
搜索引擎开发实战：基于 Lucene 和 Solr

搜索引擎核心技术与实现

——Lucene+Solr

罗刚

2011

搜索引擎核心技术与实现	1
第 1 章 搜索引擎总体结构.....	2
1.1 搜索引擎基本模块.....	2
1.2 开发环境.....	3
1.3 搜索引擎工作原理.....	4
1.3.1 网络爬虫.....	5
1.3.2 全文索引结构与 Lucene 实现	5
1.3.3 搜索用户界面.....	10
1.3.4 计算框架.....	10
1.3.5 文本挖掘.....	12
1.4 本章小结.....	12
第 2 章 网络爬虫的原理与应用	14
2.1 爬虫的基本原理.....	14
2.1.1 广度优先遍历.....	16
2.1.2 最好优先遍历.....	18
2.1.3 遍历特定网站.....	19
2.2 爬虫架构.....	19
2.2.1 基本架构.....	20
2.2.2 分布式爬虫架构.....	22
2.2.3 垂直爬虫架构.....	23
2.3 下载网络资源.....	24
2.3.1 下载网页的基本方法.....	24

2.3.2 HTTP 协议.....	27
2.3.3 使用 HttpClient 下载网页.....	33
2.3.4 重定向.....	39
2.3.5 解决套结字连接限制.....	40
2.3.6 下载图片.....	43
2.3.7 抓取 FTP.....	43
2.3.8 RSS 抓取.....	44
2.3.9 网页更新.....	46
2.3.10 抓取限制应对方法.....	48
2.3.11 URL 地址提取.....	51
2.3.12 抓取需要登录的网页.....	52
2.3.13 抓取 JavaScript 动态页面.....	58
2.3.14 抓取即时信息.....	61
2.3.15 抓取暗网.....	61
2.3.16 信息过滤.....	62
2.4 URL 地址查新.....	70
2.4.1 BerkeleyDB.....	70
2.4.2 布隆过滤器.....	72
2.5 增量抓取.....	75
2.6 并行抓取.....	75
2.6.1 多线程爬虫.....	76
2.6.2 垂直搜索的多线程爬虫.....	78

2.6.3 异步 IO	82
2.7 Web 结构挖掘	86
2.7.1 存储 Web 图	86
2.7.2 PageRank 算法	90
2.7.3 HITs 算法.....	97
2.7.4 主题相关的 PageRank.....	102
2.8 部署爬虫.....	104
2.9 本章小结.....	105
第 3 章 索引内容提取.....	108
3.1 从 HTML 文件中提取文本	108
3.1.1 字符集编码.....	108
3.1.2 识别网页的编码.....	111
3.1.3 网页编码转换为字符串编码.....	114
3.1.4 使用正则表达式提取数据	114
3.1.5 使用 HTMLParser 实现定向抓取.....	117
3.1.6 结构化信息提取.....	123
3.1.7 网页的 DOM 结构	126
3.1.8 使用 NekoHTML 提取信息.....	127
3.1.9 使用 XPath 提取信息.....	136
3.1.10 网页去噪.....	137
3.1.11 网页结构相似度计算	141
3.1.12 提取标题.....	143

3.1.13 提取日期.....	145
3.2 从非 HTML 文件中提取文本	145
3.2.1 提取标题的一般方法.....	146
3.2.2 PDF 文件	151
3.2.3 Word 文件	154
3.2.4 Rtf 文件.....	156
3.2.5 Excel 文件.....	168
3.2.6 PowerPoint 文件	171
3.3 图像的 OCR 识别	172
3.3.1 图像二值化.....	173
3.3.2 切分图像.....	175
3.3.3 SVM 分类.....	179
3.4 提取垂直行业信息.....	183
3.4.1 医疗行业.....	183
3.4.2 旅游行业.....	184
3.5 流媒体内容提取.....	184
3.5.1 音频流内容提取.....	185
3.5.2 视频流内容提取.....	188
3.6 存储提取内容.....	189
3.6.1 存入数据库.....	189
3.6.2 写入维基.....	190
3.7 本章小结.....	191

第 4 章 中文分词原理与实现.....	193
4.1 Lucene 中的中文分词.....	193
4.1.1 Lucene 切分原理.....	194
4.1.2 Lucene 中的 Analyzer.....	195
4.1.3 自己写 Analyzer.....	198
4.1.4 Lietu 中文分词.....	201
4.2 查找词典算法.....	201
4.2.1 标准 Trie 树.....	202
4.2.2 三叉 Trie 树.....	206
4.3 中文分词的原理.....	210
4.4 中文分词流程与结构.....	214
4.5 全切分词图.....	215
4.5.1 保存切分词图.....	215
4.5.2 形成切分词图.....	219
4.6 概率语言模型的分词方法.....	222
4.7 N 元分词方法.....	227
4.8 语料库.....	229
4.9 新词发现.....	230
4.10 未登录词识别.....	231
4.11 词性标注.....	232
4.11.1 隐马尔可夫模型.....	236
4.11.2 基于转换的错误学习方法.....	246

4.12 平滑算法.....	248
4.13 机器学习的方法.....	252
4.13.1 最大熵.....	253
4.13.2 条件随机场.....	256
4.14 有限状态机.....	256
4.15 本章小结.....	264
第 5 章 让搜索引擎理解自然语言	265
5.1 停用词表.....	265
5.2 句法分析树.....	267
5.3 相似度计算.....	274
5.4 文档排重.....	278
5.4.1 语义指纹.....	279
5.4.2 SimHash.....	282
5.4.3 分布式文档排重.....	293
5.5 中文关键词提取.....	294
5.5.1 关键词提取的基本方法.....	294
5.5.2 HITS 算法应用于关键词提取	297
5.5.3 从网页中提取关键词.....	299
5.6 相关搜索词.....	299
5.6.1 挖掘相关搜索词.....	300
5.6.2 使用多线程计算相关搜索词.....	302
5.7 信息提取.....	303

5.8 拼写检查与建议.....	308
5.8.1 模糊匹配问题.....	311
5.8.2 英文拼写检查.....	314
5.8.3 中文拼写检查.....	316
5.9 自动摘要.....	319
5.9.1 自动摘要技术.....	319
5.9.2 自动摘要的设计.....	320
5.9.3 基于篇章结构的自动摘要.....	326
5.9.4 Lucene 中的动态摘要.....	326
5.10 文本分类.....	330
5.10.1 特征提取.....	332
5.10.2 关键词加权法.....	335
5.10.3 朴素贝叶斯.....	338
5.10.4 支持向量机.....	348
5.10.5 多级分类.....	357
5.10.6 规则方法.....	359
5.10.7 网页分类.....	362
5.11 文本聚类.....	362
5.11.1 K 均值聚类方法.....	363
5.11.2 K 均值实现.....	365
5.11.3 深入理解 DBScan 算法.....	370
5.11.4 使用 DBScan 算法聚类实例.....	372

5.12 拼音转换.....	374
5.13 概念搜索.....	375
5.14 多语言搜索.....	383
5.15 跨语言搜索.....	384
5.16 情感识别.....	385
5.16.1 确定词语的褒贬倾向.....	388
5.16.2 实现情感识别.....	390
5.16.3 用户协同过滤.....	391
5.17 本章小结.....	393
第 6 章 Lucene 原理与应用.....	394
6.1 Lucene 深入介绍.....	394
6.1.1 常用查询.....	395
6.1.2 查询语法与解析.....	396
6.1.3 查询原理.....	400
6.1.4 遍历索引库.....	401
6.1.5 索引数值列.....	404
6.1.6 检索结果排序.....	407
6.1.7 处理价格.....	408
6.2 Lucene 中的压缩算法.....	408
6.2.1 变长压缩.....	409
6.2.2 PForDelta.....	411
6.2.3 VSEncoding.....	413

6.2.4 前缀压缩.....	414
6.2.5 差分编码.....	416
6.2.6 设计索引库结构.....	418
6.3 创建和维护索引库.....	419
6.3.1 创建索引库.....	419
6.3.2 向索引库中添加索引文档.....	420
6.3.3 删除索引库中的索引文档.....	422
6.3.4 更新索引库中的索引文档.....	422
6.3.5 索引的合并.....	424
6.3.6 索引文件格式.....	424
6.3.7 多线程写索引.....	427
6.3.8 分发索引.....	430
6.3.9 修复索引.....	433
6.4 查找索引库.....	433
6.4.1 基本查询.....	433
6.4.2 排序.....	434
6.4.3 使用 Filter 筛选搜索结果	435
6.5 读写并发.....	435
6.6 优化使用 Lucene.....	436
6.6.1 索引优化.....	436
6.6.2 查询优化.....	437
6.6.3 实现时间加权排序.....	440

6.6.4 实现字词混合索引.....	444
6.6.5 重用 Tokenizer.....	448
6.6.6 定制 Tokenizer.....	449
6.7 检索模型.....	451
6.7.1 向量空间模型.....	451
6.7.2 BM25 概率模型.....	456
6.7.3 统计语言模型.....	461
6.8 查询大容量索引.....	463
6.9 实时搜索.....	464
6.10 本章小结.....	466
第 7 章 搜索引擎用户界面.....	467
7.1 实现 Lucene 搜索.....	467
7.1.1 测试搜索功能.....	467
7.1.2 加载索引.....	468
7.2 搜索页面设计.....	470
7.2.1 Struts2 实现的搜索界面.....	471
7.2.2 实现翻页.....	473
7.3 实现搜索接口.....	475
7.3.1 编码识别.....	475
7.3.2 布尔搜索.....	479
7.3.3 指定范围搜索.....	479
7.3.4 搜索结果排序.....	481

7.3.5 索引缓存与更新.....	482
7.4 历史搜索词记录.....	489
7.5 实现关键词高亮显示.....	489
7.6 实现分类统计视图.....	492
7.7 实现相似文档搜索.....	498
7.8 实现 AJAX 搜索联想词	499
7.8.1 估计查询词的文档频率.....	500
7.8.2 搜索联想词总体结构.....	500
7.8.3 服务器端处理.....	501
7.8.4 浏览器端处理.....	507
7.8.5 拼音提示.....	509
7.8.6 部署总结.....	510
7.9 集成其他功能.....	510
7.9.1 拼写检查.....	510
7.9.2 分类统计.....	515
7.9.3 相关搜索.....	522
7.9.4 再次查找.....	525
7.9.5 搜索日志.....	525
7.10 搜索日志分析.....	527
7.10.1 日志信息过滤.....	527
7.10.2 信息统计.....	528
7.10.3 挖掘日志信息.....	531

7.11 部署网站	532
7.12 本章小结	533
第 8 章 使用 Solr 实现企业搜索	535
8.1 Solr 简介	535
8.2 Solr 基本用法	536
8.2.1 Solr 服务器端的配置与中文支持	537
8.2.2 把数据放进 Solr	542
8.2.3 删除数据	545
8.2.4 Solr 客户端与搜索界面	545
8.2.5 Solr 索引库的查找	548
8.2.6 索引分发	552
8.2.7 Solr 搜索优化	555
8.3 从 FAST Search 移植到 Solr	558
8.4 Solr 扩展与定制	560
8.4.1 Solr 中字词混合索引	560
8.4.2 相关检索	562
8.4.3 搜索结果去重	564
8.4.4 定制输入输出	567
8.4.5 分布式搜索	572
8.4.6 分布式索引	574
8.4.7 SolrJ 查询分析器	576
8.4.8 扩展 SolrJ	585

8.4.9 扩展 Solr.....	586
8.4.10 日文搜索.....	590
8.4.11 查询 Web 图.....	591
8.5 Solr 的.NET 客户端	594
8.6 Solr 的 PHP 客户端.....	598
8.7 本章小结.....	601
第 9 章 使用 Hadoop 实现分布式计算	602
第 10 章 地理信息系统案例分析.....	605
10.1 新闻提取.....	606
10.2 POI 信息提取	611
10.2.1 提取主体.....	617
10.2.2 提取地区.....	618
10.2.3 指代消解.....	620
10.3 本章小结.....	622
第 11 章 户外活动搜索案例分析	623
11.1 爬虫.....	623
11.2 信息提取.....	624
11.3 分类.....	627
11.3.1 活动分类.....	627
11.3.2 资讯分类.....	628
11.4 搜索.....	628
11.5 本章小结.....	629

第 12 章 英文价格搜索.....	629
专业英语词汇列表.....	630
参考资源.....	633
书籍.....	633
网址.....	633

第1章 搜索引擎总体结构

本章首先概要的介绍搜索引擎的总体结构和基本模块，然后会介绍其中的最核心的模块：全文检索的基本原理。为了尽快普及搜索引擎开发技术，本章介绍的搜索引擎结构可以采用开源软件实现。为了通过实践来深入了解相关技术，本章中会介绍相关的开发环境。本书介绍的搜索技术使用 Java 编程语言实现，之所以没有采用性能可能会更好的 C/C++，是希望读者不仅能够快速完成相关的开发任务，而且可以把相关实践作为一个容易上手的游戏。另外，为了集中关注程序的基本逻辑，书中的 Java 代码去掉了一些错误和异常处理，实际可以运行的代码可以在本书附带的光盘中找到。在以后的各章中会深入探索搜索引擎的每个组成模块。

1.1 搜索引擎基本模块

一个最简单的搜索引擎由索引和搜索界面两部分组成，相对完整的搜索结构如图 1-1 所示。

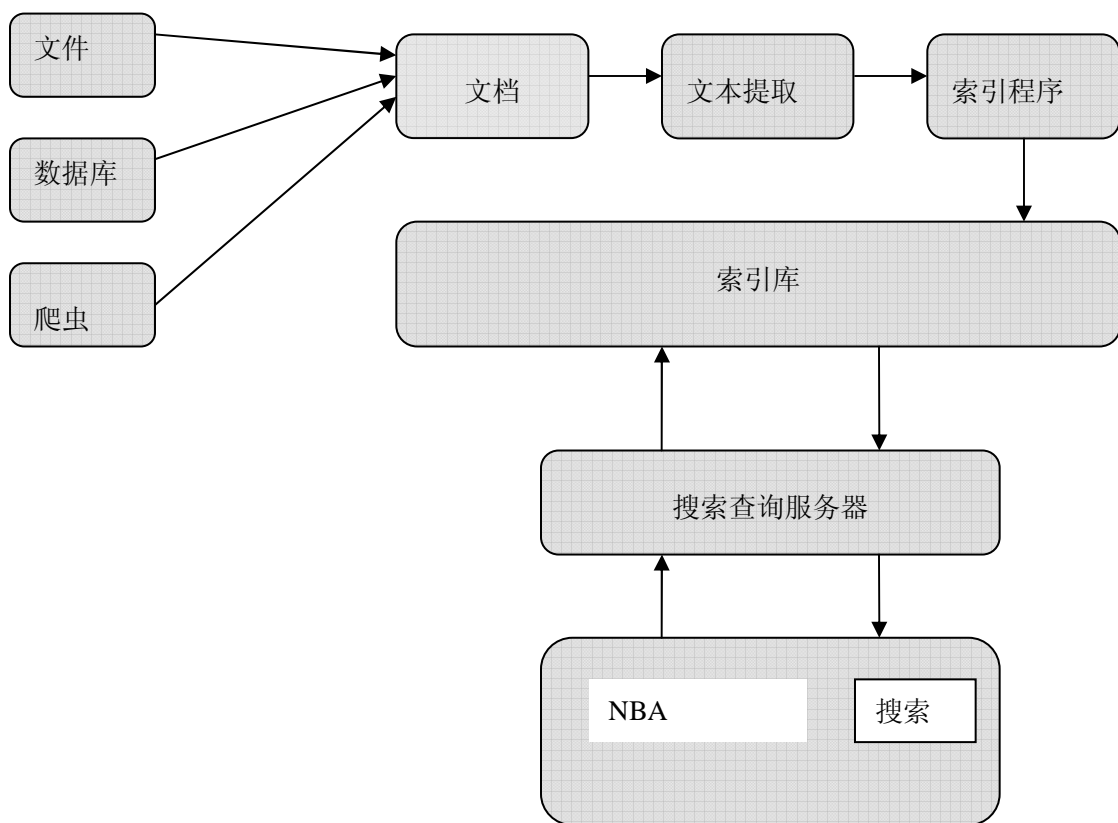


图 1-1 搜索引擎的简单结构

实现按关键字快速搜索的方法是建立全文索引库，所以最基础的程序是管理全文索引库

的程序。搜索的数据来源可以是互联网或者数据库，也可以是本地路径等。搜索引擎的基本模块从底向上的结构如图 1-2 所示：



图 1-2 搜索引擎中的主要模块

1.2 开发环境

由于开源软件的迅速发展，可以借助开源软件简化搜索引擎开发工作。很多开源软件用 Java 语言开发，例如最流行的全文索引库 Lucene，所以本书采用 Java 来自己实现搜索。为了实现一个简单的指定目录文件的搜索引擎，首先要准备好 JDK 和集成开发环境 Eclipse。当前可以使用 JDK1.6。JDK1.6 可以从 Java 官方网站 <http://java.sun.com> 下载得到。使用缺省方式安装即可。本书中的程序在附赠的光盘中都能找到，可以直接导入到 Eclipse 中。Eclipse 缺省是英文界面，如果习惯用中文界面可以从 <http://www.eclipse.org/babel/downloads.php> 下载支持中文的语言包。

Lucene 是一个 Java 实现的 jar 包用来管理搜索引擎索引库。可以从 <http://lucene.apache.org/java/docs/index.html> 下载到最新版本的 Lucene，当前的版本是 3.0。

如果需要用 Web 搜索界面，还要下载 Tomcat，当前可以从 <http://tomcat.apache.org/> 下载到，推荐使用 Tomcat6 以上的版本。使用开源的全文检索包 Lucene 做索引后，要把实现搜索的界面发布到 Tomcat。

对于 Web 搜索界面建议使用 MyEclipse 开发。对于其他的普通的非 Web 开发工作则不建议使用 MyEclipse。例如开发爬虫，建议只使用 Eclipse，而不要用 MyEclipse。MyEclipse 开发普通的 Java 项目时速度慢。

Lucene 及一些相关项目的源代码由版本管理工具 SVN 管理，如果要构建源代码工程，可以使用工具 Ant 和 Maven。

如果需要导出 Lucene 的最新开发版本，就需要用到 SVN 的客户端。小乌龟 TortoiseSVN 是最流行的 SVN 客户端。TortoiseSVN 的下载地址是 <http://tortoisesvn.tigris.org/>。安装 TortoiseSVN 后，选择一个存放源代码的文件夹，单击右键，选择 TortoiseSVN 菜单中的 Export...选项导出源代码。

Ant 与 Maven 都和项目管理软件 make 类似。虽然 Maven 正在逐步替代 Ant，但当前仍然有很多开源项目在继续使用 Ant。从<http://ant.apache.org/bindownload.cgi> 可以下载到 Ant 的最新版本。

在 windows 下 ant.bat 和三个环境变量相关 ANT_HOME、CLASSPATH 和 JAVA_HOME。需要用路径设置 ANT_HOME 和 JAVA_HOME 环境变量，并且路径不要以\或/结束，不要设置 CLASSPATH。使用 echo 命令检查 ANT_HOME 环境变量：

```
>echo %ANT_HOME%
```

```
D:\apache-ant-1.7.1
```

如果把 Ant 解压到 c:\apache-ant-1.7.1 则修改环境变量 PATH，增加当前路径 c:\apache-ant-1.7.1\bin。

如果一个项目的源代码根路径包括一个 build.xml 文件，则说明这个项目可能是用 Ant 构建的。大部分用 Ant 构建的项目只需要如下一个命令：

```
#ant
```

可以从<http://maven.apache.org/download.html>下载最新版本的 Maven，当前版本是 maven-2.2.1。解压下载的 Maven 压缩文件到 C: 根路径，将创建一个 c:\apache-maven-2.2.1 路径。修改 Windows 系统环境变量 PATH，增加当前路径 c:\apache-maven-2.2.1\bin。如果一个项目的源代码根路径包括一个 pom.xml 文件，则说明这个项目可能是用 Maven 构建的。大部分用 Maven 构建的项目只需要如下一个命令：

```
#mvn clean install
```

盖大楼的时候需要搭建最终不会交付使用的脚手架。很多单元测试代码也不会正式环境中运行，但是必须写的。可以使用 JUnit 做单元测试。

1.3 搜索引擎工作原理

一个基本的搜索包括采集数据的爬虫和索引库管理以及搜索页面展现等部分。

1.3.1 网络爬虫

网络爬虫(Crawler)又被称作网络机器人(Robot)，或者蜘蛛(Spider)，它的主要目的是为获取在互联网上的信息。只有掌握了“吸星大法”，才能源源不断的获取信息。网络爬虫利用网页中的超链接遍历互联网，通过 URL 引用从一个 HTML 文档爬行到另一个 HTML 文档。<http://dmoz.org>可以作为整个互联网抓取的入口。网络爬虫收集到的信息可有多种用途，如建立索引、HTML 文件的验证、URL 链接验证、获取更新信息、站点镜像等。为了检查网页内容是否更新过，网络爬虫建立的页面数据库往往包含有根据页面内容生成的文摘。

在抓取网页时大部分网络爬虫会遵循 Robot.txt 协议。网站本身可以有两种方式声明不想被搜索引擎收入的内容：第一种方式是在站点的根目录增加一个纯文本文件 `http://www.yourdomain.com/robots.txt`；另外一种方式是直接在 HTML 页面中使用 robots 的 meta 标签。

1.3.2 全文索引结构与 Lucene 实现

查找文档最原始的方式是通过文档编号找。就像一个人生下来就有一个身份证号，一个文档从创建开始就有一个文档编号。

早在计算机出现之前，为了方便查询，已经出现了人工为图书建立的索引，比如图 1-4 中的名词索引：

——按名词的拼音顺序检索

名词索引

半地下室 semi-basement

房间地面低于室外地平面的高度超过该房间净高的1/3，且不超过1/2者。
(摘自《住宅设计规范》(GB 50096-1999) 3页 中国建筑工业出版社 1999.5第一版)

壁柜 cabinet

住宅套内与墙壁结合而成的落地贮藏空间。
(摘自《住宅设计规范》(GB 50096-1999) 3页 中国建筑工业出版社 1999.5第一版)

比例 proportion

建筑构成各部分和各部分之间的相互关系，以及各部分与整体之间的比较关系。建筑比例是建筑构成中的一种量度尺度，有了具体的尺度才具有比例的真正意义。
(摘自《中国土木建筑百科全书》(建筑卷) 24页 中国建筑工业出版社 1995.5第一版)

变形缝 Deformation joint

为防止建筑物在外界因素作用下，结构内部产生附加变形和应力，导致开裂甚至破坏而预留的构造缝。变形缝包括
(摘自《中国土木建筑百科全书》(建筑卷) 27页 中国建筑工业出版社 1995.5第一版)

标准层 typical floor

平面布置相同的住宅楼层。
(摘自《住宅设计规范》(GB 50096-1999) 2页 中国建筑工业出版社 1999.5第一版)

不对称均衡 asymmetrical balance

图 1-4 人工建立的名词索引

为了按词快速定位抓取过来的文档，需要以词为基础建立全文索引，也叫倒排索引(Inverted index)，如图 1-5 所示。在这里，索引中的文档用编号表示。

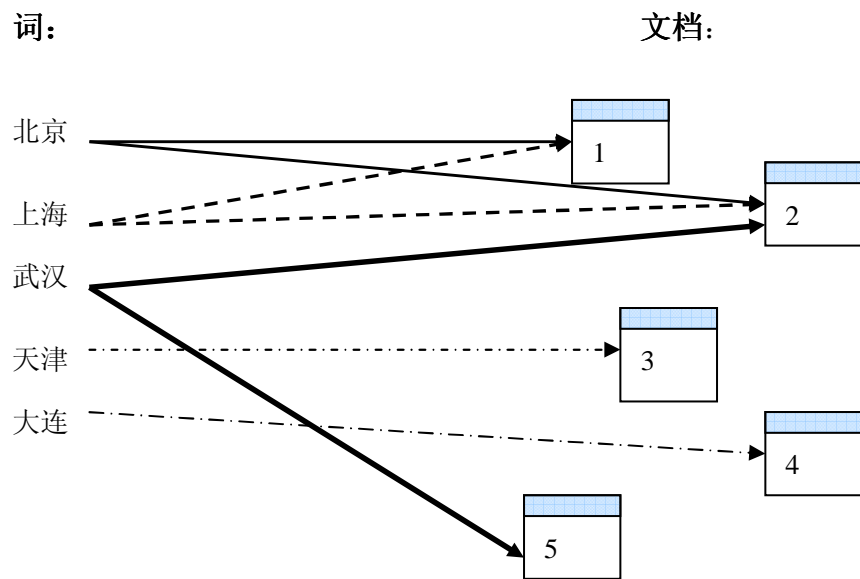


图 1-5 以词为基础的全文索引

倒排索引是相对于正向索引来说的，首先用正向索引来存储每个文档对应的单词列表，然后再建立倒排索引，根据单词来索引文档编号。

例如要索引如下两个文档：

Doc Id 1: 自己动手写搜索引擎

Doc Id 2: 自己动手写网络爬虫

首先把这些文档中的内容分成一个个的词：

Doc Id 1: 自己/动手/写/搜索引擎

Doc Id 2: 自己/动手/写/网络爬虫

按单词建立的倒排索引结构如表 1-1 所示：

词	(文档, 频率)	在文档中出现的位置
动手	(1, 1), (2, 1)	(2), (2)
搜索引擎	(1, 1)	(4)
网络爬虫	(2, 1)	(4)

词	(文档, 频率)	在文档中出现的位置
写	(1, 1), (2, 1)	(3), (3)
自己	(1, 1), (2, 1)	(1), (1)

表 1-1 倒排索引结构

每个单词(term)后面的文档编号(docId)列表叫做 posting list。在 Lucene 中，倒排索引结构存储在二进制格式的多个索引文件中，其中以 tis 为后缀的文件中包含了单词信息，frq 后缀的文件记录单词的文档编号和这个单词在文档中出现了多少次，也就是频率信息，prx 后缀的文件包含了单词出现的位置信息。

为了快速的查找单词，可以先对单词列表排序，例如：《新华字典》和《现代汉语词典》按拼音排序。从排好序的词表中查找一个词可以采用折半查找的方法快速查询。下面是实现折半查找的代码。

```
int low = fromIndex; //开始位置
int high = toIndex - 1; //结束位置

while (low <= high) {
    int mid = (low + high) >> 1; //相当于 mid = (low + high)/2
    Comparable midVal = (Comparable)a[mid]; //取中间的值
    int cmp = midVal.compareTo(key); //中间值和要找的关键字比较

    if (cmp < 0)
        low = mid + 1;
    else if (cmp > 0)
        high = mid - 1;
    else
        return mid; // 查找成功，返回找到的位置
}
return -(low + 1); // 没找到，返回负值
```

在 Lucene 中，org.apache.lucene.index.TermInfosReader 类的 getIndexOffset 方法实现了一个类似的折半查找。对于特别大的顺序集合可以用插值法查找提高查找速度。

Lucene(<http://lucene.apache.org/>)是一个开放源代码的全文索引库。经过 10 多年的发展，Lucene 拥有了大量的用户和活跃的开发团队。Eclipse 软件和 Twitter 网站等都在使用 Lucene。如果说 Google 是拥有最多用户访问的搜索引擎网站，那么拥有最多开发人员支持的搜索软件项目也许是 Lucene。

Lucene 的整体结构如图 1-6 所示。

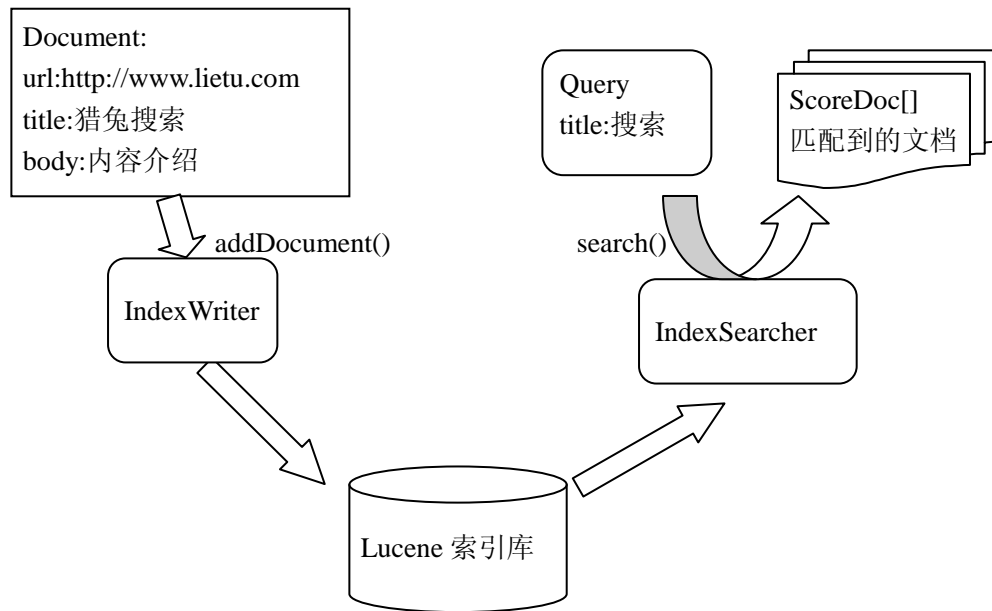


图 1-6 Lucene 原理图

Lucene 中的基本概念介绍如下：

- 第一个概念是 **Index**，也就是索引库。文档的集合组成索引。和一般的数据库不一样，Lucene 不支持定义主键。在 Lucene 中并不存在一个叫做 **Index** 的类。通过 **IndexWriter** 来写索引，通过 **IndexReader** 读索引。索引库在物理形式上一般是位于一个路径下的一系列文件。
- 一段有意义的文字需要通过 **Analyzer** 分割成一个个词语后才能按关键词搜索。**Analyzer** 就是分析器，**StandardAnalyzer** 是 Lucene 中最常用的分析器。为了达到更好的搜索效果，不同的语言可以使用不同的分析器，例如 **CnAnalyzer** 就是一个主要处理中文的分析器。
- **Analyzer** 返回的结果就是一串 **Token**。**Token** 包含一个代表词本身含义的字符串和该词在文章中相应的起止偏移位置，**Token** 还包含一个用来存储词类型的字符串。
- 一个 **Document** 代表索引库中的一条记录，也叫做文档。要搜索的信息封装成 **Document** 后通过 **IndexWriter** 写入索引库。调用 **Searcher** 接口按关键词搜索后，返回的也是一个封装后的 **Document** 的列表。
- 一个 **Document** 可以包含多个列，叫做 **Field**。例如一篇文章可以包含“标题”、“正文”、“修改时间”等 **field**。创建这些列对象以后，可以通过 **Document** 的 **add** 方法增加这些列。和一般的数据库不一样，一个文档的一个列可以有多个值。例如一篇文档既可以属于互联网类，又可以属于科技类。

- **Term** 是搜索语法的最小单位，复杂的搜索语法会分解成一个一个的 **Term** 查询。它表示文档的一个词语，**Term** 由两部分组成：它表示的词语和这个词语所出现的 **Field**。

Lucene 中的 API 相对数据库来说比较灵活，没有类似数据库先定义表结构后使用的过程。如果前后两次写索引时定义的列名称不一样，Lucene 会自动创建新的列，所以 **Field** 的一致性需要我们自己掌握。

1.3.3 搜索用户界面

随着搜索引擎技术逐渐走向成熟，搜索用户界面也形成了一些比较固定的模式。

- **输入提示词**：在用户在搜索框中输入查询词的过程中随时给予查询提示词。对中文来说，当用户输入拼音时，也能提示。
- **相关搜索提示词**：当用户对当前搜索结果不满意时，也许换一个搜索词就能够得到更有用的信息。一般会根据用户当前搜索词给出多个相关的提示词。可以看成是协同过滤在搜索词上的一种具体应用。
- **相关文档**：返回和搜索结果中的某一个文档相似的文档。例如：Google 搜索结果中的“类似结果”。
- **在结果中查询**：如果返回结果很多，则用户在返回结果中再次输入查询词以缩小查询范围。
- **分类统计**：返回搜索结果在类别中的分布图。用户可以按类别缩小搜索范围，或者在搜索结果中导航。有点类似数据仓库中的向下钻取和向上钻取。
- **搜索热词统计界面**：往往按用户类别统计搜索词，例如按用户所属区域或者按用户所属部门等，当然也可以直接按用户统计搜索热词。例如 Google 的 Trends。

搜索界面的改进都是以用户体验为导向。所以搜索用户界面往往还根据具体应用场景优化。所有这一切都是为了和用户的交互达到最大的效果。

1.3.4 计算框架

互联网搜索经常面临海量数据。需要分布式的计算框架来执行对网页重要度打分等计算。有的计算数据很少，但是计算量很大，还有些计算数据量比较大，但是计算量相对比较小。例如计算圆周率是计算密集型，互联网搜索中的计算往往是数据密集型。所以出现了数据密集型的云计算框架。**MapReduce** 是一种常用的云计算框架。

MapReduce 把计算任务分成两个阶段。映射(**Map**)阶段按数据分类完成基本计算，化简(**Reduce**)阶段收集基本的计算结果。使用 **MapReduce** 统计词频的例子如图 1-8 所示。

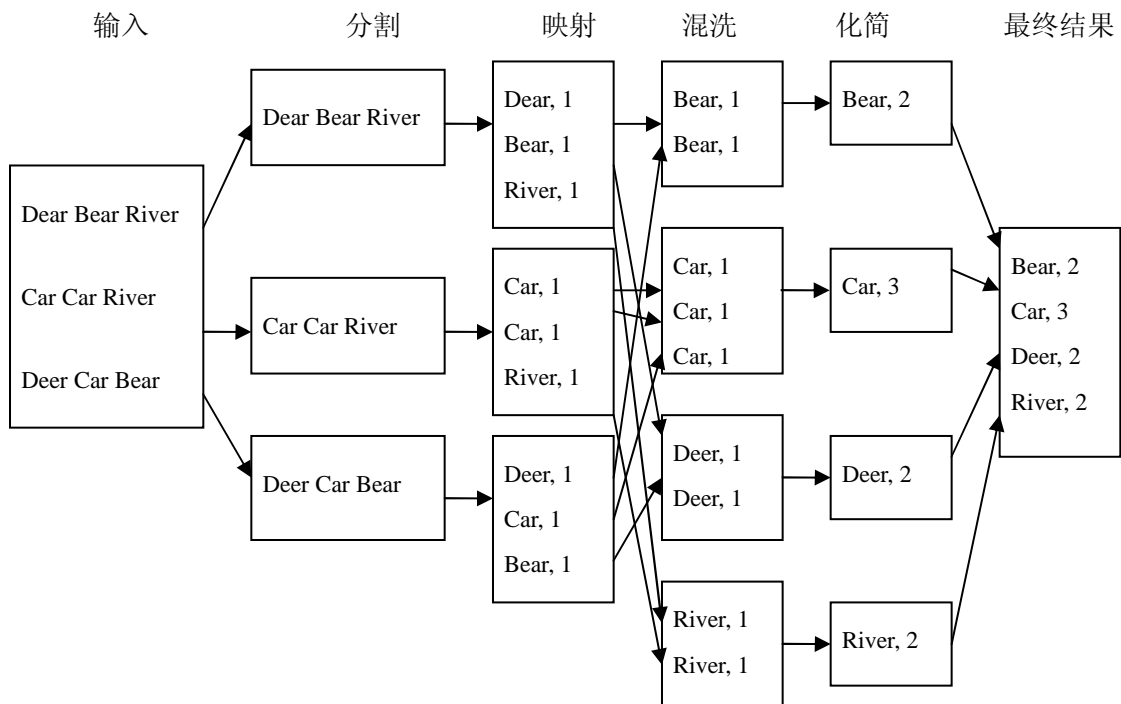


图 1-8 词频统计的例子

Hadoop(<http://hadoop.apache.org/>)是 MapReduce 思想实现的一个开源计算平台，已经在包括百度等搜索引擎开发公司得到商用。使用 Hadoop 实现词频统计的 `TokenCounterMapper` 类如下：

```
public class TokenCounterMapper extends Mapper{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context) throws IOException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.collect(word, one);
        }
    }
}
```

但是 MapReduce 是批处理的操作方式。一般来说，直到完成上一阶段的操作后才能启动下一阶段的操作。

要有一种计算，可以尽快出结果，随着时间的延长，计算结果会越来越好。很多计算可

以用迭代的方式做，迭代次数越多，结果往往越好 比如 PageRank 或者 KMeans、EM 算法。当然，这个应该不只需要迭代，还需要向最优解收敛。

1.3.5 文本挖掘

搜索文本信息需要理解人类的自然语言。文本挖掘指从大量文本数据中抽取隐含的，未知的，可能有用的信息。

常用的文本挖掘方法包括：全文检索、中文分词、句法分析、文本分类、文本聚类、关键词提取、文本摘要、信息提取、智能问答等。文本挖掘相关技术的结构如图 1-9 所示：

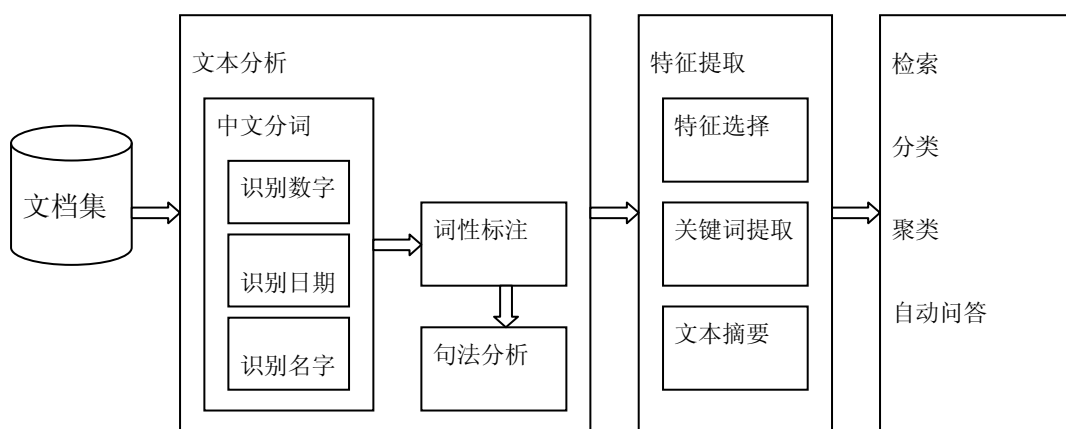


图 1-9 文本挖掘的结构

1.4 本章小结

牛顿用简单的原理来近似自然界中的复杂现象，搜索引擎开发也是一门复杂的技术，但是可以从简单的折半查找开始理解。本章介绍了互联网搜索 Google 及其创新原则。在 Google 出现之前，Yahoo 使用人工对网站分类，提供按目录导航和搜索目录数据库功能。在 Google 尚未占据互联网搜索绝对优势之前，也是在笔者第一次听人推荐 Google 之前，就出现了元搜索引擎(Meta Search Engine)。用户只需提交一次搜索请求，由元搜索引擎负责转换处理后提交给多个预先选定的独立搜索引擎，并将从各独立搜索引擎返回的所有查询结果，集中起来处理后再返回给用户。但 Google 开始独家垄断全球互联网搜索后，元搜索引擎逐渐被人遗忘。

Google 早期的时候使用 MapReduce 实现分布式索引。后来之所以放弃这种方式，是因为它并不能为 Google 提供它所想要的索引速度。工程师需要等待 8 个小时的计算时间才能够得到计算的全部结果，然后把它发布到索引系统中去。随着实时检索时代的到来，Google 需要在几秒内刷新索引内容，而非 8 小时。

Hadoop 来源于开源的分布式搜索项目 Nutch。Powerset 公司在 Hadoop 的基础上开发了

基于 BigTable 架构的数据库 Hbase(<http://hbase.apache.org/>)。2008 年, 微软收购了 Powerset。

与文本挖掘技术对应的是包括语音识别、基于内容的图像检索等技术的流媒体挖掘技术。随着网络电视和视频网站的流行, 流媒体挖掘技术正越来越引起人们的关注。

除了像 Google 的网页搜索这样的常规搜索引擎, 还有些特殊的搜索引擎。搜索的输入不一定是简单的关键词, 例如 Wolfram|Alpha(<http://www.wolframalpha.com/>)是一个特殊的可计算的知识引擎。它可以根据用户的问句式的输入精确地返回一个答案。TextRunner 搜索(<http://www.cs.washington.edu/research/textrunner/>)是另外一个问答式的搜索。搜索引擎不一定只是简单列出搜索结果, Vivisimo(<http://vivisimo.com/>)是实现聚类的搜索, 也可以分类统计搜索结果。搜索结果不一定是文档, 例如 Aardvark(<http://vark.com/>)是一个社会化搜索, 它可以自动选择合适的人来回答用户提出的问题。

第2章 网络爬虫的原理与应用

大的搜索引擎，例如 Google，对整个互联网做了一个镜像。很多有专门用途的信息也需要汇总，例如网上购物或者旅游。这些专门收集互联网中的信息的程序叫做网络爬虫。如果把互联网比喻成一个覆盖地球的蜘蛛网，那么抓取程序就是在网上爬来爬去的蜘蛛。

虽然存在一些通用的采集器，但是因为应用目的不同，很多爬虫程序都是定制开发的。网络爬虫需要实现的基本功能包括下载网页以及对 URL 地址的遍历。为了高效的快速遍历大量的 URL 地址，还需要应用专门的数据结构来优化。爬虫很消耗带宽资源，设计爬虫时需要仔细的考虑如何节省网络带宽。

2.1 爬虫的基本原理

如果把网页看成节点，网页之间的超链接看成边，则可以把整个互联网看成是一个巨大的非连通图。网络爬虫有两个基本的任务：发现包含有效信息的 URL 和下载网页。

为了获取网页，需要有一个初始的 URL 地址列表。然后通过网页中的超链接访问到其他的页面。有人可能会奇怪像 Google 或百度这样的搜索门户怎么设置这个初始的 URL 地址列表。一般来说，网站拥有者把网站提交给分类目录，例如 dmoz(<http://www.dmoz.org/>)，爬虫则可以从开放式分类目录 dmoz 抓取。

抓取下来的网页中包含了想要的信息，一般存放在数据库或索引库这样的专门的存储系统中，如图 2-1 所示。如果把网页原文存储下来，就可以实现“网页快照”功能。

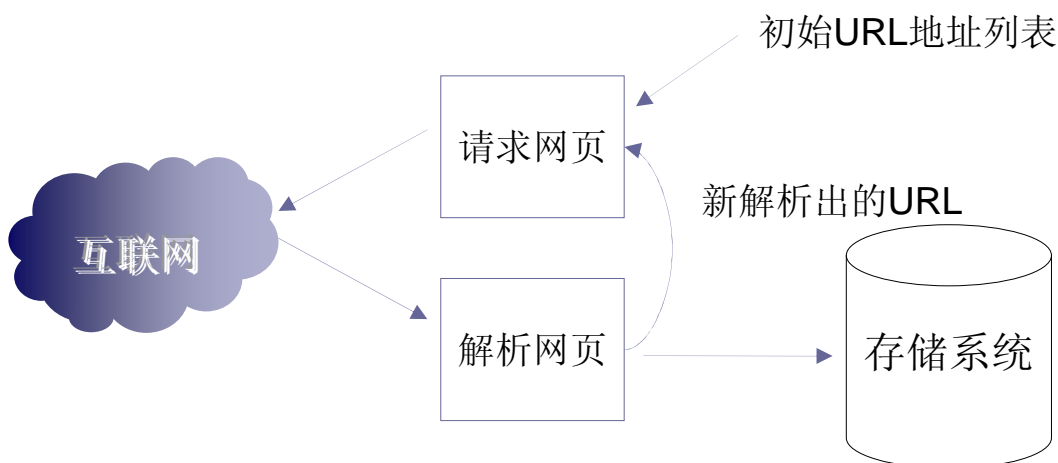


图 2-1 爬虫基本结构图

爬虫程序的工作是从一个种子链接的集合开始。把种子 URL 集合作为参数传递给网络

爬虫。爬虫先把这些初始的 URL 放入 URL 工作队列(Todo 队列, 又叫做 Frontier), 然后遍历所有工作队列中的 URL, 下载网页并把其中新发现的 URL 再次放入工作队列。为了判断一个 URL 是否已经遍历过, 把所有遍历过的 URL 放入历史表(Visited 表)。爬虫抓取的基本过程如图 2-1 所示:

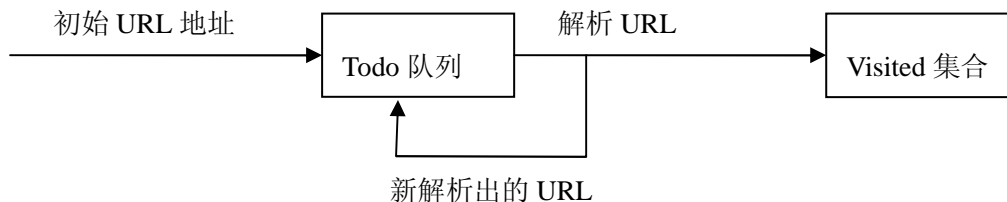


图2-2 网页遍历流程图

抓取的主要流程如下:

```
while (todo.size() > 0) { //如果 Todo 队列不是空的
    //从 Todo 队列里面提取 URL
    String strUrl = todo.iterator().next();
    //下载 URL 对应的网页内容
    String content = downloadPageContent(strUrl);
    //把网页内容存储到本地
    //提取网页内容中新发现的 URL 链接
    HashSet<String> newLinks = retrieveLinks(content, new URL(strUrl));
    //把新发现的链接加入 Todo 队列
    todo.addAll(newLinks);

    //从 Todo 队列里删除已经爬过的 URL
    todo.remove(strUrl);
    //把从 Todo 队列里删除的 URL 添加到 Visited 集合中
    visited.add(strUrl);
}
```

整个互联网是一个大的图, 其中, 每个 URL 相当于图的一个节点, 因此, 网页遍历就可以采用图遍历的算法进行。通常, 网络爬虫的遍历策略有三种, 广度优先遍历, 深度优先遍历和最佳优先遍历。其中, 深度优先遍历由于极有可能使爬虫陷入黑洞, 因此, 广度优先遍历和最佳优先遍历就成为了常用的爬虫策略。垂直搜索往往采用定制的遍历方法抓取特定的网站。

2.1.1 广度优先遍历

广度优先是指网络爬虫会先抓取起始网页中链接的所有网页,然后再选择其中的一个链接网页,继续抓取在此网页中链接的所有网页。这是最常用的方式,这个方法也可以让网络爬虫并行处理,提高其抓取速度。以图 2-2 中的图为例说明广度遍历的过程。

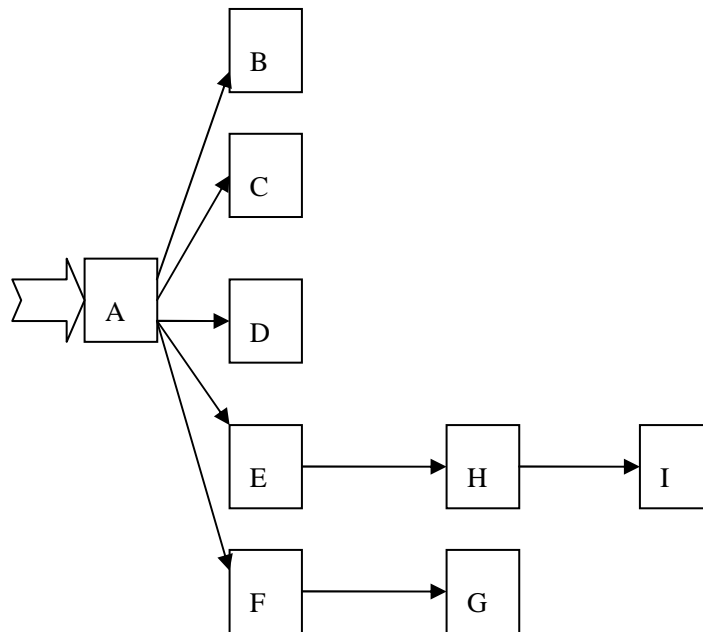


图2-2 网络爬虫遍历的图

例如在图 2-2 中, A 为种子节点, 则首先遍历 A(第一层), 接着是 BCDEF(第二层), 接着遍历 GH(第三层), 最后遍历 I(第四层)。蜘蛛从蛛网中心往外, 一圈一圈的织网, 可以类比作广度遍历。

广度优先遍历使用一个队列来实现 Todo 表, 先访问的网页先扩展。针对上图, 广度优先遍历的执行过程如下:

Todo 队列	Visited 集合
a	null
b c d e f	a
c d e f	a b
d e f	a b c
e f	a b c d
f h	a b c d e
h g	a b c d e f

Todo 队列	Visited 集合
g i	a b c d e f h
i	a b c d e f h g
null	a b c d e f h g i

表2-1 广度优先遍历过程表

庄子曾说：“吾生也有涯，而知也无涯，以有涯随无涯，殆已”。在学习和工作的时候，需要分辨事情的轻重缓急，否则一味蛮干，最终结果只能是--“殆已”。对于浩瀚无边的互联网而言，网络爬虫涉及到页面确实只是冰山一角。因此，需要以最小的代价(硬件、带宽)获取到最大的利益(数量最多的重要的网页)。

为了先抓取重要的网页，可以采用最佳优先。最佳优先爬虫策略也称为“页面选择问题”(PageSelection)，通常，这样保证在有限带宽条件下，尽可能的照顾到重要性高的网页。

如何实现最佳优先爬虫呢，最简单的方式可以使用优先级队列(PriorityQueue)来实现TODO表，这样，每次选出来扩展的URL就是具有最高重要性的网页。在队列中，先进入的元素先出，但是在优先队列中，优先级高的元素先出队列。

比如，假设上图的节点重要性 D>B>C>A>E>F>I>G>H，则整个遍历过程如表 2-2 所示：

Todo 优先级队	Visited 集合
A	null
B,C,D,E,F	A
B,C,E,F	A,D
C,E,F	A,B,D
E,F	A,B,C,D
F,H	A,B,C,D,E
H,G	A,B,C,D,E,F
H	A,B,C,D,E,F,G
I	A,B,C,D,E,F,H,G
null	A,B,C,D,E,F,H,I

表2-2 最佳优先遍历过程表

2.1.2 最好优先遍历

下面实现一个 Best-First 的爬虫，也就是说每次取的 URL 地址都是 Todo 列表中的最好的 URL 地址。为了实现快速的查找和操作，一般使用 Berkeley DB 来实现 TodoList。

Berkeley DB 中存储的一般是一个关键字和值的 Map。为了同时实现按照 URL 地址查找和按照 url 地址分值排序，简单的 map 不能满足要求，这时候可以使用 SecondaryIndex 和 PrimaryIndex 来达到多列索引的目的。实现持久化的基本类如下：

```
@Entity
public class NewsSource {
    @PrimaryKey public String URL;
    public String source;
    public int level;
    public int rank;
    public String urlDesc = null;
    @SecondaryKey(related=Relationship.MANY_TO_ONE) public int score;
}
```

Todo 列表的构造方法如下：

```
public TodoTaskList(Environment env) throws Exception {
    StoreConfig storeConfig = new StoreConfig();
    storeConfig.setAllowCreate(true);
    storeConfig.setTransactional(false);
    store = new EntityStore(env, "classDb", storeConfig);
    newsByURL = store.getPrimaryIndex(String.class, NewsSource.class);
    secondaryIndex =
        store.getSecondaryIndex(this.newsByURL, Integer.class, "score");
}
```

从 Todo 列表取得最大分值的 URL 地址的方法如下：

```
public NewsSource removeBest() throws DatabaseException {
    Integer score = secondaryIndex.sortedMap().lastKey();
    if (score != null){
        EntityIndex<String,NewsSource> urlLists = secondaryIndex.subIndex(score);
        EntityCursor<String> ec = urlLists.keys();
        String url = ec.first();
        ec.close();
        NewsSource source = urlLists.get(url);
    }
}
```

```
        urlLists.delete(url);  
        return source;  
    }  
    return null;  
}
```

2.1.3 遍历特定网站

从首页提取类别信息，然后按类别信息找到目录页。通过翻页遍历所有的目录页，提取详细页。从详细页面提取商品信息。把商品信息存入数据库。

以新浪新闻为例，同一个目录下的 URL 是：

<http://roll.news.sina.com.cn/news/gjxw/hqqw/index.shtml>

http://roll.news.sina.com.cn/news/gjxw/hqqw/index_2.shtml

http://roll.news.sina.com.cn/news/gjxw/hqqw/index_3.shtml

以购物网站为例：

<http://www.mcmelectronics.com/browse/Cameras/0000000002>

<http://www.mcmelectronics.com/browse/Cameras/0000000002/p/2>

<http://www.mcmelectronics.com/browse/Cameras/0000000002/p/3>

<http://www.mcmelectronics.com/browse/Cameras/0000000002/p/4>

不断增加翻页参数，一直到找不到新的商品为止。

```
boolean parserIndexPage(String indexUrl){  
    //解析目录页中的商品，看看是否有新的商品  
    return isNew;  
}
```

2.2 爬虫架构

本节首先介绍爬虫的基本架构，然后介绍可以在多台服务器上运行的分布式爬虫架构。

2.2.1 基本架构

一般的爬虫软件，通常都包含以下几个模块：

- 1、保存种子 URL 和待抓取的 URL 的数据结构。
- 2、保存已经抓取过的 URL 的数据结构，防止重复抓取。
- 3、页面获取模块。
- 4、对已经获取的页面内容的各个部分进行抽取的模块。例如抽取 HTML、JavaScript 等。

其他可选的模块包括：

- 5、负责连接前处理模块。
- 6、负责连接后处理模块。
- 7、过滤器模块。
- 8、负责多线程的模块。
- 9、负责分布式的模块。

各模块详细介绍如下：

- 1、保存种子和爬取出来的 URL 的数据结构

农民会把有生长潜力的籽用做种子，这里把一些活跃的网页用做种子 URL，例如网站的首页或者列表页，因为在这些页面经常会发现新的链接。通常，爬虫都是从一系列的种子 URL 开始爬取，一般从数据库表或者配置文件中读取这些种子 URL。种子 URL 描述表如下：

字段名	字段类型	说明
Id	NUMBER(12)	唯一标识
url	Varchar(128)	网站 url
source	Varchar(128)	网站来源描述
rank	NUMBER(12)	网站 PageRank 值

表 2-3 最佳优先遍历过程表

但是保存待抓取的 URL 的数据结构确因系统的规模、功能不同而可能采用不同的策略。一个比较小的示例爬虫程序，可能就使用内存中的一个队列，或者是优先级队列进行存储。一个中等规模的爬虫程序，可能使用 BekerlyDB 这种内存数据库来存储，如果内存中存放不下的话，还可以序列化到磁盘上。但是，真正的大规模爬虫系统，是通过服务器集群来存储已经爬取出来的 URL 的。并且，还会在存储 URL 的表中附带一些其他信息，比如说 PageRank 值等，供之后的计算用。

2、保存已经抓取过的 URL 的数据结构

已经抓取过的 URL 的规模和待抓取的 URL 的规模是一个相当的量级。正如我们前面介绍的 TODO 表和 Visited 表。但是，他们唯一的区别是，Visited 表会经常被查询，以便确定发现的 URL 是否已经处理过。因此，Visited 表数据结构如果是一个内存数据结构的话，可以采用 Hash(HashMap 或者 HashSet)来存储，如果保存在数据库中的话，可以对 URL 列建立索引。

3、页面获取模块

当从种子 URL 队列或者抓取出来的 URL 队列中获得 URL 后，便要根据这个 URL 来获得当前页面的内容，获得的方法非常简单，就是普通的 IO 操作。在这个模块中，仅仅是把 URL 所指的内容按照二进制的格式读出来，而不对内容做任何处理。

4、提取已经获取的网页的内容中的有效信息

从页面获取模块的结果是一个表示 HTML 源代码的字符串。从这个字符串中抽取各种相关的内容，是爬虫软件的目的。因此，这个模块就显得非常重要。

通常，在一个网页中，除了包含有文本内容还有图片，超链接等。对于文本内容，首先把 HTML 源代码的字符串保存成 HTML 文件即可。关于超链接提取，可以根据 HTML 语法，使用正则表达式来提取，并且把提取的超链接加入到 TODO 表中。也可以使用专门的 HTML 文档解析工具。

在网页中，超链接不光指向 HTML 页面，还会指向各种文件，对于除了 HTML 页面的超链接之外，其他内容的链接不能放入 TODO 表中，而要直接下载。因此，在这个模块中，还必须包含提取图片，JavaScript，PDF，DOC 等等内容的部分。并且，在提取过程中，还要针对 HTTP 协议处理返回的状态码。这章我们主要研究网页的架构问题，将在下一章详细研究从各种文件格式提取有效信息。

5、负责连接前处理模块，负责连接后处理模块，过滤器模块

如果只抓取某个网站的网页，则可以对 URL 按域名过滤。

6、多线程模块

爬虫主要消耗三种资源：网络带宽，中央处理器和磁盘。三者中任何一者都有可能成为

分布式是当今计算的主流。这项技术也可以同时用在网络爬虫上面。下节介绍多台机器并行采集的方法。

把抓取任务分布到不同的节点分布主要是为了可扩展性,也可以使用物理分布的爬虫系统,让每个爬虫节点抓取靠近它的网站。例如,北京的爬虫节点抓取北京的网站,上海的爬虫节点抓取上海的网站。还比如,电信网络中的爬虫节点抓取托管在电信的网站,联通网络中的爬虫节点抓取托管在联通的网站。

```

graph LR
    Web[Web] <--> DNS[DNS]
    Web --> Download[下载网页]
    DNS <--> Download
    Download --> Parse[解析页面]
    Parse <--> Check[内容是否重复?]
    Check <--> Doc[(文档)]
    Check --> Filter[URL 过滤]
    Filter --> Split[按域名分割]
    Split --> Dedup[URL 去重]
    Dedup <--> URL[(URL)]
    Dedup --> Send[发送到其他节点]
    Dedup --> Frontier[URL Frontier]
    Frontier --> Download

```

要点在于按域名分配采集任务。每台机器扫描到的网址，不属于它自己的会交换给属于它的机器。例如，专门有一台机器抓取 s 开头的网站：<http://www.sina.com.cn> 和 <http://www.sohu.com>，而另外一台机器抓取 q 开头的网站：<http://www.qq.com>。

垂直信息分布式抓取的基本设计：

1. 要处理的信息根据首字母散列到 26 个不同的爬虫服务器，让不同的机器抓取不同的信息。
2. 每台机器通过配置文件读取自己要处理的字母。每台机器抓取完一条结果后把该结果写入到统一的一个数据库中。比如说有 26 台机器，第一台机器抓取字母 a 开头的公司，第二台机器抓取字母 b 开头的公司，依次类推。
3. 如果某一台机器抓取速度太慢，则把该任务拆分到其它的机器。

可以把检测重复内容的功能专门交给一个集群处理。解析页面的功能和下载网页的功能可以直接在同一台机器处理。

2.2.3 垂直爬虫架构

垂直爬虫往往抓取指定网站的新闻或论坛等信息。可以指定初始抓取的首页或者列表页，然后提取相关的详细页中的有效信息存入数据库，总体结构如图 2-4 所示。

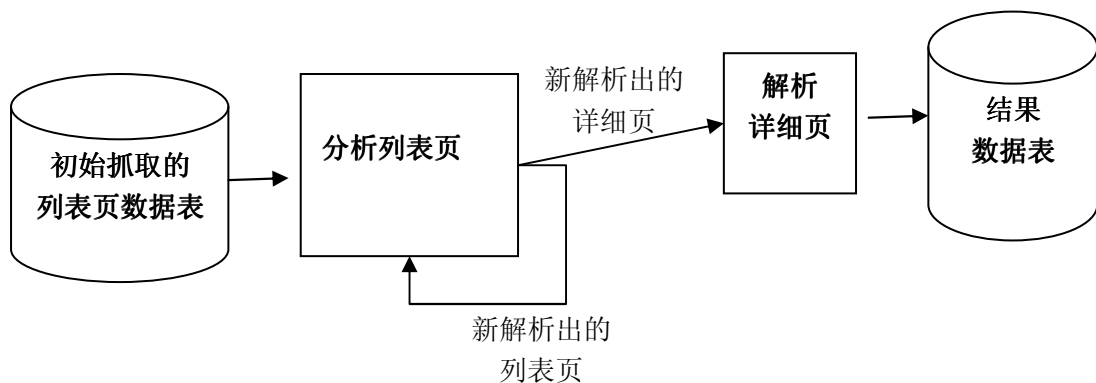


图 2-4 垂直爬虫架构图

垂直爬虫涉及的功能有：

- 从首页提取不同栏目的列表页。
- 网页分类：把网页分类成列表页或详细页或者未知类型。
- 列表页链接提取：从列表页提取同一个栏目下的列表页，这些页面往往用“下一页”、“尾页”等信息描述。往往有个参数来控制翻页，把这个参数叫做翻页参数，例如：http://...&_pgn=2中的_pgn。有时候，列表页会显示总共有多少页，可以根据翻页参数列出所有的列表页。如果没有，还可以根据翻页参数一直往后翻页，直到找不到新信息或

者出错为止。

- 详细页面内容提取：从详细页提取网页标题、主要内容、发布时间等信息。一般把每行数据封装到一个对象中。

可以每个网站用一个线程抓取，这样方便控制对被抓网站的访问频率。最好有通用的信息提取方式来解析网页，这样可以减少人工维护成本。同时，也可以采用专门的提取类来处理数据量大的网站，这样可以提高抓取效率。

解析详细页面成为结构化数据并存储结果的工作可以交给专门的机器去做。所以可以把下载的详细页面放入消息队列，然后再由解析详细页面的机器从消息队列中取出要分析的详细页面。

2.3 下载网络资源

做生意经常需要签协议，爬虫和网站打交道，把数据抓下来，也有一些相关的协议。首先介绍和网站打交道所使用的相关协议。然后用 Java 中现成的类实现最基本的下载网页功能。为了完成一些扩展功能，再介绍使用专门的开源项目下载网页。

2.3.1 下载网页的基本方法

Java 语言中，`java.net.URL` 类能够对实际的 URL 进行建模，通过这个类，可以对相应的 Web 服务器发出请求并且获得相应的文档。`java.net.URL` 类有一个默认的构造函数，使用 URL 地址作为参数，构造 URL 对象。

```
URL pageURL = new URL(path);
```

之后，可以通过获得的 URL 对象来取得输入流，进而像读入本地文件一样来下载网页。

```
InputStream stream = pageURL.openStream();
```

然后将网页看做网络文件，然后按照文件读取的方式把它读出来，保存到本地。以下是一个下载网页的小程序，简单的说明网页下载的原理。

```
public class RetrivePage {
    public static String downloadPage(String path){
        //根据传入的路径构造 URL
        URL pageURL = new URL(path);
        //创建网络流
        BufferedReader reader = new BufferedReader(new InputStreamReader(
            pageURL.openStream()));
        String line;
        //读取网页内容
```

```

        StringBuilder pageBuffer = new StringBuilder();
        while ((line = reader.readLine()) != null) {
            pageBuffer.append(line);
        }
        //返回网页内容
        return pageBuffer.toString();
    }
    /**
     * 测试代码
     */
    public static void main(String[] args) {
        //抓取 lietu 首页然后输出
        System.out.println(RetrivePage.downloadPage("http://www.lietu.com"));
    }
}

```

代码中的 `reader.readLine()` 有可能会抛出异常，因为网速可能不稳定，如果下载网页的过程中出现错误，还需要重试。如果重试仍然出错，下载线程可以调用 `sleep()` 方法休息片刻等待网速稳定。

用 `Scanner` 对象下载网页的例子：

```

Scanner scanner = new Scanner(new InputStreamReader(
                                pageURL.openStream(),"utf-8")); //指定编码格式 utf-8
scanner.useDelimiter("\\z"); //可以用正则表达式分段读取网页
//读取网页内容
StringBuilder pageBuffer = new StringBuilder();
while (scanner.hasNext()){
    pageBuffer.append( scanner.next() );
}

```

很多网页的编码格式是 `utf-8`，如果读入有乱码，则可以填入 `Firefox` 识别出的编码。下一章会专门介绍如何自动识别网页的编码。

网络中的数据是通过 `TCP/IP` 协议来传输的。一般使用套接字来对 `TCP/IP` 协议编程。`URL` 对象的 `openStream` 方法使用了 `HTTP` 的 `GET` 命令返回 `Web` 页的正文内容。下面是一个直接使用套接字(`socket`)向 `Web` 服务器发送 `GET` 命令并输出返回结果的例子：

```

String host = "www.lietu.com"; //主机名
String file = "/index.jsp"; //网页路径
int port = 80; //端口号，默认是 80

```



```
s = new Socket(host, port);

OutputStream out = s.getOutputStream();
PrintWriter outw = new PrintWriter(out, false);
outw.print("GET " + file + " HTTP/1.0\r\n");
outw.print("Accept: text/plain, text/html, text/*\r\n");
outw.print("\r\n");
outw.flush();//发送 GET 命令

InputStream in = s.getInputStream();
// InputStreamReader 的构造方法的第二个参数可以指定下载网页的编码格式
InputStreamReader inr = new InputStreamReader(in);
BufferedReader br = new BufferedReader(inr);
String line;
while ((line = br.readLine()) != null){
    System.out.println(line); //输出返回的网页
}
```

Web 服务器返回的结果除了网页内容，在此之前还包括头域(header field)信息，例如：

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=UTF-8
Content-Length: 5367
Date: Tue, 29 Jun 2010 00:32:12 GMT
Connection: close
```

网页的编码方式由 Content-Encoding 或 Content-Type 定义，它的长度由 Content-Length 或 Content-Range 定义。

URLConnection 类把头信息封装成了 HashMap 的形式，例如：key 是“Content-Length”，而 value 是“5367”。表 2-4 介绍了 HttpURLConnection 中的方法。

表 2-4 HttpURLConnection 中的方法

方法	描述
getHeaderField	取得头域信息
getContentType	取得网页类型

getLastModified	取得网页修改时间，如果没有则返回 0
getContentEncoding	取得编码类型
getContentLength	取得网页长度

getContentType 等方法的错误之处在于只识别头信息中小写的字符串，需要修改成大小写不敏感。

```
public String getHeaderField(String fieldKey) throws IOException {
    URLConnection con = url.openConnection();
    header = con.getHeaderFields(); //返回一个 HashMap

    Iterator i = getHeaderFields().keySet().iterator();
    String key = null;
    while (i.hasNext()) {
        key = (String) i.next();
        if (key == null) {
            if (fieldKey == null) {
                return (String) ((List) (getHeaderFields().get(null))).get(0);
            }
        } else {
            if (key.equalsIgnoreCase(fieldKey)) {
                return (String) ((List) (getHeaderFields().get(key))).get(0);
            }
        }
    }
    return null;
}
```

2.3.2 HTTP 协议

网络资源一般是 Web 服务器上的一些各种格式的文件。通过 HTTP 协议和 Web 服务器打交道，所以 Web 服务器又叫做 HTTP 服务器。HTTP 服务器存储了互联网上的数据并且根据 HTTP 客户端的请求提供数据。网络爬虫也是一种 HTTP 客户端。

在 Java 中，一旦和被采集的 Web 服务器建立网络连接，对网络资源的操作就好像对本地文件的操作一样简单。通常，网络中使用 URL(统一资源定位符)来标示网络资源的位置。可以直接通过 Java 中的 URL 类和存放网页的服务器建立连接，并且获得网页源代码。

Java 中的 URL 类代表一个统一的资源定位符，资源可以是简单的文件或目录，也可以是对更为复杂的对象的引用。例如，以下是一个 URL 的例子。

```
http://www.lietu.com/index.jsp
```

通常，URL 可分成几个部分。上面的 URL 示例指示使用的协议为超文本传输协议 (HTTP) 并且该信息驻留在一台名为 `www.lietu.com` 的主机上，可以调用 URL 类的 `getHost()` 方法取得 URL 地址的主机名。主机上的信息名称为 `/index.jsp`。主机上此名称的准确含义取决于协议和主机。该信息一般存储在文件中，但可以随时生成。该 URL 的这一部分称为路径部分，可以调用 `getPath()` 方法取得路径部分。

需要通过域名服务 (Domain Name Service 简称为 DNS) 取得域名对应的 IP 地址。爬虫程序首先连接到一个 DNS 服务器上，由 DNS 服务器返回域名对应的 IP 地址。使用不适当的 DNS 服务器可能导致无法解析有些域名。在 Linux 下 DNS 解析的问题可以用 `dig` 或 `nslookup` 命令来分析，例如：

```
#dig www.lietu.com
```

```
#nslookup www.lietu.com
```

如果需要更换更好的 DNS 域名解析服务器，可以编辑 DNS 配置文件 “`/etc/resolv.conf`”。DNS 解析是一个网络爬虫性能瓶颈。由于域名服务的分布式特点，DNS 可能需要多次请求转发，并在互联网上往返，需要几秒有时甚至更长的时间解析出 IP 地址。一个补救措施是引入 DNS 缓存，这样最近完成 DNS 查询的网址可能会在 DNS 缓存中找到，避免了访问互联网上的 DNS 服务器。JDK1.6 内部有个 30 秒的 DNS 缓存。通过 `sun.net.InetAddressCachePolicy.get()` 方法可以查看缓存时间设置。Java 中要查找一个域名的 IP 地址最方便的办法就是调用 `java.net.InetAddress.getByName("www.lietu.com")`。

URL 可选择指定一个“端口”，它是用于建立到远程主机 TCP 连接的端口号。如果未指定该端口号，则使用协议默认的端口。HTTP 协议的默认端口号是一个很吉利的数字：80。可以在 URL 地址中显式的指定端口号，如下所示：

```
http://www.lietu.com:80/index.jsp
```

为了深入的了解下载网页的原理，需要了解 HTTP 协议。HTTP 是一个客户端和服务端请求和应答的标准。HTTP 的客户端往往是 Web 浏览器，但是在这里是网络爬虫。客户端也叫做用户代理 (user agent)。服务器端是网站，例如 Tomcat。客户端发起一个到服务器上指定端口 (默认端口为 80) 的 HTTP 请求，服务器端按指定格式返回网页或者其他网络资源。

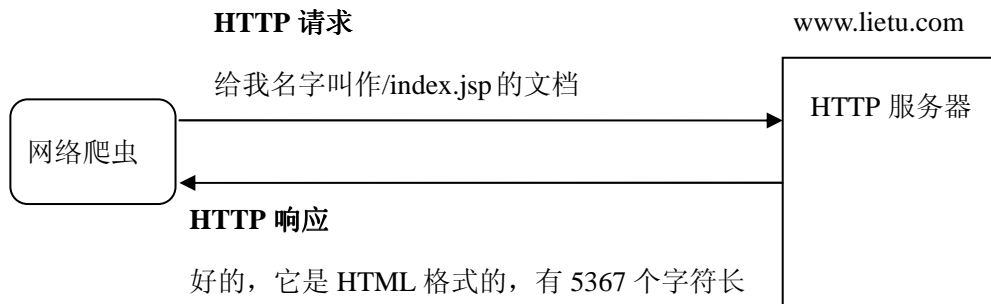


图 2-5 HTTP 协议的原理

URI 包括 URL 和 URN。但是 URN 并不常用，很少有人知道 URN。URL 由 3 部分组成，如下图所示：



图 2-6 URL 分为三部分

客户端向服务器发送的请求头包含请求的方法、URL、协议版本、以及包含请求修饰符、客户信息和内容。服务器以一个状态行作为响应，相应的内容包括消息协议的版本，成功或者错误编码加上包含服务器信息、实体元信息以及可能的实体内容。

HTTP 请求格式是：

```
<request line>
<headers>
<blank line>
[<request-body>]
```

在 HTTP 请求中，第一行必须是一个请求行(request line)，用来说明请求类型、要访问的资源以及使用的 HTTP 版本。紧接着是头信息(header)，用来说明服务器要使用的附加信息。在头信息之后是一个空行，再此之后可以添加任意的其他数据，这些附加的数据称之为主体(body)。

HTTP 规范定义了 8 种可能的请求方法。爬虫经常用到 GET、HEAD 和 POST 三种，分别说明如下：

- **GET**: 检索 URI 中标识资源的一个简单请求。例如爬虫发送请求: GET /index.html HTTP/1.1
- **HEAD**: 与 GET 方法相同, 服务器只返回状态行和头标, 并不返回请求文档。例如用 HEAD 请求检查网页更新时间。
- **POST**: 服务器接受被写入客户端输出流中的数据的请求。可以用 POST 方法来提交表单参数。

例如请求头:

```
Accept: text/plain, text/html
```

客户端说明了可以接收文本类型的信息, 最好不要发送音频格式的数据。

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

代表从这个网页知道正在请求的网页。

```
Accept-Charset: GB2312,utf-8;q=0.7
```

每个语言后包括一个 q-value。表示用户对这种语言的偏好估计。缺省值是 1.0, 这个也是最大值。

```
Keep-alive: 115
```

```
Connection: keep-alive
```

Keep-alive 是指在同一个连接中发出和接收多次 HTTP 请求, 单位是毫秒。

介绍完客户端向服务器的请求消息后, 然后再了解服务器向客户端返回的响应消息。这种类型的消息也是由一个起始行, 一个或者多个头信息, 一个指示头信息结束的空行和可选的消息体组成。

HTTP 的头信息包括通用头, 请求头, 响应头和实体头四个部分。每个头信息由一个域名, 冒号 (:) 和域值三部分组成。域名是大小写无关的, 域值前可以添加任何数量的空格符, 头信息可以被扩展为多行, 在每行开始处, 使用至少一个空格或制表符。

例如, 爬虫程序发出 GET 请求:

```
GET /index.html HTTP/1.1
```

服务器返回响应:

```
HTTP /1.1 200 OK
```

```
Date: Apr 11 2006 15:32:08 GMT
```

```
Server: Apache/2.0.46(win32)
```

```
Content-Length: 119
```

```
Content-Type: text/html
```

```
<HTML>
<HEAD>
<LINK REL="stylesheet" HREF="index.css">
</HEAD>
<BODY>
<IMG SRC="image/logo.png">
</BODY>
</HTML>
```

因为 HTTP 使用 TCP/IP，是基于文本的，和一些使用二进制格式的协议不同，直接和 Web 服务器交互很简单。

Telnet 工具连接键盘输入到目的 TCP 端口，并且连接 TCP 端口输出返回到显示屏。Telnet 本来是用于远程终端会话。但是可以用它连接到任何 TCP 服务器，包括 HTTP 服务器。

使用 Telnet 获取 URL 地址 <http://www.lietu.com:80/index.jsp> 所在的网页：

```
# telnet www.lietu.com 80
Trying 211.147.214.145...
Connected to www.lietu.com (211.147.214.145).
Escape character is '^'.
GET /index.jsp HTTP/1.1
Host: www.lietu.com

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=UTF-8
Content-Length: 5367
Date: Sat, 21 May 2011 09:59:09 GMT
```

在提交表单的时候，如果不指定方法，则默认为 GET 请求，表单中提交的数据将会附加在 url 之后，以?与 url 分开。字母数字字符原样发送，但空格转换为“+”号，其它符号转换为%XX，其中 XX 为该符号以 16 进制表示的 ASCII 值。GET 请求提交的数据放置在 HTTP 请求协议头中，而 POST 提交的数据则放在实体数据中。GET 方式提交的数据最多只能有 1024 字节，而 POST 则没有此限制。

程序发出 POST 请求的例子：

```
try {
    // 构造 POST 数据
    String data = URLEncoder.encode("key1", "UTF-8") +
```

```

        "=" + URLEncoder.encode("value1", "UTF-8");
data += "&" + URLEncoder.encode("key2", "UTF-8") +
        "=" + URLEncoder.encode("value2", "UTF-8");

// 创建一个到 Web 服务器的套接字
String hostname = "hostname.com";
int port = 80;
InetAddress addr = InetAddress.getByName(hostname);
Socket socket = new Socket(addr, port);

// 发送头信息
String path = "/servlet/SomeServlet";
BufferedWriter wr =
    new BufferedWriter(new OutputStreamWriter(socket.getOutputStream(), "UTF8"));
wr.write("POST "+path+" HTTP/1.0\r\n");
wr.write("Content-Length: "+data.length()+"\r\n");
wr.write("Content-Type: application/x-www-form-urlencoded\r\n");
wr.write("\r\n");

// 发送 POST 数据
wr.write(data);
wr.flush();

// 取得响应
BufferedReader rd =
    new BufferedReader(new InputStreamReader(socket.getInputStream()));
String line;
while ((line = rd.readLine()) != null) {
    // 处理读入行...
}
wr.close();
rd.close();
} catch (Exception e) {
}

```

例如，程序发出 HEAD 请求：

```
HEAD /index.jsp HTTP/1.0
```

服务器返回响应:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=UTF-8
Content-Length: 5367
Date: Fri, 08 Apr 2011 11:08:24 GMT
Connection: close
```

使用 TCP/IP 协议来传输数据。

2.3.3 使用 HttpClient 下载网页

在实际的项目中，网络环境比较复杂，因此，只用 `java.net` 包中的 API 来模拟浏览器客户端的工作，代码量非常大。例如，需要处理 HTTP 返回的状态码，设置 HTTP 代理，处理 HTTPS 协议，设置 Cookie 等工作。为了便于应用程序的开发，实际开发时常常使用开源项目 HttpClient(<http://hc.apache.org/httpcomponents-client-ga/>)。HttpClient 的 JavaDoc API 说明文档可以在 `httpcomponents-core-4.1-bin.zip` 找到。下载地址是 <http://hc.apache.org/downloads.cgi>。

它完全能够处理 HTTP 连接中的各种问题，使用起来非常方便。只需在项目中引入 HttpClient.jar 包，就可以模拟浏览器来获取网页内容。例如：

```
//创建一个客户端，类似于打开一个浏览器
DefaultHttpClient httpclient = new DefaultHttpClient();

//创建一个 GET 方法，类似于在浏览器地址栏中输入一个地址
HttpGet httpget = new HttpGet("http://www.lietu.com/");

//类似于在浏览器地址栏中输入回车，获得网页内容
HttpResponse response = httpclient.execute(httpget);

//查看返回的内容，类似于在浏览器察看网页源代码
HttpEntity entity = response.getEntity();
if (entity != null) {
    //读入内容流，并以字符串形式返回，这里指定网页编码是 UTF-8
    System.out.println(EntityUtils.toString(entity,"utf-8"));
    EntityUtils.consume(entity);//关闭内容流
}
```



```
//释放连接
```

```
httpClient.getConnectionManager().shutdown();
```

在这个示例中，只是简单地把返回的内容打印出来，而在实际项目中，通常需要把返回的内容写入本地文件并保存。最后还要关闭网络连接，以免造成资源消耗。

这个例子是用 GET 方式来访问 Web 资源。通常，GET 请求方式把需要传递给服务器的参数作为 URL 的一部分传递给服务器。但是，HTTP 协议本身对 URL 字符串长度有所限制。因此不能传递过多的参数给服务器。为了避免这种问题，有些表单采用 POST 方法提交数据，HttpClient 包对 POST 方法也有很好的支持。例如按城市抓取酒店信息。<http://hotels.ctrip.com/Domestic/SearchHotel.aspx>网页中包括如下源代码信息：

```
<form name="aspnetForm" method="post" action="SearchHotel.aspx" id="aspnetForm">
    <input type="hidden" name="cityId" value="" />
    <input type="hidden" name="checkIn" value="" />
    <input type="hidden" name="checkOut" value="" />
</form>
```

POST 方法需要提交的三个参数包括：所在城市(cityId)、入住日期(checkIn)、离店日期(checkOut)。提交一个参数包括名字和值两项。NameValuePair 是一个接口，而 BasicNameValuePair 则是这个接口的实现。使用 BasicNameValuePair 封装名字/值对。例如参数名 cityId 对应的值是 1，可以这样写：

```
new BasicNameValuePair("cityId", "1");
```

模拟提交表单并返回结果的代码如下：

```
HttpClient httpClient = new DefaultHttpClient();
HttpPost httpPost = new HttpPost("http://hotels.ctrip.com/Domestic/ShowHotelList.aspx");

// POST 数据
List<NameValuePair> nameValuePairs = new ArrayList<NameValuePair>(3);//3 个参数
nameValuePairs.add(new BasicNameValuePair("checkIn", "2011-4-15")); //入住日期
nameValuePairs.add(new BasicNameValuePair("checkOut", "2011-4-25")); //离店日期
nameValuePairs.add(new BasicNameValuePair("cityId", "1")); //城市编码
httpPost.setEntity(new UriEncodedFormEntity(nameValuePairs));

// 执行 HTTP POST 请求
HttpResponse response = httpClient.execute(httpPost);

//取得内容流
InputStream is = response.getEntity().getContent();
BufferedInputStream bis = new BufferedInputStream(is);
```

```

ByteArrayBuffer baf = new ByteArrayBuffer(20);

//按字节读入内容流到字节数组缓存
int current = 0;
while ((current = bis.read()) != -1) {
    baf.append((byte) current);
}

String text = new String(baf.toByteArray(), "gb2312"); // 指定编码
System.out.println(text);

```

上面的例子说明了如何使用 POST 方法来访问 Web 资源。与 GET 方法不同，POST 方法可以提交二进制格式的数据，因此可以传递“无限”多的参数。而 GET 方法采用把参数写在 URL 里面的方式，由于 URL 有长度限制，因此传递参数的长度会有限制。

为了自动搜索 Google，可以使用如下的 GET 请求：

```

HttpGet httpget = new HttpGet
("http://www.google.com/search?hl=en&q=httpclient&btnG=Google+Search&aq=f&oq=");

```

HttpClient 提供了一系列实用的方法来简化创建和修改请求 URI。URI 可以组装编程：

```

URI uri = URIUtils.createURI("http", "www.google.com", -1, "/search",
    "q=httpclient&btnG=Google+Search&aq=f&oq=", null);
HttpGet httpget = new HttpGet(uri);
//输出 http://www.google.com/search?q=httpclient&btnG=Google+Search&aq=f&oq=
System.out.println(httpget.getURI());

```

提交的参数可以分解成 Key=Value 的形式，叫做名字/值对。HttpClient 使用专门的 NameValuePair 类来表示名字/值对。通过添加参数列表来生成 Google 查询字符串的代码如下：

```

List<NameValuePair> qparams = new ArrayList<NameValuePair>();
qparams.add(new BasicNameValuePair("q", "httpclient")); //查询词是 httpclient
qparams.add(new BasicNameValuePair("btnG", "Google Search"));
//判断搜索用户是否是第一次查询，如果用户第一次进行查询，则 aq=f
qparams.add(new BasicNameValuePair("aq", "f"));
qparams.add(new BasicNameValuePair("oq", null));
URI uri = URIUtils.createURI("http", "www.google.com", -1, "/search",
    URLEncodedUtils.format(qparams, "UTF-8"), null);

HttpGet httpget = new HttpGet(uri);
//输出 http://www.google.com/search?q=httpclient&btnG=Google+Search&aq=f&oq=

```

```
System.out.println(httpget.getURI());
```

HttpClient 是一个接口，用来代表基本的 HTTP 请求执行约定。而 DefaultHttpClient 是 HttpClient 接口的一个实现类。如果没有特别指定，DefaultHttpClient 使用 HTTP/1.1 版本，设置如下的参数：

- Version: HttpVersion.HTTP_1_1
- ContentCharset: HTTP.DEFAULT_CONTENT_CHARSET
- NoTcpDelay: true
- SocketBufferSize: 8192
- UserAgent: Apache-HttpClient/release (java 1.5)

此外还可以使用 AbstractHttpClient 定制 HttpClient。可以用 AbstractHttpClient 代替 DefaultHttpClient。

```
AbstractHttpClient httpImpl = new ContentEncodingHttpClient ();

HttpGet httpget = new HttpGet(uri);
httpget.addHeader ("User-Agent",
    "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)");
httpget.addHeader ("Accept-Encoding", "gzip,deflate");
httpImpl.execute (httpget);
```

HttpGet 是 HttpRequestBase 的子类。使用 HttpGet 的例子：

```
HttpClient httpclient = new DefaultHttpClient();

HttpHost targetHost = new HttpHost("www.google.cn");
HttpGet httpget = new HttpGet("/");

// 查看默认 request 头部信息
System.out.println("Accept-Charset:"
    + httpget.getFirstHeader("Accept-Charset"));
//以下这条如果不加会发现无论你设置 Accept-Charset 为 gbk 还是 utf-8，都会默认返回
gb2312（本例针对 google.cn 来说）
httpget.setHeader("User-Agent",
    "Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN; rv:1.9.1.2)");
// 用逗号分隔显示可以同时接受多种编码
```

```
httpget.setHeader("Accept-Language", "zh-cn,zh;q=0.5");
httpget.setHeader("Accept-Charset", "GB2312,utf-8;q=0.7,*;q=0.7");
HttpResponse response = httpclient.execute(targetHost, httpget);
```

EntityUtils 提供了一些方法来读取 HttpEntity 实体中的内容。例如：

```
EntityUtils.toString(entity,"utf-8");
```

上面的代码读入实体中的内容流，并以字符串形式返回，这里指定实体中的内容以 UTF-8 编码。

有些网站页面内容返回格式为 gzip 压缩格式，所以在得到返回结果后要判断内容是否压缩过，如果是，则先要解压缩，然后再解析内容。这样的网页返回的头信息会说明：“Content-Encoding: gzip”。

```
public class ClientGZipContentCompression {

    public final static void main(String[] args) throws Exception {
        DefaultHttpClient httpclient = new DefaultHttpClient();
        try {
            httpclient.addRequestInterceptor(new HttpRequestInterceptor() {
                public void process(
                    final HttpRequest request,
                    final HttpContext context)throws HttpException, IOException {
                    if (!request.containsHeader("Accept-Encoding")) {
                        request.addHeader("Accept-Encoding", "gzip");
                    }
                }
            });

            httpclient.addResponseInterceptor(new HttpResponseInterceptor() {
                public void process(
                    final HttpResponse response,
                    final HttpContext context)throws HttpException, IOException {
                    HttpEntity entity = response.getEntity();
                    Header ceheader = entity.getContentEncoding();
                    if (ceheader != null) {
                        HeaderElement[] codecs = ceheader.getElements();
                        for (int i = 0; i < codecs.length; i++) {
                            if (codecs[i].getName().equalsIgnoreCase("gzip")) {
                                response.setEntity(
```

```
                new
GzipDecompressingEntity(response.getEntity()));
                return;
            }
        }
    }
}

});

HttpGet httpget = new HttpGet("http://www.sohu.com");

// Execute HTTP request
System.out.println("executing request " + httpget.getURI());
HttpResponse response = httpclient.execute(httpget);

System.out.println("-----");
System.out.println(response.getStatusLine());
System.out.println(response.getLastHeader("Content-Encoding"));
System.out.println(response.getLastHeader("Content-Length"));
System.out.println("-----");

HttpEntity entity = response.getEntity();

if (entity != null) {
    String content = EntityUtils.toString(entity, "GBK");
    System.out.println(content);
    System.out.println("-----");
    System.out.println("Uncompressed size: "+content.length());
}

} finally {
    // When HttpClient instance is no longer needed,
    // shut down the connection manager to ensure
    // immediate deallocation of all system resources
    httpclient.getConnectionManager().shutdown();
}
```

```

    }

    static class GzipDecompressingEntity extends HttpEntityWrapper {
        public GzipDecompressingEntity(final HttpEntity entity) {
            super(entity);
        }

        @Override
        public InputStream getContent() throws IOException, IllegalStateException {
            // the wrapped entity's getContent() decides about repeatability
            InputStream wrappedin = wrappedEntity.getContent();
            return new GZIPInputStream(wrappedin);
        }

        @Override
        public long getContentLength() {
            // length of ungzipped content is not known
            return -1;
        }
    }
}

```

2.3.4 重定向

客户端浏览器必须采取更多操作来实现请求。例如，浏览器可能不得不请求服务器上的不同的页面，或通过代理服务器重复该请求。

HttpClient 可以处理任何如下的和重定向相关的响应代码：

- 301 永久移动。HttpStatus.SC_MOVED_PERMANENTLY
- 302 临时移动。HttpStatus.SC_MOVED_TEMPORARILY
- 303 See Other。HttpStatus.SC_SEE_OTHER
- 307 临时重定向。HttpStatus.SC_TEMPORARY_REDIRECT

例如：redirect.cgi 发送一个 302 响应，同时提供一个 Location 头信息指向 redirect2.cgi：

HTTP/1.1 302 Moved

```
Date: Sat, 15 May 2004 19:30:49 GMT
Server: Apache/2.0.48 (Fedora)
Location: /cgi-bin/jccook/redirect2.cgi
Content-Length: 0
Content-Type: text/plain; charset=UTF-8
```

如果使用 `HttpClient`，则可以通过如下代码得到跳转的网址：

```
HttpResponse response = httpclient.execute(httppost);
response.getLastHeader("Location"); //跳转的网址
response.getStatusLine().getStatusCode(); //返回的状态码
```

另外有些网页通过 `JavaScript` 跳转实现重定向。以如下这个页面为例：

<http://www.universidadperu.com/empresas/aliterm-sociedad-anonima-cerrada.php>

查看这个网页的代码。重定向的代码就是这个 `JavaScript`：

```
<form name="InfoEmpresa" action="" id="InfoEmpresa" onsubmit="var work
= document.InfoEmpresa.buscaempresa.value;work=work.replace(/ /g,
'-');work=work.toLowerCase();window.location='http://www.universidadperu.com/empres
as/busqueda/'
+ work; return false;">
```

如果搜索“BANCO CONTINENTAL”这个公司名，则用正则表达式把“BANCO CONTINENTAL”中间的空格替换成-，然后小写化之后跳转到：

<http://www.universidadperu.com/empresas/busqueda/banco-continental>

2.3.5 解决套结字连接限制

在 Windows 平台下，当爬虫运行多日以后，由于操作系统对于进程废弃的网络连接回收不完全，导致爬虫出错，只有重起操作系统后才能继续运行爬虫。所以要考虑使用连接池来重用网络连接。

在 Linux 下对连接 `socket` 数量也有限制，可以通过系统配置增加 `socket` 数量上限。当然也可以使用 `socket connection pool` 连接池循环利用 `socket` 连接。

增加短暂的端口范围，并减少 `fin_timeout`。用下面两个命令发现缺省值：

```
#sysctl net.ipv4.ip_local_port_range
```

```
#sysctl net.ipv4.tcp_fin_timeout
```

短暂端口范围定义了一个主机可以从一个特定的 IP 地址创建的出去 socket 的最大的数量。fin_timeout 定义了这些 socket 保持在 TIME_WAIT 状态的最小时间。

通常系统缺省值是：

```
net.ipv4.ip_local_port_range = 32768 61000
```

```
net.ipv4.tcp_fin_timeout = 60
```

这基本上意味着在任何时候你的系统不能保证多于 $(61000 - 32768) / 60 = 470$ socket。如果觉得还不够多，可以从增加 port_range 开始。设置范围到 15000 61000 很常见。可以通过减少 fin_timeout 进一步增加可获得的连接。同时做这两步，可以有多于 1500 个出连接。

sysctl 是一个允许您改变正在运行中的 Linux 系统的接口。它包含一些 TCP/IP 堆栈和虚拟内存系统的高级选项，这可以让有经验的管理员提高引人注目的系统性能。可以使用 sysctl 修改系统变量。

```
#sysctl -w net.ipv4.tcp_fin_timeout=30
```

```
#sysctl -w net.ipv4.ip_local_port_range="15000 61000"
```

也可以通过编辑 sysctl.conf 文件来修改系统变量。sysctl.conf 看起来很像 rc.conf。它用 variable=value 的形式来设定值。修改配置文件，增加如下行到/etc/sysctl.conf：

```
# increase system IP port limits
```

```
net.ipv4.ip_local_port_range = 1024 65535
```

可以使用如下的命令检查是否已经正确的设置了这个值：

```
#cat /proc/sys/net/ipv4/ip_local_port_range
```

当一个新的连接请求进来的时候，连接池管理器检查连接池中是否包含任何没用的连接，如果有的话，就返回一个。如果连接池中所有的连接都忙并且最大的连接池数量没有达到，就创建新的连接并且增加到连接池。当连接池中在用的连接达到最大值，所有的新连接请求进入队列，直到一个连接可用或者连接请求超时。

HttpClient 有自己的连接池管理器。ThreadSafeClientConnManager 管理一个连接池，能够为多个执行线程的连接请求提供服务。按路由提供连接缓存。

ThreadSafeClientConnManager 维护最大的连接限制。每个路由不超过 2 个并行连接。总共不超过 20 个连接。可以使用 HTTP 参数调整连接限制。

```
public class HttpClientUtil {
    private static HttpClient httpClient = null;
```



```
private static final Integer MAX_CONNECTIONS = 10;
private static final String HTTP = "http";
private static final Integer HTTP_SCHEME_PORT = 80;
private static final Integer HTTPS_SCHEME_PORT = 443;
private static final String HTTPS = "https";
private static ThreadSafeClientConnManager threadSafeClientConnManager = null;

public static HttpClient getMultiThreadedClient() {
    if (httpClient == null) {
        httpClient = new DefaultHttpClient(getThreadSafeConnectionManager(),
getParams());
    }
    return httpClient;
}

public static ThreadSafeClientConnManager getThreadSafeConnectionManager() {
    if (threadSafeClientConnManager == null) {
        // Create and initialize scheme registry
        SchemeRegistry schemeRegistry = new SchemeRegistry();
        schemeRegistry.register(new
Scheme(HTTP,HTTP_SCHEME_PORT,PlainSocketFactory.getSocketFactory()));
        schemeRegistry.register(new
Scheme(HTTPS,HTTPS_SCHEME_PORT,SSLSocketFactory.getSocketFactory()));

        // Create an HttpClient with the ThreadSafeClientConnManager.
        // This connection manager must be used if more than one thread will
        // be using the HttpClient.
        threadSafeClientConnManager =
            new ThreadSafeClientConnManager(schemeRegistry);
        threadSafeClientConnManager.setMaxTotal(MAX_CONNECTIONS);
    }
    return threadSafeClientConnManager;
}

private static HttpParams getParams() {
    // Create and initialize HTTP parameters
    HttpParams params = new BasicHttpParams();
    HttpProtocolParams.setVersion(params, HttpVersion.HTTP_1_1);
}
```

```

        return params;
    }
}

```

使用连接池:

```

HttpClient httpclient = HttpClientConnManager.getMultiThreadedClient();
HttpGet httpget = new HttpGet(url);
HttpEntity entity = httpclient.execute(httpget).getEntity();

```

2.3.6 下载图片

为了减少存储空间, 需要把 BMP 格式的图片转换成 JPG 格式。javax.imageio.ImageIO 类的 write 方法可以把 BufferedImage 对象保存成 JPG 格式。

```

public class ReadImage {
    public static void download(String imageUrl,String imageFileName) {
        URL url=new URL(imageUrl);

        Image src = javax.imageio.ImageIO.read(url); //构造 Image 对象
        int width = src.getWidth(null); //得到源图宽
        int height = src.getHeight(null); //得到源图长
        BufferedImage thumb = new BufferedImage(width / 1, height / 1,
                                                BufferedImage.TYPE_INT_RGB);

        //绘制缩小后的图
        thumb.getGraphics().drawImage(src, 0, 0, width / 1, height / 1, null);
        File file = new File(imageFileName); //输出到文件流
        ImageIO.write(thumb,"jpg", file);
    }
    public static void main(String[] args) throws Exception {
        ReadImage.download("http://www.51766.com/img/jhzdd/1167040251883.bmp",
                            "D:/HH.jpg");
    }
}

```

2.3.7 抓取 FTP

Commons VFS(<http://commons.apache.org/vfs/index.html>)提供一个统一的 API 用于处理各种不同的文件系统, 包括 FTP 中的文件或者 Zip 压缩包中的文件。使用 VFS 遍历 FTP 中的文件的例子:

```

FileSystemManager manager = VFS.getManager();
FileObject ftpFile = manager.resolveFile("ftp://hcl:hcl@localhost:21/loveapple");
FileObject[] children = ftpFile.getChildren();
System.out.println("Children of " + ftpFile.getName().getURI());
for (FileObject child : children) {
    String baseName = child.getName().getBaseName();
    System.out.println("文件名: " + baseName + " -- "
        + new String(baseName.getBytes("iso-8859-1"), "UTF-8"));
}

```

2.3.8 RSS 抓取

因为 XML 比 HTML 更规范。所以出现了 XML 格式封装的数据源。RSS 是对网站栏目的一种 XML 格式的封装。一些博客或者新闻网站提供 RSS(Really Simple Syndication)格式的输出，方便程序快速访问更新的信息。RSS 抓取的第一步是解析 RSS 数据源。Informa(<http://informa.sourceforge.net/>)提供了一个解析包。ROME(<https://rome.dev.java.net/>)是另外一个常用的解析包。WebNews Crawler(<http://senews.sourceforge.net/>)是一个在 Informa 基础上构架的新闻爬虫。

RSS 种子(Feed)就是一个发布最新信息的 URL 地址。为了读取一个 RSS 种子，首先定义读取种子的源，然后定义构建新闻频道对象模型的 ChannelBuilder。

```

ChannelBuilder builder = new ChannelBuilder();
String url = "http://rss.news.yahoo.com/rss/topstories"; // RSS 种子
Channel channel = (Channel) FeedParser.parse(builder, url);
System.out.println("标题: " + channel.getTitle());
System.out.println("描述: " + channel.getDescription());
System.out.println("内容:");
for (Object x : channel.getItems()) { //遍历最新的新闻条目
    Item anItem = (Item) x;
    System.out.print("标题:"+anItem.getTitle() + " 描述: ");
    System.out.println(anItem.getDescription());
}

```

如何从网页发现 RSS 种子？下面是一个对 RSS 种子的描述：

```

<link          href="http://blog.csdn.net/myth1979/rss.aspx"          title="RSS"
type="application/rss+xml" rel="alternate" />

```

根据 type="application/rss+xml" 可以确定 link 标签中包含的 URI <http://blog.csdn.net/myth1979/rss.aspx> 是 RSS 种子。利用 HTMLParser 来解析出有效的 Feed 的程序如下：

```

TagNode tagNode = (TagNode)node;
String name = ((TagNode)node).getTagName();
if (name.equals("LINK") && !tagNode.isEndTag() ){
    String href=tagNode.getAttribute("HREF");
    if(href!=null &&
        (href.indexOf("rss")>=0 ||
         href.indexOf("feed")>=0 ||
         href.indexOf("rdf")>=0||
         href.indexOf("xml")>=0||
         href.indexOf("atom")>=0)) {
        //验证 Feed 的有效性。
        boolean ret = rParser.valid(href);
        if(ret)
            rrsUrls.add(href);
        if("alternate".equals(tagNode.getAttribute("REL"))) {
            outURLs.clear();
            System.out.println("end find");
            return rrsUrls;
        }
    }
}
}

if( name.equals("A") ) {
    String href = tagNode.getAttribute("HREF");
    if(href != null &&
        (href.indexOf("rss")>=0 ||
         href.indexOf("feed")>=0 ||
         href.indexOf("rdf")>=0||
         href.indexOf("xml")>=0||
         href.indexOf("atom")>=0)) {
        boolean ret = rParser.valid(href);
        if(ret)
            rrsUrls.add(href);
    }
    if(href != null &&
        outURLs!=null &&
        href.startsWith("http://") &&
        href.indexOf(domain)>=0) {

```

```

        outURLs.add(href);
    }
}

```

从一个网页发现种子的步骤如下：

1. 首先，用一个函数来验证种子是否有效。
2. 如果这个 URI 已经指向一个种子，则只是返回它，否则分析这个页面。
3. 看这个页面的头信息是否包含 LINK 标签。
4. <A>链接到同一个服务器上以".rss"、".rdf"、".xml"或".atom"结尾的种子。
5. <A>链接到同一个服务器上包含".rss"、".rdf"、".xml"或".atom"的种子。
6. <A>链接到以".rss"、".rdf"、".xml"或".atom"结尾的外部服务器种子。
7. <A>链接到包含".rss"、".rdf"、".xml"或".atom"的外部服务器种子。
8. 尝试猜测一些可能有 Feed 的通用的地方，例如 index.xml、atom.xml 等。

RSS 抓取流程是：首先从网站中自动发现 RSS。然后遍历每一个 RSS，必要的时候分析详细页面。

2.3.9 网页更新

经常有人会问：“有没有什么新消息？”这说明人的大脑是增量获取信息的，对爬虫来说也是如此。网站中的内容经常会变化，这些变化经常在网站首页或者目录页有反应。为了提高采集效率，往往考虑增量采集网页。可以把这个问题看成是被采集的 Web 服务器和存储库同步的问题。更新网页内容的基本原理是：下载网页时，记录网页下载时的时间，增量采集这个网页时，判断 URL 地址对应的网页是否有更新。

网页更新过程符合泊松过程。泊松过程是指一种累计随机事件发生次数的最基本的独立增量过程。例如，随着时间增长累计某电话交换台收到的呼唤次数就构成一个泊松过程。网页更新时间间隔符合泊松指数分布。对于不同类型的网站采用不同的更新策略。例如.com 域名的网站更新频率较高，而.gov 域名的网站更新频率低。

对于一些不太可能会更新的网页，只抓取一遍即可。但有些网页，例如首页或者列表页更新频率较高，所以需要隔一段时间就检测这些网页是否更新。如果只是想看看网页是否有更新，可以用 HTTP 的 HEAD 命令查看网页的最后修改时间。

```

String host = "www.lietu.com"; //主机名
String file = "/index.jsp"; //网页路径

```

```

int port = 80; //端口号
Socket s = new Socket(host, port); //建立套接字对象

OutputStream out = s.getOutputStream();
PrintWriter outw = new PrintWriter(out, false);
outw.print("HEAD " + file + " HTTP/1.0\r\n"); //发送 HEAD 命令
outw.print("Accept: text/plain, text/html, text/*\r\n");
outw.print("\r\n");
outw.flush();

InputStream in = s.getInputStream(); //返回头信息
InputStreamReader inr = new InputStreamReader(in);
BufferedReader br = new BufferedReader(inr);
String line;
while ((line = br.readLine()) != null) {
    System.out.println(line);
}

```

上面的程序在控制台打印结果:

```

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
ETag: W/"6810-1268491592000"
Last-Modified: Sat, 13 Mar 2010 14:46:32 GMT
Content-Type: text/html
Content-Length: 6810
Date: Tue, 15 Jun 2010 01:48:27 GMT
Connection: close

```

上面的输出结果中“Last-Modified”行记录了网页的修改时间。“Date”行返回的是 Web 服务器的当前时间。可以通过 `URLConnection` 对象取得网页的修改时间，代码如下：

```

URL u = new URL("http://www.lietu.com");
URLConnection http = (URLConnection) u.openConnection();
http.setRequestMethod("HEAD");
System.out.println(u + " 更新时间 " + new Date(http.getLastModified()));

```

有些网页头信息没有包括更新时间，这时候可以通过判断网页长度来检测网页是否有更新。当然也可能网页更新了，但是长度没变，但实际上，这种可能性非常小。

条件下载命令可以根据时间条件下载网页。再次请求已经抓取过的页面时，爬虫往 Web

服务器发送 **If-Modified-Since** 请求头，其中包含的时间是先前服务器端发过来的 **Last-Modified** 最后修改时间戳。这样让 Web 服务器端进行验证，通过这个时间戳判断爬虫上次抓过的页面是否有修改。如果有修改，则返回 **HTTP** 状态码 200 和新的内容。如果没有变化，则只返回 **HTTP** 状态码 304，告诉爬虫页面没有变化。这样可以大大减少在网络上传输的数据，同时也减轻了被抓取的服务器的负担。下面的代码通过套接字发送 **If-Modified-Since** 头信息。

```
outw.print("GET " + file + " HTTP/1.0\r\n");
outw.print("If-Modified-Since: Thu, 01 Jul 2011 07:24:54 GMT\r\n");
```

HTTP/1.1 中还有一个 **Etag** 可以用来判断请求的文件是否被修改。可以把 **Etag** 看成网页的版本标志。**Etag** 主要为了解决 **Last-Modified** 无法解决的一些问题：

- 1、一些文件也许会周期性的更改，但是它的内容并不改变(仅仅改变了修改时间)，这个时候我们并不希望客户端认为这个文件被修改了，而重新下载。
- 2、某些文件修改非常频繁，比如在秒以下的时间内进行修改，(比方说 1 秒内修改了 N 次)，**If-Modified-Since** 能检查到的粒度是秒级的，无法判断这种修改。
- 3、不能精确的得到某些 Web 服务器的文件的最后修改时间。

在第一次抓取时记录网页的 **Etag**。下次发起 **HTTP GET** 请求一个文件，同时发送一个 **If-None-Match** 头，这个头的内容就是我们第一次请求时 Web 服务器返回的 **Etag**：6810-1268491592000。如果已经修改，则返回 **HTTP** 状态码 200 和新的内容。如果没有修改，则 **If-None-Match** 为 **False**，返回 **HTTP** 状态码 304。例如：

```
outw.print("GET " + file + " HTTP/1.0\r\n");
outw.print("If-None-Match: \"1272af65f918cb1:84f\" \r\n");
```

可以用 **Range** 条件下载部分网页。比如某网页的大小是 1000 字节，爬虫请求这个网页时用 “**Range: bytes=0-500**”，那么 Web 服务器应该把这个网页开头的 501 个字节发回给爬虫。

2.3.10 抓取限制应对方法

为了避免抓取的网站响应请求的负担过重，爬虫需要遵循礼貌性原则，不要同时发起过多的下载网页请求。这样才可能有更多的网站对爬虫友好。为了减少网站对爬虫的抱怨，建议每秒只抓取几次，把抓取任务尽量平均分配到每个时间段，并且避免高峰时段对访问的网站负担过重。

有些网站为了保护自己的内容，采取各种方法限制爬虫抓取，常见的方法有：

- 判断 **HTTP** 请求的头信息，只有头信息和浏览器一样才正常返回结果。
- 当来自同一个 IP 的访问次数达到一定限制值后封 IP，也就是把这个 IP 列入黑名单，超

过一段时间后再解封。

- 采用 JavaScript 动态显示内容。

有些网站中的网页在浏览器中可以正常访问，但却不能用程序正常下载，这时候需要模仿浏览器下载网页。可以用 Firebug 看到 FireFox 浏览器发送的头信息。

可以发送和浏览器类似的头信息来简单的模仿浏览器访问。使用 `HttpURLConnection` 下载网页时，设置请求头信息：

```
HttpURLConnection urlConnection = (HttpURLConnection) pageURL.openConnection();
//设置和浏览器相同的头信息
urlConnection.setRequestProperty("User-Agent","MSIE 6.0");
urlConnection.setRequestProperty("Host", pageURL.getHost());
urlConnection.setRequestProperty("Accept-Language", "ru");
urlConnection.setRequestProperty("Accept-Charset", "gb2312,utf-8;q=0.7,*;q=0.7n");
urlConnection.setRequestProperty("content-type", "text/html");
```

使用 `HttpClient` 下载网页时，例子代码如下：

```
DefaultHttpClient httpclient = new DefaultHttpClient();
BasicHttpParams hp = new BasicHttpParams();
hp.setParameter("http.useragent", "Mozilla/1.0 (compatible; linux 2015 plus; yep)"); // you
oughta change this into your own UA string....
httpclient.setParams(hp);

try {
    HttpGet httpget = new HttpGet("http://www.lietu.com/");
    HttpContext HTTP_CONTEXT = new BasicHttpContext();
    HttpResponse response = httpclient.execute(httpget, HTTP_CONTEXT);

    HttpEntity entity = response.getEntity();
    if (entity != null) {
        System.out.println(EntityUtils.toString(entity,"utf-8"));
        EntityUtils.consume(entity);
    }
} finally {
    httpclient.getConnectionManager().shutdown();
}
```

有些登陆后才能看到的信息可以人工登陆后，手工在程序中设置动态的 Cookie 值。


```
//假设 Cookie 值在 cookieValue 中
uc.setRequestProperty("Cookie", cookieValue);
```

有些网站对于同一个 IP 在一段时间内的访问次数有限制。可以使用 Socket 代理来更改请求的 IP。这时可以通过大量不同的 Socket 代理循环访问网站。

首先建立有效代理列表，proxyIP.txt：

```
219.93.178.162:3128
222.135.79.253:8080
203.160.1.38:554
132.239.17.225:3124
169.229.50.5:3124
203.160.1.146:554
203.160.1.49:554
```

有些可以自动发现有效代理的软件，例如“花刺代理”等。然后建立 ProxyDB 类，循环使用这些 Socket 代理：

```
int pos = proxyIpList.get(count).toString().indexOf(":");
if (pos > 0) {
    String port = proxyIpList.get(count).toString().substring(pos+1);
    proxyAddr = proxyIpList.get(count).toString().substring(0,pos);
    proxyPort = Integer.parseInt(port);
    SocketAddress socketaddress = new InetSocketAddress(proxyAddr,proxyPort);
    proxy = new Proxy(Proxy.Type.HTTP,socketaddress);
}
```

最后通过下载网页的 URL 的 openConnection 方法使用它：

```
url.openConnection(ProxyDB.getProxy());
```

通过 Modem 方式上网的计算机，每次上网所分配到的 IP 地址都不相同，这就是动态 IP 地址。因为 IP 地址资源很宝贵，大部分用户都是通过动态 IP 地址上网的。所谓动态就是指，当你每一次上网时，电信或网通会随机给你分配一个 IP 地址。重新启动上网的 Modem 就可以更换 IP 地址，使用新的 IP 地址继续抓取信息。使用浏览器 Firefox 下的插件 Firebug 分析出点击“重启路由器”时，浏览器向路由器发送的请求内容如下：

```
GET
/userRpm/SysRebootRpm.htm?Reboot=%D6%D8%C6%F4%C2%B7%D3%C9%C6%F7
HTTP/1.1
Host:192.168.1.1
Authorization:Basic YWRtaW46YWRtaW4=
```

其中，Authorization 请求头的内容中，“Basic”表示“Basic authorization 验证”；

"YWRtaW46YWRtaW4="是使用 Base64 编码后的用户名和密码，解密后是"admin:admin"。

同样，也可以用程序实现以上的 HTTP 请求：

```
//modem 的用户名和密码
String data = "admin:admin";
String authorization = Base64.encode(data.getBytes());

String host = "192.168.1.1"; //modem 的 ip 地址
String file =
"/userRpm/SysRebootRpm.htm?Reboot=%D6%D8%C6%F4%C2%B7%D3%C9%C6%F7"; //网页路径
int port = 80; //端口号

Socket s = new Socket(host, port);

OutputStream out = s.getOutputStream();
PrintWriter outw = new PrintWriter(out, false);
outw.print("GET " + file + " HTTP/1.1\r\n");
outw.print("Authorization:Basic "+authorization+"\r\n");
outw.print("\r\n");
outw.flush();
```

为了找出抓取中的问题，可以采用网络工具 wireshark 追踪。

2.3.11 URL 地址提取

在 Windows 的控制台窗口中，可以根据当前路径的相对路径转移到一个路径，例如 `cd ..` 转移到当前路径的上级路径。在 HTML 网页中也经常使用相对 URL。

绝对 URL 就是不依赖其他的 URL 路径，例如："http://www.lietu.com/index.jsp"。在一定的上下文环境下可以使用相对 URL。网页中的 URL 地址可能是相对地址例如“./index.html”。可以在<A>和标签中使用相对 URL。例如：

```

```

可以根据所在网页的绝对 URL 地址，把相对地址转换成绝对地址。为了灵活的引用网站内部资源，相对路径在网页中很常见。爬虫为了后续处理方便，需要把相对地址转化为绝对地址。URL 对象的 `toString` 方法返回的是绝对地址，因此为了把相对地址转化成绝对地址，只需要生成相对地址对应的 URL 对象即可。`new URL(fromUrl, url)`方法可以实现从相对地址 url 和来源 URL 对象 fromUrl 生成新的 URL 对象。测试功能：

```
URL fromUrl = new URL("http://www.lietu.com/news/");
String url = "../index.html";//网页中的相对超链接地址
String newUrl = (new URL(fromUrl, url)).toString();//转成字符串类型
System.out.println(newUrl);//打印 http://www.lietu.com/index.html
```

封装成一个方法:

```
public static String parseRealURL(String urlSource, String url)
    throws MalformedURLException {
    URL from = new URL(urlSource);
    return (new URL(from, url)).toString();
}
```

有些相对地址的解析不正确, 例如, 源地址是:

```
http://www.bradfordexchange.com/mcategory/apparel-and-accessories_9750/womens-jackets.htm
l
```

相对地址是:

```
mcategory/apparel-and-accessories_9750/womens-jackets_pg2.html
```

采用如下的代码修正这个错误:

```
char firstChar = s.charAt(0);
if (firstChar >= 'a' && firstChar <= 'z' || firstChar >= 'A'
    && firstChar <= 'Z') {
    if (!s.startsWith("https:") && !s.startsWith("http:")) {
        s = "/" + s;
        URI newUri = baseURI.resolve(s);
        return newUri;
    }
}
```

有些网址是通过提交表单, 或者通过 JavaScript 来跳转的。可以参考 HtmlUnit (<http://htmlunit.sourceforge.net/>)中的相关实现来获取。

2.3.12 抓取需要登录的网页

很多网站的内容都只是对注册用户可见的, 这种情况下就必须要求使用正确的用户名和口令登录成功后, 方可浏览到想要的页面。因为 HTTP 协议是无状态的, 也就是连接的有效期只限于当前请求, 请求内容结束后连接就关闭了。在这种情况下为了保存用户的登录信息

必须使用到 Cookie 机制。以 JSP/Servlet 为例，当浏览器请求一个 JSP 或者是 Servlet 的页面时，应用服务器会返回一个参数，名为 `jsessionid`(因不同应用服务器而异)，值是一个较长的唯一字符串的 Cookie，这个字符串值也就是当前访问该站点的会话标识。浏览器在每访问该站点的其他页面时候都要带上 `jsessionid` 这样的 Cookie 信息，应用服务器根据读取这个会话标识来获取对应的会话信息。

对于需要用户登录的网站，一般在用户登录成功后会将用户资料保存在服务器的会话中，这样当访问到其他的页面时候，应用服务器根据浏览器送上的 Cookie 中读取当前请求对应的会话标识以获得对应的会话信息，然后就可以判断用户资料是否存在于会话信息中，如果存在则允许访问页面，否则跳转到登录页面中要求用户输入帐号和口令进行登录。这就是一般使用 JSP 开发网站在处理用户登录的比较通用的方法。

这样一来，对于 HTTP 的客户端来讲，如果要访问一个受保护的页面时就必须模拟浏览器所做的工作，首先就是请求登录页面，然后读取 Cookie 值；再次请求登录页面并加入登录页所需的每个参数；最后就是请求最终所需的页面。当然在除第一次请求外其他的请求都需要附上 Cookie 信息以便服务器能判断当前请求是否已经通过验证。HttpClient(<http://hc.apache.org/httpcomponents-client-ga/>)自动管理了 cookie 信息，只需要先传递登录信息执行登录过程，然后直接访问想要的页面，跟访问一个普通的页面没有任何区别，因为 HttpClient 已经帮忙发送了 Cookie 信息。下面的例子实现了这样一个访问的过程。

```
public class RenRen {  
    // 配置参数  
    private static String userName = "邮箱地址";  
    private static String password = "密码";  
    private static String redirectURL =  
        "http://blog.renren.com/blog/304317577/449470467"; //要抓取的网址  
  
    // 登录 URL 地址  
    private static String renRenLoginURL = "http://www.renren.com/PLogin.do";  
  
    // 用于取得重定向地址  
    private HttpResponse response;  
    // 在一个会话中用到的 httpclient 对象  
    private DefaultHttpClient httpClient = new DefaultHttpClient();  
  
    //登录到页面  
    private boolean login() {  
        //根据登录页面地址初始化 httpost 对象  
        HttpPost httpost = new HttpPost(renRenLoginURL);  
        //POST 给网站的所有参数  
        List<NameValuePair> nvps = new ArrayList<NameValuePair>();  
    }  
}
```

```
nvps.add(new BasicNameValuePair("origURL", redirectURL));
nvps.add(new BasicNameValuePair("domain", "renren.com"));
nvps.add(new BasicNameValuePair("islogin", "true"));
nvps.add(new BasicNameValuePair("formName", ""));
nvps.add(new BasicNameValuePair("method", ""));
nvps.add(new BasicNameValuePair("submit", "登录"));
nvps.add(new BasicNameValuePair("email", userName));
nvps.add(new BasicNameValuePair("password", password));
try {
    httpost.setEntity(new UrlEncodedFormEntity(nvps, HTTP.UTF_8));
    response = httpclient.execute(httpost);
} catch (Exception e) {
    e.printStackTrace();
    return false;
} finally {
    httpost.abort();
}
return true;
}

//取得重定向地址
private String getRedirectLocation() {
    Header locationHeader = response.getFirstHeader("Location");
    if (locationHeader == null) {
        return null;
    }
    return locationHeader.getValue();
}

//根据重定向地址返回内容
private String getText(String redirectLocation) {
    HttpGet httpget = new HttpGet(redirectLocation);
    // 创建一个响应处理器
    ResponseHandler<String> responseHandler = new BasicResponseHandler();
    String responseBody = "";
    try {
        //取得网页内容
```

```

        responseBody = httpclient.execute(httpget, responseHandler);
    } catch (Exception e) {
        e.printStackTrace();
        responseBody = null;
    } finally {
        httpget.abort();
        httpclient.getConnectionManager().shutdown();//关闭连接
    }
    return responseBody;
}

public void printText() {
    if (login()) {
        String redirectLocation = getRedirectLocation();
        if (redirectLocation != null) {
            System.out.println(getText(redirectLocation));
        }
    }
}

public static void main(String[] args) {
    RenRen renRen = new RenRen();
    renRen.printText();
}
}

```

抓取需要 ASP.NET 表单验证的网页更加复杂，涉及到网页自动生成的一些隐藏字段。可以在 ASP.NET 的页面中的 `<form runat="server">` 控件中放置一个名字叫做 `ViewState` 的隐藏域来定义页面的状态。

源代码可能类似这样：

```

<form name="_ctl0" method="post" action="page.aspx" id="_ctl0">
<input type="hidden" name="__VIEWSTATE"
value="dDwtNTI0ODU5MDE1Ozs+ZBCF2ryjMpeVgUrY2eTj79HNI4Q=" />

.....some code

</form>

```

在配置中使用 `<pages enableEventValidation="true"/>` 或在页面中使用 `<%@ Page EnableEventValidation="true" %>` 将启用事件验证。出于安全目的，此功能验证回发或回调事件的参数是否来源于最初呈现这些事件的服务器控件。此时会在页面中生成一个 `__EVENTVALIDATION` 隐藏字段。所以需要提交 `__VIEWSTATE` 和 `__EVENTVALIDATION` 两个隐藏字段。抓取网页的主要代码如下：

```
String url = "http://www.2552.net/Book/LC/1.aspx";
// 构造 HttpClient 的实例
HttpClient httpClient = new DefaultHttpClient();
HttpPost httpPost = new HttpPost(url);
List<NameValuePair> nvps = new ArrayList<NameValuePair>();
nvps.add(new BasicNameValuePair("__VIEWSTATE", "省略")); // 这里填实际的值
nvps.add(new BasicNameValuePair("__EVENTTARGET", "_ctl0:pager"));
nvps.add(new BasicNameValuePair("__EVENTARGUMENT", "2"));
httpPost.setEntity(new UrlEncodedFormEntity(nvps, HTTP.UTF_8));

ResponseHandler<String> responseHandler = new BasicResponseHandler();
String responseBody = "";
try {
    responseBody = httpClient.execute(httpPost, responseHandler);
} catch (Exception e) {
    e.printStackTrace();
    responseBody = null;
} finally {
    httpPost.abort();
    httpClient.getConnectionManager().shutdown();
}
System.out.println(responseBody);
```

查询 `__VIEWSTATE` 和 `__EVENTVALIDATION` 两个隐藏字段的值。

```
String url = "http://www.925city.cn/cityadmin/StockSearch.aspx";
HttpClient httpClient = new DefaultHttpClient();
HttpPost httpPost = new HttpPost(url);

ResponseHandler<String> responseHandler = new BasicResponseHandler();
String responseBody = "";
try {
    responseBody = httpClient.execute(httpPost, responseHandler);
} catch (Exception e) {
```

```
e.printStackTrace();
responseBody = null;
return;
} finally {
    httpost.abort();
}

//从网页提取值
int state = responseBody.indexOf("id=\"__VIEWSTATE\"");
String viewState = responseBody.substring(state + 24);
viewState = viewState.substring(0, viewState.indexOf("\""));
int c = responseBody.indexOf("id=\"__EVENTVALIDATION\"");
String validation = responseBody.substring(c + 30);
validation = validation.substring(0, validation.indexOf("\""));

System.out.println("VIEWSTATE:"+viewState);
System.out.println("VALIDATION:"+validation);

//查询商品库存情况
httpost = new HttpPost(url);
List<NameValuePair> nvps = new ArrayList<NameValuePair>();
nvps.add(new BasicNameValuePair("__VIEWSTATE", viewState));
nvps.add(new BasicNameValuePair("__EVENTVALIDATION", validation));
nvps.add(new BasicNameValuePair("txtbn", "b11312")); //商品编号
nvps.add(new BasicNameValuePair("btnSearch", "查 询"));

httpost.setEntity(new UrlEncodedFormEntity(nvps, HTTP.ASCII));

try {
    responseBody = httpClient.execute(httpost, responseHandler);
} catch (Exception e) {
    e.printStackTrace();
    responseBody = null;
} finally {
    httpost.abort();
    httpClient.getConnectionManager().shutdown();
}
```



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

2.3.13 抓取 JavaScript 动态页面

很多网页中包含 JavaScript 代码。例如一些网页链接嵌入在 JavaScript 代码中，因此爬虫最好能够识别 JavaScript。虽然 JavaScript 从语义上看来和 Java 非常相近，但实际上 JavaScript 来自一个和 Java 完全不同的编程语言家族。JavaScript 是 Smalltalk 和 Lisp 语言的一个直系后裔。

Mozilla 发布了纯 Java 语言编写的 JavaScript 脚本解释引擎 Rhino(<http://www.mozilla.org/rhino/>)。Rhino 包含所有 JavaScript 1.7 的特征。Rhino 附带一个可以运行 JavaScript 脚本的 JavaScript 外壳。有一个 JavaScript 编译器把 JavaScript 源文件翻译成 Java 的 class 文件。有一个 JavaScript 调试器用于 Rhino 执行脚本。Rhino 可以实现从 Java 对象到动态页面脚本片段常用语言——JavaScript 对象的直接映射，这有利于简化脚本解析环境的构建工作，减少脚本解释引擎与脚本片段在实现语言方面的差异。同时，在脚本解析的过程中，Rhino 对于 JavaScript 对象的操作结果可以通过访问已经本地创建的、与其一一对应的 Java 对象直接获得，示例代码如下：

```
ScriptEngineManager factory = new ScriptEngineManager();
ScriptEngine engine = factory.getEngineByName("JavaScript");

String s = GetJSText.getAllJS().toString(); // 读取 JavaScript 的文本
engine.eval(s); // 加载 JavaScript

// engine.eval(new FileReader("HelloWorld.js")); // 从文件读入 JavaScript 脚本

// Invocable inv = (Invocable) engine;
// Object obj = inv.invokeFunction("getI"); // 调用 JavaScript 的 function getI()
// System.out.println(obj);
```

Rhino 的主要功能是脚本执行时的运行环境管理。运行环境是指用来保存所要执行的脚本中的变量、对象和执行上下文的空间。运行环境中的变量和对象由运行环境内所有的执行上下文共享，即一个执行上下文创建的变量或对象其他上下文也可以访问，运行环境负责处理变量或对象访问时的同步和互斥问题。运行环境和执行上下文是执行脚本语句的场所，因此在应用程序中应首先调用 Rhino 提供的 API(应用程序接口)建立一个运行环境和若干个执行上下文,然后调用相应的 API 建立脚本语言的内置对象。

HTML DOM 中只有 Window 和 Document 对象的方法参数中含有超链接网络地址信息和页面主体内容。因此在进行 HTML DOM 对象本地创建时，将其余对象的属性和方法简单地设置为空(NULL)。

在 Window 和 Document 对象的方法参数中，与超链接网络地址、页面主体内容相关的函数可以分为两类：第一类以 Window 对象的 open 方法为代表，open 方法的参数是动态

页面中的超链接网络地址，参数类型是 JavaScript 语言内置 String 类型。在引擎外创建该类方法时，声明该方法的行为是把参数，即超链接网络地址送入信息采集环节的待获取 URL 队列中；第二类以 Document 对象的 write 方法为代表，write 方法的参数(同为 JavaScript 语言内置 String 类型)是一段表达脚本片段最终在浏览器中呈现内容的静态网页源文件。类似于常见的静态网页，在作为 write 方法参数的网页源文件中，超链接网络地址和页面主体内容被分别以 URL 和文本信息方式直接嵌入 HTML 标记中。在引擎外创建该类方法时，声明该方法的行为是把参数，即静态网页源文件写入位于本地特定的文件中。

由于 Rhino 能够自动在 Java 对象和 JavaScript 对象之间根据“对象名称一致性”的原则实现一一对应。因此，当 Rhino 在执行脚本片段中的“Window.open()”与“Document.write()”时，实际上是分别调用在 Java 语言作用域中定义的，与这两个方法同名的 Java 函数，执行函数体中关于函数行为的描述。

在完成脚本片段提取和 HTML DOM 本地创建后，就可以调用 Rhino 提取 JavaScript 动态页面中的超链接网络地址及页面主体内容。当遇到脚本片段中的 HTML DOM 时，Rhino 根据引擎外创建的同名函数体中的行为描述执行相应动作。

根据 HTML DOM 本地创建结果，Rhino 将脚本片段中 Window 对象方法 open 参数体现的超链接网络地址直接送入信息采集环节的待获取 URL 队列中，实现动态页面内含超链接的递归获取功能。与其类似，把脚本片段中 Document 对象方法 write 参数所指向的静态网页源文件写入本地特定的文件中。在此基础上，使用传统的 HTML 标记识别方法，提取得到的静态网页源文件中的超链接网络地址与页面主体内容，将前者送入信息采集环节的 URL 队列，把后者交信息采集环节统一实现数据存储。下面的网页按钮上存在的 JavaScript 形式的跳转链接：

```
<html>
<head>
<script language="javascript">
function redirectNow(){
window.location.href="/AboutUs.htm";
}
</script>
</head>
<body>
<input type="button" value="Redirect using location.href" onclick="redirectNow()">
</body>
</html>
```

提取这个跳转链接可以采用 HtmlUnit (<http://htmlunit.sourceforge.net/>)实现。HtmlUnit 底层也是调用了 Rhino，但是绕过了 Rhino 的一些错误。主要代码如下：

```
public static void main(String[] args) throws Exception {
    WebClient webClient = new WebClient();
```

```
String url = "http://www.lietu.com/lab/location.html";
HtmlPage page = (HtmlPage) webClient.getPage(url);
DomNodeList nodeList = page.getChildNodes();
for (Object node : nodeList) {
    HTMLElement n = (HTMLElement) node;
    extract(n);
}

public static void extract(HTMLElement node)
    throws IOException {
    if ("input".equals(node.getNodeName())) {
        HtmlInput inputHtml = (HtmlInput) node;
        HtmlPage newPage = inputHtml.click();
        System.out.println(newPage.toString());//按钮跳转的新页面
    }

    DomNode childNode = node.getFirstChild();

    if (childNode instanceof HTMLElement) {
        HTMLElement child = (HTMLElement) childNode;
        while (child != null) {
            // 递归调用 extract 方法
            if(child == childNode) {
                extract(child);
            }
            childNode = childNode.getNextSibling();
            if (childNode == null) {
                break;
            }
            if (childNode instanceof HTMLElement) {
                child = (HTMLElement) childNode;
            }
        }
    }
}
```

但是 HtmlUnit 对于 JavaScript 框架 jQuery 的支持不太好。

另外一种方式是用 selenium(<http://seleniumhq.org>)抓取 JavaScript 动态信息。有广泛的用户群和活跃的开发团队，Google 公司资助并且使用 Selenium。selenium 的核心代码通过 JavaScript 完成，可以运行在 FireFox 或 IE 等浏览器中。Watir(<http://watir.com/>)是一个控制浏览器行为的类似项目。虽然可以利用 jrex 提取渲染后的 html 文件，但推荐用 c#来实现。

2.3.14 抓取即时信息

信息的即时性往往很重要。例如，刚出现的经济信息可能在几秒钟以后就会影响相关股价。

为了加快获取信息的速度，可以先只抓取标题及 URL 地址。对于首页，通过 HTTP 协议的 HEAD 命令判断页面是否已经更新。在不同的时间段，用不同的频率来检查网页更新。

很多行业网站的首页是按板块组织的，可以按板块提取 URL 地址。有的板块全部是新的，也就是说，新的信息从这个板块溢出了。对于从板块溢出的地址，再从该版块对应的索引页补全信息。查找一个板块对应的索引页的方法是：查找该板块的“更多”链接对应的 URL 地址。

为了节省带宽，抓取到某个页面时，如果已经没有新的信息，可以不再继续往下检查这个页面的后续网页是否有更新。许多 Web 服务器具有发送压缩数据的能力，这可以将网络线路上传输的大量数据消减 60% 以上。

假设一个网站的首页有 90K 字节*8 位=720K 位，每秒需要同步 200 个网站，则需要大约 140M 带宽。

2.3.15 抓取暗网

互联网就像是深不可测的大海。在大海中，很多鱼深藏在水下。很多有用的信息隐藏在互联网中。

暗网的表现形式一般是：前台是一个表单来获取，提交后返回一个列表形式的搜索结果页，它们是由暗网后台数据库动态产生的。

为了用程序提交表单，这里用到 Web 操作录制工具 badboy(<http://www.badboy.com.au/>)和 Web 应用测试工具 JMeter(<http://jakarta.apache.org/jmeter/>)。使用 badboy 录制出登录时向网站发出的请求参数。然后导出成 JMeter 文件，在 JMeter 中就可以看到登录时向网站发送的参数列表和相应的值。

搜索引擎本身也可以看作是一个暗网。搜索结果页包含了指向详细内容页的链接。下面讨论抓取搜索引擎中的内容。例如，有一个乡镇词表。“郭河”是一个小地名，但不知道它是一个镇还是一个村。把这个词提交给搜索引擎，搜索引擎返回的结果中可能包含了“郭河镇”这样的结果，这样就知道了“郭河”可能是一个镇。例如，把公司名称作为查询词提交

给 Google 并从返回结果中提取联系方式：

```
String companyName = "SILK INDIA";
String searchWord = URLEncoder.encode(companyName, "utf-8");//返回编码后的查询词
String searchURL = "http://www.google.com/search?q="+searchWord;
String content = RetrivePage.downloadPage(searchURL);
```

如果要提取公司的网址，可以提交搜索“website:www 公司名”给 Google。但是同一个 IP 每天 Google 最多只返回 1000 次查询。

以搜农网(www.sounong.net)为例，按照作物分类，将作物名称，供求类型等信息组装出查询 URL 地址，获得该 URL 的返回页面内容，匹配出当前查询的作物信息共有多少条，然后计算出页面数量，这样组装 URL 时，动态替换页码进行翻页。代码如下：

```
final static String ORDER_FORMAT =
"http://www.sounong.net/newsounong/gq.jsp?q=%s&flag=show&adr=&sa%2F=%B9%A9%
C7%F3%CB%D1%CB%F7&flt=1&type=%C7%F3%B9%BA&sort=2";
String searchWord = URLEncoder.encode(keyword, "GBK");
String url = ORDER_FORMAT.replace("%s", searchWord);//执行参数替换
```

此外还可以抓取手机号码和所属地区的对应关系。提交一个手机号码到网站，返回所属地区。一般来说，前 7 位相同的手机都是属于同一个地区，所以可以根据手机号码的前 7 位来判断地区。

对不同的语言，可以考虑抓取当地的搜索引擎网站，例如 Google 的葡萄牙文网站是 <http://www.google.pt>。

对于暗网内容，有时候可以直接循环编码。例如，抓取酒店信息：

```
http://hotels.ctrip.com/hotel/58.html
http://hotels.ctrip.com/hotel/5829.html
http://hotels.ctrip.com/hotel/55829.html
```

直接循环遍历酒店编码。

2.3.16 信息过滤

有时候可能需要按一个关键词列表来过滤信息，例如过滤黄色或其它非法信息。

调用 indexOf 方法来查找关键词集合看起来效率不高，Aho-Corasick 算法可以用来在文本中搜索多个关键词。当有一个关键词的集合，想发现文本中的所有出现关键词的位置，或者检查是否有关键词集合中的任何关键词出现在文本中时，这个实现代码是有用的。有很多不经常改变的关键词时，可以使用 Aho-Corasick 算法。其运行时间是输入文本的长度加上匹配关键词数目的线性和。Aho-Corasick 算法根据两个发明人 Alfred V. Aho 和 Margaret J.

Corasick 命名。

Aho-Corasick 把所有要查找的关键词构建一个 Trie 树。Trie 树包含后缀树一样的链接集合，从代表字符串的每个节点(例如 abc)到最长的后缀(例如如果词典中存在 bc，则链接到 bc，否则如果词典中存在 c，则链接到 c，否则链接到根节点)。每个节点的失败匹配属性(failure)存储这个最长后缀节点。也就是说，还包含一个从每个节点到存在于词典中的最长后缀节点链接。

假设词典中存在词：a, ab, bc, bca, c, caa。6 个词组成如下的 Trie 树结构如图 2-4：

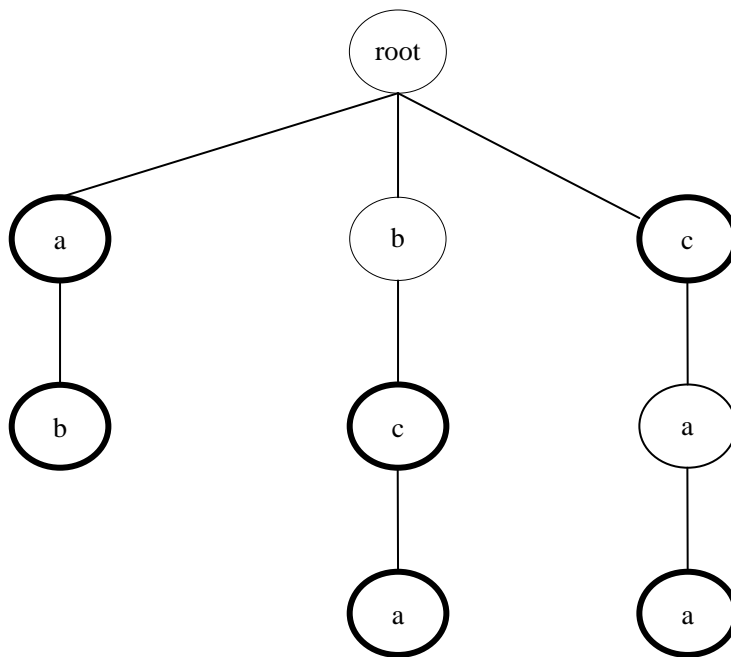


图 2-4 Trie 树结构

由指定的字典构造的 Aho-Corasick 算法的数据结构如表 2-1，表里面的每一行代表树中的一个节点，表里的列路径用从根到节点的唯一字符序列说明。

路径	是否在字典里	后缀链接	词典后缀链接
()	否		
(a)	是	()	
(ab)	是	(b)	
(b)	否	()	

(bc)	是	(c)	(c)
(bca)	是	(ca)	(a)
(c)	是	()	
(ca)	否	(a)	(a)
(caa)	是	(a)	(a)

表 2-1 Trie 树结构

每取一个待匹配的字符后，当前的节点通过寻找它的孩子来匹配，如果孩子不存在，也就是匹配失败，则试图匹配该节点后缀的孩子，如果这样也没有匹配上，则匹配该节点的后缀的后缀的孩子，最后如果什么也没有匹配上，就在根节点结束。

分析输入字符串“abccab”的匹配过程如表 2-2：

节点	剩余的字符串	输出：结束位置	跳转	输出
()	abccab		从根节点开始	
(a)	bccab	a:1	从根节点()转移到孩子节点 (a)	当前节点
(ab)	ccab	ab:2	节点(a) 转移到孩子节点 (ab)	当前节点
(bc)	cab	bc:3, c:3	节点(ab) 转到后缀 (b) ，再跳转到孩子节点 (bc)	当前节点，词典后缀节点
(c)	ab	c:4	节点(bc) 转到后缀节点 (c) ，再跳转到根节点()，再跳转到孩子节点(c)	当前节点
(ca)	b	a:5	节点(c) 跳转到孩子节点 (ca)	词典后缀节点
(ab)		ab:6	节点(ca) 跳转到后缀节点 (a)，再跳转到孩子节点(ab)	当前节点

表 2-2 Aho-Corasick 算法匹配过程

首先定义 Trie 树结点类:

```
private final class TreeNode {
    private char _char;//节点代表的字符
    private TreeNode _parent;//该节点的父节点
    private TreeNode _failure;//匹配失败后跳转的节点
    private ArrayList<String> _results;//存储模式串的数组变量
    private TreeNode[] _transitionsAr;
    //存储孩子节点的哈希表
    private Hashtable<Character, TreeNode> _transHash;
    public TreeNode(TreeNode parent, char c) {
        _char = c;
        _parent = parent;
        _results = new ArrayList<String>();//存储所有没有重复的模式串的数组
        _transitionsAr = new TreeNode[] {};
        _transHash = new Hashtable<Character, TreeNode>();
    }

    //将模式串中不在_results 中的模式串添加进来
    public void addResult(String result) {
        if (_results.contains(result))//如果已经包含该模式则不增加到模式串中
            return;
        _results.add(result);
    }

    public void addTransition(TreeNode node) {//增加一个孩子节点
        _transHash.put(node._char, node);
        TreeNode[] ar = new TreeNode[_transHash.size()];
        Iterator<TreeNode> it = _transHash.values().iterator();
        for (int i = 0; i < ar.length; i++) {
            if (it.hasNext()) {
                ar[i] = it.next();
            }
        }
        _transitionsAr = ar;
    }

    public TreeNode getTransition(char c) {
```



```

        return _transHash.get(c);
    }

    public boolean containsTransition(char c) {
        return getTransition(c) != null;
    }

    public char getChar() {
        return _char;
    }

    public TreeNode parent() {
        return _parent;
    }

    public TreeNode failure(TreeNode value) {
        _failure = value;
        return _failure;
    }

    public TreeNode[] transitions() {
        return _transitionsAr;
    }

    public ArrayList<String> results() {
        return _results;
    }
}

```

Trie 树的构建过程由构建树本身和增加失败匹配属性两步构成：

```

public StringSearch(String[] keywords) {
    buildTree(keywords); //构建树
    addFailure(); //增加失败匹配属性
}

```

构建树的过程：

```

void buildTree() {
    _root = new TreeNode(null, ' ');
}

```

```

for (String p : _keywords) {
    TreeNode nd = _root;

    for(char c : p.toCharArray()){
        TreeNode ndNew = null;
        for (TreeNode trans : nd.transitions())
            if (trans.getChar() == c) {
                ndNew = trans;
                break;
            }

        if (ndNew == null) {
            ndNew = new TreeNode(nd, c);
            nd.addTransition(ndNew);
        }
        nd = ndNew;
    }
    nd.addResult(p);
}
}

```

增加失败匹配属性的过程:

```

private void addFailure(){
    //所有词的第 n 个字节点的集合，n 从 2 开始
    ArrayList<TreeNode> nodes = new ArrayList<TreeNode>();

    //所有词的第 2 个字节点的集合
    for (TreeNode nd : _root.transitions()) {
        nd.failure(_root);
        for (TreeNode trans : nd.transitions()){
            nodes.add(trans);
        }
    }

    //所有词的第 n+1 个字节点的集合
    while (nodes.size() != 0) {

```

```

ArrayList<TreeNode> newNodes = new ArrayList<TreeNode>();
for (TreeNode nd : nodes) {
    TreeNode r = nd.parent()._failure;
    char c = nd.getChar();

    //如果在父节点的失败节点的孩子节点中没有同样字符结尾的
    //，则继续在失败节点的失败节点中找
    while (r != null && !r.containsTransition(c))
        r = r._failure;
    if (r == null)
        nd._failure = _root;
    else {
        nd._failure = r.getTransition(c);
        for (String result : nd._failure.results()) {
            nd.addResult(result);
        }
    }

    for (TreeNode child : nd.transitions()){
        newNodes.add(child);
    }
}
nodes = newNodes;
}
_root._failure = _root;
}

```

为了加深理解，可以打印生成的 Trie 树。遍历二叉树的方法有：先序遍历、后序遍历、中序遍历。深度遍历、广度遍历是针对普通树。因为这是一棵普通的树，所以采用深度遍历的方式打印树的结构：

//深度遍历的递归函数

```

public void depthSearch(TreeNode node, StringBuilder ret,int deapth) {
    if (node != null) {
        for (int i = 0; i < deapth; i++)
            ret.append("| ");
        ret.append("|——");
        ret.append(node._char + "\n");
        for (TreeNode child : node.transitions()) {

```

```

        int childDeapth = deapth + 1;//计算深度并赋值
        depthSearch(child, ret, childDeapth);
    }
}
}

```

//打印树节点

```

public String toString() {
    StringBuilder ret = new StringBuilder();
    ret.append("打印树节点: \n");
    int deapth = 0;
    depthSearch(_root, ret, deapth);
    return ret.toString();
}

```

从输入文本查找关键词集合的过程:

```

public StringSearchResult[] findAll(String text) {
    ArrayList<StringSearchResult> ret = new ArrayList<StringSearchResult>();
    TreeNode ptr = _root;
    int index = 0;

    while (index < text.length()) {
        TreeNode trans = null;
        while (trans == null) {
            trans = ptr.getTransition(text.charAt(index));

            if (ptr == _root)
                break;
            if (trans == null) {
                ptr = ptr._failure;
            }
        }
        if (trans != null)
            ptr = trans;
        //增加找到的每一个词到结果中
        for (String found : ptr.results())
            ret.add(new StringSearchResult(index - found.length() + 1, found));
    }
}

```

```

        index++;
    }

    return ret.toArray(new StringSearchResult[ret.size()]);
}

```

2.4 URL 地址查新

在科技论文发表时，为了避免重复研究和抄袭，需要到专门的科技情报所做论文查新。为了避免重复抓取，URL 地址也需要查新。判断解析出的 URL 是否已经遍历过也叫做 URLSeen 测试。URLSeen 测试对爬虫性能有重要的影响。本节介绍两种实现快速 URLSeen 测试的方法。

在介绍爬虫架构的时候，我们讲解了 Frontier 组件的作用。它作为一个基础的组件，为爬虫提供 URL。因此，在 Frontier 中有一个数据结构来存储 URL。在一些小的爬虫程序中，使用内存队列(ArrayList、HashMap 或 Queue)或者优先级队列来存储 URL，但内存是有限的。在通常商业应用中，URL 地址数据量非常大。早期的爬虫经常把 URL 地址放在数据库表中，但数据库对于这种简单的结构化存储来说效率太低。因此，考虑使用内存数据库来存储。BerkeleyDB 就是一种常用的内存数据库。

2.4.1 BerkeleyDB

BerkeleyDB(<http://www.oracle.com/database/berkeley-db/index.html>) 是一个嵌入式数据库。这里的嵌入式和嵌入式系统无关，嵌入式数据库的意思是不需要通过 JDBC 访问数据库，也不单独启动进程来管理数据。BerkeleyDB 中的一个数据库只能存储一些键/值对，也就是键和值两列。BerkeleyDB 底层实现采用 B 树，可以把它看成可以存储大量数据的 HashMap。BerkeleyDB 的 c++版本首先出现，然后在此基础上又实现了 Java 本地版本。但本书中只用到 Java 版本。可以用它来存储已经抓取过的 URL 地址。如果使用 Berkeley DB Java Edition 4.0.103，在 Eclipse 项目中只需要导入一个 jar 包——je-4.0.103.jar。BerkeleyDB 的数据库需要在一个环境类的实例中打开。所以首先要实例化一个环境类：

```

//创建一个环境配置对象
EnvironmentConfig envConfig = new EnvironmentConfig();
envConfig.setTransactional(false); //不需要事务功能
envConfig.setAllowCreate(true); //允许创建新的数据库文件
//在路径 envDir 下创建数据库环境
Environment exampleEnv = new Environment(envDir, envConfig);

```

释放环境变量的方法：

```

exampleEnv.sync(); //同步内存中的数据到硬盘
exampleEnv.close(); //关闭环境变量

```

创建数据库，键是字符串，值是一个叫做 `NewsSource` 的类：

```
String databaseName= "ToDoTaskList.db";
//创建一个数据库配置对象
DatabaseConfig dbConfig = new DatabaseConfig();
dbConfig.setAllowCreate(true); //允许创建新的数据库文件
dbConfig.setTransactional(false); //不需要事务功能

// 打开用来存储类信息的数据库
dbConfig.setSortedDuplicates(false);
//用来存储类信息的数据库不要求能够存储重复的关键字
Database myClassDb = exampleEnv.openDatabase(null, "classDb", dbConfig);
//初始化用来存储序列化对象的 catalog 类
catalog = new StoredClassCatalog(myClassDb);
TupleBinding keyBinding = TupleBinding.getPrimitiveBinding(String.class);
//把值作为对象的序列化方式存储
SerialBinding valueBinding = new SerialBinding(catalog, NewsSource.class);
store = exampleEnv.openDatabase(null, databaseName, dbConfig);
store.close();//关闭存储数据库
myClassDb.close();//关闭元数据库
```

建立数据的映射：

```
// 创建数据存储的映射视图
this.map = new StoredSortedMap(store, keyBinding, valueBinding, true);
```

然后可以像操作普通的散列表一样，对 `StoredSortedMap` 用 `put` 方法写入数据，`get` 方法读出数据，也通过 `containsKey` 方法判断某个关键字是否存在，还可以通过迭代器遍历访问 `StoredSortedMap` 中的元素。`StoredSortedMap` 和 `HashMap` 的区别是，`HashMap` 只能把所有数据存储在内存，而 `StoredSortedMap` 则会通过内外存交换支持更多的数据存取。实现 `URLSeen` 测试的完整的例子如下：

```
String dir = "./db/";
EnvironmentConfig envConfig = new EnvironmentConfig();
envConfig.setTransactional(false);
envConfig.setAllowCreate(true);

Environment env = new Environment(new File(dir), envConfig);

//使用一个通用的数据库配置
DatabaseConfig dbConfig = new DatabaseConfig();
```

```

dbConfig.setTransactional(false);

dbConfig.setAllowCreate(true);
//如果有序列化绑定则需要类别数据库
Database catalogDb = env.openDatabase(null, "catalog", dbConfig);
ClassCatalog catalog = new StoredClassCatalog(catalogDb);

//关键字数据类型是字符串
TupleBinding<String> keyBinding = TupleBinding.getPrimitiveBinding(String.class);

//值数据类型也是字符串
SerialBinding<String> dataBinding = new SerialBinding<String>(catalog, String.class);

Database db = env.openDatabase(null, "url", dbConfig);

//创建一个映射
SortedMap<String,String> map =
    new StoredSortedMap<String, String> (db, keyBinding, dataBinding, true);
//把抓取过的 URL 地址作为关键字放入映射
String url = "http://www.lietu.com";
map.put(url, null);
if(map.containsKey(url)){
    System.out.println("已抓取");
}

```

为了能正确处理大量数据，需要增加 Java 虚拟机运行的内存使用量，否则会出现内存溢出的错误。例如增加最大内存用量到 800M，可以使用虚拟机参数“-Xmx800m”。

2.4.2 布隆过滤器

判断 URL 地址是否已经抓取过还可以借助于布隆过滤器(Bloom Filter)。布隆过滤器的实现方法是：利用内存中的一个长度是 m 的位数组 B ，对其中所有位都置 0，如图 2-5。

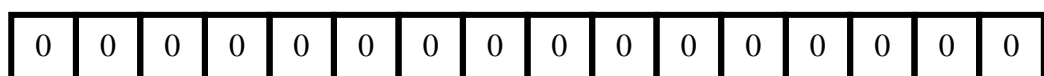


图 2-5 位数组 B 的初始状态

然后对每个遍历过的 URL 根据 k 个不同的散列函数执行散列，每次散列的结果都是不大于 m 的一个整数 a 。根据散列得到的数在位数组 B 对应的位上置 1，也就是让 $B[a]=1$ 。图 2-6 显示了放入 3 个 URL 后位数组 B 的状态，这里 $k=3$ 。

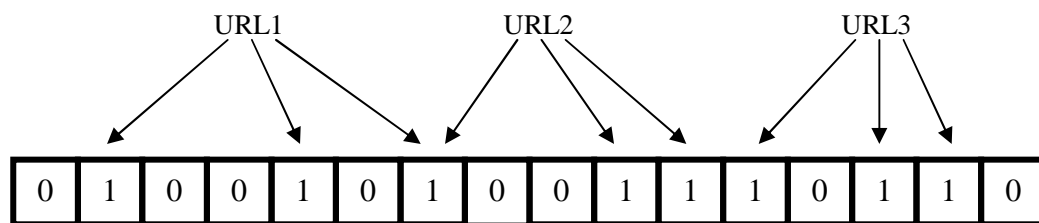


图 2-6 放入数据后位数组 B 的状态

每次插入一个 URL，也执行 k 次散列，只有当全部位都已经置 1 了才认为这个 URL 已经遍历过。如下是使用布隆过滤器(<http://code.google.com/p/java-bloomfilter/>)的一个例子：

```
// 创建一个 100 位的布隆过滤器，优化成包含 4 个项目
BloomFilter<String> urlSeen = new BloomFilter<String>(100, 4);
// 增加内容到布隆过滤器
urlSeen.add("www.lietu.com");
urlSeen.add("www.sina.com");
urlSeen.add("www.qq.com");
urlSeen.add("www.sohu.com");
// 测试布隆过滤器是否记得这个项目
if (urlSeen.contains("www.lietu.com")) {
    System.out.println("已经存在的概率 "
        + (1 - urlSeen.expectedFalsePositiveProbability()));
} else {
    System.out.println("一定不存在");
}
```

布隆过滤器如果返回不包含某个项目，那肯定就是没往里面增加过这个项目，如果返回包含某个项目，但其实可能没有增加过这个项目，所以有误判的可能。对爬虫来说，使用布隆过滤器的后果是可能导致漏抓网页。如果想知道需要使用多少位才能降低错误概率，可以从表 2-1 的存储项目和位数比率估计布隆过滤器的误判率。

表 2-1 布隆过滤器误判率表

比率(items:bits)	误判率
1:1	0.63212055882856

比率(items:bits)	误判率
1:2	0.39957640089373
1:4	0.14689159766038
1:8	0.02157714146322
1:16	0.00046557303372
1:32	0.00000021167340
1:64	0.00000000000004

为每个 URL 分配两个字节就可以达到千分之几的冲突。例如一个比较保守的实现，为每个 URL 分配了 4 个字节，项目和位数比是 1:32，误判率是 0.00000021167340。对于 5000 万数量级的 URL，布隆过滤器只占用了 200M 的空间，并且排重速度超快，一遍下来不到两分钟。

SimpleBloomFilter 中实现的把对象映射到位集合的方法：

```
public void add(E element) throws UnsupportedOperationException {
    long hash;
    String valString = element.toString();
    for (int x = 0; x < k; x++) {
        hash = createHash(valString + Integer.toString(x));
        hash = hash % (long)bitSetSize;
        bitset.set(Math.abs((int)hash), true);
    }
}
```

这个实现方法计算了 k 个相互独立的散列值，因此误判率较低。

为了支持重启动后运行，把布隆过滤器的状态保存到文件中：

```
public void saveBit(String filename) {
    File file = new File(filename);
    ObjectOutputStream oos =
        new ObjectOutputStream(new FileOutputStream(file, false));
    oos.writeObject(bitSet);
    oos.flush();
    oos.close();
}
```

如下的代码把布隆过滤器的状态从先前保存的文件中读出来：

```
public void readBit(String filename) {
    File file = new File(filename);
    if (!file.exists()) {
        return;
    }
    bitSet.clear();
    ObjectInputStream ois = new ObjectInputStream(new FileInputStream(file));
    bitSet = (BitSet)ois.readObject();
    ois.close();
}
```

2.5 增量抓取

专门的库保存一个网站的所有目录页的首页。看首页中是否包含新的商品信息。如果目录页全部是新的商品信息，则抓取下一页，否则不抓取下一页。

考虑用 `TreeSet` 保存同一个栏目下的所有目录页。例如，`TreeSet<IndexPage>` 保存目录页。到一定页面编号以后的页面就不用抓取了。`TreeSet` 中的元素要实现 `Comparable` 接口。

```
public class IndexPage implements Comparable<IndexPage> {
    String url; //目录页的 URL 地址
    int pageNo; //页面编号，有翻页参数的可以直接用翻页参数
    public int compareTo(IndexPage o) {
        return pageNo - o.pageNo;
    }
}
```

2.6 并行抓取

单机并行抓取往往采用多线程同步 IO 或者单线程异步 IO。为了同时抓取多个网站的信息，可以考虑并行抓取。

在 Java 中，为了爬虫稳定性，也可以用多线程来实现爬虫。一般情况下，爬虫程序需要能在后台长期稳定运行。下载网页时，经常会出现异常。有些异常无法捕获，导致爬虫程序退出。为了主程序稳定，可以把下载程序放在子线程执行，这样即使子线程因为异常退出了，但是主线程并不会退出。测试代码如下：

```
public class MyThread extends Thread{
    public void run() {
```

```
        System.out.println("Throwing in " +  
                            "MyThread");  
        throw new RuntimeException();  
    }  
}  
  
public class ThreadTest {  
    public static void main(String[] args) {  
        MyThread t = new MyThread();  
        t.start();  
        try {  
            Thread.sleep(2000);  
        }  
        catch (Exception x) {  
            System.out.println("Caught it");  
        }  
        System.out.println("Exiting main");  
    }  
}
```

2.6.1 多线程爬虫

利用多线程进行抓取是当前搜索引擎中普遍采用的架构模式,一个多线程爬虫架构如图 2-5 所示:

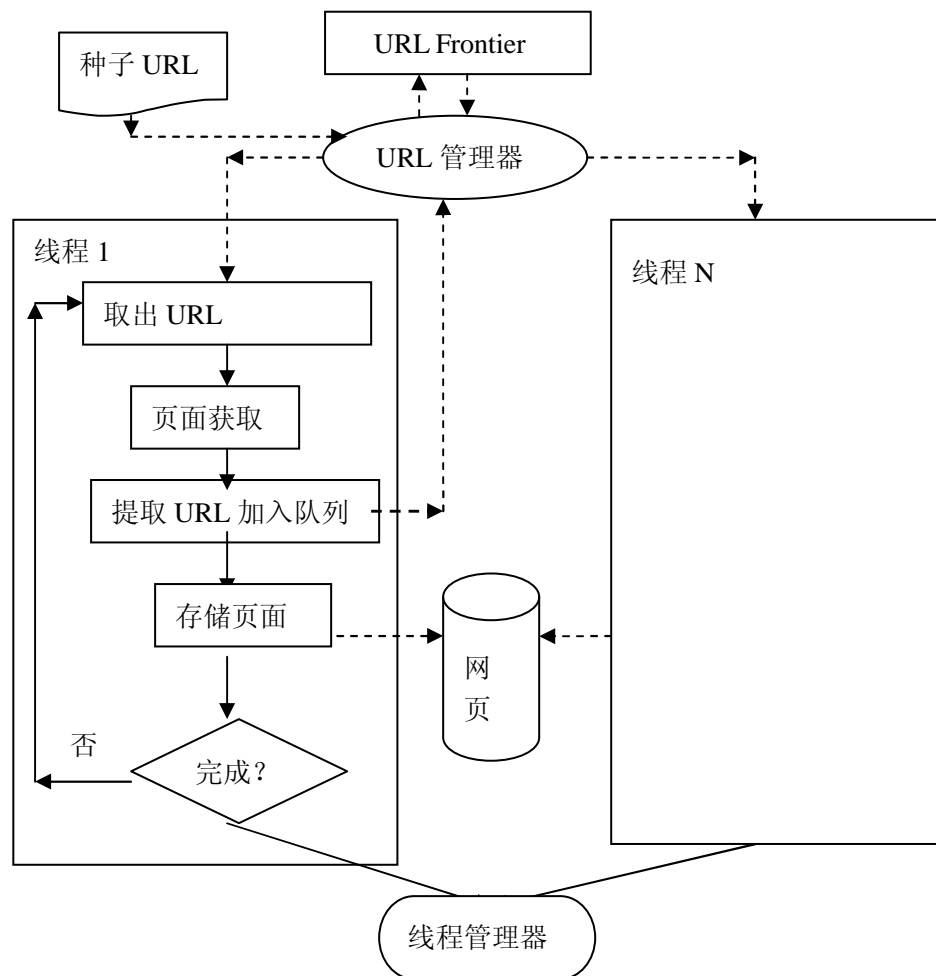


图 2-5 并行爬虫架构

JDK1.5 以后的版本提供了一个轻量级线程池 `ThreadPool`。可以使用线程池执行一组任务，最简单的任务不返回值给主调线程。要返回值的任务可以实现 `Callable<T>` 接口，线程池执行任务并通过 `Future<T>` 的实例获取返回值。

`Callable` 是类似于 `Runnable` 的接口，在其中定义可以在线程池中执行的任务。`Future` 表示异步计算的结果，它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。`Future` 的 `cancel` 方法取消任务的执行，`cancel` 方法有一个布尔参数，参数为 `true` 表示立即中断任务的执行，参数为 `false` 表示允许正在运行的任务运行完成。`Future` 的 `get` 方法等待计算完成，获取计算结果。

下面使用 `ThreadPool` 实现并行下载网页。在继承 `Callable` 方法的任务类中下载网页：

```
public class DownloadCall implements Callable<String> {
    private URL url;    //待下载的 url

    public DownloadCall(URL u) {
```

```

        url = u;
    }

    @Override
    public String call() throws Exception {
        String content = null;
        //下载网页
        return content;
    }
}

```

主线程类创建 `ThreadPool` 并执行下载任务的实现如下：

```

int threads = 4; //并发线程数量
ExecutorService es = Executors.newFixedThreadPool(threads); //创建线程池

Set<Future<String>> set = new HashSet<Future<String>>();

for (final URL url : urls) {
    DownloadCall task = new DownloadCall(url);
    Future<String[]> future = es.submit(task); //提交下载任务
    set.add(future);
}

//通过 future 对象取得结果，这一步不能省略，如果不需要返回值可以调用 Runnable
for (Future<String> future : set) {
    String content = future.get();
    //处理下载网页的结果
}

```

采用线程池可以充分利用多核 CPU 的计算能力，并且简化了多线程的实现。

2.6.2 垂直搜索的多线程爬虫

垂直行业网站一般不是太多，为了及时采集每个网站，可以每个线程抓取一个指定的网站，然后把抓取的信息直接写入到索引。对 `Lucene` 来说，同一个时刻只能由一个线程写索引，而可以多个抓取线程同时并行下载。套用线程的生产者和消费者模型，这里索引线程是消费者，抓取线程是生产者。为了简化实现，以传递整数为例。

使用 `BlockingQueue` 存储待索引的文档。`take()`方法会让调用的线程等待新的元素。

```
public class Indexer implements Runnable { //索引类
    private BlockingQueue<Integer> dataQueue;
    public Indexer(BlockingQueue<Integer> dataQueue) {
        this.dataQueue = dataQueue;
    }
    @Override
    public void run() {
        Integer i;
        while (!Thread.interrupted()){
            try {
                i = dataQueue.take();
                System.out.println("索引: " + i);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Spider implements Runnable { //爬虫类
    private BlockingQueue<Integer> dataQueue;
    private static int i = 0;
    public Spider(BlockingQueue<Integer> dataQueue) {
        this.dataQueue = dataQueue;
    }

    @Override
    public void run() {
        while (!Thread.interrupted()){
            try {
                dataQueue.add(new Integer(++i));
                System.out.println("抓取: " + i);
                //可以把每个线程休眠的时间根据抓取的线程数量动态调整,
                //如果同时抓取的线程数量多, 则延长线程休眠的时间。
                TimeUnit.MILLISECONDS.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

    }
  }
}
}

```

测试方法如下：

```

public static void main(String[] args) {
    BlockingQueue<Integer> dataQueue = new LinkedBlockingQueue<Integer>();

    Thread pt = new Thread(new Spider(dataQueue)); //爬虫线程
    pt.start();

    Thread ct = new Thread(new Indexer(dataQueue)); //索引线程
    ct.start();
}

```

这里的爬虫线程可以有多个，而只有一个索引线程把抓取下来的数据写入索引。如果需要结束，可以调用 `thread.interrupt()` 方法终止索引线程。

`take` 方法会等待，而 `poll` 方法则会立即返回。下面是采用 `poll` 方法的实现。

```

public void run() {
    while (keepRunning || !documents.isEmpty()) {
        Document d = (Document) documents.poll();
        try {
            if (d != null) {
                writer.addDocument(d);
            } else {
                //队列是空的，所以等待
                Thread.sleep(sleepMilisecondOnEmpty);
            }
        } catch (ClassCastException e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        } catch (InterruptedException e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        } catch (CorruptIndexException e) {
            e.printStackTrace();
        }
    }
}

```

```

        throw new RuntimeException(e);
    } catch (IOException e) {
        e.printStackTrace();
        throw new RuntimeException(e);
    }
}

isRunning = false;
}

```

Logback 记录爬虫运行的日志。

SiftingAppender 它可以用来分割日志文件根据任何一个给定的运行参数。如，**SiftingAppender** 能够根据爬虫的线程编号区别日志事件，然后每个线程会有一个日志文件。

SiftingAppender 包含和管理多个根据判别值动态创建的 **appender**。**SiftingAppender** 缺省情况下使用 **MDC** 键/值对作为判别器。在配置 **logback** 后，**SiftExample** 应用程序记录一条日志表明应用程序已经启动。然后把 **MDC** 键"spiderid"设置成为"lietu"，并且记录一条消息。代码如下：

```

logger.debug("爬虫启动");
MDC.put("spiderid", "lietu");
logger.debug("开始抓取 lietu");

```

配置文件如下：

```

<configuration>

  <appender name="SIFT" class="ch.qos.logback.classic.sift.SiftingAppender">
    <!-- in the absence of the class attribute, it is assumed that the
         desired discriminator type is
         ch.qos.logback.classic.sift.MDCBasedDiscriminator -->
    <discriminator>
      <key>spiderid</key>
      <defaultValue>unknown</defaultValue>
    </discriminator>
    <sift>
      <appender name="FILE-${spiderid}" class="ch.qos.logback.core.FileAppender">
        <file>${spiderid}.log</file>
        <append>false</append>
        <layout class="ch.qos.logback.classic.PatternLayout">

```



```

        <pattern>%d [%thread] %level %mdc %logger{35} - %msg%n</pattern>
    </layout>
</appender>

</sift>
</appender>

<root level="DEBUG">
    <appender-ref ref="SIFT" />
</root>
</configuration>

```

2.6.3 异步 IO

对单线程并行抓取来说，异步(asynchronous)I/O 是很重要的基本功能。异步 I/O 模型大体上可以分为两种，反应式(Reactive)模型和前摄式(Proactive)模型。传统的 select/epoll/kqueue 模型，以及 Java NIO 模型，都是典型的反应式模型，即应用代码对 I/O 描述符进行注册，然后等待 I/O 事件。当某个或某些 I/O 描述符所对应的 I/O 设备上产生 I/O 事件（可读、可写、异常等）时，系统将发出通知，于是应用便有机会进行 I/O 操作并避免阻塞。由于在反应式模型中应用代码需要根据相应的事件类型采取不同的动作，最常见的结构便是嵌套的 if {...} else {...} 或 switch，并常常需要结合状态机来完成复杂的逻辑。前摄式模型则恰恰相反。在前摄式模型中，应用代码主动地投递异步操作而不管 I/O 设备当前是否可读或可写。投递的异步 I/O 操作被系统接管，应用代码也并不阻塞在该操作上，而是指定一个回调函数并继续自己的应用逻辑。当该异步操作完成时，系统将发起通知并调用应用代码指定的回调函数。在前摄式模型中，程序逻辑由各个回调函数串联起来：异步操作 A 的回调发起异步操作 B，B 的回调再发起异步操作 C，以此往复。

Java6 版本开始引入的 NIO 包，通过 Selectors 提供了非阻塞式的 IO。Java7 附带的 NIO.2 文件系统中包含了异步 I/O 支持。也可以使用框架实现异步 I/O，例如：

- Mina(<http://mina.apache.org/>)为开发高性能和高可用性的网络应用程序提供了非常便利的框架。当前发行的 MINA 版本支持基于 Java 的 NIO 技术的 TCP/UDP 应用程序开发。MINA 是借由 Java 的 NIO 的反应式实现的模拟前摄式模型。
- Grizzly 是 Web 服务器 GlassFish 的 I/O 核心。Grizzly 还是一个独立于 GlassFish 的框架结构，可以单独用来扩展和构建自己的服务器软件。Grizzly 通过队列模型提供异步读/写。
- Netty(<http://www.jboss.org/netty>)是一个 NIO 客户端服务器框架。
- Naga (<http://naga.googlecode.com>)是一个很小的库，提供了一些 Java 类把普通的 Socket 和 ServerSocket 封装成支持 NIO 的形式。

从性能测试上比较，Netty 和 Grizzly 都很快，而 Mina 稍慢一些。

JDK1.6 内部并不使用线程来实现非阻塞式 I/O。在 Windows 平台下，使用 select()，在新的 Linux 核下，使用 epoll 工具。

Niocchi(<http://www.niocchi.com>)是 Java 实现的开源异步 I/O 爬虫。

在爬虫中使用 NIO 的时候，最主要用到的就是下面 2 个类：

- **java.nio.channels.Selector**: Selector 类通过调用 select 方法，将注册的 channel 中有事件发生的 SelectionKey 取出来进行处理。如果想要把管理权交到 Selector 类手中，首先就先要在 Selector 对象中注册相应的 Channel。
- **java.nio.channels.SocketChannel**: SocketChannel 用于和 Web 服务器建立连接。

下面是使用 NIO 下载网页的例子，代码结构如图 2-6:

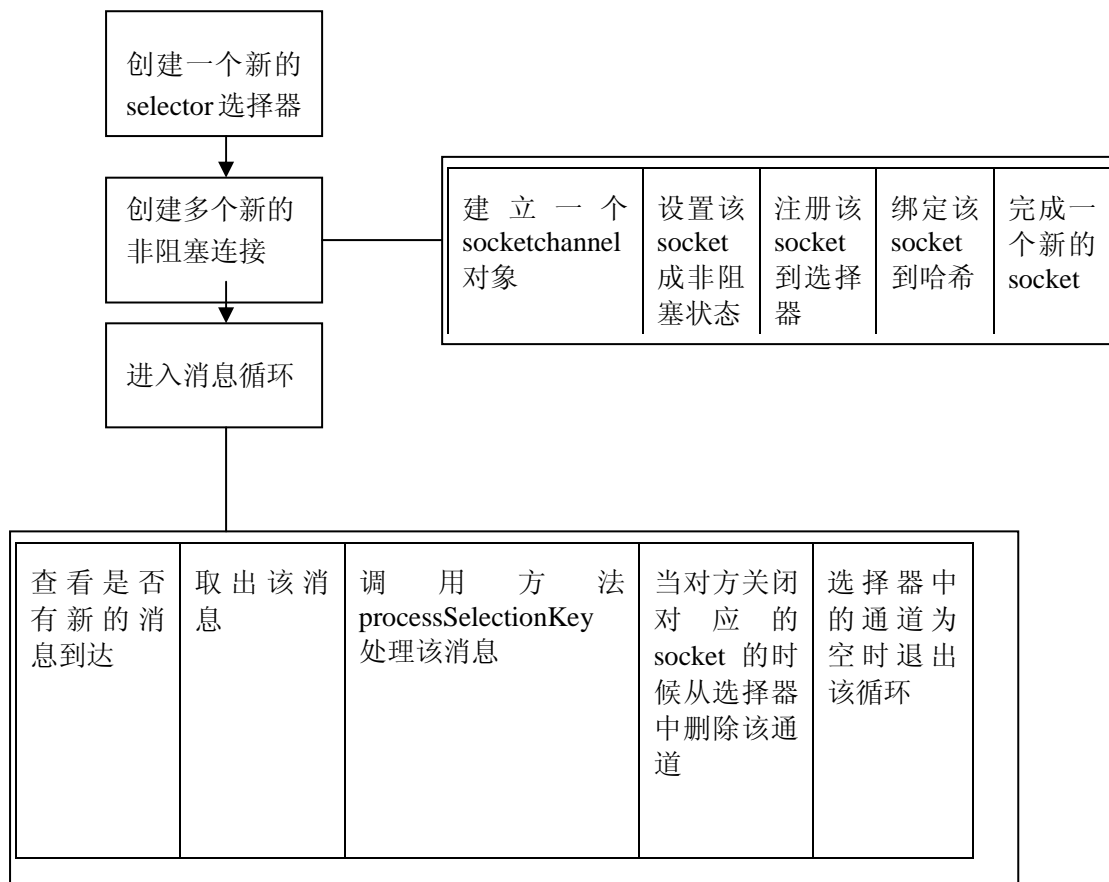


图 2-6 异步 IO

使用 NIO 下载网页的具体实现如下：

```

public static Selector sel = null;
public static Map<SocketChannel, String> sc2Path = new HashMap<SocketChannel, String>();

public static void setConnect(String ip, String path, int port) {
    try {

```

```
        SocketChannel client = SocketChannel.open();
        client.configureBlocking(false);
        client.connect(new InetSocketAddress(ip, port));
        client.register(sel, SelectionKey.OP_CONNECT | SelectionKey.OP_READ
            | SelectionKey.OP_WRITE); //
        sc2Path.put(client, path);
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(0);
    }
}

public static void main(String args[]) {
    try {
        sel = Selector.open();
        setConnect("www.lietu.com", "/index.jsp", 80); //添加一个下载网址
        setConnect("hao123.com", "/book.htm", 80); //添加另一个下载网址

        while (!sel.keys().isEmpty()) {
            if (sel.select(100) > 0) {
                Iterator<SelectionKey> it = sel.selectedKeys().iterator();
                while (it.hasNext()) {
                    SelectionKey key = it.next();
                    it.remove();
                    try {
                        processSelectionKey(key);
                    } catch (IOException e) {
                        key.cancel();
                    }
                }
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
        System.exit(0);
    }
}
```

```

public static void processSelectionKey(SelectionKey selKey)
    throws IOException {
    SocketChannel sChannel = (SocketChannel) selKey.channel();
    if (selKey.isValid() && selKey.isConnectable()) {
        boolean success = sChannel.finishConnect();
        if (!success) {
            selKey.cancel();
        }
        sendMessage(sChannel, "GET " + sc2Path.get(sChannel)
            + " HTTP/1.0\r\nAccept: */*\r\n\r\n");
    } else if (selKey.isReadable()) {
        String ret = readMessage(sChannel);
        if (ret != null && ret.length() > 0) {
            System.out.println(ret);
        } else {
            selKey.cancel();
        }
    }
}

```

//下载网页

```

public static String readMessage(SocketChannel client) {
    String result = null;
    ByteBuffer buf = ByteBuffer.allocate(1024);
    try {
        int i = client.read(buf);
        buf.flip();
        if (i != -1) {
            result = new String(buf.array(), 0, i);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return result;
}

```

```
//发送 HTTP 请求
public static boolean sendMessage(SocketChannel client, String msg) {
    try {
        ByteBuffer buf = ByteBuffer.allocate(1024);
        buf = ByteBuffer.wrap(msg.getBytes());
        client.write(buf);
    } catch (IOException e) {
        return true;
    }
    return false;
}
```

2.7 Web 结构挖掘

有一些垃圾网页，虽然包含大量的查询词，但却并非满足用户需要的文档。网页本身的重要性在网页排序中起着很重要的作用。PageRank 和 HITs 等算法可以根据 Web 结构自动学习出网页的重要性。在根据超链接生成的 Web 结构图中，每个节点代表一个网页。

2.7.1 存储 Web 图

挖掘网页之间的相互引用关系可以用来提高搜索结果排序或分析相似网页等。互联网搜索引擎需要连接服务器，能支持在网络图上快速的查询连接。典型的连接查询是哪些 URL 链接到一个指定的 URL。为此，希望在内存中存储 URL 外指链接和指向该 URL 的链接的映射。连接查询的应用包括爬虫控制，网络图分析，复杂的抓取优化和链接分析。

可以把每个网页看成一个节点，网页间的链接关系是有向边。几百万以上的网页将会抽象成一个几百万节点以上的图，因此不能简单的使用邻接表或矩阵完全在内存中处理这样大的图，往往需要通过压缩文件来存储 Web 图。

因为 BerkeleyDB 可以用来快速高效的存储海量数据。先采用它实现一个简单的 Web 图。来源 URL 和目的 URL 可以定义成一个多对多的映射。

```
@Entity
public class Link {
    @PrimaryKey public String fromURL;
    @SecondaryKey(related=Relationship.MANY_TO_MANY)
    public HashSet<String> toURL = new HashSet<String>();
}
```

在 WebGraph 类中定义从源 URL 到目标 URL 的主索引和目标 URL 到源 URL 的二级索引。

```
private PrimaryIndex<String,Link> outLinkIndex;
private SecondaryIndex<String,String,Link> inLinkIndex;
```

记录链接到 WebGraph。

```
public void addLink (String fromLink, String toLink) throws DatabaseException {
    Link outLinks = new Link();
    outLinks.fromURL = fromLink;
    outLinks.toURL = new HashSet<String>();
    outLinks.toURL.add(toLink);

    boolean inserted = outLinkIndex.putNoOverwrite(outLinks);

    if(!inserted){
        outLinks = outLinkIndex.get(fromLink);
        outLinks.toURL.add(toLink);
        outLinkIndex.put(outLinks);
    }
}
```

最后实现功能：取得指向一个 URL 地址的 Link 列表。

```
public String[] inLinks ( String URL ) throws DatabaseException {
    EntityIndex<String,Link> subIndex = inLinkIndex.subIndex(URL);
    String[] linkList = new String[(int)subIndex.count()];
    int i=0;
    EntityCursor<Link> cursor = subIndex.entities();
    try {
        for (Link entity : cursor) {
            linkList[i++] = entity.fromURL;
        }
    } finally {
        cursor.close();
    }
    return linkList;
}
```

假设有四十亿的网页，每个网页有十个链接到其他页面。需要 32 位或 4 个字节来指定每个链接的源和目标，要求内存的总字节数为： $4 * 10^9 * 10 * 8 = 3.2 * 10^{11}$ 。

可以利用 Web 图的一些基本性质使内存的使用要求降低到 10%。初看起来, 这似乎是一个数据压缩的问题, 对于这个问题有一些标准的解决方案。然而, 我们的目标不是简单地压缩网络图以便内存能放下, 还需要有效地支持连接查询。索引压缩和 Web 图的压缩问题类似。

假设每个网页用一个唯一的整数表示, 最简单的存储方法是如果有网页 $p_1 \rightarrow p_2$, 则用一行 “ $p_1 \ p_2$ ” 表示。还有一种方法是对 Web 图建立一个邻接表, 类似反向索引: 每行对应一个页面, 行之间用网页对应的整数排序。任何网页 p 的对应行包含一个排好序的整数数组, 每一个整数表示对应的网页链接到 p 。这个表方便查询哪些网页链接到 p 。用类似的方式, 还可以建立由包含 p 指向的页面的条目组成的表。这种邻接表的表现形式所占空间比最简单的表现形式减少一半。

下面的描述将集中讨论从每一个页面发出的链接表, 把这些技术应用到指向每个网页的链接也是一样的。为进一步减少表的存储空间, 我们利用如下几点:

1. 数组相似性: 因为表中的许多行中有许多项是相同的, 所以如果我们为几个相似行用一个原型行来代表, 那么剩下的差异就可以根据原型行简洁的表现出来。
2. 局域性: 从一个网页发出的许多链接都是到邻近的网页。例如, 在同一台主机上的网页, 因此建议在编码目标连接的时候, 使用小的整数来节约空间。可以使用差分编码的排序数组来实现这一点。不是直接存储每一条链接的终点网页的编号, 而是存储与前一项的偏移量。

现在来实现这些压缩技术。先把每个 URL 作为一个字符串并对这些字符串排序。图 2.8 显示了这些排好序了的表的一部分。对于一个网页字典排序, URL 的域名部分应该被反转, 因此 `ww.lietu.com` 变为 `com.lietu.www`, 但是如果主要关心的是和本地一台主机上的链接有关, 就没有必要这样做。

```
1: www.lietu.com
2: www.lietu.com/demo
3: www.lietu.com/english
4: www.lietu.com/news
5: www.lietu.com/job
6: www.lietu.com/train
```

图 2.8 URL 字典排序

对每个 URL, 取得它在排序后的 URL 地址列表中的位置作为表示它的唯一的整数。图

2.9 显示这样编号后的 Web 图的例子。在这个例子的序列中 `www.lietu.com/demo` 分配给整数 2，因为它在数组中排第二。

```
1: 1,2,4,8,16,32,64
2: 1,4,9,16,25,36,49,64
3: 1,2,3,5,8,13,21,34,55,89,144
4: 1,4,8,16,25,36,49,64
```

图 2.9 链接表的四行片断

下面来分析 Web 图的相似性和局域性。许多网站都有模板，站点的每个网页链接到一个固定的网页集合，固定的网页集合常常包括：版权说明页、网站首页、站点地图页等。在这种情况下，Web 图中的相应网页的行有许多项是共同的。此外，在按照 URL 字典的排序下，从一个网站出来的网页及有可能作为 Web 图中临近的行。

我们采取引用压缩(Reference compression)策略：从上往下遍历每一行，根据前面 7 行来编码当前行。在图 2.9 的例子中，可以编码第 4 行和偏移量为 2 的行相同，只需要用 8 代替 9。仅使用前面 7 行有两个有利条件：

1. 仅用 3 位来表示偏移量；这种选择在概率上是最优化的。
2. 固定的最大偏移量为一个像 7 这样的小的值，避免了在许多可能表示当前行的候选原型中选择，因为这是代价昂贵的搜索。

有时候前面 7 行没有一个能很好的表达当前行的原型，例如在每个不同站点的边界。这时候，简单的表示该行作为空行开始并向其中添加每一个整数。Web 图的每行中，使用差分编码(gap encoding)来存储增量，而不是实际值，基于值的分布来编码这些增量，可以获得更多的空间减少。这里提到的一系列技术使平均每个链接所需空间减少到 3 位，而不是最简单方法的 64 位。

虽然这些方法能够把 Web 图的表示存入内存中，但是仍然需要支持连通性查询的要求。如何查询一个网页指向哪些网页？首先，我们需要从哈希表中查询 URL 对应的行号。接下来，需要跟踪偏移量指示的其他行来重建这些被压缩了的项，而且，这个间接过程可能重复多次。

然而，在实践中不会经常发生这样的事情。Web 图的构建过程中可以引入启发式控制：检查前面 7 行哪个作为当前行的原型时，当前行和候选原型之间有个相似性的阈值。必须谨慎的选择阈值，如果设置的太高，会很少使用原型并重新表示许多行。如果设置的太低，许多行将会用原型项表示，在查询时的重建速度就会变慢。

WebGraph(<http://webgraph.dsi.unimi.it/>)是一个开源项目，它采用了上面介绍的引用压缩技术把图压缩成 BV 格式。

下面分析 BV 压缩格式的使用。因为 WebGraph 存储的节点是整型，不是直接的 URL 地址，首先要把 Web 网页映射成 0 到 n 的节点编号。然后形成有向边的图，例如 example.arcs 文件中包括(0,1,2,3,4)5 个节点组成的图，这个文件的内容如下：

0 2

1 2

1 4

2 3

3 4

下面这个命令将会产生一个压缩图到 bvexample 文件：

```
#java it.unimi.dsi.webgraph.BVGraph -g ArcListASCIIGraph example.arcs bvexample
```

生成压缩图以后我们可以统计图的出度：

```
#java it.unimi.dsi.webgraph.examples.OutdegreeStats -g BVGraph bvexample
```

执行结果是：

```
The minimum outdegree is 0, attained by node 4
```

```
The maximum outdegree is 2, attained by node 1
```

```
The average outdegree is 1.0
```

2.7.2 PageRank 算法

为了得到更好的搜索结果，使搜索引擎自动抵制那些堆砌关键词的垃圾网页，需要计算网页本身的重要性。假设用户在随机访问互联网的每个页面。看完一个页面后，可能再通过这个页面的链出链接访问其他的网页。为了在搜索结果中更好的对网页排序，考虑把用户更有可能访问的页面排在前面。有的网页被链接的次数多，所以用户访问的可能性更大。

一个简单的方法是用一个网页的入链数量通常表示此网页的重要程度。对应的概念叫做链接人气值。一般来说，如果从其他网页链接到一个网页的数量越多，那么这个网页就越重要。链接人气值的概念通常可以避免那些只被创造出来欺骗搜索引擎并且没有任何实际意义的网页得到好的等级，然而，许多网站管理员为了一个网页取得更好的排名，他们从大量其他没有意义的网页链接到该网页。链接人气值的问题在于没有考虑入站链接本身的权威性。

与链接人气值相比较，PageRank 的概念并不是简单地根据入站链接的总数评价一个网页的重要度。PageRank 对入站链接做加权计算，越是重要的网页通过链接指向一个网页，则这个网页就越重要。

首先介绍简化版本的 PageRank 计算方法。假设有 n 个网页指向网页 A ，这 n 个网页分别是 $T_1, \dots, T_i, \dots, T_n$ 。则网页 A 的 PageRank 值计算方法是：

$$PR(A) = PR(T_1)/C(T_1) + \dots + PR(T_n)/C(T_n)$$

这里， $PR(T_i)$ 是链接到网页 A 的网页 T_i 的 PageRank 值； $C(T_i)$ 表示网页 T_i 的出度链接数量。可以把链接比做互联网中的钞票，如果某个网站的钞票发行太多，那么它的钞票就要贬值，所以这里要除以 $C(T_i)$ 。

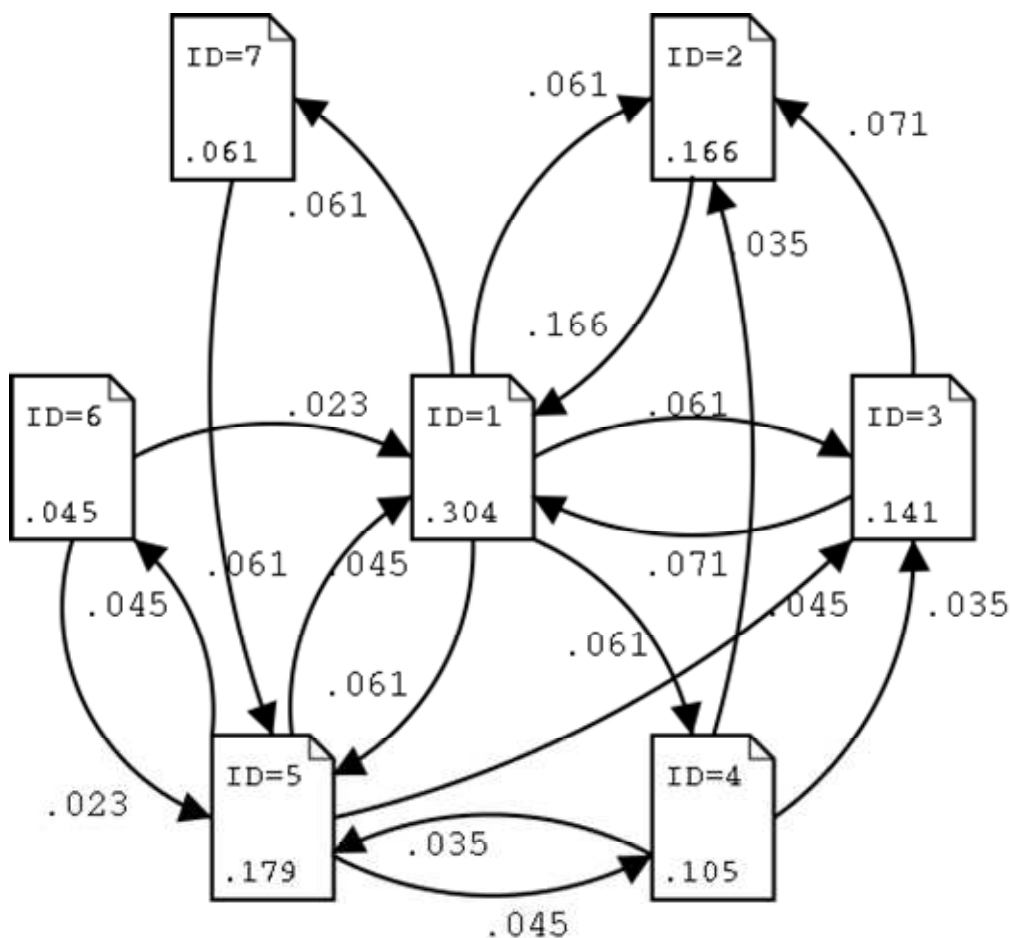


图 6-6 Web 图

例如图 6-6 中的第一个节点的 PageRank 值

= (ID=2 发出的 Rank)

+ (ID=3 发出的 Rank)

+ (ID=5 发出的 Rank)

+ (ID=6 发出的 Rank)

$$= 0.166 + 0.141/2 + 0.179/4 + 0.045/2$$

$$= 0.30375$$

用矩阵形式表示 PageRank 计算过程。Google 矩阵是一个随机矩阵用来表示一个图，图中的边表示网页之间的连接。随机矩阵的特点是每行值的和是 1。

$$A_{m \times n} = (a_{ij})$$

矩阵中的元素 a_{ij} 的取值方法是：如果存在从网页 i 指向网页 j 的超级链接，则 $a_{ij} = 1/N(i)$ ，这里 $N(i)$ 表示从网页 i 向外的链接数目。如果不存在这样的链接，则 $a_{ij} = 0$ 。例如图 6-6 表示的 Web 图的 Google 矩阵是：

$$\begin{pmatrix} 0 & 1/5 & 1/5 & 1/5 & 1/5 & 0 & 1/5 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1/3 & 1/3 & 0 & 1/3 & 0 & 0 \\ 1/4 & 0 & 1/4 & 1/4 & 0 & 1/4 & 0 \\ 1/2 & 0 & 0 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

令 $x = \text{PageRank}$ ， $M = A^T$ ，则可以通过迭代的方法计算：

$$x = M \times x$$

用 Google 矩阵计算 PageRank 的实现代码如下：

```
//p 是 A 矩阵的转置
double[][] p = Matrix.transpose(counts);

//初始化 pagerank 向量
double[] rank = new double[N];
for (int i = 0; i < N; i++) {
    rank[i] = 1.0/N; //N 是网页的总数，给所有的页面 rank 赋初始值 1/N
```

```

}

int T=100;
//执行乘积的 T 次迭代
for (int t = 0; t < T; t++) {
    rank = Matrix.multiply(p,rank);
}

```

标准的 PageRank 算法是：

$$PR(A) = (1-d)/N + d (PR(T_1)/C(T_1) + \dots + PR(T_n)/C(T_n))$$

这里，N 表示网页总数；PR(A)表示网页 A 的 PageRank 值；PR(T_i)表示链接到 A 的网页 T_i 的 PageRank 值；C(T_i)表示网页 T_i 的出站链接数量；d 是阻尼系数，取值范围是：0<d<1。

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 的页面的 PageRank 的递归值。

每个 PR(T_i)值并不是同等程度影响 PR(A)。在 PageRank 的计算公式里，T 对于 A 的影响还受 T 的出站链接数 C(T)的影响。这就是说，T 的出站链接越多，A 受 T 的这个连接的影响就越少。

PR(A)是所有 PR(T_i)之和。所以，对于 A 来说，每多增加一个入站链接都会增加 PR(A)。

最后，所有 PR(T_i)之和乘以一个阻尼系数 d，它的值在 0 到 1 之间。因此，阻尼系数的使用，减少了其它页面对当前页面 A 的重要度贡献。

PageRank 的特性可以通过以下图 6.7 表示：

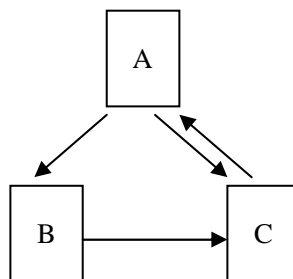


图 6.7 一个简单的 Web 图

假设一个小网站由三个页面 A、B、C 组成，A 连接到 B 和 C，B 连接到 C，C 连接到 A。虽然 Google 实际上将阻尼系数 d 设为 0.85，但这里我们为了简便计算就将其设为 0.5。

尽管阻尼系数 d 的精确值无疑是影响到 PageRank 值的，可是它并不影响 PageRank 计算的原理。因此，我们得到以下计算 PageRank 值的方程：

$$PR(A) = 0.5 + 0.5 PR(C)$$

$$PR(B) = 0.5 + 0.5 (PR(A) / 2)$$

$$PR(C) = 0.5 + 0.5 (PR(A) / 2 + PR(B))$$

求解这些方程，得到每个页面的 PageRank 值为：

$$PR(A) = 14/13 = 1.07692308$$

$$PR(B) = 10/13 = 0.76923077$$

$$PR(C) = 15/13 = 1.15384615$$

很明显所有页面 PageRank 之和为 3，等于网页的总数。就像以上所提的，此结果对于这个简单的范例来说并不特殊。

对于这个只有三个页面的简单范例来说，通过方程组很容易求得 PageRank 值。但实际上，互联网包含数以亿计的文档，是不可能解方程组的。

由于实际的互联网网页数量巨大，Google 搜索引擎使用了一个近似的、迭代的计算方法计算 PageRank 值。就是说先给每个网页一个初始值，然后利用上面的公式，循环进行有限次运算得到近似的 PageRank 值。我们再次使用“三页面”的范例来说明迭代计算，这里设每个页面的初始值为 1。计算过程如表 6-1。

迭代次数	PR(A)	PR(B)	PR(C)
0	1	1	1
1	1	0.75	1.125
2	1.0625	0.765625	1.1484375
3	1.07421875	0.76855469	1.15283203
4	1.07641602	0.76910400	1.15365601
5	1.07682800	0.76920700	1.15381050
6	1.07690525	0.76922631	1.15383947
7	1.07691973	0.76922993	1.15384490
8	1.07692245	0.76923061	1.15384592

迭代次数	PR(A)	PR(B)	PR(C)
9	1.07692296	0.76923074	1.15384611
10	1.07692305	0.76923076	1.15384615
11	1.07692307	0.76923077	1.15384615
12	1.07692308	0.76923077	1.15384615

表 6.1 PageRank 迭代计算方法

实际计算大的 Web 图时，往往需要上百次计算才能得到稳定的 PageRank 值。这里只给出了 12 次迭代计算作为示例。

根据标准的 PageRank 算法，Page 类的 recalculatePageRank 方法计算单个网页的 PageRank 值实现代码如下：

```
/**
 * 使用提供的阻尼系数重新计算这个网页的 PageRank
 *
 * @param decayFactor 模拟图遍历中的随机游走
 */
public void recalculatePageRank(double decayFactor) {

    //取得 PageRank 的收入部分：
    //每个进入网页的 PageRank 除以它的链出数之和
    double incomingPortion = 0.0;
    Iterator iterator = this.graph.getIncomingVertices(this).iterator();
    while (iterator.hasNext()) {
        RankablePage page = (RankablePage) iterator.next();
        incomingPortion += (page.getPageRank() / this.graph.outDegreeOf(page));
    }

    this.pageRank = (1.0 - decayFactor) + (decayFactor * incomingPortion);
}
```

PageRank 类的 run 方法计算整个图中的网页的 PageRank 值实现代码如下：

```
/**
 * 使用给定的迭代次数和阻尼系数计算图中的网页的 PageRank 值
 *
```

```
* @param numIterations 迭代次数
* @param decayFactor 模拟随机游走的阻尼系数
*/
public void run(int numIterations, double decayFactor) {

    //迭代式的重新计算网页的 PageRank 值
    for (int i = 1; i <= numIterations; i++) {

        // 计算 PageRank 值
        Iterator pagerIterator = this.graph.vertexSet().iterator();
        while (pagerIterator.hasNext()) {
            RankablePage page = (RankablePage) pagerIterator.next();
            if (graph.outDegreeOf(page) > 0) {
                page.recalculatePageRank(decayFactor);
            }
        }
    }

    //输出最终结果
    Iterator pagerIterator = this.graph.vertexSet().iterator();
    while (pagerIterator.hasNext()) {
        RankablePage page = (RankablePage) pagerIterator.next();
        System.out.println(page.getUrl() + " (" + page.getPageRank()+ ")");
    }
}
```

图 6.8 显示了如何在搜索引擎中使用计算出来的 PageRank 来改进搜索结果排序。在搜索系统实际运行时，因为 Web 图是动态变化的，所以网页的 PageRank 值也是动态变化的。

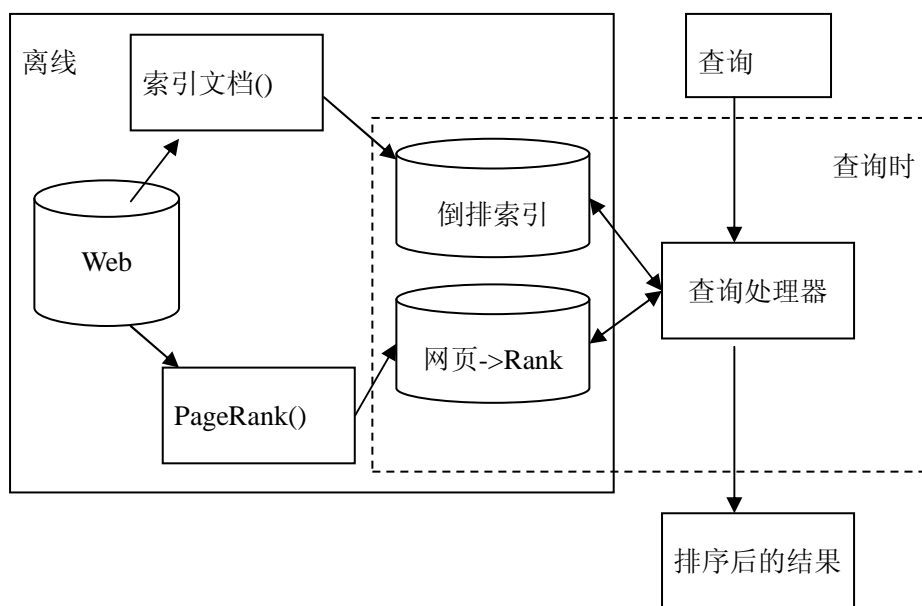


图 6.8 在搜索引擎中使用 PageRank

当把 PageRank 值大的网页排在前面的搜索引擎本身对用户访问网页的行为影响很大时，要减小 PageRank 值对于搜索结果排序的影响。因为考虑搜索引擎本身的影响后，PageRank 值大的网页访问量过高，用户过多的访问了这些网页。而引入 PageRank 只是为了让用户更快的访问到通过多次点击后会访问到的页面。

PageRank 除了可以用于评价网页的重要度，还可以评价学术论文的重要性。根据论文间的引用关系，模仿网页间的链接关系，构建论文间的引用关系图，利用 PageRank 的思想，计算每一篇论文的重要性。

网络爬虫可以利用 PageRank 值决定某个 URL 所需要抓取的网页数量和深度。重要性高的网页抓取的页面数量相对多一些，反之，则少一些。

用于关键词抽取：节点是词，边是词间的共现关系。即共现的词之间，判断为互相链接，这样组成一个无向图。某个词出现的次数越多，则链向它的词可能也就越多，这样的词可能就越重要。被重要的词所链向的词，也越重要。这种方法叫做 WordRank。

可以把 PageRank 算法用于句子抽取(文本摘要)：节点是句子，边是句子间的相似度。此外，还有用于文档评分的 DocRank。

2.7.3 HITs 算法

PageRank 算法中对于向外链接的权值贡献是平均的，也就是不考虑不同链接的重要性。而有些链接具有注释性，也有些链接是起导航或广告作用。有注释性的链接才用于权威判断。

下面介绍的 HITS 算法允许链接本身带权重。HITS 算法是由康奈尔大学的 Jon Kleinberg 博士于 1998 年首先提出的，HITS 的英文全称为 Hypertext-Induced Topic Search。

HITS 算法的实现代码如下：

```
public class HITS {
    /** 存储 Web 图的数据结构 */
    private WebMemGraph graph;

    /** 包含每个网页的评分 */
    private Map<Integer,Double> hubScores; //<id,value>

    /** 包含每个网页的 Authority */
    private Map<Integer,Double> authorityScores; //<id,value>

    /**
     *构造函数
     */
    public HITS ( WebGraph graph ) {
        this.graph = graph;
        this.hubScores = new HashMap<Integer,Double>();
        this.authorityScores = new HashMap<Integer,Double>();
        int numLinks = graph.numNodes();
        for(int i=1; i<=numLinks; i++) {
            hubScores.put(new Integer(i),new Double(1));
            authorityScores.put(new Integer(i),new Double(1));
        }
        computeHITS();
    }

    /**
     * 计算网页的 Hub 和 Authority 值
     */
    public void computeHITS() {
        computeHITS(25);
    }

    /**
```

```

    *计算网页的 Hub 和 Authority 值
    */
    public void computeHITS(int numIterations) {
        while(numIterations-->0) {
            for (int i = 1; i <= graph.numNodes(); i++) {
                Map<Integer,Double> inlinks    = graph.inLinks(new Integer(i));
                Map<Integer,Double> outlinks   = graph.outLinks(new Integer(i));
                double authorityScore = 0;
                double hubScore = 0;
                for (Integer id:inlinks.keySet()) {
                    authorityScore += (hubScores.get(id)).doubleValue();
                }

                for (Integer id:outlinks.keySet()) {
                    hubScore += (authorityScores.get(id)).doubleValue();
                }

                authorityScores.put(new Integer(i),new Double(authorityScore));
                hubScores.put(new Integer(i),new Double(hubScore));
            }
            normalize(authorityScores);
            normalize(hubScores);
        }
    }

    public void computeWeightedHITS(int numIterations) {
        while(numIterations-->0) {
            for (int i = 1; i <= graph.numNodes(); i++) {
                Map<Integer,Double> inlinks    = graph.inLinks(new Integer(i));
                Map<Integer,Double> outlinks   = graph.outLinks(new Integer(i));
                double authorityScore = 0;
                double hubScore = 0;
                for (Entry<Integer,Double> in:inlinks.entrySet()) {
                    authorityScore += (hubScores.get(in.getKey())).doubleValue() *
in.getValue();
                }
            }
        }
    }

```

```

        for (Entry<Integer,Double> out:outlinks.entrySet()) {
            hubScore += (authorityScores.get(out.getKey()).doubleValue() *
out.getValue());
        }

        authorityScores.put(new Integer(i),new Double(authorityScore));
        hubScores.put(new Integer(i),new Double(hubScore));
    }
    normalize(authorityScores);
    normalize(hubScores);
}
}

/**
 * 归一化集合
 */
private void normalize(Map<Integer,Double> scoreSet) {
    Iterator<Integer> iter = scoreSet.keySet().iterator();
    double summation = 0.0;
    while (iter.hasNext())
        summation += ((scoreSet.get((Integer)(iter.next())))).doubleValue();

    iter = scoreSet.keySet().iterator();
    while (iter.hasNext()) {
        Integer id = iter.next();
        scoreSet.put(id , (scoreSet.get(id)).doubleValue()/summation);
    }
}

/**
 * 返回与给定链接关联的 Hub 值
 */
public Double hubScore(String link) {
    return hubScore(graph.URLToIdentifier(link));
}

/**

```

```
    *返回与给定链接关联的 Hub 值
    */
    private Double hubScore(Integer id) {
        return (Double)(hubScores.get(id));
    }

    /**
     *初始化与给定链接关联的 Hub 值
     */
    public void initializeHubScore(String link, double value) {
        Integer id = graph.URLToIdentifier(link);
        if(id!=null) hubScores.put(id,new Double(value));
    }

    /**
     *初始化与给定链接关联的 Hub 值
     */
    public void initializeHubScore(Integer id, double value) {
        if(id!=null) hubScores.put(id,new Double(value));
    }

    /**
     *返回与给定链接关联的 Authority 值
     */
    public Double authorityScore(String link) {
        return authorityScore(graph.URLToIdentifier(link));
    }

    /**
     *返回与给定链接关联的 Authority 值
     */
    private Double authorityScore(Integer id) {
        return (Double)(authorityScores.get(id));
    }

    /**
     *初始化与给定链接关联的 Authority 值
```

```
*/  
public void initializeAuthorityScore(String link, double value) {  
    Integer id = graph.URLToidentifyer(link);  
    if(id!=null) authorityScores.put(id,new Double(value));  
}  
  
/**  
 *初始化与给定链接关联的 Authority 值  
 */  
public void initializeAuthorityScore(Integer id, double value) {  
    if(id!=null) authorityScores.put(id,new Double(value));  
}  
}
```

2.7.4 主题相关的 PageRank

如果不考虑出现在页面中或者指向该网页的锚点文本中的关键词,某个领域重要的页面可能在其他领域不重要。因此产生了主题相关的 PageRank(Topic-Sensitive PageRank)的想法。

在基本的 PageRank 算法中,使用 Web 链接结构计算出一个 PageRank 向量(向量的长度是网页的数量)来获取相对重要的网页。网页的重要性不依赖于任何特定的搜索查询词。为了达到更准确的搜索结果,可以基于一个话题集合计算一组 PageRank 向量来更精确的捕捉和一个特定主题相关的重要性的概念。通过使用这些预计算的有偏见的 PageRank 向量,在查询时生成和查询相关的每个网页的重要性分值。这样可以比一个单一的通用 PageRank 向量生成更准确的打分。使用查询关键词的话题为满足查询词的页面计算出话题相关的 PageRank 分值。

主题相关的 PageRank 结构如图 6.9 所示。其中需要用到已经把网站分好类的目录 Yahoo! 或 ODP。

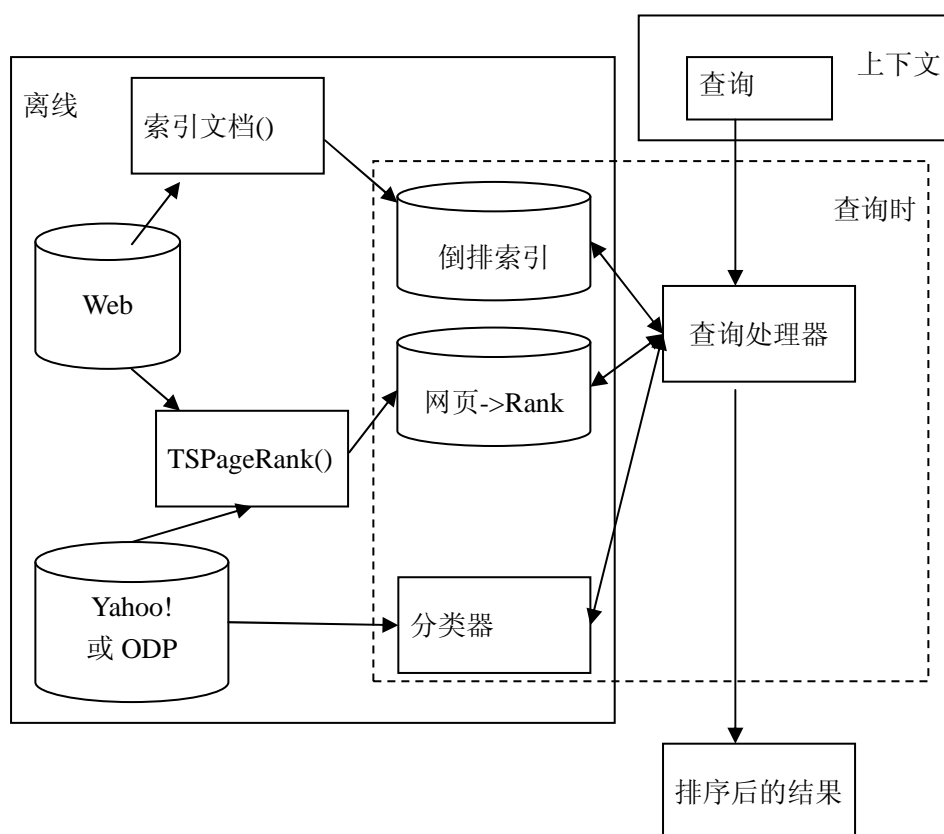


图 6.9 在搜索引擎中使用主题相关的 PageRank

改动主要在两个阶段：主题相关的 PageRank 向量集合的计算和在线查询时主题的确。首先使用一个有偏见的话题集生成一个 PageRank 向量的集合。这一步是在爬虫抓取网页时离线计算的。计算网页 i 在第 j 个类别上的 PageRank 值时，使用阻尼系数：

$$v_{ji} = \begin{cases} \frac{1}{|T_j|} & i \in T_j \\ 0 & i \notin T_j \end{cases}$$

这里的 T_j 是 ODP 类别 c_j 下的 URL 地址的集合。

在线查询时，对每个类别 c_j ，计算：

$$P(C_j | q) = \frac{P(C_j)P(q | C_j)}{P(q)} \propto P(C_j) \prod_i P(q_i | C_j)$$

为了计算 $P(c_j)$ ，对某个用户 k ，可以使用分布 $P_k(c_j)$ 来反映用户 k 的兴趣。

最后，对于在搜索结果集中的网页 d 计算相关性分值：

$$\text{Score}_{qd} = \sum_j P(C_j | q) \times \text{rank}_{jd}$$

这里， rank_{jd} 是网页 d 对于话题 c_j 的 PageRank 值。

2.8 部署爬虫

为了抓取更加稳定，往往让爬虫运行在一个独立的控制台进程中。可以在 MANIFEST.MF 文件中声明要运行的类。例如假设 `com.lietu.crawler.Spider` 包含 `main` 方法。

Manifest-Version: 1.0

Sealed: true

Main-Class: com.lietu.crawler.Spider

Class-Path: nekohtml.jar lucene-core-3.0.2.jar .

其中的 Class-Path 声明了依赖的 jar 包，最后的点代表当前路径。

通过 Ant 执行的 `build.xml` 来自动生成可执行的 jar 包。Ant 通过调用 `target` 树，就可以执行各种 `task`。每个 `task` 实现了特定接口对象。由于 Ant 构建文件是 XML 格式的文件，所以很容易维护和书写，而且结构很清晰。Ant 可以集成到开发环境中。由于 Ant 的跨平台性和操作简单的特点，Eclipse 已经集成了 Ant。

`crawler.jar` 包 里面要正好包含有用的 class 文件，既不能包含测试部分代码，也不能包含源文件。如果 Java 源代码文件编码不一致可能会出错，可以把编码统一成 GBK 或者 UTF-8。

```
<javac      encoding="utf-8"      debug="true"      srcdir="${src}"      destdir="${bin}"
classpathref="project.class.path" target="1.6" source="1.6"/>
```

完成打包后，执行 `crawler.jar`。

```
java -jar crawler.jar
```

安装 `lrzsz` 之后可以使用 `rz` 上传文件：

```
#yum install lrzsz
```

运行 `spider.sh`

```
# cat ./spider.sh
```

```
nohup java -Xmx1000m -jar crawler.jar &
```

```
#chmod +x ./spider.sh
```

2.9 本章小结

有人说：知识就是力量，爬虫就是自动获取知识的第一步。在互联网出现以前，大量收集信息是很困难的事情。公元前的埃及托勒密一世有一个梦想：要把世界上所有的文献、所有值得记下来的东西全部收藏。托勒密一世和他的后代托勒密二世、托勒密三世在港口城市亚历山大兴建图书馆，雇人抄写图书。所有到亚历山大港的船只都要把携带的书交出供检验，如发现有图书馆没有的书，则马上抄录，留下原件，将复制件奉还原主。还有重金收购等方法，建立起当时世界上最大的图书馆。在这个图书馆建立以前，知识在很大程度上是区域性的，但自建成第一座世界性图书馆后，知识也随之成为世界性的了。

没做过爬虫，可能以为爬虫只是一个广度有限遍历的应用。做过爬虫的可能就不这么想了。本章首先熟悉了网络爬虫工作的基本原理。然后了解抓取网页的实现。网络爬虫在某种程度上可以看成是一个 HTTP 客户端应用。首先介绍和网站打交道所使用的 HTTP 协议以及 TCP/IP 协议。然后用最简单的方法下载网页。再介绍实际下载网页所使用的 HTTPClient。

除了网页这样的文本信息，还有图片、FTP 等。介绍了比普通网页更规范的 RSS 种子下载方法。对于网页中的 JavaScript 的处理是难点问题。此外，还介绍了 Web 图的存取和挖掘算法。

可以用布隆过滤器或者 BerkeleyDB 实现 URL 地址查新。还可以使用 BerkeleyDB 实现网页快照功能，根据 URL 地址查询网页原文。

接下来简单回顾下网络爬虫的开发历史：

1994 年，休斯敦大学的 Eichmann(<http://slis.uiowa.edu/eichmann/index.shtml>)在美国航空航天局的资助下开发了互联网爬虫 RBSE(Repository Based Software Engineering)。RBSE 将爬虫和索引程序分离，这样不需要重新抓取就可以更新索引。爬虫执行宽度优先遍历或者有限深度优先的遍历。Spider 程序创建和操作存储在 Oracle 数据库中的 Web 图。一个修改后的 ASCII 字符浏览器(mite)根据 URL 下载指定的网页存储到本地文件，并把这个网页中的链接提取出来。

1994 年，华盛顿大学计算机系的学生 Pinkerton 开发了分布式爬虫 WebCrawler。运行在多个机器上的爬虫代理(Agent)用 HTTP 协议下载网页并把 HTML 格式的网页解析成纯文本。WebCrawler 使用简单的广度优先遍历方法抓取部分互联网中的网页。因为早期开源数据库不稳定，为了避免数据库服务器崩溃，它使用商业的 Oracle 数据库作为存储核心。1997 年 Excite 收购了 WebCrawler。Pinkerton 后来担任为 Lucene 和 Solr 提供商业服务的 Lucid Imagination 公司的首席架构师。

1998 年, 斯坦福大学的学生 Brin 和 Page 用 Python 开发了分布式爬虫系统 Google Crawler。URL 服务器给几个爬虫程序发送要抓取的 URL 列表。在解析文本的时候, 把新发现的 URL 传送给 URL 服务器并检测这个 URL 是不是已经存在, 如果不存在的话, 就把该 URL 加入到 URL 服务器。爬虫程序使用了 DNS 缓存来提高 DNS 查询效率。

1999 年, Compaq 公司的 Heydon 和 Najork 开发了 Mercator。C 语言的 IO 速度比 Java 快, 因此直到今天很多在线运行的爬虫采用 C 或 C++ 开发, 但 Mercator 却是一个用 Java 实现的成功网络爬虫。Mercator 是分布式的、模块化的。Mercator 用做 Alta Vista 搜索引擎的网络爬虫。Mercator 的模块包括可互换的“协议模块”和“处理模块”。“协议模块”负责怎样获取网页(例如使用 HTTP 协议), “处理模块”负责怎样处理页面。标准处理模块仅仅包括了解析页面和抽取新的 URL, 可以用其他处理模块来索引网页中的文本, 或者搜集 Web 统计数据。Mercator 使用 64 位的文档语义指纹来判断网页内容是否重复。因为布隆过滤器存在误判的情况, Mercator 没有采用 Internet Archive 爬虫中曾经采用的布隆过滤器。Mercator 用带缓存的 checksum 来判断 URL 是否抓取过。Najork 后来加入微软研究院, 担任研究员。

2001 年, IBM 的 Almaden 研究中心的 Edwards 等人开发了一个与 Mercator 类似的分布式的模块化的爬虫 WebFountain。它的特点是一个“控制者”机器控制一系列的“蚂蚁”机器, 可以实现增量抓取。经过多次下载页面后, 可以推测出每个页面的变化率, 然后可以获得一个最大的新鲜度的访问策略。商业信息网站 Factiva 使用 WebFountain 收集企业信用信息。WebFountain 是使用 C++ 实现的。

2002 年, FAST 公司的 Risvik 和 Michelsen 开发了分布式爬虫 FAST Crawler, 在 Fast Search&Transfer 和 www.AllTheWeb.com 网站使用。节点之间通过 distributor 模块交换发现的链接。每个机器有一个“文档调度器”来维护一个要下载的文档队列。“文档处理器”下载完网页后把网页存储在本地存储子系统。FAST Crawler 实现了增量式抓取, 优先抓更新活跃的网页。2004 年, Yahoo 收购 AllTheWeb 网站。Risvik 后来在 Yahoo 和 Google 工作过。

Cho 和 Garcia-Molina 在论文“Parallel Crawlers”研究了如何设计有效的并行爬虫, 在论文“Effective Page Refresh Policies For Web Crawlers”研究了网页更新方法。

2003 年初, 为了对网上的资源进行归档, 建立网络数字图书馆, 开发了开源网络爬虫 Heritrix。这个系统以运行时高可配置性的模块式开发, 所以非常适合做实验。

早期的互联网搜索研究中有很多是关于主题爬虫的。Menczer 和 Belew 在论文“Adaptive information agents in distributed textual environments”中设想了一种为个人用户服务的, 由自主软件代理的主题爬虫。用户同时输入网址以及关键字, 代理就会尝试寻找对用户有用的网页, 而且用户也可以对这些网页进行评估并反馈给系统。

Chakrabarti 在论文“Focused crawling: a new approach to topic-specific web resource discovery”中主要研究了主题爬虫, 他们的爬虫使用分类来判断抓取的网页的主题。他们认为对于提供主题链接方面来说, 非主题爬虫是无法与主题爬虫相比较。主题爬虫可以成功截获主题, 广泛的网络连接结构却使非主题爬虫迅速地移动到其他主题上。

Unicode 规范是一项难以置信的复杂的工作,包含了上万的字符。因为一些非西方语言特性的原因,许多象形文字都是一组 Unicode 字符组成的。所以这个规范的细节不仅说明这些字是什么,而且解释他们是如何组合的。新的字符仍然在源源不断地加入到 Unicode。

Bergman 的论文"The Deep Web: Surfacing Hidden Value"是对深网的一个深入的研究。尽管这项研究和 Web 标准一样的古老。它向我们展示了如何通过搜索引擎进行的抽样才能对在网络中的量指数挂钩有帮助。这个研究估算了一下大概有 5500 亿个网页存在于深网中,而与此同时却只有 10 亿个网页出现在可见网络中。He 在一个更近期的研究调查表明,深网近几年一直在持续地迅速地发展状态中。于是一种由 Ipeirotis 与 Gravano 所提出来的叫做查询探测的用于对深网的数据库进行探测分析的模型就出现了。抓取暗网可以参考一些 Web 应用自动测试工具。

网站地图、robots.txt 文件、RSS feeds 和 Atom feeds 都有它们自己的规范。这些格式说明那些成功的 Web 标准一般都是很简单的。

对于一些应用程序来说,可以用数据库系统来存储爬虫下载的文件。这方面有一些教科书,例如 Garcia-Molina 的著作《Database Systems: The Complete Book》提供了许多数据库如何运行的信息,也包含一些重要功能,例如查询语句、锁、恢复等。Chang 在论文“Bigtable: 一个结构化数据的分布式存储系统”描述了 Bigtable。

另外大型互联网公司出于类似的目的:大规模分布、高吞吐量、不需要昂贵的查询语句或者详细的事务支持,纷纷建立了自己的数据库系统。DeCandia 在论文“Dynamo: Amazon’s Highly Available Key-value Store”描述的亚马逊公司的 Dynamo 系统拥有低延迟的保证。Yahoo! 使用 UDB 系统处理巨大的数据量。

DEFLATE 和 LZW 是 2 个文本压缩工具。DEFLATE 是现在流行的 Zip、gzip 和 zlib 这些压缩工具的基础,LZW 是 Unix 压缩命令的基础,同样也适用于 GIF, Postscript, 以及 PDF 等文件形式。Witten 写的《Managing Gigabytes: Compressing and Indexing Documents and Images》提供了一些关于文本和图片压缩算法的详细讨论。

Yu 的论文 “Improving Pseudo-Relevance Feedback in Web. Information Retrieval Using Web Page Segmentation” 和 Gupta 的论文 “DOM-based Content Extraction of HTML Documents” 是对从网页中提取正文非常有用的文献。关于基于内容的网页排重将在后续章节介绍。

第3章 索引内容提取

搜索引擎经常要处理的文档格式包括 HTML、Word、PDF 等。这些文档格式中如 Word 和 PDF 是专有和非公开的格式，HTML 虽然是公开的标准，但是具体的实现却千差万别。而且这些文档格式往往存在不同的版本，比如 Word 包括 doc 和 docx 格式，PDF 有从 1.0 到 1.7 及其扩展版等 9 种不同的格式。

3.1 从 HTML 文件中提取文本

从 HTML 提取有效的文本，首先需要判断网页编码，这样才能确保不出现乱码。然后还需要从中提取需要的信息，因为很多网页中包括对用户不想要搜索的信息，例如广告、版权信息、导航条等。可以把网页保存成本地的文本文件，然后再开发提取信息的程序，这样开发的时候就不用担心网速不稳定。

有很多种方法实现网页信息提取，一般可以分为两种类型：一种是针对特定的网页特征提取结构化信息；还有一种就是通用的信息提取方法，例如网页去噪。下面首先介绍字符集编码以及判断网页编码的基本过程，然后介绍网页信息提取所用到的一些工具和方法，例如 HTMLParser 和 NekoHTML。网页提取的正确率是正确提取的文档数量除以测试集中的文档总数。

3.1.1 字符集编码

ASCII 码(American Standard Code for Information Interchange，美国信息交换标准码)是目前计算机中用得最广泛的字符集及其编码，由美国国家标准局(ANSI)制定。它已被国际标准化组织(ISO)定为国际标准，称为 ISO 646 标准。ASCII 字符集由控制字符和图形字符组成。

在计算机的存储单元中，一个 ASCII 码值占一个字节(8 个二进制位)，其最高位(b7)用作奇偶校验位。所谓奇偶校验，是指在代码传送过程中用来检验是否出现错误的一种方法，一般分奇校验和偶校验两种。奇校验规定：正确的代码一个字节中 1 的个数必须是奇数，若非奇数，则在最高位 b7 添 1。偶校验规定：正确的代码一个字节中 1 的个数必须是偶数，若非偶数，则在最高位 b7 添 1。

ISO 8859，全称 ISO/IEC 8859，是国际标准化组织(ISO)及国际电工委员会(IEC)联合制定的一系列 8 位字符集的标准，已经定义了 15 个字符集。

ASCII 收录了空格及 94 个“可印刷字符”，足够英语使用。但是，其他使用拉丁字母的语言(主要是欧洲国家的语言)，都有一定数量的变音字母，故可以使用 ASCII 及控制字符以外的区域来储存及表示。

除了使用拉丁字母的语言外，使用西里尔字母的东欧语言、希腊语、泰语、现代阿拉伯语、希伯来语等，都可以使用这个形式来储存及表示。

- ISO 8859-1 (Latin-1) - 西欧语言。
- ISO 8859-2 (Latin-2) - 中欧语言。
- ISO 8859-3 (Latin-3) - 南欧语言。世界语也可用此字符集显示。
- ISO 8859-4 (Latin-4) - 北欧语言。
- ISO 8859-5 (Cyrillic) - 斯拉夫语言。
- ISO 8859-6 (Arabic) - 阿拉伯语。
- ISO 8859-7 (Greek) - 希腊语。
- ISO 8859-8 (Hebrew) - 希伯来语(视觉顺序)。
- ISO 8859-8-I - 希伯来语(逻辑顺序)。
- ISO 8859-9 (Latin-5 或 Turkish) - 它把 Latin-1 的冰岛语字母换走，加入土耳其语字母。
- ISO 8859-10 (Latin-6 或 Nordic) - 北日耳曼语支，用来代替 Latin-4。
- ISO 8859-11 (Thai) - 泰语，从泰国的 TIS620 标准字集演化而来。
- ISO 8859-13 (Latin-7 或 Baltic Rim) - 波罗的语族。
- ISO 8859-14 (Latin-8 或 Celtic) - 凯尔特语族。
- ISO 8859-15 (Latin-9) - 西欧语言，加入 Latin-1 欠缺的法语及芬兰语重音字母，以及欧元符号。
- ISO 8859-16 (Latin-10) - 东南欧语言。主要供罗马尼亚语使用，并加入欧元符号。

很明显，ISO8859-1 编码表示的字符范围很窄，无法表示中文字符。但是，由于是单字节编码，和计算机最基础的表示单位一致，所以很多时候，仍旧使用 ISO8859-1 编码来表示。而且在很多协议上，默认使用该编码。

通用字符集(Universal Character Set, UCS)是由 ISO 制定的 ISO 10646(或称 ISO/IEC 10646)标准所定义的字符编码方式，采用 4 字节编码。

UCS 包含了已知语言的所有字符。除了拉丁语、希腊语、斯拉夫语、希伯来语、阿拉

伯语、亚美尼亚语、格鲁吉亚语，还包括中文、日文、韩文这样的象形文字，UCS 还包括大量的图形、印刷、数学、科学符号。

- UCS-2: 与 Unicode 的 2 字节编码基本一样。
- UCS-4: 4 字节编码，目前是在 UCS-2 前加上 2 个全零的字节。

Unicode 是一种在计算机上使用的字符编码。它是 <http://www.unicode.org> 制定的编码机制，要将全世界常用文字都函括进去。Unicode 为每种语言中的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换、处理的要求。1990 年开始研发，1994 年正式公布。随着计算机计算能力的增强，Unicode 也在面世以来的十多年里得到普及。

但自从 Unicode2.0 开始，Unicode 采用了与 ISO 10646-1 相同的字库和字码，ISO 也承诺 ISO10646 将不会给超出 0x10FFFF 的 UCS-4 编码赋值，使得两者保持一致。

Unicode 的编码方式与 ISO 10646 的通用字符集(Universal Character Set, UCS)概念相对应，目前的用于实用的 Unicode 版本对应于 UCS-2，使用 16 位的编码空间。也就是每个字符占用 2 个字节，基本满足各种语言的使用。实际上目前版本的 Unicode 尚未填满这 16 位编码，保留了大量空间作为特殊使用或将来扩展。

UTF 是 Unicode 的实现方式，不同于编码方式。一个字符的 Unicode 编码是确定的，但是在实际传输过程中，由于不同系统平台的设计不一定一致，以及出于节省空间的目的，对 Unicode 编码的实现方式有所不同。Unicode 的实现方式称为 Unicode 转换格式(Unicode Translation Format，简称为 UTF)。UTF 的两种实现方式是：

- UTF-8: 8 位变长编码，对于大多数常用字符集(ASCII 中 0~127 字符)它只使用单字节，而对其它常用字符(特别是朝鲜和汉语会意文字)，它使用 3 字节。
- UTF-16: 16 位编码，是变长码，大致相当于 20 位编码，值在 0 到 0x10FFFF 之间，基本上就是 Unicode 编码的实现，与 CPU 字序有关。

汉字编码有如下 4 种：

- GB2312 字集是简体字集，全称为 GB2312(80)字集，共包括国标简体汉字 6763 个。
- BIG5 字集是台湾繁体字集，共包括国标繁体汉字 13053 个。
- GBK 字集是简繁体字集，包括了 GB 字集、BIG5 字集和一些符号，共包括 21003 个字符。
- GB18030 是国家制定的一个强制性大字符集标准，全称为 GB18030-2000，它的推出使汉字集有了一个“大一统”的标准。

在 Windows 系统中保存文本文件时通常可以选择编码为 ANSI、Unicode、Unicode big endian 和 UTF-8，这里的 ANSI 和 Unicode big endia 是什么编码呢？

ANSI 使用 2 个字节来代表一个字符的各种汉字延伸编码方式，称为 ANSI 编码。在简体中文系统下，ANSI 编码代表 GB2312 编码，在日文操作系统下，ANSI 编码代表 JIS 编码。

UTF-8 以字节为编码单元，没有字节序的问题。UTF-16 以两个字节为编码单元，在解释一个 UTF-16 文本前，首先要弄清楚每个编码单元的字节序。

Unicode 规范中推荐的标记字节顺序的方法是 BOM(Byte Order Mark)。在 UCS 编码中有一个叫做"ZERO WIDTH NO-BREAK SPACE"的标记，它的编码是 FEFF。而 FFFE 在 UCS 中是不存在的字符，所以不应该出现在实际数据中。UCS 规范建议在传输字节流前，先传输标记"ZERO WIDTH NO-BREAK SPACE"。这样如果接收者收到 FEFF，就表明这个字节流是 Big-Endian 的；如果收到 FFFE，就表明这个字节流是 Little-Endian 的。因此标记"ZERO WIDTH NO-BREAK SPACE"又被称作 BOM。Windows 就是使用 BOM 来标记文本文件的编码方式的。

3.1.2 识别网页的编码

在实现从 Web 网页提取文本之前，首先要识别网页的编码，有时候还需要进一步识别网页所使用的语言。因为同一种编码可能对应多种语言，例如 UTF-8 编码可能对应英文或中文等任何语言。识别编码整体流程如下：

1. 从 Web 服务器返回的 content type 头信息中提取编码，如果是 GB2312 类型的编码要当成 GBK 处理。
2. 从网页的 Meta 标签中识别字符编码，如果和 content type 中的编码不一致，以 Meta 中声明的编码为准。
3. 如果仍然无法确定网页所使用的字符集，需要从返回流的二进制格式判断。
4. 确定网页所使用的语言，往往采用统计的方法来估计网页的语言。

从 Web 服务器返回的头信息中提取网页编码的代码如下。

```
final String CHARSET_STRING = "charset";
//输入头信息，返回网页编码
public static String getCharset (String content){
    int index;
    String ret = null;
    if (null != content) {
        index = content.indexOf (CHARSET_STRING);

        if (index != -1) {
            content =
```

```

        content.substring (index + CHARSET_STRING.length ()).trim ();
    if (content.startsWith ("=")) {
        content = content.substring (1).trim ();
        index = content.indexOf (",");
        if (index != -1)
            content = content.substring (0, index);

        //从字符串开始和结尾处删除双引号
        if (content.startsWith ("\"") &&
            content.endsWith ("\"") &&
            (1 < content.length ()))
            content = content.substring (1, content.length () - 1);

        //删除围绕字符串的任何单引号
        if (content.startsWith ("'") &&
            content.endsWith ("'") &&
            (1 < content.length ()))
            content = content.substring (1, content.length () - 1);

        ret = findCharset (content, ret);
    }
}

return (ret);
}

```

有的页面在头信息中不包括编码格式内容，需要从网页内部的 meta 标签中提取编码。例如下面这个：

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
```

下面利用 HTMLParser 包提取网页中的 meta 信息，关于 HTMLParser 将在后面的小节详细介绍。

```
String contentCharSet = tagNode.getAttribute("CONTENT");
```

另外一个和字符编码相关的问题是，有时候碰到 GB2312 编码的网页会有乱码问题，因为浏览器能正常显示包含 GBK 字符的 GB2312 编码网页。把 org.htmlparser.lexar.InputStreamSource 中的设置编码方法修改一下，把设置字符集为 GB2312 改成 GBK：

```
public void setEncoding (String character_set){
    if(character_set!= null && character_set.toLowerCase().equals("gb2312")){
        character_set = "GBK";
    }
}
...
```

JuniversalCharDet(<http://code.google.com/p/juniversalchardet/>)可以根据读入的字节流自动猜测页面或文件使用的字符集。实现原理是基于统计学的字符特征分析，统计哪些字符是最常见的字符。JuniversalCharDet 可以检测的字符编码有：中文、日文、韩文、西里尔文(Cyrillic)、希腊文、希伯来文。下面的代码调用 JuniversalCharDet 来自动判断文件的编码。

```
byte[] buf = new byte[4096];
String fileName = args[0];
java.io.FileInputStream fis = new java.io.FileInputStream(fileName);

//构建一个 org.mozilla.universalchardet.UniversalDetector 对象的实例
UniversalDetector detector = new UniversalDetector(null);

int nread;
while ((nread = fis.read(buf)) > 0 && !detector.isDone()) {
    detector.handleData(buf, 0, nread); //给编码检测器提供数据
}
//通知编码检测器数据已经结束
detector.dataEnd();

//取得检测出的编码名
String encoding = detector.getDetectedCharset();
if (encoding != null) {
    System.out.println("Detected encoding = " + encoding);
} else {
    System.out.println("No encoding detected.");
}

//在再次使用编码检测器之前，先调用 UniversalDetector.reset()
detector.reset();
```

此外还有 Jchardet 和 cpdetector。Jchardet 是基于比较老的 chardet 模块，而 JuniversalCharDet 基于新的 UniversalCharDet 模块，因此检测结果更准确。

3.1.3 网页编码转换为字符串编码

网页中的字符可能被转义成英文字符表示，例如“网站导航”是“网站导航”的转义。这类符号以“&”开始，以“;”结束。http://commons.apache.org 项目中的 `StringEscapeUtils.unescapeHtml(String str)` 可以把 HTML 编码的字符串转换成原文。下面的代码把输入的转义后的字符串转换成原文。

```
String htmlStr = "&#32593;&#31449;&#23548;&#33322;";
String textStr = Entities.HTML40.unescape(htmlStr);
```

3.1.4 使用正则表达式提取数据

正则表达式可以定义一些概率无关的提取模式。`java.util.regex` 包提供了对正则表达式的支持。其中的 `Pattern` 类代表一个编译后的正则表达式。通过 `Matcher` 类根据给定的模式查找输入字符串。通过调用 `Pattern` 对象的 `matcher` 方法得到一个 `Matcher` 对象。使用正则表达式提取字符串的例子如下：

```
String example = "This is my small example string which I'm going to use for pattern matching.";
Pattern pattern = Pattern.compile("\\w+");
Matcher matcher = pattern.matcher(example);
// 检查所有的出现
while (matcher.find()) {
    System.out.print("开始位置: " + matcher.start());
    System.out.print(" 结束位置: " + matcher.end() + " ");
    System.out.println(matcher.group());
}
```

下面的例子提取网页中的链接：

```
String pageContents = "<a href='http://www.lietu.com'>猎兔</a>";
Pattern p = Pattern.compile("<a\\s+href\\s*=\\s*\"?(.*?)\"?|>",
    Pattern.CASE_INSENSITIVE); //忽略大小写
Matcher m = p.matcher(pageContents);
while (m.find()) { //打印网页中所有的链接
    String link = m.group(1).trim();
    System.out.println(link);
}
```

有些链接的形式是：

```
<a href='http://www.lietu.com'>猎兔</a>
```

为了更好的匹配单引号，可以把模式修改成：

```
"<a\\s+href\\s*=\\s*[\\\"']?(.*?)\\s*>"
```

例如“2009-12-6”这样的格式用正则表达式匹配如下：

```
Pattern p = Pattern.compile("\\d{2,4}-\\d{1,2}-\\d{1,2}");
Matcher m = p.matcher(inputStr);
if(m.find()){
    String strDate = m.group();
}
```

其他的一些匹配日期的正则表达式有：“\\d{2,4}\\d{1,2}\\d{1,2}”和“\\d{2,4}年\\d{1,2}月\\d{1,2}日”以及“\\d{2,4}\\d{1,2}\\d{2,4}”。

邮政编码查询					
邮编查询:	<input type="text"/>	<input type="button" value="查询"/>	区号查询:	<input type="text"/>	<input type="button" value="查询"/>
省名查询:	<input type="text" value="山西"/>	<input type="button" value="查询"/>	市名查询:	<input type="text"/>	<input type="button" value="查询"/>
县名查询:	<input type="text"/>	<input type="button" value="查询"/>	村名查询:	<input type="text"/>	<input type="button" value="查询"/>

省名	地区	县市	乡镇村	邮政编码	区号
山西	朔州	怀仁县	鹅毛口	038301	0349
山西	朔州	怀仁县	海北头	038301	0349
山西	朔州	怀仁县	何家堡	038301	0349
山西	朔州	怀仁县	河头	038301	0349
山西	朔州	怀仁县	金沙滩	038302	0349
山西	朔州	怀仁县	里八庄	038301	0349

图 3-1 地址页面

图 3-1 是一个包含地址信息的网页。利用 HTMLParser 解析包可以方便的解析网页，提取想要的网页信息，但是在有些情况下利用正则表达式可能更方便(如果对正则表达式不熟悉请参考相关 Java 正则表达式教程)。同样以上述网页为例，利用正则表达式提取网页表格数据如下：

首先，根据要提取的网页信息查看网页源代码，这里我们提取的是网页表格中地名相关的信息。查看网页源码如下：

```
</table> <br><table width="533" border="0" cellpadding="0" cellspacing="1" align="center"
bgcolor="#3366CC"><tr align="center" bgcolor="#6699CC"><td width="70"> 省 名 </td><td
width="100"> 地 区 </td><td width="100"> 县 市 </td><td width="50"> 乡 镇 村 </td><td
width="60"> 邮 政 编 码 </td><td width="50"> 区 号 </td></tr><tr align="center"
onmouseout="this.style.background='#FFFFFF'"
```

```
onmouseover="this.style.background='BDDFFF'"                bgcolor="#FFFFFF"><td><a
href=Code_Default.asp?Type=Sm&SearchKeyStr= 山 西 > 山 西 </a></td><td><a
href=Code_Default.asp?Type=Ds&SearchKeyStr= 朔 州 > 朔 州 </a></td><td><a
href=Code_Default.asp?Type=Xs&SearchKeyStr= 怀 仁 县 > 怀 仁 县 </a></td><td><a
href=Code_Default.asp?Type=Ys&SearchKeyStr= 鹅 毛 口 > 鹅 毛 口 </a></td><td><a
href=Code_Default.asp?Type=Yb&SearchKeyStr=038301>038301</a></td><td><a
href=Code_Default.asp?Type=Qh&SearchKeyStr=0349>0349</a></td></tr><tr align="center"
```

根据要提取的表格信息构造正则表达式如下：

```
<td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td></tr>"
```

在这里要注意“(.*?)”用法，括号在这里表示分组，即每一个数据项表示一组，“?”表示非贪婪匹配，仅匹配到紧跟其后“”为止。

其次，根据读取的网页字符流和正则表达式提取表格信息，其相关代码如下：

```
//输入参数 input 是当前网页的内容字符串
public static void parserHtml(String input) {
    String regex =
" <td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td><td><a.+?>(.*?)</a></td></tr>";
    Pattern p = Pattern.compile(regex); //编译正则表达式
    Matcher m = p.matcher(input); //匹配模式
    while (m.find()) { //按行提取表格数据
        String province = m.group(1);
        String area = m.group(2);
        String city = m.group(3);
        String village = m.group(4);
        String postcode = m.group(5);
        String code = m.group(6);
        System.out.println(province); //打印出山西
        System.out.println(area); //打印出朔州
        System.out.println(city); //打印出怀仁县
        System.out.println(village); //打印出鹅毛口
        System.out.println(postcode); //打印出 038301
        System.out.println(code); //打印出 0349
    }
}
```

3.1.5 使用 HTMLParser 实现定向抓取

正则表达式中按字符处理网页源代码，但网页源代码是按标签组织的。有专门的程序包可以把源代码字符串转换成一个一个的标签，例如 HTMLParser。

HTMLParser(<http://htmlparser.sourceforge.net/>)是一个良好实现的提取 HTML 文件解析程序库。它的主要功能是对纯 HTML 文档和内嵌的 HTML 进行语法分析。可以使用它完成对非规范的 HTML 文件解析。HTMLParser 主要包括两个类：在 Lexer 类中实现了对 HTML 语法的基本解析，也就是把页面中的字符组装成标签并封装成节点(Node)对象；在 Parser 类中实现了对节点的过滤和选择。

为了调用 HTMLParser，需要添加 `htmllexer.jar` 或者 `htmlParser.jar` 到项目的类路径中。`htmllexer.jar` 提供一个底层接口，可以通过线性，平坦且连续的方式遍历节点。`htmlparser.jar` 能够提供一系列嵌套标签节点，例如字符串，备注和其他标记节点。

虽然可以由一个 URL 对象直接构造 Lexer 对象，但是为了避免读取页面的乱码问题，可以直接由包含 HTML 页面内容的字符串构造 Lexer 对象：

```
//用 HTML 页面内容做参数
Lexer le=new Lexer(contentStr);
//生成一个网页解析器
Parser parser=new Parser(le);
```

HTMLParser 将网页转换成一个个串联的 Node。例如下面把一个 URL 指定的网页中的标签都打印出来。

```
Lexer lexer = new Lexer (url.openConnection ());
Node node;
while (null != (node = lexer.nextNode()))//取得下一个节点
    System.out.println (node.toString());
```

在上面这个循环中，每次调用 Lexer 的 `nextNode()` 方法返回下一个 Node 节点，直到没有节点的时候返回空值。

各种不同的 HTML 标签则用不同的 Node 类型来区分。例如超链接标签是 `LinkTag`，而图像标签则是 `ImageTag`，这些都是 Node 的子类。可以通过 `getTagName()` 判断标签的类型。注意，返回的标签名字都是大写的。

Node 有三个直接的子类，分别是：

- **RemarkNode**：代表 HTML 中的注释。
- **TagNode**：标签节点，TagNode 有很多继承的子类，是种类最多的节点类型，例如 `LinkTag` 和 `ImageTag` 等具体的节点类都是 TagNode 的实现。TagNode 经常有各种各样的属性，

可以通过 `getAttribute` 方法得到属性值。例如通过 `ImageTag.getAttribute("src")` 取得图像节点中的图像地址。

- **TextNode**: 文本节点。

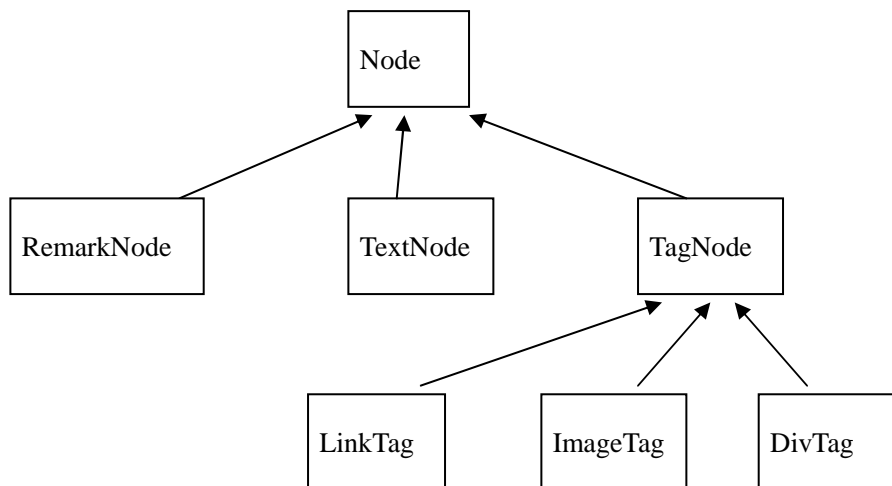
例如：“`广告业务`” 由下面三个 Node 组成：

- `` 是 **TagNode**。
- “广告业务” 是 **TextNode**。
- `` 也是 **TagNode**，不过它是结束标签，它的 `isEndTag()` 是真。

再如：

```
<!--enpproperty   <date>2006-01-06</date><author></author><title> 广 告 业 务
</title><nodename></nodename><cnodename>      联      系      我      们
</cnodename><parentid>2788</parentid><parentname>BTV      广      告
</parentname><Pparentid>2781</Pparentid><Pparentname>    网      站      栏      目
</Pparentname><source></source><keyword></keyword><SiteID>1</SiteID><ip></ip><
ChannelID>2781</ChannelID><imp>0</imp><subtitle></subtitle><introtitle></introtitle>
/enpproperty-->
```

这整段就是一个 **Remark Node**。



节点类型继承关系图

有的网址位于 `iframe` 标签中，这样可以把几个网址中的内容同时在浏览器中呈现。例如：

```
<iframe src="html_intro.asp" width="100%" height="300">
```

```
<p>Your browser does not support iframes.</p>
</iframe>
```

为了追踪抓取嵌套在 iframe 标签中的网页，通过遍历节点寻找 iframe 中的 URL 地址：

```
String path = "http://tianya.cn";
URL urlPage = new URL(path);
Lexer lexer = new Lexer(urlPage.openConnection()); // 解析网页
Node node;
while (null != (node = lexer.nextNode())) {
    if (!(node instanceof TagNode)) { // 判断 node 是否 TagNode 类型的节点
        continue;
    }

    TagNode tagNode = (TagNode) node; // 强制转换类型
    String name = tagNode.getTagNames(); // 根据名字判断是否 IFRAME
    if (name.equals("IFRAME") && !tagNode.isEndTag()) {
        String newURL = tagNode.getAttribute("src");
        URL u = new URL(new URL(path), newURL); // 取得 URL 的绝对地址
        System.out.println(u.toString());
    }
}
```

下面的例子读入并通过 Lexer 分析本地硬盘文件中的 HTML 页面。

```
//读入 HTML 文件中的内容
File htmlFile = new File("D:/content.htm");
Scanner scanner = new Scanner(htmlFile, "GBK");
scanner.useDelimiter("\\z");

StringBuilder buffer = new StringBuilder();
while (scanner.hasNext()){
    buffer.append(scanner.next());
}
scanner.close();

//分析 HTML 中的内容
Node node;
Lexer lexer = new Lexer(buffer.toString());
lexer.setNodeFactory(new PrototypicalNodeFactory ()); //设置要分析的节点类型
```

```
while (null != (node = lexer.nextNode ())) {
    //打印 Node 节点类型
    System.out.println(node.getClass().toString());
}
```

Lexer 可以识别的 Node 类型在 NodeFactory 的子类中定义，PrototypicalNodeFactory 是 HTMLParser 自带的工厂类，通过 setNodeFactory 方法设置 Node 工厂类：

```
lexer.setNodeFactory(new PrototypicalNodeFactory ());
```

缺省情况下 HTMLParser 所有能够识别的 HTML 标签都在 PrototypicalNodeFactory 中注册对应的 Node 类型。可以通过在 PrototypicalNodeFactory 注册新的 TagNode 来让 HTMLParser 识别新的 Node。比如为了单独识别 Strong 标签，可以先新建一个 Strong 类。

```
public class Strong extends CompositeTag {
    private static final String[] mlds = new String[] {"STRONG"}; //标签名称

    public Strong () {
    }

    public String[] getIds () {
        return (mlds);
    }
}
```

然后在 Node 工厂类中注册这个类：

```
PrototypicalNodeFactory factory = new PrototypicalNodeFactory ();
factory.registerTag (new Strong());
```

使用 Parser 类可以提取出符合特定条件的 Node 的集合。以提取出网页中所有的“<a>”标签为例。Parser 类的方法 extractAllNodesThatAre(LinkTag.class)将 HTML 内容中存在的所有的标签 LinkTag 给解析出来放到一个 Node 数组 NodeList。或者可以创建 new NodeClassFilter(LinkTag.class)，然后把它作为提取的依据。

```
NodeClassFilter linkFilter = new NodeClassFilter(LinkTag.class);
// 得所有过滤后的标签
NodeList list = parser.extractAllNodesThatMatch(linkFilter);
```

打印包含属性 class=l 的“<a>”标签中的链接地址：

```
NodeFilter filter=new AndFilter(new NodeClassFilter(LinkTag.class),
                                new HasAttributeFilter("class","l"));
NodeList nodelist=parser.extractAllNodesThatMatch(filter);
```

```
int listCount=nodelist.size();
for(int i=0;i<listCount;i++){
    TagNode node=(TagNode)nodelist.elementAt(i);
    System.out.println(node.getAttribute("href"));
}
```

下面的例子用 NodeFilter 提取一个表格中的内容：

```
Parser parser=new Parser(1e); //Parser 类解析 html 工具;
parser.setEncoding("gb2312"); //设置编码格式
NodeFilter filter=new AndFilter(new TagNameFilter("tr"),
                                new HasAttributeFilter("onmouseout",
                                                         "this.style.background='#FFFFFF'"));
//迭代所有节点
NodeList nodelist=parser.extractAllNodesThatMatch(filter);
int listCount=nodelist.size();
for(int i=0;i<listCount;i++){
    Node node_title=nodelist.elementAt(i);
    System.out.println((node_title.toHtml()));
}
```

有各种各样的 Filter，说明如表 3-1：

Filter 名称	用途
AndFilter	相当于一个 AND 操作符，接受所有同时满足两个 Filter 的节点
CssSelectorNodeFilter	接受所有支持 CSS2 选择器的节点
HasAttributeFilter	接受所有否含有某个属性(还可以按该属性的值过滤)的节点
HasChildFilter	接受所有含有子节点符合该 Filter 的节点
HasParentFilter	接受所有含有父节点符合该 Filter 的节点
HasSiblingFilter	接受所有含有兄弟节点符合该 Filter 的节点
IsEqualFilter	接受所有和某个特定的节点相同的节点
LinkRegexFilter	接受所有 linkTag 标签的 link 值，匹配给定的正则表达式的节点
LinkStringFilter	接受所有 linkTag 标签的 link 值,匹配给定的字符串的节点

Filter 名称	用途
NodeClassFilter	接受所有接受指定的类的节点
NotFilter	接受所有不符合 Filter 的节点
OrFilter	相当于一个 AND 操作符,接受所有满足两个 Filter 中任意一个的节点
RegexFilter	接受所有满足指定正则表达式的 String Nodes
StringFilter	接受所有满足指定 String 的 String Nodes
TagNameFilter	接受所有满足指定 Tag 名的 TagNodes
XorFilter	相当于一个 XOR 操作符,接受所有只满足其中 1 个 Filter 的节点

表 3-1 HTMLParser 中的 Filter 表

除了提取的方法,还可以通过访问者(visitor)得到网页中所有的图像标签:

```
ObjectFindingVisitor visitor = new ObjectFindingVisitor(ImageTag.class);
parser.visitAllNodesWith(visitor);
Node[] nodes = visitor.getTags();//通过特定的 visitor 得到符合条件的 Tag
for (int i = 0; i < nodes.length; i++) {
    ImageTag imgLink = (ImageTag) nodes[i];
    System.out.println(nodes[i]);
    System.out.println("ImageURL = " + imgLink.getImageURL());
    System.out.println("ImageLocation = " + imgLink.extractImageLocn());
    System.out.println("SRC = " + imgLink.getAttribute("SRC"));
}
```

通过访问者定义了节点对象上的操作。用访问者方式提取 Node 中包含的文本的实现:

```
public String extractText(String str) {
    Parser parser_text = new Parser(str);
    TextExtractingVisitor visitor = new TextExtractingVisitor();// 操作方式是提取文本
    parser_text.visitAllNodesWith(visitor); // 遍历所有节点
    return visitor.getExtractedText(); // 提取文本信息
}
```

每次测试解析网址的时候都下载内容往往太慢,可以通过字符串快速测试提取代码:

```
String content = "<a href=\"a.html\" class=\"title\"></a>";
Lexer lexer = new Lexer(content); // 解析网页
```

```
Parser parser = new Parser(lexer);
NodeFilter filter=new AndFilter(new NodeClassFilter(LinkTag.class),
    new HasAttributeFilter("class","title"));
NodeList nodelist=parser.extractAllNodesThatMatch(filter);
System.out.println(nodelist.size());
System.out.println(nodelist.elementAt(0).toHtml());
```

HTMLParser 几乎解析不出来没有闭合的标签，而 NekoHTML 则可以做到。

3.1.6 结构化信息提取

一般把要提取的数据结构定义成为一个类，然后有一个解析网页的方法根据输入网页返回解析出来的类实例。例如，需要实现一个招聘职位搜索引擎。从招聘网页提取职位信息。

首先定义好用来接收网页数据的 POJO(Plain Old Java Object)类 Job。这个类用来描述职位信息：

```
public class Job {
    public String comName = null;
    public String positionName = null;
    public String email = null;
    public String releaseData = null;
    public String city = null;
    public String number = null;
    public String experience = null;
    public String salary = "面议";
    public String knowledge = null;
    public String acceptResumeLanguage = null;
    public String positionDescribe = null;
    public String comintro = null;
    public String comHomePage = null;
    public String comAddress = null;
    public String postCode = null;
    public String fax = null;
    public String connectPerson = null;
    public String telephone = null;
    public String languageAbility = null;
    public String funtype_big = null;
    public String funtype = null;
```

```

public String province = null;
public String toString(){
    return "公司名称: "+comName+"\n 职位名称: "+positionName+
        "\n 电子邮箱: " +email+"\n 发布日期: "+releaseData+
        "\n 工作地点: "+city+"\n 招聘人数: "+number+
        "\n 工作年限: "+experience+"\n 薪水范围: "+salary+
        "\n 学历: "+knowledge+
        "\n 接受简历语言: "+acceptResumeLanguage+
        "\n 职位描述: "+positionDescribe+"\n 公司简介: "+comintro+
        "\n 公司网站: "+comHomePage+"\n 地址: "+comAddress+
        "\n 邮政编码: "+postCode+"\n 传真: "+fax+
        "\n 联系人: "+connectPerson+"\n 电话: "+telephone+
        "\n 外语要求: "+languageAbility;
}

```

然后从下面这个职位发布的网页提取公司名称。查看公司名称周围的特征：

```
<td align="left" class="title02">北京亿达网通科技发展有限公司</TD>
```

可以利用 AndFilter 来处理：

//提取公司名字的 Filter

```

NodeFilter filter_title = new AndFilter(new TagNameFilter("TD"),
    new HasAttributeFilter("class","title02"));

```

提取公司名称的实现如下：

```

NodeList nodelist = parser.extractAllNodesThatMatch(filter_title);
Node node_title = nodelist.elementAt(0);
String comName = extractText(node_title.toHtml());
comName = comName.replaceAll("[\t\n\r ]+", ""); //去掉多余的空格

```

从职位详细页面 URL 地址提取职位信息的完整的过程如下：

//输入网址，返回和该网址正文信息对应的职位对象

```

public Job extractContent(String url){
    Job position = new Job();
    Parser parser = new Parser(url);
    //设置编码方式
    parser.setEncoding("GB2312");
    //提取公司名字的 Filter
    NodeFilter filter_title = new AndFilter(new TagNameFilter("TD"),

```

```

        new HasAttributeFilter("class","title02"));
//提取公司名称
NodeList nodelist = parser.extractAllNodesThatMatch(filter_title);
Node node_title = nodelist.elementAt(0);
position.comName = extractText(node_title.toHtml());
position.comName = position.comName.replaceAll("[\t\n\r ]+","");
//提取职位名称的 Filter
NodeFilter filter_job_name = new AndFilter(new TagNameFilter("td"),
        new HasAttributeFilter("bgcolor","#FFEEE0"));
NodeFilter job_description_end1 = new AndFilter(new TagNameFilter("table"),
        new HasAttributeFilter("width","100%"));
NodeFilter job_description_end2 = new HasAttributeFilter("cellpadding","5");
NodeFilter company_description = new AndFilter(new TagNameFilter("table"),
        new HasAttributeFilter("width","98%"));
//提取职位的名称
parser.reset();
nodelist.removeAll();
nodelist = parser.extractAllNodesThatMatch(filter_job_name);
Node node_job_name = nodelist.elementAt(0);
position.positionName = extractText(node_job_name.toHtml());
//去掉空格
position.positionName =
        position.positionName.toString().replaceAll("[\t\n\r ]+","");
//提取职位描述
NodeFilter job_description_end =
        new AndFilter(job_description_end1,job_description_end2);
parser.reset();
NodeList nodelist_description =
        parser.extractAllNodesThatMatch(job_description_end);
Node node_job_description = nodelist_description.elementAt(0);
position.positionDescribe = extractText(node_job_description.toHtml());
position.positionDescribe =
        position.positionDescribe.replaceAll("[\t\n\r ]+"," ");
//提取公司简介
parser.reset();
nodelist_description.removeAll();
nodelist_description = parser.extractAllNodesThatMatch(company_description);

```

```
Node node_company_description = nodelist_description.elementAt(0);
//处理公司简介
position.comintro = doCompanyDescription(node_company_description);
return position;
}
```

3.1.7 网页的 DOM 结构

虽然表示网页的 HTML 文档格式不如 XML 规范,但是仍然可以把它转换成 DOM(文档对象模型)树组织的结构。其中标签是 DOM 树内部的节点,而详细的文本、图象或者超链接则是叶节点。图 3-3 展示了 HTML 的一部分以及它相应的 DOM 树。

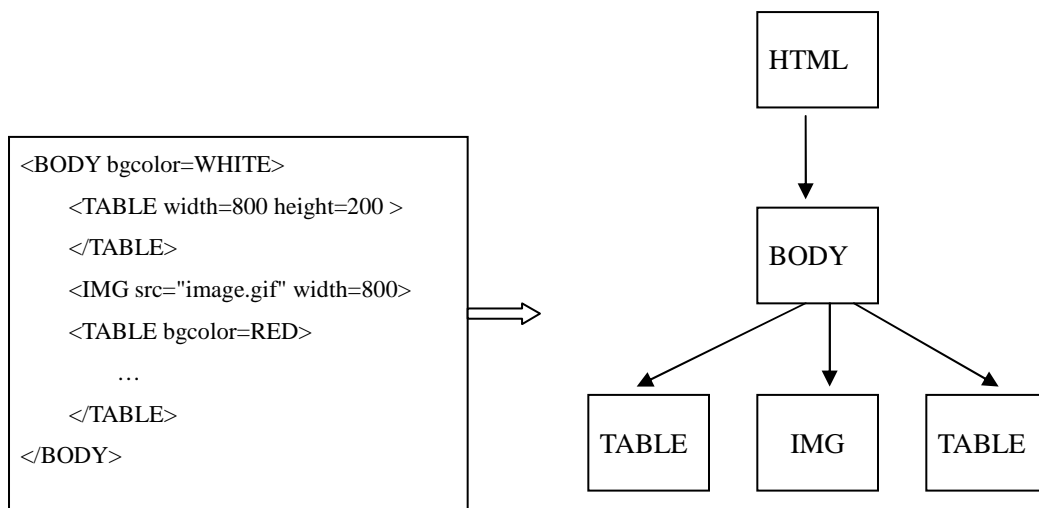


图 3-3 网页 DOM 树

在 Firefox 浏览器中,使用 DOM 查看器(DOM Inspector)和 Firebug 这两个插件工具都可以看到网页的 DOM 树。

访问 <http://www.lietu.com/>。在浏览器菜单中,选择“工具”→“DOM 查看器”,打开 DOM 查看器。在 DOM 查看器的左侧视图中,会看到 DOM 节点的树状图。

如果觉得依次展开 DOM 节点树中每层很不方便,可以使用 Inspect Element 扩展,它能迅速找到 DOM 查看器中的指定元素。XPath Helper(<http://xpathhelper.mozdev.org/>)可以扩展 DOM 查看器。它在 DOM 查看器上增加了一个可以输入 XPath 的工具条,可以在当前网页计算 XPath 表达式。

Firebug(<http://getfirebug.com/>)是 Firefox 的一个插件。通过 Firebug,可以在 Firefox 中查看任何页面的已解析的文档对象模型(DOM)。可以获得每个 HTML 元素、属性和文本节点的详细信息;也可以看到每个页面样式表中的所有 CSS 规则;也可以看到每个对象的所有脚本属性。在 Firefox 中安装 Firebug 插件后,输入网址“<http://www.lietu.com>”即可看到网

页 DOM 树。比较而言，DOM 查看器中显示的网页 DOM 树概念更符合下面介绍到的网页解析器 NekoHTML 所生成的 DOM 树。

org.w3c.dom 是操作 DOM 的标准接口。其中的基本概念有：

- Node 表示 DOM 树中的节点；
- Document 表示整个 HTML 文档，也就是 DOM 树中的根节点，所以是 Node 的子类；
- Attr 表示节点中的属性，例如，IMG 类型的节点包括 src 属性；
- Comment 表示一个注释类型的节点，所以它也是 Node 的子类。

3.1.8 使用 NekoHTML 提取信息

NekoHTML(<http://nekohtml.sourceforge.net/>)可以解析 HTML 文档并形成标准的 DOM 树。因为 HTML 文档不如 XML 规范，可能存在一些格式不完整的元素，例如有些标签没有对应的结束标签。NekoHTML 可以通过补偿标签来整理这些有缺陷的网页，也就是说把 HTML 文档转换成 XML 格式。然后就可以通过 XML 解析器 xerces 访问 NekoHTML 解析出的网页 DOM 树。使用 NekoHTML 的 Java 项目中需要导入的包是 nekohtml.jar、xercesImpl.jar 和 xalan.jar。

解析某些网站可能会出现 HIERARCHY_REQUEST_ERR 错误。这是因为 NekoHTML 不能解析有命名空间的 xhtml 页面。

```
DOMParser parser = new DOMParser();
//不执行命名空间处理
parser.setFeature("http://xml.org/sax/features/namespace", false);
```

得到一个网页的 DOM 树的流程如下：

```
DOMParser parser = new DOMParser(); //创建 DOM 解析器
//设置参数
parser.setFeature("http://xml.org/sax/features/namespace", false);
parser.parse("http://www.lietu.com"); //解析网页
Document doc = parser.getDocument(); //得到根节点
```

节点有三种类型：元素节点、注释节点和文本节点。判断各种节点类型的代码如下：

```
int type = node.getNodeType(); //取得节点类型
if (type == Node.ELEMENT_NODE) {
    System.out.print("元素节点");
} else if (type == Node.COMMENT_NODE){
```

```

        System.out.print("注释节点");
    } else if(type == Node.TEXT_NODE){
        System.out.println("文本节点");
        //打印文本节点中的文字信息
        System.out.println(node.getNodeValue());
    }
}

```

以一个显示图片的 **IMG** 标签节点为例，**Node** 的基本概念如图 3-6 所示。节点中的属性和属性值可以看成键/值对。比如要得到图片节点中的 **src** 属性中的值，可以调用 `node.getAttributes().getNamedItem("src")` 方法。

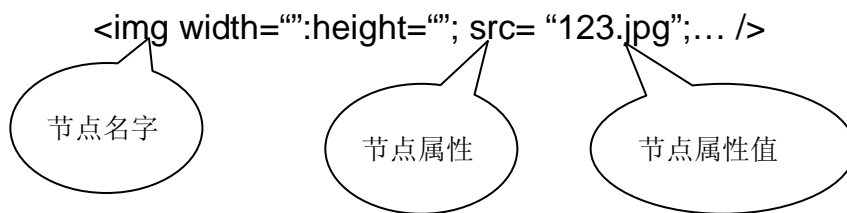


图 3-6 Node 的基本概念

可以修改节点，比如要移除节点属性，可以用下面的方法：

```

private void removeAttribute(final Node iNode, final String iAttr) {
    iNode.getAttributes().removeNamedItem(iAttr);
}

```

每一个节点在 DOM 树中都有一个特定的位置。**Node** 接口中有一些发现一个指定节点的周边节点的方法，如图 3-7 所示，说明如下：

- `Node getFirstChild();` //得到第一个孩子节点
- `Node getLastChild();` //得到最后一个孩子节点
- `Node getNextSibling();` //得到下一个兄弟节点
- `Node getPreviousSibling();` //得到上一个兄弟节点
- `Node getParentNode();` //得到父亲节点
- `NodeList getChildNodes();` //得到所有的孩子节点

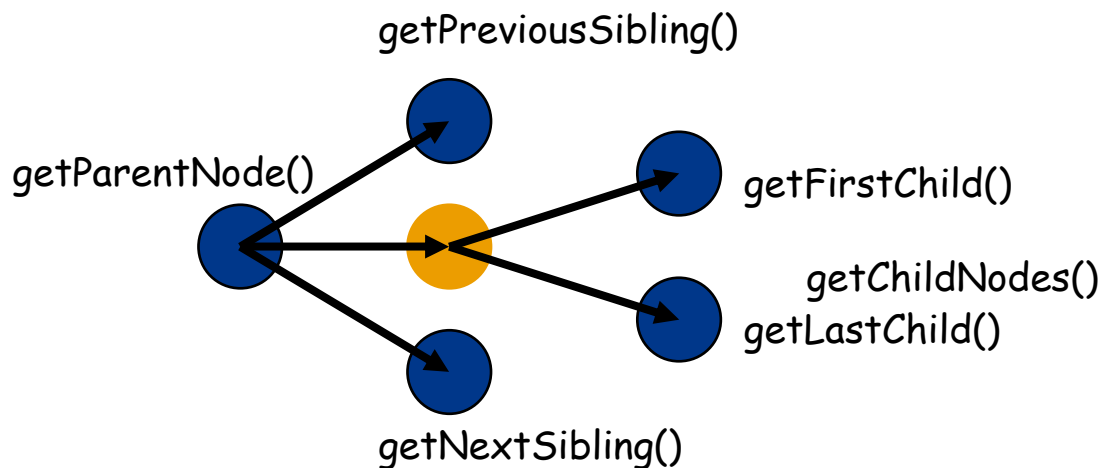


图 3-7 发现周边节点

下面的程序使用一个递归方法来遍历 DOM 树并打印出节点内容：

```

public static void main(String[] argv) throws Exception {
    DOMParser parser = new DOMParser(); //创建 DOM 解析器
    parser.parse("http://www.lietu.com"); //解析网页
    print(parser.getDocument(), ""); //从根节点开始遍历
}

public static void print(Node node, String indent){
    if (node.getNodeValue() != null){
        if(!"".equals(node.getNodeValue().trim())){
            System.out.print(indent);
            System.out.println(node.getNodeValue()); //打印节点内容
        }
    }
    Node child = node.getFirstChild(); //取得第一个孩子节点
    while (child != null){
        //递归调用 print 方法
        print(child, indent+" ");
        child = child.getNextSibling(); //取得孩子节点的下一个兄弟节点
    }
}

```

HTMLParser 中的 TextExtractingVisitor 实现了提取文本，但 NekoHTML 中却没有现成的实现。使用 NekoHTML 从网页中抽取文本的实现如下：


```

public static String TextExtractor(Node root){
    //若是文本节点的话，直接返回
    if (root.getNodeType() == Node.TEXT_NODE) {
        return root.getNodeValue().trim();
    }
    if(root.getNodeType() == Node.ELEMENT_NODE) {
        Element elmt = (Element) root;
        //风格节点 STYLE 和 脚本节点 SCRIPT 下也可能包括了文本节点类型的子节点
        //往往需要跳过这样的文本节点
        if (elmt.getTagName().equals("STYLE")
            || elmt.getTagName().equals("SCRIPT"))
            return "";

        NodeList children = elmt.getChildNodes();
        StringBuilder text = new StringBuilder();
        for (int i = 0; i < children.getLength(); i++) {
            text.append(TextExtractor(children.item(i)));// 递归调用 TextExtractor
        }
        return text.toString();
    }
    //对其它类型的节点，返回空值
    return "";
}

```

下面的代码打印出网页中所有的链接：

```

public static void main(String[] argv) throws Exception {
    DOMParser parser = new DOMParser();
    parser.parse("http://www.lietu.com");
    Set<String> urlSet = new HashSet<String>();// 存储网页中的 Url
    extract(parser.getDocument(), urlSet);
    for (String url : urlSet) { //打印出所有的网页链接
        System.out.println(url);
    }
}

//提取链接
public static void extract(Node node, Set<String> urlSet) {
    if (node instanceof HTMLAnchorElementImpl) {

```

```

//添加到集合中
urlSet.add(((HTMLAnchorElementImpl)node).getAttribute("href"));
}
Node child = node.getFirstChild();//取得第一个孩子节点
while (child != null) {
    // 递归调用 extract 方法
    extract(child, urlSet);
    child = child.getNextSibling();//取得兄弟节点
}
}

```

xerces-2_9_1 包括了一个 LSSerializer 对象可以保存 DOM 解析后的文档或节点, 下面的代码将 iNode 节点的内容输出到控制台:

```

registry = DOMImplementationRegistry.newInstance();
DOMImplementationLS impl =
    (DOMImplementationLS)registry.getDOMImplementation("LS");
LSSerializer writer = impl.createLSSerializer();
LSOutput output = impl.createLSOutput();
output.setByteStream(System.out);
output.setEncoding(System.getProperty("file.encoding"));
writer.write(iNode, output);

```

还可以使用 `writeToString` 方法将文档或文档中的节点所代表的一段网页写入字符串:

```
String nodeString=domWriter.writeToString(iNode);
```

找出符合条件的 DIV 节点下的 URL 链接。网页的 DOM 如图 3-8 所示:

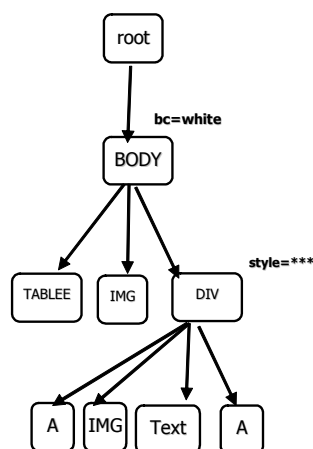


图 3-8 选取 DIV 下的节点

```
DOMParser parser = new DOMParser();
parser.parse("http://news.steelhome.cn/zhzx/");
Node divNode = visit(parser.getDocument()); //取得符合条件的 DIV 节点
visitDiv(divNode); //取得 DIV 节点下的 URL 地址
```

visit 方法通过从网页根节点递归遍历返回符合条件的 DIV 节点。

```
public static Node visit(Node node) {
    int type = node.getNodeType();

    if (type == Node.ELEMENT_NODE && "DIV".equals(node.getNodeName())) {
        Node att = node.getAttributes().getNamedItem("style");
        if (att != null
            && "float:left;width:464px;margin:1px;border: 1px solid #ccc;"
                .equals(att.getNodeValue())) {
            return node;
        }
    }

    Node child = node.getFirstChild();
    while (child != null) {
        // 递归调用
        Node t = visit(child);
        if (t != null) {
            return t;
        }
        child = child.getNextSibling();
    }
    return null;
}
```

visitDiv 方法通过从 DIV 节点递归遍历，打印出 DIV 节点下的 URL 地址。

```
public static void visitDiv(Node node){
    int type = node.getNodeType();

    if (type == Node.ELEMENT_NODE && "A".equals(node.getNodeName())) {
        Node t = node.getAttributes().getNamedItem("title");
        if(t!=null) {
            String title = t.getNodeValue();
        }
    }
}
```

```

        Node h = node.getAttributes().getNamedItem("href");
        String url = h.getNodeValue();
        System.out.println(title+"\t"+url);
    }
}

Node child = node.getFirstChild();
while (child != null) {
    // 递归调用
    Node t = visitDiv(child);
    child = child.getNextSibling();
}
}

```

提取"更多"文本对应的链接:

```

public static String getMoreUrl(Node node) throws Exception {
    int type = node.getNodeType();
    if(type == Node.TEXT_NODE && "更多".equals(node.getNodeValue())) {
        return
node.getParentNode().getAttributes().getNamedItem("href").getNodeValue();
    }

    Node child = node.getFirstChild();
    while (child != null) {
        // 递归调用
        String t = getMoreUrl(child);
        if (t != null) {
            return t;
        }
        child = child.getNextSibling();
    }
    return null;
}

```

在 `HTMLParser` 中, 经常使用 `NodeFilter` 来选择出想要的节点, 也可以使用另外一个 `NodeFilter` 筛选 DOM 树上的节点。这里的 `NodeFilter` 是一个接口。在自己定义的类中实现这个接口。`acceptNode()`方法包含自定义的逻辑决定节点是否通过过滤器。

```
// 接收标签名叫做'A'的 DOM 元素
```

```
private static final class LinkFilter implements NodeFilter {
    // 检查每个节点，看是否符合条件
    public short acceptNode(Node n) {
        if (n instanceof Element) {
            if (((Element) n).getTagName().equals("A")) {
                return NodeFilter.FILTER_ACCEPT; //接收
            }
        }
        return NodeFilter.FILTER_REJECT; //拒绝
    }
}
```

可以使用文档遍历器(DocumentTraversal)遍历 DOM 树，并返回符合条件的节点。并没有显式的 get 方法可以得到文档遍历器，通过强制类型转换得到文档遍历器。代码如下：

```
DOMParser parser = new DOMParser();
parser.parse("http://www.lietu.com");
Document document = parser.getDocument();

//把一个文档实例转换成文档遍历器
DocumentTraversal traversal = (DocumentTraversal) document;
NodeIterator iterator = traversal.createNodeIterator(document.getDocumentElement(),
                                                    NodeFilter.SHOW_ALL,
                                                    new LinkFilter(),
                                                    true);

//遍历符合条件的节点
for (Node n = iterator.nextNode(); n != null; n = iterator.nextNode()) {
    String href = ((HTMLAnchorElementImpl) n).getAttribute("href");
    System.out.println(href);
}
```

假设有几种类型的节点需要处理，其中一种包含图片，另外一种不包含。

```
private static enum NodeType {
    Good, //商品
    Good_No_Img//没有图片的商品
}
```

节点过滤器修改成：

```
private static final class GoodFilter implements NodeFilter {
```

```

public NodeType nodeType = null;
@Override
public short acceptNode(Node n) {
    if (n instanceof Element) {
        if (((Element) n).getTagName().equals("IMG")
            && ((Element) n).hasAttribute("alt")
            && ((Element) n).getAttribute("alt").equals(
                "Product Details")) {
            nodeType = NodeType.Good;
            return NodeFilter.FILTER_ACCEPT;// 接受节点
        }
        if (((Element) n).getTagName().equals("A")
            && ((Element) n).hasAttribute("class")
            && ((Element) n).getAttribute("class").equals("title")) {
            nodeType = NodeType.Good_No_Img;
            return NodeFilter.FILTER_ACCEPT;// 接受节点
        }
    }
}
}

```

提取文本类型的节点

```

//过滤出想要的文本节点
private static final class TextFilter implements NodeFilter {

    @Override
    public short acceptNode(Node n) {
        if (n instanceof Text && n.getNodeValue().equals("搜索产品")) {
            return NodeFilter.FILTER_ACCEPT;// 接受节点
        }
        return NodeFilter.FILTER_REJECT;// 拒绝节点
    }
}

```

下面的代码移除节点 iNode:

```
iNode.getParentNode().removeChild(iNode);
```

通过如下三行代码创建节点:

```
Document mTree = parser.getDocument(); //取得根节点
```

```
Element nDiv = mTree.createElement("DIV"); //创建一个 DIV 节点
parentNode.appendChild(nDiv); //增加子节点
```

为了避免乱码问题，可以先下载整个网页缓存到字符串，然后再解析。代码如下：

```
//content 是包含网页内容的字符串
InputSource inputsource=new InputSource(new StringReader(content));
parser.parse(inputsource);
```

或者使用 HTTPClient 下载网页，然后交给 NekoHTML 解析：

```
String pageUrl = "http://www.lietu.com/";
HttpGet httpget = new HttpGet(pageUrl);
HttpClient httpclient = new DefaultHttpClient();
HttpEntity entity = httpclient.execute(httpget).getEntity();
InputSource is = new InputSource(entity.getContent());
is.setEncoding("utf-8");
parser.parse(is); // 解析网页
Document doc = parser.getDocument();
```

3.1.9 使用 XPath 提取信息

NekoHTML 可以把 HTML 文档转换成 XML 文档。XML 文档的某一部分可以用 XPath 来描述。可以用计算节点特征的方法找到覆盖主要正文的内容节点。找到这样一个节点后，可以把这个节点用 XPath 表示出来，例如：

```
/HTML[1]/BODY[1]/DIV[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]
```

但是这种 XPath 的绝对路径表示方式当 DOM 树的节点有删除或修改后就失效了。还可以用 XPath 中的相对路径来选择节点，例如选择网页中所有链接的 XPath 是 “//a”。选择一个标题 DIV 可能是 “//div[@class='title']”。如果路径以单斜线 / 开始，那么该路径就表示到一个元素的绝对路径。如果路径以双斜线 // 开头，则表示相对路径，选择文档中所有满足双斜线//之后规则的元素。

星号 * 表示选择所有由星号之前的路径所定位的元素，例如：“/HTML[1]/BODY[1]/*”。

为了支持通过 XPath 取得节点，Java 项目必须导入 xalan.jar。下面的例子提取当当网图书的 ISBN 信息：

```
DOMParser parser = new DOMParser();

parser.setProperty("http://cyberneko.org/html/properties/default-encoding", "gb2312");
parser.setFeature("http://xml.org/sax/features/namespace", false);
```

```
String bookURL =
    "http://product.dangdang.com/product.aspx?product_id=20733895";
parser.parse(bookURL);

Document doc = parser.getDocument();
String isbnXpath = "/HTML/BODY/DIV[3]/DIV[6]/DIV[2]/DIV/DIV[3]/UL/LI[9]";
NodeList products;
products = XPathAPI.selectNodeList(doc, isbnXpath);
System.out.println("found: " + products.getLength());
Node node = null;
for (int i = 0; i < products.getLength(); i++) {
    node = products.item(i);
    System.out.println(i + ":\n" + node.getTextContent());
}
```

3.1.10 网页去噪

一个页面中的经常包括导航栏或底部的公司介绍等信息,这样的信息在很多页面都会出现,可以看作噪音信息。一般情况下可以去掉网页的 DOM 树中的 FORM、Select、IFRAME、INPUT、STYLE 中的节点。

不同的页面类型可以使用不同的去噪方法。常见的两种网页类型是目录导航式页面(List Page)和详细页面(Detail Page)。详细页面需要抽取的正文信息包括标题和内容等。

详细页面的特征有:

- 非锚点文本较多。
- 一般都有明显的文本段落,文字较多,相应的标点符号也较多。
- URL 较长。在一般的 Web 网站链接导航树上,主题型网页主要分布于底层,多为叶节点。对于同一网站而言,主题型网页的 URL 相对较长。URL 体现了网站内容管理的层次,对于大型网站而言,URL 往往非常有规律。
- 链接较少。主题型网页的主体在于“文字”,相对于导航型网页,其链接数较少。
- 主体文字在 DOM 树中的层次较深。
- 网页标题可能出现在 Title 标签或 H1 标签中, H2 标签可能表示文章的段落标题。

详细页面中网页噪音特征:

- 多以链接的形式出现，链接到别的相关页面。
- 有很多锚文本，但标点符号较少，锚文本往往是对其他链接页面的说明。
- 有许多常见的噪音文本，如版权声明等。在视觉上，多出现于网页的边缘。

网页去噪的方法基本方法是利用各种通用的特征来区分有效的正文和页眉，页脚，广告等其他信息。其中一个常用的特征是链接文字比率。可以把 Html 转换成 DOM 树，对每个节点计算链接文字比率。如果该节点的链接文字比率高于 1/4 左右，就把这个节点去掉。

下面首先用递归调用的方法计算一个节点下的链接数。

```
/**
 * 计算一个节点下的链接数
 *
 * @param iNode 开始计算的节点
 * @return 链接数
 */
public static int getNumLinks(final Node iNode) {
    int links = 0;
    if (iNode.hasChildNodes()) {
        Node next = iNode.getFirstChild();
        while (next != null) {
            Node current = next;
            next = current.getNextSibling();
            //递归调用计算链接数的方法
            links += getNumLinks(current);
        }
    }

    if (isLink(iNode))
        links++;

    return links;
}
```

计算一个节点下的有效正文长度，忽略锚点上的字。

```
/**
 * 计算一个节点下的单词数
 *
```

```

* @param iNode 开始计算的节点
* @return 单词数
*/
private int getNumWords(final Node iNode) {
    int words = 0;

    if (iNode.hasChildNodes()) {
        Node next = iNode.getFirstChild();

        while (next != null) {
            Node current = next;
            next = current.getNextSibling();

            //如果当前节点是一个链接，则不往下深入
            if (!isLink(current))
                words += getNumWords(current);
        }
    }

    //检查节点是文本节点还是元素节点
    int type = iNode.getNodeType();

    //文本节点
    if (type == Node.TEXT_NODE) {
        String content = iNode.getNodeValue();
        words += getHTMLLen(content);
    }

    return words;
}

```

计算一个文本中大概包括的正文实际长度的方法如下：

```

private int getHTMLLen(String text) {
    int len = 0;
    for(int i=0;i<text.length();++i) {
        if(text.charAt(i)==' ') {

```

```

    }
    else if(text.charAt(i)=='') {

    }
    else if(text.charAt(i)==' ') {

    }
    else if(text.charAt(i)=='&') {
        i+=5;
    } else {
        ++len;
    }
}
if( len<10)
    len = 0;
return len;
}

```

根据链接文字比删除无效节点的过程如下：

1. 计算节点下的链接数；
2. 计算节点下的文字数；
3. 计算节点的 链接文字比 = 节点下的链接数 / 节点下的文字数；
4. 如果节点的链接文字比大于某一个阈值则删除这个节点。

删除无效节点的实现代码如下：

```

/**
 * 如果链接比率合适则删除这个表单元
 * @param iNode 表单元节点
 */
public void testRemoveCell(final Node iNode) {
    //如果这个表单元没有孩子节点则不处理这个表单元
    if (!iNode.hasChildNodes())
        return;

    double links;//iNode 节点下的链接数

```

```

double words;//iNode 节点下的单词数

//计算链接和单词数
links = getNumLinks(iNode);
words = getNumWords(iNode);

//计算链接文字比并检查是否被 0 除
double ratio = 0;
if (words == 0)
    ratio = settings.linkTextRatio + 1;
else
    ratio = links / words;

if (ratio > settings.linkTextRatio) { //如果链接文字比大于指定值则删除该节点
    iNode.getParentNode().removeChild(iNode);
}
}

```

3.1.11 网页结构相似度计算

自动提取结构化信息的关键是从同样类型的实例中发现编码模版。为了确定两个网页是否由同一个网页模板生成，需要计算两个网页的结构相似度。一个自然的方法是从 HTML 编码字符串检测重复的模式。检测的方法有最长公共子序列和树编辑距离。

举例说明两个序列 s_1 和 s_2 的最长公共子序列(Longest Common Subsequence 简称 LCS)。 $s_1 = \{ a, b, c, b, d, a, b \}$, $s_2 = \{ b, d, c, a, b, a \}$, 则从前往后找, s_1 和 s_2 的最长公共子序列为 $LCS(s_1, s_2) = \{ b, c, b, a \}$, 如图 3-7 所示。

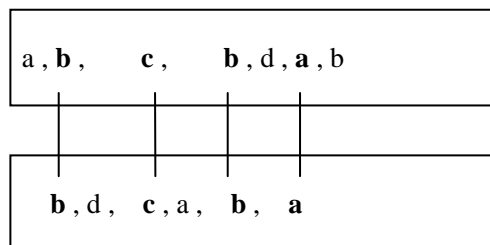


图 3-7 最长公共子序列

LCS 问题的最优解只取决于其子序列 LCS 问题的最优解，而非最优解对问题的求解没有影响。同时子序列 LCS 的值和当前对比字符的值一旦确定，则此后过程的 LCS 计算不再受此前各状态及决策的影响。因为满足动态规划的最优化原理和无后效性原则，因此使用动

态规划的思想计算 LCS 的方法：引进一个二维数组 `num[][]`，用 `num[i][j]` 记录 `s1` 的前 `i` 个长度的子串与 `s2` 的前 `j` 个长度的子串的 LCS 的长度。

自底向上进行递推计算，那么在计算 `num[i][j]` 之前，`num[i-1][j-1]`，`num[i-1][j]` 与 `num[i][j-1]` 均已计算出来。此时再根据 `s1[i-1]` 和 `s2[j-1]` 是否相等，就可以计算出 `num[i][j]`。

采用动态规划的方法计算两个序列的最长公共子序列的实现代码如下：

```
public static <E> List<E> longestCommonSubsequence(E[] s1, E[] s2){
    int[][] num = new int[s1.length+1][s2.length+1]; //二维数组

    //实际算法
    for (int i = 1; i <= s1.length; i++)
        for (int j = 1; j <= s2.length; j++)
            if (s1[i-1].equals(s2[j-1]))
                num[i][j] = 1 + num[i-1][j-1];
            else
                num[i][j] = Math.max(num[i-1][j], num[i][j-1]);

    System.out.println("length of LCS = " + num[s1.length][s2.length]);

    int s1position = s1.length, s2position = s2.length;
    List<E> result = new LinkedList<E>();

    while (s1position != 0 && s2position != 0) {
        if (s1[s1position - 1].equals(s2[s2position - 1])) {
            result.add(s1[s1position - 1]);
            s1position--;
            s2position--;
        }
        else if
            (num[s1position][s2position - 1] >= num[s1position - 1][s2position]) {
            s2position--;
        }
        else {
            s1position--;
        }
    }
    Collections.reverse(result);
}
```

```

    return result;
}

```

比较网页结构相似度的基本流程是：首先把网页抽象成一个 Node 数组，然后比较两个 Node 数组的最长公共子序列。输入两个 URL 地址，返回网页相似度的实现代码如下：

```

public static double getPageDistance(String urlStr1,String urlStr2) {
    ArrayList<Node> pageNodes1 = new ArrayList<Node>();//网页转换成 Node 数组

    URL url = new URL(urlStr1);
    Node node;
    Lexer lexer = new Lexer (url.openConnection ());
    lexer.setNodeFactory(new PrototypicalNodeFactory ());
    while (null != (node = lexer.nextNode ())) {
        pageNodes1.add(node);
    }

    ArrayList<Node> pageNodes2 = new ArrayList<Node>();
    URL url2 = new URL(urlStr2);

    lexer = new Lexer (url2.openConnection ());
    lexer.setNodeFactory(new PrototypicalNodeFactory ());
    while (null != (node = lexer.nextNode ())) {
        pageNodes2.add(node);
    }
    int lcs = longestCommonSubsequence (pageNodes1, pageNodes2);
    return lcs / (double)Math.min(pageNodes1.size(), pageNodes2.size()) ;
}

```

3.1.12 提取标题

有两种提取标题的方式：一种是利用锚点描述文字；还有一种是利用标题中的描述信息。例如：

<title>中国燃气总经理和执行总裁疑被警方带走_网易新闻中心</title>

<title>干部考核不宜唯“德孝”是举 - 第一视点 - 华声评论 - 华声在线</title>

在“-”或“_”等分隔符前面的文字是真正的标题。

网页链接中的锚点文本是对目标网页主题内容的概括,所以往往用它作为一个网页的标题描述。图 3-12 显示了来源于新华网的两个页面之间的对应关系:

[·发展中国家强调以技术合作应对气候变化](#) 20:51
[·经济适用房新政解读 重新回归“自身居住”功能](#) 17:13
[·“红岛”渔排上的“红色”演出](#) 19:51
[·“伊朗威胁论”的前世今生|美国的前后矛盾](#) 18:25
[·中国愿支援台湾发展航太技术](#) 16:15
[·陕西咸阳市大力推进农村节能减排](#) 21:13
[·财经观察: 纳斯达克为何而来](#) 17:26
[·坡头区开展“12·4”法制宣传日活动](#) 20:46
[·秦刚: 战略经济对话是中美合作的重要机制和平台](#) 17:52
[·“土地储备”首度立规 “土地融资”将受约束](#) 21:15

发展中国家强调以技术合作应对气候变化

2007年12月04日 20:51:05 来源: 新华网

【字号 大 中 小】 【留言】 【打印】 【关闭】 【Email推荐: 提交】

新华网印度尼西亚巴厘岛12月4日电(记者齐紫剑 林小春)《联合国气候变化框架公约》秘书处执行秘书德博埃尔4日在巴厘岛说,在[联合国](#)气候变化大会上,很多发展中国家强调应对气候变化的谈判应解决相关的技术合作与技术转让问题,要关注它们目前的“关切”。

图 3-12 网页标题和锚点文本的对应关系

先把列表页上的连接 url 对应的<a> 标签的“title”属性获取下来,称为列表页描述标题。详细页页面的标题属性称为内部标题。比较列表页描述标题和内部标题,估计出详细页的标题。由于这两个标题的标题字符串不是全部相同,例如列表页描述标题如果较长就可能以省略号结尾或者末尾截短字,而内部标题可能包含“_网站名”等冗余信息。所以可以考虑最长公共子串的方式比较两个标题进行相似度判断。

取得了网页标题以后,还可以利用标题信息计算网页的内容和标题之间的距离。

```
public static double getSimiarity(String title,String body){
    int matchNum = 0;
    for(int i=0;i<title.length();++i) {
        if(body.indexOf(title.charAt(i))>=0) {
            ++matchNum;
        }
    }
}
```

```
double score = (double)matchNum/( (double)title.length() );
return score;
}
```

可以对每个文本类型的节点根据内容分类。例如是标题、正文、或者来源，控制页面的文本或者广告文本、版权信息文本等。

把所有文本类型的节点收集到一个数组中。然后对每一个节点应用规则提取内容，并且组织成树型结构，最后再按节点解析成一个大的树型结构。

3.1.13 提取日期

经常需要从网页提取日期来判断网页的新旧程度等，可以用正则表达式提取。但日期格式很灵活，有很多种，不能用一个正则表达式模式完全覆盖，所以考虑直接写有限状态机提取日期。

很多新闻网站的新闻都是按发布日期分目录存放的，例如 <http://finance.sina.com.cn/g/20101226/09339163619.shtml> 是 2010 年 12 月 26 日的新闻。所以还可以从 URL 地址提取日期。

3.2 从非 HTML 文件中提取文本

在很多知识库系统中，为了查询大量积累下来的文档，需要从 PDF、Word、Rtf、Excel 和 PowerPoint 等格式的文档提取可以描述文档的文字。其中 Word、Rtf、Excel 和 PowerPoint 格式都来自微软。POI 项目(<http://poi.apache.org/>)是一个专门处理微软文档格式的开源项目，是一个纯 Java 的实现，可以在 Linux 平台运行。S1000D 是一种 XML 文档格式，主要用于航空航天和国防业(军品和民品)中的技术出版物。

有很多文件名的命名没有意义，比如说是类似“20090224153208138.pdf”由数字组成的文件名。在搜索结果中显示一个有意义的文件标题而不是文件名能够改进用户的搜索体验。这里首先从一般意义上来讨论从文件内容提取标题的方法。然后按文件类型分别具体实现提取标题。

设计一个通用的接口来处理待索引的文档。

```
//处理文档
public interface IFilter {
    String getTitle(File file);//返回标题
    String getBody(File file);//返回内容
    IDocument getDocument(File file);//返回全部索引信息
}
```


3.2.1 提取标题的一般方法

为了从正文比较合理的提取标题，设计提取标题流程如图 3-13:

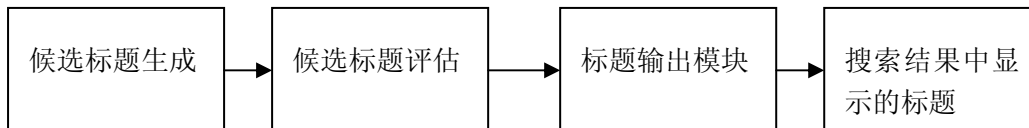


图 3-13 提取标题流程

- **候选标题生成模块：**首先从提取出来的原子文本组织成文档结构树。构建文档结构树既可以用自底向上的方法，从原子节点构造起，也可以用自顶向下的方法从根节点首先将文字划分成大的单元，然后逐步从大块文字细分。如果采用自底向上的方法，如果文字在同一行，并且字体、字体大小、颜色都一致，则视为不可拆分。然后根据字体、字体大小、颜色、位置等信息再次合并文字。从文档结构树给出几个可能的候选标题。
- **候选标题评估模块：**对每个候选标题，按照特征打分。可以把要用到的特征都作为候选标题类的属性。其中，最主要的特征是标题字符串，此外还有字符串所在位置，字体大小，字体颜色等信息。可以根据标题字符串对整个文章的概括程度和通顺性与意义完整性打分。从对文章的概括程度考虑，可以按照 $TF*IDF$ 等方法选取重要性较高的词作为关键词，然后根据关键词对每个候选标题字符串给出可能性权重。还可以比较候选标题字符串和首页中的其它文字，看候选标题相对其它文字的代表度或者说是相对其它文字的可替代性，也就是说候选标题对其它文字的覆盖度。从通顺性与意义完整性考虑，可以考虑准备一个标题语料库，提取出词法规则和作为标题常用的搭配规则，也可以对大量标题训练一个 **HMM** 模型。或者用信息提取的方法来评估候选标题字符串。
- **标题输出模块：**把权重最大的候选标题挑出来，按照可读的方式输出。

作为标题的文字长度往往既不是太长，也不会太短，大部分标题的长度在 10 到 30 个字之间。超长的标题往往会被搜索引擎截短，网站制作者为了优化搜索排名，也倾向于采用这样的标题长度。可以用 **sweetSpot** 来对候选标题的长度打分。

```

private static int ln_min = 10; //标题最短长度
private static int ln_max = 30; //标题最长长度
private static float ln_steep = 0.5f;

//如果长度在 10 到 30 之间，则 sweetSpotScore 返回值是 1，
//否则返回值会低于 1，返回值随长度降低的程度由 ln_steep 决定。
public static double sweetSpotScore(int length) {
    return (float) (1.0f /

```

```

Math.sqrt((ln_steep * (float) ( Math.abs(length - ln_min)
+ Math.abs(length - ln_max) - (ln_max - ln_min)) ) + 1.0f) );
}

```

ln_steep 这个值一般取值在 0 到 1 之间，ln_steep 越小，则返回值下降幅度越小。

政府公文有比较固定的格式，如图 3-14 所示。有“发文单位”，“文号”，“接收单位”等。例如上面那个文件“发文单位”是“合肥市物价局文件”，“文号”是“合价服〔2009〕219 号”，“接收单位”是“三县四区物价局”。

合肥市物价局文件

合价服〔2009〕219 号

提取为标题

关于降低合肥市税务师事务所 服务收费项目和标准的通知

三县四区物价局：

根据市政府办公厅转发《省政府办公厅转发省物价局关于进

图 3-14 政府公文

判断是否发文机关：

```

public static boolean isProSendGovUnit(String s,Color tcolor){
    double pro = 0;

    if(tcolor.equals(Color.RED))    {
        pro+=0.2;
    }

    if(s.endsWith("文件")){

```

```

        pro+=0.4;
    }

    int numBlank = 0;
    int numGovUnit = 0;
    for(int i=0; i<s.length(); i++){
        char c = s.charAt(i);
        if(c == ' '){
            numBlank ++;
        }
        else if(c == '厅' ||
                c == '所' ||
                c == '局' ||
                c == '会' ||
                c == '委' ||
                c == '府')
            numGovUnit++;
    }
    if(numBlank > 0) {
        pro
(0.6*((double)numGovUnit+(double)numBlank)/((double)numBlank+s.length())); +=
    }
    if(pro > 0.6)
        return true;
    return false;
}

```

判断是否收文机关：

```

public static boolean isProReceiGovUnit(String s,Color tcolor){
    if(!tcolor.equals(Color.BLACK)) {
        return false;
    }

    float pro =0 ;
    if(s.endsWith(": ") || s.endsWith(":")){
        pro += 0.5;
    }
}

```

```

for(int i=0; i<s.length(); i++){
    char c = s.charAt(i);
    if(c == '\ ' || c == ' (' || c == ') ' || c == ', '){
        pro +=0.2;
    }
    else if(c == '厅' ||
            c == '部' ||
            c == '所' ||
            c == '局' ||
            c == '会' ||
            c == '委' ||
            c == '县' ||
            c == '区' ||
            c == '市' ||
            c == '省')
        pro +=0.11;
    }
    pro = pro/s.length();
    if(pro >= 0.07)
        return true;
    return false;
}

```

判断文号，例如“川国税发（2007）065 号”：

```

public static boolean isFileNum(String s,Color tcolor){
    if(!tcolor.equals(Color.BLACK)) {
        return false;
    }
    boolean isSymbol = false;

    //如果碰到开始符号，则匹配对应的结束符号
    for(int index=0; index<s.length(); index++){
        if('[' == s.charAt(index)){
            isSymbol = matchSym(s,index,']');
        }
        else if(' (' == s.charAt(index)) {
            isSymbol = matchSym(s,index,') ');
        }
    }
}

```

```

    }
    else if ('(' == s.charAt(index)) {
        isSymbol = matchSym(s,index,' ');
    }
}

return isSymbol;
}

//如果匹配上指定字符则成功退出
public static boolean matchSym(String s,int index,char endMark){
    boolean isSymbol = false;

    boolean markBegSym = true;
    boolean markEndSym = false;
    for(int i =index+1; i<s.length(); i++) {
        if(!markEndSym) {
            if((s.charAt(i)>='0'&& s.charAt(i)<='9')||s.charAt(i) == ' '
                ||(s.charAt(i)>='0' && s.charAt(i)<='9 '))
            {}
            else if(s.charAt(i) == endMark) {
                markEndSym = true;
                i++;
            }
            else {
                markBegSym = false;
                markEndSym = false;
            }
        }
        if(markBegSym && markEndSym) {
            if((s.charAt(i)>='0'&& s.charAt(i)<='9')||s.charAt(i) == ' '
                ||(s.charAt(i)>='0' && s.charAt(i)<='9 '))
            {}
            else if(s.charAt(i) == '号') {
                isSymbol = true;
                index = s.length();
            }
        }
    }
}

```

```

    }
}

return isSymbol;
}

```

3.2.2 PDF 文件

PDF 是 Adobe 公司开发的电子文件格式。这种文件格式与操作系统的平台无关，可以在多数操作系统上通用。我们经常会遇到这种情况，就是想把 PDF 文件中的文字复制下来，可是会发现经常不能复制，因为这个 PDF 文件的内容可能加密了。那么怎么把 PDF 文件中的内容提取出来？这就是我们这一节要解决的问题。现在已经有很多工具可以让我们利用来完成这项任务了，PDFBox(<http://pdfbox.apache.org/>)就是专门用来解析 PDF 文件的 Java 项目。

下载文件 pdfbox-app-1.5.0.jar，然后在 Eclipse 项目中引入这个文件。提取文本只需要三行代码：

```

//加载 fileName 代表的 PDF 文件
PDDocument doc = PDDocument.load(fileName);
//用 PDFTextStripper 提取文本
PDFTextStripper stripper = new PDFTextStripper();
//返回提取的文本字符串
String content = stripper.getText(doc);

```

这样可以抽取 PDF 文件中的文本，但是对于 PDF 文件中的图片 PDFBox 是无能为力的。对于包含文字的图片，需要借助 OCR 软件从图片中识别出文字。

需要说明的是这个版本支持中文。使用早期版本的 PDFBox 抽取中文 PDF 文本时，可能会遇到 Unknown encoding for 'GBK-EUC-H'错误。因为 PdfReader 内部缺省支持一些中文字体，但 PDFBox 内部却没有对'GBK-EUC-H'字体的支持。

因为 PDF 文件中有可能存在重叠位置的文本，PDFTextStripper 有对于重叠位置文本的判断。如果两个 TextPosition 位于差不多同样的位置，则不重复抽取重叠的 TextPosition 代表的文字。

```

private boolean overlap( TextPosition tp1, TextPosition tp2 ){
    if(!within(tp1.getHeight(),tp2.getHeight(), 1.1f))
        return false;

    float diff = (tp1.getHeight() + tp2.getHeight())*0.02f;
    return within( tp1.getX(), tp2.getX(), diff) && within( tp1.getY(), tp2.getY(), diff) ;
}

```

在提取 PDF 文件中可搜索的文字的问题解决后，下一个问题是提取标题。例如，从 Google 搜索 “filetype:pdf” 可以看到 Google 提取的 PDF 文件标题。

如果 PDF 元数据中已经存储了文档标题，可以通过元数据取得文档标题：

```
//取得文档元数据
PDDocumentInformation info = document.getDocumentInformation();
this.title = info.getTitle();
```

但是很多 PDF 文件中元数据并没有存储任何内容。所以在大多数情况下，仍然需要从内容中分析出标题。

可以利用文本的位置或字体等信息帮助选取标题。例如在首页字体最大的文字有可能是 PDF 文件的标题。可以通过 `TextPosition` 对象的 `getFontSizeInPt` 方法返回文字字体大小。

为了取得文本的位置信息和颜色等，需要更加深入的了解 `PDFBox` 的运行原理。`PDF` 规范可以从 http://www.adobe.com/devnet/pdf/pdf_reference.html 下载。`PDF` 内容流中包含许多操作符(`Operator`)。在 `PDFBox` 中，每种操作符都有专门对应的处理类，记录在配置文件 `Resources/PageDrawer.properties` 中。例如其中的一行：

```
BT=org.apache.pdfbox.util.operator.BeginText
```

表示 `BT` 操作的处理类是 `org.apache.pdfbox.util.operator.BeginText`

`PDFBox` 中的应用程序 `PageDrawer` 实现输出 `PDF` 文件到可视化窗口。`PageDrawer` 有完整的对于文字显示方面的处理。为了提取标题，可以通过在 `PageDrawer` 的实现基础上，简化一些与 `Path` 和 `Line` 相关的操作符的处理，只把位置信息和颜色等信息提取出来。

文字的颜色信息并不能直接得到，而是依赖于上下文的。`PageDrawer` 类包含了对图像的处理。

```
if( this.getGraphicsState().getTextState().getRenderingMode() ==
    PDTextState.RENDERING_MODE_FILL_TEXT ){
    graphics.setColor( this.getGraphicsState().getNonStrokingColorSpace().createColor() );
}
else if( this.getGraphicsState().getTextState().getRenderingMode() ==
    PDTextState.RENDERING_MODE_STROKE_TEXT ) {
    graphics.setColor( this.getGraphicsState().getStrokingColorSpace().createColor() );
}
```

用 `PdfTitle` 表示候选标题。

```
public class PdfTitle {
    public String title = null; //标题文字
```

```

public Color textColor = null; //文字颜色
public float fontSize; //字体大小
public float y; //高度
ArrayList<TextPosition> texts; //包含的文本

public PdfTitle(String t,float h) {
    title = t;
    y=h;
}

public PdfTitle(Color c,float f,float h,ArrayList<TextPosition> text) {
    title = delOverlap(text);
    textColor = c;
    fontSize = f;
    y = h;
    texts = text;
}
}

```

候选标题单独组成一个段落，或者位于一个段落内部。通过 **PDFDegger** 可以看到 PDF 文件的树形结构组织。因为通过 **PDFStreamEngine** 类导出的文字块没有明显的段落信息，所以需要编写程序寻找段落边界。大的字体差别和颜色差别以及文本的垂直位置都可以用来确定是一个独立段落的开始，但有些相对模糊的边界需要更细致的判断。因此把 **PdfTitle** 设计成可以再次拆分的单元。

1. 根据文本的垂直位置寻找该段落的最大垂直区隔；
2. 根据最大垂直区隔拆分 **PdfTitle**；
3. 如果没有找到合适的标题，对于拆分出来的新的 **PdfTitle** 重新应用 1,2 步骤直到不可再拆分或找到合适的标题为止。

可以通过扩展 **org.pdfbox.util.PDFTextStripper** 类，覆盖 **processTextPosition** 方法来遍历 PDF 文件中的 **TextPosition** 对象。下面的代码简单的提取最大字体的文字作为标题。

```

float currentFontSize = text.getFontSizeInPt();
if(currentFontSize < biggestFontSize){ //字体不一致，则不是标题文字
    consistent = false;
} else if (currentFontSize > biggestFontSize){ //字体最大，是标题文字
    titleGuess = text.getCharacter();
    biggestFontSize = currentFontSize;
}

```



```

        consistent = true;
    } else if(currentFontSize == biggestFontSize
        && consistent
        && !"".equals(text.getCharacter().trim())){
        //字体和以前文字的最大字体一样大，是标题文字
        titleGuess += text.getCharacter();
    }
}

```

PDF 文件分成两类：一类首页没有正文，文字比较少，文字比较多的部分可能是标题。还有一类，首页有正文，挨着正文往上的可能是标题，也可能是段落标题。首页没有正文的情况，不计算标题对于首页正文的概括性。

政府公文中往往包含一些主题词，可以利用主题词或者文章内容来判断标题。或者建立一个标题的语料库，例如很多标题都是采用“关于**的通知”这样的形式，可以利用模版匹配的方式来保证标题的完整性。

3.2.3 Word 文件

Word 是微软公司开发的字处理文件格式，以“doc”或者“docx”作为文件后缀名。Apache 的 POI(<http://poi.apache.org/>)可以用来在 Windows 或 Linux 平台下提取 Word 文档。用 POI 提取文本的基本方法如下：

```

public static String readDoc(InputStream is) throws IOException{
    //创建 WordExtractor
    WordExtractor extractor=new WordExtractor(is);
    // 对 DOC 文件进行提取
    return extractor.getText();
}

```

为了提取 Word 文档的标题，需要深入了解 POI 接口。一个 Word 文档包含一个或者多个 Section，每个 Section 下面包含一个或者多个 Paragraph，每个 Paragraph 下面包含一个或者多个 CharacterRun。

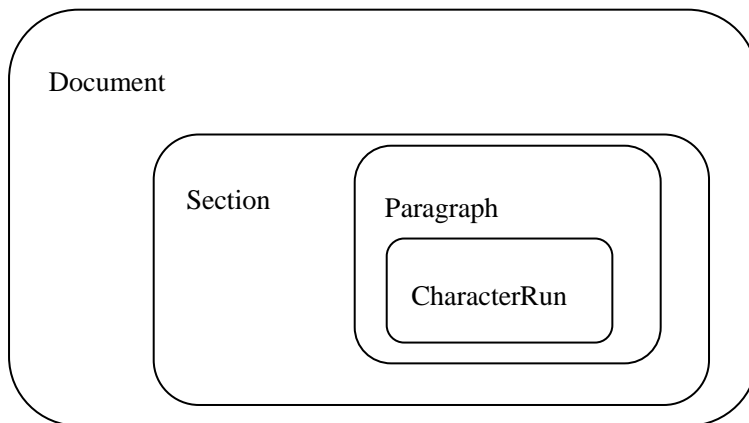


图 3-16 Word 文档结构图

遍历 Word 文档结构的代码如下：

```
Range r = doc.getRange();

for (int x = 0; x < r.numSections(); x++) {
    Section s = r.getSection(x);
    for (int y = 0; y < s.numParagraphs(); y++) {
        Paragraph p = s.getParagraph(y);
        for (int z = 0; z < p.numCharacterRuns(); z++) {
            CharacterRun run = p.getCharacterRun(z);
            //字符串文本
            String text = run.text();
            System.out.println(text);
        }
    }
}
```

标题往往是居中对齐的。可以通过 Paragraph 的 getJustification 方法得到段落的对齐方式。getJustification 的返回值 0 表示左对齐，1 表示居中对齐，2 表示右对齐，3 表示两端对齐。

可以通过 CharacterRun 对象取得文字内容、字体大小、文字颜色等信息。

```
run.text();    //文字内容
run.getFontSize(); //字体大小
run.getColor(); //文字颜色
```

3.2.4 Rtf 文件

在 1992 年，微软公司为了定义简单的格式化的文本和嵌入的图片引入了富文本格式 (RTF)。最开始是为了在不同的操作系统(例如 MS-DOS, Windows 和 OS/2 以及苹果的 Macintosh)上的不同的应用程序之间转移数据，现在这种格式已经广泛用于 Windows 系统，因为它可以在 RichTextBox 控件中编辑。

Rtf 的版本从 1.0 到 1.9。Rtf 是 8 位格式的，为了方便传输，标准的 Rtf 文件只由 ASCII 字符组成，中文字符用转义符来表示。每个 Rtf 文件都是一个文本文件。文件开始处是{\rtf，它作为 Rtf 文件的标志是必不可少的，Rtf 阅读器根据它来判断一个文件是否为 Rtf 格式。然后是文件头和正文，文件头包括字体表、文件表、颜色表等几个数据结构，正文中的字体、表格的风格就是根据文件头的信息来格式化的。每个表用一对大括号括起来，其中包含很多用字符“\”开始的命令。举个最简单的 Rtf 文件的例子，文件中只包含一行文字：{\rtf1foobar}。用写字板打开这个文件，显示：“foobar”。

许多开源的 Rtf 文件解析器不能正确处理多字节编码内容，例如：javax.swing.text.rtf。当然有的项目也能够处理包含 Unicode 编码的 Rtf 文件，例如 RtfConverter(下载地址是 <http://www.codeproject.com/KB/recipes/RtfConverter.aspx>)，不过这个项目是 c#实现的，下节介绍如何用 Java 实现一个 RTF 文件解析器 RtfConverter4J。

Rtf 文件解析器 RtfConverter4J 的设计目标是：

- 可以在各层次上分析 Rtf 数据。
- 把对 Rtf 数据的解析和解释分开。
- 保持解析器和解释器的可扩展性。
- Rtf 转换应用程序容易上手。
- 采用开放式架构，能够很容易自定义 Rtf 转换器。

因此，设计了如图 3.5 所示的这个开放式架构。

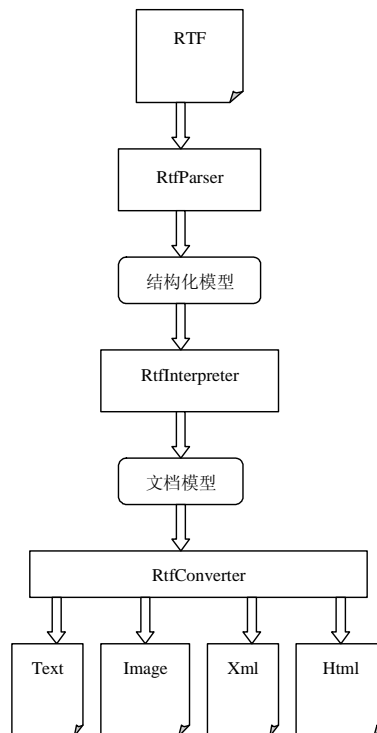


图 3.5 Rtf 转换程序总体结构图

RtfParser 类用于完成实际的数据解析。除了识别标签(Tag)，也处理字符编码，支持 Unicode。RtfParser 把 RTF 数据分成如下几种基本的元素：

- RtfGroup: Rtf 元素的集合。
- RtfTag: 一个 Rtf 标签的名字和值对。
- RtfText: 任意的文本内容，但是文本内容不一定是可见的。

在 RtfParser 中对 Unicode 编码的处理代码如下：

```
if (tagName.equals(RtfSpec.TagUnicodeCode)) {  
    //读入 Unicode 编码的字符的值  
    int unicodeValue = tag.getValueAsNumber();  
    char unicodeChar = (char) unicodeValue;  
    this.curText.append(unicodeChar);  
    for (int i = 0; i < this.unicodeSkipCount; i++) {  
        // 跳过指定数量的文本  
        char skip1 = (char) reader.read();  
        if (skip1 == ' '){  
            char skip2 = (char) PeekNextChar(reader, false);  
            if (skip2 == '\\') {  
                reader.read();  
                char skip3 = (char) PeekNextChar(reader, false);  
                while (skip3 != '\\' && skip3 != '}' && skip3 != '{') {
```

```

        reader.read();
        skip3 = (char) PeekNextChar(reader, false);
    }
}
} else if (skip1 == '\\') {
    reader.read();
    char skip3 = (char) PeekNextChar(reader, false);
    while (skip3 != '\\' && skip3 != '}' && skip3 != '{') {
        reader.read();
        skip3 = (char) PeekNextChar(reader, false);
    }
} else if (skip1 == '\r') {
    reader.read();
    char skip2 = (char) PeekNextChar(reader, false);

    if (skip2 == '\\') {
        reader.read();
        char skip3 = (char) PeekNextChar(reader, false);
        while (skip3 != '\\' && skip3 != '}' && skip3 != '{') {
            reader.read();
            // System.Console.WriteLine((char)reader.Read());
            skip3 = (char) PeekNextChar(reader, false);
        }
    }
}
}
} else if (tagName.Equals(RtfSpec.TagUnicodeSkipCount)) {
    //读入 UnicodeSkipCount 的值
    int newSkipCount = tag.GetValueAsNumber();
    if (newSkipCount < 0 || newSkipCount > 10) {
        throw new Exception("invalid unicode skip count: " + tag);
    }
    this.unicodeSkipCount = newSkipCount;
}
}

```

因为 Rtf 数据中可能包含另外一种十六进制表示的 Unicode，所以增加了对 TagUnicodeSkipCount 的处理。

Rtf 文件解析器的结构如图 5.6 所示。实际的解析过程可以通过 `ParserListener` 监听。`ParserListener` 通过观察者模式提供了对某个事件反应的机会并执行相应的动作。系统已经集成的解析器监听器 `RtfParserListenerFileLogger` 可以用来把 Rtf 元素的结构写入日志文件 (主要用在开发阶段的调试)。输出可以通过 `RtfParserLoggerSettings` 配置。

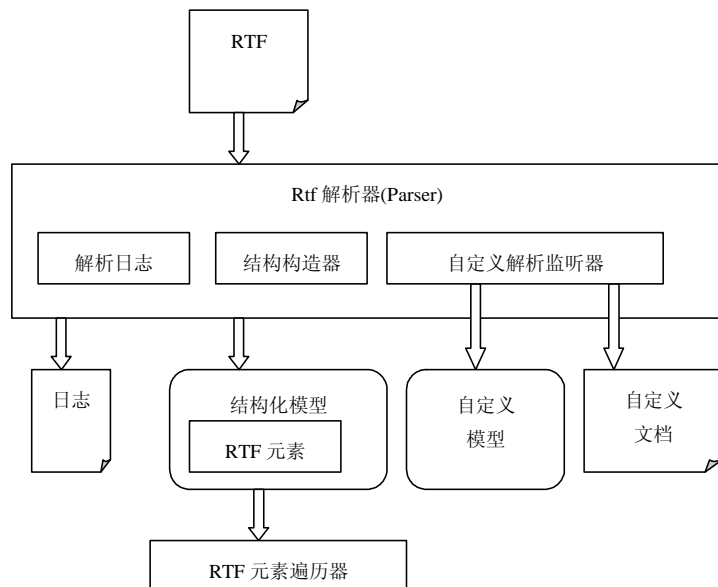


图 5.6 RTF 解析器结构图

`RtfParserListenerStructureBuilder` 根据解析过程中碰到的 `rtf` 元素生成结构化模型。这个模型把基本元素表示成 `IrtfGroup`、`IrtfTag` 和 `IrtfText` 的实例。可以通过 `RtfParserListener-StructureBuilder.StructureRoot` 取得层次结构。

当 Rtf 文档解析成结构化模型后，就可以通过 Rtf 解释器来解释。Rtf 文件解释器的结构如图 5.7 所示。解释结构化模型的一个方法是构造文档模型来提供对文档内容意义的高层次抽象。一个简单的文档模型由如下块组成：文档信息：标题、主题和作者等；用户属性；颜色信息；字体信息；文本格式；可视化信息。其中，可视化信息包括：

- 文本相关的格式化信息。
- 分隔符。线，段落，节，页。
- 特殊字符。制表符、段落开始/结束、下划线、空格、圆点、引号、连字符。
- 图片。

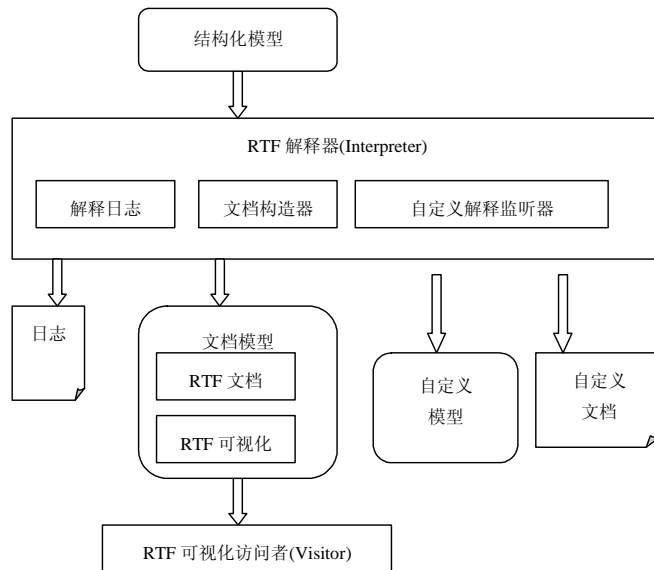


图 5.7 Rtf 解释器结构图

下面的例子展示了如何使用文档模型的高层 API:

//显示 Rtf 文件内容

```
public static void RtfWriteDocumentModel(String rtfStream) throws Exception {
    RtfInterpreterListenerFileLogger logger = null;
    IRtfDocument document = RtfInterpreterTool.BuildDoc(rtfStream, logger);
    RtfWriteDocument(document);
} // RtfWriteDocumentModel
```

//根据文档模型遍历

```
public static void RtfWriteDocument(IRtfDocument document) {
    System.out.println("RTF Version: " + document.getRtfVersion());

    // 显示文档信息
    System.out.println("Title: " + document.getDocumentInfo().getTitle());
    System.out.println("Subject: "
        + document.getDocumentInfo().getSubject());
    System.out.println("Author: " + document.getDocumentInfo().getAuthor());

    //显示字体表中的字体
    for (String fontName : document.getFontTable().keySet()) {
        System.out.println("Font: " + fontName);
    }
}
```

```

//显示颜色表中的颜色
for (IRtfColor color : document.getColorTable()) {
    System.out.println("Color: " + color.getAsDrawingColor());
}

//取得文档的用户属性，例如文档“创建者”或者“时间”
for (IRtfDocumentProperty documentProperty : document
    .getUserProperties()) {
    System.out.println("User property: " + documentProperty.getName());
}

//遍历所有可视化元素
for (IRtfVisual visual : document.getVisualContent()) {
    RtfVisualKind visualKind = visual.getKind();
    if (visualKind == RtfVisualKind.Text) { //文本
        System.out.println("Text: "
            + ((IRtfVisualText) visual).getText());
    } else if (visualKind == RtfVisualKind.Break) { //换行符号
        System.out.println("Tag: "
            + ((IRtfVisualBreak) visual).
                getBreakKind().toString());
    } else if (visualKind == RtfVisualKind.Special) { //特别字符
        System.out.println("Text: "
            + ((IRtfVisualSpecialChar) visual).getCharKind()
                .toString());
    } else if (visualKind == RtfVisualKind.Image) { //图像
        IRtfVisualImage image = (IRtfVisualImage) visual;
        System.out.println("Image: " + image.getFormat().toString()
            + " " + image.getWidth() + "x" + image.getHeight());
    }
}
}

```

下面调用 RtfConverter4J 提取文本：

```

URL u = new URL("http://www.silo.net/LaReja-2005-05-07/texto_chino_7M.rtf");
// 创建一个 URL 连接对象
URLConnection uc = u.openConnection();

```



```
// 提取内容并输出
InputStream is = uc.getInputStream();
RtfExtractor extractor=new RtfExtractor(is);
String text = extractor.getText();
is.close();
System.out.println("text:"+text);
```

提取 Rtf 文件的标题设计流程如下：

- (1) 生成候选标题：把字体最大的文字块或者居中对齐的文字块作为候选标题。
- (2) 评估候选标题：根据候选标题所在的位置和候选标题与正文的相似度来评分。
- (3) 输出标题模块：选择分值最大的标题输出。

首先定义候选标题类，代码如下：

```
public static class TitleInfo {
    public String fontName; //字体名
    public int fontSize; //字体大小
    public int position; //位置信息
    public int mergeTo; //合并块编号
    public boolean isBold; //是否粗体
    public String text; //内容
    public HashMap<String, Double> words; //内容切分结果
    public double weight; //权重
}
```

提取标题整体流程实现代码如下：

```
public static String getRtfTitle(String fileName) throws Exception {
    StringBuffer content = new StringBuffer();//用来存储全文内容
    //取得候选标题
    ArrayList<TitleInfo> candidates=getCandidates(fileName,content);

    if (candidates == null || candidates.size() == 0)// 没有候选标题
        return "";
    else if (candidates.size() == 1)
        return candidates.get(0).text;
    else {
        //候选标题评分
```

```

        rankTitle(candidates, content.toString());
        //把评分最高的候选标题作为标题提取结果
        return getBestTitle(candidates);
    }
}

```

取得候选标题部分实现代码：

```

public static ArrayList<TitleInfo> getCandidates(String fileName,
                                                StringBuffer content) {
    IRtfGroup rtfStructure = ParseRtf(fileName);// 获取 rtf 结构

    RtfInterpreterListenerDocumentBuilder docBuilder =
        new RtfInterpreterListenerDocumentBuilder(); // 初始化
    RtfInterpreterListenerLogger interpreterLogger = null;

    RtfInterpreterTool.Interpret(rtfStructure, interpreterLogger, docBuilder);
    RtfDocument doc = (RtfDocument) docBuilder.getDocument();

    if (doc == null)//读取失败
        return null;
    ArrayList<IRtfVisual> rtfVisuals = doc.getVisualContent();

    int maxFontSize = 0;
    int maxPosition = 0;
    for (IRtfVisual rtfv : rtfVisuals) {// 找最大字体
        if (rtfv.getKind() == RtfVisualKind.Text) {
            int currentFontSize = ((IRtfVisualText) rtfv).getFormat().getFontSize();
            maxFontSize = Math.max(currentFontSize, maxFontSize);
        }
    }

    ArrayList<TitleInfo> candidates = new ArrayList<TitleInfo>();
    for (int i = 0; i < rtfVisuals.size(); i++) {
        IRtfVisual rtfv = rtfVisuals.get(i);

        if (RtfVisualKind.Text == rtfv.getKind())//只有 Text 才有文字属性
        {

```

```

// 换行前以第一种字符格式作为整行格式，除了字体大小。
String fontName = ((IRtfVisualText) (rtfv)).getFormat()
    .getFont().getName();// 首字符字体
Color tc = ((IRtfVisualText) (rtfv)).getFormat()
    .getForegroundColor().getAsDrawingColor();
boolean isBold = ((IRtfVisualText) (rtfv)).getFormat()
    .getIsBold();
RtfTextAlignment alignment = ((IRtfVisualText) (rtfv))
    .getFormat().getAlignment();
int fontSize = ((IRtfVisualText) (rtfv)).getFormat()
    .getFontSize();

String oneRow = "";
while (RtfVisualKind.Break != rtfv.getKind()) { // 换行前的部分是一个整体
    if(rtfv.getKind()==RtfVisualKind.Text)
        oneRow += ((IRtfVisualText)rtfv).getText();

    i++;
    rtfv = rtfVisuals.get(i);
}

//公文属性判断
if (isProSendGovUnit(oneRow, tc) || //发文单位
    isProReceiGovUnit(oneRow, tc) || //收文单位
    isFileNum(oneRow)) { //文号
    maxPosition++;
    continue;
}

if (RtfTextAlignment.Center == alignment
    || fontSize == maxFontSize) { // 居中、最大字体加入候选标题
    candidates.add(
        new TitleInfo(oneRow,
            fontSize,
            maxPosition,
            fontName,
            isBold));
    } else

```

```

        // 不居中的是正文内容
        content.append(oneRow + " ");
    }
    maxPosition++;
}

return candidates;
}

```

对候选标题评分的代码如下：

```

public static void rankTitle(ArrayList<TitleInfo> titles, String content) {
    HashSet<String> stopWords = StopSet.getInstance();//停用词表
    HashMap<String, Double> contentWords = new HashMap<String, Double>();

    int maxLength = 0;
    int maxFontSize = 0;
    int width = 440;// 正文宽度
    int firstPosition = 9999;

    for (int i = 0, j = 1; j < titles.size(); i++, j++) { // 第一步：合并可能的标题
        TitleInfo ti = titles.get(i);
        TitleInfo tj = titles.get(j);
        firstPosition = Math.min(firstPosition, ti.position);
        if (ti.fontSize == tj.fontSize // 字体大小一致
            && ti.position + 1 == tj.position // 上下连贯
            && titles.get(i).text.length() > 1
            && titles.get(j).text.length() > 1) {
            if (!(tj.text.startsWith(" "))) {
                TitleInfo tTmp = ti;
                tj.mergeTo = i;
                while (tTmp.mergeTo != -1) {
                    tj.mergeTo = tTmp.mergeTo;
                    tTmp = titles.get(tTmp.mergeTo);
                }
                // 向前合并标题同时删除；
                titles.get(tj.mergeTo).text = titles.get(tj.mergeTo).text
                    .trim()
            }
        }
    }
}

```

```

        + tj.text.trim());
    tj.text = " ";
    if (ti.fontSize * ti.text.length() > width) { //因为字符过多而换行的
        width *= 2;
        titles.get(tj.mergeTo).weight += 0.2;
    }
}
}
}

for (TitleInfo m : titles) { // 第二步：分词，确定候选标题的长度与位置范围
    if (m.text.trim().length() >= 2) { // 非空标题分词
        maxLength = Math.max(maxLength, m.text.length());
        maxFontSize = Math.max(maxFontSize, m.fontSize);

        // 标题分词，建向量
        ArrayList<CnToken> taggedTitle = Tagger.getFormatSegResult(m.text);

        m.words = new HashMap<String, Double>();
        for (CnToken ct : taggedTitle) {
            if ("m".equals(ct.type()) || "t".equals(ct.type())
                || stopWords.contains(ct.termText()))
                continue; // 去除停用词

            Double val = m.words.get(ct.termText());
            if (val != null) {
                m.words.put(ct.termText(), new Double(val + 1.0));
            } else
                m.words.put(ct.termText(), new Double(1.0));
        }
    }
    m.position -= firstPosition;
}

// 对正文分词，建向量
ArrayList<CnToken> taggedContent = Tagger.getFormatSegResult(content);
for (CnToken ct : taggedContent) {
    if ("w".equals(ct.type()) || "m".equals(ct.type()))

```

```

        || "t".equals(ct.type()))
        || stopWords.contains(ct.termText()))
        continue;// 去除停用词

    if (contentWords.containsKey(ct.termText())) {
        contentWords.put(ct.termText(), new Double(contentWords.get(ct
            .termText()) + 1));
    } else
        contentWords.put(ct.termText(), new Double(1.0));
}
double contentNorm = calculateNorm(contentWords);

for (int i = 0; i < titles.size(); i++){// 第三步：综合评价候选标题权重
    TitleInfo t = titles.get(i);
    if (t.text.trim().length() >= 2) {
        double lengthWeight = getLengthWeight(t.text.length());
        double fontSizeWeight = getFontSizeWeight(t.fontSize,
            maxFontSize);
        double positionWeight = getPositionWeight(t.position);
        //计算可选标题与全文的相似度，用夹角余弦来衡量相似度
        double semanticWeight = getSimilarity(t.words, contentWords,
            contentNorm);

        //计算综合权重
        Double compositiveWeight = new Double(t.weight
            * Math.pow(lengthWeight * fontSizeWeight
                * positionWeight, 1.0 / 3) * semanticWeight);
        //得出标题的最终权重
        if (t.isBold)
            t.weight = compositiveWeight * 1.1;
        else
            t.weight = compositiveWeight;
    }
}
}
}

```

最后简单地取最大分值对应的标题：

```

public static String getBestTitle(ArrayList<TitleInfo> titles) {
    double max = 0; //记录最大分值
    String bestTitle = null; //记录最好标题
    for (TitleInfo t : titles) {
        if (t.weight > max && t.text.trim().length() >= 2) {
            max = t.weight;
            bestTitle = t.text;
        }
    }
    return bestTitle;
}

```

3.2.5 Excel 文件

Excel 文件由一个工作簿(Workbook)组成。工作簿由一个或多个工作表(Sheet)组成，每个工作表都有自己的名称。每个工作表又包含多个单元格(Cell)。除了 POI 项目，还有开源项目 jxl (<http://www.andykhan.com/jexcelapi/index.html>)可以用来读写 Excel 文件。

调用 Apache 的 POI 提取文本的代码如下所示：

```

public static String readDoc(InputStream is) throws IOException{
    //读入 Excel 文件数据流
    ExcelExtractor extractor = new ExcelExtractor(new POIFSFileSystem(is));
    //不返回公式
    extractor.setFormulasNotResults(true);
    //不包括 Sheet 名称
    extractor.setIncludeSheetNames(false);
    return extractor.getText();
}

```

为了提取 Excel 文件的标题，首先从每个工作表找最可能的标题，然后从多个工作表中再次挑选最可能的标题。

首先定义封装单元格属性的类，其中包含了用来计算标题重要度的一些属性：

```

public class CellInfo{
    public String text;//文本内容
    public short fontSize;//字体大小
    public short alignment;//对齐方式
    public boolean boldness;//是否黑体
}

```

```

public int rowPos;//所在行的位置
public double weight;//重要度
public boolean isUnique;//独立成行
public CellInfo(String t, short fs, int rp, short align, boolean bold){
    text = t;
    fontSize = fs;
    rowPos = rp;
    alignment = align;
    boldness = bold;
    weight = 1.0;
    isUnique = false;
}
}

```

通过遍历工作表中的每个字符型单元格来取得每个工作表的最好标题：

```

private TitleInfo getSheetBestTitle(HSSFSheet sheet, HSSFWorkbook wb){
    Iterator<HSSFRow> riter = sheet.rowIterator();//按行遍历工作表
    ArrayList<CellInfo> titles = new ArrayList<CellInfo>();

    int maxFontSize = 0;
    int rowCount = 0;
    while (riter.hasNext()) {
        rowCount ++;
        int columnCount = 0;
        HSSFRow row = (HSSFRow) riter.next();//按行遍历
        Iterator<HSSFCell> citer = row.cellIterator();

        while(citer.hasNext()) {
            HSSFCell cell = citer.next();//每行再按列遍历
            int cellType = cell.getCellType();
            HSSFCellStyle cellStyle = cell.getCellStyle();

            if (cellType != HSSFCell.CELL_TYPE_BLANK ) { //非空
                columnCount ++;
            }
            if (cellType == HSSFCell.CELL_TYPE_STRING) { //字符型
                String cellString = cell.toString().trim();//取得单元格内的文本
            }
        }
    }
}

```



```

        if (cellString.length() >= 2) {
            HSSFFont cellFont = cell.getCellStyle().getFont(wb);

            short fontheight = cellFont.getFontHeight();
            short al = cellStyle.getAlignment();
            short boldness = cellFont.getBoldweight() ;
            maxFontSize = Math.max(maxFontSize, (int)fontheight);
            CellInfo ci = new CellInfo(cellString,
                                      fontheight,
                                      rowCount,
                                      al,
                                      boldness == HSSFFont.BOLDWEIGHT_BOLD);

            titles.add(ci);
        }
    }
}

if (columnCount == 1) //这行只有这个
    titles.get(titles.size() - 1).isUnique = true;
}

if (titles.size() == 0)
    return new TitleInf("", 0);
else
    return selectBestTitle(titles, maxFontSize, rowCount);
}

```

标题类包含了标题的文本内容和重要度：

```

public class TitleInf{
    public String text;//文本内容
    public double weight;//重要度
    public TitleInf(String t, double w){
        text = t;
        weight = w;
    }
}

```

取得整个 Excel 文件的最可能的标题：

```

public String getTitle(String fileName) throws Exception{

```

```

InputStream is = new FileInputStream(fileName);
HSSFWorkbook wb = new HSSFWorkbook(new POIFSFileSystem(is));
ArrayList<TitleInf> candidate = new ArrayList<TitleInf>(); //候选标题
int activeSheetIndex = wb.getActiveSheetIndex(); //取得活跃工作表的编号

int sheetsNum = wb.getNumberOfSheets();
for (int i = 0 ; i < sheetsNum ; i++){ //取得每个工作表的最可能标题
    TitleInf bti = getSheetBestTitle(wb.getSheetAt(i), wb);
    if (i == activeSheetIndex)
        bti.weight *= 3.0;
    candidate.add(bti);
}

//取得最评分最高的候选标题
double maxWeight = 0;
String bestTitle = null;
for (TitleInf curTitle : candidate) {
    if (curTitle.weight >= maxWeight){
        maxWeight = curTitle.weight;
        bestTitle = curTitle.text;
    }
}

is.close();
return bestTitle;
}

```

3.2.6 PowerPoint 文件

在 PowerPoint 视图编辑区的上半部分显示幻灯片的缩略图，下半部分是备注编辑区。所以可提取的文本包括幻灯片显示的文本和备注中的文本。调用 Apache 的 POI 提取 PowerPoint 文件中的文本代码如下所示：

```

public static String readDoc(InputStream is) throws IOException{
    //创建 PowerPointExtractor
    PowerPointExtractor extractor=new PowerPointExtractor(is);
    //取得所有的幻灯片文本，但是不包括备注中的文本。
    //如果要返回备注中的文本，可以调用 setNotesByDefault(true)
}


```

```
return extractor.getText();  
}
```

PowerPoint 文件由一个或多个幻灯片(Slide)组成。第一张幻灯片的标题往往是整个 PowerPoint 文件的标题，提取标题实现代码如下：

```
SlideShow ss = new SlideShow(new HSLFSlideShow(is)); //is 是 ppt 文件的输入流  
Slide[] slides = ss.getSlides(); //获得每一张幻灯片  
return slides[0].getTitle(); //返回第一张幻灯片的标题
```

3.3 图像的 OCR 识别

抓取过程中，如果碰到包含价格或联系方式等信息的图片，需要转换成文字信息。例如这个以图片表示的价格：。在搜索引擎中实现图片或视频搜索时，也可以把图片或视频中的文字信息提取出来，然后再按文字来查找。

OCR(Optical Character Recognition)识别图像的过程的核心是对图像切割和分类。图像识别过程流程如下：

1. 对要识别的图像区域进行二值化处理；识别出字体和背景的颜色。如果有必要，还需要对图像中的干扰去噪。
2. 对二值化处理后的图像切分；图像切分就好像裁缝裁布一样，一般从空白处切开图片。因为字号变化导致对应的字符图像尺寸相差可能达到数十倍，所以还需要把切割后的图像大小归一化。
3. 判断包含单个字符的图像应该识别成哪个字符，这是个分类的问题，可以采用机器学习的方法实现，是有监督的学习(supervised learning)过程。先要准备好包含对应字符的图像样本及应该识别出的字符答案作为训练库。事先训练出模型后，识别时加载用分类的方法学习出的分类模型；分类方法可以选择 SVM 或者最大熵分类等。在 OCR 识别过程中，需要对从图像提取出的数据集进行分类，因此合适的分类器对结果的预测起着非常重要的作用。近年来，支持向量机(SVM)作为一种基于核方法的机器学习技术，不仅有强烈的理论基础而且有极好的成功经验。这里采用开源的 SVM 软件 LIBSVM 来对图像分类。
4. 执行分类并输出结果。如果有必要可以根据上下文猜测最有可能的输出结果来降低识别错误。

调用图像识别模块接口的代码如下：

```
String imgFile = "D:/lg/ocr/sample-images/ESfjydz.png";
```

```
OCR ocr = new OCR();
ocr.preload(); //学习的过程
String text = ocr.recognize(imgFile); //识别的过程
System.out.println("\nresult:"+text);
```

3.3.1 图像二值化

对要识别的图像区域进行二值化处理是很重要的步骤。比如一个 b 这样的图像区域变成一个 0 和 1 组成的图像，如图 2-7 所示：

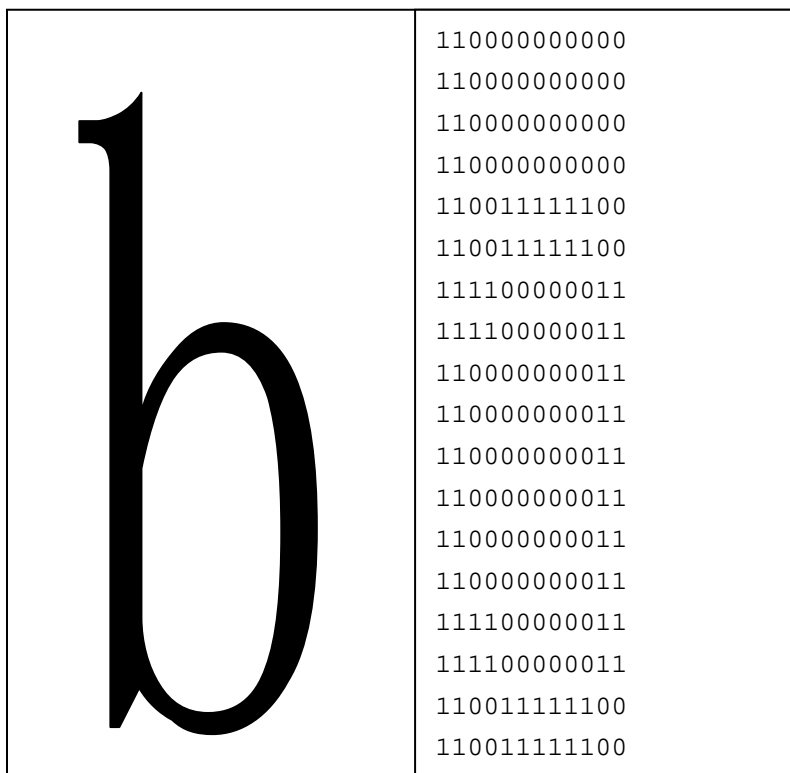


图 2-7 字符 b 以及它的 0 和 1 组成的二值化结果

在实际应用过程中图像二值化处理很重要，否则识别出来的可能都是 1 了，也就是说图片需要预处理成 0101 这样的黑白格式。

首先要支持读入多种常见图片文件格式，例如 jpg、gif 等。Java 中进行图像 I/O 有一些方法，其中的 Image I/O API 支持读写常见图片格式。Image I/O API 提供了加载和保存图片的方法。支持读取 GIF、JPEG 和 PNG 图像，也支持写 JPEG 和 PNG 图像，但是不支持写 GIF 文件。Java Image I/O API 主要在 javax.imageio 包。JDK 已经内置了常见图片格式的插件，但它提供了插件体系结构，第三方也可以开发插件支持其他图片格式。

javax.imageio.ImageIO 类提供了一组静态方法进行最简单的图像 I/O 操作。读取一个标准格式(GIF、PNG 或 JPEG)的图片很简单：

```
File f = new File("c:\images\myimage.gif");
BufferedImage bi = ImageIO.read(f);
```

Java Image I/O API 会自动探测图片的格式并调用对应的插件进行解码，当安装了一个新插件，新的格式会被自动理解，程序代码不需要改变。

有些待识别的图片存在干扰，可以考虑按颜色分辨出干扰。例如，有的图片只有三种颜色，白色背景，红色字，黑色水平线，黑色水平线是干扰。把黑色去掉，只留下白色背景，红色字。

将彩色图片黑白化的常规做法是先转化为灰度图，在把灰度图转化为黑白图。为了用计算机来表示和处理颜色，必须采用定量的方法来描述颜色，即建立颜色模型。常用的有红、绿、蓝为原色的 RGB 颜色模型和描述色彩饱和度的 HSV 颜色模型。RGB 彩色包含了亮度信息，即 YUV 模型中的 Y 分量，灰度计算公式如下：

$$\text{gray} = 0.229 * r + 0.587 * g + 0.114 * b;$$

计算一个 RGB 颜色模型表示的像素的灰度的代码如下：

```
/**
 * 计算一个 RGB 颜色的灰度
 * @param pixel
 * @return 对应的灰度
 */
private static int getGray(int pixel) {
    int r = (pixel >> 16) & 0xff;
    int g = (pixel >> 8) & 0xff;
    int b = (pixel) & 0xff;

    // 按公式计算出灰度值
    int gray = (int)(0.229 * r + 0.587 * g + 0.114 * b);

    return gray;
}
```

统计图片的灰度直方图的代码：

```
int grayValues[]; //每个像素对应的灰度值

BufferedImage bi = readImageFromFile(imageFile);

//得到图片的宽和高
int width = bi.getWidth(null);
int height = bi.getHeight(null);

//读取像素
```

```

int pixels[] = new int[width * height];
bi.getRGB(0, 0, width, height, pixels, 0, width);

//计算每个像素的灰度，保存下来
grayValues = new int[width * height];
for(int i = 0; i < width * height; i++) {
    grayValues[i] = getGray(pixels[i]);
}

int hist[] = new int[256]; // 存放每种灰度的出现次数

for(int i = 0; i < grayValues.length; i++) {
    hist[grayValues[i]]++; // 统计每种灰度出现的次数，即得到直方图
}

```

可以根据灰度值来确定像素的二值化结果。最简单的方式是固定阈值法。例如，若灰度大于 128，则认为是白色，灰度小于 128，则认为黑色。

二值化的双峰法的原理是：认为图像由前景和背景组成，在灰度直方图上，前后二景都形成高峰，在双峰之间的最低谷处就是图像的阈值所在，可根据此值，对图像进行二值化。除此外，还可以考虑对颜色聚类，用 K 平均聚类方法方法把颜色聚成两类。

3.3.2 切分图像

对字符区域定位的 CharRange 类把带有字符的区域从图像中分离出一个矩形框出来。

```

public class CharRange {
    int x; //横坐标位置
    int y; //纵坐标位置
    int width; //宽度
    int height; //高度
}

```

对图片进行垂直和水平方向的扫描的 Entry 类实现切割字符并且把字符大小归一化，也就是统一图像宽度和高度。Entry 的主要成员变量及方法如下：

```

public class Entry {
    static final int DOWNSAMPLE_WIDTH = 12; // 样本数据宽度
    static final int DOWNSAMPLE_HEIGHT = 18; // 样本数据高度
    protected Image entryImage; // 存储检测的图像
    protected Graphics entryGraphics; // 处理图形图像
    protected int pixelMap[]; // 存储图像像素
}

```

```
// 水平扫描图像并进行像素检测
protected boolean hLineClear(int x, int w, int y) {
    int totalWidth = entryImage.getWidth(null);
    for (int i = x; i <= w; i++) {
        if (pixelMap[(y * totalWidth) + i] != -1)
            return false;
    }
    return true;
}

// 垂直扫描图像并进行像素检测
protected boolean vLineClear(int x) {
    int w = entryImage.getWidth(null);
    int h = entryImage.getHeight(null);
    for (int i = 0; i < h; i++) {
        if (pixelMap[(i * w) + x] != -1)
            return false;
    }
    return true;
}

// 找到水平扫描时的上边界和下边界
void findVBound(CharRange cr) {
    for (int i = 0; i < cr.height; i++) {
        if (!hLineClear(cr.x, cr.width, i)) {
            cr.y = i;
            break;
        }
    }
    for (int i = cr.height - 1; i >= 0; i--) {
        if (!hLineClear(cr.x, cr.width, i)) {
            cr.height = i;
            break;
        }
    }
}
```

// 找到垂直扫描时的左边界和右边界

```
protected ArrayList<CharRange> findHBounds(int w, int h) {
    ArrayList<CharRange> bounds = new ArrayList<CharRange>();
    int begin = 0;
    int end = w;
    boolean lastState = false;
    boolean curState = false;
    for (int i = 0; i < w; i++) {
        if (vLineClear(i)) {
            System.out.println("find blank:" + i);
            curState = false;
        } else {
            curState = true;
        }
        if (!lastState && curState) {
            begin = i;
        } else if (lastState && !curState) {
            end = (i - 1);
            CharRange cr = new CharRange(begin, 0, end, h);
            bounds.add(cr);
        }
        lastState = curState;
    }
    if (curState) {
        CharRange cr = new CharRange(begin, 0, w - 1, h);
        bounds.add(cr);
    }
    return bounds;
}
```

// 发现图像边界

```
protected ArrayList<CharRange> findBounds(int w, int h) {
    ArrayList<CharRange> bounds = findHBounds(w, h);
    for (CharRange cr : bounds) {
        findVBound(cr);
    }
}
```



```
        return bounds;
    }

    // 对样本数据进行采样、归一化
    public ArrayList<SampleData> downSample() {
        int w = entryImage.getWidth(null);
        int h = entryImage.getHeight(null);
        ArrayList<SampleData> samples = new ArrayList<SampleData>();
        PixelGrabber grabber = new PixelGrabber(entryImage, 0, 0, w, h, true);
        grabber.grabPixels();
        pixelMap = (int[]) grabber.getPixels();
        ArrayList<CharRange> bounds = findBounds(w, h);
        for (CharRange cr : bounds) {
            SampleData data = new SampleData("?", DOWNSAMPLE_WIDTH,
                DOWNSAMPLE_HEIGHT);
            System.out.println(cr);
            double ratioX = (double) (cr.width - cr.x + 1)
                / (double) DOWNSAMPLE_WIDTH;
            double ratioY = (double) (cr.height - cr.y + 1)
                / (double) DOWNSAMPLE_HEIGHT;
            for (int y = 0; y < data.getHeight(); y++) {
                for (int x = 0; x < data.getWidth(); x++) {
                    if (downSampleQuadrant(x, y, ratioX, ratioY, cr.x, cr.y))
                        data.setData(x, y, true);
                    else
                        data.setData(x, y, false);
                }
            }
            data.ratio = (double) (cr.width - cr.x + 1)
                / (double) (cr.height - cr.y + 1);
            data.ratio = (data.ratio - 1) / 4;
            samples.add(data);
        }
        return samples;
    }
}
```

// 在样本数据的指定范围内进行像素扫描

```

protected boolean downSampleQuadrant(double x, double y, double ratioX,
                                     double ratioY, double downSampleLeft, double downSampleTop) {
    int w = entryImage.getWidth(null);
    int startX = (int) (downSampleLeft + (x * ratioX));
    int startY = (int) (downSampleTop + (y * ratioY));
    int endX = (int) (startX + ratioX);
    int endY = (int) (startY + ratioY);
    for (int yy = startY; yy <= endY; yy++) {
        for (int xx = startX; xx <= endX; xx++) {
            int loc = xx + (yy * w);
            if (pixelMap[loc] != -1)
                return true;
        }
    }
    return false;
}
}

```

3.3.3 SVM 分类

为了调用通用的模式识别代码来识别采样后的字符图像代表哪个字符,需要把这个图像识别问题抽象化一般的分类问题。对于一个输入对象 x , 有对应的输出类型 y 。在这里, 考虑对数字图片分类, 输入是采样后的“0101”序列, 输出则是 0-9 十个数字。“0101”序列中的每一位作为一个特征。训练集中的每个 x_i 都有对应的 y_i 这样对应的 m 个训练实例, 记作: $(x_1; y_1); \dots; (x_m; y_m)$ 。每个 x_i 有 k 个特征。

SVM 的分类方法可以分为训练的过程和识别的过程: 在训练的过程中, 要准备一些识别好的图片, 学习出分类模型; 在识别过程中, 调用学习好的分类模型来分类。LIBSVM(<http://www.csie.ntu.edu.tw/~cjlin/libsvm/>)是台湾大学的林智仁教授开发的可以解决分类问题的 SVM 软件包。LIBSVM 支持多种编程语言, 这里采用 Java 版本。LIBSVM 使用 `svm_train` 方法训练并返回一个支持向量机模型, 然后可以调用 `svm_predict` 方法根据已训练好的模型进行预测。LIBSVM 的输入特征是整数, 输出特征是正整数表示的类型。在 `svm_node` 类的实例中用稀疏的方式存储单个训练向量中的单个特征:

```

public class svm_node{
    public int    index; //特征编号
    public double value; //特征对应的值
}

```

因为汉字是大字符集, 识别较困难, 一般以笔画为依据。对于价格和邮件地址等单纯由数字和字母组成的文本, 可以简单的采用像素取样的方法。

在 `svm_problem` 类的实例中存储训练集中的所有样本及其所属类别。

```

public class svm_problem {
    //训练数据的实例数
    public int l;
    //存放它们的目标值，类型值用整型数据，回归值用实数
    public double[] y;
    //训练向量用二维矩阵表示
    //矩阵的第一个维度的长度是训练的实例数，第二个维度是特征数量。
    public libsvm.svm_node[][] x;
}

```

训练数据的输入样本组成了一个 SampleData 实例的数组：

```

public class SampleData{
    protected boolean grid[][];//用二维布尔数组存储的采样数据
    public double ratio;//宽高比，即 width/height
    protected String letter;//对应的字符或字符串
}

```

学习输入样本，得到分类模型：

```

//输入特征数量
int featureNum = Entry.DOWNSAMPLE_HEIGHT * Entry.DOWNSAMPLE_WIDTH + 1;
int outputNumber = sampleList.size();

//准备好训练集
TrainingSet set = new TrainingSet(featureNum, outputNumber);
set.setTrainingSetCount(sampleList.size());

for (int t = 0; t < sampleList.size(); t++) {
    int idx = 0;
    SampleData ds = (SampleData) sampleList.get(t);
    for (int y = 0; y < ds.getHeight(); y++) {
        for (int x = 0; x < ds.getWidth(); x++) {
            set.setInput(t, idx++, ds.getData(x, y) ? .5 : -.5);
        }
    }
    set.setInput(t, idx++, (ds.ratio));
    set.setOutput(t, t);
}

```

```
svm_parameter param = new svm_parameter();//设置模型参数
param.svm_type = svm_parameter.C_SVC;//标准算法
param.kernel_type = svm_parameter.RBF; //训练采用 RBF 核函数
param.degree = 3; //多项式核函数的阶次
param.gamma = 1.0 / featureNum;
param.coef0 = 0;
param.nu = 0.5;
param.cache_size = 100; //缓存内存大小设置为 100M
param.C = 1;
param.eps = 1e-3;
param.p = 0.1;
param.shrinking = 1; //使用启发式
param.probability = 0;
param.nr_weight = 0;
param.weight_label = new int[0];
param.weight = new double[0];

svm_problem prob = new svm_problem();//训练集

prob.l = outputNumber;
prob.x = new svm_node[prob.l][];
prob.y = new double[prob.l];
for (int i = 0; i < prob.l; i++){//设置输入及对应的输出
    prob.x[i] = set.getInputSet(i);
    prob.y[i] = set.getOutput(i);
}

//检查参数的有效性
String error_msg = svm.svm_check_parameter(prob, param);

//如果有错误则退出
if (error_msg != null) {
    System.err.print("Error: " + error_msg + "\n");
    System.exit(1);
}

//执行训练
```

```
model = svm.svm_train(prob, param);
```

执行对指定图片的分类过程:

```
LoadImage li = new LoadImage();//加载图像
```

```
Entry entry = new Entry();
```

```
entry.entryImage = li.loadImageFromFile(new File(imgFile));//加载图像
```

```
ArrayList<SampleData> samples = entry.downSample();//采样
```

```
//特征数组
```

```
svm_node[] input =
```

```
    new svm_node[Entry.DOWNSAMPLE_WIDTH* Entry.DOWNSAMPLE_HEIGHT + 1];
```

```
StringBuilder text = new StringBuilder();//存放识别结果
```

```
for (SampleData ds : samples) { //分别识别每个切分出的图像
```

```
    int idx = 0;
```

```
    for (int y = 0; y < ds.getHeight(); y++) {
```

```
        for (int x = 0; x < ds.getWidth(); x++) {
```

```
            input[idx] = new svm_node();//设置分类特征，把采样点作为分类特征
```

```
            input[idx].index = idx + 1;
```

```
            input[idx].value = ds.getData(x, y) ? .5 : -.5;//设置分类特征值
```

```
            ++idx;
```

```
        }
```

```
    }
```

```
    input[idx] = new svm_node();
```

```
    input[idx].index = idx + 1;
```

```
    input[idx].value = ds.ratio;
```

```
int best = (int) svm.svm_predict(model, input);//用 SVM 方法分类
```

```
String result = map[best];//取得分类 Id 对应的字符串
```

```
//对全黑图像字符的特殊处理
```

```
if (result.equals("I") || result.equals(".") || result.equals("-")) {
```

```
        if (ds.ratio == 0) {
            result = ".";
        } else if (ds.ratio < 0) {
            result = "l";
        } else if (ds.ratio > 0) {
            result = "-";
        }
    }

    text.append(result);
}
```

尽管 SVM 分类方法返回的结果准确度已经很高，但是仍然可能把字母“o”识别成了数字“0”，这样图像过于相似的情况下要靠上下文来识别，比如后面是 h，则前面的符号更可能是字母“o”而不是数字“0”，实现上可以参考 HMM(隐马模型)的介绍。

3.4 提取垂直行业信息

本节介绍在旅游或者医疗行业的特别的考虑。

3.4.1 医疗行业

医学编码是国际疾病编码(ICD)和手术操作编码，这些在发达国家是医保部门来对医院进行支付的首要条件，美国非常重视，在近几年用自然语言处理技术大大简化了这个编码过程，预测在 2014 年在美国就有 27 亿美元的市场。

比如你感冒了，感冒的编码是 0001，然后打了个吊瓶，打吊瓶的编码是 1001。再配合一些药物的编码。然后利用这些编码，结合一个价格表，就能知道你这次就医的花销，保险公司给你报销。编码后的用途是医保给医院进行支付，就是说什么代码有什么支付标准，所以医院很重视。

目前疾病编码是医院病案室专门有人查看病历并通过关键字查询等初级的检索方式进行查找，效率低，成本高。因为要确定一个疾病编码不只是医生下的诊断，还要参照病理学，实验室诊断，图像诊断等结果，综合起来才能确定。

有些疾病不是医生凭经验就能下决断的，还需要从相关文献，相关疾病史等很多很多的资料中查找出相应的资料出来，给医生下决断以参考。

3.4.2 旅游行业

出去玩的时候需要知道出发地和目的地。出发地组织人往往会提供手机号码。从手机号码可以提取地域信息。根据手机号码的前 7 位判断所属地域。取得地域信息的另外一个方法是根据用户 IP 地址判断。

```
private static final String QUERY_SQL = "select city from mobileloc where num = ?";
//根据手机号得到所属城市
public static String getStartCityByPhone(String mobile) {
    String startCity = null;
    if(mobile.trim().length() == 11) {
        mobile = mobile.substring(0,7);//根据前 7 位判断
        ResultSet rs = DBManager.getInstance().
            executeQueryByAdd(QUERY_SQL, mobile);
        if(rs.next()) {
            startCity = rs.getString("city");
        }
    }
    return startCity;
}
//测试方法
public static void main(String[] args) {
    System.out.println(getStartCityByPhone("18903045009"));
}
```

3.5 流媒体内容提取

相对于文本检索，音频和视频检索技术研究的比较少。常用的方法是提取出相关的文字描述来索引音频和视频。例如，把视频里面的声音通过语音识别(Speech Recognition)，然后放入索引库，同时记录时间点。语音识别的方法有很多技术问题，比如电影里面有背景音乐，如何去除噪声和音乐比较麻烦，这种将导致转换率低，而且多语言混杂，需要多种语言处理包，另外，说话的人可能有各种方言，导致转换率非常低。另外把 rmvb 等视频文件格式语言中的字用屏幕文字识别转换也非常麻烦，因为涉及到多语言的转换，尤其是汉字这样的大字符集文字，除非写得很标准。

视频搜索可以用在电台或电视台制作节目上，他们往往有录制了几十年的数据，现在想从中找出一些内容做合集，语音搜索和人脸搜索可以用上；视频网站每天都会有很多用户上传内容，这些有可能被插入一些违法的内容进去，如果全部人工审核的话，成本很大。先用一边内容检索过滤，可以大大降低人工参与的工作量；用户个人拍录像时可以加一些语音标签，方便日后编辑。

3.5.1 音频流内容提取

音频是多媒体中的一种重要媒体。我们能够听见的音频频率范围是 60Hz~20kHz，其中语音大约分布在 300Hz~4kHz 之内，而音乐和其他自然声响是全范围分布的。男人说话声音低沉，因为声带振动频率较低。而女人说话声音尖细，因为声带振动频率较高。童声高音频率范围为 260Hz-880Hz，低音频率范围 196Hz-700Hz。女声高音频率范围为 220Hz-1.1KHz，低音频率范围为 200Hz-700Hz。男声高音频率范围为 160Hz-523Hz，低音频率范围为 80Hz-358Hz。声音的响度对应强弱，而音高对应频率。频率是声音的物理特性，而音调则是频率的主观反映。

声音经过模拟设备记录或再生，成为模拟音频，再经数字化成为数字音频。数字化时的采样率必须高于信号带宽的 2 倍，才能正确恢复信号。样本可用 8 位或 16 位比特表示。一般保存成 wav 文件格式。尽管 wav 文件可以包括压缩的声音信号，但是一般情况下是没有压缩的。

以前的许多研究工作涉及到语音信号的处理，如语音识别。当前容易自动识别孤立的字词，如用在专用的听写和电话应用方面，而对连续的语音识别则较困难，错误较多，但目前在这方面已经取得了突破性的进展。同时还有些研究辨别说话人的技术。

语音检索是以语音为中心的检索，采用语音识别等处理技术。可以检索如电台节目、电话交谈、会议录音等。许多语音信号处理的研究成果可以用于语音检索：

(1)利用大词汇语音识别技术进行检索

这种方法是利用语音识别技术把语音转换为文本，从而可以采用文本检索方法进行检索。虽然好的连续语音识别系统在小心地操作下可以达到 90% 以上的词语正确度，但在实际应用中，如电话和新闻广播等，识别率并不高。即使这样，自动语音识别出来的文本仍然对信息检索有用，这是因为检索任务只是匹配包含在音频数据中的查询词句，而不是要求一篇可读性好的文章。例如，采用这种方法把视频的语音对话轨迹转换为文本脚本，然后组织成适合全文检索的形式支持检索。

(2)基于识别关键词进行检索

在无约束的语音中自动检测词或短语通常称为关键词的发现(Spotting)。利用该技术，识别或标记出长段录音或音轨中反映用户感兴趣的事件，这些标记就可以用于检索。如通过捕捉体育比赛解说词中“进球”的词语可以标记进球的内容。

(3)基于说话人的辨认进行分割

这种技术是简单地辨别出说话人语音的差别，而不是识别出说的是什么，叫做声纹识别。它在合适的环境中可以做到非常准确。将来也许可以用声纹识别代替刷卡来识别人。利用这种技术，可以根据说话人的变化分割录音，并建立录音索引。如用这种技术检测视频或多媒体资源的声音轨迹中的说话人的变化，建立索引和确定某种类型的结构(如对话)。例如，分

割和分析会议录音,分割的区段对应于不同的说话人,可以方便地直接浏览长篇的会议资料。

Sphinx-4(<http://cmusphinx.sourceforge.net/>)是采用 Java 实现的一个语音识别软件。Sphinx 是一个基于隐马尔科夫模型的系统,首先它需要学习一套语音单元的特征,然后根据所学来推断出所需要识别的语音信号最可能的结果。学习语音单元特征的过程叫做训练。应用所学来识别语音的过程有时也被称为解码。在 Sphinx 系统中,训练部分由 Sphinx Trainer 来完成,解码部分由 Sphinx Decoder 来完成。为了识别普通话,可以使用 Sphinx Trainer 自己建立普通话的声学模型。训练时需要准备好语音信号(Acoustic Signals),与训练用语音信号对应的文本(Transcript File)。当前 Sphinx-4 只能使用 Sphinx-3 Trainer 生成的 Sphinx-3 声学模型。有计划创建 Sphinx-4 trainer 用来生成 Sphinx-4 专门的声学模型,但是这个工作还没完成。

讲稿(transcript)文件中记录了单词和非讲话声的序列。序列接着一个标记可以把这个序列和对应的语音信号关联起来。

例如有 160 个 wav 文件,每个文件对应一个句子的发音。例如,播放第一个声音文件,会听到“a player threw the ball to me”,而且就这一句话。可以把这些 wav 或者 raw 格式的声音文件放到 myasm/wav 目录。

接下来,需要一个控制文件。控制文件只是一个文本文件。这里把控制文件命名为 myam_train.fileids (必须把它命名成[name]_train.fileids 的形式,这里 [name]是任务的名字,例如 myam),有每个声音文件的名字(注意,没有文件扩展名)。

0001

0002

0003

0004

接下来,需要一个讲稿文件,文件中的每行有一个独立文件的发声。必须和控制文件相对应。例如,如果控制文件中第一行是 0001,因此讲稿文件中的第一行的讲稿就是“A player threw the ball to me”,因为这是 0001.wav 的讲稿。讲稿文件也是一个文本文件,命名成 myam.corpus,应该有和控制文件同样多行。讲稿不包括标点符号,所以删除任何标题符号。例如:

a player threw the ball to me

does he like to swim out to sea

how many fish are in the water

you are a good kind of person

以这样的顺序，对应 0001、0002、0003 和 0004 文件。

现在有了一些声音文件，一个控制文件和一个讲稿文件。

Sphinx-4 由 3 个主要模块组成：前端处理器(FrontEnd)、解码器(Decoder)、和语言处理器(Linguist)。前端把一个或多个输入信号参数化成特征序列。语言处理器把任何类型的标准语言模型和声学模型以及词典中的发声信息转换成搜索图。这里，声学模型用来表示字符如何发音，语言模型用来评估一个句子的概率。解码器中的搜索管理器使用前端处理器生成的特征执行实际的解码，生成结果。在识别之前或识别过程中的任何时候，应用程序都可以发出对每个模块的控制，这样就可以有效的参与到识别过程中来。

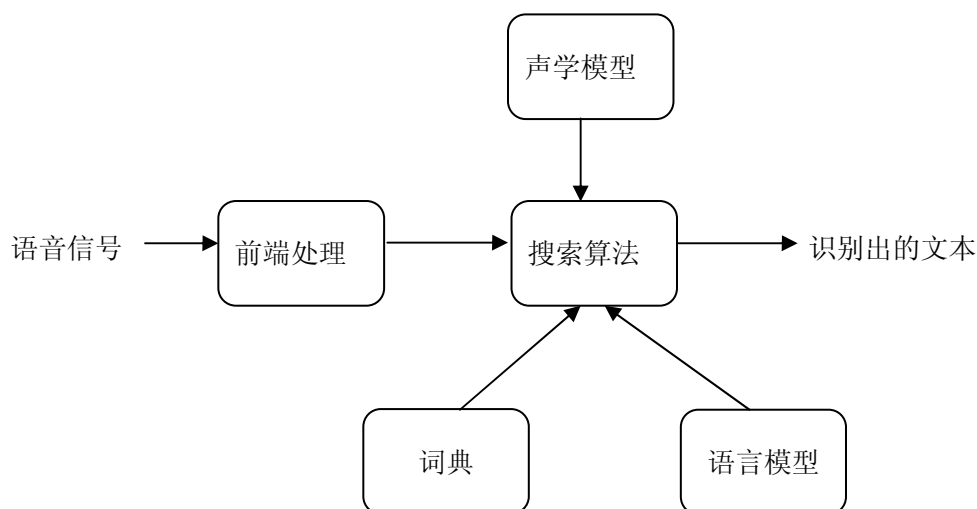


图 5.8 Sphinx-4 结构

语音识别的准确率受限于其识别的内容，内容越简单，则识别准确率越高。所以一般根据某个应用场景来识别语音。例如电视台要给录制的新闻节目加字幕。有批处理和实时翻译两种方式，这里采用批处理的方式。以识别新闻节目为例，开发流程如下：

- 1.准备新闻语料库：语料库就是一个文本文件，每行一个句子。
- 2.创建语言模型：一般采用基于统计的 n 元语言模型，例如 ARPA 格式的语言模型。可以使用语言模型工具 Kylm(<http://www.phontron.com/kylm/>)生成出 ARPA 格式的语言模型文件。
- 3.创建发声词典：对于英文可以采用 ARPABET 格式注音的发音词典。由于汉语是由音节(Syllable)组成的语言，所以可以采用音节作为汉语语音识别基元。每个音节对应一个汉字，比较容易注音。此外，每个音节由声母和韵母组成，声韵母作为识别基元也是一种选择。
- 4 设置配置文件：在配置文件中设置词典文件和语言模型路径。
- 5.在 eclipse 中执行语音识别的 Java 程序。初始情况下，需要执行 jsapi.exe 或 jsapi.sh 生成出

jsapi.jar 文件。

edu.cmu.sphinx.tools.feature.FeatureFileDumper 可以从音频文件导出特征文件，例如 MFCC 特征。一般提取语音信号的频率特征。找出语音信号中的音节叫做端点检测，也就是找出每个字的开始端点和结束端点。因为语音信号中往往存在噪音，所以不是很容易找准端点。

Transcriber.jar 可以实现从声音文件导出讲稿文件：

```
D:\sphinx4-1.0beta5\bin>java -jar -mx300M Transcriber.jar
one zero zero zero one
nine oh two one oh
zero one eight zero three
```

3.5.2 视频流内容提取

视频信息一般由四部分组成：帧、镜头、情节、节目。视频流的内容可以从多个层次分析：

- 底层内容建模，包括颜色、纹理、形状、空间关系、运动信息等。
- 中层内容建模，指视频对象(Video Object)，在 MPEG-4 中包括了视频对象。对象的划分可根据其独特的纹理、运动、形状、模型和高层语义为依据。
- 高层内容建模（语义概念等），例如一段体育视频，可以提取出扣篮或射门的片断。

关键帧提取策略：

1. 设置一个最大关键帧数 M ；
2. 每个镜头的非边界过渡区的第一帧确定为关键帧；
3. 使用非极大值抑制法确定镜头边界系数极大值，并排序，以实现基于镜头边界系数的关键帧提取。

镜头边界检测方法有：只使用镜头边界系数的镜头边界检测——固定阈值法；结合相邻帧差的镜头边界检测——自适应阈值法。

可以使用 Java 媒体框架(JMF)API 在 Java 语言中处理声音和视频等时序性的媒体。下面我们首先通过 vid2jpg.java 类来提取视频中所有的帧。

```
/**
 * 用这个方法转换从 PushBufferDataSource 发出的数据
 */
```

```
public void transferData(PushBufferStream stream)  {
    stream.read(readBuffer);

    //为了防止数据对象中的内容被其他线程修改
    Buffer inBuffer = (Buffer)(readBuffer.clone());

    // 检查流是否已经结束
    if(readBuffer.isEOM()) {
        System.out.println("End of stream");
        return;
    }
    //实际把视频中的帧保存到文件
    useFrameData(inBuffer);
}
```

除了关键帧抽取，还可以考虑根据说话人识别和人脸识别来标注或搜索视频。例如，当某人说话的时候，找出视频库里所有该人说的话。对于有声音的视频可以利用视频中的音频信息检索内容。视频流和音频流在时间上是同步的。

手机应用 IntoNow 可以捕捉视频中的声音，通常它只需“听”20 秒左右，就可以判断出你正在看的是什么节目，并列出了这个节目的基本信息，以及同样在看这个节目的其他 IntoNow 用户。

3.6 存储提取内容

3.6.1 存入数据库

把提取出来的信息保存到 Access 数据库。

Access 数据库管理软件是微软 Office 中的一部分。使用 Access2003 新建数据库文件 goods.mdb，然后创建表。

如果微软 Office 中 Access 安装不上，则可以用 MDB Viewer Plus 这个不需要安装的软件来管理 mdb 文件。下载地址：http://www.alexnolan.net/software/mdb_viewer_plus.htm

MDB Viewer Plus 使用 Microsoft Data Access Components (MDAC)访问数据库。MDAC 是 Windows 的一部分。

使用 JDBC 写入数据。使用 JDBC-ODBC 桥。

```
String accessFile = "d:/db/POI.mdb";
```

```

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); //装载驱动程序
Connection con = DriverManager.getConnection(
    "jdbc:odbc:driver={Microsoft Access Driver (*.mdb)};DBQ="
    + accessFile); //建立连接
String strSql = "insert into Table_Address(省名,地区,县市,乡镇村,邮政编码,区号) values
(?,?,?,?,?,?)";
//创建一个 PreparedStatement 对象
PreparedStatement psmt = con.prepareStatement(strSql);
psmt.setString(1, record.getProvince());
psmt.setString(2, record.getArea());
psmt.setString(3, record.getCity());
psmt.setString(4, record.getVillage());
psmt.setString(5, record.getPostcode());
psmt.setString(6, record.getCode());
psmt.executeUpdate();
con.close(); //关闭连接，这样才能把数据保存到 mdb 文件

```

3.6.2 写入维基

可以把提取出的结果写到索引或者维基。例如，Java Wiki Bot Framework(<http://jwbfs.sourceforge.net>)可以实现把内容写入到 MediaWiki。

```

MediaWikiBot mediaWikiBot = new MediaWikiBot(
    "http://wiki.1798hw.com/index.php");//维基首页
mediaWikiBot.login("1798hw", "1798hw");//登陆到维基
String title = "黄山";//标题
Article article = new Article(mediaWikiBot, title);//设置标题
article.setText("黄山内容介绍");//设置内容
article.save();//保存文章

```

为了实现自动写维基，可以把维基百科中在编辑状态的信息抓取过来。

<http://zh.wikipedia.org/w/index.php?title=%E7%94%B5%E8%A7%86&action=edit>

为了能够上传图片链接。需要修改 mediawiki 的配置文件 Localsetting.php。增加：

```

$wgAllowExternalImages = true;
$wgAllowCopyUploads = true;

```

就可以上传外部图片链接地址。

在写入维基时，只需要加入图片地址，就会自动显示图片。以下是上传图片链接示例：

```
MediaWikiBot mediaWikiBot = new MediaWikiBot("http://wiki.1798hw.com/index.php");
mediaWikiBot.login("1798hw", "1798hw");
Article article = new Article(mediaWikiBot, title);
article.setText(body);
article.addTextnl(imgUrl);
article.save();
```

但这样做图片仍然保存在远程的服务器，而不是本地的地址。一般无法访问维基百科的图片，需要使用国外的代理来抓取维基百科中的正常的图片。

3.7 本章小结

怎么识别网页的编码？网页的头信息和 meta 域，还有二进制流本身。二进制流采用概率估计的方法。如果三个地方编码各不相同，则判别依据是：meta 域 > 头信息 > 二进制流。

除了本章已经介绍过的 HTMLParser 和 NekoHTML 以外，Jsoup(<http://jsoup.org/>)提供了类似于 JQuery 的操作方法来取出和操作数据。

如果用 Python，抓一个页面只需一句话：

```
html = urllib2.urlopen('http://www.baidu.com')
```

至于解析库，Python 中有基于 HTML 语法分析的 lxml.html、beautifulsope 和 PyQuery。Python 也有对正则表达式的支持。

在本章介绍的从各种数据来源提取索引需要的信息，是爬虫开发中重要而且往往容易碰到问题的部分。各种文档格式处理方式总结如表所示。

格式	解析包
Microsoft Office OLE2 Compound Document Format(Excel, Word, PowerPoint, Visio, Outlook)	Apache POI
Microsoft Office 2007 OOXML	Apache POI
Adobe Portable Document Format (PDF)	PDFBox
Rich Text Format (RTF)	RtfParser

格式	解析包
英文文本	ICU4J
HTML	NekoHTML
XML	Java 的 javax.xml 类
ZIP Archives	Java 内部的 ZIP 类
TAR Archives	Apache Ant
GZIP compression	Java 内部的 GZIPInputStream
BZIP2 compression	Apache Ant
Image formats (metadata only)	Java 的 javax.imageio 类
Java class files	ASM library (JCR-1522)
Java JAR files	ZIP + Java Class files
MP3 audio	org.farng.mp3
Open Document	直接解析 XML
Microsoft Office 2007 XML	Apache POI
MIDI 文件	Java 内部的 javax.sound.midi.*
DWG 文件	DWGDirect

第4章 中文分词原理与实现

有个经典笑话：护士看到病人在病房喝酒，就走过去小声叮嘱说：小心肝！病人微笑道：小宝贝。在这里，“小心肝！”这句话有歧义，从护士的角度理解是：“小心/肝”，在病人的角度理解是“小心肝”。如果使用中文分词切分成“小心/肝”则可以消除这种歧义。

和英文不同，中文词之间没有空格。很多读者可能会同意这样，因为可以省纸。实现专业的中文搜索引擎，比英文多了一项分词的任务。英语、法语和德语等西方语言通常采用空格或标点符号将词隔开，具有天然的分隔符，所以词的获取简单。但是中文、日文和韩文等东方语言，虽然句子之间有分隔符，但词与词之间没有分隔符，所以需要靠程序切分出词。另外，除了可以用于全文查找，中文分词的方法也被应用到英语手写体识别中。因为在识别手写体时，单词之间的空格就不很清楚了。

要解决中文分词准确度的问题，是否提供一个免费版本的分词程序供人下载使用就够了？但像分词这样的自然语言处理领域的问题，很难彻底的全部解决。例如，通用版本的分词也许需要做很多修改后才能用到手机上。所以需要让人能看懂其中的代码与实现原理，并参与到改进的过程中才能更好的应用。

本章的中文分词和下章介绍的文档排重和关键词提取等技术都属于自然语言处理技术的范围。因为在中文信息处理领域，中文分词一直是一个值得专门研究的问题，所以单独作为一章。

4.1 Lucene 中的中文分词

Lucene 中处理中文的常用方法有三种。以“咬死猎人的狗”这句话的输出结果为例：

1. 单字方式：[咬] [死] [猎] [人] [的] [狗]；
2. 二元覆盖的方式：[咬死] [死猎] [猎人] [人的] [的狗]；
3. 分词的方式：[咬] [死] [猎人] [的] [狗]。

例如：《新华字典》或者《现代汉语词典》是按字索引的。

Lucene 中的 `StandardTokenizer` 采用了单字分词的方式。`CJKTokenizer` 采用了二元覆盖的实现方式。笔者开发的 `CnTokenizer` 采用了分词的方式，本章将介绍部分实现方法。

4.1.1 Lucene 切分原理

Lucene 中负责语言处理的部分在 `org.apache.lucene.analysis` 包。其中 `TokenStream` 类用来进行基本的分词工作。`Analyzer` 类是 `TokenStream` 的外围包装类，负责整个解析工作。有人把文本解析比喻成人体的消化过程，输入食物，分解出有用的氨基酸和葡萄糖等。`Analyzer` 类接收的是整段的文本，解析出有意义的词语。

通常不需要直接调用分词的处理类 `analysis`，而是由 Lucene 内部来调用，其中：

- 在做索引阶段，调用 `addDocument(doc)` 时，Lucene 内部使用 `Analyzer` 来处理每个需要索引的列，如图 4-1 所示。

```
IndexWriter index = new IndexWriter(indexDirectory,
                                    new CnAnalyzer(), //用支持分词的分析器
                                    !incremental,
                                    IndexWriter.MaxFieldLength.UNLIMITED);
```

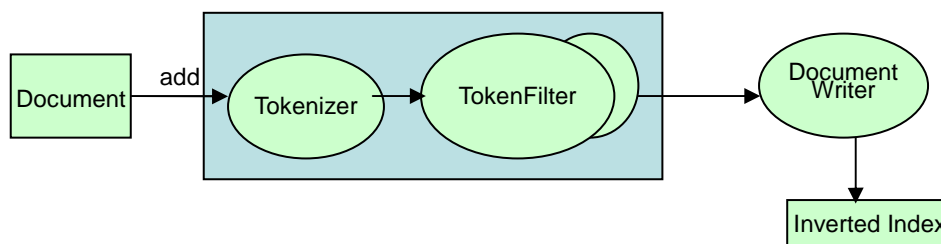


图 4-1 Lucene 对索引文本的处理

- 在搜索阶段，调用 `QueryParser.parse(queryText)` 来解析查询串时，`QueryParser` 会调用 `Analyzer` 来拆分查询字符串，但是对于通配符等查询不会调用 `Analyzer`。

```
Analyzer analyzer = new CnAnalyzer(); //支持中文的分词
QueryParser parser = new QueryParser(Version.LUCENE_CURRENT, "title", analyzer);
```

因为在索引和搜索阶段都调用了分词过程，索引和搜索的切分处理要尽量一致。所以分词效果改变后需要重建索引。另外，可以用个速度快的版本，用来在搜索阶段切分用户的查询词，另外用一个准确切分的慢速版本用在索引阶段的分词。

为了测试 Lucene 的切分效果，下面是直接调用 `Analysis` 的例子：

```
Analyzer analyzer = new CnAnalyzer(); //创建一个中文分析器
//取得 Token 流
TokenStream ts = analyzer.tokenStream("myfield", new StringReader("待切分文本"));
while (ts.incrementToken()) { //取得下一个词
    System.out.println("token: " + ts);
}
```

```
}
```

4.1.2 Lucene 中的 Analyzer

为了更好的搜索中文，先通过图 4-2 了解在 Lucene 中通过 `WhitespaceTokenizer`、`WordDelimiterFilter`、`LowercaseFilter` 处理英文字符串的流程。

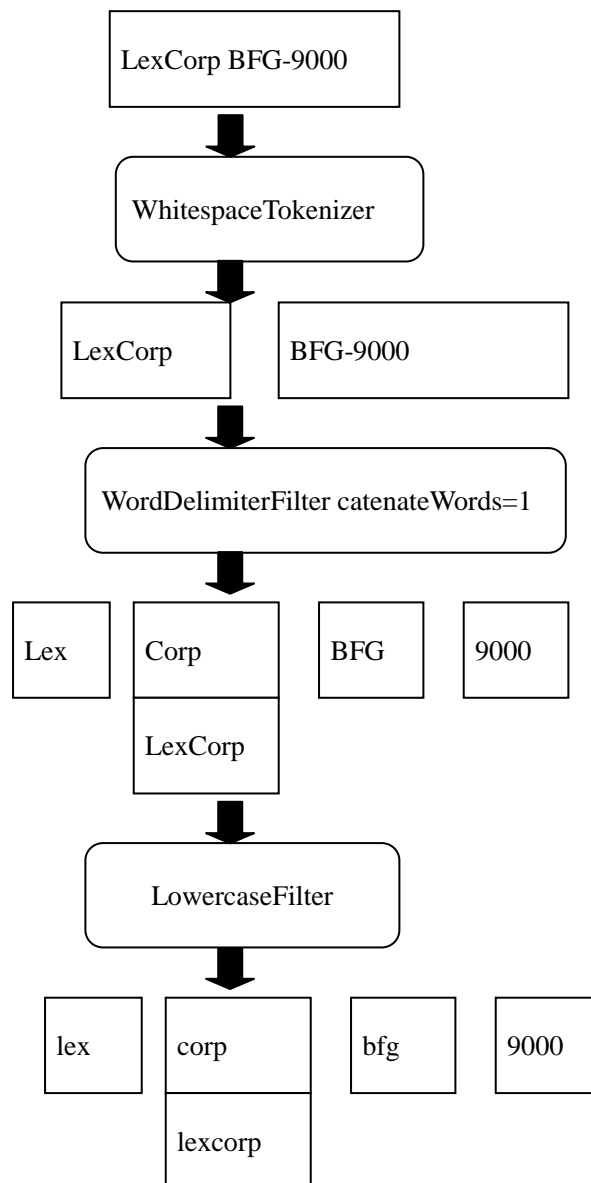


图 4-2 Lucene 处理英文字符串流程

Lucene 中的 `StandardAnalyzer` 对于中文采用了单字切分的方式。这样的结果是单字匹配的效果。搜索“上海”，可能会返回和“海上”有关的结果。

`CJKAnalyzer` 采用了二元覆盖的方式实现。小规模搜索网站可以采用二元覆盖的方法，

这样可以解决单字搜索“上海”和“海上”混淆的问题。采用中文分词的方法适用于中大规模的搜索引擎。猎兔搜索提供了一个基于 Lucene 接口的 Java 版中文分词系统。

可以对不同的索引列使用不同的 Analyzer 来切分。例如可以对公司名采用 CompanyAnalyzer 来分析，对地址列采用 AddressAnalyzer 来分析。这样可以通过更细分化的切分方式来实现更准确合适的切分效果。

例如把“唐山聚源食品有限公司”拆分成表 4-1 所示的结果：

词	开始位置	结束位置	标注类型
唐山	0	2	City
聚源	2	4	KeyWord
食品	4	6	Feature
有限公司	6	10	Function

表 4-1 公司名拆分结果表

这里的开始位置和结束位置是指词在文本中的位置信息，也叫做偏移量。例如“唐山”这个词在“唐山聚源食品有限公司”中的位置是 0-2。OffsetAttribute 属性保存了词的位置信息。TypeAttribute 属性保存了词的类型信息。

切分公司名的流程如图 4-4 所示：

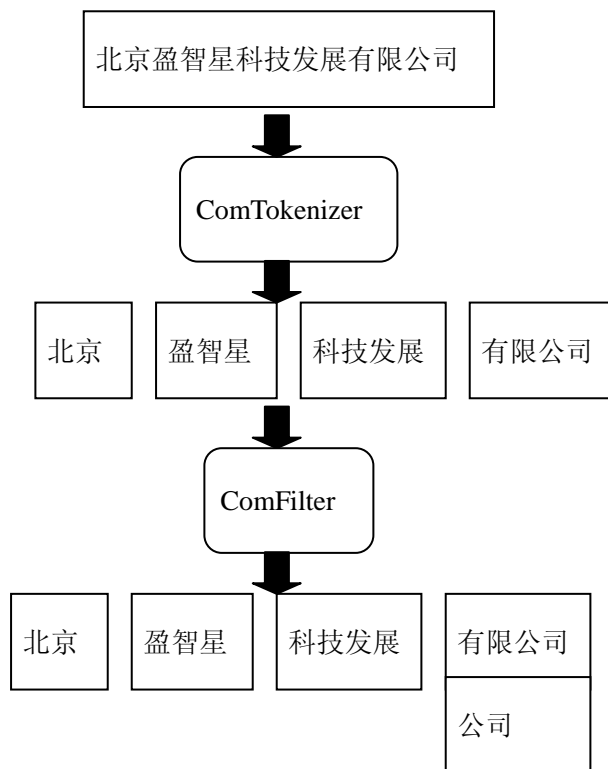


图 4-4 Lucene 处理公司名流程

专门用来处理公司名的 `CompanyAnalyzer` 实现如下：

```
public class CompanyAnalyzer extends Analyzer {  
    public TokenStream tokenStream(String fieldName, Reader reader) {  
        //调用 ComTokenizer 切分公司名  
        TokenStream stream = new ComTokenizer(reader);  
        //调用 ComFilter 后续加工  
        stream = new ComFilter(stream);  
        return stream;  
    }  
}
```

对不同的数据写了不同用途的分析器，需要在同一个索引中针对不同的索引列使用。一般情况下只使用一个分析器。为了对不同的索引列使用不同的分析器，可以使用 `PerFieldAnalyzerWrapper`。在 `PerFieldAnalyzerWrapper` 中，可以指定一个缺省的分析器，也可以通过 `addAnalyzer` 方法对不同的列使用不同的分析器。例如：

```
PerFieldAnalyzerWrapper aWrapper =  
    new PerFieldAnalyzerWrapper(new CnAnalyzer());  
aWrapper.addAnalyzer("address", new AddAnalyzer());
```

```
aWrapper.addAnalyzer("companyName", new CompanyAnalyzer());
```

在这个例子中，将对所有的列使用 `CnAnalyzer`，除了地址列使用 `AddAnalyzer`，公司名称列使用 `CompanyAnalyzer`。像其他分析器一样，`PerFieldAnalyzerWrapper` 可以在索引或者查询解析阶段使用。

4.1.3 自己写 Analyzer

`Analyzer` 内部包括一个对字符串的处理流程，先由 `Tokenizer` 做基本的切分，然后再由 `Filter` 做后续处理。一个简单的分析器(`Analyzer`)的例子：

```
public class MyAnalyzer extends Analyzer {
    public TokenStream tokenStream(String fieldName, Reader reader) {
        //以空格方式切分 Token
        TokenStream stream = new WhitespaceTokenizer(reader);
        //删除过短或过长的词，例如 in、of、it
        stream = new LengthFilter(stream, 3, Integer.MAX_VALUE);
        //给每个词标注词性
        stream = new PartOfSpeechAttributeImpl.PartOfSpeechTaggingFilter(stream);
        return stream;
    }
}
```

一般在 `Tokenizer` 的子类实际执行词语的切分。需要设置的值有：和词相关的属性 `termAtt`、和位置相关的属性 `offsetAtt`。在搜索结果中高亮显示查询词时，需要用到和位置相关的属性。但是在切分用户查询词时，一般不需要和位置相关的属性。此外还有声明词类型的属性 `TypeAttribute`。`Tokenizer` 的子类需要重写 `incrementToken` 方法。通过 `incrementToken` 方法遍历 `Tokenizer` 分析出的词，当还有词可以获取时，返回 `true`，已经遍历到结尾时，返回 `false`。

基于属性的方法把无用的词特征和想要的词特征分隔开。每个 `TokenStream` 在构造时增加它想要的属性。在 `TokenStream` 的整个生命周期中都保留一个属性的引用。这样在获取所有和 `TokenStream` 实例相关的属性时，可以保证属性的类型安全。

```
protected CnTokenStream(TokenStream input) {
    super(input);
    termAtt = (TermAttribute) addAttribute(TermAttribute.class);
}
```

在 `TokenStream.incrementToken()` 方法中，一个 `token` 流仅仅操作在构造方法中声明过的属性。例如，如果只要分词，则只需要 `TermAttribute`。其他的属性，例如 `PositionIncrementAttribute` 或者 `PayloadAttribute` 都被这个 `TokenStream` 忽略掉了，因为这时不需要其他的属性。

```

public boolean incrementToken() throws IOException {
    if (input.incrementToken()) {
        final char[] termBuffer = termAtt.termBuffer();
        final int termLength = termAtt.termLength();
        if (replaceChar(termBuffer, termLength)) {
            termAtt.setTermBuffer(output, 0, outputPos);
        }
        return true;
    }
    return false;
}

```

虽然也可以通过 `termAtt` 对象中的 `term` 方法返回词，但这个方法返回的是字符串，直接返回字符数组的 `termBuffer` 方法性能更好。下面是采用正向最大长度匹配实现的一个简单的 `Tokenizer`。

```

public class CnTokenizer extends Tokenizer {
    private static TernarySearchTrie dic = new TernarySearchTrie("SDIC.txt");//词典
    private TermAttribute termAtt;// 词属性
    private static final int IO_BUFFER_SIZE = 4096;
    private char[] ioBuffer = new char[IO_BUFFER_SIZE];

    private boolean done;
    private int i = 0;// i 是用来控制匹配的起始位置的变量
    private int upto = 0;

    public CnTokenizer(Reader reader) {
        super(reader);
        this.termAtt = ((TermAttribute) addAttribute(TermAttribute.class));
        this.done = false;
    }

    public void resizeIOBuffer(int newSize) {
        if (ioBuffer.length < newSize) {
            // Not big enough; create a new array with slight
            // over allocation and preserve content
            final char[] newCharBuffer = new char[newSize];
            System.arraycopy(ioBuffer, 0, newCharBuffer, 0, ioBuffer.length);
            ioBuffer = newCharBuffer;
        }
    }
}

```

```
    }  
}  
  
@Override  
public boolean incrementToken() throws IOException {  
    if (!done) {  
        clearAttributes();  
        done = true;  
        upto = 0;  
        i = 0;  
        while (true) {  
            final int length = input.read(ioBuffer, upto, ioBuffer.length  
                - upto);  
            if (length == -1)  
                break;  
            upto += length;  
            if (upto == ioBuffer.length)  
                resizeIOBuffer(upto * 2);  
        }  
    }  
  
    if (i < upto) {  
        char[] word = dic.matchLong(ioBuffer, i, upto); // 正向最大长度匹配  
        if (word != null) { // 已经匹配上  
            termAtt.setTermBuffer(word, 0, word.length);  
            i += word.length;  
        } else {  
            termAtt.setTermBuffer(ioBuffer, i, 1);  
            ++i; // 下次匹配点在这个字符之后  
        }  
        return true;  
    }  
    return false;  
}  
}
```

不太可能搜索某些词, 这样不被索引的词叫做停用词。**Analyzer** 可能会去掉一些停用词。

4.1.4 Lietu 中文分词

Lietu 中文分词程序由 seg.jar 的程序包和一系列词典文件组成。通过系统参数 dic.dir 指定词典数据文件路径。我们可以写一个简单的分词测试代码：

```
String sentence = "有关刘晓庆偷税案";  
//输入句子，返回单词组成的数组  
String[] result = com.lietu.seg.result.Tagger.split(sentence);  
for (int i=0; i<result.length;i++){  
    System.out.println(result[i]);  
}
```

使用分词的时候为了高亮显示关键字，必需保留词的位置信息。如果分词处理的网页有很多无意义的乱码，这些乱码导致的无意义的位置信息存储甚至可能会导致 5 倍以上的膨胀率。

4.2 查找词典算法

最常见的分词方法是基于词典匹配的。在基于词典的中文分词方法中，最经常用到的功能是从字符串的指定位置向后查找词，也就是最大长度查找方法。

《现代汉语词典》收录 6.5 万词条。中文分词实际使用的词典规模往往在几十万词以上，查找词典所占的时间可能在总的分词时间的 1/3 左右。为了保证切分速度，需要选择一个好的查找词典算法。

在讨论查找词典方法之前，首先看看词典格式。词典格式可以是方便人工查看和编辑的文本文件格式，也可以是方便机器读入的二进制格式，可以按约定存放在 dic 路径下或者由用户指定存放路径。最基本的词典的文本文件格式就是每行一个词。读入代码如下：

```
InputStream file = null;  
if (System.getProperty("dic.dir")==null)//用户没有指定词典存放路径时，从缺省的路径加载  
    file = getClass().getResourceAsStream(Dictionary.getDir() + dic);  
else  
    file = new FileInputStream(new File(Dictionary.getDir() + dic));  
  
BufferedReader in = new BufferedReader(new InputStreamReader(file, "GBK"));  
String word;  
while ((word = in.readLine()) != null) {  
    //按行处理读入的文本格式的词典  
}
```



```
in.close();
```

4.2.1 标准 Trie 树

假设一种简单的情况,词典中全部是英文单词。首先考虑散列这种常见的高效查找方法,它根据数组下标查询,所以速度快。首先根据词表构造散列表,具体来说就是用给定的散列函数构造词典到数组下标的映射,如果存在冲突,则根据选择的冲突处理方法解决地址冲突。然后可以在散列表的基础上执行散列查找。冲突导致散列性能降低。不存在冲突的散列表叫做完美散列(perfect hash)。但是整词散列不适合分词的最长匹配查找方式。

英文单词中的每个词都是由 26 个小写英文字母中的一个组成的,可以采用逐字散列的方法,这就是标准 Trie 树。可以把标准 Trie 树看成是一种逐字的完美散列。一个标准 Trie 树(retrieve 树)的一个节点只保留一个字符。如果一个单词比一个字符长,则包含第一个字符的节点记录指向下一个字符的节点的属性,依次类推。这样组成一个层次结构的树,树的第一层包括所有单词的第一个字符,树的第二层包括所有单词的第二个字符,依次类推,标准 Trie 树的最大高度是词典中最长单词的长度。例如,如下单词序列组成的词典(as at be by he in is it to) 会生成如图 4-2 所示的标准 Trie 树:

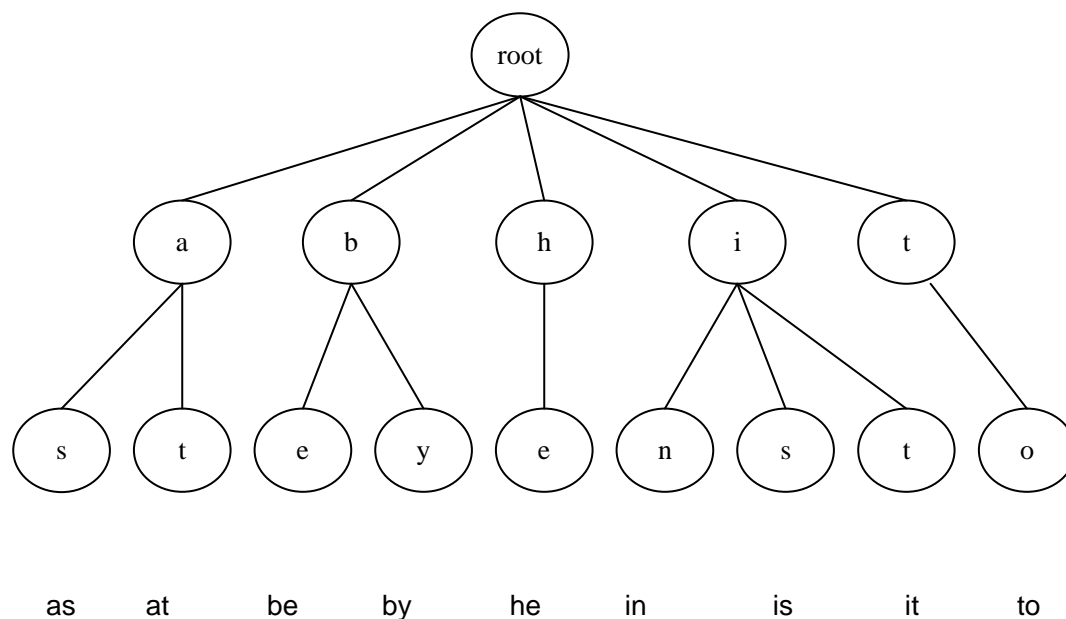


图4-2 标准Trie树

标准 Trie 树的结构独立于生成树时单词进入的顺序。这里, Trie 树的高度是 2。因为树的高度很小,在标准 Trie 树中搜索一个单词的速度很快。但是,这是以内存消耗为代价的,树中的每一个节点都需要很多内存。假设每个词都是由 26 个小写英文字母中的一个组成的,则这个节点中会有 26 个属性。所以不太可能直接用这样的标准 Trie 树来存储中文这样的大字符集。

Trie树在实现上有一个树类(SearchTrie)和一个节点类(TrieNode)。SearchTrie的主要

方法有两个：

- 往搜索树上增加一个单词，方法原型是：`addWord(String word)`。
- 从给定字符串的指定位置开始匹配单词，方法原型是：`matchLong(String text, int offset)`。

Trie树的节点类定义如下：

```
public static final class TrieNode {  
    protected TrieNode[] children; //孩子节点组成的数组，类似散列表中的数组  
    protected char splitChar;    //分隔字符，这里是英文字符  
  
    /**  
     * 构造方法  
     *  
     * @param splitchar 分隔字符  
     */  
    protected TrieNode(char splitchar) {  
        children = new TrieNode[26]; //26 个小写英文字母  
        this.splitChar = splitchar;  
    }  
}
```

一个应用场景：给定一个固定电话号码，找出这个电话号码对应的区域。固定电话号码都是以0开始的多位数字，可以通过给定电话号码的前缀找出对应的地区，例如：

0995:新疆:托克逊县

0856:贵州:铜仁

0996:新疆:焉耆回族自治县

可以使用标准 Trie 树算法快速查找电话号码前缀。例如：0 3 7 1 这个区号有四个节点对应。也就是说四个 TrieNode 对象。第一个对象中的 `splitChar` 属性是 0，第二个对象中的 `splitChar` 属性是 3，第三个对象中的 `splitChar` 属性是 7，第四个对象中的 `splitChar` 属性是 1。每个 TrieNode 对象有 10 个孩子节点，分别对应孩子节点的 `splitChar` 为 0 到 9。0 3 7 1 形成的 Trie 树如图 4-3 所示。

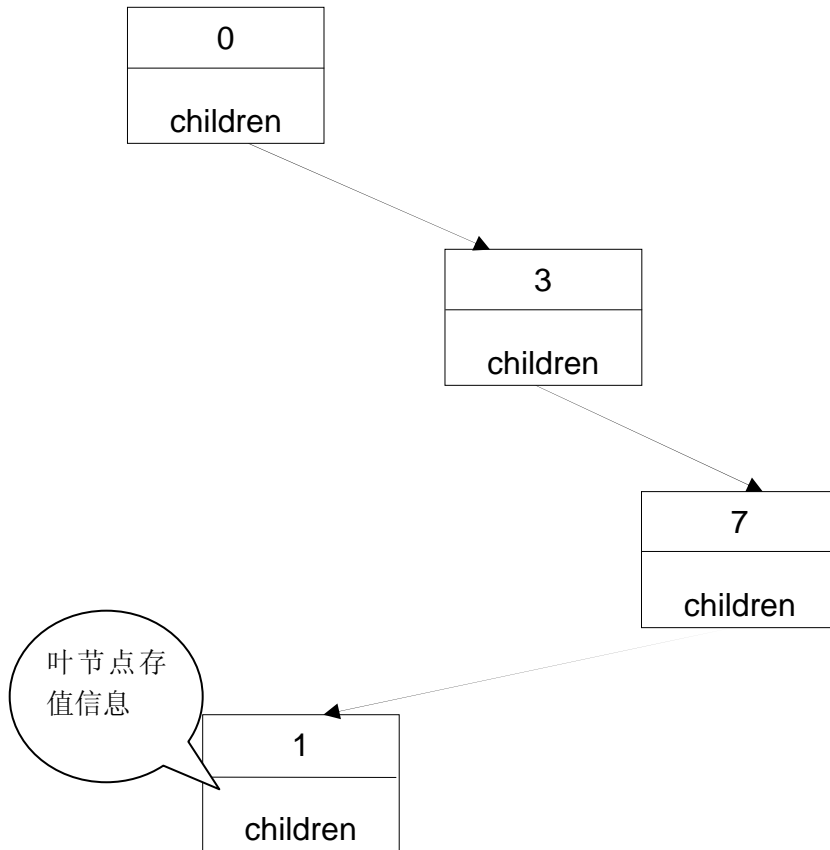


图4-3 电话号码组成的标准Trie树

Trie树的节点类定义如下：

```

public static final class TrieNode {
    protected TrieNode[] children; //孩子节点组成的数组
    protected char splitChar;    //分隔字符
    protected String area; //电话所属地区信息

    /**
     * 构造方法
     *
     * @param splitchar 分隔字符
     */
    protected TrieNode(char splitchar) {
        children = new TrieNode[10]; //10 个数字
        area = null;
        this.splitChar = splitchar;
    }
}

```

加载词，形成标准Trie树的方法如下：

```
private void addWord(String string, TSTNode root, String area) {
    TSTNode tstNode = root;
    for (int i = 1; i < string.length(); i++) {
        char c0 = string.charAt(i);
        int ind = c0 - '0';
        TSTNode tmpNode = tstNode.children[ind];
        if (null == tmpNode) {
            tmpNode = new TSTNode(c0);
        }
        if (i == string.length() - 1) {
            tmpNode.area = area;
        }
        tstNode.children[ind] = tmpNode;
        tstNode = tmpNode;
    }
}
```

查询的过程对于查询词来说，从前往后一个字符一个字符的匹配。对 Trie 树来说，是从根节点往下匹配的过程。从给定电话号码搜索前缀的方法：

```
public String search(String tel) {
    TrieNode tstNode = root;
    for (int i = 1; i < tel.length(); i++) { //从前往后一个字符一个字符的匹配
        tstNode = tstNode.children[(tel.charAt(i)-'0')];
        if (null != tstNode.area) {
            return tstNode.area;
        }
    }
    return null; //没找到
}
```

考虑把 Trie 树改成通用的结构，使用散列表存储孩子节点。使用范型定义值类型。

```
public class TrieNode<T> {
    private Character splitChar; //分隔字符
    private T nodeValue; //值信息
    private Map<Character, TrieNode<T>> children =
        new HashMap<Character, TrieNode<T>>(); //孩子节点
}
```

4.2.2 三叉 Trie 树

在一个三叉 Trie 树(TernarySearchTrie)中, 每一个节点也是包括一个字符。但和标准 Trie 树不同, 三叉 Trie 树的节点中只有三个位置相关的属性, 一个指向左边的树, 一个指向右边的树, 还有一个向下, 指向单词的下一个数据单元。三叉 Trie 树是二叉搜索树和标准 Trie 树的混合体。它有和标准 Trie 树差不多的速度但是和二叉搜索树一样只需要相对较少的内存空间。

通过选择一个排序后的词表的中间值, 并把它作为开始节点, 可以创建一个平衡的三叉树。再次以有序的单词序列(as at be by he in is it of on or to)为例。首先把关键字“is”作为中间值并且构建一个包含字母“i”的根节点。它的直接后继结点包含字母“s”并且可以存储任何与“is”有关联的数据。对于“i”的左树, 选择“be”作为中间值并且创建一个包含字母“b”的结点, 字母“b”的直接后继结点包含“e”。该数据存储在“e”结点。对于“i”的右树, 按照逻辑, 选择“on”作为中间值, 并且创建“o”结点以及它的直接后继结点“n”。最终的三叉树如图 4-3 所示:

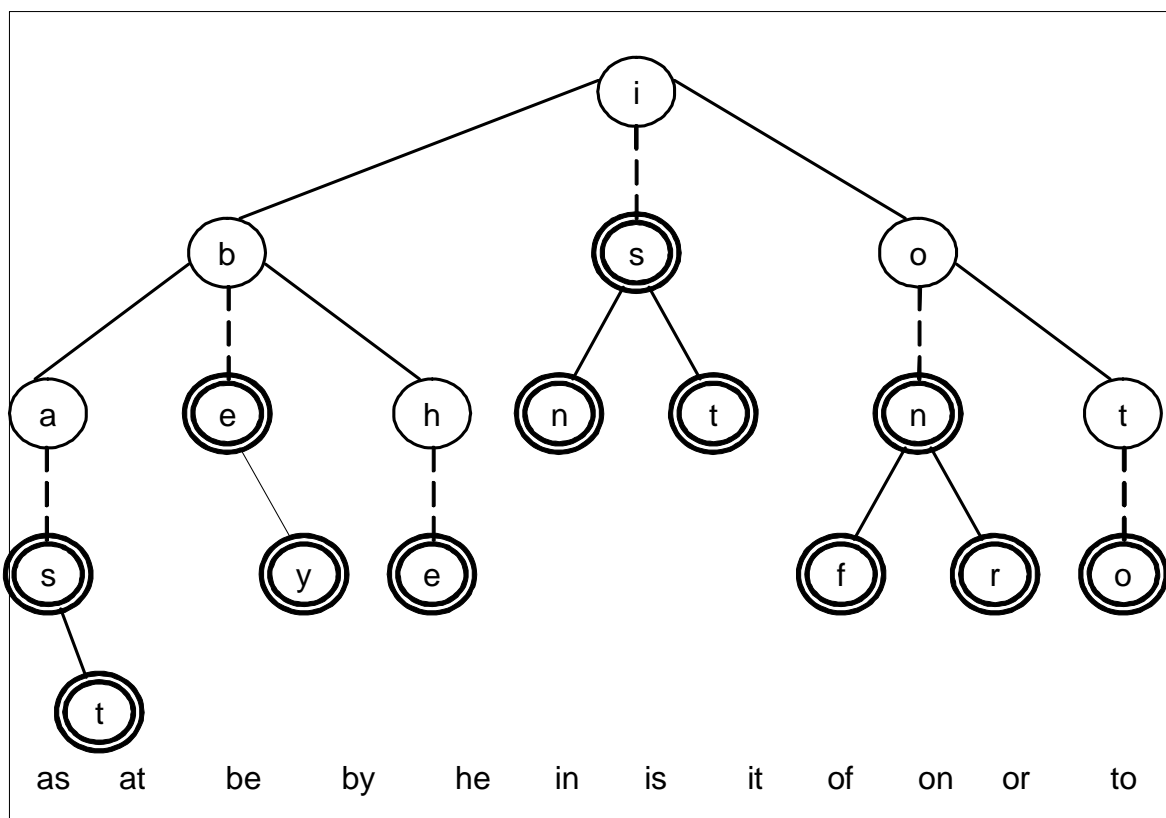


图4-3 三叉树

垂直的虚线代表一个父节点的下面的直接的后继结点。只有父节点和它的直接后继节点才能形成一个数据单元的关键字；“i”和“s”形成关键字“is”，但是“i”和“b”不能形成关键字，因为它们之间仅用一条斜线相连，不具有直接后继关系。图 4-3 中带圈的节点为终止节点，如果查找一个词以终止节点结束，则说明三叉树包含这个词。从根节点开始查找

单词。以搜索单词“is”为例，向下到相等的孩子节点“s”，在两次比较后找到“is”。查找“ax”时，执行三次比较达到首字符“a”，然后经过两次比较到达第二个字符“x”，返回结果是“ax”不在树中。

TernarySearchTrie 本身存储关键字到值的对应关系，可以当作 HashMap 对象来使用。关键字按照字符拆分成了许多节点，以 TSTNode 的实例存在。值存储在 TSTNode 的 data 属性中。TSTNode 的实现如下：

```
public final class TSTNode {
    /** 节点的值 */
    public Data data=null;// data 属性可以存储词原文和词性、词频等相关的信息

    protected TSTNode loNode; //左边节点
    protected TSTNode eqNode; //中间节点
    protected TSTNode hiNode; //右边节点

    protected char splitchar; // 本节点表示的字符
    /**
     * 构造方法
     *
     * @param splitchar 该节点表示的字符
     */
    protected TSTNode(char splitchar) {
        this.splitchar = splitchar;
    }
    public String toString() {
        return "splitchar:"+ splitchar;
    }
}
```

基本的查找词典过程是：输入一个词，返回这个词对应的 TSTNode 对象，如果该词不在词典中则返回空。查找词典的过程中，从树的根节点匹配输入查询词。按字符从前往后匹配 Key。匹配过程如下：

```
protected TSTNode getNode(String key, TSTNode startNode) {
    if (key == null ) {
        return null;
    }
    int len = key.length();
    if (len ==0)
```

```

        return null;
    TSTNode currentNode = startNode; //匹配过程中的当前节点的位置
    int charIndex = 0; //表示当前要比较的字符在 Key 中的位置
    char cmpChar = key.charAt(charIndex);
    int charComp;
    while (true) {
        if (currentNode == null) { //没找到
            return null;
        }
        charComp = cmpChar - currentNode.splitchar;
        if (charComp == 0) { //相等
            charIndex++;
            if (charIndex == len) { //找到了
                return currentNode;
            }
        } else {
            cmpChar = key.charAt(charIndex);
        }
        currentNode = currentNode.eqNode;
    } else if (charComp < 0) { //小于
        currentNode = currentNode.loNode;
    } else { //大于
        currentNode = currentNode.hiNode;
    }
}
}
}

```

三叉树的创建过程也就是在 Trie 树上创建和单词对应的节点。实现代码如下：

//向词典树中加入一个单词的过程

```

private TSTNode addWord(String key) {
    TSTNode currentNode = root; //从树的根节点开始查找
    int charIndex = 0; //从词的开头匹配
    while (true) {
        //比较词的当前字符与节点的当前字符
        int charComp = key.charAt(charIndex) - currentNode.splitchar;
        if (charComp == 0) { //相等
            charIndex++;

```

```

        if (charIndex == key.length()) {
            return currentNode;
        }
        if (currentNode.eqNode == null) {
            currentNode.eqNode = new TSTNode(key.charAt(charIndex));
        }
        currentNode = currentNode.eqNode;
    } else if (charComp < 0) { //小于
        if (currentNode.loNode == null) {
            currentNode.loNode = new TSTNode(key.charAt(charIndex));
        }
        currentNode = currentNode.loNode;
    } else { //大于
        if (currentNode.hiNode == null) {
            currentNode.hiNode = new TSTNode(key.charAt(charIndex));
        }
        currentNode = currentNode.hiNode;
    }
}
}
}

```

相对于查找过程，创建过程在搜索的过程中判断出链接的空值后创建相关的节点，而不是碰到空值后结束搜索过程并返回空值。

同一个词可以有不同的词性，例如“朝阳”既可能是一个“区”，也可能是一个“市”。可以把这些和某个词的词性相关的信息放在同一个链表中。这个链表可以存储在 `TSTNode` 的 `Data` 属性中。

树是否平衡取决于单词的读入顺序。如果按排序后的顺序插入，则生成方式最不平衡。单词的读入顺序对于创建平衡的三叉搜索树很重要，但对于二叉搜索树就不是太重要。通过选择一个排序后的数据单元集合的中间值，并把它作为开始节点，我们可以创建一个平衡的三叉树。可以写一个专门的过程来生成平衡的三叉树词典。

```

/**
 * 在调用此方法前，先把词典数组 k 排好序
 * @param fp 写入的平衡序的词典
 * @param k 排好序的词典数组
 * @param offset 偏移量
 * @param n 长度
 * @throws Exception

```



```
*/  
void outputBalanced(BufferedWriter fp,ArrayList<String> k,int offset, int n){  
    int m;  
    if (n < 1) {  
        return;  
    }  
    m = n >> 1; //m=n/2  
  
    String item= k.get(m + offset);  
  
    fp.write(item);//把词条写入到文件  
    fp.write("\n");  
  
    outputBalanced(fp,k,offset, m); //输出左半部分  
    outputBalanced(fp,k, offset + m + 1, n - m - 1); //输出右半部分  
}
```

取得平衡的单词排序类似于对扑克洗牌。假想有若干张扑克牌，每张牌对应一个单词，先把牌排好序。然后取最中间的一张牌，单独放着。剩下的牌分成了两摞。左边一摞牌中也取最中间的一张放在取出来的那张牌后面。右边一摞牌中也取最中间的一张放在取出来的牌后面，依次类推。

4.3 中文分词的原理

中文分词就是对中文断句，这样能消除文字的部分歧义。除了基本的分词功能，为了消除歧义还可以进行更多的加工。中文分词可以分成如下几个子任务：

- 分词：把输入的标题或者文本内容等分成词。
- 词性标注(POS)：给分出来的词标注上名词或动词等词性。词性标注可以部分消除词的歧义，例如“行”作为量词和作为形容词表示的意思不一样。
- 语义标注：把每个词标注上语义编码。

很多分词方法都借助词库。词库的来源是语料库或者词典，例如“人民日报语料库”或者《现代汉语大词典》。

为了探索适合中国国情的中文分词，不妨借鉴一下汽车产业曾经发生过的事情。日本的汽车充电电池公司使用全自动化生产线，大部分工作都交给机器人来完成，这样的生产线建一条至少需要数千万元的投资，可位于深圳的比亚迪公司把生产线分解成一个个可以人工完

成的工序，在最后容易产生误差的环节则设计了很多简单实用的夹具来保证质量。因为加工后的语料库往往很稀缺，为了使得人工可调整分词结果，这里所有的词典都采用方便人工编辑的文本格式。

中文分词的两类方法：

- 机械匹配的方法：例如正向最大长度匹配(Forward Maximum Match)的方法和逆向最大长度匹配(Reverse Maximum Matching)的方法。
- 统计的方法：例如最大概率分词方法和最大熵的分词方法等。

正向最大长度匹配的分词方法实现起来很简单。每次从词典找和待匹配串前缀最长匹配的词，如果找到匹配词，则把这个词作为切分词，待匹配串减去该词，如果词典中没有词匹配上，则按单字切分。例如，Trie 树结构的词典中包括如下的 8 个词语：

大 大学 大学生 活动 生活 中 中心 心

为了形成平衡的 Trie 树，把词先排序，排序后为：

中 中心 大 大学 大学生 心 活动 生活

按平衡方式生成的词典 Trie 树如图 4-4 所示，其中粗黑显示的节点可以做为匹配终止节点：

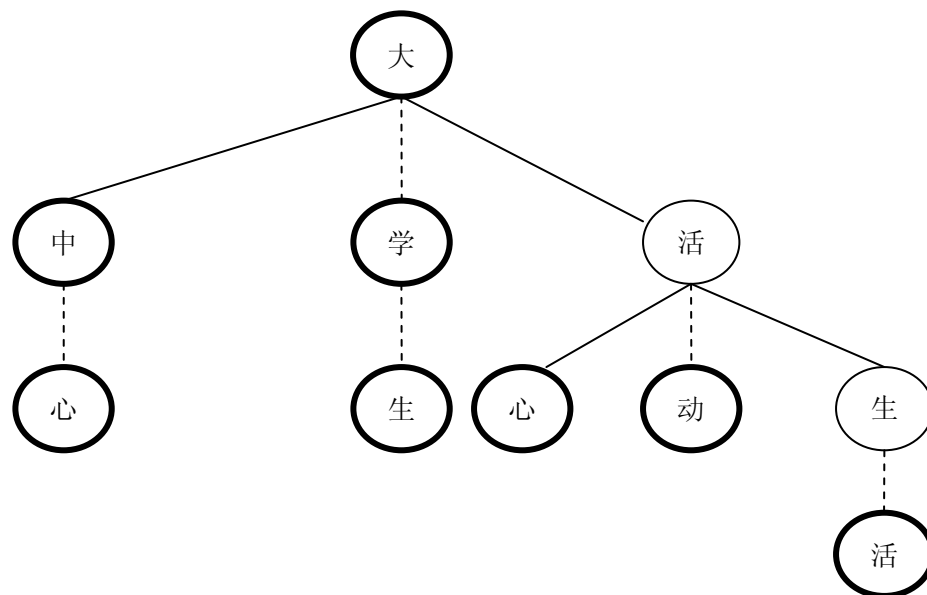


图 4-4 三叉树

输入：“大学生活动中心”，首先匹配出“大学生”，然后匹配出“活动”，最后匹配出“中

心”。切分过程如表 4-2:

已匹配上的结果	待匹配串
NULL	大学生活动中心
大学生	活动中心
大学生/活动	中心
大学生/活动/中心	NULL

表 4-2 正向最大长度匹配切分过程表

最后分词结果为：“大学生/活动/中心”。

在最大长度匹配的分词方法中,需要用到从指定字符串返回指定位置的最长匹配词的方法。例如,当输入串是“大学生活动中心”,则返回“大学生”这个词,而不是返回“大”或者“大学”。从 Trie 树搜索最长匹配单词的方法如下:

```
public String matchLong(String key,int offset) { //输入字符串和匹配的起始位置
    String ret = null;
    if (key == null || rootNode == null || "".equals(key)) {
        return ret;
    }
    TSTNode currentNode = rootNode;
    int charIndex = offset;
    while (true) {
        if (currentNode == null) {
            return ret;
        }
        int charComp = key.charAt(charIndex) - currentNode.splitter;

        if (charComp == 0) {
            charIndex++;

            if(currentNode.data != null){
                ret = currentNode.data; //候选最长匹配词
            }
            if (charIndex == key.length()) {
                return ret; //已经匹配完
            }
        }
    }
}
```

```

        }
        currentNode = currentNode.eqNode;
    } else if (charComp < 0) {
        currentNode = currentNode.loNode;
    } else {
        currentNode = currentNode.hiNode;
    }
}
}
}

```

测试 matchLong 方法:

```

String sentence = "大学生活动中心";//输入字符串
int offset = 0;//匹配的开始位置
String ret = dic.matchLong(sentence,offset);
System.out.println(sentence+" match:"+ret);

```

返回结果:

大学生活动中心 match:大学生

正向最大长度分词的实现代码如下:

```

public void wordSegment(String sentence) { //传入一个字符串作为要处理的对象
    int senLen = sentence.length(); //首先计算出传入的字符串的字符长度
    int i=0; //控制匹配的起始位置

    while(i < senLen) { //如果 i 小于此字符串的长度就继续匹配
        String word = dic.matchLong(sentence, i); //正向最大长度匹配
        if(word!=null) { //已经匹配上
            //下次匹配点在这个词之后
            i += word.length();
            //如果这个词是词库中的那么就打印出来
            System.out.print(word + " ");
        }
        else { //如果在词典中没有找到匹配上的词，就按单字切分
            word = sentence.substring(i, i+1);
            //打印一个字
            System.out.print(word + " ");
            ++i; //下次匹配点在这个字符之后
        }
    }
}

```

```
    }  
  }  
}
```

因为采用了 Trie 树结构查找单词，所以和用 HashMap 查找单词的方式比较起来，这种实现方法代码更简单，而且切分速度更快。例如：“有意见分歧”这句话，正向最大长度切分的结果是：“有意/见/分歧”，逆向最大长度切分的结果是：“有/意见/分歧”。因为汉语的主干成分后置，所以逆向最大长度切分的精确度稍高。另外一种最少切分的方法是使每一句中切出的词数最小。

4.4 中文分词流程与结构

中文分词总体流程与结构如图 4-5：

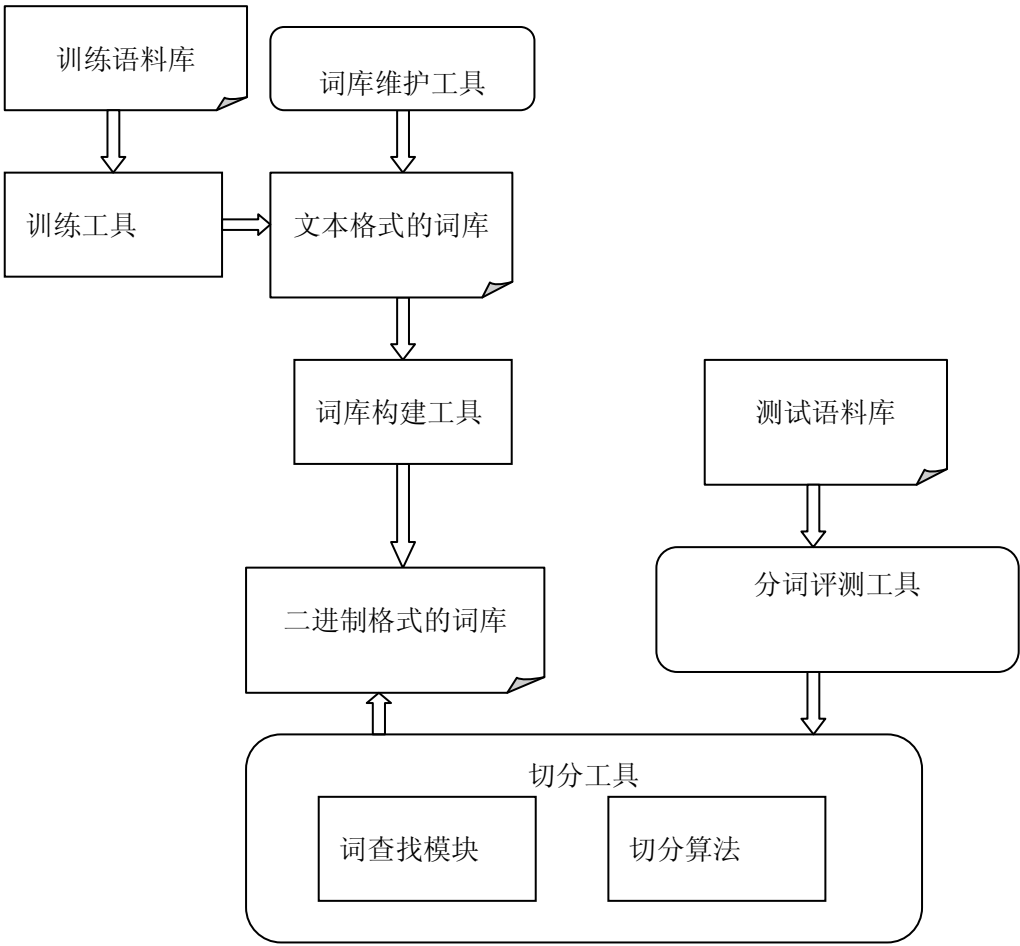


图 4-5 中文分词结构图

简化版本的中文分词切分过程如下：

1. 生成全切分词图：根据基本词库对句子进行全切分，并且生成一个邻接链表表示的词图；
2. 计算最佳切分路径：在这个词图的基础上，运用动态规划算法生成切分最佳路径；
3. 词性标注：可以采用 HMM 方法；
4. 未登录词识别：应用规则识别未登录词；
5. 按需要的格式输出结果。

复杂版本的中文分词切分过程如下：

1. 对输入字符串切分成句子：对一段文本进行切分，首先是依次从这段文本里面切分出一个句子出来，然后再对这个句子进行切分；
2. 原子切分：对于一个句子的切分，首先是通过原子切分，将整个句子切分成一个个的原子单元(即不可再切分的形式，例如 ATM 这样的英文单词可以看成不可再切分的)；
3. 生成全切分词图：根据基本词库对句子进行全切分，并且生成一个邻接链表表示的词图；
4. 计算最佳切分路径：在这个词图的基础上，运用动态规划算法生成切分最佳路径；
5. 未登录词识别：进行中国人名、外国人名、地名、机构名等未登录名词的识别；
6. 重新计算最佳切分路径；
7. 词性标注：可以采用 HMM 方法或最大熵方法等；
8. 根据规则调整切分结果：根据每个分词的词形以及词性进行简单的规则处理，如：日期分词的合并；
9. 按需要的格式输出结果：例如输出成 Lucene 需要的格式。

4.5 全切分词图

为了消除分词中的歧异，提高切分准确度，需要找出一句话所有可能的词，生成全切分词图。

4.5.1 保存切分词图

如果待切分的字符串有 m 个字符，考虑每个字符左边和右边的位置，则有 $m+1$ 个点对应，点的编号从 0 到 m 。把候选词看成边，可以根据词典生成一个切分词图。切分词图是一

个有向正权重的图。“有意见分歧”这句话的切分词图如图 4-5 所示：

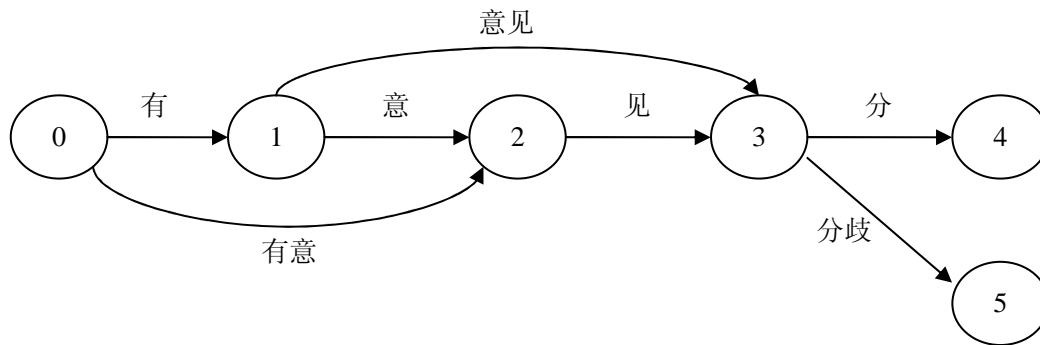


图 4-5 中文分词切分路径

在“有意见分歧”的切分词图中：“有”这条边的起点是 0，终点是 1；“有意”这条边的起点是 0，终点是 2；依次类推。切分方案就是从源点 0 到终点 5 之间的路径。存在两条切分路径：

路径 1： 0—1—3—5 对应切分方案 S_1 ： 有/ 意见/ 分歧/

路径 2： 0—2—3—5 对应切分方案 S_2 ： 有意/ 见/ 分歧/

切分词图中的边都是词典中的词，边的起点和终点分别是词的开始和结束位置。

```

public class CnToken{
    public String termText;//词
    public int start;//词的开始位置
    public int end;//词的结束位置
    public int freq;//词在语料库中出现的频率
    public CnToken(int vertexFrom, int vertexTo, String word) {
        start = vertexFrom;
        end = vertexTo;
        termText = word;
    }
}

```

邻接表表示的切分词图由一个链表表示的数组组成。首先实现一个单向链表：

```

public class TokenLinkedList implements Iterable<TokenInf> {
    public static class Node {

```

```
public TokenInf item;
public Node next;

Node(TokenInf item) {
    this.item = item;
    next = null;
}
}

private Node head;

public TokenLinkedList() {
    head = null;
}

public void put(TokenInf item) {
    Node n = new Node(item);
    n.next = head;
    head = n;
}

public Node getHead() {
    return head;
}

public Iterator<TokenInf> iterator() { //迭代器
    return new LinkIterator(head);
}

private class LinkIterator implements Iterator<TokenInf> {
    Node itr;

    public LinkIterator(Node begin) {
        itr = begin;
    }

    public boolean hasNext() {
```



```

        return itr != null;
    }

    public TokenInf next() {
        if (itr == null) {
            throw new NoSuchElementException();
        }
        Node cur = itr;
        itr = itr.next;
        return cur.item;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}

public String toString() {
    StringBuilder buf = new StringBuilder();
    Node cur = head;

    while (pCur != null) {
        buf.append(cur.item.toString());
        buf.append('\t');
        cur = cur.next;
    }

    return buf.toString();
}
}

```

为了方便用动态规划的方法计算最佳前驱词，在此单向链表的基础上形成逆向邻接表：

```

public class AdjList {
    private TokenLinkedList list[]; // AdjList 的图结构

    /**
     * 构造方法：分配空间

```

```

    */
    public AdjList(int verticesNum) {
        list = new TokenLinkedList[verticesNum];

        // 初始化数组中所有的链表
        for (int index = 0; index < verticesNum; index++) {
            list[index] = new TokenLinkedList();
        }
    }

    public int getVerticesNum() {
        return list.length;
    }

    /**
     * 增加一个边到图中
     */
    public void addEdge(TokenInf newEdge) {
        list[newEdge.end].put(newEdge);
    }

    /**
     * 返回一个迭代器，包含以指定点结尾的所有的边
     */
    public Iterator<TokenInf> getAdjacencies(int vertex) {
        TokenLinkedList ll = list[vertex];
        if (ll == null)
            return null;
        return ll.iterator();
    }
}

```

4.5.2 形成切分词图

首先从词典中形成以某个字符串的前缀开始的词集合。例如，以“中华人民共和国成立了”这个字符串前缀开始的词集合是“中”、“中华”、“中华人民共和国”，一共三个词。这三个词都存在于当前的词典中。如果要找出指定位置开始的所有词，下面是匹配的方法：

```
public static class PrefixRet {
    public ArrayList<String> values;
    public int end; //记录下次匹配的开始位置
}

//如果匹配上则返回 true, 否则返回 false
public boolean getMatch(String sentence, int offset, PrefixRet prefix) {
    if (sentence == null || rootNode == null || "".equals(sentence)) {
        return false;
    }
    boolean match = matchEnglish(offset, sentence, prefix);
    if (match) {
        return true;
    }

    match = matchNum(offset, sentence, prefix);
    if (match) {
        return true;
    }

    prefix.end = offset+1;
    ArrayList<String> ret = new ArrayList<String>();
    TSTNode currentNode = rootNode;
    int charIndex = offset;
    while (true) {
        if (currentNode == null) {
            prefix.values = ret;
            if (ret.size() > 0) {
                return true;
            }
            return false;
        }
        int charComp = sentence.charAt(charIndex) - currentNode.splitChar;
        if (charComp == 0) {
            charIndex++;
            if (currentNode.data != null) {
                //System.out.println(currentNode.data);
                ret.add(currentNode.data);
            }
        }
    }
}
```

```

    }
    if (charIndex == sentence.length()) {
        prefix.values = ret;
        if (ret.size() > 0) {
            return true;
        }
        return false;
    }
    currentNode = currentNode.eqNode;
} else if (charComp < 0) {
    currentNode = currentNode.loNode;
} else {
    currentNode = currentNode.hiNode;
}
}
}
}

```

对于英文和数字等需要特殊处理。“ATM 柜员机” 这个字符串前缀开始的词集合是“ATM”。这里的“ATM”并不一定存在于当前的词典中。为了区分这两种情况，可以把每类需要特殊处理的词放在单独的过程执行。

//匹配英文的过程。返回第一个不是英文的位置。

```
int matchEnglish (int start, String sentence)
```

//匹配数字的过程。返回第一个不是数字的位置。

```
int matchNum (int start, String sentence)
```

然后在统一的过程调用 matchEnglish 和 matchNum。

```

getMatch(String sentence, int offset){
    int ret= matchEnglish(offset,key);
    if(ret>offset)
        return EnglishWord;
    ret = matchNum(offset,key);
    if(ret>offset)
        return NumWord;
    //匹配普通词;
}

```

这里还要用到对字符类型的判断。字符类型有字母、数字、中文。

```
public enum CharType {
```

```

/** char type: SINGLE byte */
SINGLE,
/** char type: delimiter */
DELIMITER,
/** char type: Chinese Char */
CHINESE,
/** char type: letter */
LETTER,
/** char type: chinese number */
NUM,
/** char type: index */
INDEX,
/** char type: other */
OTHER
}

```

可以用查表法来快速判断字符类型。

通过查词典形成切分词图的主体过程：

```

for(int i=0;i<len;){
    boolean match = dict.getMatch(sentence, i, wordMatch);//到词典中查询
    if (match) { // 已经匹配上
        for (String word:wordMatch.values) { //把查询到的词作为边加入切分词图中
            j = i+word.length();
            g.addEdge(new CnToken(i, j, 10, word));
        }
        i=wordMatch.end;
    }else{//把单字作为边加入切分词图中
        j = i+1;
        g.addEdge(new CnToken(i,j,1,sentence.substring(i,j)));
        i=j;
    }
}
}

```

4.6 概率语言模型的分词方法

从统计思想的角度来看，分词问题的输入是一个字串 $C=c_1, c_2, \dots, c_n$ ，输出是一个词串

$S = w_1, w_2, \dots, w_m$ ，其中 $m \leq n$ 。对于一个特定的字符串 C ，会有多个切分方案 S 对应，分词的任务就是在这些 S 中找出概率最大的一个切分方案。也就是对输入字符串切分出最有可能的词序列。

$$\text{Seg}(c) = \arg\max_{S \in G} P(S | C) = \arg\max_{S \in G} \frac{P(C | S)P(S)}{P(C)}$$

例如对于输入字符串 C “有意见分歧”，有 S_1 和 S_2 两种切分可能：

S_1 : 有/ 意见/ 分歧/

S_2 : 有意/ 见/ 分歧/

计算条件概率 $P(S_1 | C)$ 和 $P(S_2 | C)$ ，然后采用概率大的值对应的切分方案。根据贝叶斯

$$\text{公式，有： } P(S | C) = \frac{P(C | S) \times P(S)}{P(C)}$$

其中 $P(C)$ 是字串在语料库中出现的概率，只是一个用来归一化的固定值。从词串恢复到汉字串的概率只有唯一的一种方式，所以 $P(C | S) = 1$ 。因此，比较 $P(S_1 | C)$ 和 $P(S_2 | C)$ 的大小变成比较 $P(S_1)$ 和 $P(S_2)$ 的大小。

概率语言模型分词的任务是：在全切分所得的所有结果中求某个切分方案 S ，使得 $P(S)$ 为最大。那么，如何来表示 $P(S)$ 呢？为了容易实现，假设每个词之间的概率是上下文无关的，则：

$$P(S) = P(w_1, w_2, \dots, w_m) \approx P(w_1) \times P(w_2) \times \dots \times P(w_m) \propto \log P(w_1) + \log P(w_2) + \dots + \log P(w_m)$$

其中，对于不同的 S ， m 的值是不一样的，一般来说 m 越大， $P(S)$ 会越小。也就是说，分出的词越多，概率越小。这符合实际的观察，如最大长度匹配切分往往会使得 m 较小。计算任意一个词出现的概率如下：

$$P(w_i) = \frac{w_i \text{ 在语料库中的出现次数 } n}{\text{语料库中的总词数 } N}$$

$$\text{因此 } \log P(w_i) = \log(\text{Freq}_w) - \log N$$

这个 $P(S)$ 的计算公式也叫做基于一元模型的计算公式，它综合考虑了切分出的词数和词

频。词数少，而词频高的切分方案概率更高。

从另外一个角度来看，计算最大概率等于求切分词图的最短路径。但是这里不采用 Dijkstra 算法，而采用动态规划的方法求解最短路径。

常用的词语概率表如表 4-1 所示。

表 4-1 词语概率表

词语	词频	概率
有	180	0.0180
有意	5	0.0005
意见	10	0.0010
见	2	0.0002
分歧	1	0.0001

$$P(S_1) = P(\text{有}) * P(\text{意见}) * P(\text{分歧}) = 1.8 \times 10^{-9}$$

$$P(S_2) = P(\text{有意}) * P(\text{见}) * P(\text{分歧}) = 1 \times 10^{-11}$$

可得 $P(S_1) > P(S_2)$ ，所以选择 S_1 对应的切分。

如何尽快找到概率最大的词串？因为假设每个词之间的概率是上下文无关的。因此满足用动态规划求解所要求的最优子结构性质和无后效性。在动态规划求解的过程中并没有先生成所有可能的切分路径 S_i ，而是求出值最大的 $P(S_i)$ 后，利用回溯的方法直接输出 S_i 。

到节点 Node_i 为止的最大概率称为节点 Node_i 的概率：

$$\text{因此 } P(\text{Node}_i) = P_{\max}(w_1, w_2, \dots, w_i) = \max_{w_j \in \text{prev}(\text{Node}_i)} (P(\text{StartNode}(w_j))) * P(w_j)$$

如果 w_j 的结束节点是 Node_i ，就称 w_j 为 Node_i 的前驱词。这里的 $\text{prev}(\text{Node}_i)$ 就是节点 i 的前驱词集合。比如上面的例子中，候选词“有”就是节点 1 的前驱词，“意见”和“见”都是节点 3 的前驱词。

这里的 $\text{StartNode}(w_j)$ 是 w_j 的开始节点，也是节点 i 的前驱节点。

因此切分的最大概率 $\max(P(S))$ 就是 $P(\text{Node}_m) = P(\text{节点 } m \text{ 的最佳前驱节点}) * P(\text{节点 } m \text{ 的最佳前驱词})$ 。

按节点编号，从前往后计算如下：

$P(\text{Node}_0) = 1$

$P(\text{Node}_1) = P(\text{有})$

$P(\text{Node}_3) = P(\text{Node}_1) * P(\text{意见})$

修改词的描述类成为：

```
public class CnToken {
    public String termText; //词
    public int start; //开始位置
    public int end; //结束位置
    public double logProb; //概率取对数，这样做是为了避免计算结果向下溢出
}
```

为了计算词的概率，需要在词典中保存所有词的总次数。

首先把切分方案抽象成一个邻接矩阵表示的图，然后按节点从左到右计算最佳前驱节点。计算节点 i 的最佳前驱节点实现代码如下：

```
//计算节点 i 的最佳前驱节点，以及它的概率
void getPrev(AdjList g, int i) {
    //得到前驱词的集合
    Iterator<CnToken> it = g.getPrev(i);
    double maxProb = Double.NEGATIVE_INFINITY;
    int maxNode = -1;
    //根据前驱词集合挑选最佳前趋节点
    while(it.hasNext()) {
        CnToken itr = it.next();
        double nodeProb = prob[itr.start] + itr.logProb; //候选节点概率
        if (nodeProb > maxProb) {
            //候选节点概率最大的开始节点就是最佳前趋节点
            maxNode = itr.start;
            maxProb = nodeProb;
        }
    }
}
```



```

        prob[i] = maxProb;//节点概率
        prevNode[i] = maxNode;//最佳前驱节点
    }

```

找出最大概率的切分路径整体计算流程如下：

```

public static ArrayList<Integer> maxProb(AdjList g) {
    int[] prevNode = new int[g.verticesNum]; //最佳前驱节点
    double[] prob = new double[g.verticesNum]; //节点概率
    prob[0] = 0;//节点 0 的初始概率是 1,取 log 后是 0

    //按节点求最佳前驱
    for (int index = 1; index < g.verticesNum; index++) {
        //求出最大前驱
        getBestPrev(g,index,prevNode,prob);
    }

    ArrayList<Integer> ret = new ArrayList<Integer>();
    //从右向左取最佳前驱节点，通过回溯发现最佳切分路径
    for(int i=(g.verticesNum-1);i>0;i=prevNode[i]){
        ret.add(i);
    }
    return ret;
}

```

上面的计算中假设相邻两个词之间是上下文无关的。但实际情况并不如此，例如：如果前面一个词是数词，后面一个词更有可能是量词。如果前后两个词都只有一种词性，则可以利用词之间的搭配信息对分词决策提供帮助。

```

while (it.hasNext()) {
    CnToken itr = it.next();
    long currentCost = itr.cost;
    int transProb = getTransProb(itr.type, i.type);//前一个词到下一个词的转移概率
    currentCost += transProb;//注意不要让 currentCost 的值溢出

    if (currentCost > maxFee) {
        maxID = itr;
        maxFee = currentCost;
    }
}

```

4.7 N 元分词方法

在介绍 N 元模型之前，让我们先来做个香农游戏(Shannon Game)。我们给定一个词，然后猜测下一个词是什么。当我说“NBA”这个词时，你想到下一个词是什么呢？我想大家有可能会想到“篮球”，基本上不会有人想到“足球”吧。

切分出来的词序列越通顺，越有可能是正确的切分方案。N 元模型来衡量词序列搭配的合理性。N 元模型指在句子中在 n 个单词序列后出现的单词 w 的概率。

$$P(S)=P(w_1, w_2, \dots, w_n)=P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)\dots P(w_n|w_1 w_2 \dots w_{n-1})$$

但是这种方法存在两个致命的缺陷：一个缺陷是参数空间过大，不可能实用化；另外一个缺陷是数据稀疏严重。为了解决这个问题，我们引入了马尔科夫假设：一个词的出现仅仅依赖于它前面出现的有限的一个或者几个词。

如果简化成一个词的出现仅依赖于它前面出现的一个词，那么就称为二元模型(Bigram)。即：

$$P(S) = P(w_1, w_2, \dots, w_n) = P(w_1) P(w_2|w_1) P(w_3|w_1, w_2) \dots P(w_n|w_1 w_2 \dots w_{n-1})$$

$$\approx P(w_1) P(w_2|w_1) P(w_3|w_2) \dots P(w_n|w_{n-1})$$

如果简化成一个词的出现仅依赖于它前面出现的两个词，就称之为三元模型(Trigram)。如果一个词的出现不依赖于它前面出现的词，叫做一元模型(Unigram)，也就是已经介绍过的概率语言模型的分词方法。

在实践中用的最多的就是二元和三元了，而且效果很不错。高于四元的用的很少，因为训练它需要更庞大的语料，而且数据稀疏严重，时间复杂度高，精度却提高的不多。

二元模型考虑一个单词后出现另外一个单词的概率，是 n 元模型中的一种。例如：一般来说，“中国”之后出现“北京”的概率大于“中国”之后出现“北海”的概率，也就是：

$$P(\text{北京}|\text{中国}) > P(\text{北海}|\text{中国})$$

$$P(w_i|w_{i-1}) \approx \text{freq}(w_{i-1}, w_i) / \text{freq}(w_{i-1})$$

二元词表的格式是“左词@右词:组合频率”，例如：

中国@北京:100

中国@北海:1

可以把二元词表看成是基本词表的常用搭配。分词初始化时，先加载基本词表，对每个词编号，然后加载二元词表，只存储词的编号。

对于拼音转换等歧义较多的情况下也可以采用三元模型(Trigram)，例如：

$P(\text{海淀}|\text{中国,北京}) > P(\text{海龙}|\text{中国,北京})$

$P(w_i|w_{i-2},w_{i-1}) \approx \text{freq}(w_{i-2},w_{i-1},w_i) / \text{freq}(w_{i-2},w_{i-1})$

因为有些词作为开始词的可能性比较大，例如“在那遥远的地方”，“在很久以前”，这两个短语都以“在”这个词作为开始词。因此，在实际的 n 元分词过程中，增加虚拟的开始节点(Start)和结束节点(End)，分词过程中考虑 $P(\text{在}|\text{Start})$ 。因此，如果把“有意见分歧”当成一个完整的输入，分词结果实际是：“Start/ 有/ 意见/ 分歧/ End”。

想象在下跳棋，跳 2 次涉及到 3 个位置。二元连接中的前后 2 个词涉及到 3 个节点。二元连接的实现代码如下：

```
//计算节点 i 的最佳前驱节点
void getBestPrev(AdjList g,int i) {
    Iterator<CnToken> it1 = g.getPrev(i);//得到一级前驱词集合
    double maxProb = Double.NEGATIVE_INFINITY;
    int maxPrev1 = -1;
    int maxPrev2 = -1;

    while(it1.hasNext()) {
        CnToken t1 = it1.next();
        //得到一级前驱词对应的二级前驱词集合
        Iterator<CnToken> it2 = g.getPrev(t1.start);
        while(it2.hasNext()) {
            CnToken t2 = it2.next();

            int bigramFreq=getBigramFreq(t2,t1);//从二元词典找二元频率
            //平滑后的二元概率
            double biProb = lambda1*t1.freq + lambda2*(bigramFreq/t2.freq);
            double nodeProb = prob[t2.start]+(Math.log(biProb));

            if (nodeProb > maxProb) { //概率最大的算作最佳前趋
                maxPrev1 = t1.start;
                maxPrev2 = t2.start;
                maxProb = nodeProb;
            }
        }
    }
}
```

```

    prob[i] = maxProb;
    prev1Node[i] = maxPrev1;
    prev2Node[i] = maxPrev2;
}

```

查找 N 元词典的方法有：可以采用 Trie 树的形式来存放 N 元模型的参数。与词典 Trie 树的区别在于：词典 Trie 树上每个结点对应一个汉字，而 N 元模型 Trie 树的一个结点对应一个词。或者可以把搭配信息存放在词典 Trie 树的叶子节点上。存储从词编号到频率的映射，采用折半查找。

```

public class BigramMap {
    public int[] keys;//词编号
    public int[] vals;//频率
}

```

在自然语言处理中，N 元模型可以应用于字符，衡量字符之间的搭配，或者词，衡量词之间的搭配。应用于字符的例子：可以应用于编码识别，将要识别的文本按照 GB 码和 BIG5 码分别识别成不同的汉字串，然后计算其中所有汉字频率的乘积。取乘积大的一种编码。

4.8 语料库

计算值用 log 形式存储。

所有的 n 元概率都在 log 空间计算。这样做是为了避免向下溢出。另外一个考虑是：加法比乘法快。

$$P1 * P2 * P3 * P4 = \text{EXP}(\log P1 + \log P2 + \log P3 + \log P4)$$

ARPA 文件格式是：

n 元 log(词序列的估计概率) 词序列 回退值

语言建模工具有：

- C++实现的 SRILM(<http://www-speech.sri.com/projects/srilm/>)，它，可运行在 Windows 或 Linux。
- Java 实现的 Kyoto Language Modeling Toolkit(<http://www.phontron.com/kylm/>)。

通过困惑度(Perplexity)来衡量语言模型。

困惑度是和一个语言事件的不确定性相关的度量。考虑词级别的困惑度。“行”后面可以跟的词有“不行”、“代码”、“善”、“走”。所以“行”的困惑度较高。但有些词不太

可能跟在“行”后面，例如“您”、“分”。而有些词的困惑度比较低，例如“康佳”等专有名词，后面往往跟着“彩电”等词。语言模型的困惑度越低越好，相当于有比较强的消除歧义能力。如果从更专业的语料库学习出语言模型，则有可能获得更低的困惑度，因为专业领域中的词搭配更加可预测。

4.9 新词发现

词典中没有的，但是结合紧密的字或词有可能组成一个新词。比如：“水立方”如果不在词典中，可能会切分成两个词“水”和“立方”。如果在一篇文档中“水”和“立方”结合紧密，则有“水立方”可能是一个新词。可以用信息熵来度量两个词的结合紧密程度。信息熵的一般公式是：

$$I(X,Y) = \log_2 \frac{P(X,Y)}{P(X)P(Y)}$$

如果 x 和 y 的出现相互独立，则 $P(x,y)$ 的值和 $p(x)p(y)$ 的值相等， $I(x,y)$ 为 0。如果 x 和 y 密切相关， $P(x,y)$ 将比 $P(x)P(y)$ 大很多， $I(x,y)$ 值也就远大于 0。如果 x 和 y 的几乎不会相邻出现，而它们各自出现的概率又比较大，那么 $I(x,y)$ 将取负值，这时候 x 和 y 负相关。设 $f(C)$ 是词 C 出现的次数， N 是文档的总词数，则：

$$P(C_1, C_2) = P(C_1) * P(C_2 | C_1) = \frac{f(C_1)}{N} * \frac{f(C_1 C_2)}{f(C_1)} = \frac{f(C_1 C_2)}{N}$$

$$\text{因此，两个词的信息熵 } I(C_1, C_2) = \log_2 N + \log_2 \frac{f(C_1 C_2)}{f(C_1) f(C_2)}$$

$$= \log_2 N + \log_2 f(C_1 C_2) - \log_2 f(C_1) - \log_2 f(C_2)$$

两个相邻出现的词叫做二元连接串。

```
public class Bigram {
    String one;//上一个词
    String two;//下一个词
    private int hashvalue = 0;

    Bigram(String first, String second) {
        this.one = first;
        this.two = second;
        this.hashvalue = (one.hashCode() ^ two.hashCode());
    }
}
```

```
}
}
```

从二元连接串中计算二元连接串的信息熵的代码如下：

```
int index = 0;
fullResults = new BigramsCounts[table.size()];
Bigrams key;
int freq;//频率
double logn = Math.log((double)n); //文档的总词数取对数
double temp;
double entropy;
int bigramCount; //f(c1,c2)
for( Entry<Bigrams,int[]> e : table.entrySet()){//计算每个二元连接串的信息熵
    key = e.getKey();
    freq1 = oneFreq.get(key.one).freq;
    freq2 = oneFreq.get(key.two).freq;
    temp = Math.log((double)freq1) + Math.log((double)freq2);
    bigramCount = (e.getValue())[0];
    entropy = logn+Math.log((double)bigramCount) - temp;//信息熵
    fullResults[index++] =
        new BigramsCounts(
            bigramCount,
            entropy,
            key.one,
            key.two);
}
```

新词语有些具有普遍意义的构词规则，例如“模仿秀”由“动词+名词”组成。统计的方法和规则的方法结合对每个文档中重复子串组成的候选新词打分候选新词打分，超过阈值的候选新词选定为新词。此外，可以用 Web 信息挖掘的方法辅助发现新词：网页锚点上的文字可能是新词，例如“美甲”。另外，可以考虑先对文档集合聚类，然后从聚合出来的相关文档中挖掘新词。

4.10 未登录词识别

有人问道：南京市长叫**江大桥**？你怎么知道的？因为看到一个标语——南京市长江大桥欢迎您。

未登录词在英文中叫做 Out Of Vocabulary (简称 OOV)词。常见的未登录词包括：人名、地名、机构名。

对识别未登录词有用的信息：

- 未登录词所在的上下文。例如：“**教授”，这里“教授”是人名的下文；“邀请**”，这里“邀请”是人名的上文。
- 未登录词本身的概率。例如：不依赖上下文，直观的来看，“刘宇”可能是个人名，“史光”不太可能是个人名。采用未登录词的频率作为这种可能性的衡量依据。“刘宇”作为人名的概率是：“刘宇”作为人名出现的次数/人名出现的总次数。

例如：我爸叫李刚。 这里“动词 + 姓 + 名 + 标点符号”组成了一个识别规则。可以根据这个识别规则识别出“李刚”这个人名。这个规则的完整形式是：

动词 + 中国人名 + 标点符号 => 动词 + 姓 + 名 + 标点符号

所以可以通过匹配规则来识别未登录词。为了实现同时查找多个规则，可以把右边的模式组织成 Trie 树，左边的模式作为节点属性。全切分词图匹配上右边的模式后用左边的模式替换。

可以用二元或三元语言模型来整合未登录词本身的概率和未登录词所在的上下文这两种信息。

未登录地名识别过程是：

- 1.选取未登录地名候选串。
- 2.未登录地名特征识别。
- 3.对每个候选未登录地名根据特征判断是否真的地名。判断方法可以用 SVM 二值分类。
- 4.整合地名词图。

4.11 词性标注

词性用来描述一个词在上下文中的作用。例如描述一个概念的词叫做名词，在下文引用这个名词的词叫做代词。有的词性经常会出现一些新的词，例如名词，这样的词性叫做开放式的词性。另外一些词性中的词比较固定，例如代词，这样的词性叫做封闭式的词性。因为存在一个词对应多个词性的现象，所以给词准确的标注词性并不是很容易。比如：“改革”在“中国开始对计划经济体制进行**改革**”这句话中作为一个动词，在“医药卫生**改革**中的经济问题”中作为一个名词。把这个问题抽象出来就是已知单词序列 $W_1 W_2 \dots W_n$ ，给每

个单词标注上词性 $C_1C_2\cdots C_n$ 。

不同的语言有不同的词性标注集。比如英文有反身代词，例如 **myself**，而中文中则没有反身代词。为了方便指明词的词性，可以给每个词性编码。例如《PFR 人民日报标注语料库》中把“形容词”编码成 **a**，名词编码成 **n**，动词编码成 **v**，...

词性标注有小标注集和大标注集。例如小标注集代词都归为一类，大标注集可以把代词进一步分成三类：

- 人称代词：你 我 他 它 你们 我们 他们...
- 疑问代词：哪里 什么 怎么
- 指示代词：这里 那里 这些 那些

采用小标注集比较容易实现，但是太小的标注集可能会导致类型区分度不够。例如在黑白二色世界中，可以通过颜色的深浅来分辨出物体，但是通过七彩颜色可以分辨出更多的物体。

参考《PFR 人民日报标注语料库》的词性编码表，如表 4-2 所示：

表 4-2 词性编码表

代码	名称	举例
a	形容词	最/d 大/a 的/u
ad	副形词	一定/d 能够/v 顺利/ad 实现/v 。/w
ag	形语素	喜/v 煞/ag 人/n
an	名形词	人民/n 的/u 根本/a 利益/n 和/c 国家/n 的/u 安稳/an 。/w
b	区别词	副/b 书记/n 王/nr 思齐/nr
c	连词	全军/n 和/c 武警/n 先进/a 典型/n 代表/n
d	副词	两侧/f 台柱/n 上/f 分别/d 雄踞/v 着/u
dg	副语素	用/v 不/d 甚/dg 流利/a 的/u 中文/nz 主持/v 节目/n 。/w
e	叹词	嗨/e ！/w

代码	名称	举例
f	方位词	从/p 一/m 大/a 堆/q 档案/n 中/f 发现/v 了/u
g	语素	例如 dg 或 ag
h	前接成分	目前/t 各种/r 非/h 合作制/n 的/u 农产品/n
i	成语	提高/v 农民/n 讨价还价/i 的/u 能力/n 。/w
j	简称略语	民主/ad 选举/v 村委会/j 的/u 工作/vn
k	后接成分	权责/n 明确/a 的/u 逐级/d 授权/v 制/k
l	习用语	是/v 建立/v 社会主义/n 市场经济/n 体制/n 的/u 重要/a 组成部分/l 。/w
m	数词	科学技术/n 是/v 第一/m 生产力/n
n	名词	希望/v 双方/n 在/p 市政/n 规划/vn
ng	名语素	就此/d 分析/v 时/ng 认为/v
nr	人名	建设部/nt 部长/n 侯/nr 捷/nr
ns	地名	北京/ns 经济/n 运行/vn 态势/n 喜人/a
nt	机构团体	[冶金/n 工业部/n 洛阳/ns 耐火材料/l 研究院/n]nt
nx	字母专名	A T M/nx 交换机/n
nz	其他专名	德士古/nz 公司/n
o	拟声词	汨汨/o 地/u 流/v 出来/v
p	介词	往/p 基层/n 跑/v 。/w
q	量词	不止/v 一/m 次/q 地/u 听到/v ， /w
r	代词	有些/r 部门/n
s	处所词	移居/v 海外/s 。/w
t	时间词	当前/t 经济/n 社会/n 情况/n

代码	名称	举例
tg	时语素	秋/Tg 冬/tg 连/d 旱/a
u	助词	工作/vn 的/u 政策/n
ud	结构助词	有/v 心/n 栽/v 得/ud 梧桐树/n
ug	时态助词	你/r 想/v 过/ug 没有/v
uj	结构助词的	迈向/v 充满/v 希望/n 的/uj 新/a 世纪/n
ul	时态助词了	完成/v 了/ul
uv	结构助词地	满怀信心/l 地/uv 开创/v 新/a 的/u 业绩/n
uz	时态助词着	眼看/v 着/uz
v	动词	举行/v 老/a 干部/n 迎春/vn 团拜会/n
vd	副动词	强调/vd 指出/v
vg	动语素	做好/v 尊/vg 干/j 爱/v 兵/n 工作/vn
vn	名动词	股份制/n 这种/r 企业/n 组织/vn 形式/n , /w
w	标点符号	生产/v 的/u 5 G/nx 、/w 8 G/nx 型/k 燃气/n 热水器/n
x	非语素字	生产/v 的/u 5 G/nx 、/w 8 G/nx 型/k 燃气/n 热水器/n
y	语气词	已经/d 3 0 /m 多/m 年/q 了/y 。/w
z	状态词	势头/n 依然/z 强劲/a ; /w

以[把][这][篇][报道][编辑][一][下]为例，有 $5 \times 1 \times 1 \times 2 \times 2 \times 2 \times 3 = 120$ 种词性标注可能性，哪种可能性最大？

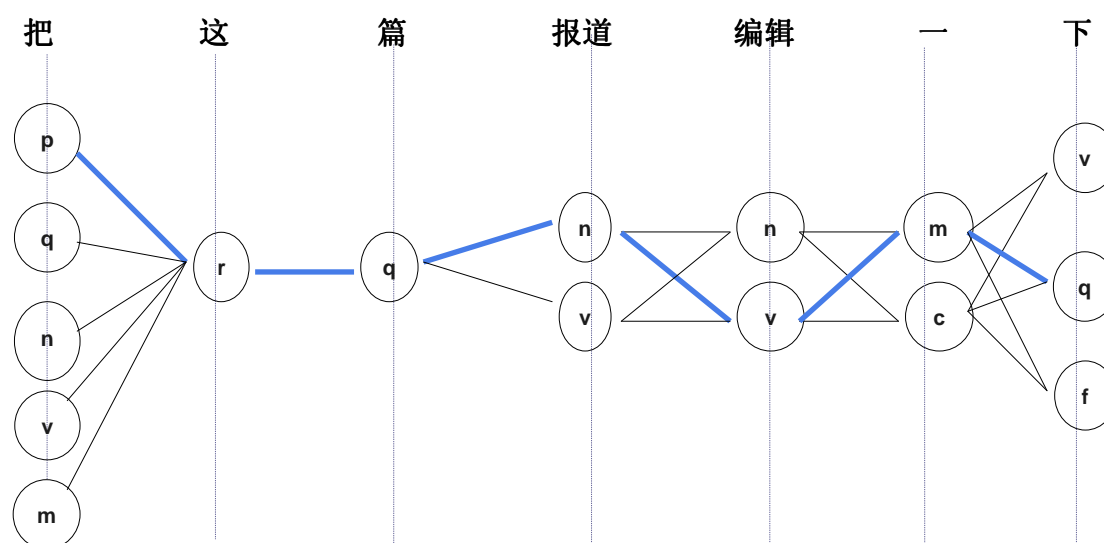


图 4-6 词性标注

解决标注歧义问题最简单的一个方法是从单词所有可能的词性中选出这个词最常用的词性作为这个词的词性，也就是一个概率最大的词性，比如“改革”大部分时候作为一个名词出现，那么可以机械的把这个词总是标注成名词，但是这样标注的准确率会比较低，因为只考虑了频率特征。

考虑词所在的上下文可以提高标注准确率。例如在动词后接名词的概率很大。“推进/改革”中的“推进”是动词，所以后面的“改革”很有可能是名词。这样的特征叫做上下文特征。

隐马尔可夫模型(Hidden Markov Model 简称 HMM)和基于转换的学习方法是两种常用的词型标注方法。这两种方法都整合了频率和上下文两方面的特征来取得好的标注结果。具体来说，隐马尔可夫模型同时考虑到了词的生成概率和词性之间的转移概率。

很多生物也懂得同时利用两种特征信息。例如，箭鼻水蛇是一种生活在水中以吃鱼或虾为生的蛇。它是唯一一种长着触须的蛇类。箭鼻水蛇最前端触须能够感触非常轻微的变动，这表明它可以感触到鱼类移动时产生的细微水流变化。当在光线明亮的环境中，箭鼻水蛇能够通过视觉捕食小鱼。因此它能够同时利用触觉和视觉，也就是说光线的变化和水流的变化信息来捕鱼。

4.11.1 隐马尔可夫模型

词性标注的任务是：给定词序列 $W=w_1, w_2, \dots, w_n$ ，寻找词性标注序列 $T=t_1, t_2, \dots, t_n$ ，最大化概率： $P(t_1, t_2, \dots, t_n | w_1, w_2, \dots, w_n)$ 。

使用贝叶斯公式重新描述这个条件概率：

$$P(t_1, t_2, \dots, t_n) * P(w_1, w_2, \dots, w_n | t_1, t_2, \dots, t_n) / P(w_1, w_2, \dots, w_n)$$

忽略掉分母 $P(w_1, w_2, \dots, w_n)$ ，同时做独立性假设，使用 n 元模型近似计算 $P(t_1, t_2, \dots, t_n)$ 。例如使用二元连接，则有：

$$P(t_1, t_2, \dots, t_n) \propto \prod_{i=1}^n P(t_i | t_{i-1})$$

近似计算 $P(w_1, w_2, \dots, w_n | t_1, t_2, \dots, t_n)$ ：假设一个类别中的词独立于它的邻居，则有：

$$P(w_1, w_2, \dots, w_n | t_1, t_2, \dots, t_n) \propto \prod_{i=1}^n P(w_i | t_i)$$

寻找最有可能的词性标注序列实际的计算公式：

$$P(t_1, t_2, \dots, t_n) * P(w_1, w_2, \dots, w_n | t_1, t_2, \dots, t_n) \propto \prod_{i=1}^n P(t_i | t_{i-1}) * P(w_i | t_i)$$

这里把词 w 叫做显状态，词性 t 叫做隐状态。条件概率 $P(t_i | t_{i-1})$ 叫做转移概率。条件概率 $P(w_i | t_i)$ 叫做发射概率。

抽象来说，基本的马尔可夫模型中的状态之间有转移概率。隐马尔可夫模型中有隐状态和显状态。隐状态之间有转移概率。一个隐状态对应多个显状态。隐状态生成显状态的概率叫做生成概率或者发射概率。在初始概率、转移概率以及发射概率已知的情况下，可以从观测到的显状态序列计算出可能性最大的隐状态序列，这个算法叫做维特比(Viterbi)算法。对于词性标注的问题来说，显状态是分词出来的结果——单词 W ，隐状态是需要标注的词性 C 。词性之间存在转移概率。词性按照某个发射概率产生具体的词。可以把初始概率、转移概率和发射概率一起叫做语言模型。因为它们可以用来评估一个标注序列的概率。采用隐马尔可夫模型标注词性的总体结构如图 4-7 所示。

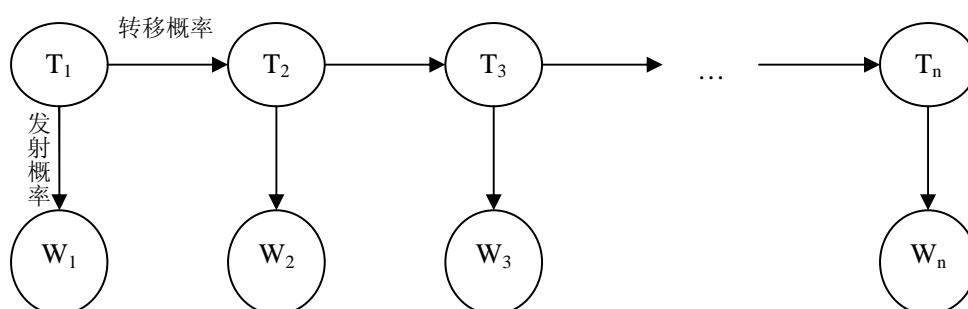


图 4-7 词性标注中的隐马尔可夫模型

举例说明隐马尔可夫模型。假设只有词性：代词(r)、动词(v)、名词(n)和方位词(f)。

有如下一个简化版本的语言模型描述：

start: go(r,1.0) emit(start,1.0)

f: emit(来,0.1) go(n,0.9) go(end,0.1)

v: emit(来,0.4) emit(会,0.3) go(f,0.1) go(v,0.3) go(n,0.5) go(end,0.1)

n: emit(会,0.1) go(f,0.5) go(v,0.3) go(end,0.2)

r: emit(他,0.3) go(v,0.9) go(n,0.1)

这里的 start 和 end 都是虚拟的状态，start 表示开始，end 表示结束。emit 表示发射概率，go 表示转移概率。语言模型中的值可以事前统计出来。中文分词中的语言模型可以从语料库统计出来。

这个语言模型的初始概率向量是表 4-3。

表 4-3 初始概率表

	r	n	f	end
start	1.0	0	0	0

这个初始概率的意思是，代词是每个句子的开始。

转移概率矩阵是表 4-4。

表 4-4 转移概率表

<div> <div></div> <div>下个词性</div> <div>上个词性</div> </div>	start	f	v	n	r	end
start					1	
f				0.9		0.1
v		0.1	0.3	0.5		0.1
n		0.5	0.3			0.2

r			0.9	0.1		
---	--	--	-----	-----	--	--

例如第 3 行表示动词后是名词的可能性比较大，仍然是动词的可能性比较小，所以上个词性是动词，下一个词性是名词的概率是 0.5，而上个词性是动词，下一个词性还是动词的概率是 0.3。

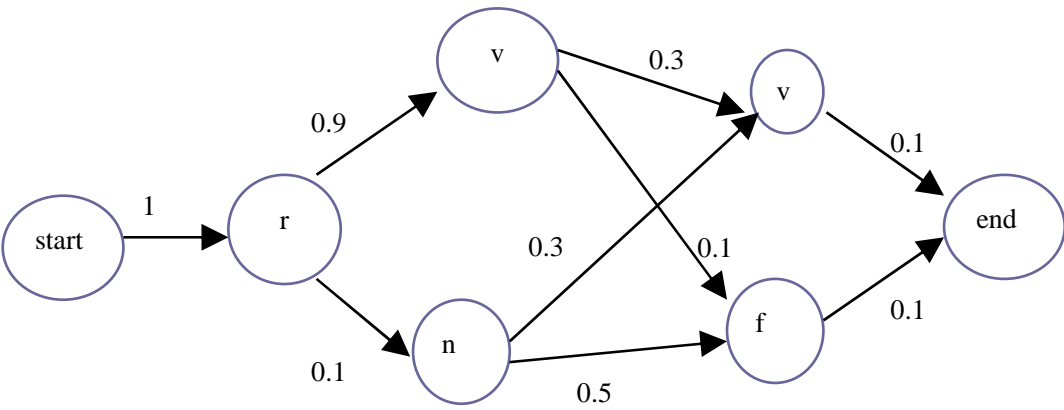


图 4-6 转移概率图

代表的发射概率(混淆矩阵)是：

表 4-5 发射概率表

	他	会	来
f			0.1
v		0.3	0.4
n		0.6	
r	0.3		

以发射概率表的第二行为例：如果一个词是动词，那么这个词是“来”的概率比“会”的概率大。

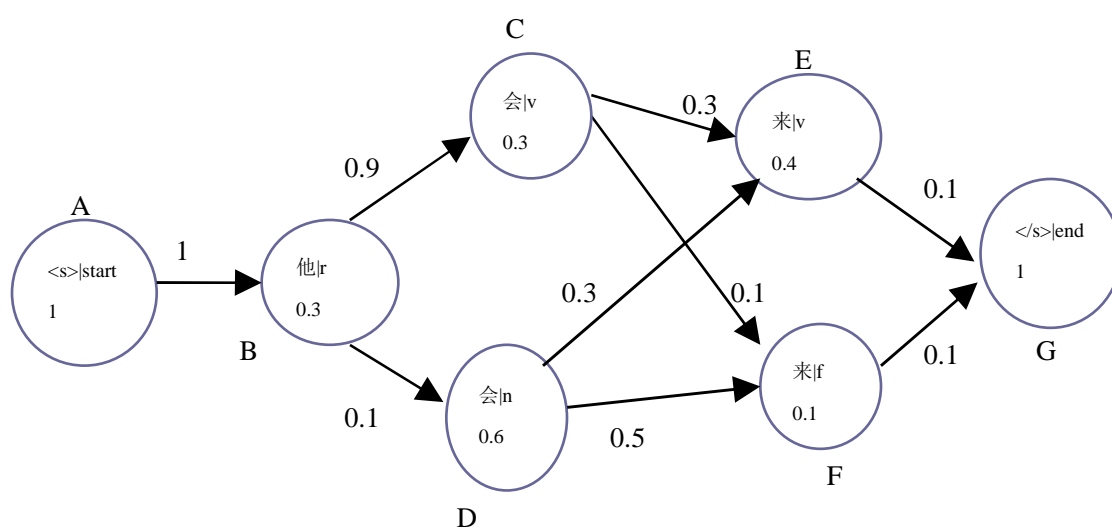


图 4-7 发射概率图

分词后的输入是：[start] [他] [会] [来] [end]。

考虑到某些词性更有可能作为一句话的开始，有些词性更有可能作为一句话的结束。这里增加了开始和结束的虚节点 start 和 end。

每个隐状态和显状态的每个阶段组合成一个如图 4-8 所示的由节点组成的二维矩阵：

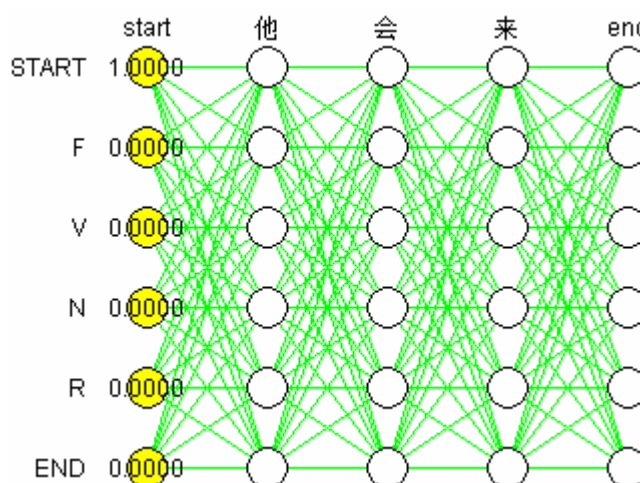


图 4-8 维特比求解格栅

采用动态规划的方法求解最佳标注序列。每个词对应一个求解的阶段，当前节点概率的计算依据是：

- 上一个阶段的节点概率；

- 上一个阶段的节点到当前节点的转移概率；
- 当前节点的发射概率。

动态规划的思想产生了维特比算法。维特比求解方法由两个过程组成：前向累积概率计算过程和反向回溯过程。前向过程按阶段计算。从图上看就是从前向后按列计算。分别叫做阶段“start”、“他”、“会”、“来”、“end”。

在阶段“start”计算：

- $\text{Best}(A) = 1$

在阶段“他”计算：

- $\text{Best}(B) = \text{Best}(A) * P(r|\text{start}) * P(\text{他}|r) = 1 * 1 * .3 = .3$

在阶段“会”计算：

- $\text{Best}(C) = \text{Best}(B) * P(v|r) * P(\text{会}|v) = .3 * .9 * .3 = .081$
- $\text{Best}(D) = \text{Best}(B) * P(n|r) * P(\text{会}|n) = .3 * .1 * .6 = .018$

在阶段“来”计算：

- $\text{Best}(E) = \text{Max} [\text{Best}(C) * P(v|v), \text{Best}(D) * P(v|n)] * P(\text{来}|v) = .081 * .3 * .4 = .00972$
- $\text{Best}(F) = \text{Max} [\text{Best}(C) * P(f|v), \text{Best}(D) * P(f|n)] * P(\text{来}|f) = .081 * .1 * .1 = .00081$

在阶段“end”计算：

- $\text{Best}(G) = \text{Max} [\text{Best}(E) * P(\text{end}|v), \text{Best}(F) * P(\text{end}|f)] * P(</s>|\text{end}) = .00972 * .1 * 1 = .000972$

执行回溯过程发现最佳隐状态(粗黑线节点)，如图 4-9 所示：

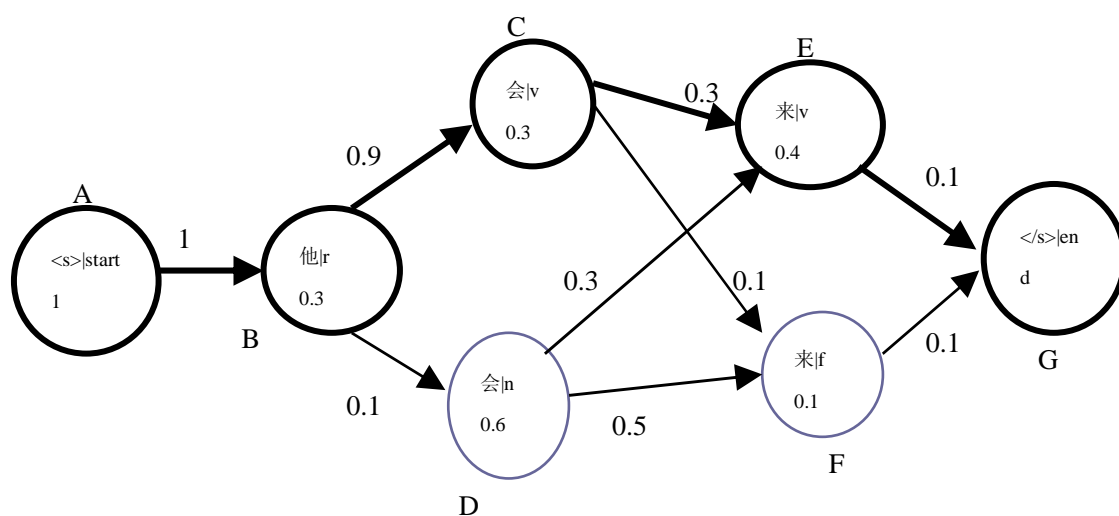


图 4-9 维特比求解过程

G 的最佳前驱节点是 E，E 的最佳前驱节点是 C，C 的最佳前驱节点是 B，B 的最佳前驱节点是 A。所以猜测词性输出：他/r 会/v 来/v。这样消除了歧义，判断出[会]的词性是动词而不是名词，[来]的词性是动词而不是方位词。

然后开始实现维特比算法。维特比算法又叫做在格栅上运行的算法，暗示可以用二维数组存储计算的中间结果，也就是累积概率。但是一个词可能的词性往往只是很少几种，所以一般采用稀疏的方式存储词性。首先定义节点类：

```
public class Node {
    public Node bestPrev; //最佳前驱
    public State tag; //隐状态
    public double prob; //累积概率
}
```

初始化存储累积概率的二维节点：

```
al = new ArrayList[symbols.size()];
for (int i = 0; i < symbols.size(); i++) {
    al[i] = new ArrayList<Node>(states.size());
}
// 添加初始节点，即把发射到 start 概率不为零的节点添加到 al[0]中
for (int j = 0; j < states.size(); j++) {
    double emitProb = states.get(j).emitprob(symbols.get(0));
    if (emitProb > 0) {
        Node tmNode = new Node(states.get(j), emitProb);
        al[0].add(tmNode);
    }
}
```

```

    }
}
// 初始化第一阶段以后的节点
for (int i = 1; i < symbols.size(); i++) {
    for (int j = 0; j < states.size(); j++) {
        double emitProb = states.get(j).emitprob(symbols.get(i));
        if (emitProb > 0) {
            Node tmNode = new Node(states.get(j));
            al[i].add(tmNode);
        }
    }
}
}

```

维特比求解的前向累积过程主要代码如下：

```

// stage 表示显状态序号，跳过 start 状态
for (stage = 1; stage < al.length; stage++) {
    // i1 表示显状态序号 stage 对应的隐状态节点维的序号
    for (i1 = 0; i1 < al[stage].size(); i1++) {
        // i0 表示 i1 上一显状态维的隐状态节点编号
        for (i0 = 0; i0 < al[stage - 1].size(); i0++) {
            sym1 = symbols.get(stage);
            // 上一显状态维的隐状态节点
            Node prevNode = al[stage - 1].get(i0);
            // 当前显状态维的隐状态节点
            Node nextNode = al[stage].get(i1);
            s0 = prevNode.tag; // 上一显状态维的隐状态
            s1 = nextNode.tag; // 当前显状态维的隐状态
            // 上一显状态维的隐状态节点累积概率
            prob = prevNode.prob;
            // 上一显状态维的隐状态到当前显状态维的隐状态的转移概率
            double transprob = s0.transprob(s1);

            prob = prob * transprob;
            emitprob = s1.emitprob(sym1);
            // 这一步的累积概率是
            // 上一步的累积概率 *
            // 上一个状态到这一个状态的转移概率 *

```

```

        //当前节点的发射概率
        prob = prob * emitprob;
        // 找到当前节点的最大累积概率
        if (nextNode.prob <= prob) {
            // 记录当前节点的最大累积概率
            nextNode.prob = prob;
            // 记录当前节点的最佳前驱
            nextNode.bestPrev = prevNode;
        }
    }
}
}
}

```

维特比求解的反向回溯过程用来寻找最佳路径，主要代码如下：

```

public ArrayList<Node> backward() {
    ArrayList<Node> maxNode = new ArrayList<Node>();
    Node endNode=al[symbols.size() - 1].get(0);//结束阶段对应的最有可能的隐状态

    //从后往前找最有可能的隐状态
    for (Node i = endNode; i != null; i = i.bestPrev) {
        maxNode.add(i);// 被选中的隐状态加入到结果路径
    }
    return maxNode;
}

```

统计人民日报语料库中词性间的转移概率矩阵：

```

for (int i = 0; i < wordAndPOS.length - 1; i++) {
    int j = i + 1;
    String[] a = wordAndPOS[i].split("/");
    String[] b = wordAndPOS[j].split("/");
    int pre = getPOSId(a[1]);//前一个词性 ID
    int next = getPOSId(b[1]);//后一个词性 ID
    addWordPOSToMatrix(pre,next);//转移矩阵数组增加一个计数
}

```

统计某个词的发射概率：

```

public String getFireProbability(CountPOS countPOS){
    StringBuilder ret = new StringBuilder();

```

```

for(Entry<String,Integer> m: posFreqMap.entrySet()){
    //某词的这个词性的发射概率是 某词出现这个词性的频率 / 这个词性的总频率
    double prob =
        (double)m.getValue()/
        (double)(countPOS.getFreq(CorpusToDic.getPOSId(m.getKey())));
    ret.append(m.getKey() + ":" + prob + " ");
}
return ret.toString();
}

```

测试一个词的发射概率和相关的转移概率：

```

String testWord = "成果";
System.out.println(testWord+" 的词频率: \n"+this.getWord(testWord));
System.out.println("词性的总频率: \n"+posSumCount);
System.out.println      (testWord+"      发      射      概      率      :
\n"+this.getWord(testWord).getFireProbability(posSumCount));
System.out.println("转移频率计数取值测试: \n "+this.getTransMatrix("n","w"));
printTransMatrix();

```

例如，“成果” 这个的词的频率：

nr:5 b:1 n:287

词性的总频率：

a:34578 ad:5893 ag:315 an:2827 b:8734 c:25438 d:47426 dg:125 e:25 f:17279 g:0 h:48 i:4767
j:9309 k:904 l:6111 m:60807 n:229296 ng:4483 nr:35258 ns:27590 nt:3384 nx:415 nz:3715 o:72
p:39907 q:24229 r:32336 s:3850 t:20675 tg:480 u:74751 ud:0 ug:0 uj:0 ul:0 uv:0 uz:0 v:184775
vd:494 vg:1843 vn:42566 w:173056 x:0 y:1900 z:1338

“成果”的发射概率：

nr: 0.00014181178739576834 b: 0.0001144950767117014 n:0.0012516572465285046

即“成果”这个词的作为名词的发射概率

=“成果”作为名词出现的次数/名词的总次数

= 287/229296

= 0.0012516572465285046

为了支持词性标注，需要在词典中存储词性和对应的次数。格式如下：

滤波器 n 0
 堵击 v 0
 稿费 n 7
 神机妙算 i 0
 开设 vn 0 v 32

每行一个词，然后是这个词可能的词性和语料库中按这个词性出现的次数。存储基本词性相关信息的类如下：

```
public class POSInf {
    public short pos=0;    //词性，或者词的类别
    public int freq=0;    //词频，就是一个词在语料库中出现的次数。词频高就表示这个词是常用词。
    public String seCode = null; //词的语义编码
}
```

同一个词可以有不同的词性，可以把这些和某个词的词性相关的信息放在同一个链表中。

4.11.2 基于转换的错误学习方法

想象在画一幅油画。一幅由蓝天、草地、白云组成的油画。先把整块画布涂成蓝色，然后把下面的三分之一用绿色覆盖，表示草地，然后把上面的蓝色用白色覆盖，表示白云。这样通过从大到小的修正，越来越接近最终的精细化结果。

基于转换的学习方法(Transformation Based Learning 简称 TBL)先把每个词标注上最可能的词性，然后通过转换规则修正错误的标注，提高标注精度。

一个转换规则的例子：如果一个词左边第一个词的词性是量词(q)，左边第二个词的词性是数词(m)，则把这个词的词性从动词(v)改为名词(n)。

他/r 做/v 了/u 一/m 个/q 报告/v

转换成：

他/r 做/v 了/u 一/m 个/q 报告/n

另一个转换规则的例子：如果一个词左边第一个词的词性是介词(p)，则把这个词的词性从动词(v)改为名词(n)。

从形式上来看，转换规则由激活环境和改写规则两部分组成。例如，对于刚才的例子：

- 改写规则是：将一个词的词性从动词(v)改为名词(n)；

- 激活环境是：该词左边第一个紧邻词的词性是量词(q)，第二个词的词性是数词(m)。

可以从训练语料库中学习出转换规则。学习转换规则序列的过程如下：

1. 初始状态标注：用从训练语料库中统计的最有可能的词性标注语料库中的每个词。
2. 考察每个可能的转换规则：选择能最多的消除语料库标注错误数的规则，把这个规则加到规则序列最后。
3. 用选择出来的这个规则重新标注语料库。
4. 返回到 2，直到标注错误没有明显的减少为止。

这样得到一个转换规则集序列，以及每个词最有可能的词性标注。

标注新数据分两步：首先，用最有可能的词性标注每个词。然后依次应用每个可能的转换规则到新数据。

使用基于转换的学习方法标注词性的实现代码如下：

```
public List<String> tag(List<String> words) {
    List<String> ret = new ArrayList<String>(words.size());
    for (int i = 0, size = words.size(); i < size; i++) {
        String[] ss = (String[]) lexicon.get(words.get(i));
        if (ss == null)
            ss = lexicon.get(words.get(i).toLowerCase());
        if (ss == null && words.get(i).length() == 1)
            ret.add(words.get(i) + "^");
        if (ss == null)
            ret.add("NN");
        else
            ret.add(ss[0]); // 先给每个词标注上最有可能的词性
    }
    // 然后依次应用每个可能的转换规则
    for (int i = 0; i < words.size(); i++) {
        String word = ret.get(i);
        // rule 1: DT, {VBD | VBP} --> DT, NN
        if (i > 0 && ret.get(i - 1).equals("DT")) {
            if (word.equals("VBD")
                || word.equals("VBP")
                || word.equals("VB")) {
```

```

        ret.set(i, "NN");
    }
}
return ret;
}

```

4.12 平滑算法

语料是有限的，不可能覆盖所有的词汇。比如说 N 元模型，当 N 较大的时候，由于样本数量有限，导致很多的先验概率值都是 0，这就是零概率问题。当 n 值是 1 的时候，也存在零概率问题，例如一些词在词表中，但是却没有出现在语料库中。这说明语料库太小了，没能包括一些本来可能出现的词的句子。

做过物理实验的都知道，我们一般测量了几个点后，就可以画出一条大致的曲线，这叫做回归分析。利用这条曲线，可以修正测量的一些误差，并且还可以估计一些没有测量过的值。平滑算法用观测到的事件来估计的未观察到事件的概率。例如从那些比较高的概率值中匀一些给那些低的或者是 0。为了更合理的分配概率，可以根据整个直方图分布曲线去猜那些为 0 的实际值应该是多少。

由于训练模型的语料库规模有限且类型不同，许多合理的搭配关系在语料库中不一定出现，因此会造成模型出现数据稀疏现象。数据稀疏在统计自然语言处理中的一个表现就是零概率问题。有各种平滑算法来解决零概率的问题。例如，我们对自己能做到的事情比较了解，而不太了解别人是否能做到一些事情，这样导致高估自己而低估别人。所以需要开发一个模型减少已经看到的事件的概率，而允许没有看到的事件发生。

平滑有黑盒方法和白盒方法两种。黑盒平滑方法把一个项目作为不可分割的整体。而白盒平滑方法把一个项目作为可分拆的，可用于 n 元模型。

加法平滑算法是最简单的一种平滑。加法平滑的原理是给每个项目增加 λ ($1 \geq \lambda \geq 0$)，然后再除以总数作为项目新的概率。因为数学家拉普拉斯(laplace)首先提出用加 1 的方法估计没有出现过的现象的概率，所以加法平滑也叫做拉普拉斯平滑。

下面是加法平滑算法的一个实现，注释中以词为例来理解代码：

```

//根据原始的计数器生成平滑后的分布
public static <E> Distribution<E> laplaceSmoothedDistribution(
    GenericCounter<E> counter,
    int numberOfKeys,
    double lambda) {
    Distribution<E> norm = new Distribution<E>(); //生成一个新的分布
    norm.counter = new Counter<E>();
}

```

```

double total = counter.totalDoubleCount();//原始的出现次数
double newTotal = total + (lambda * (double) numberOfKeys);//新的出现次数
//有多大可能性出现零概率事件
double reservedMass =
    ((double) numberOfKeys - counter.size()) * lambda / newTotal;
norm.numberOfKeys = numberOfKeys;
norm.reservedMass = reservedMass;
for (E key : counter.keySet()) {
    double count = counter.getCount(key);
    //对任何一个词来说，新的出现次数是原始出现次数加 lambda
    norm.counter.setCount(key, (count + lambda) / newTotal);
}
if (verbose) {
    System.err.println("unseenKeys=" + (norm.numberOfKeys - norm.counter.size()) +
        " seenKeys=" + norm.counter.size() + " reservedMass=" + norm.reservedMass);
    System.err.println("0 count prob: " + lambda / newTotal);
    System.err.println("1 count prob: " + (1.0 + lambda) / newTotal);
    System.err.println("2 count prob: " + (2.0 + lambda) / newTotal);
    System.err.println("3 count prob: " + (3.0 + lambda) / newTotal);
}
return norm;
}

```

需要注意的是 reservedMass 是所有零概率词的出现概率的总和，而不是其中某个词出现概率的总和。取得指定 key 的概率实现代码可以看到：

```

public double probabilityOf(E key) {
    if (counter.containsKey(key)) {
        return counter.getCount(key);
    } else {
        int remainingKeys = numberOfKeys - counter.size();
        if (remainingKeys <= 0) {
            return 0.0;
        } else {
            //如果有零概率的词，
            //则这个词的概率是 reservedMass 分摊到每个零概率词的概率
            return (reservedMass / remainingKeys);
        }
    }
}

```



```
}
```

这种方法中的 λ 值不好选取，在接下来介绍的另外一种平滑算法 Good-Turing 方法中则不需要 λ 值。为了说明 Good-Turing 方法，首先定义一些标记：

假设词典中共有 x 个词。在语料库中出现 r 次的词有 N_r 个。例如，出现 1 次的词有 N_1 个。

则语料库中的总词数 $N = 0 \cdot N_0 + 1 \cdot N_1 + r \cdot N_r + \dots$

而 $x = N_0 + N_1 + N_r + \dots$

使用观察到的类别 $r+1$ 的全部概率去估计类别 r 的全部概率。计算中的第一步是估计语料库中没有见过的词的总概率 $p_0 = N_1 / N$ ，分摊到每个词的概率是 $N_1 / (N \cdot N_0)$ 。第二步估计语料库中出现过一次的词的总概率 $p_1 = N_2 \cdot 2 / N$ ，分摊到每个词的概率是 $N_2 \cdot 2 / (N \cdot N_1)$ 。依次类推，当 r 值比较大时， N_r 可能是 0，这时候不再平滑。

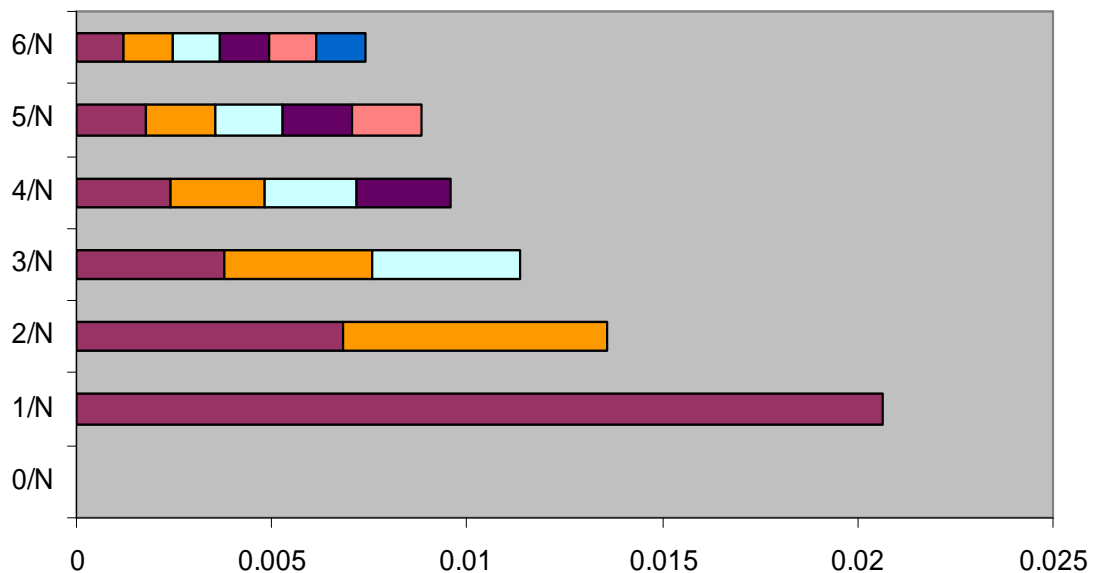


图 4-10 词的概率图

Good-Turing 平滑实现代码如下：

```
public static <E> Distribution<E> goodTuringSmoothedCounter(
    GenericCounter<E> counter,
    int numberOfKeys) {

    //收集计数数组，也就是直方图
    int[] countCounts = getCountCounts(counter);

    //如果计数数组不可靠，就不要用 Good-Turing 方法
```

```
//而采用拉普拉斯平滑方法
for (int i = 1; i <= 10; i++) {
    if (countCounts[i] < 3) {
        return laplaceSmoothedDistribution(counter, numberOfKeys, 0.5);
    }
}

double observedMass = counter.totalDoubleCount();
double reservedMass = countCounts[1] / observedMass;

//计算和缓存调整后的频率，同时也调整观察到的项目的总数
double[] adjustedFreq = new double[10];
for (int freq = 1; freq < 10; freq++) {
    adjustedFreq[freq] = (double) (freq + 1) * (double) countCounts[freq + 1] /
        (double) countCounts[freq];
    observedMass -= ((double) freq - adjustedFreq[freq]) * countCounts[freq];
}

double normFactor = (1.0 - reservedMass) / observedMass;

Distribution<E> norm = new Distribution<E>();
norm.counter = new Counter<E>();

//填充新的分布，同时重新归一化
for (E key : counter.keySet()) {
    int origFreq = (int) Math.round(counter.getCount(key));
    if (origFreq < 10) {
        norm.counter.setCount(key, adjustedFreq[origFreq] * normFactor);
    } else {
        norm.counter.setCount(key, (double) origFreq * normFactor);
    }
}

norm.numberOfKeys = numberOfKeys;
norm.reservedMass = reservedMass;
return norm;
}
```

对条件概率的 N 元估计平滑：

$$P_{GT}(w_i | w_1, \dots, w_{i-1}) = \frac{c^*(w_i, w_{i-1})}{c^*(w_1, \dots, w_{i-1})}$$

这里的 c^* 来源于 GT 估计。

例如：估计三元条件概率：

$$P_{GT}(w_3 | w_1, w_2) = \frac{c^*(w_1, w_2, w_3)}{c^*(w_1, w_2)}$$

对于一个没见过的三元联合概率是：

$$P_{GT}(w_1, w_2, w_3) = \frac{c_0^*}{N} = \frac{N_1}{N_0 * N}$$

对于一元和二元模型也是如此。

4.13 机器学习的方法

可以把机器学习的方法分成产生式和判别式两种。假定输入是 x ，类别标签是 y 。产生式模型估计联合概率 $P(x, y)$ ，因为可以根据联合概率来生成样本，所以叫做产生式模型。判别式模型估计条件概率 $P(y|x)$ ，因为没有 x 的知识，无法生成样本，只能判断分类，所以叫做判别式模型。

产生式模型可以根据贝叶斯公式得到判别式模型，但反过来不行。例如下面的情况：

(1,0), (1,1), (2,0), (2, 1)

联合概率 $p(x, y)$ 如下：

$P(1, 0) = 1/2, P(1, 1) = 0, P(2, 0) = 1/4, P(2, 1) = 1/4$

计算出条件概率 $P(y|x)$ ：

$P(0|1) = 1, P(1|1) = 0, P(0|2) = 1/2, P(1|2) = 1/2$

4.13.1 最大熵

熵(information entropy)是信息论的核心概念，它描述一个随机系统的不确定度。设离散

随机变量 X 取值 A_k 的概率为 P_k ，这里 $k=1, 2, \dots, n$ ， $P_k > 0$ ， $\sum_{k=1}^n P_k = 1$ 。则熵定义为：

$$H(X) = -\sum_{k=1}^n P_k \log(P_k)$$

最大熵原理(Maximum Entropy)描述在一定条件下，随机变量满足何种分布时熵取得最大值。例如，在离散随机变量情况下，当概率分布为平均分布，即 $P_k = \frac{1}{N}$ 时，熵取最大熵。

举个例子，一个快餐店提供 3 种食品：汉堡(B)、鸡肉(C)、鱼(F)。价格分别是 1 元、2 元、3 元。已知人们在这家店购买一种食品的平均消费是 1.75 元，求顾客购买这 3 种食品的概率。符合条件的概率有很多，但是一个稳定的系统往往都趋向于使得熵最大。如果假设买这三种食品的概率相同，那么根据熵公式，这不确定性就是 1(熵等于 1)。但是这个假设很不合适，因为这样算出来的平均消费是 2 元。我们已知的信息是：

$$P(B)+P(C)+P(F) = 1$$

$$1*P(B)+2*P(C)+3*P(F) = 1.75$$

以及关于对概率分布的不确定性度量，熵：

$$S = -P(B)\log(P(B)) - P(C)\log(P(C)) - P(F)\log(P(F))$$

对前两个约束，两个未知概率可以由第三个量来表示，可以得到：

$$P(C) = 0.75 - 2*P(F)$$

$$P(B) = 0.25 + P(F)$$

把上式代入熵的表达式中，熵就可以用单个概率 $P(F)$ 来表示：

$$S = -(0.25 + P(F))\log(0.25 + P(F)) - (0.75 - 2*P(F))\log(0.75 - 2*P(F)) - P(F)\log(P(F))$$

对这个单变量优化问题，很容易求出 $P(F)=0.216$ 时熵最大， $S=1.517$ ， $P(B)=0.466$ ， $P(C)=0.318$ 。

宾夕法尼亚州立大学的 Ratnaparkhi 将上下文信息、词性(名词、动词和形容词等)、句子

成分通过最大熵模型结合起来，做出了当时世界上最准确的词性标注系统和句法分析器。

opennlp.maxent(<http://maxent.sourceforge.net/>)是一个最大熵分类器的实现。

而<http://opennlp.sourceforge.net>则利用 maxent 实现了 Ratnaparkhi 提出的句法解析器算法。

maxent 训练多类分类问题。Maxent 的输出和 SVM 分类方法不同，SVM 训练的是 2 类分类问题。如果训练的是多类问题，可以考虑用最大熵。

例如要做一个从英文句子查找人名的分类器。假设已经有了一个训练好的模型。从下面这个句子判断 Terrence 是否一个人名。

对于下面这句话：

He succeeds Terrence D. Daniels, formerly a W.R. Grace vice chairman, who resigned.

首先要从句子中提取针对当前词“Terrence”的特征，并且转换成 context：

previous=succeeds current=Terrence next=D. currentWord IsCapitalized

送一个带有所有特征的字符串数组给模型，然后调用 eval 方法。

```
public double[] eval(String[] context);
```

返回的 double[] 包含了对每个类别的隶属概率。这些隶属概率是根据 context 输入的各种特征计算出来的。double[] 的索引位置和每个类别对应。例如，索引位置 0 是代表“TRUE”的概率，而索引位置 1 是代表“FALSE”的概率。可以调用下面的方法查询某个索引类别的字符串名字：

```
public String getOutcome(int i);
```

可以通过下面的方法取得 eval 返回的隶属概率数组中的最大概率所对应的类名：

```
public String getBestOutcome(double[] outcomes);
```

简单的根据特征分类的例子：

```
RealValueFileEventStream rvfes1 = new RealValueFileEventStream("D:/  
/opennlp/maxent/real-valued-weights-training-data.txt");
```

```
GISModel realModel = GIS.trainModel(100,new OnePassRealValueDataIndexer(rvfes1,1));
```

```
String[] features2Classify = new String[] { "feature2", "feature5" };

//输入特征数组返回分类结果，还可以对每个特征附加特征权重

double[] realResults = realModel.eval(features2Classify);

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

    System.out.println(String.format("classify with realModel: %1$s = %2$f",
realModel.getOutcome(i), realResults[i]));

}
```

使用最大熵模型对英文切分和标注的例子：

```
public class Main{
    private static final String TOKENS = "path to EnglishTok.bin.gz";
    private static final String DICT = "path to dict.bin.gz";
    private static final String TAGDICT = "path to tag.bin.gz";

    private static final String TEXT = "This is a testing sentence";

    public static void main(String[] args) {
        try
        {
            TokenizerME tokenizer = new TokenizerME((new
SuffixSensitiveGISModelReader(new File(TOKENS))).getModel());
            tokenizer.setAlphaNumericOptimization(true);
            String[] tokens = tokenizer.tokenize(TEXT);

            POSTaggerME postagger = new POSTaggerME(getModel(TAGDICT), new
DefaultPOSContextGenerator(new Dictionary(DICT)));
            String[] tags = postagger.tag(tokens);

            if(tags!=null)
                for(int i=0;i<tags.length;i++)
                    System.out.println("tag "+i+" = "+tags[i]);

        } catch (IOException ex)
```

```
{
    Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
}
}

private static MaxentModel getModel(String name) {
    return new SuffixSensitiveGISModelReader(new File(name)).getModel();
}
}
```

4.13.2 条件随机场

条件随机场(CRF)是一种无向图模型，它是在给定需要标记的观察序列的条件下计算整个标记序列的联合概率分布，而不是在给定当前状态条件下定义下一个状态的状态分布。即给定观察序列 O ，求最佳序列 S 。

条件随机场的优点有：CRF 没有 HMM 那样严格的独立性假设条件，因而可以容纳任意的上下文信息。可以灵活的设计特征。同时，由于 CRF 计算全局最优输出节点的条件概率，它还克服了最大熵马尔可夫模型标记偏置(Label-bias)的缺点。

缺点是：训练的时间比较长。

可以用条件随机场来做词性标注。有几个条件随机场的开源包，例如：CRF(<http://crf.sourceforge.net/>)是一个专门的软件包，而 Mallet(<http://mallet.cs.umass.edu/>)和 MinorThird(<http://minorthird.sourceforge.net/>)则包含了相关的实现。

4.14 有限状态机

在中文分词中，有些词不是依据词表切分，而是用规则识别出来，例如：

- 数字：123,456.781 90.7% 3/8 11/20/2000
- 日期：1 9 9 8 年 2009 年 12 月 24 日 10:30
- 电话号码：010-56837834

有些电话号码有固定的格式，一些国家的电话号码格式是：

- 美国(USA)的电话号码是由十位数字所组成，前三码是区域码，中间三位数字是交换码，再加上后四码。例如：(202) 522-2230，1-925-225-3000，212.995.5402。

- 英国(UK)的电话号码格式为 0044-[城/郡码]-[本机号码] 城郡码与本机码长短不一，城郡码有 2-4 位。例如：[0171 378 0647]，[4.171) 830 1007]，[+44(0) 1225 753678]，[01256 468551]。
- 丹麦(Denmark)的电话号码格式：[+45 43 48 60 60]。
- 巴基斯坦(Pakistan)的电话号码格式：[95-51-279648]。
- 瑞士(Switzerland)的电话号码格式：[+441/284 3797]。
- 斯里兰卡(Sri Lanka)的电话号码格式：[(94-1)866854]。
- 德国(Germany)的电话号码格式：[+49 69 136-2 98 05]。
- 法国(France)的电话号码格式：[33 1 34 43 32 26]。
- 荷兰(Netherlands)的电话号码格式：[++31-20-5200161]。
- 俄罗斯(Russian)的电话号码格式：[(342) 218-18-01]。
- 韩国(Korea)的电话号码格式：[+82-2-2185-3800]。

比如说：“阿拉伯队”。地名+“队”转换成一个机构名词可以用如下规则表示：

机构名词 => Ns + 队

对应的有限状态转移

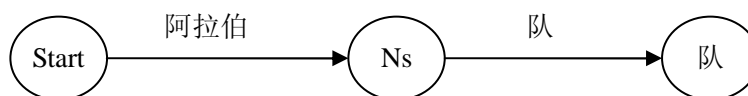


图 4-11 有限状态图

有限状态机(Finite State Machine 简称 FSM，有时也叫做有限自动机)是由状态(State)、变换(Transition)和行动(Action)组成的行为模型。有限状态机首先在初始状态(Start State)，接收输入事件(Input Event)后转移到下一个状态。有限状态机通过条件(Guard)判断是否符合转移条件。可以把一个有限状态机看成一个特殊的有向图。常用的正则表达式就是用有限状态机实现的。

可以用状态转移表展示有限状态自动机基于当前状态和输入要移动到什么状态。

一维状态表输入通常放置在左侧，输出在右侧。输出将表示状态机的下一个状态。下面是有两个状态和两个组合输入的状态机的简单例子：

有下面几种类型的动作：

- 进入动作(Entry action)：在进入状态时进行；
- 退出动作(Exit action)：在退出状态时进行；
- 输入动作(Input action)：依赖于当前状态和输入条件进行；
- 转移动作(Transition action)：在进行特定转移时进行。

例如实现接受“****年**月**日”格式的有限状态机。其中双圈的状态是结束状态。

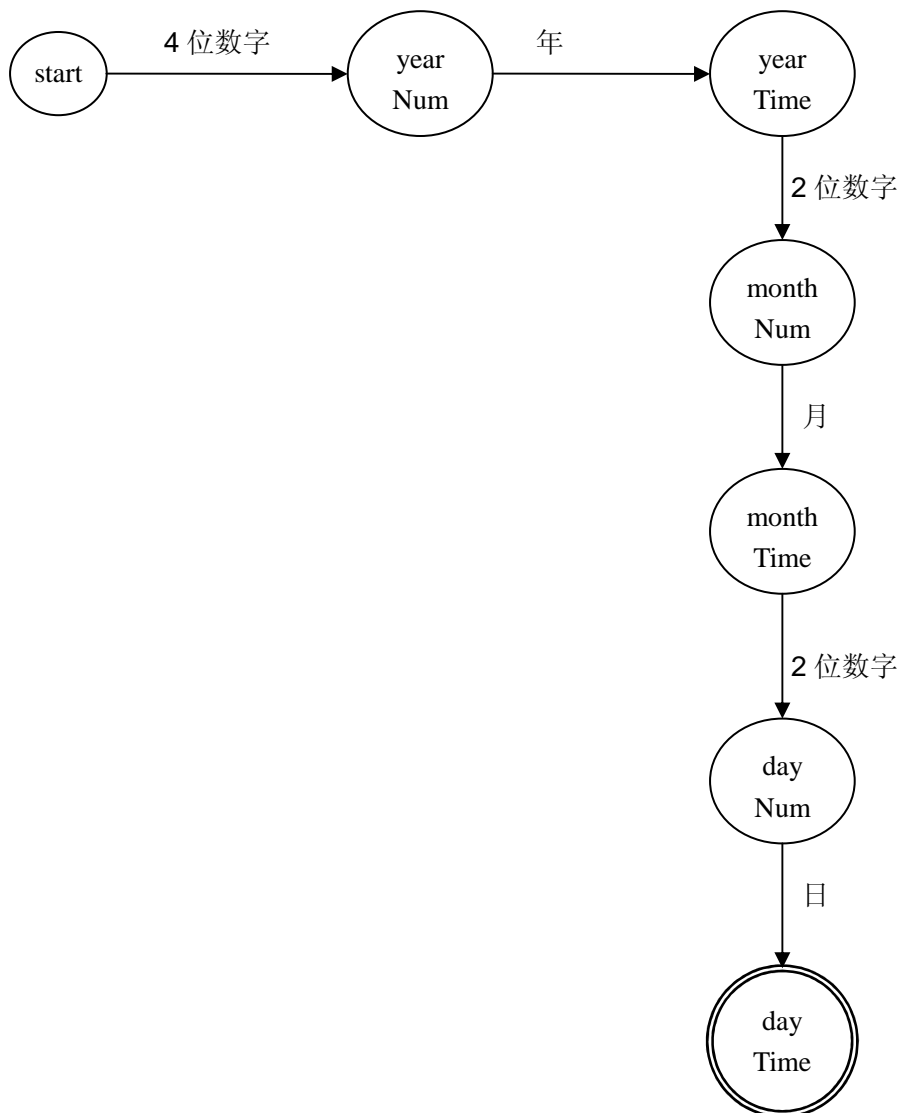


图 4-12 接收时间的有限状态图

每个事件有个名字，同时有些静态的预定义好的事件。事件类实现代码如下：

```
public class InputEvent {
    public static InputEvent digital4 = new InputEvent("Y");//四位数字对应的事件
    public static InputEvent digital2 = new InputEvent("D2");//二位数字对应的事件
    public static InputEvent digital1 = new InputEvent("D1");//一位数字对应的事件
    public static InputEvent yearUnitEvent = new InputEvent("年");//年对应的事件
    public static InputEvent monthUnitEvent = new InputEvent("月");//月对应的事件
    public static InputEvent dayUnitEvent = new InputEvent("日");//日对应的事件
    public static InputEvent splitEvent = new InputEvent("-");//分隔符对应的事件
    public String word; //事件名

    public InputEvent(String w) {
        this.word = w;
    }

    public boolean equals(Object obj) {
        if (obj == null || !(obj instanceof InputEvent)) {
            return false;
        }
        InputEvent other = (InputEvent) obj;

        if(this.word!=null) {
            if(!this.word.equals(other.word)) {
                return false;
            }
            return true;
        }

        return true;
    }
}
```

进入状态的条件类由所对应的事件定义，实现如下：

```
public class Guard {
    private InputEvent event; //对应的事件
```

```
private String label = "";
}
```

状态类有非结束状态和结束状态之分，State 类实现代码如下：

```
public class State {
    private static int nextId = 0;
    public int id = nextId++; //状态编号
    private String label = ""; //状态名称
    private Collection<Transition> transitions =
        new LinkedList<Transition>();//转换到其他状态的数组
    private boolean finalState = false;
}
```

转换类定义了了什么条件下可以从哪些状态可以转换到下个状态，实现如下：

```
public class Transition {
    private State nextState; //下个状态
    private State state; //当前状态
    private Guard guard; //从当前状态转换到下个状态的条件
}
```

在有限状态机中定义有效的状态转换：

```
State startState = createState(); //创建开始状态
startState.setLabel("start");

setStartState(startState);
reset(); //重置当前状态
//定义有限状态机
Guard numGuard = createGuard();//创建条件
numGuard.setEvent(InputEvent.yearNum);

State yearNumState = createState();
yearNumState.setLabel("yearNum");
createTransition(startState, yearNumState, numGuard); //创建状态之间的转移

State yearTimeState = createState();
yearTimeState.setLabel("yearTime");

Guard yearGuard = createGuard();
```

```

yearGuard.setEvent(InputEvent.yearUnitEvent);
createTransition(yearNumState, yearTimeState, yearGuard);

Guard num2Guard = createGuard();
num2Guard.setEvent(InputEvent.digital2);

State monthNumState = createState();
monthNumState.setLabel("monthNum");
createTransition(yearTimeState, monthNumState, num2Guard);

State monthTimeState = createState();
monthTimeState.setLabel("monthTime");

Guard monthGuard = createGuard();
monthGuard.setEvent(InputEvent.monthUnitEvent);

createTransition(monthNumState, monthTimeState, monthGuard);

Guard dayNumGuard = createGuard();
dayNumGuard.setEvent(InputEvent.digital2);

State dayNumState = createState();
dayNumState.setLabel("dayNum");
createTransition(monthTimeState, dayNumState, num2Guard);

State dayTimeState = createState();
dayTimeState.setLabel("dayTime");
dayTimeState.setFinal(true);

Guard dayGuard = createGuard();
dayGuard.setEvent(InputEvent.dayUnitEvent);

createTransition(dayNumState, dayTimeState, dayGuard);

```

执行有限状态机：

```

//从指定位置开始匹配数字
public static int matchDigital(int start, String sentence) {

```

```
int i = start;
int count = sentence.length();
while (i < count) {
    char c = sentence.charAt(i);
    if (c >= '0' && c <= '9' || c >= '0' && c <= '9') {
        ++i;
    } else {
        break;
    }
}

return i;
}

public static boolean isDate(String s) { //通过有限状态机判断输入的字符串是否有效日期
    FSM fsm = FSM.getInstance();
    //把输入串转换成有限状态机可以接收的事件
    for(int i=0; i<s.length(); ) {
        int ret = matchDigital(i,s);
        int diff = ret - i;

        InputEvent event = null;
        if(diff==4) {
            event = InputEvent.yearNum;
            i=ret;
        }
        else if(diff==2) {
            event = InputEvent.digital2;
            i=ret;
        }
        else if(s.charAt(i)=='年') {
            event = InputEvent.yearUnitEvent;
            ++i;
        }
        else if(s.charAt(i)=='月') {
            event = InputEvent.monthUnitEvent;
            ++i;
        }
    }
}
```

```

    }
    else if(s.charAt(i)=='日') {
        event = InputEvent.dayUnitEvent;
        ++i;
    }
    else if(s.charAt(i)=='-') {
        event = InputEvent.splitEvent;
        ++i;
    }
    if(event == null) {
        return false;
    }

    //接受事件
    MatchType matchRet = fsm.consumeEvent(event);
    if(matchRet == MatchType.Match) {
        return true;
    }
    else if(matchRet == MatchType.MisMatch) {
        return false;
    }
}

return false;
}

//测试方法
public static void main(String[] args) throws Exception {
    System.out.println(isDate("2009 年 05 月 08 日"));
}

```

在定义有限状态机时，如果要接收不同格式的数据，而这些数据有共同的前缀，则这些前缀从开始状态开始共用状态。

写一个识别电话号码的有限状态机的流程：

1. 在 `InputEvent` 类中定义事件；
2. 写一个 `toEvent(String s)` 方法把输入字符串转换成定义好的事件；

3. 在 FSM 类中定义状态和状态之间的转换条件;
4. 在 isPhone 方法中判断输入电话号码是否合法。

为了提高有限状态机的匹配速度,可以用幂集构造的方法把非确定有限状态机(NFA)转换成确定有限状态机(DFA),并可以最小化 DFA,也就是使状态数量最少。

4.15 本章小结

本章介绍了分词中的查找词典算法。查词典最早用首字母散列或者散列表实现,然后采用 Trie 树的方法开始流行,还有采用数组形式的双数组。后来又发展出和 AC 算法结合的 Trie 树。

因为正向最长匹配最容易实现,所以很多开源的中文分词采用此方法实现。最大概率中文分词是以词为单位的分词。之后出现了按字标注(Character-based Tagging)的 CRF 分词方法。

使用机器学习的方法来标注词性,可以分为两个过程:从训练语料库学习的过程和给一个词序列标注词性的过程。隐马尔可夫模型学习出来的结果是一个语言模型,而基于转换的方法学习出来的结果是一个转换规则集序列,以及每个词最有可能的词性标注。

早期词性标注的流行方法是采用隐马尔可夫模型实现。隐马尔可夫模型是一种产生式模型,而产生式模型根据训练集学习出的准确度不如判别式模型高。因此最大熵和 CRF 等判别式方法应用于词性标注可以取得更高的准确度。

最大熵是机器学习领域很重要的概念。根据物理学的研究,一个封闭系统内的温度趋向于平均,也就是熵趋向于最大。所以最大熵是一个应用广泛的概念。对最大熵另外一种理解是:在已知条件下,不要对未知事情的发生概率有偏见。

有限状态机在自然语言处理领域的应用逐渐发展成一个专门的学问。可以把有限状态机改成根据一个输入产生一个输出,扩展成为有限状态转换。也可以把有限状态机返回的结果改成数字,扩展成为加权有限状态机。可以把有限状态转换扩展成为加权有限状态转换。有限状态机之间可以做各种运算。例如合并(Union)、级联(Concatenation)、组合(Composition)、闭包(Closure)等运算。

初学者容易在有限状态机中有限的状态里转晕,所以有人开发出了调试有限状态机的工具软件。施乐公司用有限状态机检查日期的有效性,并开发了有限状态转换工具包 XFST(<http://www.xrce.xerox.com/>)。Google 开发出了有限状态转换的开源软件 OpenFst(<http://www.openfst.org/>),并应用于机器翻译。有限状态机还可以用于英文的词性标注。路由器里面的路由表查找算法也可以考虑采用有限状态机。

第5章 让搜索引擎理解自然语言

当用户输入更长的问题时，要能够给出更好的答案。在几乎是所有的我们从电影或电视上看到的未来，搜索引擎已经进化到类似人类助手一样能回答针对任何事物的复杂问题程度。尽管互联网搜索引擎能够导航非常巨大的知识范围，但是我们要达到智能助手的能力还很远。一个很明显的不同就是互联网搜索的查询被划分成少数几个关键词而不是以自然语言表述的实际的问题。人们可以用基于社区的回答系统通过句子甚至是段落来描述他们的信息需求，因为他们知道如果问题描述的更好，其它人会通读上下文获得更为明确的理解而给出更好的回答。相比而言，一个互联网搜索引擎对于一段很长的查询只能给出很差的返回结果或不返回任何有价值的东西。人们只能将他们的问题转化成一个或几个贴切的关键词去尝试相对贴切的回复。信息检索研究的长期目标是开发检索模型来为更长、更专门的查询提供精确的结果。因此需要更好的理解文本信息。自然语言处理(Natural Language Process)，严格来说自然语言处理包括自然语言理解和自然语言生成两部分。但这里我们只涉及自然语言理解部分。

如何预测用户查询意图？对于长的查询，可以根据特征词。例如：“感冒了可以吃海鲜吗”中的【吗】字是一个很强的疑问特征词，而“Nokia N97 港行”中【Nokia】、【N97】、【港行】都是很好的产品购买特征词，“松下 328 传真机参数”中的【参数】则是一个很好的产品详细型查询的特征词。

5.1 停用词表

在基于词的检索系统中，停用词是指出现频率太高、没有太大检索意义的词，如中文中的“的、了、吧”等，还有英文中的“of、the”等。

可以把这些词放到 stopwords.txt 文本文件中，每行一个词。例如：

```
的
了
吧
```

创建一个 StopSet 类负责查找一个词是否停用词。

```
public class StopSet extends HashSet<String>{
    private static final long serialVersionUID = -5739693689170303932L;
    private static StopSet stopSet = new StopSet();

    /**
     *
     * @return the singleton of dictionary
     */
}
```



```
public static StopSet getInstance(){
    return stopSet;
}

//取得词典文件所在的路径
public static String getDir() {
    String dir = System.getProperty("dic.dir");
    if (dir == null)
        dir = "/dic/";
    else if( !dir.endsWith("/"))
        dir += "/";
    return dir;
}

//加载停用词词典
private StopSet() {
    super(1000);
    String line;
    try{
        String dic = "/stopword.txt";
        InputStream file = null;
        if (System.getProperty("dic.dir") == null)
            file = getClass().getResourceAsStream(getDir()+dic);
        else
            file = new FileInputStream(new File(getDir()+dic));

        BufferedReader in=
            new BufferedReader(new InputStreamReader(file,"GBK"));

        while( true ) {
            line = in.readLine();
            if (line == null )
                break;

            if (!"".equals(line)) {
                this.add(line);
            }
        }
    }
}
```

```
    }  
    in.close();  
}catch (Exception e)  
{  
    e.printStackTrace(System.err);  
}  
}  
}
```

然后使用这个停用词表类：

```
Set stopSet=StopSet.getInstance();  
if( stopSet.contains(word) ){  
    //如果 word 在停用词表中  
}
```

5.2 句法分析树

句法分析树一般用在机器翻译中，但是搜索引擎也可以借助句法分析树更准确的理解文本，从而更准确的返回搜索结果。比如有用户输入：“肩宽的人适合穿什么衣服”。如果返回结果中包括“肩宽的人穿什么衣服好？”，或者“肩膀宽宽的女孩子穿什么衣服好看？”可能是用户想要的结果。

OpenNLP(<http://incubator.apache.org/opennlp/>)包含一个句法分析树的实现。输入句子“Boeing is located in Seattle.”通过该程序分析后，返回的句法树如图 5-1 所示：

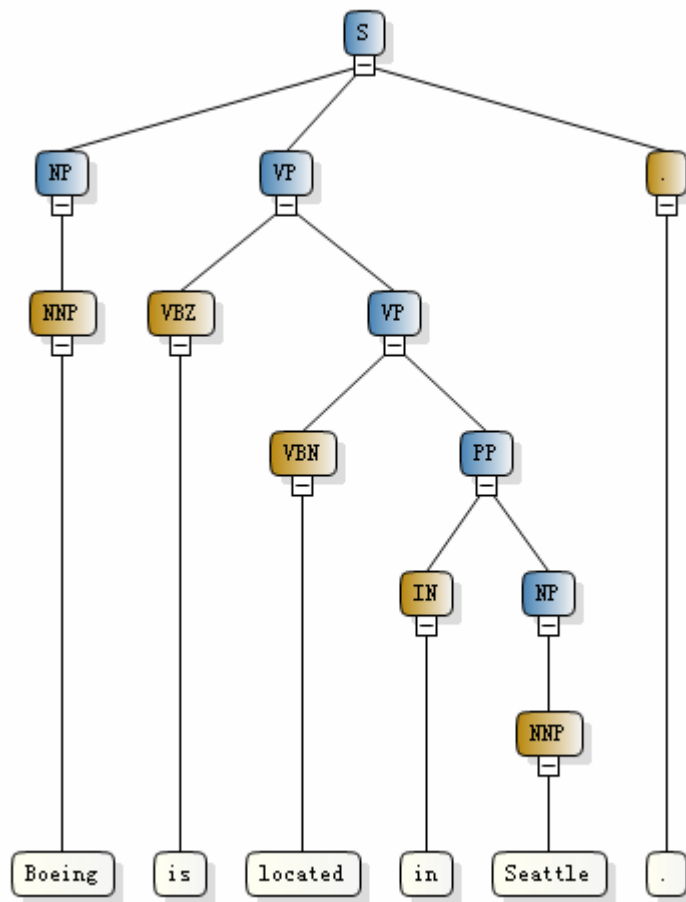


图 5-1 非词汇化的句法树

“咬/死/了/猎人/的/狗”这个经过中文分词切分后的句子有两个不同的理解。句法分析能够确定该句子的意义。也就是说句法分析树能消除歧义。

分析树的节点定义如下：

```
/** 保存解析构件的数据结构 */
public class Parse {
    /**这个解析基于的文本字符串，同一个句子的所有解析共享这个对象*/
    private String text;
    /** 这个构件在文本中代表的字符的偏移量 */
    private Span span;
    /**这个解析的句法类型*/
    private POSType type;
    /** 这个解析的孩子 */
    private Parse[] children;
```

}

在句法树的每个节点中还可以增加中心词(head)。中心词就是被修饰部分的词,比如“女教师”,中心词就是“教师”,而“女”就是定语了。

对中文来说,一般先分词然后形成树形结构。

句法分析树可以用自顶向下的方法或者自底向上的方法。chart parser 是自顶向下的分析器。Earley parser 是 chart parser 的一种。

OpenNLP 采用了移进一归约(shift-reduce parser)的方法实现分析器。移进一归约分析器是一种自底向上的分析器。移进一归约算法的基本数据结构是堆栈。检查输入词并且决定是把它移进堆栈或者规约堆栈顶部的元素,把产生式右边的符号用产生式左边的符号替换掉。

移进一归约算法的四种操作是:

- 移进(Shift): 从句子左端将一个终结符移到栈顶。
- 归约(Reduce): 根据规则,将栈顶的若干个符号替换成一个符号。
- 接受(Accept): 句子中所有词语都已移进到栈中,且栈中只剩下一个符号 S,分析成功,结束。
- 拒绝(Error): 句子中所有词语都已移进栈中,栈中并非只有一个符号 S,也无法进行任何归约操作,分析失败,结束。

例如,表 5-1 中的产生式列表:

编号	产生式
1	r -> 我
2	v -> 是
3	n -> 县长
4	np -> r
5	np -> n
6	vp -> v np
7	s -> np vp

表 5-1 产生式列表

其中第 1,2,3 条可以叫做词法规则(Lexical rule)，5,6,7 条叫做内部规则(Internal rule)。

移进—归约的方法分析词序列“我 是 县长”的过程如下：

栈	输入	操作	规则
	我 是 县长	移进	
我	是 县长	规约	(1) $r \rightarrow$ 我
$r(1)$	是 县长	规约	(4) $np \rightarrow r$
$np(4)$	是 县长	移进	
$np(4)$ 是	县长	规约	(2) $v \rightarrow$ 是
$np(4) v(2)$	县长	移进	
$np(4) v(2)$ 县长		规约	(3) $n \rightarrow$ 县长
$np(4) v(2) n(3)$		规约	(5) $np \rightarrow n$
$np(4) v(2) np(5)$		规约	(6) $vp \rightarrow v np$
$np(4) vp(6)$		规约	(7) $s \rightarrow np vp$
$s(7)$		接受	

表 5-2 分析词序列“我 是 县长”的过程

如果在每一步规约的过程中记录父亲指向孩子的引用，则可以生成一个完整的句法树，例如图 5-2 所示的句法树。

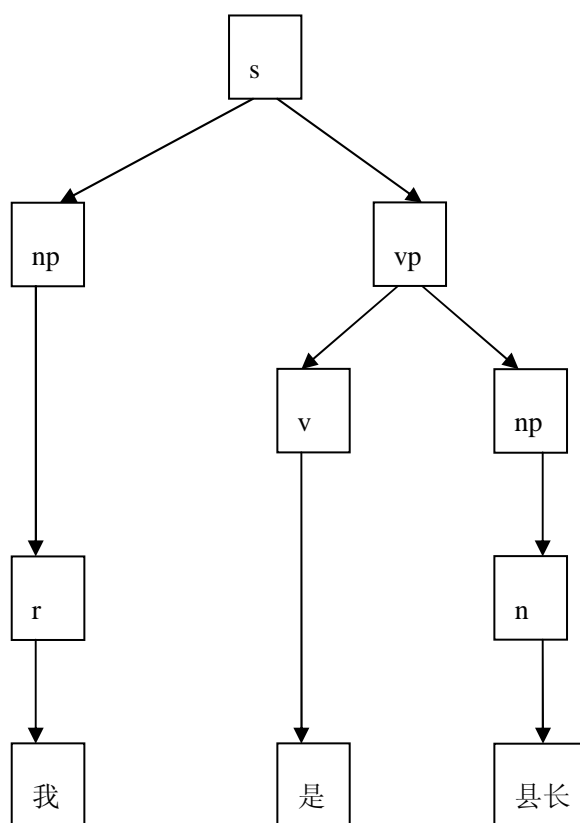


图 5-2 非词汇化的句法树

词汇化规则如表 5-3 所示。

表 5-3 词汇化规则列表

编号	产生式
4	np(我,r) -> r(我,r)
5	np(县长, n) -> n(县长, n)
6	vp(是, v)-> v(是, v) np(县长, n)
7	s(是, v)-> np(我,r) vp(是, v)

根据词汇化的规则可以生成如图 5-3 所示的词汇化句法树。

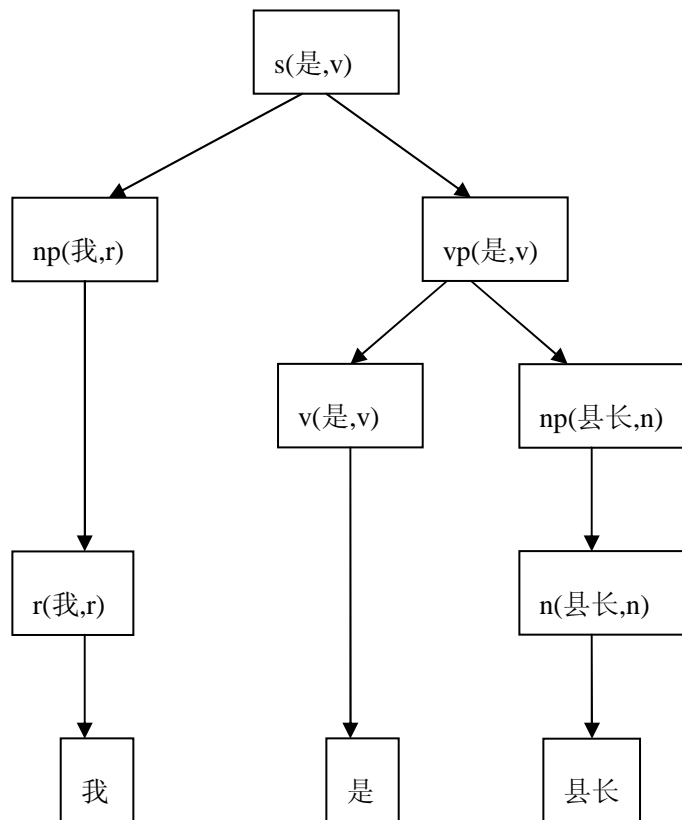


图 5-3 词汇化的句法树

产生式定义如下：

```

public class Production {
    protected TokenType lhs; //产生式左边的非终结符
    protected ArrayList<TokenType> rhs; //产生式右边的符号
}
  
```

递归方式实现的移进规约算法如下：

```

//输入待分析的字符串，判断是否可以接受这个字符串
private boolean recognise(NonTerminal cat, ArrayList string, Stack stack) {
    //可以接受吗？
    if (string.size() == 0 && stack.size() == 1 && stack.peek().equals(cat)) {
        return true;
    }
    else if (string.size() == 0 && stack.size() == 1 ) {
        return false;
    }
}
  
```

```
// 可以移进吗?
if (string.size() > 0) {
    Terminal sym = (Terminal) string.get(0);
    ArrayList prods = grammar.getTerminalProductions();
    for (int i = 0; i < prods.size(); i++) {
        // 是否是任何产生式的右侧的第一个符号?
        if (((Production) prods.get(i)).getRHS().get(0).equals(sym)) {
            // 是的, 就移进并检查剩下的
            Symbol lhs = ((Production) prods.get(i)).getLHS();
            ArrayList restString = new ArrayList(string);
            restString.remove(0);
            Stack shiftedStack = (Stack) stack.clone();
            shiftedStack.push(lhs);
            if (recognise(cat, restString, shiftedStack)) {
                return true;
            }
        }
    }
}

// 可以规约吗?
if (! stack.empty()) {
    ArrayList prods = (ArrayList) grammar.getRhsIndex().get(stack.peek());
    if (prods == null) {
        return false;
    }
    // 堆栈中顶层的符号对应产生式的右边部分吗?
    for (int i = 0; i < prods.size(); i++) {
        Symbol lhs = ((Production) prods.get(i)).getLHS();
        ArrayList rhs = ((Production) prods.get(i)).getRHS();
        Stack reducedStack = (Stack) stack.clone();

        if (rhs.size() > stack.size()) {
            continue;
        }
        ArrayList topOfStack = new ArrayList();
```



```

        for (int j = 0; j < rhs.size(); j++) {
            topOfStack.add(reducedStack.pop());
        }
        topOfStack = reverse(topOfStack);
        if (rhs.equals(topOfStack)) {
            //是的，则通过弹出右边的符号并推入左边的符号来规约
            reducedStack.push(lhs);
            // 检查其余的
            if (recognise(cat, string, reducedStack)) {
                return true;
            }
        }
    }
}

return false;
}

```

基于统计的句法分析训练集是标注了结构的语料库。经过结构标注的语料库叫做树库，例如宾夕法尼亚大学树库(<http://www.cis.upenn.edu/~chinese/>)。

Stanford Parser(<http://nlp.stanford.edu/software/lex-parser.shtml>)实现了一个基于要素模型的句法分析器，其主要思想就是把一个词汇化的分析器分解成多个要素(factor)句法分析器。Stanford Parser 将一个词汇化的模型分解成一个概率上下文无关文法(PCFG)和一个依存模型。

5.3 相似度计算

相似度计算的任务是根据两段输入文本的相似度返回从 0 到 1 之间的相似度值：完全不相似，则返回 0；完全相同，则返回 1。衡量两段文字距离的常用方法有：海明距离(Hamming distance)、编辑距离、欧氏距离、文档向量的夹角余弦距离、最长公共子串。

文档向量的夹角余弦相似度方法将两篇文档看作是词的向量，如果 x 、 y 为两篇文档的向量，则：
$$\text{Cos}(x, y) = \frac{x \cdot y}{\|x\| \|y\|}$$

如果余弦相似度为 1，则 x 和 y 之间夹角为 0° ，并且除大小(长度)之外， x 和 y 是相同的；如果余弦相似度为 0，则 x 和 y 之间夹角为 90° ，并且它们不包含任何相同的词。

衡量文档相似度的另外一种常用方法是最长公共子串法。举例说明两个字符串 s_1 和 s_2

的最长公共子串(Longest Common Subsequence 简称 LCS)。

假设 $s1 = \{ a, b, c, b, d, a, b \}$, $s2 = \{ b, d, c, a, b, a \}$, 则从前往后找, $s1$ 和 $s2$ 的最长公共子串是 $LCS(s1, s2) = \{ b, c, b, a \}$, 如图 5-4 所示。

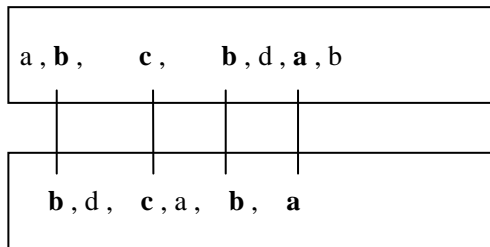


图 5-4 最长公共子串

$s1$ = “高新技术开发区北环海路 128 号”

$s2$ = “高技区北环海路 128 号”

则 $s1$ 和 $s2$ 的最长公共子串为 $LCS(s1, s2) =$ “高技区北环海路 128 号”。

使用动态规划的思想计算 LCS 的方法: 引进一个二维数组 $num[i][j]$, 用 $num[i][j]$ 记录 $s1$ 的前 i 个长度的子串与 $s2$ 的前 j 个长度的子串的 LCS 的长度。

自底向上进行递推计算, 那么在计算 $num[i][j]$ 之前, $num[i-1][j-1]$, $num[i-1][j]$ 与 $num[i][j-1]$ 均已计算出来。此时再根据 $s1[i-1]$ 和 $s2[j-1]$ 是否相等, 就可以计算出 $num[i][j]$ 。

计算两个字符串的最长公共子串的实现如下:

```
public static String longestCommonSubsequence(String s1, String s2){
    int[][] num = new int[s1.length()+1][s2.length()+1]; //初始化为 0 的二维数组

    //实际算法
    for (int i = 1; i <= s1.length(); i++)
        for (int j = 1; j <= s2.length(); j++)
            if (s1.charAt(i - 1)==s2.charAt(j - 1))
                num[i][j] = 1 + num[i-1][j-1];
            else
                num[i][j] = Math.max(num[i-1][j], num[i][j-1]);

    System.out.println("最长公共子串的长度是: " + num[s1.length()][s2.length()]);
}
```

```

int s1position = s1.length(), s2position = s2.length();
StringBuilder result = new StringBuilder();

while (s1position != 0 && s2position != 0) {
    if (s1.charAt(s1position - 1) == s2.charAt(s2position - 1)) {
        result.append(s1.charAt(s1position - 1));
        s1position--;
        s2position--;
    }
    else if (num[s1position][s2position - 1] >= num[s1position - 1][s2position]) {
        s2position--;
    } else {
        s1position--;
    }
}
result.reverse();
return result.toString();
}

```

为了返回 0 到 1 之间的一个相似度值，根据 LCS 计算的打分公式如下：

$$\text{Sim}(s1, s2) = \text{LCS-Length}(s1, s2) / \min(\text{Length}(s1), \text{Length}(s2))$$

最长公共子串(LCS)与夹角余弦相比，最长公共子串体现了词的顺序，而夹角余弦没有。显然，词的顺序在网页文档的相似性比较中本身就是一种重要的信息，一个由若干词语按顺序组成的句子和若干没有顺序的词语组成的集合有着完全不同的意义。完全有可能，两篇文档根本不同，但是夹角余弦值却很接近 1，特别是当文档数规模很大的时候。

最长公共子串中比较的是字符，可以把字符抽象成 **Token** 序列。也就是说把最长公共子串的方法抽象成最长公共子序列。对英文文档计算相似度，可以先按空格分词，然后再计算最长公共子序列。

比如说“书香门第 4 号门”和“书香门第 4 号”相似度高，但是“书香门第 4 号门”和“书香门第 5 号门”相似度低。所以单从字面比较无法更准确的反映其相似性。如果只差一个字，还要看有差别的这个字是什么类型的。所以应用带权重的最长公共子串方法，首先对输入字符串进行切分成词，然后对切分出来的词数组应用带权重的最长公共子序列的相似度打分算法。让比较的元素 E 增加权重属性 **Weight**。

```

public static double addSim(String n1, String n2) throws Exception {
    //首先执行同义词替换，例如把“地方税务局”替换成“地税局”
    String s1 = SynonymReplace.replace(n1);

```

```

String s2 = SynonymReplace.replace(n2);

double d = getSim(s1, s2);
if (d > 0.7) {
    ArrayList<PoiToken> ret1 = PoiTagger.basicTag(s1);
    ArrayList<PoiTokenWeight> poiW1 = new ArrayList<PoiTokenWeight>();

    for (int i = 0; i < ret1.size(); i++) {
        PoiToken poi = ret1.get(i);
        PoiTokenWeight poiTokenWeight = new PoiTokenWeight(poi);
        poiW1.add(poiTokenWeight);
    }
    ArrayList<PoiToken> ret2 = PoiTagger.basicTag(s2);
    ArrayList<PoiTokenWeight> poiW2 = new ArrayList<PoiTokenWeight>();

    for (int i = 0; i < ret2.size(); i++) {
        PoiToken poi = ret2.get(i);
        PoiTokenWeight poiTokenWeight = new PoiTokenWeight(poi);
        poiW2.add(poiTokenWeight);
    }
    return getSim(poiW1, poiW2);
}
return d;
}

//应用带权重的 LCS
double getSim(ArrayList<PoiToken>s1, ArrayList<PoiToken>s2){
    ...
    if (s1.get(i-1).equals(s2.get(j-1))) {
        num[i][j] = s1.get(i-1).Weight + num[i-1][j-1]; //增加权重
    } else{
        num[i][j] = Math.max(num[i-1][j], num[i][j-1]);
    }
    ...
}

```

上述的相似度计算方法中没有考虑词语之间的语义相关度。例如，“国道”和“高速公路”在字面上不相似，但是两个词在意义上有相关性。可以使用分类体系的语义词典提取词语语义相关度。基于语义词典的度量方法的计算公式，以下因素是最经常使用的：

- 1) 最短路径长度，即两个概念节点 A 和 B 之间所隔最少的边数量。
- 2) 局部网络密度，即从同一个父节点引出的子节点数量。显然，层次网络中的各个部分的密度是不相同的。例如，WordNet 中的 plant/flora 部分就非常密集的，一个父节点包含了数百个子节点。对于一个特定节点(和它的子节点)而言，全部的语义块是一个确定的数量，所以局部密度越大，节点（即父子节点或兄弟节点）之间的距离越近。
- 3) 节点在层次中的深度。在层次树中，自顶向下，概念的分类是由大到小，大类间的相似度肯定要小于小类间的。所以当概念由抽象逐渐变得具体，连接它们的边对语义距离计算的影响应该逐渐减小。
- 4) 连接的类型，即概念节点之间的关系的类型。在许多语义网络中，上下位关系是一种最常见的关系，所以许多基于边的方法也仅仅考虑 IS-A 连接。事实上，如果其他类型的信息可以得到，如部分关系和整体关系，那么其他的关系类型对于边权重计算的影响也同样应该考虑。
- 5) 概念节点的信息含量。它的基本思想[3]是用概念间的共享信息作为度量相似性的依据，方法是从语义网中获得概念间的共享信息，从语料库的统计数据中获得共享信息的信息量，综合两者计算概念间的相似性。这种方法基于一个假设：概念在语料库中出现的频率越高，则越抽象，信息量越小。
- 6) 概念的释义。在基于词典的模型中--不论是基于传统词典，还是基于语义词典--词典被视为一个闭合的自然语言解释系统，每一个单词都被词典中其他的单词所解释。如果两个单词的释义词汇集重叠程度越高，则表明这两个单词越相似。

将上述六个因素进一步合并，则可归为三大因素：结构特点，信息量和概念释义。

5.4 文档排重

不同的网站间转载内容的情况很常见。即使在同一个网站，有时候不同的 URL 地址可能对应同一个页面，或者存在同样的内容以多种方式显示出来。所以，网页需要按内容做文档排重。

判断文档的内容重复有很多种方法，语义指纹的方法比较高效。语义指纹直接提取一个文档的二进制数组表示的语义，通过比较相等来判断网页是否重复。语义指纹是一个很大的数组，全部存放在内存会导致内存溢出，普通的数据库效率太低，所以这里采用内存数据库 BerkeleyDB。可以通过 BerkeleyDB 判断该语义指纹是否已经存在。另外一种方法是通过第二章介绍过的布隆过滤器来判断语义指纹是否重复。

按词作维度的文档向量维度很高，可以把 SimHash 看成是一种维度削减技术。SimHash 除了可以用在文档排重上，还可以用在任何需要计算文档之间的距离应用上，例如文本分类或聚类。

5.4.1 语义指纹

提取网页的语义指纹的方法是：对于每张网页，从净化后的网页中，选取最有代表性的一组关键词，并使用该关键词组生成一个语义指纹。通过比较两个网页的指纹是否相同来判断两个网页是否相似。

网络一度出现过很多篇关于“罗玉凤征婚”的新闻报道。其中的两篇新闻对比如下：

文档 ID	文档 1	文档 2
标题	北大清华硕士不嫁的“最牛征婚女”	1 米 4 专科女征婚 求 1 米 8 硕士男 应征者如云
内容	...24 岁的罗玉凤，在上海街头发放了 1300 份征婚传单。传单上写了近乎苛刻的条件，要求男方北大或清华硕士，身高 1 米 76 至 1 米 83 之间，东部沿海户籍。而罗玉凤本人，只有 1 米 46，中文大专学历，重庆綦江人。...此事经网络曝光后，引起了很多人的兴趣。 “每天都有打电话、发短信求证，或者是应征。”罗玉凤说，她觉得满意的却寥寥无几，“到目前为止只有 2 个，都还不是特别满意”。...	...24 岁的罗玉凤，在上海街头发放了 1300 份征婚传单。传单上写了近乎苛刻的条件，要求男方北大或清华硕士，身高 1 米 76 至 1 米 83 之间，东部沿海户籍。而罗玉凤本人，只有 1 米 46，中文大专学历，重庆綦江人。...此事经网络曝光后，引起了很多人的兴趣。 “每天都有打电话、发短信求证，或者是应征。”罗玉凤说，她觉得满意的却寥寥无几，“到目前为止只有 2 个，都还不是特别满意”。...

对于这两篇内容相同的新闻，有可能提取出同样的关键词：

“罗玉凤”、“征婚”、“北大”、“清华”、“硕士”。

这就表示这两篇文档的语义指纹也相同。

为了提高语义指纹的准确性，需要考虑到同义词，例如：“北京华联”和“华联商厦”可以看成相同意义的词。最简单的，可以做同义词替换。把“开业之初，比这还要多的质疑的声音环绕在北京华联决策者的周围”替换为“开业之初，比这还要多的质疑的声音环绕在华联商厦决策者的周围”。

设计同义词词典的格式是：每行一个义项，前面是基本词，后面是一个或多个被替换的同义词。例如：

华联商厦 北京华联 华联超市

这样会把“北京华联”和“华联超市”替换成“华联商厦”。对指定文本，要从前往后查找同义词词库中每个要替换的词，然后实施替换。同义词替换的实现代码分为两步。首先是查找 Trie 树结构的词典过程：

```
public void checkPrefix(String sentence,int offset,PrefixRet ret) {
```

```

if (sentence == null || root == null || "".equals(sentence)) {
    ret.value = Prefix.MisMatch;
    ret.data = null;
    ret.next = offset;
    return ;
}
ret.value = Prefix.MisMatch; //初始返回值设为没匹配上任何要替换的词
TSTNode currentNode = root;
int charIndex = offset;
while (true) {
    if (currentNode == null) {
        return;
    }
    int charComp = sentence.charAt(charIndex) - currentNode.splitchar;

    if (charComp == 0) {
        charIndex++;

        if (currentNode.data != null) {
            ret.data = currentNode.data; //候选最长匹配词
            ret.value = Prefix.Match;
            ret.next = charIndex;
        }
        if (charIndex == sentence.length()) {
            return; //已经匹配完
        }
        currentNode = currentNode.eqKID;
    } else if (charComp < 0) {
        currentNode = currentNode.loKID;
    } else {
        currentNode = currentNode.hiKID;
    }
}
}

```

然后是同义词替换过程：

//输入待替换的文本，返回替换后的文本

```

public static String replace(String content) throws Exception{
    int len = content.length();
    StringBuilder ret = new StringBuilder(len);
    SynonymDic.PrefixRet matchRet = new SynonymDic.PrefixRet(null,null);

    for(int i=0;i<len;){
        //检查是否存在从当前位置开始的同义词
        synonymDic.checkPrefix(content,i,matchRet);
        if(matchRet.value == SynonymDic.Prefix.Match) //如果匹配上，则替换同义词
        {
            ret.append(matchRet.data);//把替换词输出到结果
            i=matchRet.next;//下一个匹配位置
        }
        else //如果没有匹配上，则从下一个字符开始匹配
        {
            ret.append(content.charAt(i));
            ++i;
        }
    }

    return ret.toString();
}

```

语义指纹生成算法如下：

第一步：将每个网页分词表示成基于词的特征向量，使用 $TF*IDF$ 作为每个特征项的权值。地名，专有名词等，名词性的词汇往往有更高的语义权重。

第二步：将特征项按照词权值排序。

第三步：选取前 n 个特征项，然后重新按照字符排序。如果不排序，关键词就找不到对应关系。

第四步：调用 MD5 算法，将每个特征项串转化为一个 128 位的串，作为该网页的指纹。

调用 fseg.result.FingerPrint 中的方法：

```

String fingerPrint = getFingerPrint("", "昨日，省城渊明北路一名 17 岁的少年在 6 楼晾毛巾时失足坠楼，摔在楼下的一辆面包车上。面包车受冲击变形时吸收了巨大的反作用力能量，从而“救”了少年一命。目前，伤者尚无生命危险。 据一位目击者介绍，事故发生在下午 2 时 40 分许，当时这名在某美发店工作的少年正站在阳台上晾毛巾，因雨天阳台湿滑而不小

```


心摔下。记者来到抢救伤者的医院了解到，这名少年名叫李嘉诚，今年 17 岁，系丰城市人。李嘉诚受伤后，他表姐已赶到医院陪护。据医生介绍，伤者主要伤在头部，具体伤情还有待进一步检查。");

```
String md5Value = showBytes(getMD5(fingerPrint));
System.out.println("FingerPrint:"+fingerPrint+" md5:"+md5Value);
```

MD5 可以将字符串转化成几乎无冲突的 hash 值。但是 MD5 速度比较慢，MurmurHash 或者 JenkinsHash 也可以生成冲突很少的 hash 值，在 Lucene 的企业搜索软件 Solr1.4 版本中提供了 JenkinsHash 实现的语义指纹，叫做 Lookup3Signature。调用 MurmurHash 生成 64 位的 hash 的代码：

```
public static long stringHash64(String str, int initial) {
    byte[] bytes = str.getBytes();
    return MurmurHash.hash64(bytes, initial);
}
```

5.4.2 SimHash

根据上面这个 MD5 方法的语义指纹无法找出特征近似的文档。例如，对于两个文档，如果两个文档相似，但这两个文档的 MD5 值却是完全不同的。关键字的微小差别会导致 MD5 的 hash 值差异巨大。这是 MD5 算法中的雪崩效应(avalanche effect)的结果。输入中一位的变化，散列结果中将有一半以上的位改变。

如果两个相似文档的语义指纹只相差几位或更少，这样的语义指纹叫做 SimHash。可以用海明距离来衡量近似的语义指纹。海明距离是针对长度相同的字符串或二进制数组而言的。对于二进制数组 s 和 t，H(s, t)是两个数组对应位有差别的数量。例如：1011101 和 1001001 的海明距离是 2。下面的方法可以按位比较计算两个 64 位的长整型之间的海明距离。

```
public static int hamming(long l1, long l2) {
    int counter = 0;
    for (int c=0; c<64; c++)
        counter += (l1 & (1L << c)) == (l2 & (1L << c)) ? 0 : 1;
    return counter;
}
```

这种按位比较的方法比较慢，可以把两个长整型按位异或(XOR)，然后计算结果中 1 的个数，结果就是海明距离。例如计算 A 和 B 两数的海明距离：

A = 1 1 1 0

B = 0 1 0 0

$A \text{ XOR } B = 1010$

计算 1010 中 1 的个数是 2。实现代码如下：

```
public static int hammingXOR(long l1, long l2) {
    long lxor = l1 ^ l2; //按位异或
    return BitUtil.pop(lxor); //计算 1 的个数
}
```

假设可以得到文档的一系列特征，每个特征有不同的重要度。计算文档对应的 SimHash 值的方法是把每个特征的 Hash 值叠加到一起形成一个 SimHash。计算过程如图 5-2 所示。

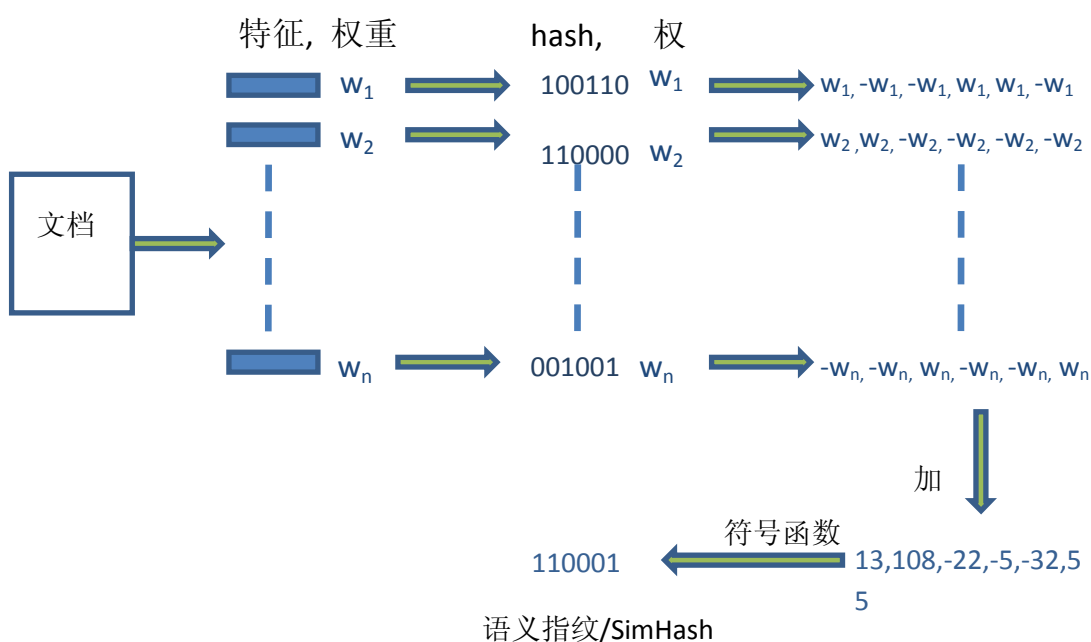


图 5-2 语义指纹计算过程

可以把特征权重看成特征在 SimHash 结果的每一位上的投票权。权重大的特征的投票权大，权重小的特征投票权小。所以权重大的特征更有可能影响文档的 SimHash 值中的很多位，而权重小的特征影响文档的 SimHash 值位数很少。

假定 SimHash 的长度为 64 位，文档的 SimHash 计算过程如下：

- (1) 初始化长度为 64 的数组，该数组的每个元素都是 0。
- (2) 对于特征列表循环做如下处理：

①取得每个特征的 64 位的 hash 值。

②如果这个 hash 值的第 i 位是 1，则将数组的第 i 个数加上该特征的权重；反之，如果 hash 值的第 i 位是 0，则将数组的第 i 个数减去该特征的权重。

(3) 完成所有特征的处理，数组中的某些数为正，某些数为负。SimHash 值的每一位与数组中的每个数对应，将正数对应的位设为 1，负数对应的位设为 0，就得到 64 位的 0/1 值的位数组，即最终的 SimHash。

输入特征和权重数组，返回 SimHash 的代码如下：

```
public static long simHash(String[] features,int[] weights){
    int[] hist = new int[64];//创建直方图

    for(int i=0;i<features.length;++i) {
        long addressHash = stringHash(features[i]);//生成特征的 hash 码
        int weight = weights[i];
        /* 更新直方图 */
        for (int c=0; c<64; c++)
            hist[c] += (addressHash & (1 << c)) == 0 ? -weight : weight;
    }

    /* 从直方图计算位向量 */
    long simHash=0;
    for (int c=0; c<64; c++)    {
        long t= ((hist[c]>=0)?1:0);
        t <<= c;
        simHash |= t ;
    }

    return simHash;
}
```

要生成好的 SimHash 编码，就要让完全不同的特征差别尽量大，而相似的特征差别比较小。如果特征是枚举类型，只有两个可能的取值，例如是 Open 和 Close。Open 返回二进制位全是 1 的散列编码，而 Close 则返回二进制位全是 0 的散列编码。下面的代码将为指定的枚举值生成尽量不一样的散列编码。

```
public static long getSimHash(MatterType matter){
    int b=1; //记录用多少位编码可以表示一个枚举类型的集合
    int x=2;
    while(x<MatterType.values().length) {
        b++;
    }
```

```

        x = x<<1;
    }

    long simHash = matter.ordinal();
    int end = 64/b;
    for(int i=0;i<end;++i) {
        simHash = simHash << b; //枚举值按枚举类型总个数向左移位
        simHash += matter.ordinal();
    }
    return simHash;
}

```

中文字符串特征的散列算法:

```

public static int byte2int(byte b) {
    return (b & 0xff);
}

private static int MAX_CN_CODE = 6768;//最大中文编码
private static int MAX_CODE = 6768+117;//最大编码

//取得中文字符的散列编码
public static int getHashCode(char c) throws UnsupportedOperationException{
    String s = String.valueOf(c);
    int maxVal = 6768;
    byte[] b = s.getBytes("gb2312");
    if(b.length==2)    {
        int index = (byte2int(b[0]) - 176) * 94  + (byte2int(b[1]) - 161);
        return index;
    }
    else if(b.length==1)    {
        int index = byte2int(b[0]) - 9 + MAX_CN_CODE;
        return index;
    }
    return c;
}

//取得中文字符串的散列编码

```

```
public static long getSimHash(String input) throws UnsupportedEncodingException{
    if(input==null || "".equals(input)) {
        return -1;
    }
    int b=13; //记录用多少位编码可以表示一个中文字符

    long simHash = getHashCode(input.charAt(0));
    int maxBit = b;
    for(int i=1;i<input.length();++i) {
        simHash *= MAX_CODE; //把汉字串看成是 MAX_CODE 进制的
        simHash += getHashCode(input.charAt(i));
        maxBit += b;
    }

    long origialValue = simHash;

    for(int i=0;i<=(64/maxBit;++i) {
        simHash = simHash << maxBit;
        simHash += origialValue;
    }
    return simHash;
}
```

SimHash 的计算依据是要比较的对象的特征，对于结构化的记录可以按列提取特征，而非结构化的文档特征则不明显。如果是新闻，特征可以用标题、最长的几句话。提取特征前，最好先进行一些简单的预处理，如全角转半角。

基于 SimHash 的文档排重流程如图 5-3。

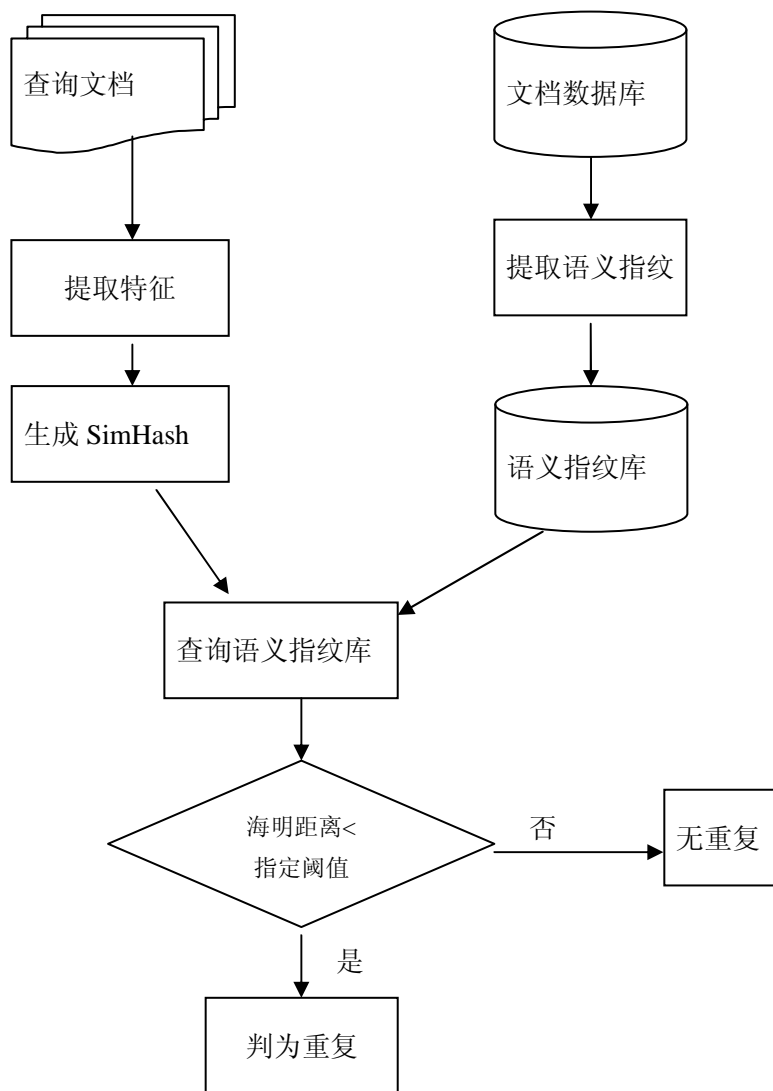


图 5-3 文档排重计算过程

根据文档排重流程设计出对结构化记录使用排重接口的方式：首先根据文档集合生成一个语义指纹的集合(fingerprintSet)，然后根据待查重的文档生成的一个 simHash 值来查找近似的文档集合。fingerprintSet.getSimSet(key, k)返回 和 key 相似的数据集合。

```
//返回一条记录的 simHash
```

```
long simHashKey = poiSimHash.getHash(poi,address,tel);
```

```
//根据一条记录的 simHash 返回相似的数据
```

```
HashSet<SimHashData> ret = fingerprintSet.getSimSet(simHashKey, k);
```

把文档转换成 SimHash 后，文档排重就变成了海明距离计算问题。海明距离计算问题是：给出一个 f 位的语义指纹集合 F 和一个语义指纹 fg ，找出 F 中是否存在与 fg 只有 k 位差异的语义指纹。

最基本的一种方法是逐次探查法，先把所有和 fg 差 k 位的指纹找出来，然后用折半查找法查找排好序的指纹集合 F 。需要多少次折半查找呢？首先借助组合数生成器

CombinationGenerator 来生成和给定的语义指纹差别在 2 位以内的语义指纹：

```
long fingerPrint = 1L; //语义指纹
int[] indices; //组合数生成的一种组合结果
//生成差 2 位的语义指纹
CombinationGenerator x = new CombinationGenerator(64, 2);
int count = 0; //计数器
while (x.hasMore()) {
    indices = x.getNext(); //取得组合数生成结果
    long simFP = fingerPrint;
    for (int i = 0; i < indices.length; i++) {
        simFP = simFP ^ 1L << indices[i]; //翻转对应的位
    }
    System.out.println(Long.toBinaryString(simFP)); //打印相似语义指纹
    ++count;
}
```

这里运行的结果是 $\text{count}=2016$ 。因为是从 64 位中选有差别的 2 位，所以计算公式是 $C_{64}^2 = 64 * 63 / 2 = 2016$ 。也就是说，要找出和给定语义指纹差别在 2 位以内的语义指纹需要探查 2016 次。

逐次探查法完整的查找过程如下：

```
//输入要查找的语义指纹和 k 值，如果找到相似的语义指纹则返回真，否则返回假
public boolean containSim(long fingerPrint, int k) {
    //首先用二分法直接查找语义指纹
    if (contains(fingerPrint)) {
        return true;
    }

    //然后用逐次探查法查找
    int[] indices;

    for (int ki = 1; ki <= k; ++ki) {
        //找差 1 位直到差 k 位的
        CombinationGenerator x = new CombinationGenerator(64, ki);
        while (x.hasMore()) {
            indices = x.getNext();
            long simFP = fingerPrint;
```

```

        for (int i = 0; i < indices.length; i++) {
            simFP = simFP ^ 1L << indices[i];
        }
        //查找相似语义指纹
        if(contains(simFP)) {
            return true;
        }
    }
}

return false;
}

```

在 k 值很小，而要找的语义指纹集合 S 中的元素不太多的情况下，可以用比逐次探查法更快的方法查找。

如果 k 值很小，例如 $k=1$ ，可以给指纹集合 S 中每个元素生成出和这个元素差别在 1 位以内的元素。对于长整型的元素，差别在 1 位以内的元素只有 65 种可能。然后再把所有这些新生成的元素排序，最后用折半查找算法查询这个排好序的语义指纹集合 simSet 。生成法查找近似语义指纹的整体流程如图 5-6 所示。

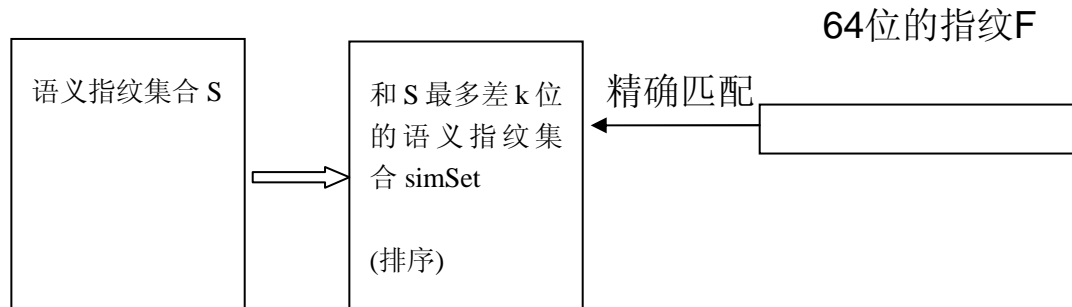


图5-6 生成法查找近似语义指纹

对给定 SimHash 值 n 生成差 1 位的 Hash 值的代码如下：

```

for(int j=0;j<64;j++){
    long newSimHash=n^1L<<j; //生成和 n 差 1 位的 Hash 值
}

```

在 k 值比较小的情况下，例如 k 不大于 3，介绍另外一种快速计算方法。假设我们有一个已经排序的容量为 2^d 的 f 位指纹集合。看每个指纹的高 d 位。该高 d 位具有以下性质：因为指纹集合中有很多的位组合存在，所以高 d 位中只有少量重复。

SimHash 排重的假设：

1. 整个表中排列组合的部分很少，不太可能出现例如：一批 8 位 SimHash，前 4 位都一样，但后 4 位出现 16 种 0-1 组合的情况。
2. 整个表在前 d 位 0-1 分布不会有很多的重复。

这两个假设得到排重的基础：前 d 位上的 0-1 分布足以当成一个指针；有能力快速搜索前 d 位。整个编码空间类似一句古话：太极生两仪，两仪生四象，四象生八卦...

现在找一个接近于 d 的数字 d'，由于整个表是排好序的，所以一趟搜索就能找出高 d' 位与目标指纹 F 相同的指纹集合 f'。因为 d' 和 d 很接近，所以找出的集合 f' 也不会很大。

$$|f'| = |s|/2^{d'}$$

最后在集合 f' 中查找和 F 之间海明距离为 k 的指纹也就很快了。总的思想是：先把检索的集合缩小 $2^{d'}$ 倍，然后在小集合中逐个检查，看剩下的 f-d' 位的海明距离是否满足要求。假设 k 值为 3，有 2^{34} 个语义指纹待查找，则有：

第一种策略：f 分为 6 块，分别是 11、11、11、11、10、10 位，最坏的可能是其中 3 块里各出现 1 位海明距离不同，把这三块限制到低位，换言之，选三块精确匹配的到高位，有 $C_6^3 = 20$ 种选法，因此需要复制 20 个 T 表。对每个表高位做精确匹配，需要匹配 31~33 位，(11*3、11*2+10、11+10*2) 那么 f' 的个数大概是 $|s|/2^{31} = 2^{[34-31]} = 8$ ，即精确匹配一次，产生大约 8 个需要算海明距离的 SimHash。

第二种策略：f 先分为 4 块，各 16 位，选 1 个精确匹配块到高位的可能有 $C_4^1 = 4$ 种选法，再对剩下 3 块 48 位切分，分成 4 块，各 12 位，选 1 个精确匹配块到高位的可能有 $C_4^1 = 4$ 种， $4*4 = 16$ ，一共要复制 16 次 T 表。那么高位就有 $16+12 = 28$ 位。每次精确匹配 28 位后，产生大约 $2^{[34-28]} = 64$ 个需要算海明距离的 SimHash。

假设 SimHash 有 f 位，现在找一个接近于 d 的数字 d'，由于整个表是排好序的，所以一趟精确匹配就能找出高 d' 位与目标指纹 F 相同的指纹集合 f'。($f' = 2^{[d-d']}$)，因为 d' 和 d 很接近，所以找出的集合 f' 也不会很大。海明距离的比较就在 f-d' 位上进行。要确保海明位不同的几位都被限制在 f-d' 上就需要考虑 f 上不同位的组合可能，即让海明位不会出现在前 d' 上，每种 f 位上的组合就需要复制一次 T。

算法本质就是采用分治法，把问题分解成更小的几个子问题，降低问题需要处理的数据规模。利用空间 (原空间的 t 倍) 和并行计算换时间。分治法查找海明距离在 k 以内的语义指纹算法步骤如下：

- 1) 先复制原表 T 为 T_t 份: T_1, T_2, \dots, T_t 。
- 2) 每个 T_i 都关联一个 p_i 和一个 π_i , 其中 p_i 是一个整数, π_i 是一个置换函数, 负责把 p_i 个 bit 位换到高位上。
- 3) 应用置换函数 π_i 到相应的 T_i 表上, 然后对 T_i 进行排序。
- 4) 然后对每一个 T_i 和要匹配的指纹 F 、海明距离 k 做如下运算: 使用 F' 的高 p_i 位检索, 找出 T_i 中高 p_i 位相同的集合, 在检索出的集合中比较剩下的 $f-p_i$ 位, 找出海明距离小于或等于 k 的指纹。
- 5) 最后合并所有 T_i 中检索出的结果。

举一个实现的例子, 假设有 100 亿左右(2^{34})的语义指纹, simhash 有 64 位。所以 $f = 64$, $d = 34$ 。海明距离 k 值是 3。

例如, SimHash 长度是 64 位, 按 16 位拆分, 复制 4 份, 分别是 T_1 、 T_2 、 T_3 、 T_4 。这里, T_i 在 T_{i-1} 的基础上左移 16 位。精确匹配每个复制表的高 16 位。然后在精确匹配出来的结果中找差 3 位以内的 SimHash。按 16 位拆分的查找方法如图 5.3 所示。

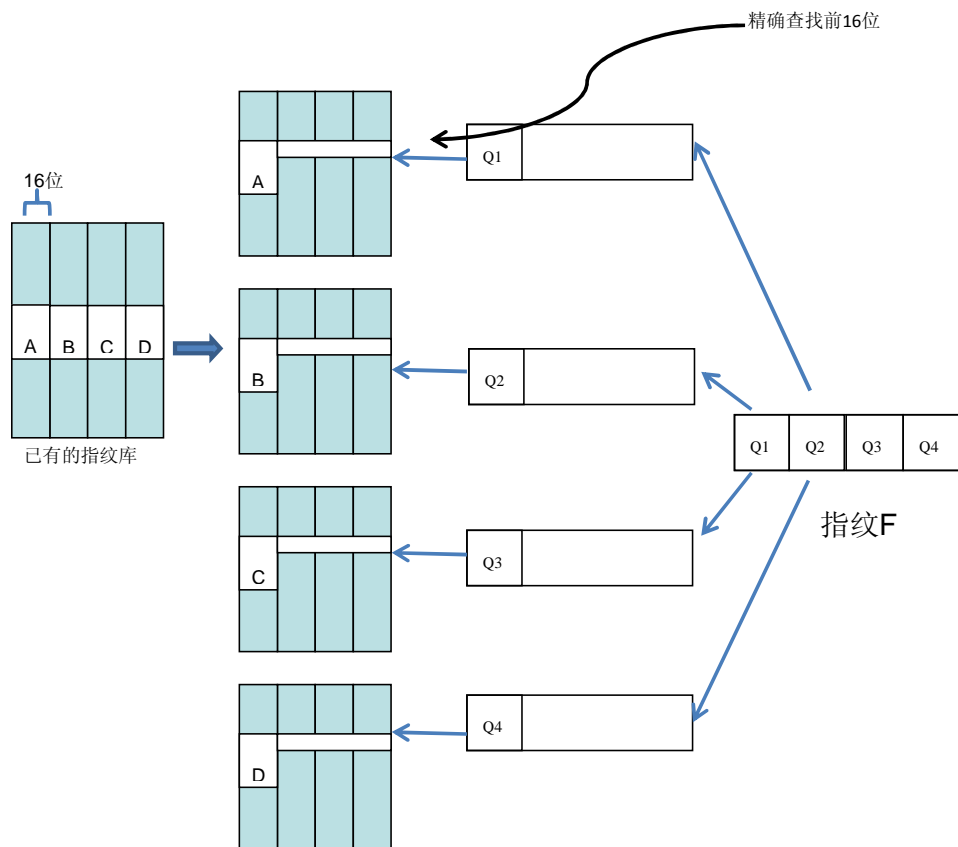


图 5.3 分 4 块查找语义指纹

比较用长整型表示的无符号 64 位语义指纹的代码如下：

```
public static boolean isLessThanUnsigned(long n1, long n2) {
    return (n1 < n2) ^ ((n1 < 0) != (n2 < 0));
}

static Comparator<SimHashData> comp = new Comparator<SimHashData>(){
    public int compare(SimHashData o1, SimHashData o2){
        if(o1.q==o2.q) return 0;
        return (isLessThanUnsigned(o1.q,o2.q)) ? 1: -1;
    }
}; // 比较无符号 64 位

static Comparator<Long> compHigh = new Comparator<Long>(){
    public int compare(Long o1, Long o2){
        o1 |= 0xFFFFFFFFFFFFL;
        o2 |= 0xFFFFFFFFFFFFL;
        //System.out.println(Long.toBinaryString(o1));
        //System.out.println(Long.toBinaryString(o2));
        //System.out.println((o1 == o2));
        if(o1.equals(o2)) return 0;
        return (isLessThanUnsigned(o1,o2)) ? 1: -1;
    }
}; // 比较无符号 64 位中的高 16 位

public void sort(){//对四个表排序
    t2.clear();
    t3.clear();
    t4.clear();
    for(SimHashData simHash:t1) {
        long t = Long.rotateLeft(simHash.q, 16);
        t2.add(new SimHashData(t,simHash.no));

        t = Long.rotateLeft(t, 16);
        t3.add(new SimHashData(t,simHash.no));
```

```

        t = Long.rotateLeft(t, 16);
        t4.add(new SimHashData(t,simHash.no));
    }

    Collections.sort(t1, comp);
    Collections.sort(t2, comp);
    Collections.sort(t3, comp);
    Collections.sort(t4, comp);
}

```

5.4.3 分布式文档排重

在批量版本的海明距离问题中，有一批查询语义指纹，而不是一个查询语义指纹。

假设已有的语义指纹库存储在文件 F 中，批量查询语义指纹存储在文件 Q 中。80 亿个 64 位的语义指纹文件 F 大小是 64GB。压缩后小于 32GB。一批有 1M 大小的语义指纹需要批量查询，因此假设文件 Q 的大小是 8MB。Google 把文件 F 和 Q 存放在 GFS 分布式文件系统。文件分成多个 64MB 的组块。每个组块复制到一个集群中的 3 个随机选择的机器上。每个组块在本地系统存储成文件。

使用 MapReduce 框架，整个计算可以分成两个阶段。在第一阶段，有和 F 的组块数量一样多的计算任务(在 MapReduce 术语中，这样的任务叫做 mapper)。

每个任务以整个文件 Q 作为输入在某个 64MB 的组块上求解海明距离问题。

```

public class SimHashMapper extends Mapper<ArrayList<SimHashData>, SimHashSet4,
SimHashData, HashSet<SimHashData>>{
    static int k=3;

    //在 64MB 的组块 fingerprintSet 上求解海明距离问题
    public void map(ArrayList<SimHashData> q,
                    SimHashSet4 fingerprintSet,
                    Context context) throws IOException, InterruptedException {
        for (SimHashData query : q) {
            HashSet<SimHashData> ret = fingerprintSet.getSimSet(query.q, k);
            if(ret!=null)
                //收集相似的语义指纹集合
                context.write(query, ret);
        }
    }
}

```

```
}
```

一个任务以发现的一个近似重复的语义指纹列表作为输出。在第二阶段，MapReduce 收集所有任务的输出，删除重复发现的语义指纹，产生一个唯一的、排好序的文件。

```
public class SimHashReducer
    extends
        Reducer<SimHashData,      ArrayList<SimHashData>,      SimHashData,
HashSet<SimHashData>> {
    public void reduce(SimHashData key,
        Iterable<ArrayList<SimHashData>> values,
        Context context) throws IOException, InterruptedException {
        HashSet<SimHashData> dup = new HashSet<SimHashData>();
        for (ArrayList<SimHashData> val : values) {
            dup.addAll(val);
        }
        context.write(key, dup);
    }
}
```

Google 用 200 个任务(mapper)，扫描组块的合并速度在 1GB 每秒以上。压缩版本的文件 Q 大小大约是 32GB(压缩前是 64GB)。因此总的计算时间少于 100 秒。压缩对于速度的提升起了重要作用，因为对于固定数量的任务(mapper)，时间大致正比于文件 Q 的大小。

通过 Job 类构建一个任务。

```
Job job = new Job(conf, "Find Duplicate");
job.setJarByClass(FindDup.class);
```

5.5 中文关键词提取

关键词提取是文本信息处理的一项重要任务，例如可以利用关键词提取来发现新闻中的热点问题。和关键词类似，很多政府公文也有主题词描述。上下文相关广告系统也可能会用到关键词提取技术。可以给网页自动生成关键词来辅助搜索引擎优化(SEO)。

有很多种方法可以应用于关键词提取。例如，基于训练的方法和基于图结构挖掘的方法、基于语义的方法等。KEA(<http://www.nzdl.org/Kea>)是一个开源的关键词提取项目。

5.5.1 关键词提取的基本方法

提取关键词整体流程如图 5-4:

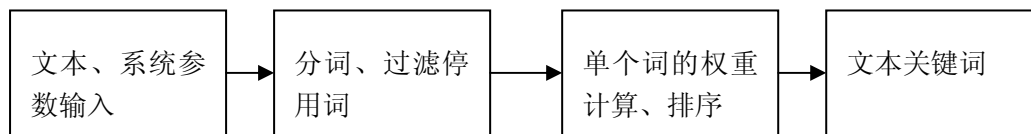


图 5-4 关键词提取流程图

为了调节计算过程中用到的参数，可以建立关键词提取训练库。训练库包括训练文件(x.txt)和对应的关键词文件(x.key)。

依据如下几点来判断词的权重：

- 利用 $TF*IDF$ 公式，计算每个可能的关键词的 $TF*IDF$ ：统计词频和词在所有文档中出现的总次数。 TF (Term Frequency)代表词频， IDF (Invert Document Frequency)代表文档频率的倒数。比如说“的”在 100 文档中的 40 篇文档中出现过，则文档频率 DF (Document Frequency)是 40， IDF 是 $1/40$ 。“的”在第一篇文档中出现了 15 次，则 $TF*IDF$ (的) = $15 * 1/40 = 0.375$ 。另外一个词“反腐败”在这 100 篇文档中的 5 篇文档中出现过，则 DF 是 5， IDF 是 $1/5$ 。“反腐败”在第一篇文档中出现了 5 次，则 $TF*IDF$ (反腐败) = $5 * 1/5 = 1$ 。结果是： $TF*IDF$ (反腐败) > $TF*IDF$ (的)。

- 利用位置信息：开始和结束位置的词往往更可能是关键词。比如，利用下面的经验公式：

```
double position = t.startOffset() / content.length();
position = position * position - position + 2;
```

或者利用一个分段函数，首段或者末段的词的权重更大。

- 标题中出现的词比内容中的词往往更重要。
- 利用词性信息：关键词往往是名词或者名词结尾的词，而介词，副词，动词结尾的词一般不能组成词组。

- 利用词或者字的互信息： $I(X,Y) = \log_2 \frac{P(X,Y)}{P(X)P(Y)}$ 互信息大的单字有可能是关键词。

比如说 $I(\text{福,娃}) = \log_2 \frac{P(\text{福,娃})}{P(\text{福})P(\text{娃})}$ 。

- 利用标点符号：《 》 和 “ ” 之间的文字更有可能是关键词。例如：“汉芯一号”造假案。
- 构建文本词网络：将单个词语当作词网络的节点，若两个词语共同出现在文本的一句话中，网络中对应节点建立一条权值为 1.0 的边。在文本词网络上运行图结构挖掘算法(例

如 Page Rank 或 HITS 算法)对节点进行权值计算。

- 把出现的名词词按语义聚类，然后提取出有概括性的词作为关键词。

首先定义词及其权重的描述类：

```
public class WordWeight implements Comparable<WordWeight> {
    public String word; //单词
    public double weight; //权重
    protected WordWeight(String word, double weight) {
        this.word = word;
        this.weight = weight;
    }
    public String toString() {
        return word + ":" + weight;
    }
    public int compareTo(WordWeight obj) {
        WordWeight that = obj;
        return (int) (that.weight - weight);
    }
}
```

返回关键词的主要代码如下：

```
//全部待选的关键词放入 PairingHeap 实现的优先队列
PairingHeap<WordWeight> h = new PairingHeap<WordWeight>();
//把单个单词放入优先队列
for (Entry<String,Double> it : wordTable.entrySet()) {
    word = it.getKey();
    java.lang.Double tempDouble;
    if( word.length() ==1)
        tempDouble = new Double(0.0);
    else
        tempDouble = it.getValue();
    h.insert( new WordWeight(word,tempDouble.doubleValue()) );
}
//把词组放入优先队列
for(WordWeight we : ngram) {
    h.insert(we);
}
```

```
retNum = Math.min(retNum,h.size()); //返回的关键词数量

WordWeight[] fullResults = new WordWeight[retNum]; //关键词返回结果
for(int i=0;i<retNum;++i)    {
    fullResults[i] = (WordWeight)h.deleteMin();
}
```

其中为了返回权重最大的几个关键词，用到了 **PairingHeap** 实现的优先队列。

5.5.2 HITS 算法应用于关键词提取

首先介绍 **HITS**(Hypertext Induced Topic Selection)算法的原理与实现。**HITS** 算法可以选出有向带权重的图中的最重要的节点。

每个节点有 **Authority** 和 **Hub** 两个值。**Authority** 值可以理解为该节点的权威性，也就是重要度。若 **B** 节点上有指向 **A** 节点的边，则称 **B** 为 **A** 的导入节点，这说明 **B** 认为 **A** 有指向价值，是一个“重要”的节点。所以 **Authority** 值是指向该节点的所有节点的 **Hub** 值之和。

如果一个节点指向另外一个节点，则可以看做这个节点向另外一个节点投了一票。**Hub** 值可以理解为该节点的投票可信度。**Hub** 值是该节点指向的那些节点的 **Authority** 值之和。可以看到，**Authority** 和 **Hub** 值以互相递归的方式定义。

HITS 算法执行一系列迭代过程，每个过程由如下两步组成：**Authority 更新**：更新每个节点的 **Authority** 值成为指向该节点的所有节点的 **Hub** 值之和。**Hub 更新**：更新每个节点的 **Hub** 值成为该节点指向的所有节点的 **Authority** 值之和。

用如下的算法计算一个节点的 **Hub** 值和 **Authority** 值：

1. 开始设置每个节点的 **Hub** 值和 **Authority** 值为 1。
2. 执行 **Authority** 更新规则。
3. 执行 **Hub** 更新规则。
4. 对 **Hub** 值和 **Authority** 值归一化。其中，**Hub** 值的归一化方式是每个 **Hub** 值除以所有的 **Hub** 值的平方和。**Authority** 值的归一化方式是每个 **Authority** 值除以所有的 **Authority** 值的平方和。
5. 重复第 2 步直达到指定迭代次数，或者 **Hub** 值和 **Authority** 值变化很小为止。

基本的算法实现如下：


```

public void computeHITS(int numIterations) {
    while(numIterations-->0) { //如果没有超过指定叠代次数
        for (int i = 1; i <= graph.numNodes(); i++) { //更新 Authority 值
            Map<Integer,Double> inlinks    = graph.inLinks(new Integer(i));
            double authorityScore = 0;

            for (Integer id:inlinks.keySet()) {
                authorityScore += (hubScores.get(id)).doubleValue();
            }
            authorityScores.put(new Integer(i), new Double(authorityScore));
        }
        for (int i = 1; i <= graph.numNodes(); i++) //更新 hub 值
        {
            Map<Integer,Double> outlinks    = graph.outLinks(new Integer(i));
            double hubScore = 0;
            for (Integer id:outlinks.keySet()) {
                hubScore += (authorityScores.get(id)).doubleValue();
            }

            hubScores.put(new Integer(i), new Double(hubScore));
        }
        normalize(authorityScores); //归一化 authority 值
        normalize(hubScores); //归一化 hub 值
    }
}

```

如果认为节点之间的连接的重要程度不一样，也就是指向关系有强弱之分。考虑节点之间的连接重要度的实现。

```

public void computeWeightedHITS(int numIterations) {
    while(numIterations-->0) {
        for (int i = 1; i <= graph.numNodes(); i++) {
            Map<Integer,Double> inlinks    = graph.inLinks(new Integer(i));
            Map<Integer,Double> outlinks    = graph.outLinks(new Integer(i));
            double authorityScore = 0;
            double hubScore = 0;
            for (Entry<Integer,Double> in:inlinks.entrySet()) {
                authorityScore+=(hubScores.get(in.getKey())).doubleValue()
in.getValue();
            }
        }
    }
}

```

```

    }

    for (Entry<Integer,Double> out:outlinks.entrySet()) {
        hubScore += (authorityScores.get(out.getKey()).doubleValue() *
out.getValue());
    }

    authorityScores.put(new Integer(i),new Double(authorityScore));
    hubScores.put(new Integer(i),new Double(hubScore));
}
normalize(authorityScores);
normalize(hubScores);
}
}

```

为了确保引号里的词一定最重要，例如：“金正日”加了引号，所以这个词很重要。还有些词在文档的标题中出现，也很重要。所以要修改 HITS 算法，把节点的初始 **authority** 提高，并且保证它会不变的很低。例如，把用 **TF*IDF** 方法算的词权重作为节点的初始 **hub** 或 **authority**。

5.5.3 从网页中提取关键词

从网页中提取关键词的处理流程是：

1. 从网页中提取正文。
2. 从正文中提取关键词。

在 **H1** 标签中的词，或者黑体加粗的词可能更重要，更有可能是网页的关键词。另外，Meta 中的 **KeyWords** 描述也有可能真实的反映了该网页的关键词，例如：

```
<meta name="keywords" content="公判" />
```

5.6 相关搜索词

搜索引擎中往往有个可选搜索词的列表，当搜索结果太少的时候，可以帮助用户扩展搜索内容，或者当搜索结果过多的时候，可以帮助用户深入定向搜索。一种方法是从搜索日志中挖掘字面相似的词作为相关搜索词列表。从一个给定的词语挖掘多个相关搜索词，可以用编辑距离为主的方法查找一个词的字面相似词，如果候选的相关搜索词很多，就要筛选出最相关的 10 个词。

5.6.1 挖掘相关搜索词

下面是利用 Lucene 筛选给定词的最相关词的方法。

```
private static final String TEXT_FIELD = "text";
/**
 * @param words 候选相关词列表
 * @param word 要找相关搜索词的种子词
 * @return
 * @throws IOException
 * @throws ParseException
 */
static String[] filterRelated(HashSet<String> words, String word) {
    StringBuilder sb = new StringBuilder();
    for(int i=0;i<word.length();++i) {
        sb.append(word.charAt(i));
        sb.append(" ");
    }
    RAMDirectory store = new RAMDirectory();
    //按字生成索引和查找，也可以按细粒度的词分开
    IndexWriter writer = new IndexWriter(store, new StandardAnalyzer(), true);
    for(String text:words) {
        Document document = new Document();
        Field textField = new Field(TEXT_FIELD, text,
                                   Field.Store.YES, Field.Index.TOKENIZED);
        document.add(textField);
        writer.addDocument(document);
    }
    writer.close();
    IndexSearcher searcher = new IndexSearcher(store);
    QueryParser queryParser = new QueryParser(TEXT_FIELD,
                                              new StandardAnalyzer());
    Query query = queryParser.parse(sb.toString());
    Hits hits = searcher.search(query);
    int maxRet = Math.min(10, hits.length());
    String[] relatedWords = new String[maxRet];
    for (int i = 0; i < maxRet ; i++) {
```

```

        Document document = hits.doc(i);
        String text = document.get(TEXT_FIELD);
        System.out.println(text);
        relatedWords[i]=text;
    }
    searcher.close();
    store.close();
    return relatedWords;
}

```

上述代码整理出这样的相关词表，第一列是关键词，后续是 10 个以内的相关搜索词：

集福轩婚礼%集福轩

手机定位跟踪系统%手机定位系统%手机定位%手机定位仪器

喷绘材料卖店电话%我要喷绘材料卖店电话

厦门房产%厦门租房%厦门新闻%厦门桑拿%房产%青岛房产%厦门%恒雄房产

送水果%送水%水果

三星传真机%三星手机

另外一种方法，可以把多个用户共同查询的词看成相关搜索词，需要有记录用户 IP 的搜索日志才能实现。

可以通过 RelatedEngine 类查找某个关键词的相关词。

```

RelatedEngine re =new RelatedEngine(
        new File("D:/lg/work/xiaoxishu/dic/relatedwords.txt"));
String word = "徐家汇";
String[] relatedWords = re.getRelated(word);
for(String w : relatedWords) {
    System.out.println(w);
}

```

输出如下：

上海徐家汇
徐汇
徐家汇价格是
上房徐家汇路附近有吗

当我们需要为新建立的搜索引擎开发相关搜索时,如果没有搜索日志而用户文本很多的时候,可以:

1. 首先运行 **IndexMaker** 从待搜索的文档中提取关键词并生成索引;
2. 然后运行 **RelatedWords** 从索引生成相关词表。

另外还考虑用日志记录用户对相关搜索词的选择。如果用户也选择了这个词,那搜索词肯定和这个选择词相关了。

隐含语义索引(latent semantic indexing 简称 LSI)的原理是在相同的上下文中的词有相似的含义。<http://code.google.com/p/airhead-research/>是 Java 版本的实现。

5.6.2 使用多线程计算相关搜索词

如果要计算任意两个查询词之间的相关性,则发现相关搜索词的时间复杂度是 $O(n^2)$ 。例如要分析 10 兆多的相关搜索词的用时,则单线程的计算量可能长达 24 小时。

我们使用 Java 中自带的轻量级线程池来实现数据分析。JDK1.5 以后的版本提供了一个轻量级线程池 **ThreadPool**。可以使用线程池执行一组任务,最简单的任务不返回值给主调线程。要返回值的任务可以实现 **Callable<T>**接口,线程池执行任务并通过 **Future<T>**的实例获取返回值。

实现 **Callable** 方法的任务类的主要实现:

```
public class FindSimCall implements Callable<String[]> {
    private HashSet<String> words; // 总的搜索词集合
    private String s;              // 待发现相关词的词

    public FindSimCall(HashSet<String> w, String source) {
        words = w;
        s = source;
    }

    @Override
    public String[] call() throws Exception {
        System.out.println(s);
        // 形成 related words 列表
        // ...
        return relatedWords;
    }
}
```

```

    }
}

```

主线程类的实现如下：

```

int threads = 4;
ExecutorService es = Executors.newFixedThreadPool(threads);

Set<Future<String[]>> set = new HashSet<Future<String[]>>();

for (final String s : words) {
    FindSimCall task = new FindSimCall(words,s);
    Future<String[]> future = es.submit(task);
    set.add(future);
}

FileOutputStream fos = new FileOutputStream(relatedWordsFile);
OutputStreamWriter osw = new OutputStreamWriter(fos,"GBK");
BufferedWriter writer = new BufferedWriter(osw);

for (Future<String[]> future : set) {
    String[] ret = future.get();

    for(String word:ret) {
        writer.write("%"+word);
    }
    writer.write( "\r\n" );
}
writer.close();

```

采用线程池可以充分利用多核 CPU 的计算能力，并且简化了多线程的实现。

5.7 信息提取

据说，看同一段视频，聪明人和一般人的差别在于：他会从视频中提取出自己感兴趣的信息。本节只介绍文本信息提取。

从文本中抽取用户感兴趣的事件、实体和关系，被抽取出来的信息以结构化的形式描述。然后存储在数据库中，为各种应用提供服务。例如：从新闻报道中抽取出什么地方发生车祸，什么地方堵车。

例如：从新闻报道中抽取出恐怖事件的详细情况：时间、地点、作案者、受害者、袭击目标、使用的武器等。从经济新闻中抽取出公司发布新产品的情况：公司名、产品名、发布时间、产品性能等。

华盛顿大学开发的开放式互联网信息提取系统“TEXTRUNNER”提取实体和它们之间的关系。例如“海德公园”和“英国”的关系是“位于”。当用户提问“海德公园位于哪里？”时，系统可以根据提取出的信息回答“英国”。依赖手工编写的提取规则，或者手工标注的训练例子来实现信息提取(Information Extraction)。

GATE(<http://gate.ac.uk>)是一个应用广泛的信息抽取的开放型基础架构，该系统对语言处理的各个环节——从语料收集、标注、重用到系统评价均能提供很好的支持。我们这里实现一个简化版本的信息抽取系统。

输入“北京盈智星公司”切分后标注成：“北京/行政区划 盈智星/关键词 公司/功能词”。根据标注的结果可以提取出“盈智星”这样的关键词。有个基本的词典用来存放行政区划，功能词表等特征，例如“北京”这个词是在一个行政区划词表中，“公司”在另外一个功能词词表中。提取地址会碰到很多词典中没有的需要识别的未登录词，例如“高东镇高东二路”，需要把“高东二路”这样不在词典中的路名识别出来。可以先把输入串抽象成待识别的序列“镇后缀 UNKNOWN 号码 街后缀”，然后利用规则(也叫模板)来识别并提取信息。未登录地址识别规则可以表示成如下的形式：

镇后缀 未登录街道 =>镇后缀 UNKNOWN 号码 街后缀

转换成代码实现：

```
lhs = new ArrayList<AddressSpan>(); //左边的符号
rhs = new ArrayList<AddressType>(); //右边的符号
//镇后缀 UNKNOWN 号码 街后缀
rhs.add(AddressType.SuffixTown);
rhs.add(AddressType.Unknown);
rhs.add(AddressType.No);
rhs.add(AddressType.SuffixStreet);
//镇后缀 未登录街道
lhs.add(new AddressSpan(1,AddressType.SuffixTown)); //归约长度是 1
//把“UNKNOWN 号码 街后缀”3个符号替换成“未登录街道”，因此归约长度是 3
lhs.add(new AddressSpan(3,AddressType.Street));
//加到规则库
addProduct(rhs, lhs);
```

“UNKNOWN 号码 街后缀”合并成“街道”，可以记录下内部结构，这样方便后续处理。

为了提高提取准确性，规则往往设计成比较长的形式。长的规则往往更多的参考上下文，

覆盖面小，但是更准确。短的规则会影响更多的提取结果，可能这一条信息靠这条规则提取正确了，却有更多的其它记录受影响。

设计规则存储格式：“右边的符号列表@左边的符号列表”。左边的符号列表用“整数值，类型”来表示，例如：

```
Town,SuffixProvince @ 2, Province
Unknow,SuffixTown,Unknow,No,SuffixStreet@2,Town,3,Street
Town,Unknow,No,SuffixStreet@1,Town,3,Street
City,Unknow,Street@1,City,2,Street
Unknow,SuffixStreet@2,Street
Unknow,SuffixLandMark@2,LandMark
Unknow,No,SuffixStreet@3,Street
```

读入规则的程序：

```
StringTokenizer st = new StringTokenizer(line, "@");//分成左右符号串
StringTokenizer rhst = new StringTokenizer(st.nextToken(), ",");//逗号分隔
StringTokenizer lhst = new StringTokenizer(st.nextToken(), ",");//逗号分隔
ArrayList<PoiSpan> rhs = new ArrayList<PoiSpan>();//右边的符号
ArrayList<PoiType> lhs = new ArrayList<PoiType>();//左边的符号
while (rhst.hasMoreTokens()) {
    lhs.add(PoiType.valueOf(rhst.nextToken()));//左边类型
}
while (lhst.hasMoreTokens()) {
    rhs.add(new PoiSpan(Integer.parseInt(lhst.nextToken()), //右边符号长度
        PoiType.valueOf(lhst.nextToken()))); //右边符号类型
}
addProduct(lhs, rhs); //加入到规则库
```

可以从 Java 源代码中抽取出规则到配置文件中，实现代码如下：

```
String[] sa=str.split("addProduct");// 分割每一条规则
for(String s : sa){ //遍历每一条规则并处理
    //通过正则表达式匹配找出该条规则中所有的右边的符号
    Pattern p=Pattern.compile("rhs\\.add\\.POIType1\\.([a-zA-Z]+)");
    Matcher m=p.matcher(s);
    String result="";
    while(m.find()) {
        result=result+m.group(1)+" ";
    }
}
```



```

        if("").equals(result)) {
            break;
        }

        //通过正则表达式匹配找出该条规则中所有的左边的符号
        result=result.substring(0, result.length()-1)+"@";

        Pattern
p2=Pattern.compile("lhs\\s\\.add\\.new\\sPoiSpan1\\.([0-9]+),\\sPOIType1\\.([a-zA-Z]+)");
        Matcher m2=p2.matcher(s);
        while(m2.find()) {
            String num=m2.group(1);
            String type=m2.group(2);
            result+=num+", "+type+", ";
        }
        //输出提取出的规则
        System.out.println(result.substring(0, result.length()-1));
    }

```

规则替换可能会进入死循环,因为可能出现重复应用规则的情况。如果规则的左边部分小于右边部分,也就是说替换后的长度越来越短,应用这样的规则不会导致死循环。当规则的左边部分和右边部分相等时,可以用 **Token** 的类型的权重和来衡量,规则左边部分的权重和必须小于右边部分的权重和。这样的规则让应用于匹配序列的 **Token** 的类型的权重和越来越小,所以也不会产生死循环。使用 **ordinal** 方法取得的枚举类型的内部值大小作为权重。下面是检查规则的实现方法:

```

/**
 * 规则校验
 *
 * @return true 表示规则不符合规范 false 表示符合符合规范
 */
public boolean check(ArrayList<DocType> key, ArrayList<DocSpan> lhs) {
    boolean isEqual = false;
    for (DocSpan span : lhs) {
        if (span.length > 1) {
            return isEqual;
        }
    }
    int leftCount = 0;

```

```

int rightCount = 0;
for (DocType dy : key) {
    leftCount += dy.ordinal();
}
for (DocSpan sd : lhs) {
    rightCount += sd.type.ordinal();
}

if (leftCount <= rightCount) {
    isEqual = true;
}
return isEqual;
}

```

规则很多的时候，需要看一段文本匹配上了哪一条规则。或者考察某一条具体的规则可能产生的影响，先总体执行一遍数据，然后看那些数据用了这条规则。

信息提取的流程如下：

1. 定义词的类别；
2. 根据词库做全切分；
3. 最大概率动态规划求解；
4. HMM 词性标注；
5. 基于规则的未登录词识别；
6. 根据切分和标注的结果提取信息。

例如，为农业相关的文档提取出作物名称，对应季节，适用地区等信息。

```

public enum DocType {
    Product, //作物名称
    Pronoun, //代词
    Address, //地名
    Season, //季节
    Start, //虚拟类型，开始状态
    End //虚拟类型，结束状态
}

```

可以自己建几个简单的词表。例如季节词表 `season.txt`。存放内容如：

春
夏
秋
冬

作物名称有个 `product.txt` 词表。存放内容如：

大豆
高粱

然后通过 `DicDoc` 类加载这些词，代码如下：

```
private DicDoc() {  
    //加载字典  
    //“product.txt” 是一类词， DocType.Product 定义好这类词性  
    load("product.txt", DocType.Product); //农作物  
    load("address.txt", DocType.Address); //地址  
    load("season.txt", DocType.Season); //季节  
}
```

信息提取的关键在于定义相关规则，用户定义好规则后程序会按照指定的规则提取相关信息，规则越多，提取的信息越精确。另外，可以把需要优先匹配的规则放到前面。因为规则库中放在前面的规则会先匹配上。

还可以用信息提取的方法提取网页中的信息。例如下面这段描述图书的网页片段：“``出版社：``中国工人出版社`
`”。要从中提取出版社信息。

把标签放到不同的词典文件中，例如“``”和“`
`”，“出版社：”。这样可以根据规则提取出“中国工人出版社”。

5.8 拼写检查与建议

输错电话号码，往往只是得到简单的提示“没有这个电话号码”。但是在搜索框中输入错误的搜索词，搜索引擎往往会提示“您是不是要找”这个正确的词。这个功能也叫做“Did you mean”。

拼写检查是查询处理极为重要的一个组成部分。在网络搜索引擎用户提交的查询中有大约 10%到 15%的拼写错误，拼写检查就是对错误的词给出正确的提示。如果有个正确的词和用户输入的词很近似，则用户的输入可能是错误的。

查询日志中包含大量的简单错误的例子，如下面：

poiner sisters -> pointer sisters

brimingham news -> birmingham news

ctamarn sailing -> catamaran sailing

类似这些错误可以通过建立正误词表来检查。然而，除此之外，将有许多查询日志包含与网站，产品，公司相关的词，对于这样的开放类的词不可能在标准的拼写词典中发现。以下是来自同一个查询日志一些例子：

akia 1080i manunal -> akia 1080i manual

ultimatwarcade -> ultimatearcade

mainsourcebank -> mainsource bank

因此不存在万能的词表，垂直(网站)搜索引擎往往需要整理和自己行业(网站)相关的词库才能达到好的匹配效果。可以从搜索日志中挖掘出“错误词->正确词”这样的词对，例如“飞利浦->飞利浦”。

根据正误词表替换用户输入。

```
public static String replace(String content) {
    int len = content.length();
    StringBuilder ret = new StringBuilder(len);
    ErrorDic.PrefixRet matchRet = new ErrorDic.PrefixRet(null,null);

    for(int i=0;i<len;){
        errorDic.checkPrefix(content,i,matchRet);//检查是否存在从当前位置开始的错词
        if(matchRet.value == ErrorDic.Prefix.Match) {
            ret.append(matchRet.data);
            i=matchRet.next;//下一个匹配位置
        }
        else //从下一个字符开始匹配
        {
            ret.append(content.charAt(i));
            ++i;
        }
    }
    return ret.toString();
}
```

因为在各种语言中导致用户输入错误的原因不一样,所以每种语言的正误词对的挖掘方式有不一样的地方。对英文单词的搜索需要专门针对英文的拼写检查,对中文词的搜索需要专门针对中文的拼写检查。

为了讨论对搜索引擎查询最有效的的拼写检查技术,我们首先看下单词拼写检查的概率模型:

$$\text{Spell}(w) = \underset{c \in C}{\operatorname{argmax}} P(c | w) = \underset{c \in C}{\operatorname{argmax}} \frac{P(w | c)P(c)}{P(w)}$$

对于任何 c 来讲,出现 w 的概率 $P(w)$ 都是一样的, 从而我们在上式中忽略它, 写成:

$$\text{Spell}(w) = \underset{c \in C}{\operatorname{argmax}} P(w | c)P(c)$$

这个式子有三个部分, 从右到左分别是:

1. $P(c)$: 文章中出现一个正确拼写词 c 的概率。也就是说, 在英语文章中, c 出现的概率有多大呢?因为这个概率完全由英语这种语言决定, 一般称之为语言模型。例如, 英语中出现 the 的概率 $P(\text{'the'})$ 就相对高, 而出现 $P(\text{'zxxzxxzy'})$ 的概率接近 0(假设后者也是一个词的话)。

2. $P(w|c)$: 在用户想键入 c 的情况下敲成 w 的概率。因为这个是代表用户会以多大的概率把 c 敲错成 w , 因此这个被称为误差模型。

3. argmax_c : 用来枚举所有可能的 c 并且选取概率最大的那个词。因为有理由相信, 一个正确的单词出现的频率高, 用户又容易把它敲成另一个错误的单词, 那么, 那个敲错的单词应该被更正为这个正确的。

为什么把最简单的一个 $P(c|w)$ 变成两项复杂的式子来计算? 因为 $P(c|w)$ 就是和这两项同时相关的, 因此拆成两项反而容易处理。举个例子, 比如一个单词 thew 拼错了, 看上去 thaw 应该是正确的, 因为就是把 a 打成 e 了。然而, 也有可能用户想要的是 the , 因为 the 是英语中常见的一个词, 并且很有可能打字时候手不小心从 e 滑到 w 了。因此, 在这种情况下, 我们想要计算 $P(c|w)$ 就必须同时考虑 c 出现的概率和从 c 到 w 的概率。把一项拆成两项让这个问题更加容易更加清晰。

对于给定词 w 可以通过编辑距离挑选出相似的候选正确词 c 的集合。编辑距离越小, 候选正确词越少, 计算也越快。76%的正确词和错误词的编辑距离是 1。所以还需要考虑编辑距离是 2 的情况。99%的正确词和错误词的编辑距离在 2 以内。因此对于拼写检查来说, 查找出编辑距离在 2 以内的候选正确词 c 的集合就可以了。这是一个模糊匹配的问题。

5.8.1 模糊匹配问题

如何从一个大的正确词表中找和输入词编辑距离小于 k 的词集合？逐条比较正确词和输入词的编辑距离太慢。

构建一个有限状态自动机准确的识别出和目标词在给定的编辑距离内的字符串集合。可以输入任何词，然后自动机可以基于是否和目标词的编辑距离最多不超过给定距离从而接收或拒绝它。

而且，由于 FSA 的内在特性，可以在 $O(n)$ 时间内实现。这里， n 是测试字符串的长度。而标准的动态规划编辑距离计算方法需要 $O(m*n)$ 时间，这里 m 和 n 是两个输入单词的长度。因此编辑距离自动机可以更快的检查许多单词和一个目标词是否在给定的在最大距离内。

编辑距离自动机的基本想法是：构建一个有限状态自动机准确的识别出和目标词在给定的编辑距离内的字符串集合。可以输入任何词，然后自动机可以基于是否和目标词的编辑距离最多不超过给定距离从而接收或拒绝它。而且，由于 FSA 的内在特性，可以在 $O(n)$ 时间内判断是可以接收或应该拒绝。这里， n 是测试字符串的长度。而标准的动态规划编辑距离计算方法需要 $O(m*n)$ 时间，这里 m 和 n 是两个输入单词的长度。因此编辑距离自动机可以更快的检查许多单词和一个目标词是否在给定的在最大距离内。

单词“food”的编辑距离自动机形成的非确定有限状态自动机，最大编辑距离是 2。开始状态在左下，状态使用 n^e 标记风格命名。这里 n 是目前为止正确匹配的字符数， e 是错误数量。垂直转换表示未修改的字符，水平转换表示插入，两类对角线转换表示替换(用*标记的转换)和删除(空转换)。

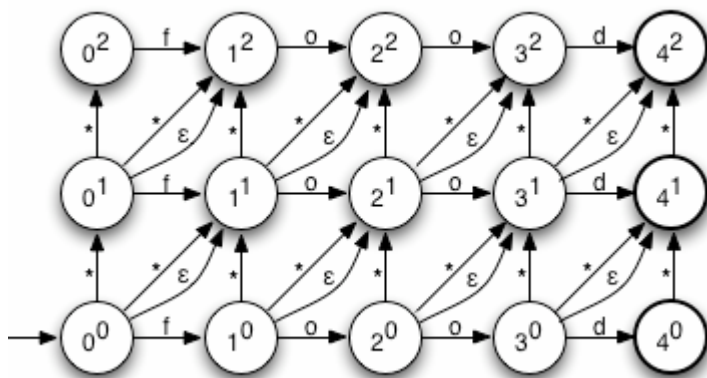


图 5-2 编辑距离自动机

单词“food”的长度是 4，所以图 5-2 有 5 列。允许 2 次错误，所以有 3 行。

编辑距离自动机实现代码如下：

```
public static NFA levenshteinAutomata(String term, int k) {
    NFA nfa = new NFA(new State(0, 0)); // 根据初始状态构建非确定有限状态机
```

```

for (int i = 0; i < term.length(); ++i) {
    char c = term.charAt(i);
    for (int e = 0; e < (k + 1); ++e) {
        //正确字符
        nfa.addTransition(new State(i, e), c, new State(i + 1, e));
        if (e < k) {
            //删除
            nfa.addTransition(new State(i, e), NFA.ANY, new State(i, e + 1));
            //插入
            nfa.addTransition(new State(i, e), NFA.EPSILON,
                               new State(i + 1, e + 1));
            //替换
            nfa.addTransition(new State(i, e), NFA.ANY, new State(i + 1, e + 1));
        }
    }
}
for (int e = 0; e < (k + 1); ++e) {
    if (e < k)
        nfa.addTransition(new State(term.length(), e), NFA.ANY,
                           new State(term.length(), e + 1));
    nfa.addFinalState(new State(term.length(), e)); //设置结束状态
}
return nfa;
}

```

76%的正确词和错误词的编辑距离是 1。23%的正确词和错误词的编辑距离是 2。需要在状态对象中记住接收的字符串和原串有几处错误。

看某个单词是否和给定单词相似：

```

//构建编辑距离自动机
NFA levenshteinAutomata = NFA.levenshteinAutomata("food",1);
//根据幂集构造转换成确定有限状态机
DFA dfa = levenshteinAutomata.toDFA();
//看单词 foxd 是否能够被接收
System.out.println(dfa.accept("foxd"));

```

把正确的词典和条件(Levenshtein automata)都表示成 DFA，有可能高效的对两个 DFA 取交集(intersect)，从词典中找到满足条件的词。步调一致的遍历两个 DFA，仅跟踪两个 DFA

都共有的边，并且记录走过来的路径。任何时候两个 DFA 都在结束状态，输出词典 DFA 对应的单词。

```
public static ArrayList<String> intersect(DFA dfa1, DFA dfa2) {
    ArrayList<String> match = new ArrayList<String>();//找到的正确单词集合
    Stack<StackValue> stack = new Stack<StackValue>();
    stack.add(new StackValue("", dfa1.startState, dfa2.startState));
    while (!stack.isEmpty()) {
        StackValue stackValue = stack.pop();
        Set<Character> ret =
            intersection(dfa1.edges(stackValue.s1), dfa2.edges(stackValue.s2));
        for(char edge:ret) {
            State state1 = dfa1.next(stackValue.s1, edge);
            State state2 = dfa2.next(stackValue.s2, edge);
            if(state1!=null && state2!=null) {
                String prefix = stackValue.s+edge;
                stack.add(new StackValue(prefix, state1, state2));
                if(dfa1.isFinal(state1) && dfa2.isFinal(state2)){
                    match.add(prefix);
                }
            }
        }
    }
    return match;
}
```

可以使用标准 Trie 树代替 DFA，标准 Trie 树中存储了正确词库。使用方法如下：

```
//错误词
NFA lev = NFA.levenshteinAutomata("foo",1);
DFA dfa = lev.toDFA();

//正确词表
Trie<String> stringTrie = new Trie<String>();
stringTrie.add("food", "food");
stringTrie.add("hammer", "hammer");
stringTrie.add("hammock", "hammock");
stringTrie.add("ipod", "ipod");
stringTrie.add("iphone", "iphone");
```



```
//返回相似的正确的词
```

```
ArrayList<String> match = DFA.intersect(dfa, stringTrie);
```

现在已经实现了从一个大的正确词表中找和输入词编辑距离小于 k 的词集合。为了计算 $P(c|w)$ ，需要用到 $P(w|c)$ 和 $P(c)$ 。根据正确词表中的词频来估计 $P(c)$ 。根据 w 和 c 之间的编辑距离估计 $P(w|c)$ 。所以实际返回的是如下对象的列表：

```
public class RightWord implements Comparable<RightWord> {
    public String word; //正确词
    public int errors; //错误数，也就是正确词和错误词之间的编辑距离
    public int freq; //在词表中的词频

    public RightWord(String w, int e, int f) {
        word = w;
        errors = e;
        freq = f;
    }

    public int compareTo(RightWord o) { //先比较错误数，再比较词频
        int diff = this.errors - o.errors;
        if(diff!=0) {
            return diff;
        }

        return o.freq - this.freq;
    }
}
```

为了简化计算选择正确词，先取编辑距离小的正确词，对于编辑距离相同的正确词，则取词频高的正确词。

5.8.2 英文拼写检查

对英文报关公司名 101919 条统计，有拼写出错的为 16663 条，出错概率为 16.35%。大部分是正确的，如果所有的词都在正确词表中，则不必再查找错误。否则先检查错误词表。最后仍然不确定的，提交查询给搜索引擎，看看是否有错误。

正确词的词典格式每行一个词，分别是词本身和词频。样例如下：

```
biogeochemistry : 1
repairer : 3
```

```
wastefulness : 3
battier : 2
awl : 3
preadapts : 1
surprisingly : 3
stuffiest : 3
```

因为互联网中的新词不断出现，正确的词并不是来源于固定的词典，而是来源于搜索的文本本身。下面直接从文本内容提取英文单词。不从索引库中提取的原因是 Term 可能经过词干化处理过了，所以我们用 StandardAnalysis 再次处理。

```
java.io.StringReader input = new java.io.StringReader(content);
TokenStream tokenizer = new StandardTokenizer(input);
for (Token t = tokenizer.next(); t != null; t = tokenizer.next()){
    if( isAllLetter(t.termText()) &&
        (t.termText().length()>=3) &&
        (t.termText().length()<=30) ){
        System.out.println(t.termText());
        fpSource.write(t.termText().toLowerCase());
        fpSource.write(" : 1\n");
    }
}
```

可以根据发音计算用户输入词和正确词表的相似度，还可以根据字面的相似度来判断是否输入错误，并给出正确的单词提示。也可以参考一下开源的拼写检查器(spell checker)的实现，例如 Aspell(<http://aspell.net/>)。

特别的，考虑公司名中的拼写错误。一般首字母拼写错误的可能性很小。可以简单的先对名称排序，然后再比较前后两个公司名就可以检测出一些非常相似的公司名称了。

另外可以考虑抓取 Google 的拼写检查结果。

```
public static String getGoogleSuggest(String name) throws Exception {
    String searchWord = URLEncoder.encode(name,"utf-8");
    String searchURL = "http://www.google.com/search?q=" + searchWord;
    String strPages=DownloadPage.downloadPage(searchURL);//下载页面

    String suggestWord="";
    Parser parser=new Parser(strPages); //使用 HTMLParser 解析返回的网页
    NodeFilter filter=
        new AndFilter(new TagNameFilter("a"),new HasAttributeFilter("class","spell"));
```

```
//取得符合条件的第一个节点
NodeList nodelist=parser.extractAllNodesThatMatch(filter);
int listCount=nodelist.size();

if(listCount>0){
    TagNode node=(TagNode)nodelist.elementAt(0);
    if (node instanceof LinkTag) {<a> 标签
        LinkTag link = (LinkTag) node;
        suggestWord = link.getLinkText();//链接文字
    }
}

return suggestWord;
}
```

5.8.3 中文拼写检查

和英文拼写检查不一样，中文的用户输入的搜索词串的长度更短，从错误的词猜测可能的正确输入更加困难。这时候需要更多的借助正误词表，词典文本格式如下：

代款:贷款

阿地达是:阿迪达斯

诺基压:诺基亚

飞利浦:飞利浦

寂么沙洲冷:寂寞沙洲冷

欧米加:欧米茄

欧米枷:欧米茄

爱力信:爱立信

西铁成:西铁城

瑞新:瑞星

登心绒:灯心绒

这里，前面一个词是错误的词条，后面是对应的错误词条。为了方便维护，我们还可以把这个词库存放在数据库中：

```
CREATE TABLE CommonMisspellings (
    [misword] [varchar] (50) COLLATE Chinese_PRC_CI_AS NULL , --错误词
    [rightword] [varchar] (50) COLLATE Chinese_PRC_CI_AS NULL  --正确词
)
```

除了人工整理，还可以从搜索日志中挖掘相似字符串来找出一些可能的正误词对。比较常用的方法是采用编辑距离(Levenshtein Distance)来衡量两个字符串是否相似。编辑距离就是用来计算从原串(s)转换到目标串(t)所需要的最少的插入，删除和替换的数目。例如源串是“诺基压”，目标串是“诺基亚”，则编辑距离是1。

当一个用户输入错误的查询词没有结果返回时，他可能会知道输入错误，然后用正确的词再次搜索。从日志中能找出来这样的行为，进而找出正确/错误词对。

例如，日志中有这样的记录：

```
2007-05-24 00:41:41.0781|DEBUG |221.221.167.147||喀尔喀蒙古|2
```

...

```
2007-05-24 00:43:45.7031|DEBUG|221.221.167.147||喀爾喀蒙古|0
```

...

处理每行日志信息，用 StringTokenizer 返回“|”分割的字符串，代码如下。

```
StringTokenizer st = new StringTokenizer(line,"|");
while(st.hasMoreTokens()) { //有更多的内容
    System.out.println(st.nextToken()); //取得子串
}
```

假设日志中有搜索结果返回的是正确词，无搜索结果返回的是错误词。挖掘日志的程序如下：

```
//存放挖掘的词及搜索出的结果数
HashMap<String,Integer> searchWords = new HashMap<String,Integer>();
while((readline=br.readLine())!=null) {
    StringTokenizer st = new StringTokenizer(readline,"|");
    if(!st.hasMoreTokens()) continue;
    st.nextToken();
    if(!st.hasMoreTokens())continue;
    st.nextToken();
```

```

        if(!st.hasMoreTokens())continue;
        st.nextToken();
        if(!st.hasMoreTokens())continue;
        st.nextToken();
        if(!st.hasMoreTokens())continue;
        st.nextToken();
        if(!st.hasMoreTokens())continue;
        st.nextToken();
        //存放搜索词
        String key = st.nextToken();
        if(key.indexOf(":")>=0) {
            continue;
        }
        //如果已经处理过这个词就不再处理
        if(searchWords.containsKey(key)) {
            continue;
        }
        if(!st.hasMoreTokens())
        {
            continue;
        }
        String results = st.nextToken();
        int resultCount = Integer.parseInt(results);//得到搜索出的结果数

        for(Entry<String,Integer> e : searchWords.entrySet()) {
            int diff= Distance.LD(key, e.getKey()) ;
            if(diff ==1 && key.length()>2) {
                if( resultCount == 0 && e.getValue()>0 ) {
                    // e.getKey()是正确词，key 是错误词
                    System.out.println(key +":"+ e.getKey());
                    bw.write(key +":"+ e.getKey()+"\r\n");
                }
                else if(e.getValue()==0 && resultCount>0) {
                    // key 是正确词，e.getKey()是错误词
                    System.out.println(e.getKey() +":"+ key);
                    bw.write(e.getKey() +":"+ key+"\r\n");
                }
            }
        }
    }

```

```

    }
    searchWords.put(key, resultCount); //存放当前词及搜索出的结果数
}

```

可以挖掘出如下一些错误、正确词对：

瑜伽服:瑜伽服

落丽塔:洛丽塔

巴甫洛:巴甫洛夫

hello kiitty:hello kitty

...

除了根据搜索日志挖掘正误词表，还可以根据拼音或字形来挖掘。例如根据拼音挖掘出“周杰论:周杰伦”，根据字形挖掘出“浙江移动:浙江移动”。搜索“HTMLParser 源代码”时，提示：您是不是要用 SVN 客户端下载？

5.9 自动摘要

减少原文的长度而保留文章的主要意思叫做摘要。摘要有各种形式。和搜索关键词相关的摘要，叫做动态摘要；只和文本内容相关的摘要叫做静态摘要。搜索引擎中显示的搜索结果就是关键词相关摘要的例子。如果在文档的内容列中没有搜索关键词，则搜索结果中内容列的动态摘要退化成静态摘要。

按照信息来源来分有来源于单个文档的摘要和合并多个相关文档意思的摘要。单文档摘要精简一篇文章的主要意思，多文档摘要同时可以过滤掉出现在多篇文档中的重复内容。

除了用于搜索结果中显示的摘要，文本自动摘要还可以用于手机 WAP 网站显示摘要信息，还可以用于发送短信息。

5.9.1 自动摘要技术

摘要的实现方法有摘取性的和概括性的。摘取性的方法相对容易实现，通常的实现方法是摘取文章中的主要句子。

MEAD(<http://www.summarization.com/mead/>)是一个功能完善的多文档摘要软件，不过是 Perl 实现的。Classifier4J(<http://classifier4j.sourceforge.net/>)包含一个简单的文本摘要实现，方法是抽取指定文本中的重要句子形成摘要。使用它的例子如下：

```
String input = "Classifier4J is a java package for working with text. Classifier4J includes a
```

```
summariser.";
//输入文章内容及摘要中需要返回的句子个数。
String result = summariser.summarise(input, 1);
```

返回结果是: "Classifier4J is a java package for working with text."。

5.9.2 自动摘要的设计

自动摘要主要方法有基于句子重要度的方法和基于篇章结构的方法。基于句子重要度的方法相对成熟，基于篇章结构的方法还处在研究阶段。

Classifier4J 也是采用了句子重要度计算的简化方法。Classifier4J 通过统计高频词和句子分析来实现自动摘要。主要流程如下：

1. 取得高频词；
2. 把内容拆分成句子；
3. 取得包含高频词的前 k 个句子。
4. 将句子按照在文中出现的顺序重新排列，添加适当的分隔符后输出。

统计文本中最常出现的 k 个高频词的基本方法如下：

- 1) 在遍历整个单词序列时，使用一个散列表记录所有的单词频率。散列表的关键字是词，而值是词频。花费 $O(n)$ 时间。
- 2) 对散列表按值从大到小排序。使用通常的排序算法花费 $O(n*\lg(n))$ 时间。
- 3) 排序后，取前 k 个词。

为了优化第 2 步和第 3 步，可以不对散列表全排序，直接取前 k 个值最大的词。用到的一个方法是从数组中快速的选取最大的 k 个数。

快速排序基于分而治之(divide and conquer)策略。数组 $A[p..r]$ 被划分为两个（可能空）子数组 $A[p..q-1]$ 和 $A[q+1..r]$ ，使得 $A[p..q-1]$ 中的每个元素都小于等于 $A(q)$ ，而且小于等于 $A[q+1..r]$ 中的元素。这里 $A(q)$ 称为中值。可以根据快速排序的原理设计接口如下：

```
//根据随机选择的中值来选取最大的 k 个数，输入参数说明如下：
// a 待选取的数组
// size 数组的长度
// k 前 k 个值最大的词
// offset 偏移量
```

```
selectRandom(ArrayList<WordFreq> a, int size, int k, int offset)
```

根据快速排序的原理，实现选取最大的 k 个数的方法如下：

```
//把数组中的两个元素交换位置
```

```
public static <E extends Comparable<? super E>> void swap(ArrayList<E> a, int i, int j) {
    E tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}
```

```
static void selectRandom(ArrayList<WordFreq> a, int size, int k, int offset) {
    if (size < 5) { //采用简单的冒泡排序方法对长度小于 5 的数组排序
        for (int i = offset; i < (size + offset); i++)
            for (int j = i + 1; j < (size + offset); j++)
                if (a.get(j).compareTo(a.get(i)) < 0)
                    swap(a, i, j);
        return;
    }
    Random rand = new Random(); //随机选取一个元素作为中值
    int pivotIdx = partition(a, size, rand.nextInt(size) + offset, offset);
    if (k != pivotIdx) {
        if (k < pivotIdx) {
            selectRandom(a, pivotIdx - offset, k, offset);
        } else {
            selectRandom(a, size - pivotIdx - 1 + offset, k, pivotIdx + 1);
        }
    }
}
```

```
static int partition(ArrayList<WordFreq> a, int size, int pivot, int offset) {
    WordFreq pivotValue = a.get(pivot); //取得中值
    swap(a, pivot, size - 1 + offset);
    int storePos = offset;
    for (int loadPos = offset; loadPos < (size - 1 + offset); loadPos++) {
        if (a.get(loadPos).compareTo(pivotValue) < 0) {
            swap(a, loadPos, storePos);
            storePos++;
        }
    }
}
```



```

swap(a, storePos, size - 1 + offset);
return (storePos);
}

```

参考 Classifier4J 的实现方法，中文自动摘要的基本实现方法如下 5 个步骤：

1. 通过中文分词，统计词频和词性等信息，抽取出关键词。
2. 把文章划分成一个个的句子。
3. 通过各句中关键词出现的情况定义出句子的重要度。
4. 确定前 K 个最重要的句子为文摘句。
5. 把文摘句按照在原文中出现的顺序输出成摘要。

文本摘要的主体程序如下：

```

ArrayList<CnToken> pltem = Tagger.getFormatSegResult(content);//分词
//关键词及对应的权重
HashMap<String,Integer> keyWords = new HashMap<String,Integer>(10);
WordCounter wordCounter = new WordCounter();//词频统计
for (int i = 0; i < pltem.size(); ++i) {
    CnToken t = pltem.get(i);
    if (t.type().startsWith("n")) {
        wordCounter.addNWord(t.termText());//增加名词词频
    } else if (t.type().startsWith("v")) {
        wordCounter.addVWord(t.termText());//增加动词词频
    }
}
//取得出现的频率最高的五个名词
WordFreq[] topNWords = wordCounter.getWords(wordCounter.wordNCount);
for (int i = 0; i < topNWords.length; i++) {
    keyWords.put(topNWords[i].word, topNWords[i].freq);
}
//取得出现的频率最高的五个动词
WordFreq[] topVWords = wordCounter.getWords(wordCounter.wordVCount);
for (int i = 0; i < topVWords.length; i++) {
    keyWords.put(topVWords[i].word, topVWords[i].freq);
}

```

```

//把内容分割成句子
ArrayList<SentenceScore> sentenceArray = getSentenceScores(content,pltem);
//计算每个句子的权重
for(SentenceScore sc:sentenceArray) {
    sc.score = 1;

    for (Entry<String,Integer> e:keyWords.entrySet()) {
        String word = e.getKey();
        if (sc.containsWord(word)) {
            sc.score = sc.score * e.getValue();
        }
    }
}

//取得权重最大的三个句子
int minSize = Math.min(sentenceArray.size(), 3);
Select.selectRandom(sentenceArray, sentenceArray.size(), minSize,0);
SentenceScore[] orderSen = new SentenceScore[minSize];
for(int i=0;i<minSize;++i){
    orderSen[i] = sentenceArray.get(i);
}
//按句子在原文中出现的顺序输出
Arrays.sort(orderSen, posCompare);//句子数组按在文档中的位置排序
String summary = "";
for (int i = 0; i<minSize; i++) {
    String curSen = orderSen[i].getSentence(content);
    summary = summary.concat(curSen);
}
return summary;

```

这只是一个简单的文本摘要程序。优化的方法有：

- 除了通过关键词，还可以通过提取基本要素(Basic Elements)来确定句子的重要程度。基本要素通过三元组<中心词，修饰，关系>来描述，其中中心词为该三元组的主要部分。
- 其中第一步，在提取关键词阶段，可以去掉停用词表，然后再统计关键词。也可以考虑利用同义词信息更准确的统计词频。
- 划分句子阶段，可以记录句子在段落中出现的位置，在段落开始或结束出现的句子更有

可能是关键句。同时可以考虑句型，陈述句比疑问句或感叹句更有可能是关键句。

首先定义句子类型：

```
public static enum SentenceType {
    declare, //陈述句
    question, //疑问句
    exclamation //感叹句
}
```

句子权重统计阶段考虑句型来打分。

```
//判断句子类型
if(sc.type == SentenceScore.SentenceType.question) {
    sc.score *= 0.1;
}
else if(sc.type == SentenceScore.SentenceType.exclamation) {
    sc.score *= 0.5;
}
```

为了使输出的摘要意义连续性更好，有必要划分段落。识别自然段和更大的意义段。自然段一般段首缩进两个或四个空格。

在对句子打分时，除了关键词，还可以查看事先编制好的线索词表。表示线索词的权值，有正面的和负面的两种。文摘正线索词就是类似“总而言之”、“总之”、“本文”、“综上所述”等词汇，含有这些词的句子权重有加分。文摘负线索词可以是“比如”，“例如”等。如果句中包括这些词权重就会降低。

为了减少文摘句之间的冗余度，可以通过句子相似度计算减少冗余句子。具体过程如下：

1. 首先将句子按其重要度从高到底排序；
2. 抽取重要度最高的句子 S_i ；
3. 选取候选句 S_i 后，调整剩下的每个待选句的重要度。待选句 S_j 的重要度按如下公式进行调整： $Score(S_j) = Score(S_j) - sim(S_i, S_j) * Score(S_i)$

其中 $sim(S_i, S_j)$ 是句子 S_i 和 S_j 的相似度。

4. 剩下的句子按重要度从高到底排序，选取重要度高的句子；
5. 重复 3、4 步，直至摘要足够长为止。

最后为了输出的摘要通顺，还需要处理句子间的关联关系。例如下面的关联句子：

“这个节目，需要的是接班人，而不是变革者。”换言之，一个节目的“心”的意义是大于“脸”的意义的；“换脸”未必就是“换心”；但《新闻联播》目前还只能接班性地“换脸”而不能变革性地“换心”。

处理关联句子的方法有三种：

1. 调整关联句的权重，使更重要的句子优先成为摘要句。
2. 调整关联句的权重，使关联的两个句子都成为或都不成为摘要句。
3. 输出摘要时，如果不能完整的保持相关联的句子，则删除句前的关联词。

句子间的关联通过关联性的词语来表示。处理关联句可以根据关联性词语的类型分别处理。表 5-3 列出了各种关联类型的处理方法。

表 5-3 关联词表

关联类型	关联词	处理方式
转折	虽然…但是…	对于这类偏正关系的，调整后部分的关键句的权重，保证其大于前面部分的权重。当只有一句是摘要句时，删除该句前的关联词。
因果	因为…所以…/因此	
递进	不但…而且… 尤其	
并列	一方面…另一方面…	
承接	接着 然后	对于这类并列关系，使关键句的权重都一样。找不到对应的关联句的删除该句子前面的关连词。
选择	或者…或者	
分述	首先…其次…	
总述	总而言之 综上所述 总之	这类可承前省略的，如果与前面的句子都是摘要句，则保持不变。否则，如果前面的句子不是摘要句则删除该句子前面的关连词。
等价	也就是说 即 换言之	
话题转移	另外	
对比	相对而言	
举例说明	比如 例如	这类可承前省略的，如果与前面的句子都是摘要句，则保持不变。否则删除后面的句子。

定义句子的关系类型：

```
public enum RelationType {
    conjunctive, //偏正
    juxtapose, //并列
    conclusion //承前省略
}
```

表示句子及它们之间的关系：

```
public class SentenceRelation {
    SentenceScore pre; //前一个句子
    SentenceScore sub; //下一个句子
    RelationType type; //关系类型

    public SentenceRelation(SentenceScore preSentence,
                           SentenceScore subSentence,
                           RelationType t) {
        pre = preSentence;
        sub = subSentence;
        type = t;
    }
}
```

5.9.3 基于篇章结构的自动摘要

对段落之间的内容语义关系进行分析，进而划分出文档的主题层次，得到文档的篇章结构。或者，可以建立句子级别的带权重的图，然后应用 PageRank 或者 HITS 算法。

5.9.4 Lucene 中的动态摘要

Lucene 扩展包中有一个实现自动摘要的包——HighLighter。HighLighter 返回一个或多个和搜索关键词最相关的段落。实现原理是：首先由分段器(Fragmenter)把文本分成多个段落，然后 QueryScorer 计算每个段落的分值。QueryScorer 只应该包含需要做高亮显示的 Term。

为了实现高亮显示，以 lucene-3.0.2 为例，除了依赖 lucene-core-3.0.2.jar 和 lucene-highlighter-3.0.2.jar 以外，还依赖 lucene-memory-3.0.2.jar。通过调用 getBestFragments 方法返回一个或多个和搜索关键词最相关的段落。实现高亮显示最简单的做法：

```
TokenStream tokenStream = analyzer.tokenStream("title", new StringReader(title));
String highLightText = highlighter.getBestFragment(tokenStream, title);
```

搜索时使用 analyzer 分析出搜索词会导致搜索速度变慢。形成索引的时候已经分过词

了，因此可以在索引时存储位置信息。可以通过 `IndexReader` 取出索引中保存的词的位置信息，通过词的位置信息来构造 `TokenStream`，这样就避免了搜索时再次分词导致的搜索速度降低。一般情况下，用户经常在使用关键词搜索，而索引只需要做一次就可以了，所以提升搜索速度很重要。如果 `IndexReader` 中的 `Token` 位置有重叠，为了把冗余的 `Token` 去掉，`TokenStream` 构造起来会麻烦一些。

对标题列高亮显示的实现代码如下：

```
SimpleHTMLFormatter simpleHTMLFormatter =
    new SimpleHTMLFormatter("<font color='red'>", "</font>");
Highlighter highlighter = new Highlighter(simpleHTMLFormatter, new QueryScorer(query));
highlighter.setTextFragmenter(new SimpleFragmenter(40));
for (int i = 0; i < hits.length; i++){
    Document hitDoc = isearcher.doc(hits[i].doc);
    String text = hitDoc.get("title");
    TermPositionVector tpv =
    (TermPositionVector)isearcher.getIndexReader().getTermFreqVector(hits[i].doc, "title");
    TokenStream tokenStream = TokenSources.getTokenStream(tpv);
    String highLightText = highlighter.getBestFragment(tokenStream, text);
    System.out.println(highLightText);
}
```

`FastVectorHighlighter` 是一个快速的高亮工具，相对于 `Highlighter` 它有三个好处：

1. `FastVectorHighlighter` 可以支持 `n` 元分词器分出来的列。
2. `FastVectorHighlighter` 可以输出不同颜色的高亮。
3. `FastVectorHighlighter` 可以对词组高亮。如检索 `lazy dog`，`FastVectorHighlighter` 返回结果是 `lazy dog`，而 `Highlighter` 则是 `dog`。

但是 `FastVectorHighlighter` 不支持所有的查询，例如 `WildcardQuery` 或 `SpanQuery` 等。

可以使用内存索引来测试 `FastVectorHighlighter` 高亮显示的效果。

```
FragListBuilder fragListBuilder = new SimpleFragListBuilder();
//创建多颜色标签 ScoreOrderFragmentsBuilder
FragmentsBuilder fragmentBuilder = new ScoreOrderFragmentsBuilder(
    BaseFragmentsBuilder.COLORED_PRE_TAGS,
    BaseFragmentsBuilder.COLORED_POST_TAGS);
FastVectorHighlighter highlighter = new FastVectorHighlighter(true, true,
    fragListBuilder, fragmentBuilder); // 创建 FastVectorHighlighter 实例
```

```
FieldQuery fieldQuery = highlighter.getFieldQuery(titleQuery); // 创建 FieldQuery
String highLightText = highlighter.getBestFragment(
    fieldQuery, isearcher.getIndexReader(),
    hits[i].doc, "title", 10000); // 高亮片断
```

FastVectorHighlighter 性能很好，但是 SimpleFragListBuilder 硬编码了 6 个字符的边界，导致匹配短文本时，左边的内容显示不全。修改后的 SimpleFragListBuilder 如下：

```
public class SimpleFragListBuilder implements FragListBuilder {
    public static final int MARGIN = 6;
    public static final int MIN_FRAG_CHAR_SIZE = MARGIN * 3;

    public FieldFragList createFieldFragList(FieldPhraseList fieldPhraseList,
        int fragCharSize) {
        if (fragCharSize < MIN_FRAG_CHAR_SIZE)
            throw new IllegalArgumentException("fragCharSize(" + fragCharSize
                + ") is too small. It must be " + MIN_FRAG_CHAR_SIZE
                + " or higher.");

        FieldFragList ffl = new FieldFragList(fragCharSize);
        List<WeightedPhraseInfo> wpil = new ArrayList<WeightedPhraseInfo>();
        Iterator<WeightedPhraseInfo> ite = fieldPhraseList.phraseList.iterator();
        WeightedPhraseInfo phraseInfo = null;
        int startOffset = 0;
        boolean taken = false;
        while (true) {
            if (!taken) {
                if (!ite.hasNext())
                    break;
                phraseInfo = ite.next();
            }
            taken = false;
            if (phraseInfo == null)
                break;
            // if the phrase violates the border of previous fragment, discard
            // it and try next phrase
            if (phraseInfo.getStartOffset() < startOffset)
                continue;
```

```

        wpil.clear();
        wpil.add(phraseInfo);
        int firstOffset = phraseInfo.getStartOffset();
        int st = phraseInfo.getStartOffset() - MARGIN < startOffset ? startOffset
            : phraseInfo.getStartOffset() - MARGIN;
        int en = st + fragCharSize;
        if (phraseInfo.getEndOffset() > en)
            en = phraseInfo.getEndOffset();

        int lastEndOffset = phraseInfo.getEndOffset();
        while (true) {
            if (ite.hasNext()) {
                phraseInfo = ite.next();
                taken = true;
                if (phraseInfo == null)
                    break;
            } else
                break;
            if (phraseInfo.getEndOffset() <= en){
                wpil.add(phraseInfo);
                lastEndOffset = phraseInfo.getEndOffset();
            } else
                break;
        }
        int matchLen = lastEndOffset - firstOffset;
        // now recalculate the start and end position to "center" the
        int newMargin = (fragCharSize - matchLen) / 2;
        st = firstOffset - newMargin;
        if (st < startOffset) {
            st = startOffset;
        }
        en = st + fragCharSize;
        startOffset = en;
        ffl.add(st, en, wpil);
    }
    return ffl;
}

```



```
}
```

5.10 文本分类

文本分类程序把一个未见过的文档分成已知类别中的一个或多个，例如把新闻分成国内新闻和国际新闻。利用文本分类技术可以对网页分类，也可以用于为用户提供个性化新闻或者垃圾邮件过滤。

把给定的文档归到两个类别中的一个叫做叫做两类分类，例如垃圾邮件过滤，就只需要确定“是”还是“不是”垃圾邮件。分到多个类别中的一个叫做多类分类，例如中图法分类目录把图书分成 22 个基本大类。

文本分类主要分为训练阶段和预测阶段。一个典型的文本分类程序框架如图 5-3 所示：

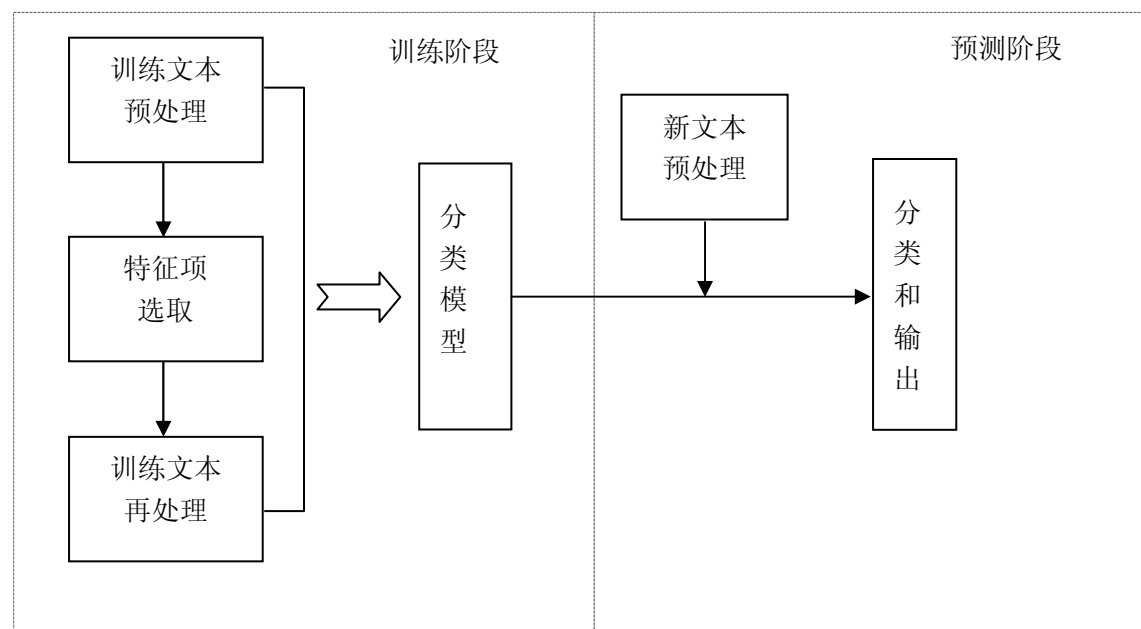


图 5-3 文本分类程序框架

首先准备好训练文本集，也就是一些已经分好类的文本。每个类别路径下包含属于该类别的一些文本文件。例如文本路径在“D:\train”，类别路径是：

```
D:\train\政治
D:\train\体育
D:\train\商业
D:\train\艺术
```

例如：“D:\train\体育”类别路径下包含属于“体育”类别的一些文本文件，每个文本文件叫做一个实例(instance)。训练文本文件可以手工整理一些或者从网络定向抓取网站中已经按栏目分好类的信息。

常见的分类方法有支持向量机(SVM)、K 个最近的邻居(KNN)和朴素贝叶斯(Naive Bayes)等。可以根据应用场景选择合适的文本分类方法。例如，支持向量机适合对长文本分类，朴素贝叶斯对短文本分类准确度也较高。

为了加快分类的执行速度，可以在训练阶段输出分类模型文件，这样在预测新文本类别阶段就不再需要直接访问训练文本集，只需要读取已经保存在分类模型文件中的信息。可以在预测之前，把分类模型文件预加载到内存中。这里采用单件模式加载分类模型文件。

```
private Classifier() { // 读取分类模型
    //...
}
private static Classifier categoryTrie = null;

public static Classifier getInstance() { //取得唯一的实例
    if (categoryTrie == null)
        categoryTrie = new Classifier();
    return categoryTrie;
}
```

依据标题、内容和商品在原有网站的类别把商品归类到新的分类，这些值都是字符串类型。使用参数个数不固定的方式定义分类方法：

```
//根据分类模型分类
public String getCategoryName(String... articals) {
    //获取参数
    for (String content : articals) {
        //根据 content 是否包含某些关键词分类
    }
    //...返回分类结果
}
```

这样可以按标题或者标题加内容等参数分类。

```
//加载已经训练好的分类模型
Classifier theClassifier = Classifier.getInstance();
//文本分类的内容
String content ="我要买把吉他， 希望是二手的， 价格 2000 元以下";
//根据内容分类
String catName = theClassifier.getCategoryName(content);
System.out.println("类别名称:"+catName);
```

交叉验证(Cross Validation)是用来验证分类器的性能的一种统计分析方法。基本思想是

把在某种意义下将原始数据集进行分组，一部分做为训练集，另一部分做为验证集。首先用训练集对分类器进行训练，再利用验证集来测试训练得到的模型，以此来做为评价分类器的性能指标。

5.10.1 特征提取

待分类的文本往往包括很多单词，很多单词对分类没有太大的贡献，所以需要提取特征词。可以按词性过滤，只选择某些词性作为分类特征，比如说，只选择名词和动词作为分类特征词。文本分类的精度随分类特征词的个数持续提高。一般至少可以选 2000 个分类特征词。

分类特征并不一定就是一个词。例如，可以通过检查标题和签名把文本分类成是否是信件内容。可以看标题是否包含“来自**”和“致**”地址，内容结束处是否包含日期和问候用语等。这样的特征集合仅适用于信件类别。

特征选择的常用方法还有：CHI 方法和信息增益(Information Gain)方法等。首先介绍特征选择的 CHI 方法。

利用 CHI 方法来进行特征抽取是基于如下假设：在指定类别文本中出现频率高的词条与在其他类别文本中出现频率比较高的词条，对判定文档是否属于该类别都是很有帮助的。

CHI 方法衡量单词 *term* 和类别 *class* 之间的依赖关系。如果 *term* 和 *class* 是互相独立的，则该值接近于 0。一个单词的 CHI 统计通过表 5-4 计算：

表 5-4 CHI 统计变量定义表

	属于 <i>class</i> 类	不属于 <i>class</i> 类	合计
包含单词 <i>term</i>	<i>a</i>	<i>b</i>	<i>a+b</i>
不含单词 <i>term</i>	<i>c</i>	<i>d</i>	<i>c+d</i>
合计	<i>a+c</i>	<i>b+d</i>	<i>a+b+c+d=n</i>

其中，*a* 表示属于类别 *class* 的文档集合中出现单词 *term* 的文档数；*b* 表示不属于类别 *class* 的文档集合中出现单词 *term* 的文档数；*c* 表示属于类别 *class* 的文档集合中没有出现单词 *term* 的文档数；*d* 表示不属于类别 *class* 的文档集合中没有出现单词 *term* 的文档数；*n* 代表文档总数。

表 5-4 中的单词 *term* 的 CHI 统计公式如下：

$$\text{chi_statistics}(\text{term}, \text{class}) = \frac{n(a-d)(b-c)^2}{(a+c)(b+d)(a+b)(c+d)}$$

类别 *class* 越依赖单词 *term*，则 CHI 统计值越大。

表 5-4 也叫做相依表 (contingency table)。开源自然语言处理项目 MinorThird(<http://minorthird.sourceforge.net/>)中 ContingencyTable 类的 CHI 统计实现代码：

```
//取对数避免溢出
public double getChiSquared(){
    double n = Math.log(total());
    double num = 2*Math.log(Math.abs((a*d) - (b*c)));
    double den = Math.log(a+b)+Math.log(a+c)+Math.log(c+d)+Math.log(b+d);
    double tmp = n+num-den;
    return Math.exp(tmp);
}
```

计算每个特征对应的 CHI 值的实现代码：

```
for (Iterator<Feature> i=index.featureIterator(); i.hasNext(); ) { //遍历特征集合
    Feature f = i.next();
    int a = index.size(f,ExampleSchema.POS_CLASS_NAME); //正类中包含特征的文档数
    int b = index.size(f,ExampleSchema.NEG_CLASS_NAME); //负类中包含特征的文档数
    int c = totalPos - a; //正类中不包含特征的文档数
    int d = totalNeg - b; //负类中不包含特征的文档数

    ContingencyTable ct = new ContingencyTable(a,b,c,d);
    double chiScore = ct.getChiSquared(); //计算特征的 CHI 值
    filter.addFeature( chiScore,f );
}
```

如果这里的 $a+c$ 或者 $b+d$ 或者 $a+b$ 或者 $c+d$ 中的任意一个值为 0，则会导致除以零溢出的错误。由于数据稀疏导致了这样的问题，可以采用平滑算法来解决。

对所有的候选特征词，按上面得到的特征区分度排序，如果候选特征词的个数大于 5000，则选取前 5000。否则选取所有特征词。

使用 ChiSquareTransformLearner 测试特征提取：

```
Dataset dataset = CnSampleDatasets.sampleData("toy",false);
System.out.println( "old data:\n" + dataset );
ChiSquareTransformLearner learner = new ChiSquareTransformLearner();
ChiSquareInstanceTransform filter =
    (ChiSquareInstanceTransform)learner.batchTrain( dataset );
filter.setNumberOfFeatures(10);
```

```
dataset = filter.transform( dataset );
System.out.println( "new data:\n" + dataset );
```

信息增益(Information Gain)是广泛使用的特征选择方法。在信息论中，信息增益的概念是：某个特征的值对分类结果的确定程度增加了多少。

信息增益的计算方法是：把文档集合 D 看成一个符合某种概率分布的信息源，依靠文档集合的信息熵和文档中词语的条件熵之间信息量的增益关系确定该词语在文本分类中所能提供的信息量。

词语 w 的信息量的计算公式为：

$$IG(w)=H(D)-H(D|w)$$

$$=-\sum_{d_i \in D} P(d_i) \times \log_2 P(d_i) + \sum_{w \in \{0,1\}} P(w) \sum_{d_i \in D} P(d_i | w) \times \log_2 P(d_i | w)$$

根据特征在每个类别的出现次数分布计算一个特征的熵：

```
//输入参数 p 是特征的出现次数分布，tot 是特征出现的总次数
public double Entropy(double[] p, double tot){
    double entropy = 0.0;
    for (int i=0; i<p.length; i++) {
        if (p[i]>0.0) { entropy += -p[i]/tot *Math.log(p[i]/tot) /Math.log(2.0); }
    }
    return entropy;
}
```

计算所有特征的熵：

```
double[] classCnt = new double[ N ];
double totalCnt = 0.0;
for (int c=0; c<N; c++){//循环遍历所有的类别
    classCnt[c] = (double)index.size(schema.getClassName(c));//每类的文档数
    totalCnt += classCnt[c];//总文档数
}
double totalEntropy = Entropy(classCnt,totalCnt);//训练文档的总熵值

for (Iterator<Feature> i=index.featureIterator(); i.hasNext(); ) {
    Feature f = i.next();
    double[] featureCntWithF = new double[ N ];//出现特征的文档在不同类别中的分布
    double[] featureCntWithoutF = new double[ N ]; //不出现特征的文档在不同类别的分布
```

```

double totalCntWithF = 0.0;
double totalCntWithoutF = 0.0;

for (int c=0; c<N; c++) {
    featureCntWithF[c] = (double)index.size(f,schema.getClassName(c));
    featureCntWithoutF[c] = classCnt[c] - featureCntWithF[c];
    totalCntWithF += featureCntWithF[c];
    totalCntWithoutF += featureCntWithoutF[c];
}

double entropyWithF = Entropy(featureCntWithF,totalCntWithF); //出现特征的熵
//不出现特征的熵
double entropyWithoutF = Entropy(featureCntWithoutF,totalCntWithoutF);

double wf = totalCntWithF /totalCnt; //出现词的概率
//特征的信息增益
double infoGain = totalEntropy -wf*entropyWithF -(1.0-wf)*entropyWithoutF;
igValues.add( new IGPair(infoGain,f) );
}

```

5.10.2 关键词加权法

为了理解机器学习的算法如何对文本分类，首先看一下人是如何对事物分类的。为了判断食物是否健康食品，可参考食品中的饱和脂肪、胆固醇、糖和钠的含量。例如把食品分类成“健康食品”和“不健康食品”。如果这些值超过一个阈值就认为该食品是“不健康的”，否则是“健康的”。首先找出一些重要的特征，然后从每个待分类的项目中寻找特征，从抽取出的特征中组合证据(combine evidence)，最后根据组合证据按照某种决策机制对项目分类。

在食品分类的例子中，特征是饱和脂肪、胆固醇、糖和钠的含量。可以通过阅读打印在食品包装上的营养成分表来取得待分类食品的特征对应的值。

为了量化食物的健康程度(记做 H)，有很多方法来组合证据，最简单的方法是按权重求和。

$H(\text{食物}) = \quad (\quad) \quad (\quad) \quad (\quad) \quad (\quad)$

ERROR: limitcheck
OFFENDING COMMAND: string

STACK:

66038
33018
32512
33019