由 table 分割的集合，不过对于无 table 的网页则很难成立。

张志刚、陈静、李晓明[2]提出以一组启发式规则为基础，利用信息检索的技术以及 Web 网页的特征，提取网页的主题以及和主题相关的内容，从而达到去噪的目的。该方法已经应用于搜索引擎系统（天网）中。

欧健文、董守斌、蔡斌[3]提出一种基于模板化的网页主题提取方法，该方法采用机器学习方式生成网页集的模板，以网页链接关系中的锚点文本作为提取目标对模板进行标记，生成对应模板的提取规则，依据模板的提取规则对网页主题信息进行提取。但是该方法只对模板型网页集效果显著。

封化民等[4]提出了一种新型的 Web 页面分析和内容提取框架，该框架既包括一种新型的含有位置信息的坐标树模型，还包括能反映空间关系的图模型，将 HTML 文档转化为坐标树，并结合位置特征和空间关系对网页进行分析和提取内容，准确率较高。

荆涛、左万利[5]提出利用网页的布局信息对页面进行划分，并在此基础上消除噪音。

孙承杰，关毅[6]提出了一种依靠统计信息从中文新闻类网页中抽取正文内容的方法。该方法先根据网页中的 HTML 标记把网页表示成一棵树，然后利用树中每个结点包含的中文字符数从中选择包含正文信息的结点。

上述方法各有各的优点和应用领域。但是处理目的都是去除网页中的噪音内容，得到网页的主题内容。可见，人们对于噪音对网页的影响是十分重视的，对网页噪音的去除工作还将继续研究下去。

3.2. 一种实用的网页去噪方法

3.2.1 HTML 基本知识

HTML（Hyper Text Markup Language）是建立发表联机文档采用的语言，称为超文本标识语言。HTML 文档也称为 Web 文档，它由文本、图形、声音和超链接等组成。下面简要地介绍一下在去噪过程中所涉及到的 HTML 中常用的一些概念。

1. 标记

HTML 用于描述功能的符号称为标记，如“HTML”、“BODY”、“TABLE”、“P”等等。标记在使用时必须用尖括号“<>”括起来，而且是成对出现。例如“<HTML>”与“</HTML>”必须成对出现、“<P>”与“</P>”必须成对出现等。其中无斜杠的标记表示该标记的作用开始，有斜杠的标记表示该标记的作用结束。注意：HTML 命令不区分大小写，所有的 HTML 页都必须用特定的标记开头，这中标记表达了文档的用途和位置。

最外面的一层元素是<html>，里面包含了整个网页文档。

下一个元素是<head>，它指定文档的初始信息，用于把与文档有关的信息与文档主体分开，它不会作为文档文字本身的一部分显示出来。在<head>里面包含的是<title>元素（文档标题）、<meta>元素（附加信息），还有一些 script 脚本元素等等。其中只有<title>元素是必需的，其余部分可选。

一个 Web 页的主体是用户在浏览时实际看到的文档正文信息。所有的 WWW 文档主体部分包括在<body>元素中。起始元素<body>用于显示信息起始的位置，结束元素</body>指出信息终止的位置。在主体内部，可以存在表格、文字、超链接、图象等信息。

一般 HTML 网页的格式如图 3.2.1 所示。

```
<html>
<head><title>example</title></head>
<body>
<table width="700" height="600" border="1">
   <tr>
      <td width="557" valign="top">
       <p>段落内容</p>
       </td>
   </tr>
</table>
</body>
</html>
```

图 3.2.1 HTML 文件

Fig.3.2.1 HTML document

2. META

在几乎所有的 HTML 网页中，我们都可以看到类似下面这段 HTML 代码：

<head>

<meta http-equiv="Content-Type" content="text/html; charset=gb2312">

</head>

这就是 META 标签的典型运用。META 标签是 HTML 语言 HEAD 区的一个辅助性标签，它位于 HTML 文档头部的<HEAD>标记和<TITLE>标记之间，它提供用户不可见的信息。META 标签通常用来为搜索引擎 robots 定义页面主题，或者是定义用户浏览器上的 cookie；它可以用于鉴别作者，设定页面格式，标注内容提要和关键字；还可以设置页面使其可以根据你定义的时间间隔刷新自己，以及设置 RASC 内容等级等等。

下面介绍一些有关 META 标记的常见例子及解释：

META 标签分两大部分：HTTP 标题信息（HTTP-EQUIV）和页面描述信息（NAME）。

(1) HTTP-EQUIV：HTTP-EQUIV 类似于 HTTP 的头部协议，它回应给浏览器一些有用的信息，以帮助正确和精确地显示网页内容。常用的 HTTP-EQUIV 类型有：

- Content-Type 和 Content-Language（显示字符集的设定）

说明：设定页面使用的字符集，用以说明主页制作所使用的文字已经语言，浏览器会根据此来调用相应的字符集显示 page 内容。

用法：<meta http-equiv="Content-Type" Content="text/html; Charset=gb2312">

<meta http-equiv="Content-Language" Content="zh-CN">

该 META 标签定义了 HTML 页面所使用的字符集为 GB2132，就是国标汉字码。如果将其中的“charset=GB2312”替换成“BIG5”，则该页面所用的字符集就是繁体中文 Big5 码。当你浏览一些国外的站点时，IE 浏览器会提示你要正确显示该页面需要下载 xx 语支持。这个功能就是通过读取 HTML 页面 META 标签的 Content-Type 属性而得知需要使用哪种字符集显示该页面的。如果系统里没有装相应的字符集，则 IE 就提示下载。其他的语言也对应不同的 charset，比如日文的字符集是“iso-2022-jp ”，韩文的是“ks_c_5601”。

- Refresh（刷新）

说明：让网页多长时间（秒）刷新自己，或在多长时间后让网页自动链接到其它网页。用法：<meta http-equiv="Refresh" Content="30">

<meta http-equiv="Refresh" Content="5; Url=http://www.sina.com.cn">

注意：其中的 5 是指停留 5 秒钟后自动刷新到 URL 网址（http://www.sina.com.cn）。

- Expires（期限）

说明：指定网页在缓存中的过期时间，一旦网页过期，必须到服务器上重新调阅。

用法：<meta http-equiv="Expires" Content="0">

<meta http-equiv="Expires" Content="Wed, 26 Feb 1997 08:21:57 GMT">

注意：必须使用 GMT 的时间格式，或直接设为 0（数字表示多少时间后过期）。

(2) NAME 变量：name 是描述网页的，对应于 Content（网页内容），以便于搜索引擎机器人查找、分类（目前几乎所有的搜索引擎都使用网上机器人自动查找 meta 值来给网页分类）。name 的 value 值（name=""）指定所提供信息的类型。有些值是已经定义好的。例如 description（说明）、keywords（关键字）、refresh（刷新）等。还可以指定其他任意值，如：creationdate（创建日期）、document ID（文档编号）和 level（等级）等。name 的 content 指定实际内容。如：如果指定 level（等级）为 value（值），则 Content 可能是 beginner（初级）、intermediate（中级）、advanced（高级）。其语法格式是：<META NAME="xxx" CONTENT="xxxxxxxxxxxxxxxxxx">。其中，xxx 主要有下面几种参数：

- Keywords（关键字）

说明：keywords 用来告诉搜索引擎你网页的关键字是什么。

用法：<Meta name="Keywords" Content="关键词 1, 关键词 2, 关键词 3, ……">

注意：各关键词间用英文逗号“,”隔开。META 的通常用处是指定搜索引擎用来提高搜索质量的关键词。当数个 META 元素提供文档语言从属信息时，搜索引擎会使用 lang 特性来过滤并通过用户的语言优先参照来显示搜索结果。例如：

<meta name="Kyewords" Lang="EN" Content="vacation,greece,sunshine">

<meta name="Kyewords" Lang="FR" Content="vacances,grèce,soleil">

- Description（简介）

说明：Description 用来告诉搜索引擎你的网站主要内容。

用法：<meta name="Description" Content="你网页的简述">

例如：<META NAME="description" CONTENT="This page is about the meaning of life, the universe, mankind and plants.">

- Robots（机器人向导）

说明：Robots 用来告诉搜索机器人哪些页面需要索引，哪些页面不需要索引。Content 的参数有 all、none、index、noindex、follow、nofollow。默认是 all。

用法：<meta name="Robots" Content="All|None|Index|Noindex|Follow|Nofollow">，其中，

all：文件将被检索，且页面上的链接可以被查询；

none：文件将不被检索，且页面上的链接不可以被查询（和“noindex, no follow” 起相同作用）；

index：文件将被检索；（让 robot/spider 登录）；

follow：页面上的链接可以被查询；

noindex：文件将不被检索，但页面上的链接可以被查询（不让 robot/spider 登录）；

nofollow：文件将不被检索，页面上的链接可以被查询（不让 robot/spider 顺着此页的连接往下探找）。

注意：许多搜索引擎都通过放出 robot/spider 搜索来登录网站，这些 robot/spider 就要用到 meta 元素的一些特性来决定怎样登录。

- Author（作者）

说明：标注网页的作者或制作组。

用法：<meta name="Author" Content="张三, zhangsan@163.com">

注意：Content 可以是你或你的制作组的名字或 Email。

以上是 META 标签的一些基本用法，其中最重要的就是：Keywords 和 Description 的设定。为什么呢？道理很简单，这两个语句可以让搜索引擎能准确的发现你，吸引更多的人访问你的站点。根据现在流行搜索引擎（Google，Lycos，AltaVista 等）的工作原理，搜索引擎先派机器人自动在 WWW 上搜索，当发现新的网站时，便于检索页面中的 Keywords 和 Description，并将其加入到自己的数据库，然后再根据关键词的密度将网站排序。

由此看来，我们必须记住添加 Keywords 和 Description 的 META 标签，并尽可能写好关键字和简介。否则，后果就会是：

- 如果你的页面中根本没有 Keywords 和 description 的 meta 标签，那么机器人是无法将你的站点加入数据库，网友也就不可能搜索到你的站点；
- 如果你的关键字选的不好，关键字的密度不高，被排列在几十甚至几百万个站点的后面被点击的可能性也是非常小的。

所以,我们必须记住添加 keywords 和 description 的 meta 标签，并尽可能写好关键字和简介。

写 keywords 的禁忌是：

- 不要用常见词汇。例如 www，homepage，net，web 等。
- 不要用形容词、副词。例如“最好的”、“最大的”等。
- 不要用笼统的词汇，要尽量精确。例如不用“摩托罗拉手机”，改用“V998”等。

寻找合适的关键词的一个技巧是：到 lycos，Alta，infoseek 等搜索引擎，搜索与你内容相仿的网站，查看排名前十位的网站的 meta 关键字，copy 下来用到自己的站点上。

这里，我们举一段微软多媒体网站的关键字供你参考：

<meta name="keywords" content="joke, music, MP3, media, audio, joke of the day, downloads, free music, horoscope, radio, video, music downloads, movies, radio station, media player, free, download, weather, horoscopes, windows media technologies, online, radio station guide, free download">

另外为了提高搜索点击率，这里还有一些“捷径”：

- 为了增加关键词的密度，将关键字隐藏在页面里（将文字颜色定义成与背景颜色一样）。
- 在图象的 alt 注释语句中加入关键字，如：<IMG SRC="xxx.gif" Alt="keywords">
- 利用 HTML 的注释语句，在页面代码里加入大量关键字。

最后，我们举一个使用 meta 的比较完整的例子，如图 3.2.2 所示：

```
<head>
<title>meta 使用示例</title>
<meta http-equiv="Content-Language" content="zh-cn">
<meta http-equiv="Content-Type" content="text/html;
        charset=gb2312">
<meta name="制作人" content="张三">
<meta name="Keywords" content="HTML, 搜索引擎,
        去噪">
<meta name="description" content="这是一篇
        关于搜索引擎方面的文章。">
</head>
```

图 3.2.2 meta 使用举例

Fig.3.2.2 Example of using meta

3. 表格

全世界的网站基本上都是使用 HTML 表格对页面区域进行规划的，表格是 HTML 一项非常重要的功能，利用其多种属性能够设计出多样化的表格，表格可以使页面更加整齐美观，产生意想不到的效果。常用表格标记如下：

1) <table>…</table>：表格指令。

相关属性：

- align：表格的水平位置，参数值为 left、center、right；
- bgcolor：表格背景颜色，参数值为 RGB 值；
- border：表格边框线宽度，参数值为正整数；
- height：表格高度，参数值为象素或百分比；
- width：表格宽度，参数值为象素或百分比。

2) <caption>…</caption>：表格标题。

相关属性：

- align：表格标题的水平位置，参数值为 left、center、right。

3) <tr>…</tr>：定义表格的一行。

相关属性：

- align：表格行的水平位置，参数值为 left、center、right。

4) <th>…</th>：表头单元格。表格中的文字将以粗体显示（</th>可以省略）。

相关属性：

- align：表格标题的水平位置，参数值为 left、center、right；
- colspan：单元格跨越的列数，参数值为正整数；
- rowspan：单元格跨越的行数，参数值为正整数。

5) <td>…</td>：表格单元格。

相关属性：

- align：单元格的水平位置，参数值为 left、center、right；
- bgcolor：单元格的背景颜色，参数值为 RGB 值；
- height：单元格的高度，参数值为象素或百分比；
- width：单元格的宽度，参数值为象素或百分比。
- colspan：单元格跨越的列数，参数值为正整数；
- rowspan：单元格跨越的行数，参数值为正整数。

现将表格中的常见属性、属性值、功能等通过表 3.2.1 来进一步说明：

表 3. 2. 1 表格属性

Tab.3.2.1 Table attributes

参数	参数值	用法示例	作用及说明
width	象素或百分比	width="778" width="90%"	定义表格的宽度。可以是绝对值(象素)也可以是相对值(相对屏幕或某区域的宽度)
height	象素或百分比	height="300" height="30%"	定义表格的高度，用法同 width。
align	left、center、right	align="left"	定义表格的水平位置 [示例为左对齐]
valign	top、middle、bottom	valign="middle"	定义表格内对象的垂直位置
border	正整数	border="1"	定义表格及单元格边框线宽度
bordercolor	RGB 值	bordercolor="#FF00FF"	定义表格线条颜色 参见[参见调色原理]
bgcolor	RGB 值	bgcolor="#C0C0C0"	定义表格背景颜色
backcolor	RGB 值	backcolor="#808080"	定义文字背景颜色
cellspacing	正整数	cellspacing="0"	定义单元格间距

cellpadding	正整数	cellpadding="0"	定义单元格填充
rowspan	正整数	rowspan="2"	定义单元格行域
colspan	正整数	colspan="2"	定义单元格列域
id	文本	id="AutoNumber1"	定义表格名称

在 html 页面中，有些网页使用表格排版是通过嵌套来完成的，即一个表格内部可以嵌套另一个表格，用表格来排版页面的思路是：由总表格规划整体的结构，由嵌套的表格负责各个子栏目的排版，并插入到表格的相应位置，这样就可以使页面的各个部分有条不紊，互不冲突，看上去清晰整洁。在实际做网页时一般不显示边框，边框的显示可根据自己的爱好来设定。表格嵌套在网页的去噪过程中是一个比较复杂的工作，我们会在后面的具体去噪算法中提到对于嵌套表格的网页的处理方法。

对于 HTML 的表格中的常见标记和属性，我们大致就分析到这里，为了更好地理解网页中的表格操作，我们通过一个小例子来具体说明，如图 3.2.3 所示：

HTML 源码

```
<table align="center" border="1" cellspacing="1" width="95%" id="AutoNumber22">
<caption><font size="3">固定活动安排</font></caption><br><br>
<tr>
<th width="30%"><font size="2">时间</font></th>
<th width="37%"><font size="2">活动安排</font></th>
<th width="33%"><font size="2">地点</font></th>
</tr>
<tr>
<th width="30%"><font size="2">星期一</font></th>
<td width="37%"><font size="2">业务学习</font></td>
<td width="33%"><font size="2">会议室</font></td>
</tr>
<tr>
<th width="30%"><font size="2">星期五</font></th>
<td width="37%"><font size="2">义务劳动</font></td>
<td width="33%"><font size="2">公园</font></td>
</tr>
</table>
```

浏览器显示结果

固定活动安排

时间	活动安排	地点
星期一	业务学习	会议室
星期五	义务劳动	公园

图 3.2.3 HTML 表格示例

Fig.3.2.3 HTML table example

4. 超链接与锚文本

超链接是 HTML 中很重要的功能。因为只有 HTML 拥有超链接的功能，才能使用户接入 Internet、WWW，享受多姿多彩的网络世界。超链接在本质上属于一个网页的一部分，它是一种允许我们同其他网页或站点之间进行连接的

元素。各个网页链接在一起后，才能真正构成一个网站。

超链接是一种对象，它以特殊编码的文本或图形的形式来实现链接，如果单击该链接，则相当于指示浏览器移至同一网页内的某个位置，或打开一个新的网页，或打开某一个新的 WWW 网站中的网页。

锚文本名字听起来难以理解，实际上锚文本就是链接文本。例如，在个人网站上把中央电视台（ www.cctv.com ）做为新闻频道的链接，访问者通过点击网站上的“新闻频道”就能进入 http://www.cctv.com 网站，那么“新闻频道”就是中央电视台网站首页的锚文本。

5. 段落

在编辑 HTML 文本时，按下回车键，浏览器中并不会形成段落。要形成段落必须显式地提示浏览器插入回车符。常见的是
、 和 （与 搭配使用）标记，其中
产生回车； 和 产生回车和一空行，文件段落的开始由 来标记，段落的结束由 来标记， 是可以省略的，因为下一个 的开始就意味着上一个 的结束。 标签还有一个属性 align，它用来指名字符显示时的对齐方式，一般值有 left、center、right 三种。下面，我们通过两个小例子来说明
和 的用法和主要区别，如图 3.2.4 和 3.2.5 所示。

HTML 源码	浏览器显示结果
<html> <head> <title>段落标签的使用示例 1 </title> </head> <body> 登鹳雀楼 白日依山尽， 黄河入海流。 欲穷千里目， 更上一层楼。 </body> </html>	登鹳雀楼 白日依山尽， 黄河入海流。 欲穷千里目， 更上一层楼。

图 3.2.4 HTML 段落示例 1

Fig.3.2.4 HTML paragraph example 1

HTML 源码	浏览器显示结果
<html> <head> <title>段落标签的使用示例 2 </title> </head> <body> 登鹳雀楼 白日依山尽，黄河入海流。 欲穷千里目，更上一层楼。 </body> </html>	登鹳雀楼 白日依山尽，黄河入海流。 欲穷千里目，更上一层楼。

图 3.2.5 HTML 段落示例 2

Fig.3.2.5 HTML paragraph example 2

3.2.2 HTMLParser 类

HTMLParser 是一个对现有的 HTML 进行分析的快速实时的解析器。它是一个开源项目，通过它，可以准确高效地对 HTML 文本中的格式、数据进行处理。利用它可以很容易地对网页的内容进行分析、过滤和抓取。详情可访问其官方站：http://htmlparser.sourceforge.net/。它的主要功能有如下几个部分：

- 文本信息抽取：提取网页中的文字。
- 链接提取：提取网页中的链接信息和锚文本。
- 资源提取：网页中图片、声音的处理。
- 链接检查：用于检查 HTML 中的链接是否有效。
- 内容检验，可以用来过滤网页上一些令人不愉快的字词。

HTMLParser 类虽然并没有对以上提到的一些功能进行专门的处理，但是它完全可以胜任上面提及的功能，在实际应用中如果遇见上面提及的问题可以使用它内部的一些方法来处理。本节要介绍的算法应用 HTMLParser 类来分割网页中的各个 table，得到各个 table 的 width 和 height 属性值，并对去噪后的网页进行去标签，也就是将其转化成为文本文件。现将本节中实现的去噪算法中使用的 HTMLParser 类及主要相关方法作一下简要介绍。

1. Parser 类和 NodeList 类

Parser 类是 HTMLParser 包中主要的类，是 HTMLParser 的入口。它的功能是解析一个网页。可以将 HTML 源代码以字符串的形式传给它，或者直接传递一个 url 地址。NodeList 可以产生网页标记（或称为节点，如无特殊说明，以下都称为节点）列表。通过 NodeList，Parser 可以提供访问网页内容的入口。二者经常结合使用，典型的使用如下：

```
Parser parser = new Parser (“yourUrl");
NodeList list = parser.parse ();
// do something with your list of nodes.
```

2. TitleTag 类

TitleTag 类可以得到网页中的 title 标签，实现方法为 getTitle()，具体使用如下：

```
TitleTag titleNode=new (TitleTag)node;
title=titleNode.getTitle();//得到网页的 title
```

3. MetaTag 类

MetaTag 类可以得到网页中的 meta 标签中的属性，实现方法为 getAttribute()，具体使用如下：

```
MetaTag metanode = (MetaTag) node;
meta=metanode.getAttribute("content");//得到 meta 中的 content 内容，包括网页
                                      //的关键字、编码、描述信息等。
```

4. TextNode 类

TextNode 类可以表示 HTML 文件中的普通文本内容。其得到文本内容的实现方法为 getText()，具体使用如下：

```
TextNode textnode = (TextNode) node;
text = textnode.getText();//得到文本内容
```

5. TableTag 类

TableTag 类可以得到网页中的 table 标签，得到某个 table 的内容的实现方法是 toHtml()，而得到某个 table 的高和宽之类的属性的方法为 getAttribute()，具体使用如下：

```
TableTag tabletag = (TableTag) node;//得到某个 table 节点
contentTemp = tabletag.toHtml();//得到 table 的内容
String width = tabletag.getAttribute("width");//得到 table 的 width 属性值
String height = tabletag.getAttribute("height");//得到 table 的 height 属性值
```

6. LinkTag 类

TableTag 类可以得到网页中的超链接标签，然后使用其中的 getLinkText()和 getText()方法得到网页中的锚文本和超链接。具体使用如下：

```
LinkTag linknode = (LinkTag) node;//得到某个 Link tag
String linkText=linknode.getLinkText();//得到锚文本
String link = linknode.getLink();//得到超链接
```

同时，该类的一些方法也可以判断出网页中的超链接是属于哪种类型的链接，具体使用如下：

6.2 isHTTPLink()：是否属于 HTTP 链接，是则返回 true，反之返回 false。

6.3 isFTPLink()：是否属于 FTP 链接，是则返回 true，反之返回 false。

6.4 isIRCLink()：是否属于 IRC 链接，是则返回 true，反之返回 false。

6.5 isJavascriptLink()：是否属于 Javascript 链接，是则返回 true，反之返回 false。

6.6 ismailLink()：是否属于邮件链接，是则返回 true，反之返回 false。

7. JspTag 类和 ScriptTag 类

我们知道，很多网页都是交互式的网页，也就是说存在比如显示当前时间、在线人数、访问量、留言板等动态内容，而这些是静态的 HTML 无法做到的，在这里就会用到 Jsp、Asp、JavaScript、VBScript 等语言进行编程。在嵌入到 HTML 源代码中时，在<%……%> 中编写 Jsp、Asp 代码，在<Script>……</Script>中编写 JavaScript、VBScript 代码。而 JspTag 类和 ScriptTag 类则可以方便的识别出其中的 Jsp、Asp、JavaScript、VBScript 代码，使用 JspTag 类中的 toString()方法和使用 ScriptTag 中的 getScriptCode()就可以得到这些代码。

8. NodeFilter 类

NodeFilter 类可以得到某个特定的节点，也就是说可以将 HTML 中存在的标签 XXXTag 解析出来，放到一个数组 Node[]中。几乎 HTML 的标签都有一个对应的类，如 TitleTag、MetaTag、TableTag 等，这些标签类都在 org.htmlparser.tags 包中。具体使用方法如下：

NodeFilter titleFilter = new NodeClassFilter(TitleTag.class); //title 标签

NodeFilter metaFilter = new NodeClassFilter(MetaTag.class); //mata 标签

NodeFilter tableFilter = new NodeClassFilter(TableTag.class); //table 标签

9. OrFilter 类

OrFilter 类可以得到其参数中设置的任何节点类型，节点之间是或（or）的关系。具体实现方法为 setPredicates()，括号里面的参数可以为需要得到的节点类型。具体实现如下：

NodeFilter titleFilter = new NodeClassFilter(TitleTag.class); //title 标签

NodeFilter metaFilter = new NodeClassFilter(MetaTag.class); //mata 标签

NodeFilter tableFilter = new NodeClassFilter(TableTag.class); //table 标签

OrFilter filter = new OrFilter();

filter.setPredicates(new NodeFilter[] {titleFilter, metaFilter, tableFilter, });//得到 title、

//meta、table

//节点

下面就 HTMLParser 类的使用给出几个示例：

1. 提取网页中的文本信息：

```
import java.io.*;
import org.htmlparser.filters.*;
import org.htmlparser.*;
import org.htmlparser.nodes.*;
import org.htmlparser.tags.*;
import org.htmlparser.util.*;
import org.htmlparser.visitors.*;

public static void readText (String result) throws Exception
{
 Parser parser ;
 NodeList nodelist ;
 parser = Parser.createParser(result,"GB2312″);
 NodeFilter textFilter = new NodeClassFilter(TextNode.class);
 OrFilter lastFilter = new OrFilter();
 lastFilter.setPredicates(new NodeFilter[]{textFilter });
 nodelist = parser.parse(lastFilter);
 Node[] nodes = nodelist.toNodeArray();
 String line ="";
 for(int i=0;i < nodes.length; i++)  {
  Node node = nodes[i];
  if(node instanceof TextNode)
  {
```

```
    TextNode textnode = (TextNode) node;
    line = textnode.getText();
    }
     System.out.println(line);
  }
}
```

图 3.2.6 提取 HTML 中的文本

Fig.3.2.6 Extract text of HTML page

在该示例中，使用到了 Parser、NodeList、NodeFilter、TextNode 等常见的类，这些类的具体使用已经在前一部分进行了介绍，再这里就不展开叙述了。在这里，我们主要对该示例的具体实现过程进行一下介绍。

1) 对 result 进行 parser，在这里，我们需要清楚示例中的 result 代表网页的 HTML 源文件（以字符串形式表示）；
2) 使用 NodeFilter 类得到 TextNode 节点，也就是得到表示文本内容的节点，然后将节点转化为数组 Nodes；
3) 对数组中的每一个节点，如果该 node 是 TextNode 类中的对象，则使用 TextNode 类中的 getText()方法得到网页的文本内容。在这里，使用到了 instanceof 操作符，它的作用是判断一个变量是否是右操作数指出的类的一个对象，由于 java 语言的多态性使得可以用一个子类的实例赋值给一个父类的变量，而在一些情况下需要判断变量到底是一个什么类型的对象，这时就可以使用 instanceof。

2. 提取网页的链接和锚文本

```
public static void readLink(String result) throws Exception
{
  Parser parser;
  NodeList nodelist;
  parser = Parser.createParser(result,"GB2312");
  NodeFilter linkFilter = new NodeClassFilter(LinkTag.class);
  OrFilter lastFilter = new OrFilter();
  lastFilter.setPredicates(new NodeFilter[] {linkFilter});
  nodelist = parser.parse(lastFilter);
  Node[] nodes = nodelist.toNodeArray();
  String link ="";
  String linkText="";
  for(int i=0;i < nodes.length; i++)  {
    Node node = nodes[i];
    if(node instanceof LinkTag)
    {
    LinkTag linknode = (LinkTag)node;
    linkText=linknode.getLinkText();
    link = linknode.getLink();
    }
     System.out.println(linkText);
    System.out.println(link);
```

```
 }
}
```

图 3.2.7 提取 HTML 中的链接和锚文本

Fig.3.2.7 Extract hyperlink and anchor text of HTML page

该功能的具体实现过程如下：

1) 对网页的源代码 result 进行 parser；
2) 使用 NodeFilter 类得到 LinkTag 节点，也就是得到表示超链接的标签，然后将其转化为 nodes 数组；
3) 对于数组中的每一个节点，如果该 node 是 LinkTag 类中的对象，则使用 LinkTag 类中的 getLinkText()和 getLink()方法分别得到网页的锚文本和超链接；

3.2.3 网页的分类

Web 上的网页根据内容可以分为三类：主题型网页、目录型网页和图片型网页。从内容上看，三种类型的网页有着较为明显的特征。主题型网页通常通过成段的文字描述一个或多个主题，虽然也会有图片和超链接，但这些图片和超链接不是网页的主体；目录型网页通常不会描述一个明确的主题，而是提供指向相关网页的超链接，因此，这样的网页中超链接密集，例如我们熟悉的一些门户网站的首页就属于目录型网页；图片型网页是通过图片来体现网页的内容，而文字仅仅是对图片的一个说明，因而文字不多。

3.2.4 算法思想

我们知道，网页都是有一定布局的，比如分左右两边或是中间和边缘。网页中的噪音一般都在页面中的次要位置，也就是说分布在主题内容周围，而重要的内容放在网页的中间部分，这符合设计者突出网页主题的做法，同时也符合人的浏览习惯。同时，噪音部分（例如导航条、广告、版权信息等）一般是以比较狭长的方式出现，这样在比较高宽比时可以轻易的去除，这也为本节提出的去除噪音算法带来了便利。如图 3.2.8 所示。

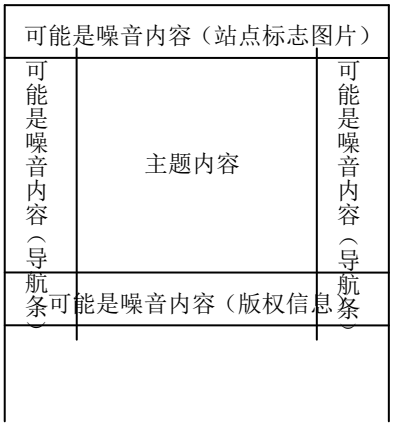


图 3.2.8 网页框架

Fig.3.2.8 Framework of web pages

对于表格型网页，即由 table 分割的网页，处理思想如下：

1. 对于主题型网页，由于主题内容块所在的表格（table）占据的空间较大，而且多是用成段的文字来描述，所以我们要根据这些信息识别出主题内容块，然后将其提取出来。

2. 对于目录型网页，由于网页中几乎没有成段的文字，而是伴随着大量的超链接，所以此时需要根据网页的表格（table）所占的空间来进行去噪，去掉比较小的表格（table）和相对狭长的表格（table）。

3. 对于图片型网页，由于网页主要内容是图片信息，文字信息较少，所以在处理时，首先识别出占据空间较大的表格（table），然后提取其中相对较少的文字就可以了。

而对于没有表格（table）的网页，即不是由 table 分割的网页，如果网页中存在段落文字，则提取出来即可，并没有考虑更多。因为对于大多数网页设计者来说，通常他们会先进行网页布局的设计，即先用表格等标记在页面上描绘出页面内容分布的区域，然后再在每个区域内部进行详细的内容设计，把需要的元素加进去。所以没有表格（table）的网页现在越来越少，这必然是以后网页制作发展的主流方向。本节就是根据这样一种思想来进行网页去噪，然后将提取出的主题内容变为文本文件，为以后对网页的一些处理，如分类、检索等提供了很大的方便。

3.2.5 算法实现依据的规则

根据观察一般网页的 HTML 文档以及其它网页的格式，可以得到一些启发式规则，如下：

1. 标签<table>和</table>之间如果有标签<p>或
，可以看做是正文内容。也就是说认为网页的主题通常是用成段的文字来描述；

2. 若标签<table>的 width 或 height 属性为其占页面的百分比，则需要根据这个百分比的值来确定其是否为主题内容。若 width 或 height 属性的百分比数值较大，则认为有可能是主题内容；

3. 对于多层嵌套的标签<table>，认为只在其中某一层 table 中存在主题内容；

4. 对于没有标签<table>的网页，即不是由表格分割的网页，如果存在段落文字，则认为是主题内容。

3.2.6 算法流程

1. 准备工作：由于去噪使用的数据是天网的 CWT200G 语料，而天网存储语料的格式又有其自身的特殊性，所以，首先根据天网存储网页的格式进行语料的切分，将其切分成单一网页的形式。当然如果对于普通的正常网页，该步骤则不需要。

2. 上述准备工作完成后，开始进行去噪工作。对于没有 table 的网页，根据网页中是否有段落文字来判断是否为主题内容。算法如图 3.2.9 所示。

```
If (无标签<table>)
{
    If  (存在可以表示段落文字的<p>或
<br>标签)
   {
       认为是主题内容，保留
    }
}
```

图 3.2.9 无 table 网页的去噪算法

Fig.3.2.9 eliminating noise content algorithm of no table web pages

3. 对于有 table 的网页，处理方法如下：

(1) 解析网页的 HTML 源代码，然后得到其所有 table 节点，主要源代码如下：

```
Parser myParser = Parser.createParser(result, "GBK");//result 为 HTML 源文件内容
String filterStr = "table";
NodeFilter filter = new TagNameFilter(filterStr);
```

(2) 将得到的 table 节点解析出来放到一个数组中，主要源代码如下：

```
nodeList = myParser.extractAllNodesThatMatch(filter);
nodes = nodeList.toNodeArray();
```

(3) 得到所有 table 节点后，对每一个 table 进行分析，得到其高（width）和宽（height）属性值。主要源代码如下：

```
Node node1 = nodes[i];//网页中的第 i 个 table
TableTag tabletag = (TableTag) node1;
String width = tabletag.getAttribute("width");//得到每一个 table 的宽属性值
String height = tabletag.getAttribute("height");//得到每一个 table 的高属性值
```

(4) 对 table 的判断需要对以下几个情况分别考虑：

- width 和 height 属性有一个不存在且 width 不以百分比的形式出现，若 width 属性值大于事先给定的阈值，然后再判断其是否存在段落内容，若二者都满足，则将该 table 保留。
- 有些情况下，table 的 width 属性值是以占页面百分比的形式出现，那么此时若 width 和 height 属性都存在且 width 属性以百分比的形式出现，若 width 属性值大于事先给顶的阈值，然后再判断其是否存在段落内容，若二者都满足，则将该 table 保留。
- width 和 height 属性都存在且 width 和 height 属性值以数值的方式出现，则比较 width 和 height 属性的比值，去掉高宽比比较大的（也就是说，比较狭长的）表格，然后对于剩余的表格，判断表格内部是否存在段落文字，若存在则保留。

- 对于目录型网页和图片型网页，认为 table 高宽比很小，但高和宽的属性值占页面很大的部分为主题内容，保留下来。

对于这四种情况具体实现的算法如图 3.2.10 所示：

```
For (对于每一个 table 表格)
{
    找到 table 的 width 和 height 属性
    If (width 和 height 属性有一个不存在且 width 不以百分比的形式出现)
    {
        If (width 属性值> δ1)   // δ1 为 width 属性阈值
            If (有段落文字)
                认为是主题内容，保留
    }
    Else if (width 和 height 属性都存在且 width 属性以百分比的形式出现)
    {
        If (width 百分比数值> δ2)   // δ2 为 width 百分比数值的阈值
            If (有段落文字)
                认为是主题内容，保留
    }
    Else if (width 和 height 属性都存在且 height 属性以百分比的形式出现)
    {
        If (height 百分比数值> δ3)   // δ3 为 height 百分比数性阈值
            If (有段落文字)
                认为是主题内容，保留
    }
    Else if (width 和 height 属性都存在且 width 和 height 属性值以数值的方式出现)
    {
        计算 width 与 height 的长宽比
        If (长宽比很小)
            If (有段落文字)   //该层判断是为了防止出现长宽比很小的图片等
                              //非主题内容
                认为是主题内容，保留
    }
    Else   //对于没有段落文字，即对于目录型网页或图片型网页
    {
        认为 table 长宽比很小，但长和宽的值占页面很大的部分为主题内容，保留
    }
}//endfor
```

图 3.2.10 有 table 网页的去噪算法

Fig.3.2.10 eliminating noise content algorithm of table web pages

4. 由于在后续的建索引工作中需要对文本文件进行处理，所以去噪后，还要将保留下来的部分进行去标签，具体实现过程如下：

```
NodeFilter textFilter = new NodeClassFilter(TextNode.class); //得到文本内容
parser = Parser.createParser(text, "GBK");//解析 text（text 为保留下来的内容）
OrFilter filter = new OrFilter();
filter.setPredicates(new NodeFilter[] {textFilter});//得到保留下来的内容的节点
nodelist = parser.parse(filter);
Node[] nodes = nodelist.toNodeArray();//将解析后的节点放到 Node[]数组中
for (int i = 0; i < nodes.length; i++) {
   Node node = nodes[i];//每一个节点 i
   TextNode textnode = (TextNode) node;
   String tempText = textnode.getText();//去标签，得到纯文本内容
}
```

5. 将每一个网页去噪后保留下来的纯文本内容输出到一个文本文件中。

3.2.7 去噪效果演示

下面以 CWT200G 中的一个网页为例来显示去除噪音前的网页和去除噪音后的文本文件，如图 3.2.11 所示。

(a) 处理前

(b) 处理后

图 3.2.11 处理前后结果对比

Fig.3.2.11 Comparison before and after processing

3.2.8 去噪方法的优点

1. 该去噪方法的去噪速度大约为 18 个/秒，所以可以较迅速地清除网页噪音；

2. 该去噪方法的准确率较高。由于网页较多，不可能对每个网页都进行准确率检查，所以随机抽取 2000 个网页进行手工检查。现将检查结果用“优、良、中、差”四个标准进行判断。其中“优”代表网页主题内容正确提取，且噪音基本去除；“良”代表网页主题内容正确提取，噪音存在一部分；“中”代表网页主题内容基本能正确提取，噪音存在较多；“差”代表网页噪音基本没有消除或者主题内容没有正确提取。结果如表 3.2.2 所示。

表 3.2.2 去除噪音结果

Tab.2.2.2 Results of eliminating noise content

测试标准	结果(%)
优	49.2
良	32.1
中	15.9
差	2.8

3. 符合网页设计者的设计习惯，即将主题内容放在网页中间部分，且占用篇幅较大；而将噪音部分放在网页边缘，这些区域占用篇幅较小且比较狭长。同时也符合读者的浏览习惯，即主要浏览页面中表达主题的中间部分，而不去过分注意导航条、广告栏、版权信息等边缘部分。

3.2.9 去噪方法的缺点

1. 对于某些类型的网页，去噪效果不佳。在对去噪效果不佳的网页的观察中，可以发现以下一些原因：

(1) 网页的主题内容不明显，即主题内容只有一句话，也就是说，主题内容不是段落文字；

(2) 网页中的主题是图片信息（即图片型网页）。

2. 一少部分网页的噪音中也存在一些段落文字的内容，也就是说存在<p>或
等认为是段落文字的标签，这样会误把这部分内容看做主题内容保留下来而没有被清除掉。该去噪算法已经对类似这样的错误进行了一定的处理，因为噪音中的段落文字通常比主题内容中的段落文字少，即噪音中的标签<p>或
的个数会比主题内容中的少，所以，针对这种情况，可以根据标签<p>或
的个数来判断该部分是噪音还是主题内容。

3.3. 网页去噪的优点和应用

随着对网页信息处理的日益增多，网页噪音所产生的负面影响越来越突出地显现出来，所以，如何对网页准确去噪便成为了不可回避的问题。快速准确地识别并清除网页内的噪音内容是提高 Web 信息处理结果准确性的一项关键技术。首先，网页去噪后，没有了噪音内容的干扰，Web 应用可以以网页的主题内容为处理对象，从而提高处理结果的准确性；其次，网页去噪可以显著地简化网页内标签结构的复杂性并减小网页的大小，从而节省后续处理过程的时间和空间开销。

将网页去噪应用到搜索引擎方面，可以大大地减少索引量、提高搜索引擎的检索速度和提高检索的准确度；应用到分类方面，可以将 Web 网页中的主题内容提取出来，存放到文本文件中，然后就可以很方便地应用目前现有的分类器进行自动分类。这样，对网页的处理就转化为对文本文件的处理，扩大了分类技术的适用范围，同时，分类的精确率也会有一定的提高。

3.4. 本章小结

针对目前对 Web 信息处理和搭建搜索引擎的应用，本章提出了一种实用高效的去噪方法——基于网页框架和规则的网页去噪方法。结果表明，该方法可以迅速地从网页中提取出主题内容并清除噪音，且清除噪音的准确率较高。同时，该方法也符合网页设计者的设计习惯，即将主题内容放在网页中间部分，且占用篇幅较大；而将噪音部分放在网页边缘，这些区域占用篇幅较小且比较狭长。网页去噪的应用很广泛，可以应用到大型搜索引擎的搭建过程中，也可以应用到 Web 网页的分类方面。但是，本章的去噪方法还有一些网页不能处理或处理的效果较差，同时算法中的阈值是在不断地实验中得出的，其合理性还有待进一步实验和观察，所以要在以后的工作和学习中不断弥补算法的不足。

第四章 WEB 搜索引擎中信息的索引

1．概述

所有搜索引擎的核心就是索引的概念：将原始数据处理成一个高效的交差引用的查找结构以便于快速的搜索。让我们对索引做一次快速的高层次的浏览。

想像一下，你需要搜索大量的文件，并且你想找出包含一个指定的词或短语的文件。你如何写一个程序来做到这个？一个简单的方法是针对给定的词或短语顺序扫描每个文件。这个方法有很多缺点，最明显的就是它不适合于大量的文件或者文件非常巨大的情况。这时就出现了索引：为了快速搜索大量的文本，你必须首先索引那个文本然后把它转化为一个可以让你快速搜索的格式，除去缓慢的顺序地扫描过程。这个转化过程称为索引，它的输出称为一条索引。你可以把索引理解为一个可以让你快速随机访问存于其内部的词的数据结构。它隐含的概念类似于一本书最后的索引，可以让你快速找到讨论指定主题的页面。在我们下面要介绍的 Lucene 中，索引是一个精心设计的数据结构，在文件系统中存储为一组索引文件。在文献[4]的附录 B 中详细地说明了索引文件的结构.[4]

2．Lucene 介绍

2．1 什么是 Lucene？

Lucene 是 apache 软件基金会[4] jakarta 项目组的一个子项目，是一个开放源代码[5]的全文检索引擎工具包，即它不是一个完整的全文检索引擎，而是一个全文检索引擎的架构，提供了完整的查询引擎和索引引擎，部分文本分析引擎（英文与德文两种西方语言）。Lucene 的目的是为软件开发人员提供一个简单易用的工具包，以方便的在目标系统中实现全文检索的功能，或者是以此为基础建立起完整的全文检索引擎。

Lucene 的原作者是 Doug Cutting，他是一位资深全文索引/检索专家，曾经是 V-Twin 搜索引擎的主要开发者，后在 Excite 担任高级系统架构设计师，目前从事于一些 Internet 底层架构的研究。早先发布在作者自己的 http://www.lucene.com/，后来发布在 SourceForge，2001 年年底成为 apache 软件基金会 jakarta 的一个子项目：http://jakarta.apache.org/lucene/。

2．2 Lucene 的应用、特点及优势

作为一个开放源代码项目，Lucene 从问世之后，引发了开放源代码社群的巨大反响，程序员们不仅使用它构建具体的全文检索应用，而且将之集成到各种系统软件中去，以及构建 Web 应用，甚至某些商业软件也采用了 Lucene 作为其内部全文检索子系统的核心。apache 软件基金会的网站使用了 Lucene 作为全文检索的引擎，IBM 的开源软件 eclipse 的 2.1 版本中也采用了 Lucene 作为帮助子系统的全文索引引擎，相应的 IBM 的商业软件 Web Sphere[10]中也采用了 Lucene。Lucene 以其开放源代码的特性、优异的索引结构、良好的系统架构获得了越来越多的应用。

Lucene 作为一个全文检索引擎，其具有如下突出的优点：

（1）索引文件格式独立于应用平台。Lucene 定义了一套以 8 位字节为基础的索引文件格式，使得兼容系统或者不同平台的应用能够共享建立的索引文件。

（2）在传统全文检索引擎的倒排索引的基础上，实现了分块索引，能够针对新的文件建立小文件索引，提升索引速度。然后通过与原有索引的合并，达到优化的目的。

（3）优秀的面向对象的系统架构，使得对于 Lucene 扩展的学习难度降低，方便扩充新 功能。

（4）设计了独立于语言和文件格式的文本分析接口，索引器通过接受 Token 流完成索引文件的创立，用户扩展新的语言和文件格式，只需要实现文本分析的接口。

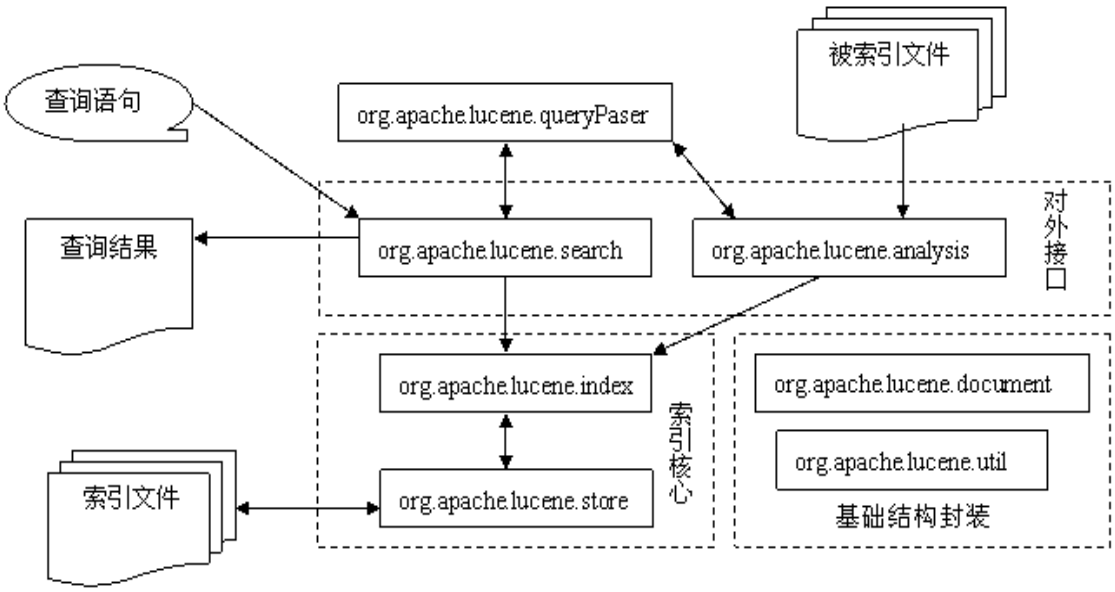
（5）已经默认实现了一套强大的查询引擎，用户无需自己编写代码即使系统可获得强大的查询能力，Lucene 的查询实现中默认实现了布尔操作、模糊查询（Fuzzy Search）、分组查询等等。

面对已经存在的商业全文检索引擎，Lucene 也具有相当的优势。首先，它的开发源代码发行方式（遵守 Apache Software License），在此基础上程序员不仅仅可以充分的利用 Lucene 所提供的强大功能，而且可以深入细致的学习到全文检索引擎制作技术和面相对象编程的实践，进而在此基础上根据应用的实际情况编写出更好的更适合当前应用的全文检索引擎。在这一点上，商业软件的灵活性远远不及 Lucene。其次，Lucene 秉承了开放源代码一贯的架构优良的优势，设计了一个合理而极具扩充能力的面向对象架构，程序员可以在 Lucene 的基础上扩充各种功能，比如扩充中文处理能力，从文本扩充到 HTML、PDF 等等文本格式的处理，编写这些扩展的功能不仅仅不复杂，而且由于 Lucene 恰当合理的对系统设备做了程序上的抽象，扩展的功能也能轻易的达到跨平台的能力。最后，转移到 apache 软件基金会后，借助于 apache 软件基金会的网络平台，程序员可以方便的和开发者、其它程序员交流，促成资源的共享，甚至直接获得已经编写完备的扩充功能。最后，虽然 Lucene 使用 Java 语言写成，但是开放源代码社区的程序员正在不懈的将之使用各种传统语言实现（例如.net framework），在遵守 Lucene 索引文件格式的基础上，使得 Lucene 能够运行在各种各样的平台上，系统管理员可以根据当前的平台适合的语言来合理的选择。

3．Lucene 系统结构

Lucene 作为一个优秀的全文检索引擎，其系统结构具有强烈的面向对象特征。首先是定义了一个与平台无关的索引文件格式，其次通过抽象将系统的核心组成部分设计为抽象类，具体的平台实现部分设计为抽象类的实现，此外与具体平台相关的部分比如文件存储也封装为类，经过层层的面向对象式的处理，最终达成了一个低耦合高效率，容易二次开发的检索引擎系统。

以下将讨论 Lucene 系统的结构组织，并给出系统结构与源码组织图：



从图中我们清楚的看到，Lucene 的系统由基础结构封装、索引核心、对外接口三大部分组成。其中直接操作索引文件的索引核心又是系统的重点。Lucene 的将所有源码分为了 7 个模块（在 java 语言中以包即 package 来表示），各个模块所属的系统部分也如上图所示。需要说明的是 org.apache.lucene.queryPaser 是做为 org.apache.lucene.search 的语法解析器存在，不被系统之外实际调用，因此这里没有当作对外接口看待，而是将之独立出来。

从面象对象的观点来考察，Lucene 应用了最基本的一条程序设计准则：引入额外的抽象层以降低耦合性。首先，引入对索引文件的操作 org.apache.lucene.store 的封装，然后将索引部分的实现建立在（org.apache.lucene.index）其之上，

完成对索引核心的抽象。在索引核心的基础上开始设计对外的接口 org.apache.lucene.search 与 org.apache.lucene.analysis。在每一个局部细节上，比如某些常用的数据结构与算法上，Lucene 也充分的应用了这一条准则。在高度的面向对象理论的支撑下，使得 Lucene 的实现容易理解，易于扩展。

Lucene 在系统结构上的另一个特点表现为其引入了传统的客户端服务器结构以外的的应用结构。Lucene 可以作为一个运行库被包含进入应用本身中去，而不是做为一个单独的索引服务器存在。这自然和 Lucene 开放源代码的特征分不开，但是也体现了 Lucene 在编写上的本来意图：提供一个全文索引引擎的架构，而不是实现。

4 Lucene 索引

4．1 Lucene 索引原理

Lucene 中使用的是倒排文件索引结构。该结构及相应的生成算法如下：

0）设有两篇文章 1 和 2

文章 1 的内容为：Tom lives in Guangzhou,I live in Guangzhou too.

文章 2 的内容为：He once lived in Shanghai.

1)由于 lucene 是基于关键词索引和查询的，首先我们要取得这两篇文章的关键词，通常我们需要如下处理措施

a.我们现在有的是文章内容，即一个字符串，我们先要找出字符串中的所有单词，即分词。英文单词由于用空格分隔，比较好处理。中文单词间是连在一起的需要特殊的分词处理。

b.文章中的”in”, “once” “too”等词没有什么实际意义，中文中的“的”“是”等字通常也无具体含义，这些不代表概念的词可以过滤掉

c.用户通常希望查“He”时能把含“he”，“HE”的文章也找出来，所以所有单词需要统一大小写。

d.用户通常希望查“live”时能把含“lives”，“lived”的文章也找出来，所以需要把“lives”，“lived”还原成“live”

e.文章中的标点符号通常不表示某种概念，也可以过滤掉在 lucene 中以上措施由 Analyzer 类完成

经过上面处理后

文章 1 的所有关键词为：[tom] [live] [guangzhou] [i] [live] [guangzhou]

文章 2 的所有关键词为：[he] [live] [shanghai]

2) 有了关键词后，我们就可以建立倒排索引了。上面的对应关系是：“文章号”对“文章中所有关键词”。倒排索引把这个关系倒过来，变成：“关键词”对“拥有该关键词的所有文章号”。文章 1，2 经过倒排后变成

关键词 文章号

guangzhou 1

he 2

i 1

live 1,2

shanghai 2

tom 1

通常仅知道关键词在哪些文章中出现还不够，我们还需要知道关键词在文章中出现次数和出现的位置，通常有两种位置：a)字符位置，即记录该词是文章中第几个字符（优点是关键词亮显时定位快）；b)关键词位置，即记录该词是文章中第几个关键词（优点是节约索引空间、词组（phase）查询快），lucene 中记录的就是这种位置。加上“出现频率”和“出现位置”信息后，我们的索引结构变为：

关键词	文章号[出现频率]	出现位置
guangzhou	1[2]	3，6
he	2[1]	1
i	1[1]	4
live	1[2],2[1]	2，5，2
shanghai	2[1]	3
tom	1[1]	1

以 live 这行为例我们说明一下该结构：live 在文章 1 中出现了 2 次，文章 2 中出现了一次，的出现位置为“2,5,2”这表示什么呢？我们需要结合文章号和出现频率来分析，文章 1 中出现了 2 次，那么“2,5”就表示 live 在文章 1 中出现的两个位置，文章 2 中出现了一次，剩下的“2”就表示 live 是文章 2 中第 2 个关键字。

以上就是 lucene 索引结构中最核心的部分。我们注意到关键字是按字符顺序排列的（lucene 没有使用 B 树结构），因此 lucene 可以用二元搜索算法快速定位关键词。实现时 lucene 将上面三列分别作为词典文件（Term Dictionary）、频率文件(frequencies)、位置文件 (positions)保存。其中词典文件不仅保存有每个关键词，还保留了指向频率文件和位置文件的指针，通过指针可以找到该关键字的频率信息和位置信息。

Lucene 中使用了 field 的概念，用于表达信息所在位置（如标题中，文章中，url 中），在建索引中，该 field 信息也记录在词典文件中，每个关键词都有一个 field 信息(因为每个关键字一定属于一个或多个 field)。

为了减小索引文件的大小，Lucene 对索引还使用了压缩技术。首先，对词典文件中的关键词进行了压缩，关键词压缩为<前缀长度，后缀>，例如：当前词为“阿拉伯语”，上一个词为“阿拉伯”，那么“阿拉伯语”压缩为<3，语>。其次大量用到的是对数字的压缩，数字只保存与上一个值的差值（这样可以减小数字的长度，进而减少保存该数字需要的字节数）。例如当前文章号是 16389（不压缩要用 3 个字节保存），上一文章号是 16382，压缩后保存 7（只用一个字节）。

下面我们可以通过对该索引的查询来解释一下为什么要建立索引。

假设要查询单词 “live”，lucene 先对词典二元查找、找到该词，通过指向频率文件的指针读出所有文章号，然后返回结果。词典通常非常小，因而，整个过程的时间是毫秒级的。

而用普通的顺序匹配算法，不建索引，而是对所有文章的内容进行字符串匹配，这个过程将会相当缓慢，当文章数目很大时，时间往往是无法忍受的。[3]

4．2 Lucene 索引文件格式

在 Lucene 的文件格式中，以字节为基础，定义了如下的数据类型：

Lucene 文件格式中定义的数据类型

<table>
<tr><th>数据类型</th><th>所占字节长度（字节）</th><th>说明</th></tr>
<tr><td>Byte</td><td>1</td><td>基本数据类型，其他数据类型以此为基础定义</td></tr>
<tr><td>UInt32</td><td>4</td><td>32位无符号整数，高位优先</td></tr>
<tr><td>UInt64</td><td>8</td><td>64位无符号整数，高位优先</td></tr>
<tr><td>VInt</td><td>不定，最少1字节</td><td>动态长度整数，每字节的最高位表明还剩多少字节，每字节的低七位表明整数的值，高位优先。可以认为值可以为无限大。其示例如下
<table>
<tr><th>值</th><th>字节1</th><th>字节2</th><th>字节3</th></tr>
<tr><td>0</td><td>00000000</td><td></td><td></td></tr>
<tr><td>1</td><td>00000001</td><td></td><td></td></tr>
<tr><td>2</td><td>00000010</td><td></td><td></td></tr>
<tr><td>127</td><td>01111111</td><td></td><td></td></tr>
<tr><td>128</td><td>10000000</td><td>00000001</td><td></td></tr>
<tr><td>129</td><td>10000001</td><td>00000001</td><td></td></tr>
<tr><td>130</td><td>10000010</td><td>00000001</td><td></td></tr>
<tr><td>16383</td><td>10000000</td><td>10000000</td><td>00000001</td></tr>
<tr><td>16384</td><td>10000001</td><td>10000000</td><td>00000001</td></tr>
<tr><td>16385</td><td>10000010</td><td>10000000</td><td>00000001</td></tr>
</table>
</td></tr>
<tr><td>Chars</td><td>不定，最少1字节</td><td>采用UTF-8编码[20]的Unicode字符序列</td></tr>
<tr><td>String</td><td>不定，最少2字节</td><td>由VInt和Chars组成的字符串类型，VInt表示Chars的长度，Chars则表示了String的值</td></tr>
</table>

以上的数据类型就是 Lucene 索引文件格式中用到的全部数据类型，由于它们都以字节为基础定义而来，因此保证了是平台无关，这也是 Lucene 索引文件格式平台无关的主要原因。

首先我们介绍 Lucene 索引后产生主要文件，下面是用 lucene 产生的索引目录：

2005-02-04 16:26 29 _a.fnm

2005-02-04 16:26 704 _a.fdt

2005-02-04 16:26 80 _a.fdx

2005-02-04 16:26 2,095 _a.frq

2005-02-04 16:26 25,967 _a.prx

2005-02-04 16:26 13,389 _a.tis

2005-02-04 16:26 181 _a.tii

2005-02-04 16:26 10 _a.f1

2005-02-04 16:26 10 _a.f2

2005-02-04 16:26 10 _a.f3

2005-02-04 16:26 27 segments

2005-02-04 16:26 4 deletable

1. segments 文件，用来表示索引文件的名字和 Document 的个数。

具体格式（在 SegmentInfos.java 描述，参见 write()）

文件头中的第 1 个 4Byte 是 SegmentName，在 Lucene 中，SegmentName 是一个 16 进制的数字，是上面的如*.fnm 中的*部分，每新增加一个 Document，SegmentName 会加 1（做的是 SegmentName++的动作），所以此部分的值会比实际的 SegmentName 大 1。

文件头中的第 2 个 4Byte 是索引文件的个数，Lucene 支持在一个目录中存放多个索引文件。

文件体是一个可重复的部分，重复的次数由文件头中的第 2 部分的大小决定。

可重复的部分包括 2 个部分：SegmentName 和该索引文件中包含的 Document 个数。　　（从这里可以看出，在 Lucene 中，单个索引文件最多可以包含 2 的 32 方个文件（4 个　byte）。

*.fnm 文件记录索引中的域名，并以域名的字符长度排序，当长度相等时根据其添 加顺序排序。（注：lucene 索引文件在写入 String 时，在 String 前都记录其长度（vint　型））.

2. *.frm 文件，用来保存 FieldName，注意，此文件只保存非切分部分的 Field，如通过 Filed.text()方法进行切分后的 Field 保存在*.tii, *.tis 中。

具体格式（在 FieldInfos.java 描述，参见 write()）。

文件头中的 FieldNum 代表索引的 Field 的个数。

文件体中描述每个 Field 的信息，个数与文件体中的 FieldNum 相同。

每个 Field 的信息包括 2 个部分：FieldName，索引的 Field 的名字，Flag 表示该 Field 是否是索引的（indexed）。0x00 表示未索引，0x01 表示索引。

3. *.fdt 文件，用于存放 Field 的值，每个 Document 是连续存放的，每个 Document 内部的格式（参看 FieldsWriter.java 文件的 addDocument()函数）头部中的 FieldNum 是该 Document 中选择了存储的 Field 的个数（field.isStroed()==true），　内容体中每个 Filed 中包括 3 个部分。

Pos：该 Field 在 Document 的顺序，是第几个 Field.

Flag：该 Field 是否是一个 Token（field.isToken()==true),）0x00,否；0x01,是。

Value：该 Field 的值。

4. *.fdx 文件，用于存放每个 Document 在 fdt 文件中的偏移。

每个 Document 中记录的是一个 Long 型的整数，代表*.fdt 文件中每个 Document 的偏移（起点-1）。

5. *.tis 文件，用于存放切分过的 Field 中的值。（参看 2，fnm 文件）

文件格式（参看 TermInfosWriter.java 文件中的 add()函数）

文件头中的 TermNum 是一个 4byte 的 Int，代表 Term 的个数。

文件体是每个 Term 的重复。每个 Term 包括 7 个域。

Start：该 Term 的偏移

Content：该 Term 的值

FieldPos：该 Term 出现的顺序，即是第几个 Field

Freq：该 Term 出现的频率

FreqPointer：Rreq 在*.frq 文件中的偏移

ProxPointer：Prox 在*.prx 文件中的偏移

IndexPointer：Index 在*.tii 文件中的偏移（我不敢确定），该项是可选项，只在 isIndex==true 时出现。（不明白这个的意思，isIndex 指什么）也可以参看 SegmentTermEnum.java 中的 next()函数，这是一个读单个 term 的函数。

6. deletable 文件，用来存放被删除的文件名。

文件格式（参看 IndexWriter.java 中的 readDeleteableFiles()函数）

文件体是一个 4byte 的整数，代表被删除的文件个数。

文件体中是一组被删除的文件名。

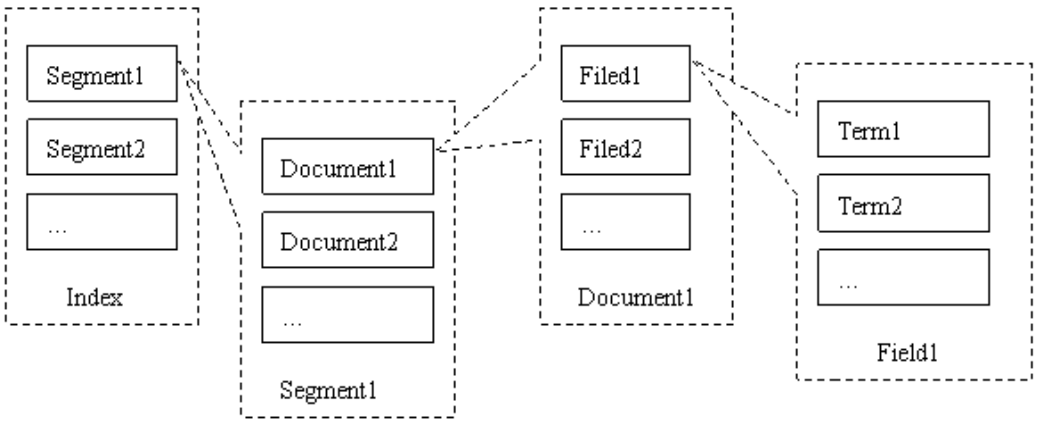
还有*.f(n)是存放 Norm 的文件,*.tii 是与*.tis 相对的，*.frq（存放词频的文件），*.prx（存放词频的位置的文件，与*.frq 文件联合使用）。

7. *_a.*文件是索引文件，segments 记录着_a，这样在查询的时候，查询所有含有_a 的文件。当然 lucene 也会限制文件名的扩展名。

每个索引文件下面还有一个 deletable 文件，记录着那个 document 被标记删除。

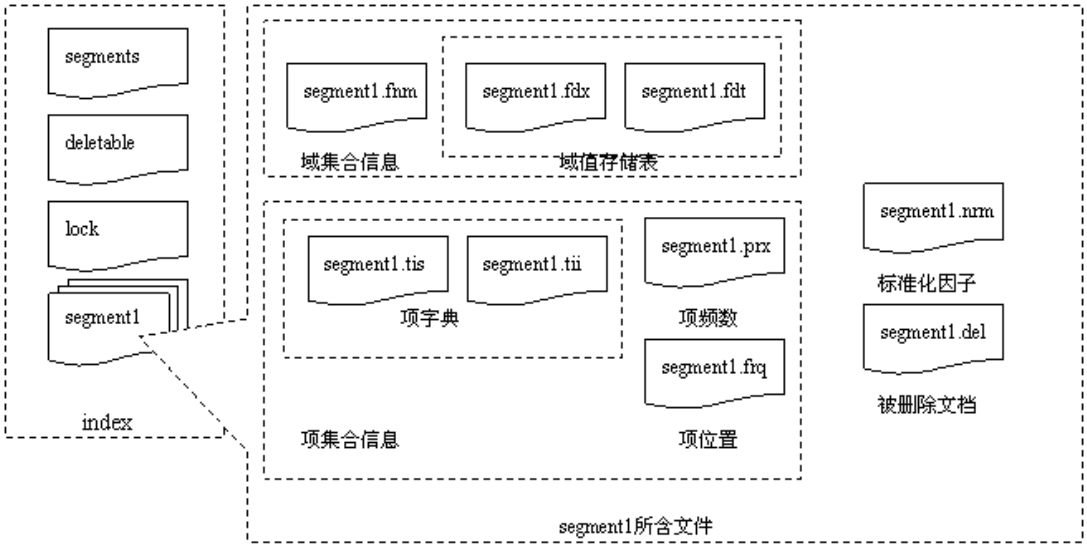
文件中有一些文件后缀名为*.fN,N 代表数字，如 1，2 等等，这个主要反映 document 中含有的 filed 数量，这个索引中含有的 filed 为 3 个，所以最大 n 为 3，假如有 100 个 field,那 n 最大为 100

在 Lucene 的 web 站点上，有关于 Lucene 的文件格式的规范，其规定了 Lucene 的文件格式采取的存储单位、组织结构、命名规范等等内容，但是它仅仅是一个规范说明，并没有从实现者角度来衡量这个规范的实现。因此，以下的内容，结合了我们自己的分析与文件格式的定义规范，以期望给出一个更加清晰的文件格式说明。接下来我们看看 Lucene 索引文件的概念组成和结构组成。



以上就是 Lucene 的索引文件的概念结构。Lucene 索引 index 由若干段(segment)组成，每一段由若干的文档（document）组成，每一个文档由若干的域（field）组成，每一个域由若干的项（term）组成。项是最小的索引概念单位，它直接代表了一个字符串以及其在文件中的位置、出现次数等信息。域是一个关联的元组，由一个域名和一个域值组成，域名是一个字串，域值是一个项，比如将“标题”和实际标题的项组成的域。文档是提取了某个文件中的所有信息之后的结果，这些组成了段，或者称为一个子索引。子索引可以组合为索引，也可以合并为一个新的包含了所有合并项内部元素的子索引。我们可以清楚的看出，Lucene 的索引结构在概念上即为传统的倒排索引结构。

从概念上映射到结构中，索引被处理为一个目录（文件夹），其中含有的所有文件即为其内容，这些文件按照所属的段不同分组存放，同组的文件拥有相同的文件名，不同的扩展名。此外还有三个文件，分别用来保存所有的段的记录、保存已删除文件的记录和控制读写的同步，它们分别是 segments，deletable 和 lock 文件，都没有扩展名。每个段包含一组文件，它们的文件扩展名不同，但是文件名均为记录在文件 segments 中段的名字。让我们看如下的结构图 3.2。



关于图 3.2 中的各个文件具体的内部格式，在参考文献 3 中，均可以找到详细的说明。接下来我们从宏观关系上说明一下这些文件组成。在这些宏观上的关系理清楚之后，仔细阅读参考文献 3，即可清楚的明白具体的 Lucene 文件格式。

每个段的文件中，主要记录了两大类的信息：域集合与项集合。这两个集合中所含有的文件在图 3.2 中均有表明。由于索引信息是静态存储的，域集合与项集合中的文件组采用了一种类似的存储办法：一个小型的索引文件，运行时载入内存；一个对应于索引文件的实际信息文件，可以按照索引中指示的偏移量随机访问；索引文件与信息文件在记录的排列顺序上存在隐式的对应关系，即索引文件中按照“索引项 1、索引项 2…”排列，则信息文件则也按照“信息项 1、信息项 2…”排列。比如在图 3.2 所示文件中，segment1.fdx 与 segment1.fdt 之间，segment1.tii 与 segment1.tis、segment1.prx、segment1.frq 之间，都存在这样的组织关系。而域集合与项集合之间则通过域的在域记录文件（比如 segment1.fnm）中所记录的域记录号维持对应关系，在图 3.2 中 segment1.fdx 与 segment1.tii 中就是通过这种方式保持联系。这样，域集合和项集合不仅仅联系起来，而且其中的文件之间也相互联系起来。此外，标准化因子文件和被删除文档文件则提供了一些程序内部的辅助设施（标准化因子用在评分排序机制中，被删除文档是一种伪删除手段）。这样，整个段的索引信息就通过这些文档有机的组成。

以上所阐述的，就是 Lucene 所采用的索引文件格式。基本上而言，它是一个倒排索引，但是 Lucene 在文件的安排上做了一些努力，比如使用索引/信息文件的方式，从文件安排的形式上提高查找的效率。这是一种数据库之外的处理方法，其有其优点（格式平台独立、速度快），也有其缺点（独立性带来的共享访问接口问题等等）。

5　Lucene 中索引使用

本节将首先介绍一个给文本文件（*.txt）建立索引的简单例子，以便读者能对索引的建立过程以及所用到的 Lucene API 有一个了解。然后我们将详细介绍 Lucene 中索引使用和其注意事项。

5．1 建立索引的一个简单例子

在本小节中，将介绍 Lucene1.9 的 Demo 中一个名为 IndexFiles 的类和它的四个静态方法。它们共同递归遍历文件

系统目录并索引所有具有.txt 扩展名的文件。当 IndexFiles 执行完毕时，为它的后续 Searcher(在后面章节中会详细介绍)留下一个创建好的 Lucene 索引。

现在我们不必要熟悉例子中用到的几个 Lucene 类和方法－－我们将会简单地解释它们。在下面注释的代码列表之后，我们将介绍如何使用 IndexFiles 类。如果你认为先看 IndexFiles 使用方法对学习编码有利，可直接跳到代码后面的 IndexFiles 使用方法部分。

使用 IndexFiles 来索引文本文件

以下 IndexFiles 命令行程序。它用到一个参数：

- 包含要索引的文本文件的路径

注：IndexFiles 命令行程序，每次都是新建立索引，并把索引文件放在当前路径下的 index 文件夹中。IndexFiles 类在包 org.apache.lucene.demo 中

列表 Indexer：遍历文件系统并且索引.txt 文件

```
package org.apache.lucene.demo;


import org.apache.lucene.analysis.standard.StandardAnalyzer;

import org.apache.lucene.index.IndexWriter;


import java.io.File;

import java.io.FileNotFoundException;

import java.io.IOException;

import java.util.Date;


/** Index all text files under a directory. */

public class IndexFiles {


  private IndexFiles() {}


  static final File INDEX_DIR = new File("index");//索引的存放路径


  /** Index all text files under a directory. */

  public static void main(String[] args) {

    String usage = "java org.apache.lucene.demo.IndexFiles <root_directory>";
```

```
    if (args.length == 0) {
      System.err.println("Usage: " + usage);
      System.exit(1);
    }

    if (INDEX_DIR.exists()) {
      System.out.println("Cannot save index to '" +INDEX_DIR+ "' directory, please delete it first");
      System.exit(1);
    }

    final File docDir = new File(args[0]);
    if (!docDir.exists() || !docDir.canRead()) {
      System.out.println("Document directory '" +docDir.getAbsolutePath()+ "' does not exist or is not readable, please check the path");
      System.exit(1);
    }

    Date start = new Date();
    try {
      //① 创建 Lucene 索引
      IndexWriter writer = new IndexWriter(INDEX_DIR, new StandardAnalyzer(), true);
      writer.setUseCompoundFile(false);
      System.out.println("Indexing to directory '" +INDEX_DIR+ "'...");
      indexDocs(writer, docDir);
      System.out.println("Optimizing...");
      writer.optimize();
      writer.close();
      Date end = new Date();
      System.out.println(end.getTime() - start.getTime() + " total milliseconds");
    } catch (IOException e) {
```

```
            System.out.println(" caught a " + e.getClass() +
              "\n with message: " + e.getMessage());
        }
    }

    static void indexDocs(IndexWriter writer, File file)
        throws IOException {
        // do not try to index files that cannot be read
        if (file.canRead()) {
            if (file.isDirectory()) {
                String[] files = file.list();
                // an IO error could occur
                if (files != null) {
                    for (int i = 0; i < files.length; i++) {
                        indexDocs(writer, new File(file, files[i]));    //② 递归
                    }
                }
            } else if (file.getName().endsWith(".txt")) {
                System.out.println("adding " + file);
                try {
                    writer.addDocument(FileDocument.Document(file));//  添加片段到 Lucene 索引
                }
                // at least on windows, some temporary files raise this exception with an "access denied" message
                // checking if the file can be read doesn't help
                catch (FileNotFoundException fnfe) {
                }
            }
        }
    }
}
```

代码的大部分是执行目录的遍历(②)。只有 IndexWriter 的创建和关闭(①)和 indexDocs 方法中使用了 Lucene API。

这个示例只关注.txt 扩展名的文本文件是为了在说明 Lucene 的用法和强大功能时保持尽量简单。在后面章节中，我们介绍如何处理非文本文件，并且我们开发了一个现成的小框架来分析和索引几种常见的格式的文档。

运行 IndexFiles

在命令行中，我们针对包含 Lucene 本身的源文件的本地工作目录运行 IndexFiles。我们使 IndexFiles 索引 txtCorpus 目录下的文件并将 Lucene 索引保存在 index 目录中。

Indexing to directory 'index'...

adding txtCorpus\UNIX 命令详解.txt

adding txtCorpus\计算机类中文核心期刊.txt

Optimizing...

1402 total milliseconds

IndexFile 打印出索引的文件名称，你可以看出它只索引扩展名为.txt 的文本文件。

注意 如果你在 Windows 平台的命令行中运行这个程序，你需要调整命令行的目录和路径分割符。Windows 命令行是 java txtcorpus

当索引完成后，IndexFile 输出它索引的文件数目和所花费的时间。因为报告的时间包含文件目录遍历和索引，你不能把它做为一个正式的性能衡量依据。在我们的示例中，每个索引的文件都很小，但只有了不到 2 秒索引这些文件还是不错的。索引速度是要关注的，但是通常，搜索更加重要.

对于以上建立的索引,我们可以使用包 org.apache.lucene.demo 中的类 SearchFiles 进行搜索,下面简单介绍给类的命令行用法, 关于 Lucene 中的搜索我们将在后面章节介绍。 [4]

Usage: java org.apache.lucene.demo.SearchFiles [-index dir] [-field f] [-repeat n] [-queries file] [-raw] [-norms field]

参数含义:

[-index dir]: 设置索引存放的路径(默认当前路径下 index 文件夹)

[-field f] : 搜索的域 field ，如：上例中的 path，contents 域(默认为 contents 域)

[-repeat n]：主要是用于测试查询所用时间，n 次查询取平均时间

[-queries file]：设置查询词的文件名，如果不使用该参数，查询词从控制台输入

[-raw]：输出格式设置，如果设置该参数，则输出文档编号和分数

[-norms field]：

5.2 理解索引中的核心类

从上例 IndexFile 类中可以看到，我们需要以下类来执行这个简单的索引过程：

- IndexWriter

- Directory
- Analyzer
- Document
- Field

接下来是对这些类的一个简短的浏览，针对它们在 Lucene 的角色，给读者一个粗略的概念。

5.2.1 IndexWriter

IndexWriter 是在索引过程中的中心组件。这个类创建一个新的索引并且添加文档到一个已有的索引中。你可以把 IndexWriter 想象成让你可以对索引进行写操作的对象，但是不能让你读取或搜索。不管它的名字，IndexWriter 不是唯一的用来修改索引的类。

5.2.2 Directory

Directory 类代表一个 Lucene 索引的位置。它是一个抽象类，允许它的子类(其中的两个包含 在 Lucene 中)在合适时存储索引。在我们的 Indexer 示例中，我们使用一个实际文件系统目录的路径传递给 IndexWriter 的构造函数来获得 Directory 的一个实例。IndexWriter 然后使用 Directory 的一个具体实现 FSDirectory，并在文件系统的一个目录中创建索引。

在你的应用程序中，你可能较喜欢将 Lucene 索引存储在磁盘上。这时可以使用 FSDirectory，一个包含文件系统真实文件列表的 Driectory 子类，如同我们在 Indexer 中一样。另一个 Directory 的具体子类是 RAMDirectory。尽管它提供了与 FSDirectory 相同的接口，RAMDirectory 将它的所有数据加载到内存中。所以这个实现对较小索引很有用处，可以全部加载到内存中并在程 序关闭时销毁。因为所有数据加载到快速存取的内存中而不是在慢速的硬盘上，RAMDirectory 适合于你需要快速访问索引的情况，不管是索引或搜索。 做为实例，Lucene 的开发者在所有他们的单元测试中做了扩展使用：当测试运行时，快速的内存驻留索引被创建搜索，当测试结束时，索引自动销毁，不会在磁盘上留下任何残余。当然，在将文件缓存到内存的操作系统中使用时 RAMDirectory 和 FSDirectory 之间的性能差别较小。

5.2.3 Analyzer

在文本索前之前，它先通过 Analyzer。Analyzer 在 IndexWriter 的构造函数中指定，司职对文本内容提取关键词并除去其它的。如果要索引的内容不是普通的文本，首先要转化成文本。Analyzer 是个抽象类，但是 Lucene 中有几个它的实现。有的处理的时候跳过停用词；有的处理时把关键字转化为小写字母，所以这个搜索不是大小写敏感等等。Analyzer 是 Lucene 的一个重要的部分并且不只是在输入过滤中使用。对一个将 Lucene 集成到应用程序中的开发者来说，对 Analyzer 的选择在程序设计中是重要元素，并且通过对 Analyzer 的基础我们可以加入中文分词，后面章节将介绍。

5.2.4 Document

一个 Document 代表域的集合。你可以把它想象为以后可获取的虚拟文档—一块数据，如一网页、 一个邮件消息或一个文本文件。一个文档的域代表这个文档或与这个文档相关的元数据。文档数据的最初来源(如一条数据库记录、一个 Word 文档、一本书的 某一章等等)与 Lucene 无关。元数据如作者、标题、主题、修改日期等等，分别做为文档的域索引和存储。

注意　当我们在本书中提到一个文档，指一个 Microsoft Word、RTF、PDF 或其它文档类型；我们不是谈论 Lucene 的 Document 类。注意大小写和字体的区别。

Lucene 只用来处理文本。Lucene 的核心只能用来处理 java.lang.String 和 java.io.Reader。尽管多文档类型都能被索引并使之可搜索，处理它们并不像处理可以简单地转化为 java 的 String 或 Reader 类型的纯文本内容那样直接。这些在前面的主题信息的提取部分。

在以上的 IndexFile 中，我们处理文本文件，所以对我们找出的每个文本文件，创建一个 Document 类的实例，用 Field(域)组装它，并把这个 Document 添加到索引中，完成对这个文件的索引。

5.2.5 Field

在索引中的每个 Document 含有一个或多个域，具体化为 Field 类。每个域相应于数据的一个片段，将在搜索时查询或从索引中重新获取。

Lucene 提供四个不同的域类型，你可以从中做出选择：

- Keyword—不被分析，但是被索引并逐字存储到索引中。这个类型适合于原始值需要保持原样的域，如 URL、文件系统路径、日期、个人名称、社会安全号码、电话号码等等。例如，我们在 Indexer(列表 1.1)中把文件系统路径作为 Keyword 域。

- UnIndexed—不被分析也不被索引，但是它的值存储到索引中。这个类型适合于你需要和搜索结果一起显示的域(如 URL 或数据库主键)，但是你从不直接搜索它的值。因为这种类型域的原始值存储在索引中，这种类型不适合于存放比较巨大的值，如果索引大小是个问题的话。

- UnStored—和 UnIndexed 相反。这个域类型被分析并索引但是不存储在索引中。它适合于索引大量的文本而不需要以原始形式重新获得它。例如网页的主体或任休其它类型的文本文档。

- Text—被分析并索引。这就意味着这种类型的域可以被搜索，但是要小心域大小。如果要索引的数据是一个 String，它也被存储；但如果数据(如我们的 Indexer 例子)是来自一个 Reader，它就不会被存储。这通常是混乱的来源，所以在使用 Field.Text 时要注意这个区别。

所有域由名称和值组成。你要使用哪种域类型取决于你要如何使用这个域和它的值。严格来说，Lucene 只有一个域类型：以各自特征来区分的域。有些是被分析的，有些不是；有些是被索引，然面有些被逐字地存储等等。

表 1.2 提供了不同域特征的总结，显示了域如何创建以及基本使用示例。

表 1.2 不同域类型的特征和使用方法

Fied method/type	Analyzed	Indexed	Stored	Example usage
Field.Keyword(String,String) Field.Keyword(String,Date)		✔	✔	Telephone and Social Security numbers, URLs, personal names, Dates
Field.UnIndexed(String, String)			✔	Document type (PDF, HTML, and so on), if not used as search criteria

Field.UnStored(String,String)	✔	✔		Document titles and content
Field.Text(String,String)	✔	✔	✔	Document titles and content
Field.Text(String,Reader)	✔	✔		Document titles and content

注意所有域类型都能用代表域名称和它的值的两个 String 来构建。另外，一个 Keyword 域可以接受一个 String 和一个 Date 对象，Text 域接受一个 String 和一个 Reader 对象。在所有情况下，这些值在被索引之前都先被转化成 Reader，这些附加方法的存在可以提供比较友好的 API。

注意 注意 Field.Text(String, String)和 Field.Text(String, Reader)之间的区别。String 变量存储域数据，而 Reader 变量不存储。为索引一个 String 而又不想存储它，可以用 Field.UnStored(String, String)。

最后，UnStored 和 Text 域能够用来创建词向量(高级的话题，在 5.7 节中描述)。为了让 Lucene 针对指定的 UnStored 或 Text 域创建词向量，你可以使用 Field.UnStored(String, String, true)，Field.Text(String, String, true)或 Field.Text(String, Reader, true)。

在使用 Lucene 来索引时你会经常用到这几个类。为了实现基本的搜索功能，你还需要熟悉同样简单的几个 Lucene 搜索类。

5．3 Lucene 中索引操作

本节中我们将详细索引过程中所用到的操作

- 执行基本索引操作
- 在索引时添加 Document 和 Field
- 索引日期、数值和用来排序搜索结果的域
- 使用影响 Lucene 索引性能和资料消耗的参数
- 优化索引

你可能想搜索存储在硬盘上的文件或者搜索你的邮件、网页、数据库中的数据甚至建立一个 Web 搜索引擎。Lucene 能够帮助你。然页，在你能够搜索之前，我们应该对其进行索引。

本节将更深入讲解 Lucene 中有关索引更新、调整索引过程的参数和更多高级索引技术的使用，以帮助你更加了解 Lucene。此处你也会发现 Lucene 索引的结构、当以多线程或多进程访问 Lucene 索引时要注意的重要问题和 Lucene 提供的防止并发修改索引的锁机制。

5.3.1 理解索引过程

从前面的例子我们可以看到，索引一个文档只需调用 Lucene API 的几个方法。所以，表面看来，用 Lucene 进行索引是个简单的操作。然而在这些简单 API 的背后隐藏了一些有趣且相当复杂的操作集合。我们可以将这个集合分为三个主要的功能，如图 2.1 所示，在随后的几个小节中描述。

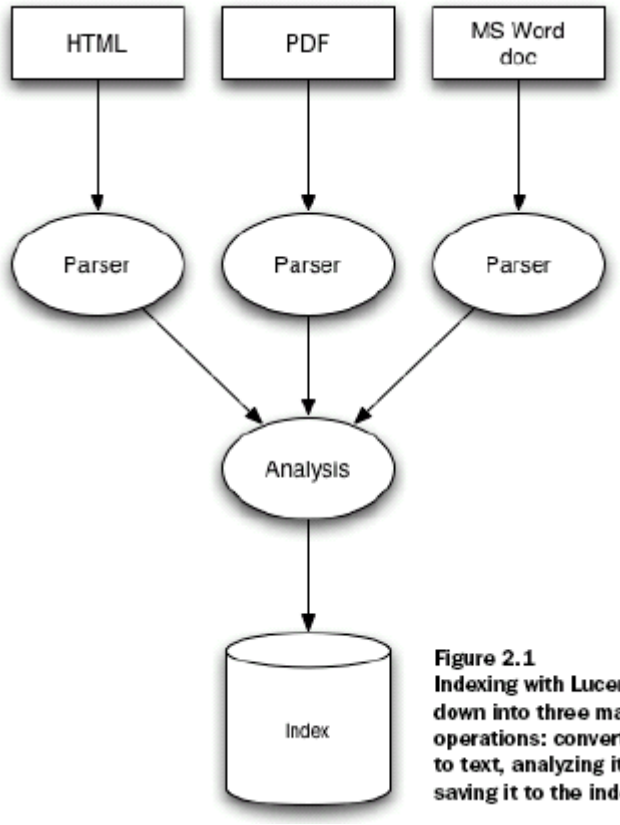


图 2.1 使用 Lucene 索引分为三个主要步骤：将数据转化为文本，分析，将它保存至索引

5.3.1.1 转化为文本

为用 Lucene 索引数据，你必须首先将它转化为纯文本词流，Lucene 能消化的格式。在前面例子中，我们限制示例索引和搜索.txt 文件，这允许我们分析它的内容并使它来生成 Field 的实例。然而，事情并不总是那么简单。

假设你要索引一些 PDF 格式的手册。为了准备这些手册以进行索引，你必须首先提取出 PDF 文档中的文本信息并使用这些提取的数据来创建 Lucene Document 及其 Field。你回顾 21 页的表 1.2，你会发现 Field 方法总是接受 String 值，有时是 Date 和 Reader 值。没有哪个方 法接受 PDF 的类型，既使这种类型存在。在索引 Microsoft Word 文档或其它任何非纯文本格式的文档时你都会遇到这个问题。甚至你在处理使用纯文本的符号的 XML 或 HTML 文档时，你仍然需要足够的智能来准备这 些数据来进行索引，避免索引类似 XML 元素或 HTML 标签的东西，而要索引这些文档中的真实数据。

文本提取的细节在第 7 章，我们构建了一个小但是完整的框架以索引图 2.1 所示的所有文档格式及其它几种格式。实际上，你会发现图 2.1 和图 7.3 很类似。

5.3.1.2 分析

一旦你准备好了要索引的数据并创建了由 Field 组成的 Lucene Document，你就可以调用 IndexWriter 的 addDocument(Document)方法，把你的数据送入 Lucene 索引。当你做这些 时，Lucene 首先分析这些数据以使它更适合索引。它将文本数据分割成块或单词，并执行一些可选择的操作。例如，单词可以在索引之前转化为小写，以保证搜索是大小写无关的。典型的，它也有可能排除转入中所有经常出现但无意义的词，例如英语终止词(a, an, the, in, on 等等)。类似的，通常分析输入单词以获取它的本质。

这个非常重要的步骤称为分析。Lucene 的输入能够以很多有趣且有用的方法进行分析，所以我们将在第 4 章详细分析这个过程。目前，把这个步骤想像为一个过滤器。

5.3.1.3 写索引

在输入被分析完后，就可以添加到索引中了。Lucene 将输入存储在一个反向索引的数据结构中。这个数据结构在允许快速关键字查询的同时有效地利用了磁盘空间。这个结构反向是因为它使用从输入中提取的单词做为查询键值而不是用处理的文档做为中枢入口。换句话说，代替尝试回答这个问题“这个文档中含有哪些单词？”，这个结构为提供快速回答“哪篇文档含有单词 X？”做了优化。

如果你想一下你常用的 Web 搜索引擎和你典型的查询格式，你会发现你想得到的精确的查询。当今所有的 Web 搜索引擎的核心都是反向索引。使得各搜索引擎不同的是一组严格保密的附加参数来改进这个索引结构。例如 Goolge 知名的级别(PageRank, PR)因素。Lucene 也有它自己的一套技术，你可以在附录 B 中学到其中一些。

5.3.2 基本索引操作

在前面章节中，我们看到了如何向索引中添加文档。但是我们将在此总结这个过程，同时描述删除和更新操作，以给你一个方便的参数点。

5.3.2.1 向索引中添加文档

为了总结已知的，让我们来看一下在本章中作为单元测试基类的代码片断。代码列表 2.1 创建一个复合的索引称做 index-dir(索引目录)，存储于系统临时目录：UNIX 的/tmp，或使用 Windows 的 C:\TEMP。(复合索引在附录 B 中描述)我们使用 SimpleAnalyzer 来分析输入文本，然后我们索引两个简单的 Document，每个都包含四种类型的 Field：Keyword、 UnIndexed、UnStored 和 Text。

列表 2.1 在基本测试类的每个测试之前准备一个新的索引

```
public abstract class BaseIndexingTestCase extends TestCase {
    protected String[] keywords = {“1”, “2”};
    protected String[] unindexed = {“Netherlands”, “Italy”};
    protected String[] unstored = {“Amsterdam has lots of bridges”, “Venice has lots of canals”};
    protected String[] text = {“Amsterdam”, “Venice”};
    protected Directory dir;

    protected void setUp() throws IOException {
      String indexDir =
        System.getProperty(“java.io.tmpdir”, “tmp”)   +
        System.getProperty(“file.separator”) + “index-dir”;
      dir = FSDirectory.getDirectory(indexDir, true);
```

```
        addDocuments(dir);
    }

    protected void addDocuments(Directory dir)
        throws IOException {
        IndexWriter writer = new IndexWriter(dir, getAnalyzer(), true);
        writer.setUseCompoundFile(isCompound());
        for (int i = 0; i < keywords.length; i++) {
            Document doc = new Document();
            doc.add(Field.Keyword("id", keywords[i]));
            doc.add(Field.UnIndexed("country", unindexed[i]));
            doc.add(Field.UnStored("contents", unstored[i]));
            doc.add(Field.Text("city", text[i]));
            writer.addDocument(doc);
        }
        writer.optimize();
        writer.close();
    }

    protected Analyzer getAnalyzer() {
        return new SimplyAnalyzer();
    }
    protected boolean isCompound() {
        return true;
    }
}
```

因为 BaseIndexingTestCase 类要被本章的其它单元测试类继承，我们将指出几个重要的细节。BaseIndexingTestCase 每次 setUp()方法调用时创建相同的索引。因为 setUp()在测试执行之前被调用，每个测试都是针对新创建的索引运行。尽管基类使用 SimpleAnalyzer，子类可以覆盖 getAnalyzer()方法以返回不同的 Analyzer 类型。

不同类型的 Document

Lucene 的一个重要特征是它允同一索引中存在不同类型的 Document（这里所说的不同类型的 Document 是指，Document 中的域可以不相同）。这就意味着你可以用一个索引来保存代表不同实体的 Document。例如，你可以存放代表零售产品，有名称和价格域的 Document 和代表 people，有名称、年龄和性别域的 Document。

附加域

假设你有个生成给定单词的同意词数组的程序，并且你想用 Lucene 来索引基本词和所有它的同意词。一个实现方法是遍历所有的同意词，并把它们添加到一个 String 中，然后你可以用它来创建 Lucene 域。索引所有同意词和基本词另一个可能更好的方法，是把不同的值添加到相同的域中，如下：

```
String baseWord = “fast”;

String synonyms[] = String {“quick”, “rapid”, “speedy”};

Document doc = new Document();

doc.add(Field.Text(“word”, baseWord));

for (int i = 0; i < synonyms.length; i++) {

    doc.add(Field.Text(“word”, synonyms));

}
```

其中，Lucene 添加所有的单词并把它们索引在同一个域 word 中，允许你在搜索时使用其中任何一个。

5.3.2.2 从索引中删除 Document

尽管大多程序关心的是添加 Document 到 Lucene 索引中，一些也需要清除它们。例如，报纸出版社可能只想在可搜索的索引中保留最近一个周的有价值的新闻。另外的程序可能想清除所有包含特定单词的 Document。

Document 的删除是由 IndexReader 来完成的。这个类并不立即从索引中删除 Document。它只做个删除的标志，等待 IndexReader 的 close()方法调用时真正的 Document 删除(注意这里说得删除只是做标记，而不是真正从物理上删除)。理解了这些之后，让我们看一下列表 2.2：它继承 BaseIndexingTestCase 类，这意味着在每次测试方法运行之前，基类重建两个文档的索引，在 2.2.1 小节中描述。

列表 2.2 根据内部文档号删除 Document

```
public class DocumentDeleteTest extends BaseIndexingTestCase {

    public void testDeleteBeforeIndexMerge() throws IOException {

        IndexReader reader = IndexReader.open(dir);

        assertEquals(2, reader.maxDoc());          ① 下一个 Document 号是 2

        assertEquals(2, reader.numDocs());   ② 索引中有两个 Document

        reader.delete(1);    ③ 删除号码为 1 的 Document

        assertTrue(reader.isDeleted(1));        ④ 删除 Document

        assertTrue(reader.hasDeletions());          ⑤ 包含删除的索引

        assertEquals(2, reader.maxDoc());          ⑥ 1 个索引的 Document，下一个 Document 号是 2
```

```
        reader.close();
        reader = IndexReader.open(dir);

        assertEquals(2, reader.maxDoc());           ⑦ 在 IndexReader 重新打开后，
        assertEquals(1, reader.numDocs());       下一个 Document 号是 2
        reader.close();
//由以上测试知,用 IndexReader 中的 delete 方法删除 Document,只是在 deletable 文件中做一个标记
    }

    public void testDeleteAfterIndexMerge() throws IOException {
        IndexReader reader = IndexReader.open(dir);
        assertEquals(2, reader.maxDoc());
        assertEquals(2, reader.numDocs());
        reader.delete(1);
        reader.close();

        IndexWriter writer = new IndexWriter(dir, getAnalyzer(), false);
        writer.optimize();
        writer.close();
        reader = IndexReader.open(dir);
        assertFalse(reader.isDeleted(1));
        assertFalse(reader.hasDeletions());           ⑧ Optimizing
        assertEquals(1, reader.maxDoc());              renumbers
        assertEquals(1, reader.numDocs());       Documents
         reader.close();
//由以上测试可知,删除文档后如果做优化(使用 optimize()方法),则被删除的文档才真正从物理上清除
    }
}
```

注: 使用 Junit 测试时,在每执行一次前,都得先执行一次 setUp 方法,即使没有 setUp 方法,前面一个测试方法如果改变了全局变量得值,也不会改变全局变量在下一个测试方法中的值.

①②③ 列表 2.2 的代示展示了如何指定 Document 的内部编号来删除 Document。它也展示了 IndexReader 经常混淆的两个方法的不同： maxDoc()和 numDocs()。前者返回下一个可用的内部 Document 号，后者返回索引中的 Document 的数目。因为我们的索引只含有两 个 Document，numDocs()返回 2；又因为 Document 号从 0 开始，maxDoc()也返回 2。

注意 每个 Lucene 的 Document 有个唯一的内部编号。这些编码不是永久分配的，因为 Lucene 索引分配时在内部重新分配 Document 的编号。因此，你不能假定一个给定的 Document 总是拥有同一个 Document 编号。

④⑤ 在 testDeleteBeforeIndexMerge()方法中的测试也示范了 IndexReader 的 hasDeletions()方法以检查一个索引是否包含有删除标志的 Document 和 isDeleted(int)方法以检查指定编号的 Document 的状态。

⑥⑦ 可见，numDocs()能够立即感知到 Document 的删除，而 maxDoc()不能。

⑧ 此外，在 testDeleteAfterIndexMerge()方法中，我们关闭 IndexReader 并强制 Lucene 优化索引以合并索引的各片断。然后我们用 IndexReader 打开索引，maxDoc()方法返回 1 而不是 2，因为在删除和合并后，Lucene 对剩余的 Document 重新编号。索引中只有一个 Document，所以下一下可能 Document 编号是 1。

除了我们通过指定 Document 编号来删除单个 Document 之外，你可以用 IndexReader 的 delete(Term)方法删除多个 Document。使用这个删除方法，允许你删除所有包含指定 Term 的 Document。例如，为了删除 city 域中包含单词 Amsterdam 的 Document，你可以这样用 IndexReader：

```
IndexReader reader = IndexReader.open(dir);

reader.delete(new Term(“city”, “Amsterdam”));

reader.close();
```

在使用这个方法时要特别小心，因为在所有索引的 Document 中指定一个 term 将会擦除整个索引。这个方法的使用类似于基于 Document 编号的删除方法。

你可能奇怪为什么 Lucene 在 IndexReader 中执行 Document 删除而不是 IndexWriter 中。这个问题在 Lucene 社区中每几个月就问一次，大概因为有缺点或者可能是容易让人误解的类名。Lucene 的用户经常认为 IndexWriter 是唯一可以修改索引的类，且 IndexReader 以只读的形式访问索引。实际上，IndexWriter 只接触索引片断列表和合并片断时的一小部分索引文件。另一方面，IndexReader 知道如何解析所有索引文件。当一个 Document 删除时，IndexReader 在标记它被删除之前首先需要定位包含指定 Document 的片断。目前还没有计划改变这两个 Lucene 类的名称或行为。

5.3.2.3 恢复 Document

因为 Document 的删除延迟到 IndexReader 实例关闭时才执行，Lucene 允许程序改变想法并恢复已做删除标记的 Document。对 IndexReader 的 undeleteAll()方法的调用通过清除索引目录中的.del 文件来恢复所有删除的 Document。所以在关闭 IndexReader 实例关闭之后 Document 就保留在索引中了。只能使用与删除 Document 时同一个 IndexReader 实例，才能调用 undeleteAll()来恢复 Document。

5.3.2.4 更新索引中的 Document

“如何才能更新索引中的文档？”是一个在 Lucene 用户邮件列表中经常问的问题。Lucene 并没有提供更新方法；Document 必须首先从索引中删除然后再重新添加它，如列表 2.3 所示。

列表 2.3 通过删除再添加的方法更新索引的 Document

```
public class DocumentUpdateTest extends BaseIndexingTestCase {
```

```
public void testUpdate() throws IOException {
    assertEquals(1, getHitCount("city", "Amsterdam"));
    IndexReader reader = IndexReader.open(dir);
    reader.delete(new Term("city", "Amsterdam"));
    reader.close();

    assertEquals(0, getHitCount("city", "Amsterdam"));
    IndexWriter writer = new IndexWriter(dir, getAnalyzer(), false);
    Document doc = new Document();
    doc.add(Field.Keyword("id", "1"));
    doc.add(Field.UnIndexed("country", "Netherlands"));
    doc.add(Field.UnStored("contents",
       "Amsterdam has lots of bridges"));
    doc.add(Field.Text("city", "Haag"));
    writer.addDocument(doc);
    writer.optimize();
    writer.close();
    assertEquals(1, getHitCount("city", "Haag"));
}
protected Analyzer getAnalyzer() {
    return new WhitespaceAnalyzer();
}
private int getHitCount(String fieldName, String searchString) throws IOException {
    IndexSearcher searcher = new IndexSeracher(dir);
    Term t = new Term(fieldName, searchString);
    Query query = new TermQuery(t);
    Hits hits = searcher.search(query);
    int hitCount = hits.length();
    searcher.close();
    return hitCount;
```

```
    }
}
```

我们首先删除了 city 域含有 Amsterdam 的所有 Document；然后添加一个域与删除的 Document 相同的新 Document，除了把 city 域设了一个新值。新的 Document 的 city 域是 Haag 而不是 Amsterdam。 我们正确地更新了索引中的一个 Document。

通过定量删除来更新

我们的例子删除和添加单个 Document。如果你需要删除和添加多个 Document，最好是进行批操作。按以下步骤：

1. 打开 IndexReader。
2. 删除所有你要删除的 Document。
3. 关闭 IndexReader。
4. 打开 IndexWriter。
5. 添加你要添加的所有 Document。
6. 关闭 IndexWriter。

要记住：批量 Document 删除和索引总是比交叉删除和添加操作快。

懂得了更新和删除操作，让我们讨论如何提升索引的性能并尽可能好地利用硬件资源。

技巧 :当删除和添加 Document 时，批量进行。这样总是要比交叉删除和添加操作快。

5.3.2.5 Document 和 Field 权重

并不是所有的 Document 和 Field 是平等创建的——或者至少你能确定选择性的 Document 或 Field 权重的情况。假设你要写一个索引和搜索公司 Email 的程序。可能需求是给公司员工的 Email 比其它 Email 消息更多的重要性。你会如何做呢？

Document 权重是个使得这种需求能够简单实现的一个特征。默认情况下，所有的 Document 都没有权重—— 或者更恰当地说，它们都有相同的增量因数 1.0。通过改变某个 Document 的增量因数，你可能让 Lucene 认为它比索引中的其他 Document 更 重要或不重要。执行这些的 API 只需一个方法，setBoost(float)，可以这样用：

```
public static final String COMPANY_DOMAIN = “example.com”;

public static final String BAD_DOMAIN = “yucky-domain.com”;


Document doc = new Document();

String senderEmail = getSenderEmail();

String senderName = getSenderName();

String subject = getSubject();

String body = getBody();
```

```
doc.add(Field.Keywork(“senderEmail”, senderEmail));
doc.add(Field.Text(“senderName”, senderName));
doc.add(Field.Text(“subject”, subject));
doc.add(Field.UnStored(“body”, body));
if (getSenderDomain().endsWithIgnoreCase(COMPANY_DOMAIN)) {
    doc.setBoost(1.5);        ① 员工增量因数：1.5
} else if (getSenderDomain().endsWithIgnoreCase(BAD_DOMAIN)) {
    doc.setBoost(0.1);        ② Bad 域增量因数：0.1
}
writer.addDocument(doc);
```

在本例中，我们检查邮件域名来决定发件人是不是公司的员工。

① 当我们索引由公司员工发送的消息时，我们把他们的增量因数设为 1.5，这默认的因数 1.0 大。

② 当我们碰到来自虚构的不可信域的发件人发送的消息时，我们通过把它们的增量因数隐为 0.1 把它们归类为不重要的。

就象你可以增量 Document 一样，你也可以增量个别的域。当你增量 Document 时，Lucene 内部使用相同的增量因数增量它的每个域。假设 Email 索引程序的另一个需求是考虑标题域比发件人的名称域更重要。换句话说，搜索时对标题域的匹配比同样 对 senderName 域的匹配更有价值。为了完成这个计划，我们使用 Field 类的 setBoost(float)方法：

```
Field senderNameField = Field.Text(“senderName”, senderName);
Field subjectField = Field.Text(“subject”, subject);
subjectField.setBoost(1.2);
```

在本例中，我们随便选了一个增量因数 1.2，就像我们随便为 Document 选了一个增量因数 1.5 和 0.1 一样。你要用的增量因数值取决于你要达到什么目的；你可能需要做一些实验和调整来达到预期的目标。

值得注意的是域可以有和它们相关联的固定增量，是由于 Lucene 的算分方式。增量总得来说是个高级特征，没有它很多程序也能工作得很好。

Lucene 的搜索结果根据每个 Document 与查询的接近程度来分级，每个匹配的 Document 分被赋于一个分值。Lucene 的评分规则受许多因素影响，增量因数是其中之一。

5.3.2.6 索引日期

邮件含有发送和接收日期，文件有很多相关的日期，HTTP 呼应有一个包含请求页面最后修改日期 Last- Modified 头。像很多其他 Lucene 用户一样，你可能需要索引日期。Lucene 带有一个 Field.Keyword(String, Date)方法，还有 DateField 类，这使得索引日期很简单。例如，为了索引当前日期，可以这样：

```
Document doc = new Document();
doc.add(Field.Keyword(“indexDate”, new Date()));
```

在内部，Lucene 使用 DateField 类把给定的日期转化成适于索引的字符串。这样处理日期比较简单，但是你在使用这个方法时必须小心：使用 DateField 把日期转化成可以索引的 String，包括日期的所有部分，甚至毫秒。你将在 6.5 节中看到，这可能会导致某些查询的性能问题。实际上，你很少需要精确到毫秒的日期，至少对查询来说是这样。通常，你可以把日期近似到小时或着甚至天。

因为所有 Field 值最后都会转化成文本，你可以像 String 一样索引日期。例如，如果你把日期近似到天，以 YYYYMMDD 格式的 String 索引日期，可以使用 Field.Keyword(String, String)方法。运用这种方法的原因是你能够索引在 Unix Epoch(1970 年 1 月 1 日)之前的日期，DateField 不能处理。尽管一些解决这个限制的修补在近几年来被很多人提出，但是没有一个很完美。所以他们只能在 Lucene 补丁列表中找到，而没有包含在 Lucene 中。根据 Lucene 用户提到这个限制的频率，不能索引 1970 年之前的日期通常不是个问题。

注意:如果你仅需要对日期进行搜索，而不是时间，用 Field.Keyword(“date”, “YYYYMMDD”)。如果你要取得完整的时间，用 Field.Keyword(“timestamp”, <java.util.Date>)索引另一个 Field。

如果你想把日期或时间格式化为其它形式，一定注意 String 是以字典顺序排序的；这样做允许期间(date- range)查询。以 YYYYMMDD 格式索引日期的一个好处是能够仅用年来查询，或用年和月，或者精确地年和月和日。仅用年查询，使用 PrefixQuery。我们将在后面的章节中深入讨论 PrefixQuery。

5.3.2.7 索引数值

有两种常见的情况，数值索引非常重要。第一种，数值包含在要被索引的文本中，并且你想确定这些数值被索引了，这样你就能在后来的搜索中使用它们。例如，你的文档可能包含“珠穆朗玛峰高 8848 米。”你想像搜索“珠穆朗玛峰”那样搜索数字“8848”,同样可以检索到包含这个句子的 Document。

第二种情况，有只含有数值的 Field，并且你想索引和搜索它们。此外，你可能想利用这些 Field 来执行范围内搜索(range queries)。例如，如果你要索引邮件消息，有个域保存邮件大小，你可能想找出所有指定大小的消息；或者，你可能想使用范围搜索来找出所有大小在一 个特定范围内的消息。你可能也要根据大小来排序结果。

Lucene 能够通过在内部把它们处理成字符串来索引数值。如果你需要索引自由格式文本中的数值，你要做的第一件事就是选择不过滤数字的 Analyzer。在后面章节中我们将讨论到的，WhiteSpaceAnalyzer 和 StandardAnalyzer 是两个可能的选择。如果你传递给他一个类似“珠穆朗玛峰高 8848 米”的句子，他们把 8848 提取为一个单词，并把它传到索引过程，允许稍后搜索 8848。而 SimpleAnalyzer 和 StopAnalyzer 将把数字从单词流中分离出来，这意味着搜索” 8848”不会有任何匹配的文档。

5.3.2.8 限制 Field 的长度

在进行索引的时候，我们有可能需要对某个域的长度进行限制。例如：1、用户用户为一篇 100 万字的文章建立索引，可能文章后面 10 万字都是附录；2、我们在索引网页的 meta 域时，有些网站管理员为了提高网站的点击率，从而欺骗搜索引擎在 meta 中加入了很多与网页内容无关得热门关键字。由于这样种种情况，我们有必要对索引域的长度进行限制。另外，有时候为了提高索引和查询的效率，我们会主动放弃一部分内容，此时也需要限制索引域的长度。

maxFieldLength：对 field 的长度(term 的数量)进行限制

maxFieldLength 可以在任何时刻设置， 设置后，接下来的 index 的 Field 会按照新的 length 截取，之前已经 index 的部分不会改变。可以设置为 Integer.MAX_VALUE

5.3.2.9 为查询优化索引(index)

Indexwriter.optimize()方法可以为查询优化索引（index），下面要讲到的性能调整是为 indexing 过程本身优化，而这里是为查询优化，优化主要是减少 index 文件数，这样让查询的时候少打开文件，优化过程中，lucene 会拷贝旧的 index 再合并，合并完成以后删除旧的 index，所以在此期间，磁盘占用增加， IO 操作也会增加，在优化完成瞬间，磁盘占用会是优化前的 2 倍,在 optimize 过程中可以同时作 search。

5.3.2.10 索引格式转换（compand － simple）

Lucene 中索引文件提供两种格式，复合索引格式和非复合索引格式（版本 1.4 后默认为复合索引）格式。复合索引格式可以减少打开文件得数目，但索引中如“段”、“文档”等概念还是一样的。

转化方法：

1. 设置索引格式：IndexWriter 中的 setUseCompoundFile(Boolean)方法

2．索引优化：IndexWriter 中的 optimize()方法

如把一个索引转化成非复合索引格式(关键语句)：

IndexWriter. setUseCompoundFile(false);

IndexWriter. optimize();

5.3.2.11 索引存放的位置

Lucene 中索引既可以存放在磁盘中也可以存放在内存中，具体存放位置依应用而定。

5.3.2.12 索引同步和 locking 机制

所有只读操作都可以并发

- 在 index 被修改期间，所有只读操作都可以并发
- 对 index 修改操作不能并发，一个 index 只能被一个线程占用
- index 的优化，合并，添加都是修改操作
- IndexWriter 和 IndexReader 的实例可以被多线程共享，他们内部是实现了同步，所以外面使用不需要同步

lucence 内部使用文件来 locking， 默认的 locking 文件放在 java. io. tmpdir, 可以通过 -Dorg. apache. lucene. lockDir=xxx 指定新的 dir，有 write. lock commit. lock 两个文件，lock 文件用来防止并行操作 index，如果并行操作， lucene 会抛出异常，可以通过设置-DdisableLuceneLocks=true 来禁止 locking，这样做一般来说很危险，除非你有操作系统或者物理级别的只读保证，比如把 index 文件刻盘到 CDROM 上。

5.3.2.13 性能调整

性能调整参数:IndexWriter 实例的属性

mergeFactor:控制把索引从内存写入磁盘时内存最大的 Document 数,同时控制内存中最 大的 Segment 数,默认为 10

maxMergeDocs:限制一个 Segment 中最大的文档数,大批量建立索引时应该设大,增量时　　应设小值

IndexWriter 提供了一些参数可供设置，列表如下

	属性	默认值	说明
mergeFactor	org.apache.lucene.mergeFactor	10	设置一个索引中子索引(段)的最大个数和每个子索引中文档的最大个数，超过文档的个数则写入下一个新段,超过子索引的个数则把子索引合并到一个新的子索引中
maxMergeDocs	org.apache.lucene.maxMergeDocs	Integer.MAX_VALUE	限制一个段中的 document 数目
minMergeDocs	org.apache.lucene.minMergeDocs	10	缓存在内存中的 document 数目，超过他以后会写入到磁盘
maxFieldLength		1000	一个 Field 中最大 Term 数目，超过部分忽略，不会 index 到 field 中，自然也就搜索不到

这些参数的的详细说明比较复杂：mergeFactor 有双重作用

设置每 mergeFactor 个 document 写入一个段，比如每 10 个 document 写入一个段

设置每 mergeFacotr 个小段合并到一个大段，比如 10 个 document 的时候合并为 1 小段，以后有 10 个小段以后合并到一个大段，有 10 个大段以后再合并，实际的 document 数目会是 mergeFactor 的指数

简单的来说 mergeFactor 越大，系统会用更多的内存，更少磁盘处理，如果要打批量的作 index，那么把 mergeFactor 设置大没错， mergeFactor 小了以后， index 数目也会增多，searhing 的效率会降低， 但是 mergeFactor 增大一点一点，内存消耗会增大很多(指数关系),所以要留意不要"out of memory"
把 maxMergeDocs 设置小，可以强制让达到一定数量的 document 写为一个段，这样可以抵消部分 mergeFactor 的作用.
minMergeDocs 相当于设置一个小的 cache,第一个这个数目的 document 会留在内存里面，不写入磁盘。这些参数同样是没有最佳值的，必须根据实际情况一点点调整。

5.3.2.8 索引合并

我们可能在不同时间和地点创建了多个索引，可能需要对这些索引进行合并。下面我们介绍 Lucene 中是如何对多个索引进行合并的。

方法：IndexWriter 类中的 addIndexes()

关键语句：

IndexWriter1(IndexWriter 实例)，IndexReaders（IndexReader 实例数组）,dirs(Directory 数组)为打开索引的两个 IndexWriter 类

IndexWriter1.addIndexes(IndexReaders)或 IndexWriter1.addIndexes(dirs)

通过以上方法就可以把 IndexReaders 或 dirs 中的索引都添加到 IndexWriter1 中

5.3.2.9 排序用的索引域

当我们进行查询时，默认情况下查询结果是按相关度排序的。但是有时，我们的应用需要按照其他标准返回结果。例如，当我们查询email信息时，可能需要按照收发邮件的时间排序，或者根据email信息的大下排序。如果我们想以某个Field的值对查询结果排序，那么该Field必须被索引且不能被分词（tokenized）（如`Field.Keyword`）。另外排序用的Field还必须能被转化成`Integer`,`Float`, or `String`等类型。

```
Field.Keyword("size", "4096");

Field.Keyword("price", "10.99");

Field.Keyword("author", "Arthur C. Clark");
```

尽管我们把数值都是以字符形式索引的，但可以指定其正确的Field类型（如`Integer`型或`Long`型）

注意：排序用的Field必须被索引且不能被分词（tokenized），还必须能被转化成`Integer`,`Float`, or `String`等类型。

第五章 字符分析器

在建立索引和检索过程中，分析是很重要的一个环节。Lucene 使用分析器（Analyzer）来对各种各样的输入进行分析，可以说 Analyzer 在 Lucene 开发包中占有举足轻重的地位，它的运行性能和分析能力直接影响到搜索到搜索引擎的许多环节。

本书在 Lucene 的开发包中已经为开发人员内建了数种分析器，本章中将向大家一一介绍。同时，在本章的最后，还将向读者介绍如何制定自己的分析器和过滤器。这些内容都是使用 Lucene 的精华所在，相信大家在阅读完本章内容后，再加上自己的不断钻研，将会逐渐成为一位使用 Lucene 的高手。

本章的主要内容包括：

- Lucene 分析器 Analyzer
- JavaCC 与 Tokenizer
- Lucene 内建的分析器
- 定制自己的分词器和过滤器

5.1 Lucene 分析器——Analyzer

什么是分析器，分析器到底如何分析？Lucene 中都有哪些分析器呢？下面将对这些问题逐一进行介绍。

5.1.1 Analyzer 的概述

Analyzer 中文可以翻译成“分析器”，是 Lucene 中内置的一种工具。它主要用于分析搜索引擎中遇到的各种文本。所谓分析，用更具体的话说其实就是“分词”和“过滤”。

从图 5-1 种可以看出，分析器位于索引和文本资源之间，这样所有进入索引库的文本资源都应当经过分析器的分析，以此来控制索引中的内容。未经过分析器分析的文本如果直接进入索引，可能会引发各种各样数据的一致性问题，同时会降低索引的效率，进而影响整个搜索引擎的性能。

正如前面分析所述，在 Lucene 中，一个分析其主要包括分词器和过滤器两种组件。分词器就是用于对文本资源进行切分，将文本按规则切分为一个个可以进行索引的最小单位。而过滤器的功能则是对这种最小单位进行预处理，比如大写转小写，复数转单数等。这种操作可以简单（如最简单的大写转小写），也可以相当复杂（如根据语义改写拼写错误的单词）。有关分词器和过滤器的具体实现，在本章后面会有详细说明，这里只介绍 Lucene 是如何实现 Analyzer 的。

在 Lucene 中，所有的 Analyzer 均继承自 org.apache.lucene.analysis.Analyzer 这个基类，该基类并没有什么特殊之处，它是一个抽象类。下面以代码 5.1 为例对 Analyzer 进行介绍。

代码 5.1　Analyzer.java

```
public abstract class Analyzer {
  public TokenStream tokenStream(String fieldName, Reader reader) {
    return tokenStream(reader);
  }
  public TokenStream tokenStream(Reader reader) {
    return tokenStream(null, reader);
  }
}
```

从上述代码中可以看出，在这个基类中实际上只定义了两个方法，这两个方法并没有实际意义，而只是提供一个供子类扩展的机制，其中，tokenStream（Reader reader）的方法已经不被推荐使用。通常情况下，都需要继承 Analyzer 类以实现更为强大的分析功能。

5.1.2 分词器（Tokenizer）和过滤器（TokenFilter）

在前面已经介绍了 Lucene 分析器的主要作用就是对传入的文本进行切分和过滤。在 Lucene 中，分词器和过滤器分别是通过 Tokenizer 和 TokenFilter 类的字类来实现。下面以代码 5.2 和代码 5.3 为例对 Tokenizer 和 TokenFilter 进行介绍。

代码 5.2　Tokenizer.java

```
public abstract class Tokenizer extends TokenStream{
  protected Reader input;
  protected Tokenizer(){
}
  protected Tokenizer(Reader input){
    this.input=input;
  }
  public void close() throws IOException {
    input.close();
  }
}
```

代码 5.3　TokenizerFilter.java

```
public abstract class TokenFilter extends TokenStream{
  protected TokenStream input;
  protected TokenFilter(){
  }
  protected TokenFilter(TokenStream input){
    this.input=input;
  }
  public void close() throws IOException{
    input.close();
  }
}
```

其实，从分析器功能角度来看，其本身并不能做任何事，所有工作都是依赖于分词器和过滤器完成的。一个分析

器所有的工作就是将分词器和过滤器进行合理的组合，使之产生对文本分词和过滤的效果。因此，分析器使用分词器和过滤器构成了一个管道，文本在“流过”这个管道后，就成为了可以进入索引的最小单元。这也是软件体系架构中管道过滤器模式在具体实现层面的一个经典运用。在后面的 StandardAnalyzer 一小节中，还要详细讲述这一部分的知识。

作为分析器的基类，Analyzer 只提供了两个方法，它们的方法类型定义都返回一个 TokenStream 的对象，那么，这个 TokenStream 类型是什么呢？仔细研究 Lucene 的 API，就可以知道，TokenStream 类其实是 Lucene 中所有分词器和过滤器的基类。下面以代码 5.4 为例对 TokenStream 进行介绍。

代码 5.4　TokenStream.java

```
public abstract class TokenStream{
  public abstract Token next() throws IOException;
  public void close() throws IOException{
  }
}
```

代码 5.4 中，只定义了 next()和 close()两种方法。其实这两个操作的意图很好理解。因为分析器处理的就是文本流，从 Tokenizer 和 TokenFilter 得源码中可以看出，Reader 型的类变量在构造函数中就被赋予了初始值。从字面上看，close 操作正好要关闭流，而 next 操作却是要从流中取出一个合理的词条。因此，Lucene 的开发人员定义这样的基类抽象汉书确实是独具匠心。

5.1.2 使用 StandardAnalyzer 进行测试

在上一小节中了解到，要使用 Lucene 的分析机制，就需要扩展 Lucene 的分析器基类。Lucene 为我们提供了一个相当标准的分析器——StandardAnalyzer。通过这个类，可以很清楚地了解到分析器的功能和用法。本小节就通过一段示例代码来为读者演示这个分析器的功能。

StandardAnalyzer 是 Lucene 开发包中内置的一种 Analyzer 的实现，可以将理解成“标准分析器”，这个分析器是最容易使用也是使用最频繁的一种 Analyzer 的实现，它使用了 Lucene 内部自带的几种分词器和过滤器，在这里不再详细解释它的内部结构，而仅演示功能，具体内容将在后面的小节中介绍。

下面以代码 5.5 为例来介绍 StandardAnalyzer 类最基本的使用方法。

代码 5.5　StandardAnalyzerTest.java

```
package IRLabDemo.analysis;
import java.io.IOException;
import java.io.StringReader;
```

```
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.SimpleAnalyzer;
import org.apache.lucene.analysis.StopAnalyzer;
import org.apache.lucene.analysis.Token;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
public class StandardAnalyzerTest {
  //构造函数
  public StandardAnalyzerTest() {  }
  public static void main(String args[]) {
    //初始化一个 StandardAnalyzer 对象
    Analyzer analyzer = new StandardAnalyzer();
    //测试字符串
    StringReader sr = new StringReader(
        "People are always talking about 'the problem of youth'");
    //生成 TokenStream 对象
    TokenStream ts = analyzer.tokenStream(sr);
    try {
      int i = 0;
      //调用 next()方法不断取得下一个切出的词
      Token t = ts.next();
      while (t != null) {
        //辅助输出时显示行号
        i++;
        //输出处理后的字符
        System.out.println("Line" + i + ":" + t.termText());
        //取得下一个切出的词
        t = ts.next();
      }
    }
    catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

在代码 5.5 中，进入分析器的测试数据为“People are always talking about 'the problem of youth'.”。

代码首先构造一个 StandardAnalyzer 类型的对象 analyze，然后通过 StandardAnalyzer 的 tokenStream 方法将测试数据从 StringReader 类型转换成一个 TokenStream 类型的对象 ts，也就是一个“词流”，最后从 ts 中循环取出处理后的字符串。

代码的运行结果如下所示：

```
Line1:people
Line2:always
Line3:talking
Line4:about
Line5:problem
Line6:youth
```

从上面结果中可以看出，测试数据被“切割”成了一个个的单词，并且测试数据中的单词还“消失”了。

通过代码 5.5 和代码运行结果，对照比较输入字符串和输出字符串，可以发现有以下几种不同。

- 对原有句子按照空格进行了分词。
- 所有的大写字母都转换成了小写字母。
- 去除掉了“are”、“of”、“the”等单词。
- 删除了所有的标点符号。

其中，第一项说明分析器已经对输入的文本流进行了切词，第二项说明切出来的词经过了一个由大写转小写的过滤器，第三项说明切出的词经过了一个“停止词”的过滤器。所谓停止词，指的就是如英文中的 a、an、the 这样的小词，它们在建立索引的过程中并无实际的意义，因此需要被过滤掉，不能被加入索引中。

当然，除了对英语处理外，StandardAnalyzer 还可以对中文进行处理（虽然效果不是很理想，下面以代码 5.6 为例进行介绍）。

代码 5.6　StandardAnalyzerTestForCH.java

```
package IRLabDemo.analysis;
import java.io.IOException;
import java.io.StringReader;
import org.apache.lucene.analysis.Analyzer;
```

```
import org.apache.lucene.analysis.SimpleAnalyzer;
import org.apache.lucene.analysis.StopAnalyzer;
import org.apache.lucene.analysis.Token;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
public class StandardAnalyzerTestForCH{
  //构造函数
  public StandardAnalyzerTestForCH(){  }
  public static void main(String args[]){
    //初始化一个 StandardAnalyzer 对象
    Analyzer a=new StandardAnalyzer();
    //测试字符串
    StringReader sr=new StringReader("龙门石窟位于山西省大同市西郊,是我国古代艺术的瑰宝!");
    //获取"词流"
    TokenStream ts=a.tokenStream(sr);
    try{
      int i=0;
      //取出下一个切分好的词
      Token t=ts.next();
      while(t!=null){
        i++;
        //打印
        System.out.print(t.termText()+" ");
        //继续获取
        t=ts.next();
      }
    }
    catch(IOException e){
      e.printStackTrace();
    }
  }
```

}

由上述的代码可以看出流程和处理英文字符串是一样，只是将输入的字符串换成了“龙门石窟位于山西省大同市西郊，是我国古代艺术的瑰宝!”，代码的运行结果如下所示：

龙 门 石 窟 位 于 山 西 省 大 同 市 西 郊 是 我 国 古 代 艺 术 的 瑰 宝

从上述结果可以看出，输入的测试数据“龙门石窟位于山西省大同市西郊，是我国古代艺术的瑰宝!”被按照单字进行了“切割”，并且去掉了句子中的标点符号。

在该过程中虽然字索引也是一种流行的索引方式，对于一些要求并不是很高的简单电子商务应用来说已经足够使用，但是字索引也存在一些明显的缺点，因为字索引可能会导致索引的内容过于庞大，而且对于索引单条目中的内容也可能过于冗长，导致建立索引和检索效率的降低。

5.2 JavaCC 与 Tokenizer

对于任意的文本，其中的信息多种多样，有不同的语言文本，还有数字、标点符号、空格等。要从其中切分出有意义的文本单位，并不是一件容易的事。分词器不仅仅要对语言进行妥善的处理，针对数字、标点符号、空格和一些格式标记等都要仔细斟酌。因此，这项工作的代码写起来相当繁琐，要进行字符串的匹配，复杂的递归算法，回溯，多种情况的处理。那么 Lucene 到底是如何实现它的标准分词器的呢？

5.2.1 JavaCC 简介

学过编译原理的人都知道，有一个著名的词法分析工具 YACC（Yet Another Compiler Compiler），它是 AT&T 为了构建 C 语言和其他高级语言解析器而开发的一个基于 C 语言的工具。YACC 和词法记号赋予器 Lex 接收由常用的巴科斯-诺尔范式形式的语言定义的输入，并生成了一个 C 语言的程序，用以对在范式中所定义语言的输入进行词法分析和相关操作。用户只要通过范式定义好某种语言，该工具就会自动生成它的解析工具，换句话说，也就是生成了一个该语言的生成器。

当这种思想应用到 Java 的领域时，就出现了 JavaCC。JavaCC 的英文全称为“Java Compiler Compiler”，它是对 YACC 的继承，它与 YACC 一样，是为加快与解析器逻辑的开发过程而设计的。但是，YACC 所生成的是 C 代码。而 JavaCC 生成的则是 Java 代码。

YACC 与 JavaCC 的出现是因为许多基于 Web 的项目都包含及时查询系统以允许普通用户登陆系统搜索自己所需要的信息。因此，作为普通用户会需要某种语言来表达他们所希望搜索的内容，这种语言被称为是“用户查询语言”。

最早的搜索引擎，对用户查询语言的定义都极其简单。通常就是几个关键字的组合，然而，如果普通用户也希望有一种更健壮更强大的语言来描述他们所要搜索的任务，比如要添加括号或是想执行“与/或”逻辑，甚至是更复杂的语义层的检索，那么他们就发现传统的搜索引擎已经无法提供这种支持。这时，就需要一种方法，用以定义用户将要

使用的语言（用户查询语言），然后对用户语言生成某种解析器，从中得到关键信息，以帮助搜索引擎理解用户的需要。

当然，这种语言定义特性并非只在搜索引擎领域才会使用，一切与用户查询相关的应用或多或少都要接触到这方面的内容。比如最经典的用户查询语言 SQL。

这就是工具 JavaCC 出现的原因。JavaCC 起源于 Sun 公司的“Jack”。“Jack”后来辗转了几家拥有者，比如著名的 Metamata 和 WebGain，最后变成了 JavaCC，然后又回到了 Sun。Sun 公司最后在 BSD 的许可下将它作为开放源代码的项目发布。关于 JavaCC 更详细的内容，大家可以查看 JavaCC 官方网站——http://javac.net.java.net 的内容。

JavaCC 的优势在于它的简单性和可扩展性。要编译由 JavaCC 生成的 Java 代码，无须任何外部 JAR 文件或目录。仅仅用基本的 JDK1.2 或者更高版本的编译器就可以进行编译。而该语言的布局也使得它易于添加产生式规则和行为。JavaCC 的官方 Web 站点上甚至描述了 任何编制异常以便给出用户合适的语法提示。

5.2.2 通过 JavaCC 构建的 Lucene 标准分析器

在 Lucene 中，标准分析器 StandardTokenizer 就是通过 JavaCC 来生成的。当然，除此之外 JavaCC 还生成了相关的一系列文件。打开 Analysis 下的 standard 目录，就可以看到 JavaCC 为 Lucene 生成的标准分析器，以及通过范式定义的各种语言规范。这种规范是被放在一个后缀为.jj 的文件中。

下面以代码 5.7 为例介绍该分析器的范式定义，即 StandardTokenizer.jj 文件。

代码 5.7　StandardTokenizer.jj

```
//一些开关
options {
    STATIC = false;
    UNICODE_INPUT = true;
    USER_CHAR_STREAM = true;
    OPTIMIE_TOKEN_MANAGER = true;
}
    PARSER_BEGIN(StandardTokenizer)
//分词器存放的包
package org.apache.lucene.analysis.standard;
//分类器的主类名
public class StandardTokenizer
    extends org.apache.lucene.analysis.Tokenizer {
  public StandardTokenizer(Reader reader) {
```

```
    this(new FastCharStream(reader));
  }
}
PARSER_END(StandardTokenizer)
    //类似于编译中的正则式定义各种词类型
    TOKEN : {
    < ALPHANUM : ( < LETTER > | < DIGIT > ) + > | < APOSTROPHE :
    < ALPHA > ("'" < ALPHA > ) + > | < ACRONYM :
    < ALPHA > "." ( < ALPHA > ".") + > | < COMPANY :
    < ALPHA > ("&" | "@") < ALPHA >> | < EMAIL :
    < ALPHANUM > ( ("." | "-" | "_") < ALPHANUM > ) * "@" < ALPHANUM >
    ( ("." | "-") < ALPHANUM > ) + >
    | < HOST : < ALPHANUM > ("." < ALPHANUM > ) + >
    | < NUM : ( < ALPHANUM > < P > < HAS_DIGIT >
              | < HAS_DIGIT > < P > < ALPHANUM >
              | < ALPHANUM > ( < P > < HAS_DIGIT > < P > < ALPHANUM > ) +
              | < HAS_DIGIT > ( < P > < ALPHANUM > < P > < HAS_DIGIT > ) +
 | < ALPHANUM > < P > < HAS_DIGIT > ( < P > < ALPHANUM > < P > < HAS_DIGIT > ) +
| < HAS_DIGIT > < P > < ALPHANUM > ( < P > < HAS_DIGIT > < P > < ALPHANUM > ) +
              )
    >
    | < # P : ("_" | "-" | "/" | "." | ",") >
    | < # HAS_DIGIT :
       ( < LETTER > | < DIGIT > ) * < DIGIT > ( < LETTER > | < DIGIT > ) *
      >
    |<#ALPHA:(<LETTER>)+>
    |<#LETTER//字母
       [
        "\u0041"-"\u005a",
        "\u0061"-"\u007a",
        "\u00c0"-"\u00d6",
```

```
        "\u00d8"-"\u00f6",
        "\u00f8"-"\u00ff",
        "\u0100"-"\u1fff"
      ]
    >
  |<CJK://中、日、韩文
    [
     "\u3040"-"\u318f",
     "\u3300"-"\u337f",
     "\u3400"-"\u3d2d",
     "\u4e00"-"\u9fff",
     "\uf900"-"\ufaff"
    ]
  >
|<#DIGIT://数字
  [
   "\u0030"-"\u0039",
   "\u0660"-"\u0669",
   "\u06f0"-"\u06f9",
   "\u0966"-"\u096f",
   "\u09e6"-"\u09ef",
   "\u0a66"-"\u0a6f",
   "\u0ae6"-"\u0aef",
   "\u0b66"-"\u0b6f",
   "\u0be7"-"\u0bef",
   "\u0c66"-"\u0c6f",
   "\u0ce6"-"\u0cef",
   "\u0d66"-"\u0d6f",
   "\u0e50"-"\u0e59",
   "\u0ed0"-"\u0ed9",
   "\u1040"-"\u1049"
```

```
    ]
  >
}
//跳过无法识别的词
SKIP:{
<NOISE:~[]>
}
//重要的 next()方法
  org.apache.lucene.analysis.Token.next() throws IOException:
{
  Token token=null;
}
{
    (
      token=<ALPHANUM>|    token=<APOSTROPHE>|          token=<ACRONYM>|          token=<COMPANY>|
      token=<EMAIL>|    token=<HOST>|     token=<NUM>|     token=<CJK>|     token=<EOF>
    )
    {
       if(token.kind==EOF){
       return null;
       }else{
         return
             new
org.apache.lucene.analysis.Token(token.image,token.beginColumn,token.endColumn,tokenImage[token.kind])
;
       }
    }
}
```

通过文件中加入的中文注释，相信大家对这个文件的含义有了一个大概的了解。通过这样一个输入文件，JavaCC 就为 Lucene 构建了它的分词器。但由于这种分词器的代码完全是程序自动生成的，所以在可读性上比较差，要是有兴趣的话，可以进入文件自动跟踪测试。

以下是分类器的各个类，它们全由 JavaCC 自动生成。

- StandardTokenizer.java
- StandardTokenizerConstants.java
- StandardTokenizerTokenManager.java
- Token.java
- TokenMgrError.java
- CharStream.java
- ParseException.java。

5.3 Lucene 分析器——Analyzer

Lucene 提供了一种不同环境和需求下使用的 Analyzer，最常用的如 StandardAnalyzer，另外还有 SimpleAnalyzer、WhitespaceAnalyzer、GermanAnalyzer 等，这些都被统一称为是 Lucene 内置的 Analyzer。这其中的每一种 Analyzer 都有它自己独特的功能和用途，因此在使用时可以根据不同的需要选择合适的 Analyzer。本节会介绍常用到的几个 Analyzer，关于其他 Analyzer 的用法，可以参考 Lucene 帮助文档中相关的部分。

5.3.1 标准分析器——StandardAnalyzer

StandardAnalyzer 是 Lucene 中最重要的一个分析器，在前面的章节中，已经介绍了它的运行效果，但是一直没有说明它的实现原理。此处就深入它的代码内部，来看一看它到底是如何与分词器结合实现分析功能的。下面以代码 5.8 为例进行详细介绍。

代码 5.8　StandardAnalyzer.java

```
public class StandardAnalyzer  extends Analyzer {
private Set stopset;
  //定义了一个停止词的集合
  public static final String[] STOP_WORDS = StopAnalyzer.ENGLISH_STOP_WORDS;
  //默认构造函数
  public StandardAnalyzer() {
    this(STOP_WORDS);
  }
  //使用停止词集合作为参数的构造函数
```

```
  public StandardAnalyzer(String[] stopWords) {
    stopset = StopFilter.makeStopSet(stopWords);
  }
  //构造词流
  public TokenStream tokenStream(String fieldName, Reader reader) {
    TokenStream result = new StandardTokenizer(reader);
    result = new StandardFilter(result);
    result = new LowerCaseFilter(result);
    result = new StopFilter(result, stopset);
    return result;
  }
}
```

从上述代码中可以看出，StandardAnalyzer 的代码相当简单，在构造函数之后就使用了 tokenStream 方法来创建一个词流，以便可以取出各个词。

在 tokenStream 方法中，当使用一个 reader 对象传入该方法时，首先声明一个 TokenStream 的对象 result，用 StandardTokenizer 为其初始化，并用 reader 作为 StandardTokenizer 的参数。在前面已经说过，StandardTokenizer 是 JavaCC 为 Lucene 构件的分析器，因此，这一做法的目的很明确，就是要先对 reader 所提供的文本流进行分词。然后，在将 result 对象传入一个叫做 StandardFilter 的过滤器，使之“流过”这个过滤器。这里需要注意的是，从完全的代码角度来看，“流”的含义其实就是将前一个构造函数构造的对象放入新的一个构造函数中。Results 将依次被 StandardFilter、LowerCaseFilter、StopFilter 重新定义，然后 return。

值得注意的一点是，虽然 results 对象依次“流过”分词器和各个过滤器，但是一直到 return 语句，reader 对象中的内容并没有发生任何改变。大家可能感到奇怪，为什么在 result 对象流过这么多的过滤器后，reader 中的内容会没有改变呢？

下面以代码 15.9 为例对 LowerCaseFilter 进行介绍。

代码 5.9　LowerCaseFilter.java

```
public final class LowerCaseFilter extends TokenFilter{
  public LowerCaseFilter(TokenStream in){
    super(in);
  }
  public final Token next() throws IOException{
    Token t=input.next();
```

```
    if(t==null)
      return null;
      t.termText=t.termText.toLowerCase();
      return t;
  }
}
```

从上述代码中可以看出，在 LowerCaserFilter 的代码中，只有一个构造函数和一个 next()方法，也就是说，在 result 对象流过这个过滤器时，并没有任何实际的操作执行。那么，实际的操作到底是什么地方执行的呢？

分析通常发生在检索和分析用户输入的关键字时，那么就让我们到检索的代码中找找答案。代码 5.10 是 index 包下的 DocumentWriter.java 中的一段代码，清楚地显示了用户在构建一个分析器后，是如何使用它来进行分词和过滤功能的。

代码 5.10　DocumentWriter.java 代码片断

```
TokenStream stream = analyzer.tokenStream(fieldName, reader);
try {
  for (Token t = stream.next(); t != null; t = stream.next()) {
    position += (t.getPositionIncreament() - 1);
    addPosition(fieldName, t.termText(), position++);
    if (++length > maxFieldLength) {
      break;
    }
  }
}
finally {
  stream.close();
}
```

代码 5.10 的执行时机就是在用户调用了 IndexWriter 的 addDocument 方法以后，并开始对某个文档的域检索。其中的 analyzer 对象就是某个分析器的实例。在这里，可以把它看成是 StandardAnalyzer 的实例。通过调用它的 tokenStream 方法，返回了一个 TokenStream 的对象，并在循环的一开始，就调用了一个 next()方法。

回到 StandardAnalyzer 的 tokenStream 方法中发现，由于 result 最后被赋予的是 StopFilter 类型，所以当循环中调用 next()方法时，首先进入的应该是 StopFilter 内的 next()函数，如代码 5.11 所示。

代码 5.11　StopFilter 中的 next()方法

```
public final Token next() throws IOException {
  for (Token token = input.next(); token != null; token = input.next()) {
    if (!stopWords.contains(token.termText)) {
      return token;
    }
  }
  return null;
}
```

从上述代码可以看出，在 StopFilter 的 next()内首先调用了 input.next()，这个 input 的初始化是在初始化 StopFilter 时进行的,其实 input 的初始化操作也就是把以参数方式传入的 result 赋给 input,该操作是在 StopFilter 的父类 TokenFilter 中完成的。这个 result 是流经上一层过滤后传入的，它所带的类型应该是上一个 Filter 的类型，返回到下面的代码：

```
result = new StandardFilter(result);
result = new LowerCaseFilter(result);
result = new StopFilter(result, stopSet);
```

发现是 LowerCaseFilter 型的，因此，此处的 input.next()就应该调用 LowerCaseFilter 的 next()函数。再到 LowerCaseFilter 的 next()函数中，发现也是如法炮制，调用了 input.next()，我们就再往上一层查看其 next()函数。

依此类推，到达最顶层的 StandardTokenizer 中的 next()函数，前面已经说过，StandardTokenizer 就是 JavaCC 为 Lucene 生成的分词器，它的 next()方法则是从 reader 对象中取出下一个词。

不断循环，就可以取出 reader 中所有的词，这样 StandardAnalyzer 也就完成了分词和过滤的整个过程。虽然这里以 StandardAnalyzer 作为实例讲解了分词过滤的过程，但对于 Lucene 中的所有分析器来说，过程都是一样的，读者可以细心揣摩，对其编程思想和方法进行学习和研究。

代码 5.12 是对 StandardAnalyzer 的一段示例代码，分别演示了对文字数字、首字母缩写、公司名称、E-mail 地址、计算机主机名、数字、带省略号的文本、序列号、IP 地址和中日韩文字的支持。

代码 StandardAnalyzer 中同样包含了 StopAnalyzer 的移除“停止词”的功能，并且其实现机制与 StopAnalyzer 也是相同的（一个默认的词表和一个自定义的 String 类型的数组列表）。

代码 5.12　StandardAnalyzer 的使用

```
package IRLabDemo.analysis;
import java.io.IOException;
import java.io.StringReader;
```

```
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.Token;
import org.apache.lucene.analysis.TokenStream;
public class StandardAnalyzerUsing{
  public StandardAnalyzerUsing(){}
  public static void main(String args[]) throws IOException {
    //构造标准分析器
    Analyzer a=new StandardAnalyzer();
StringReader sr=new StringReader("123,456"+"CNN"+"abc123@cba.com.cn"+
"Microsoft"+"I think ..."+"192.168.0.1"+"中国驰名商标");
    //获取 TokenStream
    TokenStream ts=a.tokenStream(sr);
    int i=0;
    Token t=ts.next();
    while(t!=null){
      i++;
      System.out.println("Line"+i+":"+t.termText());
      t=ts.next();
    }
  }
}
```

上述代码的运行结果如下所示：

```
Line1:123,456cnnabc123
Line2:cba.com.cnmicrosofti
Line3:think
Line4:192.168.0.1
Line5:中
Line6:国
Line7:驰
Line8:名
Line9:商
```

Line10:标

如果依次将代码 5.12 种下面的代码

```
StringReader sr=new StringReader("123,456"+"CNN"+"abc123@cba.com.cn"+
                "Microsoft"+"I think ..."+"192.168.0.1"+"中国驰名商标");
```

一行变换成：

```
StringReader sr=new StringReader("123,456");
StringReader sr=new StringReader("CNN"+":"+"Cable News Network");
StringReader sr=new StringReader("abc123@cba.com.cn");
StringReader sr=new StringReader("I think ...");
StringReader sr=new StringReader("192.168.0.1");
```

(1) 对数字的支持，代码运行结果如下所示：

Line1:123,456

(2) 对首字母缩写的支持，代码运行结果如下所示：

Line1:cnn

Line2:cable

Line3:news

Line4:network

(3) 对电子邮件地址的支持，代码运行结果如下所示：

Line1:abc123@cba.com.cn

(4) 去除空格，代码运行结果如下所示：

Line1:i

Line2:think

(5) 对 IP 地址的支持，代码运行结果如下所示：

Line1:192.168.0.1

5.3.2 “停止词”分析器——StopAnalyzer

StopAnalyzer 中的 tokenStream 方法如下：

```
public TokenStream tokenStream(String fieldName, Reader reader) {
  return new StopFilter(new LowerCaseTokenizer(reader), stopWords);
}
```

由上述代码可以看出，它在实现了基本的分词和小写转换的基础上，还会将一些“停止词”从检索数据中删除。

在 StopAnalyzer 中内嵌了一个通用的英语“停止词”列表，默认情况下都会使用这个列表，并且 Lucene 也允许自定义这个词表。这为 Lucene 的使用者提供了定制的可能。代码 5.12 是在 Lucene 开发包中已经定义的“停止词”表。

代码 5.13　StopAnalyzer 的使用

```
package org.apache.lucene.
    public static final String[] ENGLISH_STOP_WORDS = {
    "a", "an", "and", "are", "as", "at", "be", "but", "by", "for", "if", "in",
    "into", "is", "it", "no", "not", "of", "on", "or", "s", "such", "t", "that",
    "this", "the", "they", "their", "then", "there", "these", "to", "was",
    "will", "with"
}
```

StopAnalyzer 一共有两个构造函数。其中一个构造函数的方法如下：

```
public StopAnalyzer() {  }
```

这个构造函数只允许用户使用默认的停止词表，在 StopAnalyzer 还有另外一个构造函数，在这个构造函数中 Lucene 允许使用者传递一个 String 类型的数组列表来代替 Lucene 自带的“停止词”列表。这个函数的方法如下：

```
public StopAnalyzer(String[] stopWords)  {  }
```

下面通过代码 5.14 说明这两种不同构造函数的使用方法。

代码 5.14　不同构造函数的使用方法

```
package IRLabDemo.analysis;
import java.io.IOException;
import java.io.StringReader;
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.SimpleAnalyzer;
import org.apache.lucene.analysis.Token;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.StopAnalyzer;
public class StopAnalyzerTest {
  public StopAnalyzerTest(String name) {
  }
  public static void main(String args[]) throws IOException {
    //构建一个 StopAnalyzer 的分析器
    Analyzer a = new StopAnalyzer();
```

```
        StringReader sr = new StringReader(
            "People are always talking about 'the problem of the youth'."); ;
        //获取 TokenStream
        TokenStream ts = a.tokenStream(sr);
        int i = 0;
        Token t = ts.next();
        //循环从 TokenStream 中依次取出每一个词
        while (t != null) {
          i++;
          System.out.println("Line" + i + ":" + t.termText());
          t = ts.next();
        }
      }
    }
```

在上述代码中使用了 StopAnalyzer 中默认的“停止词”表，即在处理时 StopAnalyzer 会忽略如“a”、“an”、“are”

能够移除“停止词”是一项很有益的功能，不过在移除“停止词”的同时又带来了另外一个很有趣的问题：移除了这些所谓的“停止词”之后，所留下的空间当该如何处理呢？下面就用例子来解释一下为什么发生这种状况。

假设现在对下面的句子作索引“one is not enough”，根据 Lucene 自带的“停止词”列表，这个句子中的“is”和“not”会被删除掉，那么就只剩下“one”和“enough”。但是，此时 StopAnalyzer 自身并不知道有哪些词已经被移除了。所以，索引的结果同对“one enough”建立索引是一样的。

下面再分析建立索引时的状况。如果在使用 StopAnalyzer 的时候同时用到了 QueryParser，那么检索结果可能就会有些出乎意料了。文档匹配时会将“one enough”、“ one is enough”、“one but not enough”等短语作为可以匹配的短语来对待，但是原始的短语却是“one is not enough”，这样一来，所搜索到的结果与原意已经发生了很大的变化。

所有的移除“停止词”的情况都会带来这样一个语义上的问题。那么这样会不会导致失去一些语义信息呢？这还是取决于使用 Lucene 的方式和移除掉的这些词在应用程序中是否有意义，并不能一概而论。此处不再详细说明，有兴趣的话可以自行研究。

5.3.2 其他分析器

WhitespaceAnalyzer 是用于对空格进行切分的一个分析器。它的 tokenStream 方法如下：

```
    public TokenStream tokenStream(String fieldName, Reader reader) {
      return new WhitespaceTokenizer(reader);
  }
```

代码 5.16 是一段，说明 WhitespaceAnalyzer 使用方法的示例代码。

代码 5.16　WhitespaceAnalyzer 的使用方法

```
package ch12;
import java.io.IOException;
import java.io.StringReader;
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.SimpleAnalyzer;
import org.apache.lucene.analysis.StopAnalyzer;
import org.apache.lucene.analysis.Token;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.WhitespaceAnalyzer;
import org.apache.lucene.analysis.Standard.StandardAnalyzer;
public class WhitespaceAnalyzerTest() {
  public static void main(String args[]) {
    //构建一个 WhitespaceAnalyzer
    Analyzer a = new WhitespaceAnalyzer();
    //输入字符串
    StringReader sr = new StringReader("In the early days of the settlement of Australia,enterprising
settlters unwisely introduced the European rabbit.");
    //获取 TokenStream
    TokenStream ts=a.tokenStream(sr);
    try{
      int i=0;
      Token t=ts.next();
      //循环从 TokenStream 中依次取出每一个词
      while(t!=null){
        i++;
        System.out.print(t.termText()+":");
```

```
            t=ts.next();
          }
        }catch(IOException e){
          e.printStackTrace();
        }
      }
    }
```

当测试数据为“In the early days of the settlement of Australia,enterprising settlters unwisely introduced the European rabbit.”时，得到的运行结果如下所示。

In:the:early:days:of:the:settlement:of:Australia,enterprising:settlters:unwisely:introduced:the:European:rabbit.

从上述结果可以看出，测试数据被按照空格进行了“切割”，并且最后的标点符号“.”仍然在输出结果中保存了下来。

5.4 总结

本章详细介绍了 Lucene 的分析器，以及其中的分词和过滤结构。通过学习这些基本知识，已经能够很好了解了 Lucene 是如何对文本信息进行分词以及其他相应处理。有兴趣的话，可以自行尝试编写 Lucene 的分析器，以完成更复杂的功能。

第六章 Web 搜索引擎中信息的查询服务

经过前几章的学习，现在我们进入搜索引擎的最后一个环节——查询服务。查询服务包括：接受用户输入的查询短语、检索索引库、获得相应的检索结果并显示给用户。此时，我们通过前几章的学习已经拥有了由网页集生成的索引文件，需要做的就是通过查询代理实现索引数据与用户查询的沟通。

6.1 查询服务的系统结构及工作原理

6.1.1 查询服务的系统结构

经过 Web 信息的预处理，传递到服务阶段的数据结构包括网页索引库和倒排文件，倒排文件中包括倒排表和索引词表。查询服务接受用户输入的查询短语，切词后，从索引词表和倒排文件中检索获得包含查询短语的文档并返回给用户。

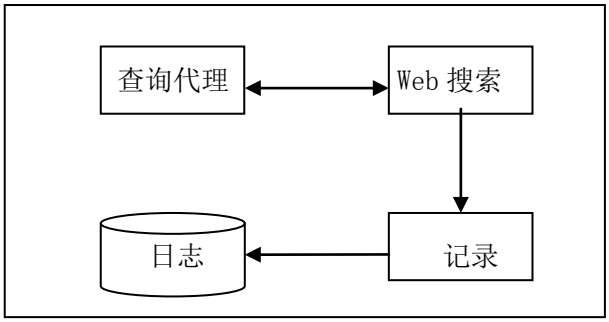


图 6.1 信息查询的系统结构

6.1.2 查询服务系统的工作原理

从一个原始网页集合 S 开始，预处理过程得到的是对 S 的一个子集的元素的某种内部表示，这种表示构成了查询服务的直接基础。对每个元素来说，这种表示至少包含如下几个方面：

- ✧ 原始网页
- ✧ URL 和标题
- ✧ 编号
- ✧ 所含的重要关键词的集合（以及它们在文档中的位置信息）
- ✧ 其他的一些指标（如重要程度，分类代码等）

而系统关键词的总体的集合和文档的编号一起构成了一个倒排文件的结构，使得一旦得到一个关键词输入，系统能够迅速给出相关文档编号的集合输出。

然而，用户通过搜索引擎看到的不是第一个“集合”，而是一个“列表”如何从集合生成一个列表，是服务子系统的主要工作。从搜索引擎系统的功能划分的角度，有时候将倒排文件的生成也作为服务子系统的一部分功能，但我们这里将它划分到预处理阶段中觉得更方便一些。换句话说，服务子系统是在服务进行的过程中涉及的相关软件程序，而为这些软件程序事先准备的数据的程序都算在预处理子系统中。下面来看对服务子系统的要求和其工作原理，主要有三个方面：

- ✧ **查询方式和匹配**

查询方式指的是系统允许用户提交的查询的形式。考虑到各种用户的不同背景和不同的信息需求，不可能有一种普遍适用的方式，一般认为，对于普通的网络用户来说，最自然的方式就是“要什么就输入什么”。但这是一种相当模糊的说法。例如，用户输入”北京大学“，可能是他想了解北京大学目前有什么信息向外发布，如想看看今年的招生政策（于是希望看到的是北大网站上的内容）；也可能上他想了解外界对北京大学有些什么评价（于是希望看到的是其他权威网站上关于北大的消息）。这是两种截然不同的需求，在其他的一些情况下，用户可能关心的是间接信息，例如，“喜马拉雅山的高度”，8848 应该是他的需要，但不可能包含在这短语中，而用户输入“海内存知己”则很可能是想知道该词的作者是谁或者希望给个提醒前面几句是什么。尽管如此，用一个词或者短语来直接表达信息需求，希望网页

中含有该词语或者该短语中的词，依然是主流的搜索引擎查询模式。这不仅是因为它的确代表了大多数的情况，还因为它比较容易实现。这样，一般来讲，系统面对的是查询短语。就英文来说，它是一个词的序列；就中文来说，它是包含若干个词的一段文字。

一般地，我们用q_0表示用户提交的原始查询，例如，q_0=“网络与分布式系统实验室”。它首先需要被“切词”或者说是“分词”，即把它分为一个词的序列。如上例，则为“网络 与 分布式 系统 实验室”。然而需要删除那些没有查询意义或者几乎在每篇文档中的都会出现的（如“的”）在本例中即为“与”。最后形成一个用户参与匹配的查询词语表$q=\{t_1,t_2,...,t_m\}$，在本例中就是q={网络 ，分布式，系统 ，实验室}。

前面讲过，倒排文件就是用词来作为索引的一个数据结构，显然，q中的词必须是包含在倒排文件的词表中才有意义。有了这样的q，它的每个元素都对应倒排文件中的一个倒排表（文档编号集合），记作 L（t_i），它们的交集即为对应的查询的结果的文档集合，从而实现了查询和文档的匹配。上述过程的基本假设是：用户希望网页包含所输入的查询文字。

✧ **结果排序**

上面，我们了解了如何得到和用户查询相关的文档集合的过程，这个集合的元素需要以一定的形式通过计算机上显示屏呈现给用户。就目前的技术情况看，列表是最常见的形式（但人们也在探求新的形式，如 Vivisimo 引擎就将结果页面以类别的形式呈现）。给定一个查询结果集合，$R=\{r_1,r_2,...,r_m\}$，所谓列表就是按照某种评价方式确定出 R 中的元素的一个顺序，让这些元素以这种顺序呈现出来。笼统的讲，r_i和q 的相关性是形成这种顺序的基本因素。但是，有效地定义相关性本身是很困难的 ，从原理上讲它不仅和查询词有关，而且还和用户的背景，以及用户的查询历史有关。不同需求的用户可能输入同一个查询，同一个用户在不同的时间输入的相同查询可能是针对不同的信息需求。

为了形成一个合适的结果输出顺序，在搜索引擎出现的早期人们采用了传统的信息检索领域很成熟的基于词汇出现频度的方法。大致上讲就是一篇文档中包含的查询（q）中的那些词越多，则该文档就应该排在越前面；再精细一些的考虑则是若一个词在越多的文档中出现，则该词用于区分文档相关性的作用就越小。这样一种思路不仅有一定直觉上的道理，而且在倒排文件数据结构上很容易实现。因为，当我们通过前述关键词的提取过程，形成一篇文档的关键词集合，$q=\{t_1,t_2,...,t_m\}$的时候，很容易同时得到每一个t_i在该文档中的出现次数，即词频，而倒排序文件中每个倒排表的长度则对应着一个词所涉及的文档的篇数，即文档频率。然而，由于网页编写的自发性、随意性较强，仅仅针对词的出现来决文档的顺序，在 Web 上做信息检索表现出明显的缺点，需要其他技术的补充。这方面最重要的成果就是前面提到的 PageRank 算法。通过在预处理阶段为每篇网页形成一个独立于查询词（也就和网页内容无关）的重要性指标，将它和查询过程中形成的相关性指标结合形成一个最终的排序，是目前搜索引擎给出查询结果排序的主要方法。

✧ **文档摘要**

搜索引擎给出的结果是一个有序的条目列表，每一个条目有三个基本的元素：标题，网址和摘要。其中的摘要需要从网页正文中生成。一般来讲，从一篇文字中生成一个恰当的摘要是自然语言理解领域的一个重要课题，人们已经做了多年的工作并取得了一些成果。但相关的技术用到网络搜索引擎来有两个基本困难，一是网页的写作通常不规范，

文字比较随意，因此从语言理解的角度难以做好；二是复杂的语言理解算法耗时太多，不适应搜索引擎要高效处理海量网页信息的需求。

有关人士做过统计，即使是分词这一项工作，在高档微机上每秒钟也只能完成 10 篇左右的网页处理。因此搜索引擎在生成摘要时要简便许多，基本上可以归纳为两种方式：一是静态方式，即独立于查询，按照某种规则，事先在预处理阶段从网页内容提取出一些文字，如截取网页正文的开头的 512 字节（对应 256 个汉字）或者将每一个段落的第一个句子拼起来，等等。这样形成的摘要存放在查询子系统中，一旦相关文档被选中能够与查询项匹配，就读出返回给用户。显然，这种方式的一个最大的缺点是摘要和查询无关，一篇网页有可能是多个不同查询的结果。例如，当用户分别查询“北大计算机网络”和“北大分布式系统”时，主页 http//net.pku.edu.cn 在两种情况下应该都作为结果返回。当用户输入某个查询，他一般是希望摘要中能够突出显示和查询直接对应的文字，希望摘要中出现和他关心的文字相关的句子。因此，我们有了“动态摘要”方式，即在响应查询的时候，根据查询词在文档中的位置，提取出周围的文字来，在显示时将查询词标亮。这是目前大多数搜索引擎采用的方式。为了保证查询的效率，需要在预处理阶段分词的时候记录每个关键词在文档中到出现的位置。

除上述外，查询服务返回的内容还有一些细节的支持。例如，对应一个查询往往会有成千上万的结果，返回个用户的内容通常都是按页组织的，一般每页显示 10 个结果。统计表明，网络用户一般没有耐心一页页看下去，平均翻页数小于 2。这告诉我们将第一页的内容组织好非常重要，如果希望用户多用搜索引擎，就要让第一页的内容尽量有吸引力。

6.1.3 查询服务系统的具体实现

本节中将详细介绍一下查询服务实现的具体过程，主要分两个方面进行讲解：结果集合的形成和显示。

6.1.3.1 结果集合的生成

根据用户输入的查询短语，产生结果集合，是检索倒排索引的过程。下面我们就“检索”的定义作一定的介绍：

首先，定义系统的元检索——单个词汇的检索方式。

设 $D=\{t_1,t_2,...,t_D\}$ 为系统的特征项词典；集合 $P=\{p_1,p_2,...,p_p\}$ 为当前保存的网页集合；系统的索引可表示为集合 $R=\{<t,p>|r(t,p)>0,(t,p)\in D\times P\}$，其中，r(t,p)是相关度函数，表示词汇 t 和网页 p 相关度，如果 t 是网页 p 的一个特征项，那么它就使用相关度算法给出相应的正值，如果不是，相关度为 0；搜索引擎系统 S 为一个三元组 $S=\{<t,p,r>|(t,p,r)\in D\times P\times R\}$。则有：

$$WP(t)=\phi(t,S)=\{p|(t,p)\in R,p\in P\} \quad (6.1)$$

其中，WP 代表检索词汇 t 的相关网页集合，ϕ 函数是系统 S 的元检索函数。

显然，用户不可能总是进行词汇级的检索。大部分的用户输入的检索是词组或自然语句。如何从检索中提取出关

键词，在各种信息检索系统中实现是不同的，有些系统直接从检索输入中提取关键词；有些系统提取关键词后，再根据一些规则对关键词进行扩充。在本模型，统一的将它们定义为关键词提取函数：$g(q,D)=\{t_1,t_2,...,t_m\}$，即 g 获得检索输入 q 的相关关键词集合。由公式（6.1），关键词t_i的相关网页集合为$WP(t_i)$，则最终检索输入 q 的对应结果为

$$WP = f(WP(t_1),WP(t_2),...,WP(t_m) = f(q,D,P,R) = f(q,S) \quad (6.2)$$

其中，函数 f 表示从 q 中提取的关键词的逻辑关系运算式，f（q，S）是系统检索的抽象表达式。由公式（6.1）和（6.2）可以得到，作为文档特征项的关键词在检索过程中起到了桥梁作用，关键词选择的好坏直接影响到检索结果的质量。

如公式（6.2）所示，f（q，S）是系统检索的抽象表达式，它代表系统选定的检索算法。目前主要的检索算法有：基于内容检索方法、基于词条位置信息检索方法、基于超链接分析的检索方法以及融合的检索方法，我们采用的检索算法是利用内容和超链分析的融合，处理步骤如下：

- ✧ 分析查询请求，将查询语句进行分词划分成多个查询关键词。
- ✧ 将所有查询关键词向量空间的形式。
- ✧ 查找每个词在索引数据库中的位置和频率信息。
- ✧ 查找索引库中的网页列表，计算查询关键词和每个网页之间的相似度值。
- ✧ 根据内容相似度和 PageRank 链接分析分数的综合权值把匹配的网页排序，组织并输出排名最前面的网页(分数值比较大的)。

算法描述如图 6.2：

```
1 初始化，结果集合R＝Φ，权值累加器A＝0.
2 for each ti in q do
    begin
      2.1.读取ti的倒排项数据reader(ti).
      2.2.逐项处理倒排项数据merge(R, ti).
          2.2.1得到ti的文档集{dti}，权值wti.
          2.2.2执行布尔查询R＝Boolean(R,{ dti}),一般布尔运算为AND
          2.2.3使用wti更新结果集合R中语素对应的权值A.
    end
3 根据累加器A和PageRank值计算所得的结果集合R中选取最大权值的K个结果
select(R,A,K)
```

图6.2 基本的检索算法

Fig. 6.1 Algorithm of the Basic Retrieval

这个算法是实际搜索引擎算法的简化。实际搜索引擎的倒排索引中记录了索引词的权重和位置信息，检索阶段应

该一起读出，并加以综合考虑；并且为了在获得结果前读取尽量少的数据，查询 q 中的 t_i 按文档频率的倒数降序排列。

6.1.3.2 查询结果的显示

用户界面主要用于和用户交互，包括响应用户的查询检索和记录用户的行为用户界面主要负责和用户直接接触的事件，它包括：

（1）从用户获得用户的查询请求，提交给查询代理。

（2）查询代理检索索引词表和倒排表，产生结果按照一定的输出格式显示给用户；

（3）记录日志，包括用户查询短语和查询时间等信息。

对于功能（1）通过 HTML 语言的<FORM>表单来实现。用户在相应的检索表格中输入需要查询的短语，然后提交即可。对于一个已经提交的检索，服务器方启动一个 CGI 程序进行响应，该程序对用户提交的各个表项进行合法性检查，通过后形成一个机构数据传给查询代理。

对于功能（2），主要用到动态网页生成技术和动态摘要算法。按照查询代理检索回来的数据，CGI 程序根据决定输出风格的模板，将结果数据整合到模板中，形成结果页面，输出给用户。用户查询 q 的向量表示 $q=\{t_1,t_2,...,t_m\}$。包括查询 q 的网页正文 Cnt，并且 Cnt 已经过滤之掉网页标签等格式信息。为了保证用户所有的 t_i 都被加亮显示，摘要算法是在一个循环中实现的。其中“定位完整汉字的起始位置 ”是很必要的一步。以中文编码的网页为例，每个汉字占用 2 字节空间，如果摘要算法不能够正确的判断一个汉字的起始位置，而是从汉字的一半位置开始，就会形成摘要信息乱码。

对于功能（3），查询代理接受用户输入的信息，记录到日志文件。一条用户查询日志记录应该包括：查询时间和查询短语。

另外，搜索引擎索引的网页不是当前互联网上最新的网，因此存在已经消失的可能性。为保证用户能够继续访问相应的信息，因此大型完备的搜索引擎需要提供网页快照功能。搜索引擎需要把网页下载到本地，并建立网页索引库以便可以迅速提供网页快照。网页索引库的建立是在信息的预处理阶段完成的。

6.2 Luceue 中的检索

只有能够检索到用户需要的内容，才能算是一个好的搜索引擎。那么 Luceue 作为一个开源的搜索引擎开发包，它的检索过程又是怎样实现的呢？在本章中将介绍 Luceue 这方面的知识。本章的内容主要有

- Luceue 的搜索流程
- Luceue 搜索与结果的表示
- Luceue 的评分机制
- 构建 Luceue 的各种 Query

- Luceue 的 QueryParser 类

6.2.1 Luceue 的检索流程

6.2.1.1 初始化 Luceue 检索工具 IndexSearcher

IndexSearcher 是 Luceue 中最基本的检索工具，所有的饿检索都会用到 IndexSearcher 检索工具，但是在使用 IndexSearcher 之前，还要做一些准备工作，即对检索工具 IndexSearcher 进行初始化。

初始化 IndexSearcher，需要设置索引存放的路径，这样才能让查询器定位索引，用于后面进行搜索。如以下为一个实例化 IndexSearcher 的过程。

Searcher searcher=new IndexSearcher（indexDir）;

返回的结果是 IndexSearcher 类的一个实例，indexDir 表示索引文件的存放路径。当然，初始化 IndexSearcher 不只这一种方式，详细内容将在后面的小节中说明。

6.2.1.2 构建 Query

Query 的中文意思就是“查询”。在 Luceue 中，它是一个很重要的概念，就是指对于需要查询的字段采用什么样的方式进行查询，如模糊查询，语义查，短语查，范围查询，组合查询等，正是因为 Query 的存在，Luceue 才有了非常丰富的查语言。

杂使用 Query 之前需要首先生成一个 Query 对象，Luceue 即允许直接生成一个 Query 类型的对象，也允许使用 QueryParser 类的 parse () 方法来返回一个 Query 类型的对象。这两种方法在功能上是完全一样的，只是后者在使用时更法官那边一些，而前者则更为灵活。

6.2.1.3 检索并处理返回结果

在构建完 Query 对象后，就可以使用前面已经初始化好的 IndexSearcher 工具来进行检索了，IndexSearcher 提供了良好的检索接口，用户只需要简单地将 Query 对象传入，就可以得到一个返回结果。当然，这个过程看似简单，其中也有许多值得思考的问题，如检索结果的排序，过滤等。

6.2.2 检索与结果

6.2.2.1 检索工具——IndexSearcher 类

前面已经介绍，Luceue 中使用 IndexSearcher 类来对索引进行检索。IndexSearcher 类继承自 Searcher 基类，是 Luceue 中最重要的一个检索用类。上一小节说过，在初始化一个 IndexSearcher 类时最重要的就是告诉它索引存放的路径，只有这样，检索工具才可以定位索引，从而完成查找的任务。IndexSearcher 有四个构造函数

Public IndexSearcher(String path) throws IOExcetion{

```
        This.( IndexReader.open(path),true);
        }
    Public IndexSearcher(Directory directory) throws IOExcetion{
        This.( IndexReader.open(directory),true);
        }
    Public IndexSearcher(IndexReader r){
        This(r,false);
        }
    Private IndexSearcher(IndexReader r，Boolean closeReader)
        {reader=r;
          This.closeReader=closeReader;
        }
```

可以看到，IndexSearcher 的四个构造函数，第一个方法，直接使用了索引存放的路径作为参数来构造对象。第二种方式则是使用 Directory 类型的饿对象构建 IndexSearcher 第三种是直接使用 IndexReader 来初始化一个 IndexSearcher 对象。第四种则是在第三种的基础上加了一个布尔型的开关，用于判断在关闭 IndexSearcher 所带的 IndexReader 对象。读者可以看出，实际上，无论传入的参数类型是什么，IndexSearcher 最终都还是使用 IndexReader 作为实际的索引目录读取器。前三种构造函数均首先根据传入的参数生成一个 IndexReader 对象，然后调用了第四种构造方法来完成 IndexSearcher 的初始化工作。

在初始化以后，在进行搜索前还需要构建一个 Query 对象。关于 Query 对象的构建将在后面的章节进行详细叙述。使用 IndexSearcher 来进行检索的大致过程如代码 6.1 所示：

代码 6.1 IndexSearcher 类的使用

```
package IRLabDemo.Search;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.index.IndexReader;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.Term;
import org.apache.lucene.queryParser.QueryParser;
import org.apache.lucene.search.Hits;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;

public class IndexSearcherTest1 {

        public static void main(String[] args) throws Exception {

                Document doc1 = new Document();
                doc1.add(Field.Text("name", "word1 word2 word3"));
                doc1.add(Field.Keyword("title", "doc1"));
```

```
Document doc2 = new Document();
doc2.add(Field.Text("name", "word4 word5 word6"));
doc2.add(Field.Keyword("title", "doc2"));

Document doc3 = new Document();
doc3.add(Field.Text("name", "word1 word3 word5"));
doc3.add(Field.Keyword("title", "doc3"));

Document doc4 = new Document();
doc4.add(Field.Text("name", "word2 word4 word6"));
doc4.add(Field.Keyword("title", "doc4"));

IndexWriter writer = new IndexWriter("c:\\index",
                        new StandardAnalyzer(), true);
writer.setUseCompoundFile(true);
writer.addDocument(doc1);
writer.addDocument(doc2);
writer.addDocument(doc3);
writer.addDocument(doc4);
writer.close();

Query query = null;
Hits hits = null;

String key1 = "word1";
String key2 = "word2";
String key3 = "word3";
String key4 = "word4";
String key5 = "word5";
String key6 = "word6";

IndexSearcher searcher = new IndexSearcher("c:\\index");

query = QueryParser.parse(key1, "name", new StandardAnalyzer());
hits = searcher.search(query);
printResult(hits, key1);

query = QueryParser.parse(key2, "name", new StandardAnalyzer());
hits = searcher.search(query);
printResult(hits, key2);

query = QueryParser.parse(key3, "name", new StandardAnalyzer());
hits = searcher.search(query);
printResult(hits, key3);
```

```
            query = QueryParser.parse(key4, "name", new StandardAnalyzer());
            hits = searcher.search(query);
            printResult(hits, key4);

            query = QueryParser.parse(key5, "name", new StandardAnalyzer());
            hits = searcher.search(query);
            printResult(hits, key5);

            query = QueryParser.parse(key6, "name", new StandardAnalyzer());
            hits = searcher.search(query);
            printResult(hits, key6);

        }

        public static void printResult(Hits hits, String key) throws Exception {
            System.out.println("查找 \"" + key + "\" :");
            if (hits != null) {
                if (hits.length() == 0) {
                    System.out.println("没有找到任何结果");
                } else {
                    System.out.print("找到");
                    for (int i = 0; i < hits.length(); i++) {
                        Document d = hits.doc(i);
                        String dname = d.get("title");
                        System.out.print(dname + "      " );
                    }
                    System.out.println();
                    System.out.println();
                }
            }
        }
    }
```

上述代码中初始化了一个 IndexSearcher 对象后，按不同的关键字创建了 4 个不同的 Document 对象，每个对象中包含两个字段：name 和 title（即 Document 的名称）。然后调用 IndexSearcher 的 search（Query）方法进行查找，这样就实现了最简单的检索功能。代码的输出结果如图 6.3 所示：

```
找到 doc1      doc3
查找  "word2" :
找到 doc1      doc4
查找  "word3" :
找到 doc1      doc3
查找  "word4" :
找到 doc2      doc4
查找  "word5" :
```

找到 doc2　　doc3
查找 "word6" :
找到 doc2　　doc4

图 6.3 代码 6.1 运行结果

6.2.2.2 检索结果——Hits

在检索完成之后，就需要把检索结果返回并显示给用户，只有这样才算是完成了检索的任务，在 Luceue 中检索结果的集合是用 Hits 类的实例来进行表示的。所有的 search 方法都返回一个类型为 Hits 的对象。主要有以下几个常用的方法：

- length() 返回检索到结果的总数量
- doc(int n) 返回第 n 个文档
- id(int n) 返回第 n 个文档的内部 ID 号
- score(n) 返回第 n 个文档的得分

其中，length() 和 doc(int n) 方法共同使用，就可以遍历结果集合中的所有文档记录。不过有一点值得注意，如果一个结果集含有 10000 条记录，而 Hits 对象一次性就把检索结果全部返回，那么这个 Hits 对象的结果就会大不一样。

但是，它并不是一次将所有的结果返回，而是采取一种懒惰的方式来加载返回结果，即当用户将要访问的某个文档的时候，Hits 对象在内部对 Luceue 的索引又进行了一次检索，将这个最新的结果返回给用户。

代码 6.2 是一个较高级的 Hits 的使用示例：

代码 6.2 Hits 对象的使用

```
package IRLabDemo.Search;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.*;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.queryParser.QueryParser;
import org.apache.lucene.search.Hits;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.Searcher;

public class HitsTest
{
   public static void main(String[] args) throws Exception
   {
        // 构建索引
```

```
buildIndex();
// 使用已经存在索引目录
Searcher searcher = new IndexSearcher("c:\\index");
// 使用标准分析器
Analyzer aStandardAnalyzer = new StandardAnalyzer();
// 从标准输入读取查询的字符串
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
while (true)
{
   System.out.println("-------------------------------------------------");
   System.out.print("Query: ");
   String line = in.readLine();
   // 判断是否直接输入的回车
   if (line.length() == 0)
      break;
   // 构造 Query 对象
   Query query = QueryParser.parse(line, "contents", aStandardAnalyzer);
   // 输出要搜索的内容
   System.out.println("查找 :       " + query.toString("contents"));
   // 使用 searcher 对象的 search 方法进行搜索，返回的是一个 Hits 类型的对象
   Hits hits = searcher.search(query);
   // 使用 Hits 对象的 length（）方法，输出搜索到的文档的数量
   System.out.println("总共找到 " + hits.length() + " 个文档");
   // 定义每次显示的搜索结果数目
   final int HITS_PER_PAGE = 10;
   // 循环输出
   for (int start = 0; start < hits.length(); start += HITS_PER_PAGE)
   {
      //
      int end = Math.min(hits.length(), start + HITS_PER_PAGE);
      for (int i = start; i < end; i++)
      {
         // 取得搜索结果中的一个文档对象
         Document doc = hits.doc(i);
         // 输出文档的 ID 编号
         System.out.println("文档的内部 ID 号:" + hits.id(i));
         // 输出文档的评分
         System.out.println("文档的分值:" + hits.score(i));
         // 输出文档的存放路径
         String path = doc.get("path");
         if (path != null)
         {
            System.out.println("路径为："+path);
         }
```

```
            }
            // 判断是否还有结果未输出
            if (hits.length() > end)
            {
               System.out.print("more (y/n) ? ");
               line = in.readLine();
               if (line.length() == 0 || line.charAt(0) == 'n')
                  break;
            }
         }
      }
      //
      searcher.close();
   }
   public static void buildIndex() throws Exception {
      Document doc1 = new Document();
      doc1.add(Field.Text("contents", "word1 word"));
      doc1.add(Field.Keyword("path", "path\\document1.txt"));

      Document doc2 = new Document();
      doc2.add(Field.Text("contents", "word2 word"));
      doc2.add(Field.Keyword("path", "path\\document2.txt"));

      Document doc3 = new Document();
      doc3.add(Field.Text("contents", "word3 word"));
      doc3.add(Field.Keyword("path", "path\\document3.txt"));

      Document doc4 = new Document();
      doc4.add(Field.Text("contents", "word4 word"));
      doc4.add(Field.Keyword("path", "path\\document4.txt"));

      IndexWriter writer = new IndexWriter("c:\\index", new StandardAnalyzer(), true);

      writer.addDocument(doc1);
      writer.addDocument(doc2);
      writer.addDocument(doc3);
      writer.addDocument(doc4);
      writer.close();
   }
}
```

建立完索引后，初始化一个 IndexSearche 对象来进行检索。对于检索结果，在代码中使用了 Hits 对象所提供的大多数方法，比如获取文档、获取文档 ID 和获取文档评分等。运行是，首先要输入要查询的字符串，然后根据输入的查询字符串进行检索，运行结果如图 6.4 所示：

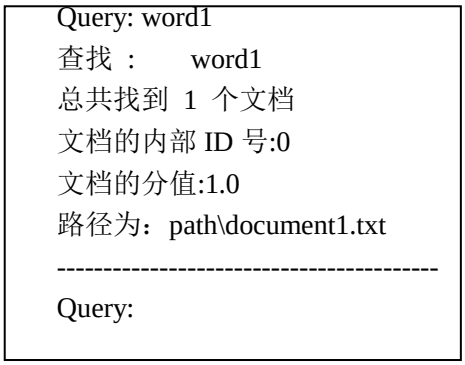


图 6.4 代码 6.2 运行结果

在代码 11.2 运行过程中，将要求用户输入 Query:（要查询的关键字），完成一次查询后将提示用户输入下一次查询。查询结果一次显示 10 条，如果查询结果大于 10，将提示用户是否浏览下 10 条记录。

6.2.3 构建各种 Query

检索流程中的第二步就是构建一个 Query。当用户输入一个关键字，查询服务接收到后，并不是立刻就将它放入后台开始进行关键字的检索，而应该首先对这个关键字进行一定的分析和处理，使之成为一种后台可以理解的形式，只有这样，才能提高检索的效率，同时检索出更加有效的结果。那么，在 Luceue 中，这种处理，其实就是构建 Query 对象。

就 Query 对象本身而言，它只是 Luceue 的 search 包中的一个抽象类，这个抽象类有许多子类，代表不同类型的检索。如常见的 TermQuery 就是将一个简单的关键字进行封装后的对象，类似的还有 BooleanQuery，即布尔型的查找。

6.2.3.1 按词条检索——TermQuery

TermQuery 是最简单，也是最常用的 Query。TermQuery 可以理解为“词条检索”，在搜索引擎中最基本的检索就是在索引中检索某一词条，而 TermQuery 就是用来完成这项工作的。

要使用 TermQuery 进行检索首先需要构建一个 Term 对象：

```
Term term=new Term("name","word1");
```

然后使用 term 对象作为参数来构造一个 TermQuery 对象：

```
Query query = new TermQuery(term);
```

代码 6.3 为 TermQuery 的具体使用过程：

代码 6. 3 TermQuery 的使用

```
package IRLabDemo.Search;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.Term;
```

```
import org.apache.lucene.search.Hits;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.TermQuery;

public class TermQueryTest
{
   public static void main(String[] args) throws Exception
   {
      Document doc1 = new Document();
      doc1.add(Field.Text("name", "word1 word2 word3"));
      doc1.add(Field.Keyword("title", "doc1"));

      IndexWriter writer = new IndexWriter("c:\\index", new StandardAnalyzer(), true);
      writer.addDocument(doc1);
      writer.close();

      Query query = null;
      Hits hits = null;
      IndexSearcher searcher = new IndexSearcher("c:\\index");
      query = new TermQuery(new Term("name","word1"));
      hits = searcher.search(query);
      printResult(hits, "word1");
      query = new TermQuery(new Term("title","doc1"));
      hits = searcher.search(query);
      printResult(hits, "doc1");
   }
   public static void printResult(Hits hits, String key) throws Exception
   {
      System.out.println("查找 \"" + key + "\" :");
      if (hits != null)
      {
         if (hits.length() == 0)
         {
            System.out.println("没有找到任何结果");
         }
         else
         {
            System.out.println("找到" + hits.length() + "个结果");
            for (int i = 0; i < hits.length(); i++)
            {
               Document d = hits.doc(i);
               String dname = d.get("title");
               System.out.print(dname + "      ");
```

```
            }
            System.out.println();
            System.out.println();
          }
        }
      }
    }
```

在代码中，TermQuery 完成了量词关键字的检索。不同的是第一次查找 name 第二次查找 title。运行结果如图所示：

```
查找 "word1" :
找到 1 个结果
doc1

查找 "doc1" :
找到 1 个结果
doc1
```

图 6.5 代码 6.3 运行结果

6.2.3.2 “与或”检索——BooleanQuery

BooleanQuery 也是实际开发过程中经常使用的一种 Query。它其实是一个组合的 Query，在使用时可以把各种 Query 对象添加进去并标明它们之间的逻辑关系。

public void add(Query query ,boolean required,boolean prohibited)；

另外值得注意的是：BooleanQuery 是可以嵌套的，一个 BooleanQuery 可以成为另一个 BooleanQuery 的条件子句。代码 6.4 为 BooleanQuery 的具体使用过程：

代码 6.4 BooleanQuery 的使用

```
package IRLabDemo.Search;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.Term;
import org.apache.lucene.search.BooleanQuery;
import org.apache.lucene.search.Hits;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.TermQuery;

public class BooleanQueryTest
{
   public static void main (String [] args) throws Exception {
```

```
    Document doc1 = new Document();
    doc1.add(Field.Text("name", "word1 word2 word3"));
    doc1.add(Field.Keyword("title", "doc1"));

    Document doc2 = new Document();
    doc2.add(Field.Text("name", "word1 word4 word5"));
    doc2.add(Field.Keyword("title", "doc2"));

    Document doc3 = new Document();
    doc3.add(Field.Text("name", "word1 word2 word6"));
    doc3.add(Field.Keyword("title", "doc3"));

    IndexWriter writer = new IndexWriter("c:\\index", new StandardAnalyzer(), true);
    writer.addDocument(doc1);
    writer.addDocument(doc2);
    writer.addDocument(doc3);
    writer.close();

    Query query1 = null;
    Query query2 = null;
    BooleanQuery query = null;
    Hits hits = null;

    IndexSearcher searcher = new IndexSearcher("c:\\index");

    query1 = new TermQuery(new Term("name","word1"));
    query2 = new TermQuery(new Term("name","word2"));

    // 构造一个布尔查询
    query = new BooleanQuery();

    // 添加两个子查询
    query.add(query1, false, false);
    query.add(query2, false, false);

    hits = searcher.search(query);
    printResult(hits, "word1 和 word2");

}

public static void printResult(Hits hits, String key) throws Exception
{
    System.out.println("查找 \"" + key + "\" :");
    if (hits != null)
```

```
        {
            if (hits.length() == 0)
            {
                System.out.println("没有找到任何结果");
            }
            else
            {
                System.out.println("找到" + hits.length() + "个结果");
                for (int i = 0; i < hits.length(); i++)
                {
                    Document d = hits.doc(i);
                    String dname = d.get("title");
                    System.out.print(dname + "      ");
                }
                System.out.println();
                System.out.println();
            }
        }
    }
}
```

首先构造两个 TermQuery，再构造一个 BooleanQuery 的对象，并将两个 TermQuery 当成它的查询子句加入到 Boolean 查询中。

再来看一下 BooleanQuery 的 add 方法，除了它的第一个参数外，它还有另外的两个布尔型的参数。第一个参数的意思是当前所加入的查子句是否必须满足，第二个参数的意思是当前所加入的查询子句是否不需要满足。这样，但这两个参数分别选择 true 和 false 时，会有 4 种不同的组合。

- True&false：表明当前加入的子句是必须要满足的。
- false& True：表明当前加入的子句是不可以被满足的。
- false& false：表明当前加入的子句是可选的。
- true & true：错误的情况。

本例中添加的两个子查询均使用 false& false 组合，代表要进行“或”运算。运行结果如下图所示：

```
查找 "word1 和 word2" :
找到 3 个结果
doc1    doc3    doc2
```

图 6.6 代码 6.4 运行结果

6.2.3.3 在某一范围内检索——RangeQuery

有时用户会需要一种在一个范围内查找某个文档，比如查找某一时间内的所有文档，此时，Luceue 提供了一种名为 RangeQuery 的类来满足这种需求。

RangeQuery 表示在某范围内的搜索条件，实现从一个开始词条到一个结束词条的搜索功能，在查询时“开始词条”和“结束词条”可以被包含在内也可以不被包含在内。它的具体用法如下：

RangeQuery query=new RangeQuery（begin，end，included）；

在参数列表中，最后一个 Boolean 值表示是否包括边界本身，即当其为 True 时表示包含边界，也就是可用闭区间[begin，end]表示；当其为 False 时表示不包含边界，也就是可用开区间（begin，end）表示。代码 6.5 为 RangeQuery y 的具体使用过程：

代码 6.5 RangeQuery 的使用

```
package IRLabDemo.Search;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.Term;
import org.apache.lucene.search.Hits;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.RangeQuery;

public class RangeQueryTest {
        public static void main (String [] args) throws Exception {
                Document doc1 = new Document();
            doc1.add(Field.Text("time", "200601"));
            doc1.add(Field.Keyword("title", "doc1"));

            Document doc2 = new Document();
            doc2.add(Field.Text("time", "200602"));
            doc2.add(Field.Keyword("title", "doc2"));

            Document doc3 = new Document();
            doc3.add(Field.Text("time", "200603"));
            doc3.add(Field.Keyword("title", "doc3"));

            Document doc4 = new Document();
            doc4.add(Field.Text("time", "200604"));
            doc4.add(Field.Keyword("title", "doc4"));

            Document doc5 = new Document();
            doc5.add(Field.Text("time", "200605"));
            doc5.add(Field.Keyword("title", "doc5"));

            IndexWriter writer = new IndexWriter("c:\\index", new StandardAnalyzer(), true);
        writer.setUseCompoundFile(true);
```

```
            writer.addDocument(doc1);
            writer.addDocument(doc2);
            writer.addDocument(doc3);
            writer.addDocument(doc4);
            writer.addDocument(doc5);
            writer.close();

            IndexSearcher searcher = new IndexSearcher("c:\\index");
            Term beginTime = new Term("time","200601");
            Term endTime = new Term("time","200605");

            Hits hits = null;
            RangeQuery query = null;

            query = new RangeQuery(beginTime, endTime, false);
            hits = searcher.search(query);
            printResult(hits, "从 200601 到 200605 的文档，不包括 200601 和 200605");

            query = new RangeQuery(beginTime, endTime, true);
            hits = searcher.search(query);
            printResult(hits, "从 200601 到 200605 的文档，包括 200601 和 200605");

    }

    public static void printResult(Hits hits, String key) throws Exception {
            System.out.println("查找 \"" + key + "\" :");
            if (hits != null) {
                    if (hits.length() == 0) {
                            System.out.println("没有找到任何结果");
                    } else {
                            System.out.print("找到");
                            for (int i = 0; i < hits.length(); i++) {
                                    Document d = hits.doc(i);
                                    String dname = d.get("title");
                                    System.out.print(dname + "      " );
                            }
                            System.out.println();
                            System.out.println();
                    }
            }
    }
}
```

代码中首先构造两个 Term 词条，再构造一个 RangeQuery 的对象，并将两个 Term 词条作为它的参数。构建的

Document 的 time 字段值介于 200601 和 200605 之间，运行结果如下所示：

```
查找 "从 200601 到 200605 的文档，不包括 200601 和 200605" :
找到 doc2    doc3    doc4
查找 "从 200601 到 200605 的文档，包括 200601 和 200605" :
找到 doc1    doc2    doc3    doc4    doc5
```

图 6.7 代码 6.5 运行结果

Luceue 中还有其他一些 Query 类，如：按前缀检索的 PrefisQuery、多关键字的检索 PhraseQuery、 使用短语缀的检索 PhrasePrefixQuery、相近词语的检索 FuzzyQuery 、使用通配符的检索 WildcardQuery 等，感兴趣的读者可参见 Luceue 源码。

6.2.4 查询字符串的解析——QueryParser 类

对于搜索引擎（比如：Google 和 baidu）来讲，很多情况下只需要用户输入所查询的内容，然后在单击“查询”就可以了，其余的事全部交给查询服务去处理，最后搜索引擎把检索到的结果显示出来。那么查询代理是怎样处理用户输入的符号串的呢？我们在“信息查询服务”那一章有所讲解，本节我们主要介绍一下 Luceue 的处理过程。

在 Luceue 中，这项工作就交给了 QueryParser 类来完成，它的作用就是把各种用户输入的符号串转为一个内部的 Query 或者一个 Query 组。Luceue 不但提供的 API 允许使用者创建各种各样的 Query（查询语句），而且它同时也允许通过 QueryParser（查询分析器）生成各种各样的 Query 子对象，这使得 Luceue 的查询功能更加灵活和强大。

6.2.4.1 QueryParser 的简单用法

QueryParser 类实际上就是一个解析用户输入的工具，可以通过扫描用户输入的字符串，生成 Query 对象，以下面的代码为例：

```
Query  query=null;
Query=QueryParser.parse(keywords,fieldName,new StandarAnalyzer());
```

其中，keywords 代表要查询的关键字，fieldName 代表在哪个字段进行查找，new StandarAnalyzer()代表所使用的分析器。从上面代码可以看出，当使用 QueryParser 构造 Query 对象时，不仅需要用户输入的关键字文本，还需要告诉 QueryParser 将在哪个字段内查找。如果不设置字段信息则 QueryParser 会在默认的字段内进行检索。

另外很重要的一点，就是在使用 QueryParser 对用户的输入进行扫描时，还需要给它一个分析器。有关分析器的概念将在其他章节中介绍。对用户输入的关键字进行分析的分析器应当与建立索引时的分析器一致，这样才能保证分析成功。

6.2.4.2 QueryParser 的与或

当用户输入两个关键字时，QueryParser 默认它们之间是“或”的关系。如果需要改变这种逻辑关系，可以用下面

的方法：

```
QueryParser Query=QueryParser.parse(keywords,fieldName,new StandarAnalyzer());
Query.setOperator(QueryParser.DEFAULT_OPERATOR_AND);
```

这样构建的 QueryParser 实例对用户输入进行扫描时，就会将用空格分开的关键字理解为“与”的关系，也就是构建了一个“与”关系的布尔型查询。

6.3 Luceue 中的高级检索技巧

6.3.1 对检索结果的排序

Luceue1.4 以前的版本，检索结果只能够以 Luceue 内部的评分为标准，采用降序排序的方式返回，这样与查询最相关的文档会出现在返回结果的较前面。

然而这也仅仅是最基本的排序功能。如果对查询结果有更复杂的要求，那么就不能满足使用者的需求了。例如，一项检索任务需要把查询结果按类别进行分组，然而在组内再按照与查询的相关度进行排序。既然 Luceue 只提供了基本的排序功能，那么，应该如何实现这个较高级的排序要求呢？

可以首先得到返回的查询结果，然后在 Luceue 系统之外通过自己编写的代码来实现排序的要求。这个方案确实可以解决问题，但也存在很大缺点。当返回的查询结果的数量非常大的时候，这样处理可能会成为系统性能上的一个瓶颈。那么该如何实现呢？

在本章中，将向大家介绍 Luceue 中各种各样的排序方法，包括通过一个或者多少个字段进行排序，还有降序排列和升序排列等。

6.3.1.1 使用 Sort 类排序

在 IndexSearch 类中包含很多的重载 search 方法，不过，在前面的章节中仅仅向大家介绍了一种最基本的方法 search（Query），这个方法返回的结果是按照与查询的相关度降序排列的。同时这个方法还有一个更高级的版本，使用者可以自定义排序的规则，这样就可以满足各方面的需要了，这个方法是 search（Query，Sort）。代码 6.6 为 search（Query，Sort）的具体使用过程：

代码 6.6 search（Query，Sort）的使用

```
package IRLabDemo.Search;

import org.apache.lucene.store.*;
import org.apache.lucene.document.*;
import org.apache.lucene.analysis.*;
import org.apache.lucene.index.*;
import org.apache.lucene.search.*;
import org.apache.lucene.queryParser.*;
```

```
class SortTest
{
   public static void main(String[] args)
   {
      try
      {
                  //生成索引目录
         Directory directory = new RAMDirectory();
                  //生成分词器
         Analyzer analyzer = new SimpleAnalyzer();
                  //索引书写器
         IndexWriter writer = new IndexWriter(directory, analyzer, true);
                  //将要进行检索的文本内容
         String[] docs =
         {
            "a b c d e",
            "a b c d e a b c d e",
            "a b c d e f g h i j",
            "a c e",
            "e c a",
            "a c e a c e",
            "a c e a b c"
         };
                  //循环依次建立索引
         for (int j = 0; j < docs.length; j++)
         {
                  //生成 Document 对象
            Document d = new Document();
                  //添加内容
            d.add(Field.Text("contents", docs[j]));
                  //添加到索引中
            writer.addDocument(d);
         }
                  //关闭索引
         writer.close();
                  //生成搜索器对象
         Searcher searcher = new IndexSearcher(directory);
                  //备用检索的字符串
         String[] queries = {
             "\"a b\"",
             //    "\"a b c\"",
             //    "\"a c\"",
             // "\"a c e\"",
         };
```

```
                //保存检索结果
        Hits hits = null;
                //生成分析器对象
        QueryParser parser = new QueryParser("contents", analyzer);

        for (int j = 0; j < queries.length; j++)
        {
                //生成检索对象
          Query query = parser.parse(queries[j]);
                //
          System.out.println("Query: " + query.toString("contents"));

                //排序对象
          Sort sort = new Sort();

                //进行检索，返回检索结果
              hits = searcher.search(query,sort);
          System.out.println(hits.length() + " total results");

                //循环遍历返回的检索结果
          for (int i = 0 ; i < hits.length() && i < 10; i++)
          {
                //取得文档
            Document d = hits.doc(i);
                //
            System.out.println(i + " " + hits.score(i)+ " " + d.get("contents"));
          }
        }
          //关闭
        searcher.close();

      } catch (Exception e)
              {
                System.out.println(" caught a " + e.getClass() +
                                        "\n with message: " + e.getMessage());
              }
    }
  }
```

在上述代码中，最重要的一点在与对 Searcher 类 search（Query，Sort）方法的调用，在以前的代码中都是直接调用 Searcher 类 search（Query）方法。这两个 Searcher 类的重载方法，都是负责执行查询，返回检索结果。只不过 search（Query）方法才用的是按照默认的排序方法返回检索结果，而 search（Query，Sort）方法是按照 Sort 类中定义的方法返回检索结果。代码的运行结果如下图所示：

```
Query: "a b"
4 total results
0 0.97357154 a b c d e a b c d e
1 0.96378666 a b c d e
2 0.82610285 a c e a b c
3 0.68841904 a b c d e f g h i j
```

图 6.8 代码 6.6 运行结果

关于 Sort 类的使用方法有很多种，在这里所采用的默认构造函数 Sort（）是最简单的一种处理方式。还可以在构造 Sort 对象的时候传递一个字符串类型的字段名作为参数，即采用构造函数 Sort（String field），这样返回的检索结果可以按照该字段进行排序，也可以在构造 Sort 对象的时候传递一个字符串叔祖类型的字段名组作为参数，即采用构造函数 Sort（String[] fields）这样返回的检索结果可以依次按照字段数学组中的字段进行分组和排序。

6.3.1.2 最简单的排序——相关度

Luceue 默认的是按照相关性降序排列的，这在 Luceue 中也被称为“评分机制”如果要对结果按照相关度排序，那么既可以通过给 Sort 对象传递一个 null 实现，也可以通过调用 Sort 默认的构造函数实现，按相关度排序的使用方法如下所示：

代码 6.7 按相关度排序

```
package IRLabDemo.Search;

import org.apache.lucene.store.*;
import org.apache.lucene.document.*;
import org.apache.lucene.analysis.*;
import org.apache.lucene.index.*;
import org.apache.lucene.search.*;
import org.apache.lucene.queryParser.*;

class SortTest2
{
   public static void main(String[] args)
   {
      try
      {
                  //生成索引目录
         Directory directory = new RAMDirectory();
                  //生成分析器
         Analyzer analyzer = new SimpleAnalyzer();
                  //生成索引书写器
         IndexWriter writer = new IndexWriter(directory, analyzer, true);
                  //将要进行检索的文本内容
```

```
String[] docs =
{
   "a b c d e",
   "a b c d e a b c d e",
   "a b c d e f g h i j",
   "a c e",
   "e c a",
   "a c e a c e",
   "a c e a b c"
};
                //循环遍历检索检索结果
for (int j = 0; j < docs.length; j++)
{
                //取得文档
   Document d = new Document();
                //添加检索内容
   d.add(Field.Text("contents", docs[j]));
                //添加进索引
   writer.addDocument(d);
}
                //关闭索引
writer.close();
                //生成检索对象
Searcher searcher = new IndexSearcher(directory);
            //备用检索的字符串
String[] queries = {
     "\"a b\"",
};
                //保存检索结果
Hits hits = null;
                //生成检索对象
QueryParser parser = new QueryParser("contents", analyzer);

                //依次进行检索
for (int j = 0; j < queries.length; j++)
{
                //生成检索对象
   Query query = parser.parse(queries[j]);
   System.out.println("Query: " + query.toString("contents"));
                //返回检索结果
                  hits = searcher.search(query,Sort.RELEVANCE);
   System.out.println(hits.length() + " total results");

   //遍历返回的检索结果
```

```
            for (int i = 0 ; i < hits.length() && i < 10; i++)
            {
                    //取得文档
               Document d = hits.doc(i);
                    //
               System.out.println(i + " " + hits.score(i)+ " " + d.get("contents"));
            }
         }
                    //关闭检索器
          searcher.close();
       } catch (Exception e)
                {
                   System.out.println(" caught a " + e.getClass() +
                                          "\n with message: " + e.getMessage());
                }
   }
}
```

在上述代码中，采用了一种不同的处理方式，尽管最后所返回的结果顺序仍和前面代码 13.1 相同。Sort 类有两个静态方法在进行排序处理时非常有用：

- Sort.RELEVANCE：该方法对检索结果按查询的相关度进行排序，这也是系统默认采用的处理方式。
- Sort.INDEXOREDR：该方法对检索结果按照建立索引时的顺序进行排序。

在代码 6.7 中，所采用的是前者的处理方法。在 Luceue 中“相关度”和“文档 ID”是两个特殊的类型，检索结果首先会以相关度为标准降序排列，当文档的相关度相同的时候，会以文档的 ID 为标准进行升序排列，文档 ID 是指文档被索引时的顺序。

Sort.RELEVANCE 只是 new Sort()的一种简写，在功能上没有任何不同。

6.3.1.3 按字段来对 Document 排序

在实际使用 Luceue 进行检索时 ，经常会遇到需要按照某一字段进行排序的情况，Luceue 提供了满足这一需求的方法。不过，要使用这一方法必须满足该字段是“可排序”的。例如，将“title”字段作为一个单独的“Field.KeyWord”类型进行索引，这就使得着个字段成为可排序的。

如果要通过某一字段进行排序，必须创建一个 Sort 对象，并且提供字段名作为构造函数的参数。具体使用方法如下面代码所示：

代码 6. 8 按字段排序

```
package ch13;
import org.apache.lucene.store.*;
import org.apache.lucene.document.*;
import org.apache.lucene.analysis.*;
import org.apache.lucene.index.*;
```

```
import org.apache.lucene.search.*;
import org.apache.lucene.queryParser.*;

class SortTest3
{
   public static void main(String[] args)
   {
      try
      {
         Directory directory = new RAMDirectory();
         Analyzer analyzer = new SimpleAnalyzer();
         IndexWriter writer = new IndexWriter(directory, analyzer, true);

         String[] docs =
         {
            "a b c d e",
            "a b c d e a b c d e",
            "a b c d e f g h i j",
            "a c e",
            "a c e a c e",
            "a c e a b c"
         };
         for (int j = 0; j < docs.length; j++)
         {
            Document d = new Document();
            d.add(Field.Text("contents", docs[j]));
            writer.addDocument(d);
         }
         writer.close();

         Searcher searcher = new IndexSearcher(directory);
         //备用检索字符串
         String[] queries = {
      "\"a b\"",
      // "\"a c e\"",
         };

         //保存检索结果
         Hits hits = null;
         //解析查询字符串
         QueryParser parser = new QueryParser("contents", analyzer);

         for (int j = 0; j < queries.length; j++)
         {
```

```
            //生成查询对象
            Query query = parser.parse(queries[j]);
          //
            System.out.println("Query: " + query.toString("contents"));
            //生成排序类
            Sort sort = new Sort("contents");
            //返回检索结果
            hits = searcher.search(query,sort);
            //
            System.out.println(hits.length() + " total results");

            for (int i = 0 ; i < hits.length() && i < 10; i++)
            {
            //取得文档
              Document d = hits.doc(i);
            //
              System.out.println(i + " " + hits.score(i)+ " " + d.get("contents"));
            }
          }
            //关闭
          searcher.close();

      } catch (Exception e)
            {
              System.out.println(" caught a " + e.getClass() +
                        "\n with message: " + e.getMessage());
            }
   }
}
```

Luceue 不仅可以按照相关度对检索结果进行排序，还可以按照某一字段对检索结果进行排序，就如代码 13.3 中的处理方式。

在代码 6.8 中，建立索引的过与前面的处理方式是一样的。所不同的是，在返回检索结果之前，首先生成了一个 Sort 类，并且传递了一个字段名“contents”作为构造函数的参数。然后调用 Searcher 类重载的 search（Query，Sort）方法，这样返回的检索结果就会按照字段“contents”进行排序，而不是仅仅如前面代码的处理那样按照与查询字符串的相关度来排序。

在这里按照字段排序是指按照指定的字段以字母表的排列顺序为标准，进行升序排列。还有一点需要说明的是，在按照字段对检索结果进行排序时，如果只指定了单独一个字段，那么 Sort 类会自动地将文档 ID 添加为最终排序的字段，也就是说在按照字母表顺序排序相同的文档，会按照它们各自的索引顺序进行最终的排序。

6.3.1.4 对多个字段排序

Luceue 不仅可以按照一个字段进行排序，还可以按照多哥字段一起排序。按照多个字段进行排序就是首先按照一个字段分组排序，然后，再在一个组内按照第二个字段排序。如果有更多的字段，则也是按照同样的原理进行，先依次按照字段分组排序，然后再按照最后一个字段排序。如下代码：

Sort sort = new Sort(new SortField[]{new SortField("title"),new SortField("contents")});

该代码使用了两个字段"title"和"contents"进行排序，即先依次按照 title 字段分组排序，然后再按照"contents"字段排序。

6.3.1.5 自定义排序

前面已经向大家介绍了 Luceue 的评分机制，ID 排序和通过字段值排序，如果这些仍然不能满足需求，Luceue 还实现了一种自定义的排序机制，用户可以根据自己的具体需求来定义排序方法。自定义排序方法的使用也很简单，只要实现了 SortComparatorSource 接口就可以。

6.3.2 多域检索和多索引检索

6.3.2.1 使用 MultiFieldQueryParser 来进行多域检索

在使用 Luceue 时，如果查询的只是那些 terms（词条），而不关心这些 terms 到底是在哪个字段中。这时就可以使用 MultiFieldQueryParser 这个类来进行多域检索。

这个类是构建在 QueryParser 之上的，默认情况下它会在每一个字段上使用 QueryParser 类的 parse（）方法解析一个 Query 表达式，然后将各个字段上的查询结果综合在一个 BooleanQuery 中。在将各个字段添加进 BooleanQuery。，默认采用最简单的“或”连接符，也就是说只要在一个字段上符合查询要求，这个文档就会被当作可以匹配的查询条件。具体使用方法如代码 6.9 所示：

代码 6.9 多域检索

```
package IRLabDemo.Search;
import java.io.IOException;
import org.apache.lucene.analysis.Analyzer;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.queryParser.MultiFieldQueryParser;
import org.apache.lucene.queryParser.ParseException;
import org.apache.lucene.search.Hits;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.Searcher;

public class SearchTest1
```

```
{
    public static void main(String[] args)
    {
        try
        {
            //生成搜索器
        Searcher searcher = new IndexSearcher("C:\\IndexDir");
            //生成分析器
        Analyzer analyzer = new StandardAnalyzer();
            //生成 Query 对象
        Query query = MultiFieldQueryParser.parse("Lucene", new String[]{"title","contents"}, analyzer);
            //
        System.out.println("Searching for: " + query.toString("contents"));
            //返回检索结果
        Hits hits = searcher.search(query);
            //
        System.out.println(hits.length() + " total matching documents");

        } catch (IOException e)
        {
            e.printStackTrace();
        } catch (ParseException e)
        {
            e.printStackTrace();
        }
    }
}
```

在使用进行检索时，除了可以使用运算符进行连接外，还有以下几个关键字控制每个字段的情况：

REQUIRED+FIELD：该字段中必须包含有输入的关键字

PROHIBITED_FIELD：该字段必须不包含有输入的关键字

NORMAL_FIELD：该字段采用默认的情况

6.3.2.2 使用 MultiSearcher 来同时检索多个索引

如果在系统中有多个索引文件，并且需要使用一个单独的 Query 跨索引进行检索，那么这时就要用到 MultiSearcher 了。

使用 MultiSearcher 类，就可以跨索引进行搜索，并且返回的检索结果可以按一定的顺序合并到一起。MultiSearcher 类的使用如下所示：

```
        // 建立检索对象
        Searcher[] searchers = new Searcher[2];
```

```
searchers[0] = new IndexSearcher(indexStoreB);
searchers[1] = new IndexSearcher(indexStoreA);
 // 创建一个多索引检索器
Searcher mSearcher = new MultiSearcher(searchers);
```

6.3.2.3 使用 ParallelMultiSearcher 来构建多线程检索

ParallelMultiSearcher 类是在 Luceue1.4 中新添加的一个类，它是一个可以进行多线程检索的 MultiSearcher，或者说 ParallelMultiSearcher 是 MultiSearcher 的一个可以进行多线程搜索的版本。基本的搜索过程和带有过滤的搜索过程都可以被并行化处理，但是使用 HitCollector 的搜索还不能够被并行化处理。

ParallelMultiSearcher 的使用方法同 MultiSearcher 是一样的，在此就不再举例。

6.3.3 对检索结果的过滤

- ✧ **日期过滤器----DateFilter**

在进行检索的时候，可能还对检索结果有时间上的约束。例如，只想系统返回 2000 年之后的相关内容，或者只想得到 2001-2002 年之间的相关内容，这时就应该使用 DateFilter 来过滤检索结果了。具体使用方法如代码 6.10 所示：

代码 6.10 DateFilter 的使用

```
package IRLabDemo.Search;

import java.io.IOException;

import org.apache.lucene.analysis.SimpleAnalyzer;
import org.apache.lucene.document.DateField;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.Term;
import org.apache.lucene.search.DateFilter;
import org.apache.lucene.search.Hits;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.TermQuery;
import org.apache.lucene.store.RAMDirectory;

public class DateFilterTest
{

	public static void main(String[] args) throws IOException
	{
```

```
        //     创建一个索引
RAMDirectory indexStore = new RAMDirectory();
        //生成索引书写器
IndexWriter writer = new IndexWriter(indexStore, new SimpleAnalyzer(), true);
                //取得系统当前时间
long now = System.currentTimeMillis();
        //生成一个 Document 对象
Document doc = new Document();
//  添加过去的一个时间
doc.add(Field.Keyword("datefield", DateField.timeToString(now - 1000)));
        //添加要索引的内容
doc.add(Field.Text("body", "Today is a very sunny day in New York City"));

          try
          {
                //将文本添到索引中
                writer.addDocument(doc);
                //优化索引
                writer.optimize();
                //关闭
                writer.close();
        } catch (IOException e1)
                {
                        e1.printStackTrace();
                }
         //创建检索器
        IndexSearcher searcher = new IndexSearcher(indexStore);

        //过滤应该在检索结果中保存下来的
       DateFilter df1 = DateFilter.Before("datefield", now);

        //过滤不应该在检索结果中出现的
        DateFilter df2 = DateFilter.Before("datefield", now - 999999);


        Query query1 = new TermQuery(new Term("body", "sunny"));
        //生成保存检索结果的对象
        Hits result;

                try
        {
        //使用 query1,返回检索结果
                result = searcher.search(query1,df1);
              //    result = searcher.search(query1,df2);
```

```
                System.out.println("There is "+result.length()+" Document(s) matched!");
                } catch (IOException e)
                        {
                                e.printStackTrace();
                        }

        }
}
```

代码的运行结果为：There is 1 Document(s) matched!，它是使用 result = searcher .search (query1,df1);进行检索的结果），如果使用 result = searcher.search(query1,df2)进行检索，则有 0 篇文档匹配。

DateFilter 不仅支持两端都固定的时间段（from , to），也支持一端固定，另一端不固定的时间区间。对于前者，应使用 DateFilter 类的构造函数 DateFilter(String f, Date from, Date to)

或 DateFilter(String f, long from, long to)；如果是后者，则应使用 DateFilter 的静态方法 DateFilter.after(String f , Date d) 或 DateFilter.Before(String f , Date d)。

✧ Luceue 中还有其他一些过滤方法，如：QueryFilter 查询过滤器、CachibgWrapperFilter 带缓存的过滤器等，感兴趣的读者可参见 Luceue 源码。

第七章：Web 搜索引擎中结果排序

传统的信息检索系统中，对检索结果的排序一般使用的都是“相关度排序“，但对于 WEB 搜索引擎这样一个十分复杂的系统，要给出的不是一个简单的相关度排序，而是某种反映多种因素的综合统计优先序。本章第一节将结合 Lucene 介绍传统的相关度排序，第二节介绍链接分析技术及其在搜索引擎结果排序中的应用。

7.1 Lucene 的评分机制

Lucene 中检索结果的排序使用的是传统的相关度排序技术,本节将详细分析 Lucene 中的排序方法。

在搜索完成后，需要把搜索结果返回并显示给用户，在 Lucene 中搜索结果的集合是用 Hits 类的实例来存储和表示的，下面我们首先简单介绍 Hits 类。

Hits 类中主要有以下几个经常使用的方法：

Length（）：返回搜索到结果的总数量；

Doc（int n）：返回第 n 个文档；

Id（int n）：返回第 n 个文档的内部 ID 号；

Score（int n）：返回第 n 个文档的得分。

其中，Length（）方法和 Doc（int n）方法共同使用，就可以遍历结果集中的所有文档记录。Hits 对象中的 score

（int n）方法是返回文档 n 的得分，而文档 n 的得分是怎么计算得到的呢？下面将介绍评分和 Lucene 中评分机制。

7.1.1 理解评分的概念

评分是搜索引擎中很重要的一个概念。通常情况下，当用户输入一个关键字，搜索引擎接收到信息后即可开始进行检索。对于检索到的结果，需要按一定的顺序返回给用户。因此，需要引入一种机制来对检索结果进行排序，以便更加合理地将结果返回给用户。

评分机制就是对检索结果按某种标准进行评估，然后按分值的高低来对结果进行排序。同时，对于一个商用的搜索引擎来说，评分机制是其收入来源的重要部分。例如某公司向搜索引擎缴纳一定数量的费用，则该搜索引擎就将其搜索结果中关于该公司的部分分值加大，以便能在检索结果返回给用户时让该公司获得更加靠前的位置。这种做法增加了用户浏览该公司网页和产品的机会，无形之中也给该公司带来了更大的社会影响和潜在商机。因此，评分机制从各方面来说都是相当重要的。

7.1.2 Lucene 中的评分机制

文档的得分是在用户进行检索时实时计算出来的。如果用户在建立索引时就已经将每个文档的得分计算好，那么

当用户输入任何关键字时，得分最高的文档都会被排在返回结果的最前面，这显然是不合理的。

因此，所有文档的得分应当与用户检索时输入的关键字有关，而且是实时运算的结果。所谓得分，可以简单理解为是某个关键字在某文档中出现的频率。

Lucene 中计算某个关键字对应于某文档的得分公式如下所示：

$$score\_d = \sum_t \left(tf\_q * idf\_t / norm\_q * tf\_d * idf\_t / norm\_d\_t \right) \quad (1)$$

其中，*score_d* 为文档 d 对应的分数；*tf_q* 是词 *t* 在查询词 *q* 中的 *tf* 值；*tf_d* 是词 *t* 在文档 *d* 中的 *tf* 值；$norm\_q$ 由公式 $\sqrt{\sum_t \left(tf\_q * idf\_t \right)^2}$ 计算得到；$norm\_d\_t$ 表示文档 d 中与 t 处于相同域中的 token 数的平方根；*idf_t* 由公式 $\log(numDocs / docFreq\_t + 1) + 1.0$ 计算得到，而其中 $numDocs$ 表示索引中文档的数目，*docFreq_t* 表示包含词条 *t* 的文档数。

7.1.3 Lucene 排序算法说明

Lucene 中的排序算法的实现比较简单，没有考虑网页中的互相链接关系。这种排序算法的要点如下：

（1） 查询词在一个文档中的位置并不重要；

（2） 一个文档中含有该查询词的次数越多，则该得分越高；

（3） 一个被命中的文档中，如果除了该查询词之外，其他的词越多，则该得分越低。

这种方法的不足之处也显而易见，主要有如下三点：

（1） 查询精确度不好；

（2） 没有进行链接分析，因此没有体现网页的重要性。

（3） 不能满足一些具体的应用要求，如 SEWM 评测中的主题提取任务及主页查询任务。

7.2 链接分析及其应用

随着技术发展和人们日益深入研究互联网自身结构特征，发现纯文本和网页一个明显区别就是网页间超链接存在。例如：在一个网页中写有新闻频道，其中网页间超链接是指 http://www.cctv.com，而“新闻频道”就是链接“http://www.cctv.com”在本网页中的锚文本（anchor text），它是 HTML 文本中的链接描述信息，向读者提示该链接所指向网页的性质或特征。HTML（Hyper Text Mark-up Language）语言，就是这样一种为普通文件中某些字句加上标示的语言，其目的在于运用标记（tag）使文件达到预期的显示效果。HTML 设计有一套自己的语法，通过不同的命令标识符来表示不同的字体、颜色、位置等版式。因此，不同的标签能给我们提供其中文字的重要程度。例如。常识告诉我们，在同一篇文字中，比较大的字体往往是作者比较强调的内容；而在一版（以区别“一篇”，如同报纸）内容分块、且有一定布局的文字上，放在前面和中间的应该是作者比较强调的，等等。许多著名搜索引擎在网页的预处理阶段记录了这些信息，并用于排序。如 Alta Vista，Inktomi，Excite，Infoseek 等等。

所谓的链接分析就是利用网页之间的链接信息来评判其重要性（或者相关性）的技术。常用的链接信息包含网页的出度、入度、锚文本内容等。常用的链接分析算法有：PageRank、HITS、SALSA、PHITS、Bayesian 等。其中，在主题提取时，我们就采用了 HITS 来实现。关于 HITS 具体算法实现过程在后面章节还有更详尽的叙述。

由于站点之间链接和被链接关系的存在，在互联网中形成了一种与传统文献引证和被引证关系十分相似的情况。WEB 是由链接点和链组成的资源网络，而链接点是超文本表达信息基本单位，表现形式可以是一个网站，网站中的某个频道，某个网页文件，文件中的段落、图像，甚至是电子邮箱。链用来连接相关的链接点，揭示链接点之间的关系，是超文本的灵魂。两个网页之间建立链接，一个是主动实施链接的网页，即施链接网页，另一个则为被链接网页。根据施链网页与被链接网页是否处于一个主机上，可以将某一站点的链接分为站内链接和站外链接两种。目前，对站内链接有几种分类方法，如 Trigg 将链接类型分为普通链接和评论性链接两个类；Joslyn 将链接类型分为实质链接与语义链接两大类；基于图形的超媒体系统（GBH）中的链分为推理链和导航链两大类；槐鹤玲指出网页之间链接关系包括相似、例如、因果、时间、行为或性质等。而站外链接类型可分为推荐链接、合作链接、相关链接、资源链接、通讯链接、广告链接。各种各样的超链接存在于网页，相互间的这种链接关系可以看成一个巨大的有向图。

研究表明，利用链接分析能够高效排序网页。这是因为链接反映的是网页之间形成的“参考”、“引用”和“推荐”关系。可以合理的假设，若一篇网页被较多的其他网页链接，则它相对较被人关注，其内容应该是较重要、或者较有

用。因此，可以认为一个网页的“入度”（即指向它的网页的个数）是衡量它重要程度的一种非常有意义的指标。这和科技论文的情况类似，被引用较多的就是较好的文章。同时，人们注意到，网页的“出度”（即从它连出的超链个数）对分析网上的信息的状况很有意义的，因此可以考虑同时使用这两个指标来衡量网页。这些想法即是斯坦福大学 Google 研究小组和 IBM 公司的 Clever 系统开发小组几乎在同一时间分别提出著名的 PageRank 技术和 HITS 技术的基础。但网页无论是在质量、用法、引用和长度上都与学术论文有很大区别，所以简单根据入度，并不能很好的说明网页重要性。这是因为：网页不像学术论文那样被严格的审阅过，几乎都是免费的。而且，通过使用简单的编程工具就可以产生成千上万的网页，任何评测网页质量高低的方法都可以被操纵。最近几年的 TREC 检索评测结果表明，在网页检索中过度依赖超链接分析算法结果往往适得其反。于是网页超链接的另一类信息——锚文本便引起人们的关注。大量实验证明，在链接分析中引入锚文本利于网页排序。

为什么引入锚文本利于网页排序呢？通过大量观察和研究发现，锚文本之所以对提高网页检索质量有帮助，是因为它在形式、结构和内容上都与网页 TITLE、用户查询词非常相似，三者均具有以下几个结构特征：

（1）形式上，三者非常简短，基本都是能概括说明某一事物的词语，这样如果用户的查询词和锚文本基本相符，这样查询到目标网页的概率就非常高。

（2）语法上，锚文本与用户的查询词都很少用停用词和高频词，并且一般的查询很少用形容词、动词，而锚文本语法格式也与此非常类似。

（3）内容上，许多用户基本都用简短的词简单概括所要查询内容。我们可以使用“概念”这样的一个词来表示用户查找的内容，而目标网页和描述它的锚文本关系，即为相同概念不同描述，汇总来源于不同源网页的目标网页锚文本形成了目标网页的一个简单摘要。锚文本在内容上优于题目，是因为锚文本是由不同的作者所编写，更能客观的描述目标网页内容，而题目仅仅有一个作者编写，具有主观性。

不同作者编写的源网页，会对目标网页作出不同的评价，在某种程度上增强了锚文本内容客观性，但作者编写水平不一，也可能存在锚文本在一定程度上错误描述目标网页的情况。源网页作者的主观理解可能会造成锚文本对目标网页的评价有失偏颇，所以要综合利用网页入度和锚文本才可以对网页相对重要性做出合理评价。

互联网是一个跨越地域与国界的全球性网络，链接关系反映的是网页在全世界范围内的影响力，这样广的范围是传统问卷调查法所望尘莫及的。直接利用搜索引擎来检索某一网页被链情况，这可大大加快了数据收集、积累和处理的过程。并且这些搜索引擎绝大部分都是免费的，其检索结果是一个客观的数字，易于操作、经济实用，有效地弥补了同行评议等定性方法的缺陷。链接分析法借鉴了引文分析法的方法和思路，是情报学研究方法在网络时代的新应用。而搜索引擎的发展必然为链接分析法的深入研究和应用提供更为有力的支持。

从整体上大体了解了链析法的一些基础知识之后，接下来我们将详细讨论学习链接分析法的具体三种算法介绍。

链接分析技术主要是指利用网页间存在的各种超链指向，对网页之间的引用关系进行分析，依据网页链入数的多少计算该网页的重要度权值，一般认为，如果 A 网页有超链指向 B 网页，相当于 A 网页投了 B 网页一票，即 A 认可 B

网页的重要性。深入理解超链分析算法，可以根据链接结构把整个网页文档集看作一个有向的拓扑图，其中每个网页都构成图中的一个结点，网页之间的链接就构成了结点间的有向边，按照这个思想，可以根据每个结点的出度和入度来评价网页的作用。

链接分析技术较有代表性的算法主要是 Larry Page 等人设计的 PageRank 算法和 Kleinberg 创造的 HITS 算法，以及由以色列学者 R.Lempel 和 S.Moran 于 2000 年在第 9 届国际互联网大会上提出的 SALSA（Stochastic Approach for Link-Structure Analysis）算法。其中 PageRank 算法在实际使用中的效果要好于 HITS 算法，这主要是由于以下原因：PageRank 算法可以一次性、脱机并且独立于查询地对网页进行预计算，以得到网页重要度的估计值，然后在具体的用户查询中，结合其他查询指标值再一起对查询结果进行相关性排序，从而节省了系统查询时的运算开销。此外，PageRank 算法是利用整个网页集合进行计算的，不像 HITS 算法易受到局部连接陷阱的影响而产生“主题漂移”，所以现在这种技术广泛地应用在许多信息检索系统和网络搜索引擎中，Google 搜索引擎的成功也表明了以超链分析为特征的网页相关度排序算法日益成熟。关于二者区别在后面还将有详细介绍，接下来将分别介绍一下上述三种链接分析算法。

7.2.1 基于随机冲浪模型的 PageRank

PageRank 可以被比作一个“随机冲浪”模型。该模型可以作为 PageRank 的理论基础，它描述网络用户对网页的访问行为，假设如下：

（1）用户随机的选择一个网页作为上网的起始网页；

（2）看完这个网页后，从该网页内所含的超链内随机的选择一个页面继续进行浏览；

（3）沿着超链前进了一定数目的网页后，用户对这个主题感到厌倦，重新随机选择一个网页进行浏览，如此反复[1]。

依据以上的用户模型行为模型和引文分析法原理（所谓“引文分析法”，是对文献进行定量评价最有名的方法之一。它主要依据和研究内容是科学文献之间的引证与被引证关系），我们可以认为每个网页可能被访问到的次数越多就越重要，利用网页“可能被访问的次数”作为衡量网页相对重要性的一个标准，即 Page Rank 值。PageRank 将各个网页间的链接关系构成一个有向图，通过多次迭代进而得到网页 PageRank 值。简单地说，PageRank 就是要从链接结构中获取网页的重要性，而网页的重要性决定着同时也依赖于其他网页的重要性。

那么实际中是如何计算 PageRank 值的呢？

根据上面的基本原理，L.Page 等给出 PageRank 的简单定义：令 u 为一个网页，N(v)表示从网页 v 向外的链接数目，B(u)表示链接到网页 u 的网页集合，R(u)表示网页 u 的 PageRank 值，C 为规范化因子，作用是保证所有网页的 PageRank 总和为常量。例如为保证总的 PageRank 值为 1，可以通过网页 PageRank 总和的倒数求得。原始简单的计算 PageRank 方法是将每个网页 PageRank 值平均分配给它所指向的每个网页，其计算公式为

$$R(u) = c \sum_{v \in B_u} \frac{R(v)}{N_v} \tag{2}$$

必须注意的是上述定义有一个假设前提，即所有的网页形成了一个牢固的链接图（即每个网页能从其他网页通过超链接达到），从公式（2）可以看出，网页 PageRank 是一个由网络的超链接结构所产生的一个网页重要性等级值，所有网页的PageRank 值都可以根据其他网页的PageRank 值和链接的数量来计算得到，即所有链接到它的网页的PageRank 值除以各自向外的链接数的商进行求和。图 1 为链接结构中的部分网页和其 PageRank 值的图例，很明显其 PageRank 的分布是符合定义的。

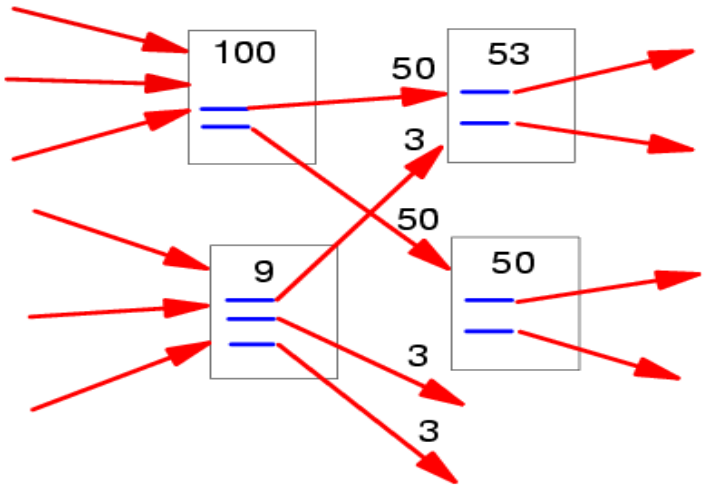


图 1 简单 PageRank 计算

为了更加容易地理解 PageRank 的定义，可以用非常直观的模拟冲浪模型来进行解释。假设一个网络冲浪者通过随机的点击超链接在网上冲浪，在前面的假设前提下（牢固的链接图），每个网页都是可能达到的，只不过是达到的可能性的大小不同。很显然，链接到哪个网页的超链接多，那么到达哪个网页的可能性大，这个网页就相对重要，PageRank 值也就高。而重要的网页链接到的网页，冲浪者到达的可能性当然也就大，其 PageRank 值也就相对高。同时也可见重要性值（PageRank）是整个网页的一个重要性概率分布结果，所以所有网页 PageRank 的总和应该是 1。

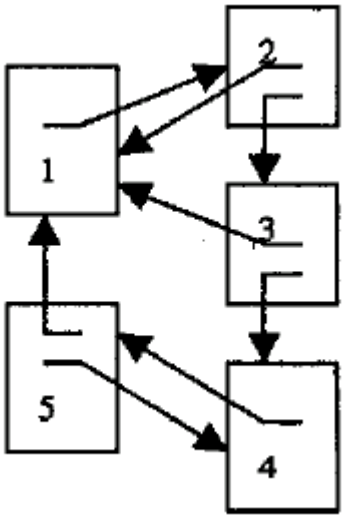


图 2 网页通过超链接的相互关系

前面所定义的 PageRank 有一个假设前提，就是所有的网页形成一个牢固的链接图。但是实际的网络超链接环境没有这么理想化，存在着两个主要问题：等级沉没(rank sink)和等级泄漏(rank leak)。整个网页图中的一组紧密链接的网页如果没有外出的链接就产生等级沉没，一个独立的网页如果没有外出的链接就产生等级泄漏。rank sink 中，不在 sink 中的网页的等级值变成了 0，即意味着不能判断出这些网页的重要性，例如图 2 中网页 5 到网页 1 的链接被删除,那么使得网页 4 和 5 产生了沉没（sink)。一个随机的冲浪者访问的时候将在 4 和 5 间陷入其中，没有出去的链接可寻，那么 1、2 和 3 的 PageRank 值都成了 0，网页 4 和 5 的 PageRank 值都变成 0.5。而 rank leak 将会丢失其它所有的等级。

针对这些问题，PageRank 的发明者采用相应的方法来解决。

对于 rank leak 可以删除所有的 leak 节点，这样 leak 节点将没有了 PageRank。另外一种方法是假设 leak 节点对所有的指向它的链接有相应的返回链接。为了解决 sink，引进了一个“rank source”（等级源）E 来不断地补充每个网页的 PageRank 值，以使得 PageRank 的分配不完全依赖着链接。修正的 PageRank 定义为：

$$PR(A) = (1-d) + d\,(PR(T1)/C(T1) + ... + PR(Tn)/C(Tn)) \qquad (3)$$

公式中各个参数意义为

PR(A)：网页 A 页的 PageRank 值；

PR(Ti)：指链接到 A 页的网页 Ti 的 PageRank 值；

C(Ti)：网页 Ti 的出站链接数量；

d：代表衰减系数。在这个修正的定义中，网页 PageRank 值仅 d 部分在它所链接到网页中分配，剩下的部分用在整个网络的所有网页中分配，d 的经验值一般取 0.85 左右。这样一个网页的 PageRank 值的 85%分配到所链接的网页，另外的 20%被分配到全部的网页中去。可以看出简单定义其实是 d=1 的时候的特例。在冲浪模型中的解释是冲浪者偶尔会感到厌倦，便跳动一个随机的网页（而不是通过所在的网页的链接），d 就表示冲浪者多少时间会感到厌倦。

可见，首先 PageRank 并不是将整个网站排等级，而是以单个页面计算的。其次，页面 A 的 PageRank 值取决于那

些连接到 A 的页面 PageRank 递归值。

如果使用公式（3）简单计算 4 个网页构成的有向图，经过迭代多次收敛得到 PageRank 值见表 1。

表 1 简单改进 PageRank 计算

OutDegree	UrlNum	LinkUrl	PageRank
3	1	2 4 3	1.4127752
1	2	3	0.5502674
1	3	1	1.4856844
1	4	3	0.5502674

Lawrence Page 和 Sergey Brin 在不同的刊物中发表了两个不同版本的 PageRank 的算法公式。在第二个版本的算法里，页面 A 的 PageRank 值是这样得到的：

$$PR(A) = (1-d) / N + d (PR(T1)/C(T1) + ... + PR(Tn)/C(Tn)) \quad (4)$$

这里的 N 是整个互联网网页的总数。很多试验是离线对网页进行 PageRank 值计算，并且网络更新和膨胀速度之快，根本无法模拟整个网络，在试验过程中的 N 取的是网页集中网页总数。公式（4）并不是完全不同于公式（3）。随机冲浪模型中，算法 2 中页面的 PageRank 值就是在点击许多链接后到达这个页面页面的实际概率。因此，互联网上所有网页的 PageRank 值形成一个概率分布，所有 RageRank 值之和为 1。将上述有向图利用公式(4)重新计算得到新的 PageRank 值见表 2。

表 2 加入网页总数的 PageRank 计算

OutDegree	UrlNum	LinkUrl	PageRank
3	1	2 4 3	0.35324374
1	2	3	0.1375835
1	3	1	0.371471
1	4	3	0.1375835

PageRank 技术的创始人 Lawrence Page 和 Surgey Brin 在相关文献指出，在计算没个网页 PageRank 值时，由于不能取样整个互联网，所以可以在某一个封闭集合中根据网页出度和入度客观计算网页 PageRank，来评价网页相对重要性。

7.2.2 HITS 算法

HITS 算法是一种有效的基于超链接结构分析的主题提取方法，它引入了 Hub/Authority 概念。Authority 网页是指与查询词的匹配度高的重要性网页，换言之，由传统搜索引擎基于查询词匹配度计算出的得分高的网页。而 Hub 页面是指包含了指向 Authority 页面链接集合的网页，它本身可能与查询词的匹配度并不高，即它本身并不是 Authority 网页，它只是一个 Authority 网页链接的汇聚点。一般来说，好的 Hub 页面指向很多好的 Authority 页面；而好的 Authority 页面会被很多好多 Hub 页面所指向。Hub 页面和 Authority 页面这种互相加强的特征，构成了 HITS 链接结构分析的基本依据。

HITS 算法的核心就是建立一张超链接的邻接关系图，通过分析该图的链接关系，对每个网页节点的 Hub/Authority 值进行迭代计算，从而得到 Hub 页面和 Authority 页面。具体来说，对某个主题，为某个网页集合中的每一个网页文档 p 计算 authority 值和 hub 值，分别用 A(p)和 H(p)表示网页 p 的 authority 值和 hub 值，A(p)定义为所有指向 p 的页面 q 的中心权重 H(q)之和，H(p)定义为所有 p 所指向的页面 q 的权重 A(q)之和，这种迭代关系如下式所示：

$$A(p)=\sum_{q_i} H(q_i)(\text{其中}q_i\text{是所有链接到}p\text{的页面}) \qquad (5)$$

$$H(p)=\sum_{q_j} A(q_j)(\text{其中}q_j\text{是所有页面}p\text{所链接到的页面}) \qquad (6)$$

HITS 算法常常和已有的文本检索系统配合使用：假设某个文本检索系统（例如搜索引擎）收到查询请求后返回一个按照相关度排序的相关页面集合，HITS 算法取该集合中的前 r（比如 r=200）个页面作为算法的根页面集合（root set）R，然后将 Web 中指向这 r 个页面和从这 r 个页面指出的页面都扩展进来，得到算法迭代所需的封闭网页集合 R'，将 R' 中的每隔网页视为图中的一个顶点，则这些页面之间的超链接就可看成图的边。

Kleinberg 已经从代数上证明了上述算法收敛，可以利用 hub 值和 authority 值对页面进行排序。HITS 算法虽然不能找出所有的相关页面，但是 hub 和 authority 之间是一种互增强关系，一个好的 hub 必然指向许多好的 authority，同样一个好的 authority 必然被许多好的 hub 链接。受到该算法的启发，我们发现可以把这种互增强的思想用到主题信息提取的关键资源的寻找上面，而且在关键资源的寻找的算法中，还可以避免由于邻接矩阵规模太大而导致的计算效率低下的问题。

7.2.3 SASLA 算法介绍

1998 年以前，全球范围内的搜索引擎在网页结果排序上多基于关键字密度算法。由于自然语言本身的特点，搜索引擎难以处理同义、歧义及作者表达风格迥异等问题，同时，专门针对该类算法弱点的“恶意关键字”行为（word spamming，即有意提高关键字出现的频率来提升网站在搜索引擎中的排序，破坏搜索引擎结果的客观性和公正性）在因特网上也非常普遍，因此检索结果很难令人满意。1998 年，Brin 和 Page 率先发明了基于链接分析的 PageRank 算法，并籍此成功地创建了后来名冠全球的 Google 搜索引擎，从此为搜索引擎的历史翻开了新的一页，Google 被称之为第二代搜索引擎的代表而备受推崇。一时间，大批有志之士对基于链接分析的网页排序算法投入了巨大的热情，SALSA 算法正是在这种氛围中产生的。

SALSA 算法由以色列学者 R. Lempel 和 S. Moran 于 2000 年在第 9 届国际互联网大会上提出，其英文全称为 Stochastic Approach for Link Structure Analysis，它是在深刻研究了 HITS 算法并在此基础上提出来的，因此国内有学者将其归并到近似 HITS 算法的一类中。

从理论基础上看，SALSA 算法基于概率论中的马尔可夫链原理，并依赖于数据收集时的“随机漫游”（random walk）的随机特性。为更好地理解 SALSA 算法，这里有必要简单介绍马尔可夫链的概念。简而言之，马尔可夫链是事件的一种概率模型，其中一个事件的概率完全取决于紧接它前面的事件，我们用 Xi 来表示某事第 i 次发生的几率，假设已

知道第 0 次到第 n 次的试验结果，则第 n + 1 次发生的概率仅和第 n 次有关，而与之前的 n - 1 次结果无关，此随机过程（X1、X2、X3…Xn+1）称为马尔可夫链（Markov chain）。马尔可夫链是概率论中随机过程的重要过程之一。

R. Lempel 和 S. Moran 在 SALSA 算法中，仍然沿用了 HITS 算法的两个基本概念，将网页分为两类，即 hubs 和 authorities。authorities 为具有较高价值的网页，依赖于指向它的页面；而 hubs 为指向较多 authorities 的网页，依赖于它所指向的页面。对每个页面而言，它们也有两个权重（weight），即 hub weight（中心权重）和 authority weight（权威权重）。与此同时，SALSA 算法更强调 Web 用户浏览的随机性及向前浏览网页的直觉知识（intuition），借鉴了 PageRank 算法的随机冲浪（random surf） 思想，同时摈弃了 HITS 算法所描述的 hub 与 authority 相互增强（mutual reinforcement，简称 MR）的方法。

7.3 PageRank、HITS、SALSA 算法比较分析

首先，我们先来比较分析一下 HITS 和 PageRank 算法。显而易见，这两种算法均是基于链接分析的搜索引擎排序算法，并且在算法中均利用了特征向量作为理论基础和收敛性依据。但两种算法的不同点也非常明显，下面就主要谈谈二者不同点：

（1）从算法思想上看，虽然均同为链接分析算法，但二者之间还是有一定的区别。HITS 的原理如前所述，其 authority 值只是相对于某个检索主题的权重，因此 HITS 算法也常被称为 query - dependent 算法。而 PageRank 算法独立于检索主题，因此也常被称为 query - independent 算法。PageRank 的发明者（Page 和 Brin）把引文分析思想借鉴到网络文档重要性的计算中来，利用网络自身的超链接结构给所有的网页确定一个重要性的等级数。当然 PageRank 并不是引文分析的完全翻版，根据因特网自身的性质等，它不仅考虑了网页引用数量，还特别考虑了网页本身的重要性。简单地说，重要网页所指向的链接将大大增加被指向网页的重要性。

（2）从权重的传播模型来看，HITS 是首先通过基于文本的搜索引擎来获得最初的处理数据，网页重要性的传播是通过 hub 页向 authority 页传递，而且 Kleinberg 认为，hub 与 authority 之间是相互增强的关系；而 PageRank 基于随机冲浪（random surfer）模型，可以认为它将网页的重要性从一个 authority 页传递给另一个 authority 页。

（3）从处理的数据量及用户端等待时间来分析。表面上看，HITS 算法对需排序的网页数量较小，所计算的网页数量一般为 1000 至 5000 个，但由于需要从基于内容分析的搜索引擎中提取根集并扩充基本集，这个过程需要耗费相当的时间，而 PageRank 算法表面上看，处理的数据数量上远远超过了 HITS 算法。据 Google 介绍，目前已收录的中文网页已达 33 亿以上，但由于其计算量在用户查询时已由服务器端独立完成，不需要用户端等待，基于该原因，从用户端等待时间来看，PageRank 算法应该比 HITS 要短。

接下来，我们探讨一下 SALSA 与 PageRank 二者联系与区别。二者均采用了随机漫游模型，但显然它们的不同点更多。

首先，二者的“随机漫游”也有明显不同，PageRank 是单条线路的漫游；而 SALSA 则是两条线路的漫游，PageRank 算法将网页权重从一个 authority 页传递给另一个 authority 页，而 SALSA 是将网页权重从 hub 页传递给 authority 页。

其次，PageRank 算法独立于检索主题，因此也常被称为 query-independent 算法，而 SALSA 算法的 authority 值只是相对于某个具体检索主题的权重，因此也常被称为 query-dependent 算法；PageRank 算法的计算是全局性的，它尽可能地对搜集到的网页进行计算排序，包括内在链接如站点地图等都进行计算，且需要多次重复迭代计算，因而计算量大，而 SALSA 算法仅对某一个主题的结果集 GR 进行计算排序，其数量一般为几千，而且其算法本身的计算特别简单，计算量很小；但从客户端响应时间来看，PageRank 算法在用户提交查询式以后，由于 PageRank 值已预先计算好，服务端只需作一个查询任务并对检索结果按 PageRank 值大小进行排列即可，而无论是 SALSA 算法还是 HITS 算法，均还需在查询的基础上再进行排序算法的计算，由于有两个步骤需要完成，因而理论上说耗时要多一些。

最后，让我们来分析一下 salsa 与 HITS 关系。前已述及，SALSA 算法与 HITS 算法有明显的渊源关系，二者使用的基本概念术语甚至算法设计的部分步骤均相同，当然，二者的区别也很明显。在 HITS 算法中，Kleinberg 认为 hub 与 authority 是相互增强关系，HITS 算法也因此被称为相互增强方法，正由于这一点，导致了密织群现象（Tightly-Knit Community Effect，简称 TKC Effect，即一小部分网站尽管对某主题不具权威性或仅为该主题的某一方面，但由于它们之间相互高度紧密链接，从而达到很高的 authority 值）的出现，从而暴露出 HITS 算法的脆弱性；而 SALSA 算法正是为了避免该现象的发生而提出的，因此，SALSA 算法比 HITS 更具健壮性。从计算量上分析，SALSA 算法的计算量比 HITS 算法明显要小；从网页作用范围来看，SALSA 算法只考虑直接相邻的网页对其的影响，而 HITS 算法是计算整个结果集 GR 对其的影响。

综上所述，基于链接分析的网页排序算法，目前的研究都还很不成熟，无论是 PageRank 算法，还是 HITS 算法，已有众多的国内外学者在算法的改进方面做出了努力。值得关注的是，目前已有学者对这两种算法相结合的可能性作了理论上的探讨。

7.3 链接分析在 SEWM 评测中的应用

7.3.1 SEWM 评测介绍

SEWM 是搜索引擎与网上信息挖掘（Search Engine and Web Mining）的简称。其中的一个任务是中文 Web 信息检索评测，采用 CWT200g 作为测试平台。希望为这个领域的研究人员、用户、企业提供一个交流的机会，在国内外各个研究小组的共同参与下一起推动中文 Web 信息检索技术的发展。

中文 Web 检索评测有两个子任务，即主题提取（TD）和主页/指定页面查询（HPNP）。

其中主题提取任务的目的是对于一个特定主题发现一组关键资源，注重以站点入口作为关键资源。如对于主题“Linux”，下面站点可能被认为是关键资源：http://www.redflag-linux.com/。返回页面如果被判断为是一个关键资源，应该是一个站点的好的首页面。判断是否一个好的首页面，考查三个方面：

（1） 是否大部分切合主题；

（2） 提供主题的可靠的信息；

（3） 不是一个更大的切合主题站点的一部分。

因此，对“Linux”这一主题，页面 http://www.redflag-linux.com/chanpin/Desktop/index.html 不符合第三个条件。

主页查询的返回结果往往是一个网站的主页，如查询词“内蒙古民族大学”对应的查询结果是 www.mzdx.com。

7.3.2 Hits 算法在主题提起中的应用

主题提取的任务不仅要找到相关联的页面，而且还要从这些相关联的页面中找到真正的资源中心。这些资源中心（key resource）可能从内容上讲不是最好的（相关但不一定是排名最靠前），但却是和查询最相关的关键资源的入口点。通过分析我们发现，如果从 HITS 算法角度来衡量的话，关键资源页面从它的定义上来看，应该是 hub 值较高的枢纽页面，因为它是指向与查询相关联的资源中心。如何将 HITS 的算法有效地应用于主题信息提取的关键资源（key resource）的查找呢？下面将介绍几种算法。

7.3.2.1 链接分析直接应用基本算法

这种方法是在内容检索的基础上直接应用基本的 HITS 算法，最后根据 hub 值的高低来确定关键资源。由于基本的 HITS 算法对于根集合的扩展是在整个 web 页面（即所用的整个的语料集合）上进行的，所以噪音页面就会比较多。为了解决这个问题，在根集合的扩展上我们对算法进行了改进，具体的算法如下：

（1）对于某一个查询，取内容检索返回的前 m 个结果作为初始集合，记为 M；从 M 中取前 r(r<m)个结果构成根集合 R；

（2）对于 R 中的每个顶点，可以根据超链接的关系按照如下规则在 M 中进行扩展；

规则 1：对于 $\forall p \in R$，如果 $\exists q$，(q, p) 是超链接，且 $q \in M$，则将 q 扩展进根集合 R

规则 2：对于 $\forall p \in R$，如果 $\exists q$，(p, q) 是超链接，且 $q \in M$，则将 q 扩展进根集合 R

（3）对于集合 R'，用向量 H、A 分别记录其中所有网页的 hub 值和 authority 值；

（4） 利用基本的 HITS 算法计算 R' 中每个元素的 hub 值和 authority 值；

（5） 取 hub 值前 k 名的页面放入集合 K 作为关键资源输出。

由于这个算法在初始集合的选取上，是从比较相关的页面中来进行根集合的扩展，通过指定相应的阈值，既控制了根集合的规模，又保证了扩展进根集合的页面从内容检索的角度讲是与主题相关的，因此在一定程度上控制了噪音页面的数量，使计算出来的 hub 值和 authority 值能够更加客观地反映页面的质量。与传统的 HITS 算法相比，降低了算法迭代的规模，减少了迭代次数。

7.3.2.2 对内容检索的结果分类，分别使用基本算法

在分析了基于内容的检索结果之后，我们发现很多排名比较好的页面往往来自于同一类，而这些页面之间又有着稠密的链接关系（可以看成一类页面）。所以我们考虑在关键资源（key resource）的提取过程中，可以针对每一类页面来选取根集合，然后对每一类页面都利用 HITS 算法，输出 hub 值最高的那些页面作为主题信息提取的关键资源（key resource）。具体算法如下：

（1）对于某一个查询，从基于内容的检索结果中取前 m 个返回结果作为初始集合，记为 M。将 M 按照页面性质分成 t 个集合 C_1，C_2，C_3，……，C_t；

（2）对于 $\forall C_i \in C_1, C_2, C_3, \cdots\cdots, C_t$ 中的每一类做如下计算：取 C_i 的前 r 个作为根集合 R_i 执行 7.3．1 中的算法在 M 中对 R_i 进行扩展，得到扩展集合 R_i'。然后对于扩展后的集合 R_i' 应用基本的 HITS 算法来计算 hub 值，并取 hub 值最大的 k_i 页面加入集合 K_i；

（3）返回集合 $K = \bigcup_{i=1}^{t} K_i$，并将 K 中元素按 hub 值排序，输出前 k 个结果。

在这个算法中，我们的想法是这样的：我们希望在同一类页面中利用链接之间的这种互增强关系来找到其中的关键资源。从上面的算法我们可以看到，其中对于根集合的扩展无论发生在哪里都是在考虑了内容检索的基础之上的，也就从一定程度上降低了扩展后的根页面集合的噪音页面的数量。从关键资源的定义我们可以看到，实质上我们所找的关键资源其实是能够标引这一类资源的入口点页面，从内容上讲，它是与查询高度相关的，从结构上讲，它又是许多权威页面的入口点。所以在对基于内容的前 m 个页面进行分类的时候考虑采用了两种策略：

策略 1：为了叙述上的方便，我们将这 m 个页面构成的 web 子图记为 $G=(V_m,E_m)$，其中 V_m 表示顶点页面的集合，E_m 表示这些页面之间的有向边的集合。在这个策略中分类的原则是将拓扑结构上连接紧密的页面聚成一类，在计算的过程中只考虑链接的有无而没有考虑链接的方向，所以我们把 web 子图 G 构成忽略了方向的无向图对应的邻接矩阵 M 中的元素是按照如下方式定义的：

$$m_{ij} = \begin{cases} 1 & if \langle i,j \rangle or \langle j,i \rangle \in E_m \\ 0 & otherwise \end{cases} \quad (7)$$

有了这样一个矩阵之后，我们就可以利用聚类算法得到在拓扑结构上连接紧密的一个个的团，这里也就使将 M 分成 t 个集合 C_1，C_2，C_3，……，C_t。

策略 2：这个分类的原则比较简单，只是按照 URL 的属性对 M 中的页面进行分类，来自于同一个站点的页面就分成同一类别，将 M 分成 t 个集合 C_1，C_2，C_3，……，C_t。

这个算法利用了主题信息在结构上显现出来的 Linkage Locality 特性（主题页面有较大可能链接同主题的页面）。对于来自于同一类页面找出其中的关键资源，由于对于内容检索之后的页面进行了分类，所以算法的迭代规模在最坏的情况下也只是 7.5.1 中算法迭代规模的 1/k（假设将页面分成了 k 类），对每一类页面通过它内部的链接结构计算出该

类别中最重要的关键资源。从本质上讲，通过分类，在通过结构信息提取关键资源的同时也考虑了内容的因素。这样可以保证提取出来的关键资源不会是重复的。

7.3.2.3 将结构信息与内容信息相结合

这种方法是结合结构和内容两个方面来进行，具体的算法如下：

对于某一个查询，取内容检索返回的前 m 个结果作为初始集合，记为 M；从 M 中取前 r(r<m)个结果构成根集合 R；

（1） 对于 R 中的每个顶点，可以根据超链接的关系按照如下规则在 M 中进行扩展；

规则 1：对于 $\forall p \in R$，如果 $\exists q$，(q, p) 是超链接，且 $q \in M$，则将 q 扩展进根集合 R

规则 2：对于 $\forall p \in R$，如果 $\exists q$，(p, q) 是超链接，且 $q \in M$，则将 q 扩展进根集合 R

（2） 对于集合 R'，用向量 H、A 分别记录其中所有网页的 hub 值和 authority 值；

（3） while(当向量 H 和 A 都不收敛)

{对所有的 $p \in R'$，$A(p) = \sum_{p_i} H(p_i) * \alpha$（其中 p_i 是链接到页面 p 的页面）

对所有的 $p \in R'$，$H(p) = \sum_{p_j} A(p_j) * \beta$（其中 p_j 是页面 p 链接到的页面）

标准化向量 H 和 A.

}

（4） 取 hub 值前 k 名的页面放入集合 K 作为关键资源输出。

其中，α 是页面 p_i 通过内容检索之后得到的一个与查询相关程度的权值，β 是页面 p_j 通过内容检索之后得到的一个与查询相关程度的权值。α 和 β 的值可以由如下方式得到，假定页面 p_i，p_j 基于内容的得分为 a，b，则取 $\alpha = a / in\_\deg ree(p)$，$\beta = b / out\_\deg ree(p)$。最后在计算关键资源时，由于要考虑到内容的因素，因此，对于扩展后根集合里的某个页面 p，可以按照下面的公式来计算关键资源的重要程度：

$$score(p) = \frac{k_1 * hub(p) + k_2 * authority(p)}{rank(p)} \quad (8)$$

其中，k_1 和 k_2 是两个可以调节的参数，且 $k_1+k_2=1$，rank(p)是页面 p 在内容检索结果中的排名。

这个算法是在 7.5.1 中算法的基础上改进的，因为考虑到内容检索直接影响到检索的质量，所以该算法在基本 HITS 算法的 hub 和 authority 值的计算过程中就考虑了内容的因素在里面，在计算关键资源的重要性时，考虑到 authority 的值越大，说明该页面是资源中心的可能性就越大，页面对于相关的查询来讲质量就越好。所以将它作为反映页面是否为关键资源的因素之一考虑到重要性的计算中。rank(p)的引入也是为了调节结构和内容在最后结果中所占的比例。

7.4 链接分析方法局限性及其发展前景

虽然链接分析法有广泛的应用空间，但是我们不能不认识到现有的引文分析方法在很多方面还不尽如人意。

（1）链接分析法的理论基石还不够稳固

由于链接分析法兴起不久，而且现有的研究也以尝试性应用研究为主，而支撑链接分析法有效性的理论还不够成熟。尤其是当我们参考了很多文献计量学的理论和方法的时候，如果不对网页与文献、链接与引文等的区别进行深入分析和探讨，就很可能犯不顾对象、简单套用的错误。

(2)链接自身的复杂性带来相当多的障碍

链接的数量巨大、动机多样、变化迅速、设置随意等特点，给链接分析法的有效性、实用性和可操作性提出了挑战。

(3)对分析工具的依赖和分析工具存在的问题

网络是科技进步的产物，而对网络信息进行研究，也常常需要一定科技含量的工具，如搜索引擎、网络数据库和数学统计分析软件等。如果缺乏这些工具，而采用传统的人工方法来面对浩如烟海的网络信息资源是完全不可想象的。然而现有的工具主要存在三方面的问题：容纳量有限、测算维度少且结果粗糙、性能不稳定。可能仅对于前两方面，人们认识较为深刻，而性能不稳定的问题其实也相当突出。

虽然存在一些有待解决的问题，但链接分析由于其在有效性、科学性和可操作性方面的突出优点，相信将会在智能处理等领域得到广泛应用，并在发展中走向完善。

参考文献：

[1] Lin S-H，Ho J-M. Discovering informative Content Blocks from Web Documents [A]. Proceedings of the ACM SIGKDD Int Conf on Knowledge Discovery & Data Mining (SIGKDD'02) [C]. 2002.

[2] 张志刚，陈静，李晓明. 一种 HTML 网页净化方法[J]. 情报学报，2004，23(4)：387-393.

[3] 欧健文，董守斌，蔡斌. 模板化网页主题信息的提取方法[J]. 清华大学学报，2005，45(S1)：1743-1747.

[4] 封化民，刘飚，刘艳敏，等. 含有位置坐标树的 Web 页面分析和内容提取框架[J]. 清华大学学报，2005，45(S1)：1767-1771.

[5] 荆涛，左万利. 基于可视布局信息的网页噪音去除算法[J]. 华南理工大学学报（自然科学版），2004，32：84-87.

[6] 孙承杰，关毅. 基于统计的网页正文信息抽取方法的研究[J]. 中文信息学报，2004，18(5)：17-22.

[7] 李晓明, 闫宏飞, 王继民. 搜索引擎——原理、技术与系统[M]. 北京：科学出版社,2005.165-167.

[8] 陆一鸣, 胡健, 马范援. 一种基于源网页质量的锚文本相似度计算方法——LAAT[J]. 情报学报. 2005 年 10 月, 第 24 卷第 5 期: 548-554

[9] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[10] N. Eiron and KS McCurley. Analysis of anchor text for web search [J]. In Proceedings of the 26th Annual International 673 Information Retrieval, ACM, 2003: 459-460.

[11] 刘雁书,方平.利用链接关系评价网络信息的可行性研究[J].情报学报,2002 年 8 月,第 21 卷第 4 期: 401-406

[12] Kleinberg. Authoritative sources in a hyperlinked environment [J]. Journal of the ACM, 46(5): 604–632,999

[13] 吕俊生. 网上信息资源的链接分析研究[J].情报科学,2005 年 1 月,第 23 卷第 1 期: 78-82

[14] Taher h. Haeliwala Topic-Sensitive PageRank [J]. In Proceedings of the 11th International World Wide Web Conference (WWW2003), 2003.

[15] Taher H.Haveliwala. Efficient computation of PageRank. Stanford University Technical Report, 1999-31, DataBase Group, Computer Science Department, Stanford University, February 1999.

[16] 曹军. Google 的 PageRank 技术剖析[J]. 情报检索,2002 年第 10 期: 15-18

[17] Google 黑板报

[18] http://www.zhanglihai.com/article/2005-01-11/Z81CGDC2RG862140TG1LH340SAEBH87H.xtp

[19] Lucene in Action

[20]