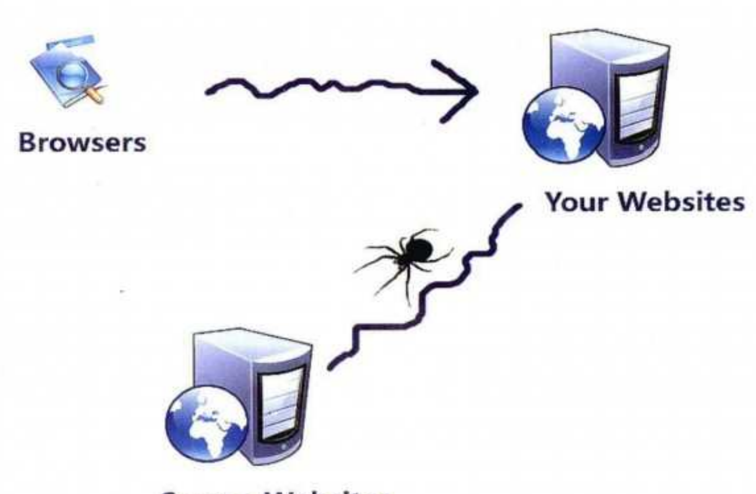
国内**第一本**专门讲解**网络爬虫**开发的书籍
介绍如何应用**云计算**架构开发分布式爬虫




罗 刚 王振东 编著

# 自己动手写网络爬虫

- 猎兔搜索工程师多年项目经验总结
- 深入介绍Web数据挖掘实现过程
- 光盘中提供了高效的代码解决方案
- 案例均使用流行的Java语言编写

清华大学出版社

自己动手写

# 网络爬虫

本书介绍了网络爬虫开发中的关键问题与Java实现，主要包括从互联网获取信息、提取信息和对Web信息挖掘等内容，本书在介绍基本原理的同时注重通过具体代码实现来深入理解。书中部分代码甚至可以直接应用。本书适用于有Java程序设计基础的开发人员，同时也可以作为计算机相关专业本科生或研究生的参考教材。

ISBN 978-7-302-23647-4

定价：43.00元（附光盘1张）

# 前　言

当你在网上冲浪时，你是否知道还有一类特殊的网络用户也在互联网上默默地工作着，它们就是网络爬虫。这些网络爬虫按照设计者预定的方式，在网络中穿梭，同时自动收集有用的信息，进行分类和整理，将整理结果提供给用户，以方便用户查找他们感兴趣的内容。由于网络爬虫的实用性，引起了很多程序员，特别是Web程序员的兴趣。

但是大多数网络爬虫的开发原理与技巧在专业的公司内部都秘而不宣，至今仍然缺少理论与实践相结合的专门介绍网络爬虫的书籍。本书将弥补这个问题，尝试理论与实践相结合，深入透彻地讲解网络爬虫的原理，并且辅以相关代码作为参考。本书相关的代码在附带光盘中可以找到。

本书的两位主要作者在搜索引擎领域都有丰富的理论和实践经验。同时，还有多个程序员帮忙开发或编写了代码实现，例如Java实现异步I/O或对PDF文件的处理等。由于作者的日常工作繁忙，做得不够的地方敬请谅解。

作者罗刚在参加编写本书之前，还独立撰写过《自己动手写搜索引擎》一书，但存在讲解不够细致、知识点不够深入等问题。此次与王振东合著本书，相对于上一本书而言，对读者反馈有更高的预期。因为作者相信如下的假设：如果能够与更多的人更好地合作，事情往往能做得更好。

本书从基本的爬虫原理开始讲解，通过介绍优先级队列、宽度优先搜索等内容引领读者入门；之后根据当前风起云涌的云计算热潮，重点讲述了云计算的相关内容及其在爬虫中的应用，以及带偏好的爬虫、信息抽取、链接分析等内容；为了能够让读者更深入地了解爬虫，本书在最后两章还介绍了有关爬虫的数据挖掘的内容。

由于搜索引擎相关领域也正在快速发展中，而且由于篇幅的限制，有些不成熟的内容，没有能够在本书体现，例如有关“暗网”的内容。随着技术的不断发展，我们将在今后的版本中加入这些内容。

本书适合需要具体实现搜索引擎的程序员使用，对于信息检索等相关研究人员也有一定的参考价值，同时猎兔搜索技术团队也已经开发出以本书为基础的专门培训课程和商业软件。目前搜索引擎开发人员仍然很稀缺，作者真诚地希望通过本书把读者带入搜索引擎开发的大门并认识更多的朋友。

感谢开源软件和我们的家人、关心我们的老师和朋友、创业伙伴以及选择猎兔搜索软件的客户多年来的支持。

编　者

# 目　录

## 第1篇　自己动手抓取数据


第1章　全面剖析网络爬虫 3

1.1 抓取网页 4
- 1.1.1 深入理解URL 4
- 1.1.2 通过指定的URL抓取网页内容 6
- 1.1.3 Java网页抓取示例 8
- 1.1.4 处理HTTP状态码 10

1.2 宽度优先爬虫和带偏好的爬虫 11
- 1.2.1 图的宽度优先遍历 12
- 1.2.2 宽度优先遍历互联网 13
- 1.2.3 Java宽度优先爬虫示例 15
- 1.2.4 带偏好的爬虫 22
- 1.2.5 Java带偏好的爬虫示例 23

1.3 设计爬虫队列 24
- 1.3.1 爬虫队列 24
- 1.3.2 使用Berkeley DB构建爬虫队列 29
- 1.3.3 使用Berkeley DB 构建爬虫队列示例 30
- 1.3.4 使用布隆过滤器构建Visited表 36
- 1.3.5 详解Heritrix爬虫队列 39

1.4 设计爬虫架构 46
- 1.4.1 爬虫架构 46
- 1.4.2 设计并行爬虫架构 47
- 1.4.3 详解Heritrix爬虫架构 52

1.5 使用多线程技术提升爬虫性能 55
- 1.5.1 详解Java多线程 55
- 1.5.2 爬虫中的多线程 59
- 1.5.3 一个简单的多线程爬虫实现 60
- 1.5.4 详解Heritrix多线程结构 61

1.6 本章小结 64

第2章　分布式爬虫 69

2.1 设计分布式爬虫 70
- 2.1.1 分布式与云计算 70
- 2.1.2 分布式与云计算技术在爬虫中的应用——浅析Google的云计算架构 71

2.2 分布式存储 72
- 2.2.1 从Ralation_DB到key/value存储 72
- 2.2.2 Consistent Hash算法 74
- 2.2.3 Consistent Hash代码实现 79

2.3 Google的成功之道——GFS 80
- 2.3.1 GFS详解 80
- 2.3.2 开源GFS——HDFS 84

2.4 Google网页存储秘诀——BigTable 88
- 2.4.1 详解BigTable 88
- 2.4.2 开源BigTable——HBase 93

2.5 Google的成功之道——MapReduce算法 98
- 2.5.1 详解MapReduce算法 100
- 2.5.2 MapReduce容错处理 101
- 2.5.3 MapReduce实现架构 102
- 2.5.4 Hadoop中的MapReduce简介 104
- 2.5.5 wordCount例子的实现 105

2.6 Nutch中的分布式 109
- 2.6.1 Nutch爬虫详解 109
- 2.6.2 Nutch中的分布式 116

2.7 本章小结 118

第3章 爬虫的“方方面面” ....121
3.1 爬虫中的“黑洞” ....122
3.2 限定爬虫和主题爬虫 ....122
3.2.1 理解主题爬虫 ....122
3.2.2 Java主题爬虫 ....128
3.2.3 理解限定爬虫 ....130
3.2.4 Java限定爬虫示例 ....136
3.3 有“道德”的爬虫 ....152
3.4 本章小结 ....155

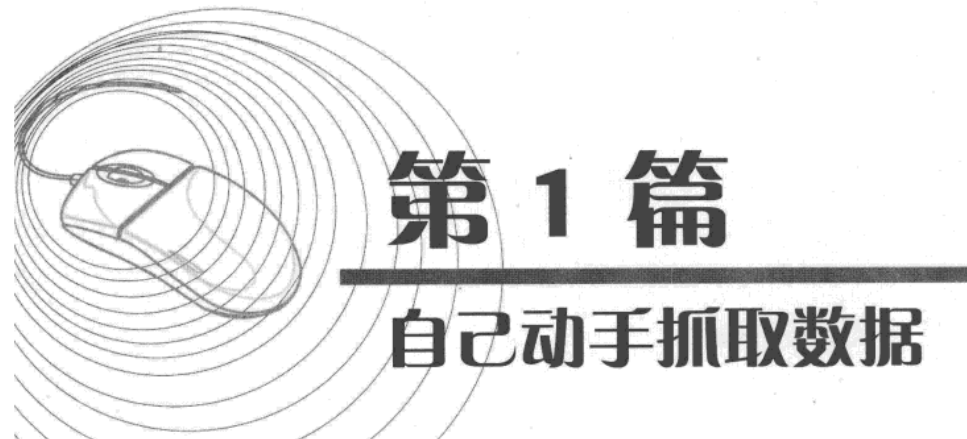
## 第2篇 自己动手抽取Web内容

第4章 “处理”HTML页面 ....159
4.1 征服正则表达式 ....160
4.1.1 学习正则表达式 ....160
4.1.2 Java正则表达式 ....164
4.2 抽取HTML正文 ....169
4.2.1 了解HtmlParser ....169
4.2.2 使用正则表达式抽取示例 ....172
4.3 抽取正文 ....179
4.4 从JavaScript中抽取信息 ....194
4.4.1 JavaScript抽取方法 ....195
4.4.2 JavaScript抽取示例 ....197
4.5 本章小结 ....199
第5章 非HTML正文抽取 ....201
5.1 抽取PDF文件 ....202
5.1.1 学习PDFBox ....202
5.1.2 使用PDFBox抽取示例 ....206
5.1.3 提取PDF文件标题 ....207
5.1.4 处理PDF格式的公文 ....208
5.2 抽取Office文档 ....212
5.2.1 学习POI ....212
5.2.2 使用POI抽取Word示例 ....213
5.2.3 使用POI抽取PPT 示例 ....215
5.2.4 使用POI抽取Excel示例 ....215
5.3 抽取RTF ....218
5.3.1 开源RTF文件解析器 ....219
5.3.2 实现一个RTF文件解析器 ....219
5.3.3 解析RTF示例 ....223
5.4 本章小结 ....229
第6章 多媒体抽取 ....231
6.1 抽取视频 ....232
6.1.1 抽取视频关键帧 ....232
6.1.2 Java视频处理框架 ....233
6.1.3 Java视频抽取示例 ....237
6.2 音频抽取 ....249
6.2.1 抽取音频 ....249
6.2.2 学习Java音频抽取技术 ....253
6.3 本章小结 ....256
第7章 去掉网页中的“噪声” ....257
7.1 “噪声”对网页的影响 ....258
7.2 利用“统计学”消除“噪声” ....259
7.2.1 网站风格树 ....262
7.2.2 “统计学去噪”Java实现 ....270
7.3 利用“视觉”消除“噪声” ....274
7.3.1 “视觉”与“噪声” ....274
7.3.2 “视觉去噪”Java实现 ....275
7.4 本章小结 ....279

## 第3篇 自己动手挖掘Web数据

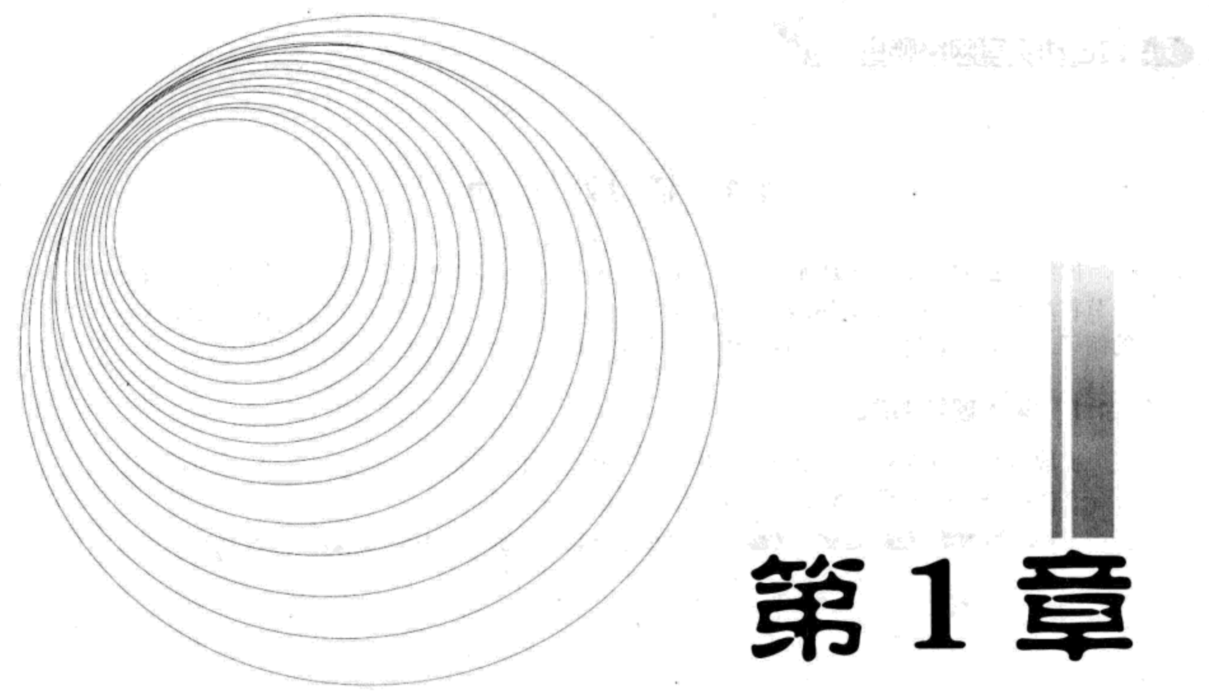
第8章 分析Web图 ....283
8.1 存储Web“图” ....284
8.2 利用Web“图”分析链接 ....293
8.3 Google的秘密——PageRank ....293
8.3.1 深入理解PageRank算法 ....293
8.3.2 PageRank算法的Java实现 ....297
8.3.3 应用PageRank进行链接分析 ....300

8.4 PageRank 的兄弟 HITS ...................... 301
8.4.1 深入理解 HITS 算法 .............. 301
8.4.2 HITS 算法的 Java 实现 .......... 302
8.4.3 应用 HITS 进行链接分析 ...... 313
8.5 PageRank 与 HITS 的比较 ................. 314
8.6 本章小结 ............................................ 315
**第 9 章 去掉重复的"文档" .................. 317**
9.1 何为"重复"的文档 ......................... 318
9.2 去除"重复"文档——排重 ............. 318
9.3 利用"语义指纹"排重 ..................... 318
9.3.1 理解"语义指纹" ................. 320
9.3.2 "语义指纹"排重的 Java 实现 ................................ 321
9.4 SimHash 排重 ..................................... 321
9.4.1 理解 SimHash ......................... 322
9.4.2 SimHash 排重的 Java 实现 .... 323
9.5 分布式文档排重 ................................ 330
9.6 本章小结 ............................................ 331
**第 10 章 分类与聚类的应用 ..................... 333**
10.1 网页分类 .......................................... 334
10.1.1 收集语料库 ........................... 334
10.1.2 选取网页的"特征" ............ 335
10.1.3 使用支持向量机进行网页分类 ................................ 338
10.1.4 利用 URL 地址进行网页分类 ................................ 340
10.1.5 使用 AdaBoost 进行网页分类 ................................ 340
10.2 网页聚类 .......................................... 343
10.2.1 深入理解 DBScan 算法 ........ 343
10.2.2 使用 DBScan 算法聚类实例 ............................... 344
10.3 本章小结 .......................................... 346



# 第1篇

## 自己动手抓取数据

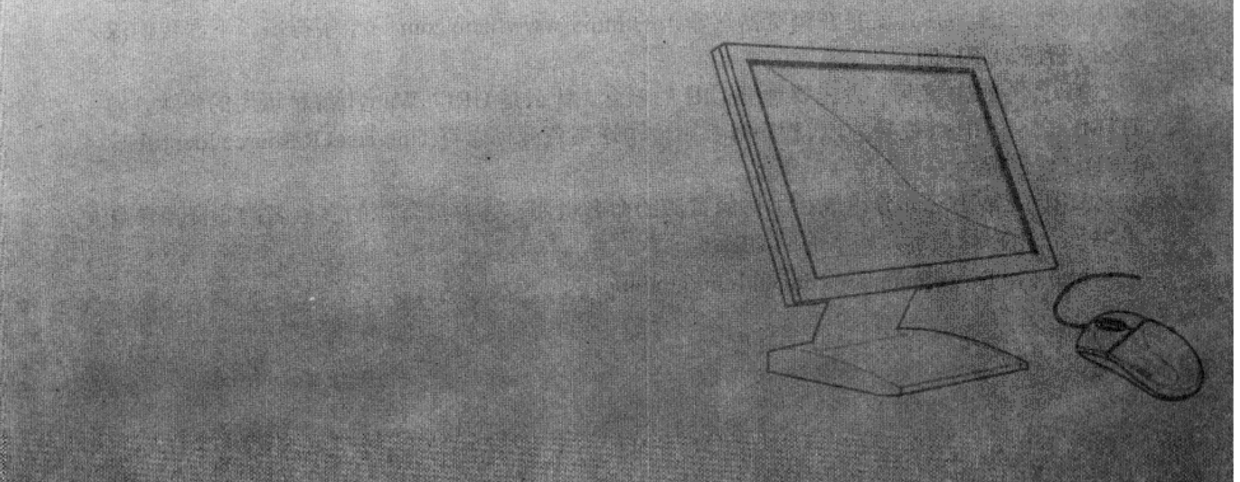


# 第1章

## 全面剖析网络爬虫

你知道百度、Google 是如何获取数以亿计的网页并且实时更新的吗？你知道在搜索引擎领域人们常说的 Spider 是什么吗？本章将全面介绍网络爬虫的方方面面。读完之后，你将完全有能力自己写一个网络爬虫，随意抓取互联网上任何感兴趣的东西。

既然百度、Google 这些搜索引擎巨头已经帮我们抓取了互联网上的大部分信息，为什么还要自己写爬虫呢？因为深入整合信息的需求是广泛存在的。在企业中，爬虫抓取下来的信息可以作为数据仓库多维展现的数据源，也可以作为数据挖掘的来源。甚至有人为了炒股，专门抓取股票信息。既然从美国中情局到普通老百姓都需要，那还等什么，让我们快开始吧。



# 1.1 抓取网页

网络爬虫的基本操作是抓取网页。那么如何才能随心所欲地获得自己想要的页面？这一节将从 URL 开始讲起，然后告诉大家如何抓取网页，并给出一个使用 Java 语言抓取网页的例子。最后，要讲一讲抓取过程中的一个重要问题：如何处理 HTTP 状态码。

## 1.1.1 深入理解 URL

抓取网页的过程其实和读者平时使用 IE 浏览器浏览网页的道理是一样的。比如，你打开一个浏览器，输入猎兔搜索网站的地址，如图 1.1 所示。

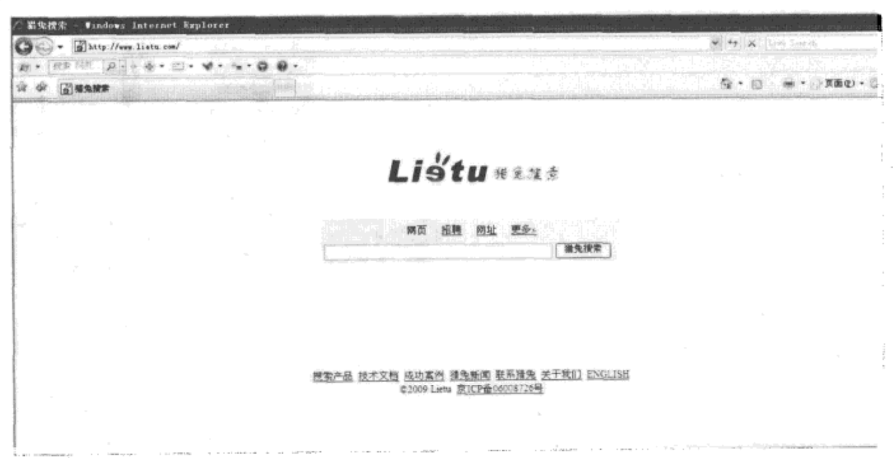



图 1.1 使用浏览器浏览网页

“打开”网页的过程其实就是浏览器作为一个浏览的“客户端”，向服务器端发送了一次请求，把服务器端的文件“抓”到本地，再进行解释、展现。更进一步，可以通过浏览器端查看“抓取”过来的文件源代码。选择“查看”|“源文件”命令，就会出现从服务器上“抓取”下来的文件的源代码，如图 1.2 所示。

在上面的例子中，我们在浏览器的地址栏中输入的字符串叫做 URL。那么，什么是 URL 呢？直观地讲，URL 就是在浏览器端输入的 http://www.lietu.com 这个字符串。下面我们深入介绍有关 URL 的知识。

在理解 URL 之前，首先要理解 URI 的概念。什么是 URI？Web 上每种可用的资源，如 HTML 文档、图像、视频片段、程序等都由一个通用资源标志符(Universal Resource Identifier，URI)进行定位。

URI 通常由三部分组成：①访问资源的命名机制；②存放资源的主机名；③资源自身的名称，由路径表示。如下面的 URI：

http://www.webmonkey.com.cn/html/html40/

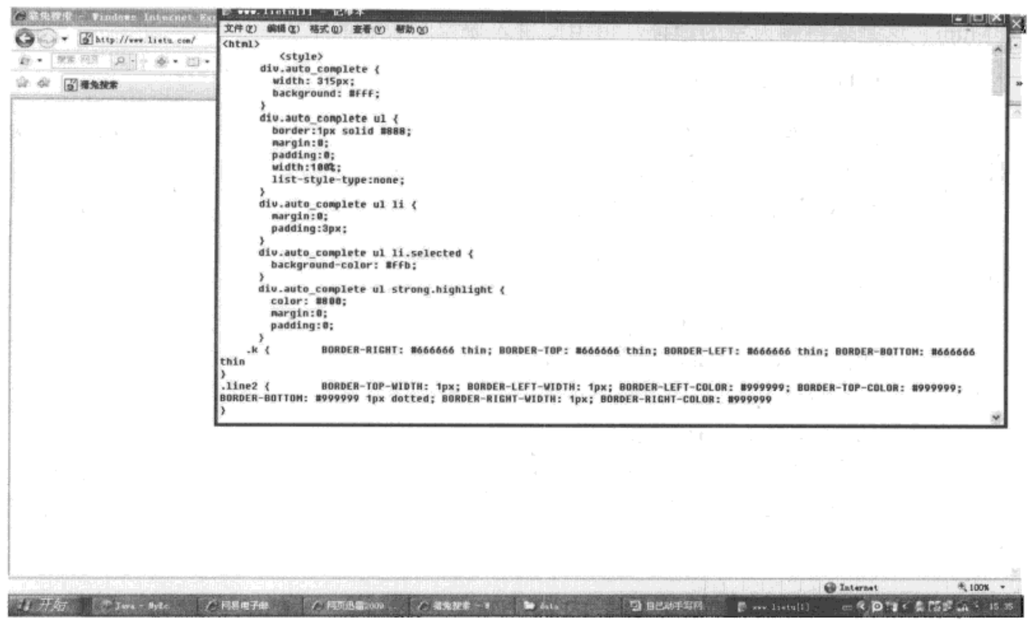



图 1.2　浏览器端源代码

我们可以这样解释它：这是一个可以通过 HTTP 协议访问的资源，位于主机 www.webmonkey.com.cn 上，通过路径“/html/html40”访问。

URL 是 URI 的一个子集。它是 Uniform Resource Locator 的缩写，译为“统一资源定位符”。通俗地说，URL 是 Internet 上描述信息资源的字符串，主要用在各种 WWW 客户程序和服务器程序上，特别是著名的 Mosaic。采用 URL 可以用一种统一的格式来描述各种信息资源，包括文件、服务器的地址和目录等。URL 的格式由三部分组成：

- 第一部分是协议(或称为服务方式)。
- 第二部分是存有该资源的主机 IP 地址(有时也包括端口号)。
- 第三部分是主机资源的具体地址，如目录和文件名等。

第一部分和第二部分用“://”符号隔开，第二部分和第三部分用“/”符号隔开。第一部分和第二部分是不可缺少的，第三部分有时可以省略。

根据 URL 的定义，我们给出了常用的两种 URL 协议的例子，供大家参考。

### 1．HTTP 协议的 URL 示例

使用超级文本传输协议 HTTP，提供超级文本信息服务的资源。

例：http://www.peopledaily.com.cn/channel/welcome.htm

其计算机域名为 www.peopledaily.com.cn。超级文本文件(文件类型为.html)是在目录/channel 下的 welcome.htm。这是中国人民日报的一台计算机。

例：http://www.rol.cn.net/talk/talk1.htm

其计算机域名为 www.rol.cn.net。超级文本文件(文件类型为.html)是在目录/talk 下的

talk1.htm。这是瑞得聊天室的地址，可由此进入瑞得聊天室的第 1 室。

**2．文件的 URL**

用 URL 表示文件时，服务器方式用 file 表示，后面要有主机 IP 地址、文件的存取路径(即目录)和文件名等信息。有时可以省略目录和文件名，但“/”符号不能省略。

例：file://ftp.yoyodyne.com/pub/files/foobar.txt

上面这个 URL 代表存放在主机 ftp.yoyodyne.com 上的 pub/files/目录下的一个文件，文件名是 foobar.txt。

例：file://ftp.yoyodyne.com/pub

代表主机 ftp.yoyodyne.com 上的目录/pub。

例：file://ftp.yoyodyne.com/

代表主机 ftp.yoyodyne.com 的根目录。

爬虫最主要的处理对象就是 URL，它根据 URL 地址取得所需要的文件内容，然后对它进行进一步的处理。因此，准确地理解 URL 对理解网络爬虫至关重要。从下一节开始，我们将详细地讲述如何根据 URL 地址来获得网页内容。

### 1.1.2 通过指定的 URL 抓取网页内容

上一节详细介绍了 URL 的构成，这一节主要阐述如何根据给定的 URL 来抓取网页。

所谓网页抓取，就是把 URL 地址中指定的网络资源从网络流中读取出来，保存到本地。类似于使用程序模拟 IE 浏览器的功能，把 URL 作为 HTTP 请求的内容发送到服务器端，然后读取服务器端的响应资源。

Java 语言是为网络而生的编程语言，它把网络资源看成是一种文件，它对网络资源的访问和对本地文件的访问一样方便。它把请求和响应封装为流。因此我们可以根据相应内容，获得响应流，之后从流中按字节读取数据。例如，java.net.URL 类可以对相应的 Web 服务器发出请求并且获得响应文档。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);
```

接着，可以通过获得的 URL 对象来取得网络流，进而像操作本地文件一样来操作网络资源：

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

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 IE 客户端的工作，代码量非常大。需要处理 HTTP 返回的状态码，设置 HTTP 代理，处理 HTTPS 协议等工作。为了便于应用程序的开发，实际开发时常常使用 Apache 的 HTTP 客户端开源项目——HttpClient。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 IE 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
HttpClient httpclient=new HttpClient();
```

```
//创建一个 get 方法，类似于在浏览器地址栏中输入一个地址
GetMethod getMethod=new GetMethod("http://www.blablabla.com");

//回车，获得响应状态码
int statusCode=httpclient.executeMethod(getMethod);

//查看命中情况，可以获得的东西还有很多，比如 head、cookies 等
System.out.println("response=" + getMethod.getResponseBodyAsString());

//释放
getMethod.releaseConnection();
```

上面的示例代码是使用 HttpClient 进行请求与响应的例子。第一行表示创建一个客户端，相当于打开浏览器。第二行使用 get 方式对 http://www.blablabla.com 进行请求。第三行执行请求，获取响应状态。第四行的 getMethod.getResponseBodyAsString()方法能够以字符串方式获取返回的内容。这也是网页抓取所需要的内容。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，通常情况下，采用 post 方法进行 HTTP 请求，HttpClient 包对 post 方法也有很好的支持。例如：

```
//得到 post 方法
PostMethod PostMethod = new PostMethod("http://www.saybot.com/postme");

//使用数组来传递参数
NameValuePair[] postData = new NameValuePair[2];

//设置参数
postData[0] = new NameValuePair("武器", "枪");
postData[1] = new NameValuePair("什么枪", "神枪");
postMethod.addParameters(postData);

//回车，获得响应状态码
int statusCode=httpclient.executeMethod(getMethod);

//查看命中情况，可以获得的东西还有很多，比如 head、cookies 等
System.out.println("response=" + getMethod.getResponseBodyAsString());

//释放
getMethod.releaseConnection();
```

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 方法可以使用 NameValuePair 来设置参数，因此可以设置“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。

有时，我们执行爬虫程序的机器不能直接访问 Web 资源，而是需要通过 HTTP 代理服务器去访问，HttpClient 对代理服务器也有很好的支持。如：

```
//创建 HttpClient 相当于打开一个代理
HttpClient httpClient=new HttpClient();

//设置代理服务器的 IP 地址和端口
httpClient.getHostConfiguration().setProxy("192.168.0.1", 9527);

//告诉 httpClient，使用抢先认证，否则你会收到“你没有资格”的恶果
httpClient.getParams().setAuthenticationPreemptive(true);

//MyProxyCredentialsProvder 返回代理的 credential(username/password)
httpClient.getParams().setParameter(CredentialsProvider.PROVIDER,
new MyProxyCredentialsProvider());

//设置代理服务器的用户名和密码
httpClient.getState().setProxyCredentials(new AuthScope("192.168.0.1",
  AuthScope.ANY_PORT, AuthScope.ANY_REALM),
new UsernamePasswordCredentials("username","password"));
```

上面的例子详细解释了如何使用 HttpClient 设置代理服务器。如果你所在的局域网访问 Web 资源需要代理服务器的话，你可以参照上面的代码设置。

这一节，我们介绍了使用 HttpClient 抓取网页的内容，之后，我们将给出一个详细的例子来说明如何获取网页。

### 1.1.3 Java 网页抓取示例

在这一节中，我们根据之前讲过的内容，写一个实际的网页抓取的例子。这个例子把上一节讲的内容做了一定的总结，代码如下：

```
public class RetrivePage {
    private static HttpClient httpClient = new HttpClient();
    // 设置代理服务器
    static {
        // 设置代理服务器的 IP 地址和端口
        httpClient.getHostConfiguration().setProxy("172.17.18.84", 8080);
    }
    public static boolean downloadPage(String path) throws HttpException,
            IOException {
        InputStream input = null;
        OutputStream output = null;
        // 得到 post 方法
        PostMethod postMethod = new PostMethod(path);
        //设置 post 方法的参数
          NameValuePair[] postData = new NameValuePair[2]; postData[0] = new
          NameValuePair("name","lietu"); postData[1] = new
```

```
            NameValuePair("password","******");
            postMethod.addParameters(postData);
          // 执行，返回状态码
        int statusCode = httpClient.executeMethod(postMethod);
          // 针对状态码进行处理 (简单起见，只处理返回值为 200 的状态码)
        if (statusCode == HttpStatus.SC_OK) {
             input = postMethod.getResponseBodyAsStream();
             //得到文件名
             String filename = path.substring(path.lastIndexOf('/')+1);
             //获得文件输出流
             output = new FileOutputStream(filename);

             //输出到文件
             int tempByte = -1;
             while((tempByte=input.read())>0){
                 output.write(tempByte);
             }
             //关闭输入输出流
             if(input!=null){
                 input.close();
             }
             if(output!=null){
                 output.close();
             }
             return true;
        }
        return false;
    }

    /**
     * 测试代码
     */
    public static void main(String[] args) {
        // 抓取 lietu 首页，输出
        try {
            RetrivePage.downloadPage("http://www.lietu.com/");
        } catch (HttpException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

上面的例子是抓取猎兔搜索主页的示例。它是一个比较简单的网页抓取示例，由于互联网的复杂性，真正的网页抓取程序会考虑非常多的问题。比如，资源名的问题，资源类型的问题，状态码的问题。而其中最重要的就是针对各种返回的状态码的处理。下一节将重点介绍处理状态码的问题。

### 1.1.4 处理 HTTP 状态码

上一节介绍 HttpClient 访问 Web 资源的时候，涉及 HTTP 状态码。比如下面这条语句：

```
int statusCode=httpClient.executeMethod(getMethod);//回车，获得响应状态码
```

HTTP 状态码表示 HTTP 协议所返回的响应的状态。比如客户端向服务器发送请求，如果成功地获得请求的资源，则返回的状态码为 200，表示响应成功。如果请求的资源不存在，则通常返回 404 错误。

HTTP 状态码通常分为 5 种类型，分别以 1～5 五个数字开头，由 3 位整数组成。1XX 通常用作实验用途。这一节主要介绍 2XX、3XX、4XX、5XX 等常用的几种状态码，如表 1.1 所示。

表 1.1　HTTP 常用状态码

| 状态代码 | 代码描述 | 处理方式 |
|---|---|---|
| 200 | 请求成功 | 获得响应的内容，进行处理 |
| 201 | 请求完成，结果是创建了新资源。新创建资源的 URI 可在响应的实体中得到 | 爬虫中不会遇到 |
| 202 | 请求被接受，但处理尚未完成 | 阻塞等待 |
| 204 | 服务器端已经实现了请求，但是没有返回新的信息。如果客户是用户代理，则无须为此更新自身的文档视图 | 丢弃 |
| 300 | 该状态码不被 HTTP/1.0 的应用程序直接使用，只是作为 3XX 类型回应的默认解释。存在多个可用的被请求资源 | 若程序中能够处理，则进行进一步处理，如果程序中不能处理，则丢弃 |
| 301 | 请求到的资源都会分配一个永久的 URL，这样就可以在将来通过该 URL 来访问此资源 | 重定向到分配的 URL |
| 302 | 请求到的资源在一个不同的 URL 处临时保存 | 重定向到临时的 URL |
| 304 | 请求的资源未更新 | 丢弃 |
| 400 | 非法请求 | 丢弃 |
| 401 | 未授权 | 丢弃 |
| 403 | 禁止 | 丢弃 |
| 404 | 没有找到 | 丢弃 |
| 5XX | 回应代码以“5”开头的状态码表示服务器端发现自己出现错误，不能继续执行请求 | 丢弃 |

当返回的状态码为 5XX 时，表示应用服务器出现错误，采用简单的丢弃处理就可以解决。

当返回值状态码为 3XX 时，通常进行转向，以下是转向的代码片段，读者可以和上一节的代码自行整合到一起：

```
//若需要转向，则进行转向操作
if ((statusCode == HttpStatus.SC_MOVED_TEMPORARILY) || (statusCode ==
HttpStatus.SC_MOVED_PERMANENTLY) || (statusCode == HttpStatus.SC_SEE_OTHER)
|| (statusCode == HttpStatus.SC_TEMPORARY_REDIRECT)) {
            //读取新的 URL 地址
            Header header = postMethod.getResponseHeader("location");
            if(header!=null){
                String newUrl = header.getValue();
                if(newUrl==null||newUrl.equals("")){
                    newUrl="/";
                    //使用 post 转向
                    PostMethod redirect = new PostMethod(newUrl);
                    //发送请求，做进一步处理……
                }
            }
    }
```

当响应状态码为 2XX 时，根据表 1.1 的描述，我们只需要处理 200 和 202 两种状态码，其他的返回值可以不做进一步处理。200 的返回状态码是成功状态码，可以直接进行网页抓取，例如：

```
//处理返回值为 200 的状态码
if (statusCode == HttpStatus.SC_OK) {
    input = postMethod.getResponseBodyAsStream();
    //得到文件名
    String filename = path.substring(path.lastIndexOf('/')+1);
    //获得文件输出流
    output = new FileOutputStream(filename);
    //输出到文件
    int tempByte = -1;
    while((tempByte=input.read())>0){
        output.write(tempByte);
    }
}
```

202 的响应状态码表示请求已经接受，服务器再做进一步处理。

## 1.2　宽度优先爬虫和带偏好的爬虫

1.1 节介绍了如何获取单个网页内容。在实际项目中，则使用爬虫程序遍历互联网，把网络中相关的网页全部抓取过来，这也体现了爬虫程序“爬”的概念。爬虫程序是如何遍历互联网，把网页全部抓取下来的呢？互联网可以看成一个超级大的“图”，而每个页面可以看作是一个“节点”。页面中的链接可以看成是图的“有向边”。因此，能够通过图

的遍历的方式对互联网这个超级大“图”进行访问。图的遍历通常可分为宽度优先遍历和深度优先遍历两种方式。但是深度优先遍历可能会在深度上过“深”地遍历或者陷入“黑洞”，大多数爬虫都不采用这种方式。另一方面，在爬取的时候，有时候也不能完全按照宽度优先遍历的方式，而是给待遍历的网页赋予一定的优先级，根据这个优先级进行遍历，这种方法称为带偏好的遍历。本小节会分别介绍宽度优先遍历和带偏好的遍历。

### 1.2.1　图的宽度优先遍历

下面先来看看图的宽度优先遍历过程。图的宽度优先遍历(BFS)算法是一个分层搜索的过程，和树的层序遍历算法相同。在图中选中一个节点，作为起始节点，然后按照层次遍历的方式，一层一层地进行访问。

图的宽度优先遍历需要一个队列作为保存当前节点的子节点的数据结构。具体的算法如下所示：

(1) 顶点 V 入队列。

(2) 当队列非空时继续执行，否则算法为空。

(3) 出队列，获得队头节点 V，访问顶点 V 并标记 V 已经被访问。

(4) 查找顶点 V 的第一个邻接顶点 col。

(5) 若 V 的邻接顶点 col 未被访问过，则 col 进队列。

(6) 继续查找 V 的其他邻接顶点 col，转到步骤(5)，若 V 的所有邻接顶点都已经被访问过，则转到步骤(2)。

下面，我们以图示的方式介绍宽度优先遍历的过程，如图 1.3 所示。

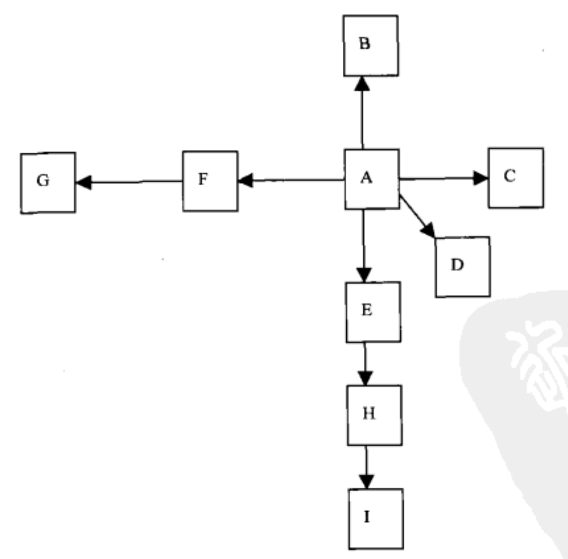



图 1.3　宽度优先遍历过程

选择 A 作为种子节点，则宽度优先遍历的过程，如表 1.2 所示。

表 1.2　宽度优先遍历过程

| 操　作 | 队列中的元素 |
|---|---|
| 初始 | 空 |
| A 入队列 | A |
| A 出队列 | 空 |
| BCDEF 入队列 | BCDEF |
| B 出队列 | CDEF |
| C 出队列 | DEF |
| D 出队列 | EF |
| E 出队列 | F |
| H 入队列 | FH |
| F 出队列 | H |
| G 入队列 | HG |
| H 出队列 | G |
| I 入队列 | GI |
| G 出队列 | I |
| I 出队列 | 空 |

在表 1.2 所示的遍历过程中，出队列的节点顺序既是图的宽度优先遍历的访问顺序。由此可以看出，图 1.3 所示的宽度优先遍历的访问顺序为

A->B->C->D->E->F->H->G->I

本节讲述了宽度优先遍历的理论基础，把互联网看成一个“超图”，则对这张图也可以采用宽度优先遍历的方式进行访问。下面将着重讲解如何对互联网进行宽度优先遍历。

### 1.2.2　宽度优先遍历互联网

1.2.1 节介绍的宽度优先遍历是从一个种子节点开始的。而实际的爬虫项目是从一系列的种子链接开始的。所谓种子链接，就好比宽度优先遍历中的种子节点(图 1.3 中的 A 节点)一样。实际的爬虫项目中种子链接可以有多个，而宽度优先遍历中的种子节点只有一个。比如，可以指定 www.lietu.com 和 www.sina.com 两个种子链接。

如何定义一个链接的子节点？每个链接对应一个 HTML 页面或者其他文件(word、excel、pdf、jpg 等)，在这些文件中，只有 HTML 页面有相应的“子节点”，这些“子节点”就是 HTML 页面上对应的超链接。如 www.lietu.com 页面中(如图 1.4 所示)，“招聘”、“网址”、“更多”以及页面下方的“搜索产品”，“技术文档”，“成功案例”，“猎兔新闻”，“联系猎兔”，“关于我们”， ENGLISH 等都是 www.lietu.com 的子节点。这些子节点本身又是一个链接。对于非 HTML 文档，比如 Excel 文件等，不能从中提取超链接，因此，可以看作是图的“终端”节点。就好像图 1.3 中的 B、C、D、I、G 等节点一样。

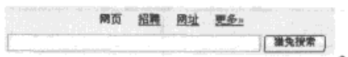



图 1.4　猎兔搜索主页

整个的宽度优先爬虫过程就是从一系列的种子节点开始，把这些网页中的“子节点”(也就是超链接)提取出来，放入队列中依次进行抓取。被处理过的链接需要放入一张表(通常称为 Visited 表)中。每次新处理一个链接之前，需要查看这个链接是否已经存在于 Visited 表中。如果存在，证明链接已经处理过，跳过，不做处理，否则进行下一步处理。实际的过程如图 1.5 所示。

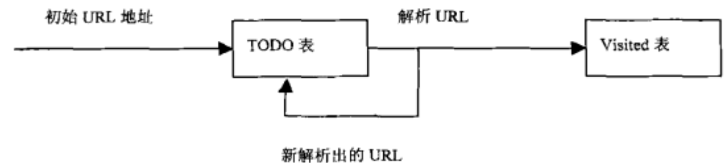



图 1.5　宽度优先爬虫过程

如图 1.5 所示，初始的 URL 地址是爬虫系统中提供的种子 URL(一般在系统的配置文件中指定)。当解析这些种子 URL 所表示的网页时，会产生新的 URL(比如从页面中的<a href="http://www.admin.com"中提取出 http://www.admin.com 这个链接)。然后，进行以下工作：

(1) 把解析出的链接和 Visited 表中的链接进行比较，若 Visited 表中不存在此链接，表示其未被访问过。

(2) 把链接放入 TODO 表中。

(3) 处理完毕后，再次从 TODO 表中取得一条链接，直接放入 Visited 表中。

(4) 针对这个链接所表示的网页，继续上述过程。如此循环往复。

表 1.3 显示了对图 1.3 所示的页面的爬取过程。

表 1.3　网络爬取

| TODO 表 | Visited 表 |
| --- | --- |
| A | 空 |
| BCDEF | A |
| CDEF | A,B |
| DEF | A,B,C |

续表

| TODO 表 | Visited 表 |
|---|---|
| EF | A,B,C,D |
| FH | A,B,C,D,E |
| HG | A,B,C,D,E,F |
| GI | A,B,C,D,E,F,H |
| I | A,B,C,D,E,F,H,G |
| 空 | A,B,C,D,E,F,H,G,I |

宽度优先遍历是爬虫中使用最广泛的一种爬虫策略，之所以使用宽度优先搜索策略，主要原因有三点：

- 重要的网页往往离种子比较近，例如我们打开新闻网站的时候往往是最热门的新闻，随着不断的深入冲浪，所看到的网页的重要性越来越低。
- 万维网的实际深度最多能达到 17 层，但到达某个网页总存在一条很短的路径。而宽度优先遍历会以最快的速度到达这个网页。
- 宽度优先有利于多爬虫的合作抓取，多爬虫合作通常先抓取站内链接，抓取的封闭性很强。

这一小节详细讲述了宽度优先遍历互联网的方法。下一节将给出一个详细的例子来说明如何实现这种方法。

### 1.2.3　Java 宽度优先爬虫示例

本节使用 Java 实现一个简易的爬虫。其中用到了 HttpClient 和 HtmlParser 两个开源工具包。HttpClient 的内容之前已经做过详细的阐述。有关 HtmlParser 的用法，我们会在 4.2 节给出详细的介绍。为了便于读者理解，下面给出示例程序的结构，如图 1.6 所示。

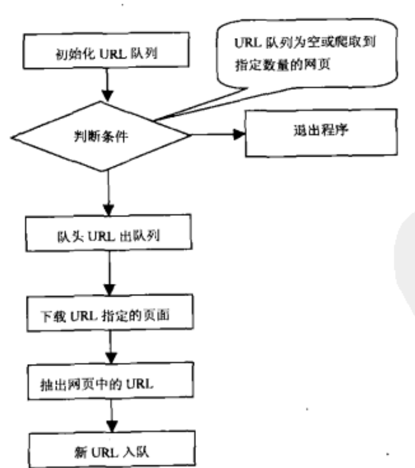



图 1.6　爬虫示例程序结构

首先，需要定义图 1.6 中所描述的“URL 队列”，这里使用一个 LinkedList 来实现这个队列。

**Queue 类：**

```
/**
 * 队列，保存将要访问的URL
 */
public class Queue {
    //使用链表实现队列
    private LinkedList queue = new LinkedList();
    //入队列
    public void enQueue(Object t) {
        queue.addLast(t);
    }
    //出队列
    public Object deQueue() {
            return queue.removeFirst();
    }
    //判断队列是否为空
        public boolean isQueueEmpty() {
            return queue.isEmpty();
    }
    //判断队列是否包含t
    public boolean contians(Object t) {
        return queue.contains(t);
    }

    public boolean empty() {
        return queue.isEmpty();
    }

}
```

除了 URL 队列之外，在爬虫过程中，还需要一个数据结构来记录已经访问过的 URL。每当要访问一个 URL 的时候，首先在这个数据结构中进行查找，如果当前的 URL 已经存在，则丢弃它。这个数据结构要有两个特点：

- 结构中保存的 URL 不能重复。
- 能够快速地查找(实际系统中 URL 的数目非常多，因此要考虑查找性能)。

针对以上两点，我们选择 HashSet 作为存储结构。

**LinkQueue 类：**

```
public class LinkQueue {
    //已访问的 url 集合
    private static Set visitedUrl = new HashSet();
    //待访问的 url 集合
    private static Queue unVisitedUrl = new Queue();
```

```
    //获得 URL 队列
    public static Queue getUnVisitedUrl() {
        return unVisitedUrl;
    }
    //添加到访问过的 URL 队列中
    public static void addVisitedUrl(String url) {
        visitedUrl.add(url);
    }
    //移除访问过的 URL
    public static void removeVisitedUrl(String url) {
        visitedUrl.remove(url);
    }
    //未访问的 URL 出队列
    public static Object unVisitedUrlDeQueue() {
        return unVisitedUrl.deQueue();
    }
    // 保证每个 URL 只被访问一次
    public static void addUnvisitedUrl(String url) {
        if (url != null && !url.trim().equals("")
                && !visitedUrl.contains(url)
                && !unVisitedUrl.contians(url))
            unVisitedUrl.enQueue(url);
    }
    //获得已经访问的 URL 数目
    public static int getVisitedUrlNum() {
        return visitedUrl.size();
    }
    //判断未访问的 URL 队列中是否为空
    public static boolean unVisitedUrlsEmpty() {
        return unVisitedUrl.empty();
    }

}
```

下面的代码详细说明了网页下载并处理的过程。和 1.1 节讲述的内容相比，它考虑了更多的方面。比如如何存储网页，设置请求超时策略等。

**DownLoadFile 类：**

```
public class DownLoadFile {
    /**
     * 根据 URL 和网页类型生成需要保存的网页的文件名，去除 URL 中的非文件名字符
     */
    public String getFileNameByUrl(String url,String contentType)
    {
        //移除 http:
        url=url.substring(7);
        //text/html 类型
        if(contentType.indexOf("html")!=-1)
```

```
    {
        url= url.replaceAll("[\\?/:*|<>\"]", "_")+".html";
        return url;
    }
    //如 application/pdf 类型
    else
    {
     return url.replaceAll("[\\?/:*|<>\"]", "_")+"."+
     contentType.substring(contentType.lastIndexOf("/")+1);
    }
}
/**
 * 保存网页字节数组到本地文件，filePath 为要保存的文件的相对地址
 */
private void saveToLocal(byte[] data, String filePath) {
    try {
        DataOutputStream out = new DataOutputStream(new
        FileOutputStream(new File(filePath)));
        for (int i = 0; i < data.length; i++)
            out.write(data[i]);
        out.flush();
        out.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
// 下载 URL 指向的网页
public String downloadFile(String url) {
    String filePath = null;
    // 1.生成 HttpClinet 对象并设置参数
    HttpClient httpClient = new HttpClient();
    // 设置 HTTP 连接超时 5s
    httpClient.getHttpConnectionManager().getParams()
    .setConnectionTimeout(5000);
    // 2.生成 GetMethod 对象并设置参数
    GetMethod getMethod = new GetMethod(url);
    // 设置 get 请求超时 5s
    getMethod.getParams().setParameter(HttpMethodParams.SO_TIMEOUT, 5000);
    // 设置请求重试处理
getMethod.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
            new DefaultHttpMethodRetryHandler());

    // 3.执行 HTTP GET 请求
    try {
        int statusCode = httpClient.executeMethod(getMethod);
        // 判断访问的状态码
        if (statusCode != HttpStatus.SC_OK) {
```

```
                System.err.println("Method failed: " + getMethod.getStatusLine());
                filePath = null;
            }

            // 4.处理 HTTP 响应内容
            byte[] responseBody = getMethod.getResponseBody();// 读取为字节数组
            // 根据网页 url 生成保存时的文件名
            filePath = "temp\\"
                    + getFileNameByUrl(url, getMethod.getResponseHeader(
                            "Content-Type").getValue());
            saveToLocal(responseBody, filePath);
        } catch (HttpException e) {
            // 发生致命的异常，可能是协议不对或者返回的内容有问题
            System.out.println("Please check your provided http address!");
            e.printStackTrace();
        } catch (IOException e) {
            // 发生网络异常
            e.printStackTrace();
        } finally {
            // 释放连接
            getMethod.releaseConnection();
        }
        return filePath;
    }
}
```

接下来，演示如何从获得的网页中提取 URL。Java 有一个非常实用的开源工具包 Html Parser，它专门针对 Html 页面进行处理，不仅能提取 URL，还能提取文本以及你想要的任何内容。关于它的有关内容，会在第 4 章详细介绍。好了，让我们看看代码吧！

**HtmlParserTool 类：**

```
public class HtmlParserTool {
    // 获取一个网站上的链接，filter 用来过滤链接
    public static Set<String> extracLinks(String url, LinkFilter filter) {
        Set<String> links = new HashSet<String>();
        try {
            Parser parser = new Parser(url);
            parser.setEncoding("gb2312");
            // 过滤 <frame >标签的 filter，用来提取 frame 标签里的 src 属性
        NodeFilter frameFilter = new NodeFilter() {
                public boolean accept(Node node) {
                    if (node.getText().startsWith("frame src=")) {
                        return true;
                    } else {
                        return false;
                }
            };
```

```
            // OrFilter 来设置过滤 <a> 标签和 <frame> 标签
            OrFilter linkFilter = new OrFilter(new NodeClassFilter(
                    LinkTag.class), frameFilter);
            // 得到所有经过过滤的标签
            NodeList list = parser.extractAllNodesThatMatch(linkFilter);
            for (int i = 0; i < list.size(); i++) {
                Node tag = list.elementAt(i);
                if (tag instanceof LinkTag)// <a> 标签
                {
                    LinkTag link = (LinkTag) tag;
                    String linkUrl = link.getLink();// URL
                    if (filter.accept(linkUrl))
                        links.add(linkUrl);
                } else// <frame> 标签
                {
                    // 提取 frame 里 src 属性的链接,如 <frame src="test.html"/>
                    String frame = tag.getText();
                    int start = frame.indexOf("src=");
                    frame = frame.substring(start);
                    int end = frame.indexOf(" ");
                    if (end == -1)
                        end = frame.indexOf(">");
                    String frameUrl = frame.substring(5, end - 1);
                    if (filter.accept(frameUrl))
                        links.add(frameUrl);
                }
            }
        } catch (ParserException e) {
            e.printStackTrace();
        }
        return links;
    }
}
```

最后，来看看宽度爬虫的主程序。

**MyCrawler 类：**

```
public class MyCrawler {
    /**
     * 使用种子初始化 URL 队列
     * @return
     * @param seeds 种子 URL
     */
    private void initCrawlerWithSeeds(String[] seeds)
    {
        for(int i=0;i<seeds.length;i++)
            LinkQueue.addUnvisitedUrl(seeds[i]);
    }
```

```
    /**
     * 抓取过程
     * @return
     * @param seeds
     */
    public void crawling(String[] seeds)
    {   //定义过滤器，提取以 http://www.lietu.com 开头的链接
        LinkFilter filter = new LinkFilter(){
            public boolean accept(String url) {
                if(url.startsWith("http://www.lietu.com"))
                    return true;
                else
                    return false;
            }
        };
        //初始化 URL 队列
        initCrawlerWithSeeds(seeds);
        //循环条件：待抓取的链接不空且抓取的网页不多于 1000

        while(!LinkQueue.unVisitedUrlsEmpty()
        &&LinkQueue.getVisitedUrlNum()<=1000)
        {
            //队头 URL 出队列
            String visitUrl=(String)LinkQueue.unVisitedUrlDeQueue();
            if(visitUrl==null)
                continue;
            DownLoadFile downLoader=new DownLoadFile();
            //下载网页
            downLoader.downloadFile(visitUrl);
            //该 URL 放入已访问的 URL 中
            LinkQueue.addVisitedUrl(visitUrl);
            //提取出下载网页中的 URL
            Set<String> links=HtmlParserTool.extracLinks(visitUrl,filter);
            //新的未访问的 URL 入队
            for(String link:links)
            {
                LinkQueue.addUnvisitedUrl(link);
            }
        }
    }
    //main 方法入口
    public static void main(String[]args)
    {
        MyCrawler crawler = new MyCrawler();
        crawler.crawling(new String[]{"http://www.lietu.com"});
    }

}
```

上面的主程序使用了一个 LinkFilter 接口，并且实现为一个内部类。这个接口的目的是

为了过滤提取出来的URL，它使得程序中提取出来的URL只会和猎兔网站相关。而不会提取其他无关的网站。代码如下：

```
public interface LinkFilter {
    public boolean accept(String url);
}
```

### 1.2.4 带偏好的爬虫

有时，在URL队列中选择需要抓取的URL时，不一定按照队列“先进先出”的方式进行选择。而把重要的URL先从队列中“挑”出来进行抓取。这种策略也称作“页面选择”(Page Selection)。这可以使有限的网络资源照顾重要性高的网页。

那么哪些网页是重要性高的网页呢？

判断网页的重要性的因素很多，主要有链接的欢迎度(知道链接的重要性了吧)、链接的重要度和平均链接深度、网站质量、历史权重等主要因素。

链接的欢迎度主要是由反向链接(backlinks，即指向当前URL的链接)的数量和质量决定的，我们定义为IB(P)。

链接的重要度，是一个关于URL字符串的函数，仅仅考察字符串本身，比如认为“.com”和“home”的URL重要度比“.cc”和“map”高，我们定义为IL(P)。

平均链接深度，根据上面所分析的宽度优先的原则计算出全站的平均链接深度，然后认为距离种子站点越近的重要性越高。我们定义为ID(P)。

如果我们定义网页的重要性为I(P)，那么，页面的重要度由下面的公式决定：

$$I(P)=X*IB(P)+Y*IL(P) \tag{1.1}$$

其中，X和Y两个参数，用来调整IB(P)和IL(P)所占比例的大小，ID(P)由宽度优先的遍历规则保证，因此不作为重要的指标函数。

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来实现TODO表，并且把每个URL的重要性作为队列元素的优先级。这样，每次选出来扩展的URL就是具有最高重要性的网页。有关优先级队列的介绍，请参考1.2.5节中的内容。

例如，假设图1.3中节点的重要性为D>B>C>A>E>F>I>G>H，则整个遍历过程如表1.4所示。

表1.4 带偏好的爬虫

| TODO优先级队列 | Visited表 |
|---|---|
| A | null |
| D,B,C, E,F | A |
| B,C,E,F | A,D |
| C,E,F | A, D, B |
| E,F | A,D,B,C |
| F,H | A,D,B,C,E |
| G,H | A,D,B,C, E,F |

续表

| Todo 优先级队列 | Visited 表 |
|---|---|
| H | A,D,B,C,E,F,G |
| I | A,D,B,C,E,F,G,H |
| null | A,D,B,C,E,F,G,H,I |

### 1.2.5 Java 带偏好的爬虫示例

在 1.2.4 节中，我们已经指出，可以使用优先级队列(Priority Queue)来实现这个带偏好的爬虫。在深入讲解之前，我们首先介绍优先级队列。

优先级队列是一种特殊的队列，普通队列中的元素是先进先出的，而优先级队列则是根据进入队列中的元素的优先级进行出队列操作。例如操作系统的一些优先级进程管理等，都可以使用优先级队列。优先级队列也有最小优先级队列和最大优先级队列两种。

理论上，优先级队列可以是任何一种数据结构，线性的和非线性的，也可以是有序的或无序的。针对有序的优先级队列而言，获取最小或最大的值是非常容易的，但是插入却非常困难；而对于无序的有衔接队列而言，插入是很容易的，但是获取最大和最小值是很麻烦的。根据以上的分析，可以使用“堆”这种折中的数据结构来实现优先级队列。

从 JDK 1.5 开始，Java 提供了内置的支持优先级队列的数据结构——java.util.PriorityQueue。我们在上边的代码中，只要稍微修改一下，就可以支持从 URL 队列中选择优先级高的 URL。

**LinkQueue 类：**

```
public class LinkQueue {
    //已访问的 URL 集合
    private static Set visitedUrl = new HashSet();
    //待访问的 URL 集合
    private static Queue unVisitedUrl = new PriorityQueue();

    //获得 URL 队列
    public static Queue getUnVisitedUrl() {
        return unVisitedUrl;
    }
    //添加到访问过的 URL 队列中
    public static void addVisitedUrl(String url) {
        visitedUrl.add(url);
    }
    //移除访问过的 URL
    public static void removeVisitedUrl(String url) {
        visitedUrl.remove(url);
    }
    //未访问的 URL 出队列
    public static Object unVisitedUrlDeQueue() {
        return unVisitedUrl.poll();
    }
```

```
    // 保证每个 URL 只被访问一次
    public static void addUnvisitedUrl(String url) {
        if (url != null && !url.trim().equals("")
                && !visitedUrl.contains(url)
                && !unVisitedUrl.contains(url))
            unVisitedUrl.add(url);
    }
    //获得已经访问的 URL 数目
    public static int getVisitedUrlNum() {
        return visitedUrl.size();
    }
    //判断未访问的 URL 队列中是否为空
    public static boolean unVisitedUrlsEmpty() {
        return unVisitedUrl.isEmpty();
    }

}
```

在带偏好的爬虫里，队列元素的优先级是由 URL 的优先级确定的。关于如何确定 URL 的优先级，有一些专用的链接分析的方法，比如 Google 的 PageRank 和 HITS 算法。有关这些算法的内容，将在第 8 章详细介绍。

## 1.3 设计爬虫队列

通过前几节的介绍，可以看出，网络爬虫最关键的数据结构是 URL 队列——通常我们称之为爬虫队列。之前的几节一直使用内存数据结构，例如链表或者队列来实现 URL 队列。但是，网络中需要我们抓取的链接成千上万，在一些大型的搜索引擎中，比如百度和 Google，大概都有十几亿的 URL 需要抓取。因此，内存数据结构并不适合这些应用。最合适的一种方法就是使用内存数据库，或者直接使用数据库来存储这些 URL。本节将讲解爬虫队列的基本知识，并且介绍一种非常流行的内存数据库——Berkeley DB，最后介绍一个成熟的开源爬虫软件——Heritrix 是如何实现爬虫队列的。

### 1.3.1 爬虫队列

爬虫队列的设计是网络爬虫的关键。爬虫队列是用来保存 URL 的队列数据结构的。在大型爬虫应用中，构建通用的、可以扩缩的爬虫队列非常重要。数以十亿计的 URL 地址，使用内存的链表或者队列来存储显然不够，因此，需要找到一种数据结构，这种数据结构具有以下几个特点：

- 能够存储海量数据，当数据超出内存限制的时候，能够把它固化在硬盘上。
- 存取数据速度非常快。
- 能够支持多线程访问(多线程技术能够大规模提升爬虫的性能，这点将在之后的章节详细介绍)。

结合上面 3 点中对存储速度的要求，使 Hash 成为存储结构的不二选择。

通常，在进行 Hash 存储的时候，key 值都选取 URL 字符串，但是为了更节省空间，通常会对 URL 进行压缩。常用的压缩算法是 MD5 压缩算法。

### MD5 算法描述

对 MD5 算法的简要叙述为：MD5 以 512 位分组来处理输入的信息，且每一分组又被划分为 16 个 32 位子分组，经过一系列的处理后，算法的输出由 4 个 32 位分组组成，将这 4 个 32 位分组级联后将生成一个 128 位的散列值。

在 MD5 算法中，首先需要对信息进行填充，使其位长度对 512 求余的结果等于 448。因此，信息的位长度(Bits Length)将被扩展至 N*512+448，即 N*64+56 个字节(Bytes)，N 为一个正整数。填充的方法如下：

在信息的后面填充一个 1 和无数个 0，直到满足上面的条件时才停止用 0 对信息的填充。然后，在这个结果后面附加一个以 64 位二进制表示的填充前信息长度。经过这两步的处理，现在的信息字节长度=N*512+448+64=(N+1)*512，即长度恰好是 512 的整数倍。这样做的原因是为了满足后面处理中对信息长度的要求。

MD5 中有 4 个 32 位被称作链接变量(Chaining Variable)的整数参数，它们分别为：A=0x01234567, B=0x89abcdef, C=0xfedcba98, D=0x76543210。

当设置好这四个链接变量后，就开始进入算法的四轮循环运算。循环的次数是信息中 512 位信息分组的数目。

将上面 4 个链接变量复制到另外 4 个变量中：A 到 a，B 到 b，C 到 c，D 到 d。

主循环有四轮(MD4 只有三轮)，每轮循环都很相似。第一轮进行 16 次操作。每次操作对 a、b、c 和 d 中的三个做一次非线性函数运算，然后将所得结果依次加上第四个变量、文本的一个子分组和一个常数。再将所得结果向右循环移动一个不定的数，并加上 a、b、c 或 d 中之一。最后用该结果取代 a、b、c 或 d 中之一。

以下是每次操作中用到的四个非线性函数(每轮一个)。

F(X,Y,Z) =(X&Y)|((~X)&Z)

G(X,Y,Z) =(X&Z)|(Y&(~Z))

H(X,Y,Z) =X^Y^Z

I(X,Y,Z)=Y^(X|(~Z))

(&是与，|是或，~是非，^是异或)

这四个函数的说明：如果 X、Y 和 Z 的对应位是独立和均匀的，那么结果的每一位也应是独立和均匀的。F 是一个逐位运算的函数。即，如果 X，那么 Y，否则 Z。函数 H 是逐位奇偶操作符。

假设 Mj 表示消息的第 j 个子分组(从 0 到 15)，FF(a,b,c,d,Mj,s,ti)表示 a=b+((a+(F(b,c,d)+Mj+ti)，GG(a,b,c,d,Mj,s,ti)表示 a=b+((a+(G(b,c,d)+Mj+ti)，HH(a,b,c,d,Mj,s,ti)表示

a=b+((a+(H(b,c,d)+Mj+ti)，II(a,b,c,d,Mj,s,ti)表示 a=b+((a+(I(b,c,d)+Mj+ti)。

这四轮(64 步)是：

第一轮

FF(a,b,c,d,M0,7,0xd76aa478)
FF(d,a,b,c,M1,12,0xe8c7b756)
FF(c,d,a,b,M2,17,0x242070db)
FF(b,c,d,a,M3,22,0xc1bdceee)
FF(a,b,c,d,M4,7,0xf57c0faf)
FF(d,a,b,c,M5,12,0x4787c62a)
FF(c,d,a,b,M6,17,0xa8304613)
FF(b,c,d,a,M7,22,0xfd469501)
FF(a,b,c,d,M8,7,0x698098d8)
FF(d,a,b,c,M9,12,0x8b44f7af)
FF(c,d,a,b,M10,17,0xffff5bb1)
FF(b,c,d,a,M11,22,0x895cd7be)
FF(a,b,c,d,M12,7,0x6b901122)
FF(d,a,b,c,M13,12,0xfd987193)
FF(c,d,a,b,M14,17,0xa679438e)
FF(b,c,d,a,M15,22,0x49b40821)

第二轮

GG(a,b,c,d,M1,5,0xf61e2562)
GG(d,a,b,c,M6,9,0xc040b340)
GG(c,d,a,b,M11,14,0x265e5a51)
GG(b,c,d,a,M0,20,0xe9b6c7aa)
GG(a,b,c,d,M5,5,0xd62f105d)
GG(d,a,b,c,M10,9,0x02441453)
GG(c,d,a,b,M15,14,0xd8a1e681)
GG(b,c,d,a,M4,20,0xe7d3fbc8)
GG(a,b,c,d,M9,5,0x21e1cde6)
GG(d,a,b,c,M14,9,0xc33707d6)
GG(c,d,a,b,M3,14,0xf4d50d87)
GG(b,c,d,a,M8,20,0x455a14ed)
GG(a,b,c,d,M13,5,0xa9e3e905)
GG(d,a,b,c,M2,9,0xfcefa3f8)
GG(c,d,a,b,M7,14,0x676f02d9)
GG(b,c,d,a,M12,20,0x8d2a4c8a)

第三轮

HH(a,b,c,d,M5,4,0xfffa3942)
HH(d,a,b,c,M8,11,0x8771f681)
HH(c,d,a,b,M11,16,0x6d9d6122)
HH(b,c,d,a,M14,23,0xfde5380c)
HH(a,b,c,d,M1,4,0xa4beea44)
HH(d,a,b,c,M4,11,0x4bdecfa9)
HH(c,d,a,b,M7,16,0xf6bb4b60)
HH(b,c,d,a,M10,23,0xbebfbc70)
HH(a,b,c,d,M13,4,0x289b7ec6)
HH(d,a,b,c,M0,11,0xeaa127fa)
HH(c,d,a,b,M3,16,0xd4ef3085)
HH(b,c,d,a,M6,23,0x04881d05)
HH(a,b,c,d,M9,4,0xd9d4d039)
HH(d,a,b,c,M12,11,0xe6db99e5)
HH(c,d,a,b,M15,16,0x1fa27cf8)
HH(b,c,d,a,M2,23,0xc4ac5665)

第四轮

II(a,b,c,d,M0,6,0xf4292244)
II(d,a,b,c,M7,10,0x432aff97)
II(c,d,a,b,M14,15,0xab9423a7)
II(b,c,d,a,M5,21,0xfc93a039)
II(a,b,c,d,M12,6,0x655b59c3)
II(d,a,b,c,M3,10,0x8f0ccc92)
II(c,d,a,b,M10,15,0xffeff47d)
II(b,c,d,a,M1,21,0x85845dd1)
II(a,b,c,d,M8,6,0x6fa87e4f)
II(d,a,b,c,M15,10,0xfe2ce6e0)
II(c,d,a,b,M6,15,0xa3014314)
II(b,c,d,a,M13,21,0x4e0811a1)
II(a,b,c,d,M4,6,0xf7537e82)
II(d,a,b,c,M11,10,0xbd3af235)
II(c,d,a,b,M2,15,0x2ad7d2bb)
II(b,c,d,a,M9,21,0xeb86d391)

常数 ti 可以如下选择：

在第 i 步中，ti 是 4294967296*abs(sin(i))的整数部分，i 的单位是弧度(4294967296 等于 2 的 32 次方)。

所有这些都完成之后，将 A、B、C、D 分别加上 a、b、c、d。然后用下一分组数据继续运行算法，最后的输出是 A、B、C 和 D 的级联。

在 Java 中，java.security.MessageDigest 中已经定义了 MD5 的计算，只需要简单地调用即可得到 MD5 的 128 位整数。然后将此 128 位(16 个字节)转换成十六进制表示即可。下面是一段 Java 实现 MD5 压缩的代码。

**MD5 压缩算法代码：**

```
/* 传入参数：一个字节数组
 * 传出参数：字节数组的 MD5 结果字符串
 */
public class MD5 {
 public static String getMD5(byte[] source) {
  String s = null;
  char hexDigits[] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a',
'b', 'c', 'd', 'e', 'f'};        // 用来将字节转换成十六进制表示的字符

   try
   {
   java.security.MessageDigest md =
   java.security.MessageDigest.getInstance( "MD5" );
    md.update( source );
    byte tmp[] = md.digest();       // MD5 的计算结果是一个 128 位的长整数，
                                    // 用字节表示就是 16 个字节
    char str[] = new char[16 * 2]; // 每个字节用十六进制表示的话，使用两个字符，
                                    // 所以表示成十六进制需要 32 个字符
    int k = 0;                      // 表示转换结果中对应的字符位置
    for (int i = 0; i < 16; i++) { // 从第一个字节开始，将 MD5 的每一个字节
                                    // 转换成十六进制字符
     byte byte0 = tmp[i];           // 取第 i 个字节
     str[k++] = hexDigits[byte0 >>> 4 & 0xf];  // 取字节中高 4 位的数字转换，
                                               // >>> 为逻辑右移，将符号位一起右移
     str[k++] = hexDigits[byte0 & 0xf];     // 取字节中低 4 位的数字转换
    }
    s = new String(str);                   // 将换后的结果转换为字符串
   }catch( Exception e )
   {
    e.printStackTrace();
   }
   return s;
 }
}
```

Hash 存储的 Value 值通常会对 URL 和相关的信息进行封装，封装成为一个对象进行存储。

综合前面分析的信息，选择一个可以进行线程安全、使用 Hash 存储，并且能够应对海量数据的内存数据库是存储 URL 最合适的数据结构。因此，由 Oracle 公司开发的内存数据库产品 Berkeley DB 就进入了我们的视线。下一节，将集中精力介绍 Berkeley DB 以及它的用法。

### 1.3.2　使用 Berkeley DB 构建爬虫队列

Berkeley DB 是一个嵌入式数据库，它适合于管理海量的、简单的数据。例如，Google 用 Berkeley DB HA (High Availability)来管理他们的账户信息。Motorola 在它的无线产品中用 Berkeley DB 跟踪移动单元。HP、Microsoft、Sun Microsystems 等也都是它的大客户。Berkeley DB 不能完全取代关系型数据库，但在某些方面，它却令关系数据库望尘莫及。

关键字/数据(key/value)是 Berkeley DB 用来进行数据库管理的基础。每个 key/value 对构成一条记录。而整个数据库实际上就是由许多这样的结构单元所构成的。通过这种方式，开发人员在使用 Berkeley DB 提供的 API 访问数据库时，只需提供关键字就能够访问到相应的数据。当然也可以提供 Key 和部分 Data 来查询符合条件的相近数据。

Berkeley DB 底层实现采用 B 树，可以看成能够存储大量数据的 HashMap。Berkeley DB 简称 BDB，官方网址是：http://www.oracle.com/database/berkeley-db/index.html。Berkeley DB 的 C++版本首先出现，然后在此基础上又实现了 Java 本地版本。Berkeley DB 是通过环境对象 EnvironmentConfig 来对数据库进行管理的，每个 EnvironmentConfig 对象可以管理多个数据库。新建一个 EnvironmentConfig 的代码如下：

```
EnvironmentConfig envConfig = new EnvironmentConfig();
   envConfig.setTransactional(false);
   envConfig.setAllowCreate(true);
exampleEnv = new Environment(envDir, envConfig);
```

其中，envDir 是用户指定的一个目录。只要是由同一个 EnvironmentConfig 指定的数据库的数据文件和日志文件，都会放在这个目录下。EnvironmentConfig 也是一种资源，当使用完毕后，需要关闭。

```
exampleEnv.sync();
exampleEnv.close();
exampleEnv = null;
```

创建好环境之后，就可以用它创建数据库了。用 Berkeley DB 创建数据库时，需要指定数据库的属性，就好比在 Oracle 中创建数据库时要指定 java_pool、buffer_size 等属性一样。Berkeley DB 使用 DatabaseConfig 来管理一个具体的 DataBase：

```
String databaseName= "ToDoTaskList.db";
DatabaseConfig dbConfig = new DatabaseConfig();
dbConfig.setAllowCreate(true);
dbConfig.setTransactional(false);

// 打开用来存储类信息的数据库
//用来存储类信息的数据库不要求能够存储重复的关键字
dbConfig.setSortedDuplicates(false);
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 value 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);
```

当数据库建立起来之后，就要确定往数据库里面存储的数据类型(也就是确定 key 和 value 的值)。Berkeley DB 数据类型是使用 EntryBinding 对象来确定的。

```
EntryBinding keyBinding =new SerialBinding(javaCatalog,String.class);
```

其中，SerialBinding 表示这个对象能够序列化到磁盘上，因此，构造函数的第二个参数一定要是实现了序列化接口的对象。

最后，我们来创建一个以 Berkeley DB 为底层数据结构的 Map：

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

### 1.3.3 使用 Berkeley DB 构建爬虫队列示例

上一节我们讲述了 Berkeley DB 的基础知识，这一节要讲述如何使用 Berkeley DB 来构建一个完整的爬虫队列。

首先，Berkeley DB 存储是一个 key/value 的结构，并且 key 和 value 对象都要实现 Java 序列化接口。因此，我们先来构建 value 对象，即一个封装了很多重要属性的 URL 类。

**Berkeley DB 中存储的 Value 类：**

```
public class CrawlUrl implements Serializable {
    private static final long serialVersionUID = 7931672194843948629L;

    public CrawlUrl() {

    }
    private String oriUrl;// 原始 URL 的值，主机部分是域名


    private String url;     // URL 的值，主机部分是 IP，为了防止重复主机的出现
    private int urlNo;                      // URL NUM
    private int statusCode;                 // 获取 URL 返回的结果码
    private int hitNum;                     // 此 URL 被其他文章引用的次数
    private String charSet;                 // 此 URL 对应文章的汉字编码
    private String abstractText;            // 文章摘要
    private String author;                  // 作者
    private int weight;                     // 文章的权重(包含导向词的信息)
    private String description;             // 文章的描述
    private int fileSize;                   // 文章大小
    private Timestamp lastUpdateTime;       // 最后修改时间
    private Date timeToLive;                // 过期时间
    private String title;                   // 文章名称
```

```
private String type;            // 文章类型
private String[] urlRefrences;  // 引用的链接
private int layer;              //爬取的层次,从种子开始,依次为第0层,第1层...

public int getLayer() {
    return layer;
}

public void setLayer(int layer) {
    this.layer = layer;
}

public String getUrl() {
    return url;
}

public void setUrl(String url) {
    this.url = url;
}

public int getUrlNo() {
    return urlNo;
}

public void setUrlNo(int urlNo) {
    this.urlNo = urlNo;
}

public int getStatusCode() {
    return statusCode;
}

public void setStatusCode(int statusCode) {
    this.statusCode = statusCode;
}

public int getHitNum() {
    return hitNum;
}

public void setHitNum(int hitNum) {
    this.hitNum = hitNum;
}

public String getCharSet() {
    return charSet;
}
```

```
public void setCharSet(String charSet) {
    this.charSet = charSet;
}

public String getAbstractText() {
    return abstractText;
}

public void setAbstractText(String abstractText) {
    this.abstractText = abstractText;
}

public String getAuthor() {
    return author;
}

public void setAuthor(String author) {
    this.author = author;
}

public int getWeight() {
    return weight;
}

public void setWeight(int weight) {
    this.weight = weight;
}

public String getDescription() {
    return description;
}

public void setDescription(String description) {
    this.description = description;
}

public int getFileSize() {
    return fileSize;
}

public void setFileSize(int fileSize) {
    this.fileSize = fileSize;
}

public Timestamp getLastUpdateTime() {
    return lastUpdateTime;
```

```
    }

    public void setLastUpdateTime(Timestamp lastUpdateTime) {
        this.lastUpdateTime = lastUpdateTime;
    }

    public Date getTimeToLive() {
        return timeToLive;
    }

    public void setTimeToLive(Date timeToLive) {
        this.timeToLive = timeToLive;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public String[] getUrlRefrences() {
        return urlRefrences;
    }

    public void setUrlRefrences(String[] urlRefrences) {
        this.urlRefrences = urlRefrences;
    }

    public final String getOriUrl() {
        return oriUrl;
    }

    public void setOriUrl(String oriUrl) {
        this.oriUrl = oriUrl;
    }
}
```

写一个 TODO 表的接口：

```
public interface Frontier {
    public CrawlUrl getNext()throws Exception;
    public boolean putUrl(CrawlUrl url) throws Exception;
    //public boolean visited(CrawlUrl url);
}
```

使用一个抽象类来封装对 Berkeley DB 的操作：

```
public abstract class AbstractFrontier {
    private Environment env;
    private static final String CLASS_CATALOG = "java_class_catalog";
    protected StoredClassCatalog javaCatalog;
    protected Database catalogdatabase;
    protected Database database;

    public AbstractFrontier(String homeDirectory) throws DatabaseException,
            FileNotFoundException {
        // 打开 env
        System.out.println("Opening environment in: " + homeDirectory);
        EnvironmentConfig envConfig = new EnvironmentConfig();
        envConfig.setTransactional(true);
        envConfig.setAllowCreate(true);
        env = new Environment(new File(homeDirectory), envConfig);
        // 设置 DatabaseConfig
        DatabaseConfig dbConfig = new DatabaseConfig();
        dbConfig.setTransactional(true);
        dbConfig.setAllowCreate(true);
        // 打开
        catalogdatabase = env.openDatabase(null, CLASS_CATALOG, dbConfig);
        javaCatalog = new StoredClassCatalog(catalogdatabase);
        // 设置 DatabaseConfig
        DatabaseConfig dbConfig0 = new DatabaseConfig();
        dbConfig0.setTransactional(true);
        dbConfig0.setAllowCreate(true);
        // 打开
        database = env.openDatabase(null, "URL", dbConfig);
    }
    //关闭数据库，关闭环境
    public void close() throws DatabaseException {
        database.close();
        javaCatalog.close();
        env.close();
    }
    //put 方法
    protected abstract void put(Object key,Object value);
    //get 方法
    protected abstract Object get(Object key);
    //delete 方法
```

```
    protected abstract Object delete(Object key);
}
```

实现真正的 TODO 表：

```
public class BDBFrontier extends AbstractFrontier implements Frontier {
    private StoredMap pendingUrisDB = null;

    //使用默认的路径和缓存大小构造函数
    public BDBFrontier(String homeDirectory) throws DatabaseException,
    FileNotFoundException{
        super(homeDirectory);
        EntryBinding keyBinding =new SerialBinding
        (javaCatalog,String.class);
        EntryBinding valueBinding =new SerialBinding(javaCatalog,
        CrawlUrl.class);
        pendingUrisDB = new StoredMap(database,keyBinding, valueBinding,
        true);
    }

    //获得下一条记录
    public CrawlUrl getNext() throws Exception {
        CrawlUrl result = null;
        if(!pendingUrisDB.isEmpty()){
            Set entrys = pendingUrisDB.entrySet();
            System.out.println(entrys);
            Entry<String,CrawlUrl>
entry=(Entry<String,CrawlUrl>)pendingUrisDB.entrySet().iterator().next();
            result = entry.getValue();
            delete(entry.getKey());
        }
        return result;
    }
    //存入 URL
    public boolean putUrl(CrawlUrl url){
    put(url.getOriUrl(),url);
    return true;
    }
    // 存入数据库的方法
    protected void put(Object key,Object value) {
        pendingUrisDB.put(key, value);
    }
    //取出
    protected Object get(Object key){
        return pendingUrisDB.get(key);
    }
    //删除
    protected Object delete(Object key){
```

```
            return pendingUrisDB.remove(key);
        }


        // 根据 URL 计算键值，可以使用各种压缩算法，包括 MD5 等压缩算法
        private String caculateUrl(String url) {
            return url;
        }
        // 测试函数
        public static void main(String[] strs) {

            try {
                BDBFrontier  bBDBFrontier = new BDBFrontier("c:\\bdb");
                CrawlUrl url = new CrawlUrl();
                url.setOriUrl("http://www.163.com");
                bBDBFrontier.putUrl(url);

        System.out.println(((CrawlUrl)bBDBFrontier.getNext()).getOriUrl());
                bBDBFrontier.close();
            } catch (Exception e) {
                e.printStackTrace();
            }finally{

            }

        }
    }
```

以上就是一个使用 Berkeley DB 来实现 TODO 表的完整示例。

### 1.3.4 使用布隆过滤器构建 Visited 表

上一节内容介绍了如何实现 TODO 表，这一节来探讨如何在一个企业级的搜索引擎中实现 Visited 表。在企业级搜索引擎中，常用一个称为布隆过滤器(Bloom Filter)的算法来实现对已经抓取过的 URL 的过滤。首先来介绍什么叫布隆过滤器(Bloom Filter)算法。

在日常生活中，包括在设计计算机软件时，经常要判断一个元素是否在一个集合中。比如在字处理软件中，需要检查一个英语单词是否拼写正确(也就是要判断它是否在已知的字典中)；在 FBI 中，一个嫌疑人的名字是否已经在嫌疑名单上；在网络爬虫里，一个网址是否被访问过等。最直接的方法就是将集合中全部的元素存在计算机中，遇到一个新元素时，将它和集合中的元素直接比较即可。一般来讲，计算机中的集合是用哈希表(Hash Table)来存储的。它的好处是快速而准确，缺点是费存储空间。当集合比较小时，这个问题不显著，但是当集合巨大时，哈希表存储效率低的问题就显现出来了。比如说，一个像 Yahoo、Hotmail 和 Gmail 那样的公众电子邮件(E-mail)提供商，总是需要过滤来自发送垃圾邮件的人(spamer)的垃圾邮件。一个办法就是记录下那些发垃圾邮件的 E-mail 地址。由于那些发送

者不停地在注册新的地址，全世界少说也有几十亿个发垃圾邮件的地址，将它们都存起来则需要大量的网络服务器。如果用哈希表，每存储一亿个 E-mail 地址，就需要 1.6GB 的内存(用哈希表实现的具体办法是将每一个 E-mail 地址对应成一个八字节的信息指纹， 然后将这个信息指纹存入哈希表，由于哈希表的存储效率一般只有 50%，因此一个 E-mail 地址需要占用十六个字节。一亿个地址大约要 1.6GB，即十六亿字节的内存)。因此存储几十亿个邮件地址可能需要上百 GB 的内存。除非是超级计算机，一般服务器是无法存储的。

一种称作布隆过滤器的数学工具，它只需要哈希表 1/8 到 1/4 的大小就能解决同样的问题。

布隆过滤器是由巴顿·布隆于 1970 年提出的。它实际上是一个很长的二进制向量和一系列随机映射函数。我们通过上面的例子来说明其工作原理。

假定存储一亿个电子邮件地址，先建立一个 16 亿二进制常量，即两亿字节的向量，然后将这 16 亿个二进制位全部设置为零。对于每一个电子邮件地址 X，用 8 个不同的随机数产生器(F1,F2, …, F8)产生 8 个信息指纹(f1, f2, …, f8)。再用一个随机数产生器 G 把这 8 个信息指纹映射到 1 到 16 亿中的 8 个自然数 g1, g2, …, g8。现在我们把这 8 个位置的二进制位全部设置为 1。当我们对这 1 亿个 E-mail 地址都进行这样的处理后。一个针对这些 E-mail 地址的布隆过滤器就建成了，如图 1.7 所示。

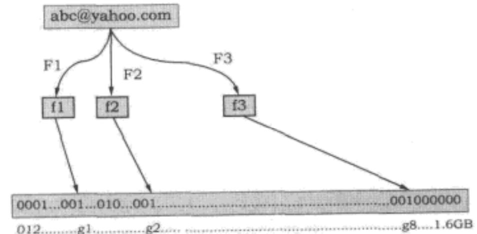



图 1.7 布隆过滤器(Bloom Filter)

现在，来看看布隆过滤器是如何检测一个可疑的电子邮件地址 Y 是否在黑名单中的。我们用 8 个随机数产生器(F1, F2, …, F8)对这个地址产生 8 个信息指纹 S1, S2, …, S8。然后将这 8 个指纹对应到布隆过滤器的 8 个二进制位，分别是 T1, T2, …, T8。如果 Y 在黑名单中，显然，T1, T2, …, T8 对应的 8 个二进制位一定是 1。这样在遇到任何黑名单中的电子邮件地址时，我们都能够准确地发现。

布隆过滤器绝对不会漏掉任何一个在黑名单中的可疑地址。但是，它有一条不足之处。也就是它有极小的可能将一个不在黑名单中的电子邮件地址判定为在黑名单中，因为有可能某个好的邮件地址正巧对应 8 个都被设置成 1 的二进制位。好在这种可能性很小。我们把它称为误识概率。在上面的例子中，误识概率大概在万分之一以下。常见的补救办法是建立一个小的白名单，存储那些可能误判的邮件地址。

下面是一个布隆过滤器(Bloom Filter)的实现：

```
public class SimpleBloomFilter implements VisitedFrontier {
    private static final int DEFAULT_SIZE = 2 << 24;
    private static final int[] seeds = new int[] { 7, 11, 13, 31, 37, 61, };
```

```
private BitSet bits = new BitSet(DEFAULT_SIZE);
private SimpleHash[] func = new SimpleHash[seeds.length];
public static void main(String[] args) {
    String value = "stone2083@yahoo.cn";
    SimpleBloomFilter filter = new SimpleBloomFilter();
    System.out.println(filter.contains(value));
    filter.add(value);
    System.out.println(filter.contains(value));
}
public SimpleBloomFilter() {
    for (int i = 0; i < seeds.length; i++) {
        func[i] = new SimpleHash(DEFAULT_SIZE, seeds[i]);
    }
}
// 覆盖方法，把 URL 添加进来
public void add(CrawlUrl value) {
    if (value != null) {
        add(value.getOriUrl());
    }
}
// 覆盖方法，把 URL 添加进来
public void add(String value) {
    for (SimpleHash f : func) {
        bits.set(f.hash(value), true);
    }
}
// 覆盖方法，是否包含 URL
public boolean contains(CrawlUrl value) {
    return contains(value.getOriUrl());
}
// 覆盖方法，是否包含 URL
public boolean contains(String value) {
    if (value == null) {
        return false;
    }
    boolean ret = true;
    for (SimpleHash f : func) {
        ret = ret && bits.get(f.hash(value));
    }
    return ret;
}
public static class SimpleHash {
    private int cap;
    private int seed;
    public SimpleHash(int cap, int seed) {
        this.cap = cap;
        this.seed = seed;
```

```
        }
        public int hash(String value) {
            int result = 0;
            int len = value.length();
            for (int i = 0; i < len; i++) {
                result = seed * result + value.charAt(i);
            }
            return (cap - 1) & result;
        }
    }
}
```

如果想知道需要使用多少位才能降低错误概率，可以从表 1.5 所示的存储项目和位数比率估计布隆过滤器的误判率。

表 1.5 布隆过滤器误判率表

| 比率(items:bits) | 误判率(False-positive) |
|---|---|
| 1∶1 | 0.63212055882856 |
| 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。比较保守的实现是，为每个 URL 分配 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 200MB 的空间，并且排重速度超快，一遍下来不到两分钟。

### 1.3.5 详解 Heritrix 爬虫队列

上一节介绍了 Berkeley DB 构建爬虫队列的基础和过程，在许多开源爬虫软件中，都是用 Berkeley DB 来实现爬虫队列。本节，将分析一个常用的开源爬虫软件，以增加读者对爬虫队列的理解。

Heritrix 是一个开源的、可扩展的爬虫项目。它始于 2003 年，最初的目的是开发一个特殊的爬虫，对网上的资源进行归档。在 Heritrix 以及很多开源爬虫软件中，爬虫队列有一个非常好听的名字——Frontier。在 Heritrix 中，Frontier 底层的数据结构也是使用了 Berkeley DB。

Heritrix 中的 Frontier 内部处理机制如图 1.8 所示。

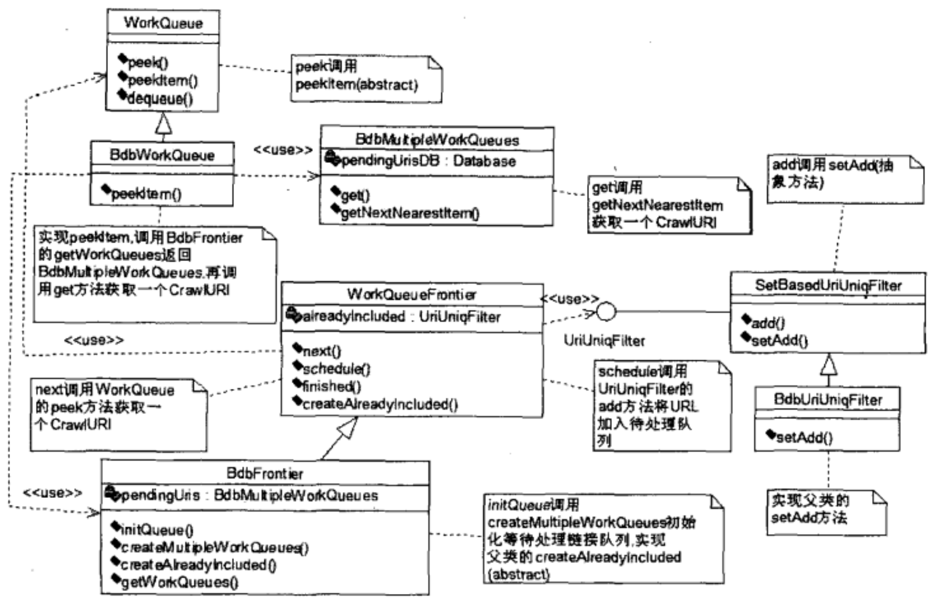



图 1.8　Heritrix 中的 Frontier 架构

### 1. BdbMultipleWorkQueues

它是对 Berkeley DB 的简单封装。在内部有一个 Berkeley Database，存放所有待处理的链接。代码如下：

```
public class BdbMultipleWorkQueues {
    // 存放所有待处理的 URL 的数据库
    private Database pendingUrisDB = null;
    // 由 key 获取一个链接
    public CrawlURI get(DatabaseEntry headKey) throws DatabaseException {
        DatabaseEntry result = new DatabaseEntry();
        // 由 key 获取相应的链接
        OperationStatus status = getNextNearestItem(headKey, result);
        CrawlURI retVal = null;
        if (status != OperationStatus.SUCCESS) {
            LOGGER.severe("See '1219854 NPE je-2.0 "
                    + "entryToObject '. OperationStatus "
                    + " was not SUCCESS: " + status + ", headKey "
                    + BdbWorkQueue.getPrefixClassKey(headKey.getData()));
            return null;
        }
        try {
            retVal = (CrawlURI) crawlUriBinding.entryToObject(result);
        } catch (RuntimeExceptionWrapper rw) {
            LOGGER.log(Level.SEVERE, "expected object missing in queue "
```

```
                + BdbWorkQueue.getPrefixClassKey(headKey.getData()), rw);
            return null;
        }
        retVal.setHolderKey(headKey);
        return retVal;// 返回链接
    }
    // 从等待处理列表中获取一个链接
    protected OperationStatus getNextNearestItem(DatabaseEntry headKey,
            DatabaseEntry result) throws DatabaseException {
        Cursor cursor = null;
        OperationStatus status;
        try {
            // 打开游标
            cursor = this.pendingUrisDB.openCursor(null, null);
            status = cursor.getSearchKey(headKey, result, null);
            if (status != OperationStatus.SUCCESS
                    || result.getData().length > 0) {
                throw new DatabaseException("bdb queue cap missing");
            }
            status = cursor.getNext(headKey, result, null);
        } finally {
            if (cursor != null) {
                cursor.close();
            }
        }
        return status;
    }
    /**
     * 添加 URL 到数据库
     */
    public void put(CrawlURI curi, boolean overwriteIfPresent)
            throws DatabaseException {
        DatabaseEntry insertKey = (DatabaseEntry)curi.getHolderKey();
        if (insertKey == null) {
            insertKey = calculateInsertKey(curi);
            curi.setHolderKey(insertKey);
        }
        DatabaseEntry value = new DatabaseEntry();
        crawlUriBinding.objectToEntry(curi, value);
        if (LOGGER.isLoggable(Level.FINE)) {
            tallyAverageEntrySize(curi, value);
        }
        OperationStatus status;
        if (overwriteIfPresent) {
            // 添加
            status = pendingUrisDB.put(null, insertKey, value);
        } else {
```

```
            status = pendingUrisDB.putNoOverwrite(null, insertKey, value);
        }
        if (status != OperationStatus.SUCCESS) {
            LOGGER.severe("failed; " + status + " " + curi);
        }
    }
}
```

### 2. BdbWorkQueue

代表一个链接队列，该队列中所有的链接都具有相同的键值。它实际上是通过调用BdbMultipleWorkQueues的get方法从等待处理的链接数据库中取得链接的。代码如下：

```
public class BdbWorkQueue extends WorkQueue
implements Comparable, Serializabl
{
    //获取一个URL
    protected CrawlURI peekItem(final WorkQueueFrontier frontier)
    throws IOException {
         // 关键:从BdbFrontier中返回pendingUris
        final BdbMultipleWorkQueues queues = ((BdbFrontier) frontier)
            .getWorkQueues();
        DatabaseEntry key = new DatabaseEntry(origin);
        CrawlURI curi = null;
        int tries = 1;
        while(true) {
            try {
                //获取链接
                curi = queues.get(key);
            } catch (DatabaseException e) {
                LOGGER.log(Level.SEVERE,"peekItem failure; retrying",e);
            }
         ...
    return curi;
    }
}
```

### 3. WorkQueueFrontier

实现了最核心的方法：

```
public CrawlURI next()
    throws InterruptedException, EndedException {
        while (true) {
            long now = System.currentTimeMillis();
            preNext(now);
            synchronized(readyClassQueues) {
                int activationsNeeded = targetSizeForReadyQueues() - rea
```

```
        dyClassQueues.size();
        while(activationsNeeded > 0 && !inactiveQueues.isEmpty()) {
            activateInactiveQueue();
            activationsNeeded--;
        }
    }
    WorkQueue readyQ = null;
    Object key = readyClassQueues.poll(DEFAULT_WAIT,TimeUnit.MIL
    LISECONDS);
    if (key != null) {
        readyQ = (WorkQueue)this.allQueues.get(key);
    }
    if (readyQ != null) {
        while(true) {
            CrawlURI curi = null;
            synchronized(readyQ) {
                /**
                *取出一个 URL，最终从子类 BdbFrontier 的
                 *pendingUris 中取出一个链接
                 */
                curi = readyQ.peek(this);
                if (curi != null) {
                    String currentQueueKey = getClassKey(curi);
                    if (currentQueueKey.equals(curi.getClassKey())) {
                        noteAboutToEmit(curi, readyQ);
                        //加入正在处理队列中
                        inProcessQueues.add(readyQ);
                        return curi; //返回
                    }
                    curi.setClassKey(currentQueueKey);
                    readyQ.dequeue(this);//出队列
                    decrementQueuedCount(1);
                    curi.setHolderKey(null);
                } else {
                    readyQ.clearHeld();
                    break;
                }
            }
            if(curi!=null) {
                sendToQueue(curi);
    }
}
     } else {
         if (key != null) {
             logger.severe("Key "+ key +
                 " in readyClassQueues but not allQueues");
         }
```

```
            }
            if(shouldTerminate) {
                throw new EndedException("shouldTerminate is true");
            }
            if(inProcessQueues.size()==0) {
                this.alreadyIncluded.requestFlush();
            }
        }
    }
//将URL加入待处理队列
    public void schedule(CandidateURI caUri) {
        String canon = canonicalize(caUri);
        if (caUri.forceFetch()) {
            alreadyIncluded.addForce(canon, caUri);
        } else {
            alreadyIncluded.add(canon, caUri);
        }
    }
```

### 4. BdbFrontier

继承了 WorkQueueFrontier，是 Heritrix 唯一具有实际意义的链接工厂。代码如下：

```
public class BdbFrontier extends WorkQueueFrontier implements Serializable
{
    /** 所有待抓取的链接*/
    protected transient BdbMultipleWorkQueues pendingUris;

    //初始化pendingUris，父类为抽象方法
    protected void initQueue() throws IOException {
        try {
            this.pendingUris = createMultipleWorkQueues();
        } catch(DatabaseException e) {
            throw (IOException)new IOException(e.getMessage()).initCause(e);
        }
    }
   private BdbMultipleWorkQueues createMultipleWorkQueues()
    throws DatabaseException {
        return new BdbMultipleWorkQueues(this.controller.getBdbEnvironment(),
            this.controller.getBdbEnvironment().getClassCatalog(),
            this.controller.isCheckpointRecover());
    }
    protected BdbMultipleWorkQueues getWorkQueues() {
```

```
        return pendingUris;
    }
...
}
```

### 5. BdbUriUniqFilter

它实际上是一个过滤器，用来检查一个要进入等待队列的链接是否已经被抓取过。
方法代码如下：

```
//添加 URL
protected boolean setAdd(CharSequence uri) {
        DatabaseEntry key = new DatabaseEntry();
        LongBinding.longToEntry(createKey(uri), key);
        long started = 0;
        OperationStatus status = null;
        try {
            if (logger.isLoggable(Level.INFO)) {
                started = System.currentTimeMillis();
            }
            //添加到数据库
      status = alreadySeen.putNoOverwrite(null, key, ZERO_LENGTH_ENTRY);
            if (logger.isLoggable(Level.INFO)) {aggregatedLookupTime +=
                    (System.currentTimeMillis() - started);
        }
    } catch (DatabaseException e) {
        logger.severe(e.getMessage());
    }
    if (status == OperationStatus.SUCCESS) {
        count++;
        if (logger.isLoggable(Level.INFO)) {
            final int logAt = 10000;
            if (count > 0 && ((count % logAt) == 0)) {
                logger.info("Average lookup " +
                    (aggregatedLookupTime / logAt) + "ms.");
                aggregatedLookupTime = 0;
            }
        }
    }
    //如果存在，返回 false
    if(status == OperationStatus.KEYEXIST) {
        return false;
    } else {
        return true;
    }
}
```

## 1.4 设计爬虫架构

上一节讲述了爬虫队列。本节要介绍如何设计爬虫架构。

一个设计良好的爬虫架构必须满足如下需求。

(1) 分布式：爬虫应该能够在多台机器上分布执行。

(2) 可伸缩性：爬虫结构应该能够通过增加额外的机器和带宽来提高抓取速度。

(3) 性能和有效性：爬虫系统必须有效地使用各种系统资源，例如，处理器、存储空间和网络带宽。

(4) 质量：鉴于互联网的发展速度，大部分网页都不可能及时出现在用户查询中，所以爬虫应该首先抓取有用的网页。

(5) 新鲜性：在许多应用中，爬虫应该持续运行而不是只遍历一次。

(6) 更新：因为网页会经常更新，例如论坛网站会经常有回帖。爬虫应该取得已经获取的页面的新的拷贝。例如一个搜索引擎爬虫要能够保证全文索引中包含每个索引页面的较新的状态。对于搜索引擎爬虫这样连续的抓取，爬虫访问一个页面的频率应该和这个网页的更新频率一致。

(7) 可扩展性：为了能够支持新的数据格式和新的抓取协议，爬虫架构应该设计成模块化的形式。

### 1.4.1 爬虫架构

爬虫的简化版本架构如图 1.9 所示。

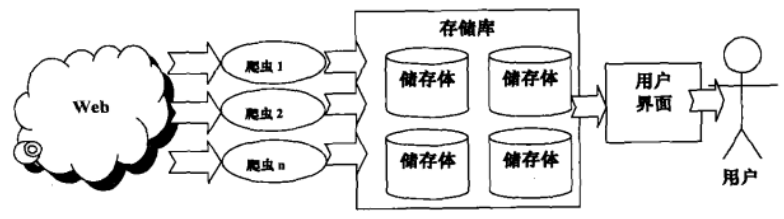



图 1.9 爬虫物理分布简化架构图

这里最主要的关注对象是爬虫和存储库。其中的爬虫部分阶段性地抓取互联网上的内容。存储库存储爬虫下载下来的网页，是分布式的和可扩展的存储系统。在往存储库中加载新的内容时仍然可以读取存储库。

实际的爬虫逻辑架构如图 1.10 所示。

其中：

(1) URL Frontier 包含爬虫当前待抓取的 URL(对于持续更新抓取的爬虫，以前已经抓取过的 URL 可能会回到 Frontier 重抓)。

(2) DNS 解析模块根据给定的 URL 决定从哪个 Web 服务器获取网页。

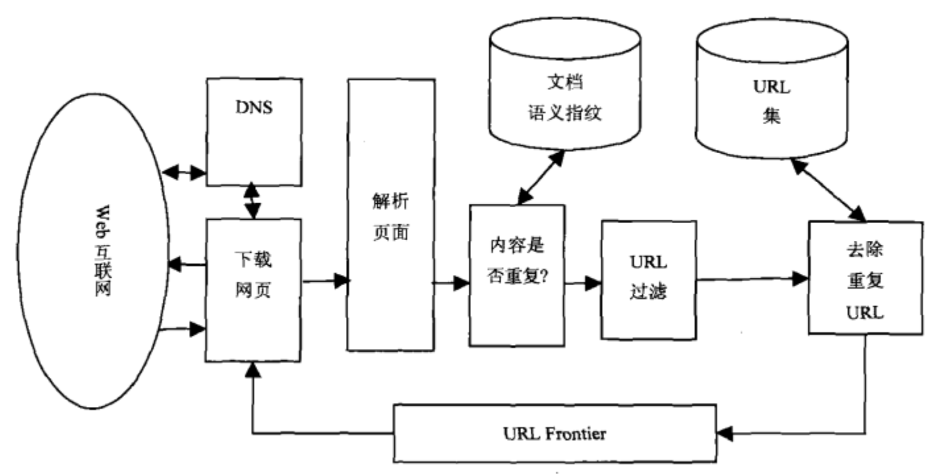



图 1.10　单线程爬虫结构

(3) 获取模块使用 HTTP 协议获取 URL 代表的页面。

(4) 解析模块提取文本和网页的链接集合。

(5) 重复消除模块决定一个解析出来的链接是否已经在 URL Frontier 或者最近下载过。

DNS 解析是网络爬虫的瓶颈。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。如果我们的目标是一秒钟抓取数百个文件，这样就达不到性能要求。一个标准的补救措施是引入缓存：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。然而，由于抓取礼貌的限制，降低了 DNS 缓存的命中率。

用 DNS 解析还有一个难点：在标准库中实现的查找是同步的。这意味着一旦一个请求发送到 DNS 服务器上，在那个节点上的其他爬虫线程也被阻塞直到第一个请求完成。为了避免这种情况发生，许多爬虫自己来实现 DNS 解析。执行解析代码的线程 i 将发送一个消息到 DNS 服务器，然后执行一个定时等待，不超过这个设定的时间段，这个线程会继续执行。一个单独的 DNS 线程侦听标准的 DNS 端口(53 端口)从名称服务器传入的响应数据包。一旦接受一个响应，它就激活对应的爬虫线程 i 并把响应数据包交给 i。如果 i 因为等待超时还没有恢复运行，则爬虫线程如果尝试 5 次全都失败，下次发送一个新的消息给 DNS 服务等待的时间会延长一倍。鉴于有的主机名需要长达几十秒的时间来解析，所以等待 DNS 解析的时间范围可以为 1～90 秒。

## 1.4.2　设计并行爬虫架构

整个爬虫系统可以由一台抓取机器或多个爬虫节点组成。多机并行抓取的分布式系统则需要考虑节点之间的通信和调度，关于分布式爬虫系统将在第 2 章介绍。在一个爬虫节点上实现并行抓取，可以考虑多线程同步 I/O 或者单线程异步 I/O。多线程爬虫需要考虑线程之间的同步问题。

对单线程并行抓取来说，异步 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，因此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。

Java 6 版本开始引入的 NIO 包，通过 Selectors 类提供了非阻塞式的 I/O。Java 7 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 通过队列模型提供异步读/写。
- 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。

JDK 1.6 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，主要用到的就是下面两个类。

- 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 Niocchi 中使用 NIO 下载网页的例子，首先发送请求：

```
SocketChannel sc = SocketChannel.open();
sc.configureBlocking( false );
sc.socket().setTcpNoDelay(true);

boolean connected = false;
try
{
    connected = sc.connect( query_.getInetSocketAddress() );
```

```
}
catch( IOException e )
{
    _logger.warn( "IOException " + query_.getURL(), e );
    query_.setStatus(INTERNAL_ERROR);
    processCrawledQuery( query_ );
    return;
}

if( connected ) //检查连接是否建立
{
    if( ! sendQuery( query_, sc ) )
    {
    query_.setStatus(UNREACHABLE);
        processCrawledQuery( query_ );
        return ;
    }

    sc.register( _selector, SelectionKey.OP_READ, query_ );
}
else
{
    sc.register( _selector, SelectionKey.OP_CONNECT, query_ );
}
```

然后接收数据：

```
int i = _selector.select( _selectTimeout );
select_total_time += new Date().getTime() - t;

if( i == 0 )
{
    _logger.debug( "Select timeout" );

//取消连接
    Set keys = _selector.keys();
    Iterator it = keys.iterator();
    while( it.hasNext() )
    {
        SelectionKey key = (SelectionKey)it.next();
        Query query = (Query) key.attachment();
        if (_logger.isDebugEnabled())
            _logger.debug( "Select timeout for '" + query.getURL() + "'" );
        query.setStatus(TIMEOUT);
        processKey( key );
    }
```

```
        if( ! _resolver_queue.hasNextQuery() && (_reg_count == 0) &&
         (!_redirection_resolver_queue.hasMore()) )
        {
            break;    // 没有更多 URL
        }

        continue;
    }

    // 检查是否超时
    long time = System.currentTimeMillis();
    Set keys = _selector.keys();
    Iterator it = keys.iterator();
    boolean some_timeout = false;
    while( it.hasNext() )
    {
        SelectionKey key = (SelectionKey)it.next();
        Query query = (Query) key.attachment();
        if( time - query.getRegisterTime() > _timeout )
        {
            if (_logger.isDebugEnabled())
                _logger.debug( "Timeout for '" + query.getURL() + "'");
            some_timeout = true;

            query.setStatus(TIMEOUT);
            processKey( key );
        }
    }
    if( some_timeout )
    {
        i = _selector.selectNow();
        if( i == -1 ) continue;
    }

    Set skeys = _selector.selectedKeys();
    it = skeys.iterator();

    while( it.hasNext() )
    {
        SelectionKey key = (SelectionKey)it.next();
        it.remove();
        if( (key.readyOps() & SelectionKey.OP_READ) ==
            SelectionKey.OP_READ )
        {
            SocketChannel sc = (SocketChannel)key.channel();
            Exception e = null;
            t = new Date().getTime();
```

```
Query query = (Query) key.attachment();
Resource resource = query.getResource();

int r = 0;
 try
{
    r = resource.read( sc );
    read_total_time += new Date().getTime() - t;
    if (_logger.isTraceEnabled())
        _logger.trace( "read " + r + "for URL '" +
        query.getURL() + "'" );
}
catch( IOException ee )
{
    _logger.warn(."For URL '" + query.getURL() + "' " +
    ee.getMessage() );
    e = ee;
}
catch( ResourceException ee )
{
    _logger.warn( "For URL '" + query.getURL() + "' " +
    ee.getMessage() );
    e = ee;
}

   // 资源完整的被抓取下来
   if (e != null)
     query.setStatus(INCOMPLETE);

   if ( r < 0 && e == null) {
       if( resource.getHTTPStatus() == 200 )
       query.setStatus(CRAWLED);
       else query.setStatus(HTTPERROR);
   }

   // 如果抓取过程出错
   if( e != null || r < 0 )
   {
       processKey( key );
   }
}
else if( (key.readyOps() & SelectionKey.OP_CONNECT) ==
        SelectionKey.OP_CONNECT )
{
   SocketChannel sc = (SocketChannel)key.channel();
   Query query = (Query) key.attachment();
   try
```

```
            {
                sc.finishConnect();
            }
            catch( IOException e )
            {
                _logger.warn( "Connection error to '" + query.getURL()
                + '\'' );
                processKey( key );
                continue;
            }

            if( ! sendQuery( query, sc ) )
            {
                processKey( key );
                query.setStatus(UNREACHABLE);
                continue;
            }


            sc.register( _selector, SelectionKey.OP_READ, query );
        }
    }
```

### 1.4.3 详解 Heritrix 爬虫架构

上一节开始了一个开源爬虫软件 Heritrix。现在，我们来看一下它的架构是如何设计的。

Heritrix 采用的是模块化的设计，各个模块是由一个控制器类(CrawlController 类)来协调的，因此控制器是它的核心。CrawlController 类结构如图 1.11 所示。

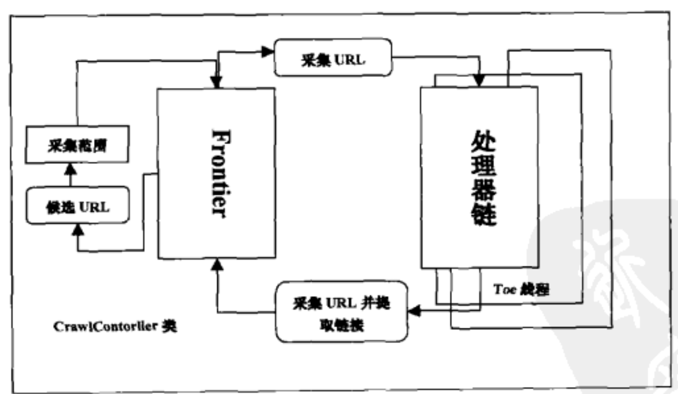



图 1.11 CrawlContorller 类结构

CrawlController 类是整个爬虫的总控制者，控制整个抓取工作的起点，决定整个抓取任务的开始和结束。CrawlController 从 Frontier 获取 URL，传递给线程池(ToePool)中的 ToeThread 处理。

Frontier(边界控制器)主要确定下一个将被处理的 URL，负责访问的均衡处理，避免对某一 Web 服务器造成太大的压力。Frontier 保存着爬虫的状态，包括已经找到的 URI、正在处理中的 URI 和已经处理过的 URI。

Heritrix 是按多线程方式抓取的爬虫，主线程把任务分配给 Teo 线程(处理线程)，每个 Teo 线程每次处理一个 URL。Teo 线程对每个 URL 执行一遍 URL 处理器链。URL 处理器链包括如下 5 个处理步骤。

(1) 预取链：主要是做一些准备工作，例如，对处理进行延迟和重新处理，否决随后的操作。

(2) 提取链：主要是下载网页，进行 DNS 转换，填写请求和响应表单。

(3) 抽取链：当提取完成时，抽取感兴趣的 HTML 和 JavaScript，通常那里有新的要抓取的 URL。

(4) 写链：存储抓取结果，可以在这一步直接做全文索引。Heritrix 提供了用 ARC 格式保存下载结果的 ARCWriterProcessor 实现。

(5) 提交链：做和此 URL 相关操作的最后处理。检查哪些新提取出的 URL 在抓取范围内，然后把这些 URL 提交给 Frontier。另外还会更新 DNS 缓存信息。

处理 URL 的流程如图 1.12 所示。在实现上，所有的处理类都是一个名称为 Processor 类的子类。例如，PreconditionEnforcer 类继承了 Processor。

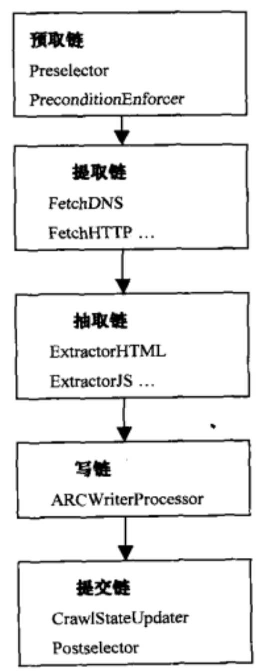



图 1.12　Heritrix 处理链图

Frontier 记录哪些 URI 被预订采集和哪些 URI 已经被采集，并选择下一个 URI，剔除已经处理的 URI。处理器链包含若干处理器获取的 URI，分析结果，并将它们传回给 Frontier。

服务器缓存(Server cache)存放服务器的持久信息，能够被爬行部件随时查到，包括被抓取的 Web 服务器信息，例如 DNS 查询结果，也就是 IP 地址。

分析完 Controller 之后，我们就要看看 Heritrix 软件的整体架构，如图 1.13 所示。

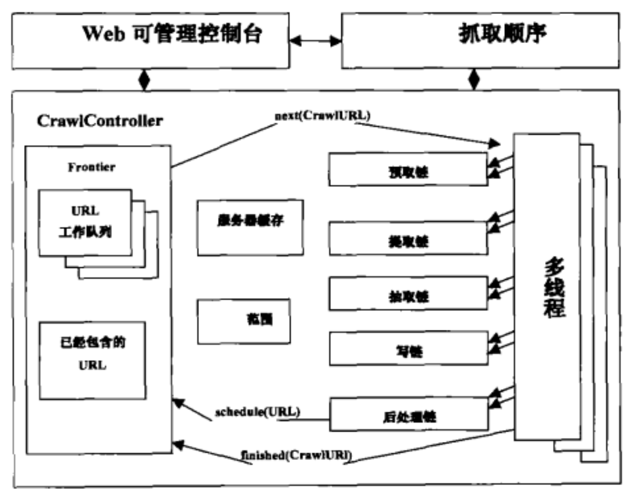



图 1.13　Heritrix 软件的整体结构

图 1.13 把 Heritrix 分成三部分：

- Web 可管理控制台。可以在界面设置运行时使用哪个模块。Heritrix 也是因为有良好的管理界面，所以得到了广泛的应用。Web 管理界面默认运行在 Heritrix 安装包自带的 Java HTTP 服务器 Jetty 中，但也可以作为 Web 应用运行在 Tomcat 或 Resin 等 Web 服务器中。操作者可以通过选择 Crawler 命令来操作控制台。
- 抓取顺序配置文件。可以在配置文件 ORDER.XML 中指定抓取的顺序。
- 总控整个爬虫的 CrawlController。

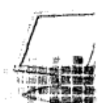
Heritrix 包含以下关键特性：

- 用单个爬虫在多个独立的站点一直不断地抓取。
- 从一个种子 URL 开始爬，不断抓取页面所指向的 URL。
- 主要是用宽度优先算法进行处理。
- 主要部件都是高效的和可扩展的。

可以配置的部分包括：

- 可设置输出日志、归档文件和临时文件的位置。
- 可设置下载的最大字节，最大数量的下载文档和最大的下载时间。
- 可设置工作线程数量。
- 可设置所利用的带宽的上界。
- 可在设置之后一定时间重新选择。
- 包含一些可设置的过滤机制、表达方式、URI 路径深度选择等。

尽管 Heritrix 是设计良好的爬虫，但是它的局限包括：



- 不支持多机分布式抓取。
- 在有限的机器资源的情况下，却要复杂的操作。
- 只有官方支持，仅仅在 Linux 上进行了测试。
- 每个爬虫是单独进行工作的，没有对更新进行修订。
- 在硬件和系统失败时，恢复能力很差。
- 性能还不够优化。

本节，我们介绍了如何设计一个爬虫架构，并且详细讲述了一个开源爬虫——Heritrix 的爬虫结构。为读者今后开发自己的爬虫提供了整体上的参考。

## 1.5　使用多线程技术提升爬虫性能

上一节讲述爬虫架构时曾经提到过，为了提升爬虫性能，需要采用多线程的爬虫技术。并且开源软件 Heritrix 已经采用了多线程的爬虫技术来提高性能。而且很多大型网站都采用多个服务器镜像的方式提供同样的网页内容。采用多线程并行抓取能同时获取同一个网站的多个服务器中的网页，这样能极大地减少抓取这类网站的时间。

### 1.5.1　详解 Java 多线程

#### 1. 创建多线程的方法

多线程是一种机制，它允许在程序中并发执行多个指令流，每个指令流都称为一个线程，彼此间互相独立。

线程又称为轻量级进程，它和进程一样拥有独立的执行控制，由操作系统负责调度，区别在于线程没有独立的存储空间，而是和所属进程中的其他线程共享存储空间，这使得线程间的通信较进程简单。

多个线程的执行是并发的，即在逻辑上是“同时”的。如果系统只有一个 CPU，那么真正的“同时”是不可能的，但是由于 CPU 切换的速度非常快，用户感觉不到其中的区别，因此用户感觉到线程是同时执行的。

为了创建一个新的线程，需要做哪些工作呢？很显然，必须指明这个线程所要执行的代码，在 Java 语言中，通过 JDK 提供的 java.lang.Thread 类或者 java.lang.Runable 接口，能够轻松地添加线程代码。

具体如何实现线程执行的代码呢？先看一看 java.lang.Thread 类。java.lang.Thread 类最重要的方法是 run()，它被 java.lang.Thread 类的方法 start()所调用，提供线程所要执行的代码。也就是说，run()方法里面的代码就是线程执行时所运行的代码。

再来看 java.lang.Runable 接口，这个接口中只有一个 run()方法，和 java.lang.Thread 类中的 run()方法类似，java.lang.Runable 接口中的 run()方法中的代码就是线程执行的代码。

因此，在 Java 语言中，要创建一个线程，可以使用以下两种方法。

方法一：继承 java.lang.Thread 类，覆盖方法 run()，在创建的 java.lang.Thread 类的子类中重写 run()方法。下面是一个例子：

```
public class MyThread extends Thread
{
    int count= 1, number;
    public MyThread(int num)
    {
        number = num;
        System.out.println("创建线程 " + number);
    }
    public void run() {
        while(true) {
                    System.out.println("线程 " + number + ":计数 " + count);
                    if(++count== 6) return;
                    }
    }
    public static void main(String args[]){
        for(int i = 0; i<5; i++)
            new MyThread(i+1).start();
    }
}
```

这种方法简单明了，但是，它也有一个很大的缺陷，如果线程类 MyThread 已经从一个类继承(如小程序必须继承自 Applet 类)，而无法再继承 java.lang.Thread 类时应该怎么办呢？这时，就必须使用下面的方法二来实现线程。

方法二：实现 java.lang.Runnable 接口

java.lang.Runnable 接口只有一个 run()方法，库创建一个类实现 java.lang.Runnable 接口并提供这一方法的实现，将线程代码写入 run()方法中，并且新建一个 java.lang.Thread 类，将实现 java.lang.Runnable 的类作为参数传入，就完成了创建新线程的任务。下面是一个例子：

```
public class MyThread implements Runnable
{
    int count= 1, number;
    public MyThread(int num){
        number = num;
        System.out.println("创建线程 " + number);
    }
    public void run(){
        while(true){
            System.out.println("线程 " + number + ":计数 " + count);
            if(++count== 6) return;
        }
    }
    public static void main(String args[]){
        for(int i = 0; i < 5; i++)
```

```
            new Thread(new MyThread(i+1)).start();
        }
    }
```

严格地说，创建 java.lang.Thread 子类的实例也是可行的，但必须注意的是，该子类不能覆盖 java.lang.Thread 类的 run() 方法，否则该线程执行的将是子类的 run() 方法，而不是执行实现 java.lang.Runnable 接口的类的 run() 方法，对此读者不妨试验一下。

方法二使得能够在一个类中包容所有的代码，有利于封装。这种方法的缺点在于，只能使用一套代码，若想创建多个线程并使各个线程执行不同的代码，则必须额外创建类，如果这样的话，在大多数情况下也许还不如直接用多个类分别继承 java.lang.Thread 来得紧凑。

### 2. Java 语言对线程同步的支持

首先讲解线程的状态，一个线程具有如下四种状态。

(1) 新状态：线程已被创建但尚未执行(start()方法尚未被调用)。

(2) 可执行状态：线程可以执行，但不一定正在执行。CPU 时间随时可能被分配给该线程，从而使得它执行。

(3) 死亡状态：正常情况下，run()方法返回使得线程死亡。调用 java.lang.Thread 类中的 stop()或 destroy()方法亦有同样效果，但是不推荐使用这两种方法，前者会产生异常，后者是强制终止，不会释放锁。

(4) 阻塞状态：线程不会被分配 CPU 时间，无法执行。

编写多线程程序通常会遇到线程的同步问题，什么是线程同步问题呢？

由于同一进程的多个线程共享存储空间，在带来方便的同时，也会带来访问冲突这个严重的问题。Java 语言提供了专门机制以解决这种冲突，有效地避免了同一个数据对象被多个线程同时访问的问题。

Java 语言中解决线程同步问题是依靠 synchronized 关键字来实现的，它包括两种用法：synchronized 方法和 synchronized 块。

(1) synchronized 方法。

通过在方法声明中加入 synchronized 关键字来声明该方法是同步方法，即多线程执行的时候各个线程之间必须顺序执行，不能同时访问该方法。如：

```
public synchronized void accessVal(int newVal);
```

在 Java 语言中，每个对象都拥有一把锁，当执行这个对象的 synchronized 方法时，必须获得该对象的锁方能执行，否则所属线程阻塞。而 synchronized 方法一旦执行，就独占该锁，直到从 synchronized 方法返回时才将锁释放，之后被阻塞的线程方能获得该锁，重新进入可执行状态。

这种对象锁机制确保了同一时刻对于每一个对象，其所有声明为 synchronized 的方法中至多只有一个处于可执行状态，从而有效地避免了类成员变量的访问冲突。

在 Java 中，不光是对象，每一个类也对应一把锁，因此也可将类的静态成员函数声明为 synchronized，以控制其对类的静态成员变量的访问。

synchronized 方法的缺陷：若将一个执行时间较长的方法声明为 synchronized，将会大

大影响程序运行的效率。因此 Java 为我们提供了更好的解决办法，那就是 synchronized 块。

(2) synchronized 块。

通过 synchronized 关键字来声明 synchronized 块。语法如下：

```
synchronized(syncObject)
{
//允许访问控制的代码
}
```

synchronized 块是这样一种代码块，块的代码必须获得 syncObject 对象(如前所述，可以是类实例或类)的锁才能执行。由于 synchronized 块可以是任意代码块，且可任意指定上锁的对象，因此灵活性较高。

**3. Java 语言对线程阻塞的支持**

讲完了线程的同步机制，下面介绍 Java 语言对线程阻塞机制的支持。

阻塞指的是暂停一个线程的执行以等待某个条件发生(如等待资源就绪)，学过操作系统的读者对它一定非常熟悉了。Java 提供了大量方法来支持阻塞，下面逐一分析。

(1) sleep()方法：sleep()允许指定以毫秒为单位的一段时间作为参数，它使得线程在指定的时间内进入阻塞状态，不能得到 CPU 时间片，指定的时间一过，线程重新进入可执行状态。例如，当线程等待某个资源就绪时，测试发现条件不满足后，让线程 sleep()一段时间后重新测试，直到条件满足为止。

(2) suspend()和 resume()方法：两个方法配套使用，suspend()使线程进入阻塞状态，并且不会自动恢复，必须对其应用 resume()方法，才能使得线程重新进入可执行状态。例如，当前线程等待另一个线程产生的结果时，如果发现结果还没有产生，会调用 suspend()方法，另一个线程产生了结果后，调用 resume() 使其恢复。

(3) yield()方法：yield()方法使得线程放弃当前分得的 CPU 时间片，但不使线程阻塞，即线程仍处于可执行状态，随时可能再次分得 CPU 时间。

(4) wait()和 notify()方法：两个方法配套使用，wait()可以使线程进入阻塞状态，它有两种形式，一种允许指定以毫秒为单位的一段时间作为参数，另一种没有参数。前者当对应的 notify()被调用或者超出指定时间时，线程重新进入可执行状态；后者则必须在对应的 notify()被调用时，线程才重新进入可执行状态。

在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其他更多资源。线程对象也不例外。当前，比较流行的一种技术是“池化技术”，即在系统启动的时候一次性创业多个对象并且保存在一个“池”中，当需要使用的时候直接从“池”中取得而不是重新创建。这样可以大大提高系统性能。

Java 语言在 JDK 1.5 以后的版本中提供了一个轻量级线程池——ThreadPool。可以使用线程池来执行一组任务。简单的任务没有返回值，如果主线程需要获得子线程的返回值时，可以使任务实现 Callable 接口，线程池执行任务并通过 Future 的实例返回线程的执行结果。Callable 和 java.lang.Runnable 的区别如下：

- Callable 定义的方法是 call()，而 Runnable 定义的方法是 run()。

- Callable 的 call()方法可以有返回值，而 Runnable 的 run()方法不能有返回值。
- Callable 的 call()方法可以抛出异常，而 Runnable 的 run()方法不能抛出异常。

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。

下面的例子使用 ThreadPool 实现并行下载网页。在继承 Callable 方法的任务类中下载网页的实现如下：

```
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
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。

### 1.5.2 爬虫中的多线程

多线程爬虫的结构如图 1.14 所示。

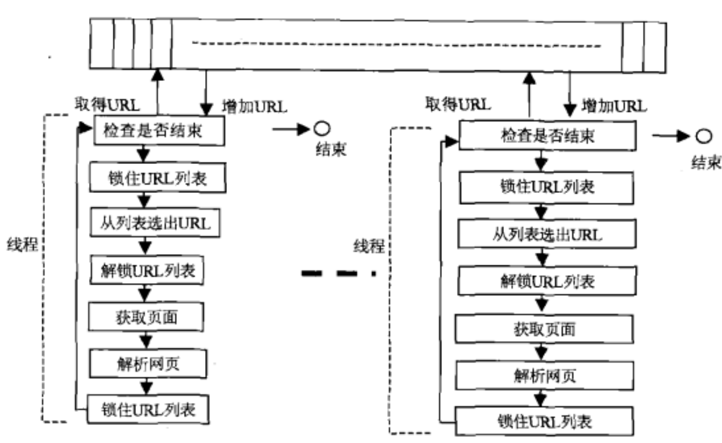



**图 1.14　多线程爬虫的结构**

对于并行爬虫架构而言，处理空队列要比序列爬虫更加复杂。空的队列并不意味着爬虫已经完成了工作，因为此刻其他的进程或者线程可能依然在解析网页，并且马上会加入新的 URL。进程或者线程管理员需要给报告队列为空的进程/线程发送临时的休眠信号来解决这类问题。线程管理员需要不断跟踪休眠线程的数目；只有当所有的线程都休眠的时候，爬虫才可以终止。

## 1.5.3　一个简单的多线程爬虫实现

以下是一个多线程爬虫程序的主线程部分和子线程部分，主线程启动子线程并等待所有子线程执行完成后才退出，实际代码如下：

```
threadList = new ArrayList<Thread>(THREAD_NUM);
for (int i = 0; i < THREAD_NUM; i++) {
    Thread t = new Thread(this, "Spider Thread #" + (i+1));
    t.start();
    threadList.add(t);
}
//当前线程等待子线程退出
while (threadList.size() > 0) {
    Thread child = (Thread)threadList.remove(0);
    child.join(); //等待这个线程执行完
}
```

子线程主要的执行程序如下：

```
//从 TODO 取出要分析的 URL 地址，同时把它放入 Visited 表
    public synchronized NewsSource dequeueURL() throws Exception {
      while (true) {
         if (!todo.isEmpty()) {
             NewsSource newitem = (NewsSource)todo.removeFirst();
             visited.add(newitem.URL,newitem.source);
```

```
                return newitem;
            }
            else {
                threads--; //等待线程数的计数器减 1
                if (threads > 0) {//如果仍然有其他的线程在活动则等待
                    wait();
                    threads++;//等待线程数的计数器加 1
                }
                else {//如果其他线程都在等待，则通知所有在等待的线程集体退出
                    notifyAll();
                    return null;
                }
            }
        }
    }
// enqueueURL 把新发现的 URL 放入 TODO 表
    public synchronized void enqueueURL(NewsSource newitem) {
        if (!visited.contains(newitem.URL)) {
            todo.add(newitem);
            visited.add(newitem.URL,newitem.source);
            notifyAll();//唤醒在等待的线程
        }
    }
    public void run() {
        NewsSource item;
        try {
            while ((item = dequeueURL()) != null) {
                indexURL(item);//包含把新的 URL 放入 TODO 表的过程
            }
        }
        catch(Exception e) {
            e.printStackTrace();
        }
        threads--;
    }
```

### 1.5.4 详解 Heritrix 多线程结构

要想更有效、更快速地抓取网页内容，则必须采用多线程抓取。开源软件 Heritrix 采用传统的线程模型实现了一个标准的线程池 ToePool，它用于管理所有的抓取线程。ToeThread 则继承了 Thread 类，并实现了 run 方法。

ToePool 和 ToeThread 都位于 org.archive.crawler.framework 包中。ToePool 的初始化，是在 CrawlController 的 initialize()方法中完成的。以下是在 CrawlController 中用于对 ToePool 进行初始化的代码：

```
//构造函数
toePool = new ToePool(this);
```

```
// 按 order.xml 中的配置，实例化并启动线程
toePool.setSize(order.getMaxToes());
```

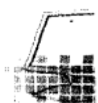
ToePool 的构造函数很简单，如下所示：

```
public ToePool(CrawlController c) {
    super("ToeThreads");
    this.controller = c;
}
```

它仅仅是调用了父类 java.lang.ThreadGroup 的构造函数，同时，将注入的 CrawlController 赋给类变量。这样，便建立起了一个线程池的实例了。

真正的工作线程在线程池中的 setSize(int)方法中创建。从名称上看，这个方法很像是一个普通的赋值方法，但实际上，这个方法调整了抓取的线程数量。代码如下：

```
public void setSize(int newsize) {
 targetSize = newsize;
 int difference = newsize - getToeCount();
 // 如果发现线程池中的实际线程数量小于应有的数量
 // 则启动新的线程
 if (difference > 0) {
     for(int i = 1; i <= difference; i++) {
      // 启动新线程
        startNewThread();
    }
 }
 // 如果线程池中的线程数量已经达到需要
 else {
     int retainedToes = targetSize;
     // 将线程池中的线程管理起来放入数组中
     Thread[] toes = this.getToes();
     // 循环去除多余的线程
     for (int i = 0; i < toes.length ; i++) {
         if(!(toes[i] instanceof ToeThread)) {
             continue;
         }
         retainedToes--;
         if (retainedToes>=0) {
             continue;
         }
         ToeThread tt = (ToeThread)toes[i];
         tt.retire(); // ToeThread 中定义的方法，通知这个线程尽早结束
     }
 }
}

// 用于取得所有属于当前线程池的线程
private Thread[] getToes() {
```



```
    Thread[] toes = new Thread[activeCount()+10];
    // 由于ToePool继承自java.lang.ThreadGroup类
    // 因此当调用enumerate(Thread[] toes)方法时,
    // 实际上是将该ThreadGroup中开辟的所有线程放入
    // toes这个数组中，以备后面的管理
    this.enumerate(toes);
    return toes;
}
// 开启一个新线程
private synchronized void startNewThread() {
    ToeThread newThread = new ToeThread(this, nextSerialNumber++);
    newThread.setPriority(DEFAULT_TOE_PRIORITY); //设置线程优先级
    newThread.start(); //启动线程
}
```

根据上面的代码可以得出这样的结论：线程池本身在创建的时候，并没有任何活动的线程实例，只有当它的setSize方法被调用时，才创建新线程；如果当setSize方法被调用多次而传入不同的参数时，线程池会根据参数里设定的值的大小来改变池中所管理的线程数量。当启动Toe线程后，执行的是其run()方法中的代码。通过run方法中的代码可以看到ToeThread到底如何处理从Frontier中获得的要抓取的链接。

```
public void run() {
    String name = controller.getOrder().getCrawlOrderName();
    logger.fine(getName()+" started for order '"+name+"'");
    try {
        while ( true ) {
        // 检查是否应该继续处理
            continueCheck();
            setStep(STEP_ABOUT_TO_GET_URI);
            // 使用Frontier的next方法从Frontier中取出下一个要处理的链接
            CrawlURI curi = controller.getFrontier().next();
            // 同步当前线程
            synchronized(this) {
                continueCheck();
                setCurrentCuri(curi);
            }
            /*
             * 处理取出的链接
             */
            processCrawlUri();
            setStep(STEP_ABOUT_TO_RETURN_URI);
            // 检查是否应该继续处理
            continueCheck();
            // 使用Frontier的finished()方法来对刚才处理的链接做收尾工作
            // 比如将分析得到的新的链接加入到等待队列中
            synchronized(this) {
                controller.getFrontier().finished(currentCuri);
```

```
                setCurrentCuri(null);
            }
            // 后续的处理
            setStep(STEP_FINISHING_PROCESS);
            lastFinishTime = System.currentTimeMillis();
            //释放链接
            controller.releaseContinuePermission();
            if(shouldRetire) {
                break; // from while(true)
            }
        }
    } catch (EndedException e) {
    } catch (Exception e) {
        logger.log(Level.SEVERE,"Fatal exception in "+getName(),e);
    } catch (OutOfMemoryError err) {
        seriousError(err);
    } finally {
        controller.releaseContinuePermission();
    }
    setCurrentCuri(null);
    // 清理缓存数据
    this.httpRecorder.closeRecorders();
    this.httpRecorder = null;
    localProcessors = null;
    logger.fine(getName()+" finished for order '"+name+"'");
    setStep(STEP_FINISHED);
    controller.toeEnded();
    controller = null;
}
```

工作线程通过调用 Frontier 的 next()方法取得下一个待处理的链接，然后对链接进行处理，并调用 Frontier 的 finished()方法来收尾、释放链接，最后清理缓存、终止单步工作等。另外，其中还有一些日志操作，主要是为了记录每次抓取的各种状态。以上代码中，最重要的语句是 processCrawlUri()，它调用处理链对链接进行处理。

## 1.6 本章小结

本节介绍了爬虫的基本原理及开源爬虫实现 Heritrix。在本节的末尾，再介绍几个相关的爬虫项目，供读者参考。

- RBSE 是第一个发布的爬虫。它有两个基础程序。第一个程序“spider”，抓取队列中的内容到一个关系数据库中；第二个程序“mite”，是一个修改后的 WWW 的 ASCII 浏览器，负责从网络上下载页面。
- WebCrawler 是第一个公开可用的，用来建立全文索引的一个子程序，它使用 WWW 库下载页面，使用宽度优先算法来解析获得 URL 并对其进行排序，并包括一个根

据选定文本和查询相似程度爬行的实时爬虫。

- World Wide Web Worm 是一个用来为文件建立包括标题和 URL 简单索引的爬虫。索引可以通过 grep 式的 Unix 命令来搜索。
- CobWeb 使用了一个中央“调度者”和一系列的“分布式的搜集者”的爬虫框架。搜集者解析下载的页面并把找到的 URL 发送给调度者，然后调度者反过来分配给搜集者。调度者使用深度优先策略，并且使用平衡礼貌策略来避免服务器超载。爬虫是使用 Perl 语言编写的。
- Mercator 是一个分布式的，模块化的使用 Java 语言编写的网络爬虫。它的模块化源自于使用可互换的“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)，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 URL，其他处理模块可以用来检索文本页面，或者搜集网络数据。
- WebFountain 是一个与 Mercator 类似的分布式的模块化的爬虫，但是使用 C++语言编写的。它的特点是一个管理员机器控制一系列的蚂蚁机器。经过多次下载页面后，页面的变化率可以推测出来。这时，一个非线性的方法必须用于求解方程以获得一个最大的新鲜度的访问策略。作者推荐在早期检索阶段使用这个爬虫，然后用统一策略检索，就是所有页面都使用相同的频率访问。
- PolyBot 是一个使用 C++和 Python 语言编写的分布式网络爬虫。它由一个爬虫管理者，一个或多个下载者，和一个或多个 DNS 解析者组成。抽取到的 URL 被添加到硬盘的一个队列里面，然后使用批处理的模式处理这些 URL。
- WebRACE 是一个使用 Java 实现的，拥有检索模块和缓存模块的爬虫，它是一个很通用的称作 eRACE 的系统的一部分。系统从用户方得到下载页面的请求，爬虫的行为有点像一个聪明的代理服务器。系统还监视订阅网页的请求，当网页发生改变的时候，它必须使爬虫下载更新这个页面并且通知订阅者。WebRACE 最大的特色是，当大多数爬虫都从一组 URL 开始的时候，WebRACE 可以连续地接收初始抓取的 URL 地址。
- Ubicrawer 是一个使用 Java 语言编写的分布式爬虫。它没有中央程序，但有一组完全相同的代理组成，分配功能通过主机前后一致的散列计算进行。这里没有重复的页面，除非爬虫崩溃了(然后，另外一个代理就会接替崩溃的代理重新开始抓取)。爬虫设计为高伸缩性。
- FAST Crawler 是一个分布式的爬虫，在 Fast Search & Transfer 中使用。节点之间只交换发现的链接。在抓取任务分配上，静态的映射超级链接到爬虫机器。实现了增量式抓取，优先抓更新活跃的网页。
- Labrador 是一个工作在开源项目 Terrier Search Engine 上的非开源的爬虫。
- TeezirCrawler 是一个非开源的可伸缩的网页抓取器，在 Teezir 上使用。该程序被设计为一个完整的可以处理各种类型网页的爬虫，包括各种 JavaScript 和 HTML 文档。爬虫既支持主题检索也支持非主题检索。
- Spinn3r 是一个通过博客构建 Tailrank.com 反馈信息的爬虫。Spinn3r 是基于 Java 的，它的大部分体系结构都是开源的。

- HotCrawler 是一个使用 C 和 PHP 语言编写的爬虫。
- ViREL Microformats Crawler 搜索公众信息作为嵌入网页的一小部分。

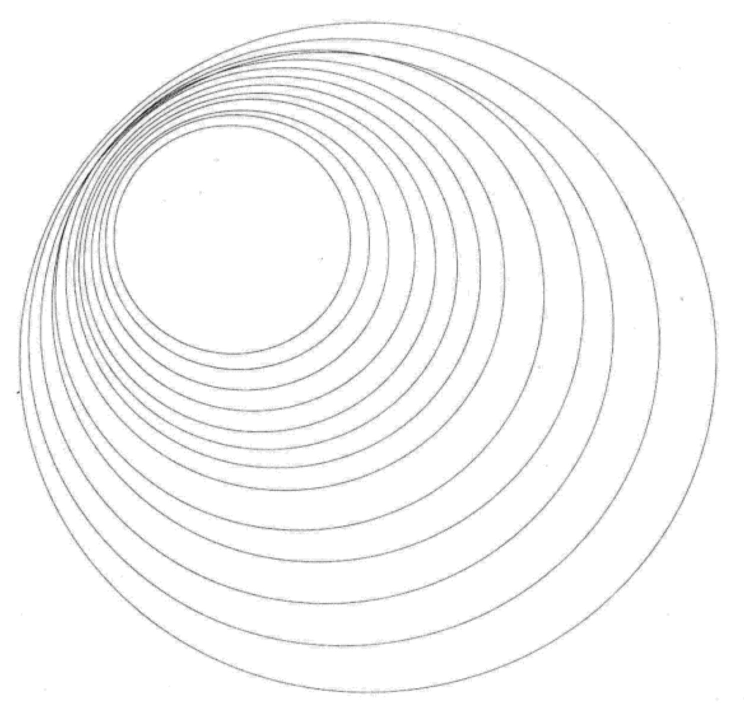
开源爬虫除了已经分析过的 Heritrix，还有下面的一些：

- DataparkSearch 是一个在 GNU GPL 许可下发布的爬虫搜索引擎。
- GNU Wget 是一个在 GPL 许可下，使用 C 语言编写的命令行式的爬虫。它主要用于网络服务器和 FTP 服务器的镜像。
- Ht://Dig 在它和索引引擎中包括了一个网页爬虫。
- HTTrack 用网络爬虫创建网络站点镜像，以便离线观看。它使用 C 语言编写，在 GPL 许可下发行。
- ICDL Crawler 是一个用 C++语言编写、跨平台的网络爬虫。它仅仅使用空闲的 CPU 资源，在 ICDL 标准上抓取整个站点。
- JSpider 是一个在 GPL 许可下发行的、高度可配置的、可定制的网络爬虫引擎。
- Larbin 是由 Sebastien Ailleret 开发的 C++语言实现的爬虫。
- Webtools4larbin 是由 Andreas Beder 开发的。
- Methabot 是一个使用 C 语言编写的高速优化的，使用命令行方式运行的，在 2-clause BSD 许可下发布的网页检索器。它的主要特性是高可配置性、模块化；它检索的目标可以是本地文件系统，HTTP 或者 FTP。
- Nutch 是一个使用 Java 编写，在 Apache 许可下发行的爬虫。它可以用来连接 Lucene 的全文检索套件。
- Pavuk 是一个在 GPL 许可下发行的，使用命令行的 Web 站点镜像工具，可以选择使用 X11 的图形界面。与 GNU Wget 和 HTTrack 相比，它有一系列先进的特性，如以正则表达式为基础的文件过滤规则和文件创建规则。
- WebVac 是斯坦福 WebBase 项目使用的一个爬虫。
- WebSPHINX 是一个由 Java 类库构成的，基于文本的搜索引擎。它使用多线程进行网页检索和 HTML 解析，拥有一个图形用户界面用来设置开始的种子 URL 和抽取下载的数据。
- WIRE-网络信息检索环境是一个使用 C++语言编写、在 GPL 许可下发行的爬虫，内置了几种页面下载安排的策略，还有一个生成报告和统计资料的模块，所以，它主要用于网络特征的描述。
- LWP：RobotUA 是一个在 Perl 5 许可下发行的，可以优异地完成并行任务的 Perl 类库构成的爬虫。
- Web Crawler 是一个用 C#语言编写的开放源代码的网络检索器。
- Sherlock Holmes 用于收集和检索本地和网络上的文本类数据(文本文件，网页)，该项目由捷克门户网站中枢(Czech web portal Centrum)赞助并且在该网站使用；它同时也在 Onet.pl 中使用。
- YaCy 是一个基于 P2P 网络的免费的分布式搜索引擎。
- Ruya 是一个在宽度优先方面表现优秀，基于等级抓取的开放源代码的网络爬虫。其在抓取英语和日语页面方面表现良好，在 GPL 许可下发行，并且完全使用 Python

语言编写。

- Universal Information Crawler 是快速发展的网络爬虫，用于检索、存储和分析数据。
- Agent Kernel 是一个当爬虫抓取时，用来进行安排、并发和存储的 Java 框架。
- Arachnod.net 是一个使用 C#语言编写，需要 SQL Server 2005 支持的，在 GPL 许可下发行的、多功能的、开源的机器人。它可以用来下载、检索和存储包括电子邮件地址、文件、超链接、图片和网页在内的各种数据。
- Dine 是一个多线程的 Java 的 HTTP 客户端。它可以在 LGPL 许可下进行二次开发。
- JoBo 是一个用于下载整个 Web 站点的简单工具。它采用 Java 实现。JoBo 直接使用 socket 下载网页。与其他下载工具相比较，它的主要优势是能够自动填充 form(如自动登录)和使用 cookies 来处理 session。JoBo 还有灵活的下载规则(如通过网页的 URL、大小、MIME 类型等)来限制下载。

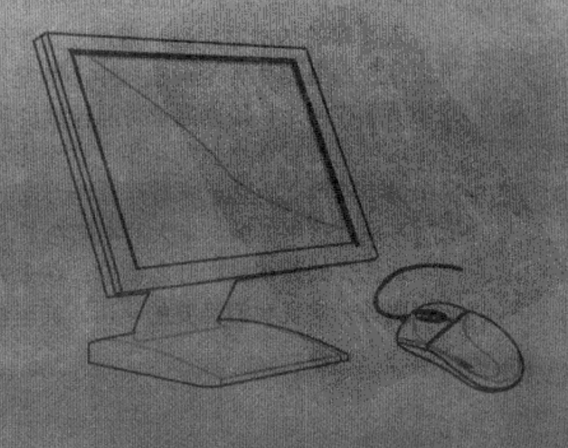
虽然有这么多公开的资源可用，但是，很多企业和个人还在不断地开发新的爬虫，尤其是主题爬虫，互联网发展过程中出现了一波一波的新技术，而在每一波都有开发爬虫的需要，现在实时搜索又成了热门，可能会需要实时爬虫，下一步如果语义网络真正发展起来了，也会需要语义搜索爬虫。



# 第2章

## 分布式爬虫

随着互联网技术的发展以及风起云涌的云计算浪潮。爬虫技术也逐渐向着分布式方向发展。比如，Google 的爬虫就是使用成千上万台小型机和微机进行合作，完成分布式抓取工作的。分布式技术不仅可以解决IT运营的成本，还可以解决爬虫效率问题，尤其是当今云计算的热潮，更把分布式推向了极致。



# 2.1 设计分布式爬虫

把抓取任务分布到不同的节点主要是为了抓取性能与可扩展性，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，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。

## 2.1.1 分布式与云计算

分布式技术是一种基于网络的计算机处理技术，与集中式相对应。近些年来，由于个人计算机的性能得到极大地提高及其使用的普及，使得将处理任务分布到网络上的所有计算机成为可能。分布式计算是和集中式计算相对立的概念，分布式计算的数据可以分布在很大区域。

在分布式网络中，数据的存储和处理都是在本地工作站上进行的。数据输出可以打印，也可以保存在软盘上。通过网络能得到更快、更便捷的数据访问。因为每台计算机都能够存储和处理数据，所以不要求服务器的功能十分强大，其价格也就不必过于昂贵。这种类型的网络可以适应用户的各种需要，同时允许他们共享网络的数据、资源和服务。在分布式网络中使用的计算机既能够作为独立的系统使用，也可以把它们连接在一起得到更强的网络功能。

分布式计算的优点是可以快速访问、多用户使用。每台计算机可以访问系统内其他计算机的信息文件，系统设计上具有更大的灵活性。既可为独立计算机的地区用户的特殊需求服务，也可为联网的企业需求服务，实现系统内不同计算机之间的通信。每台计算机都可以拥有和保持所需要的最大数据和文件，减少了数据传输的成本和风险。为分散地区和中心办公室双方提供更迅速的信息通信和处理方式，为每个分散的数据库提供作用域，数据存储于许多存储单元中，但任何用户都可以进行全局访问，使故障的不利影响最小化，以较低的成本来满足企业的特定要求。

云计算(Cloud Computing)是分布式处理(Distributed Computing)、并行处理(Parallel Computing)和网格计算(Grid Computing)的发展，或者说是这些计算机科学概念的商业实现。

云计算的基本原理是，通过使计算任务分布在大量的分布式计算机上，而非本地计算机或远程服务器中，企业数据中心的运行将与互联网更相似。这使得企业能够将资源切换到需要的应用上，根据需求访问计算机和存储系统。

这可是一种革命性的举措，打个比方，就好比是从古老的单台发电机模式转向了电厂集中供电的模式。它意味着计算能力也可以作为一种商品进行流通，就像煤气、水电一样，取用方便，费用低廉。最大的不同在于，它是通过互联网进行传输的。

云计算的蓝图已经呼之欲出：在未来，只需要一台笔记本或者一部手机，就可以通过网络服务来实现我们需要的一切，甚至包括超级计算这样的任务。从这个角度而言，最终

用户才是云计算的真正拥有者。

云计算的应用包含这样一种思想，把力量联合起来，给其中的每一个成员使用。

目前，PC 依然是我们日常工作生活中的核心工具——我们用 PC 处理文档、存储资料，用电子邮件或 U 盘与他人分享信息。如果 PC 硬盘坏了，我们会因为资料丢失而束手无策。

而在“云计算”时代，“云”会替我们做存储和计算的工作。“云”就是计算机群，每一群都包括几十万台、甚至上百万台计算机。“云”的好处还在于，其中的计算机可以随时更新，保证“云”长生不老。Google 就有好几个这样的“云”，其他 IT 巨头，如微软、雅虎、亚马逊(Amazon)也有或正在建设这样的“云”。

届时，我们只需要一台能上网的电脑，不需关心存储或计算发生在哪朵“云”上，但一旦有需要，我们可以在任何地点用任何设备，如电脑、手机等，快速地计算和找到这些资料。我们再也不用担心资料会丢失了。

云计算是虚拟化(Virtualization)、效用计算(Utility Computing)、IaaS(基础设施即服务)、PaaS(平台即服务)、SaaS(软件即服务)等概念混合演进并跃升的结果。云计算的特点如下：

(1) 超大规模。Google 云计算已经拥有 100 多万台服务器， Amazon、IBM、微软、Yahoo 等的“云”均拥有几十万台服务器。企业私有云一般拥有数百至上千台服务器。“云”能赋予用户前所未有的计算能力。

(2) 虚拟化。云计算支持用户在任意位置、使用各种终端获取应用服务。所请求的资源来自“云”，而不是固定的、有形的实体。应用在“云”中某处运行，但实际上用户无需了解、也不用担心应用运行的具体位置。只需要一台笔记本或者一部手机，就可以通过网络服务来实现我们需要的一切，甚至包括超级计算这样的任务。

(3) 高可靠性。“云”使用了数据多副本容错、计算节点同构可互换等措施来保障服务的高可靠性，使用云计算比使用本地计算机可靠。

(4) 通用性。云计算不针对特定的应用，在“云”的支撑下可以构造出千变万化的应用，同一个“云”可以同时支撑不同的应用运行。

(5) 高可扩展性。“云”的规模可以动态伸缩，满足应用和用户规模增长的需要。

(6) 按需服务。“云”是一个庞大的资源池，可以按需购买；云可以象自来水、电、煤气那样计费。

(7) 极其廉价。由于“云”的特殊容错措施可以采用极其廉价的节点来构成云，“云”的自动化集中式管理使得大量企业无需负担日益高昂的数据中心管理成本，“云”的通用性使资源的利用率较之传统系统大幅提升，因此用户可以充分享受“云”的低成本优势，经常只要花费几百美元、几天时间就能完成以前需要数万美元、数月时间才能完成的任务。

### 2.1.2　分布式与云计算技术在爬虫中的应用——浅析 Google 的云计算架构

分布式与云计算深刻地影响着搜索引擎的发展。与其说是云计算影响了搜索引擎，不如说是搜索引擎的发展产生了云计算的概念。Google 正是由于在云计算领域的领先，才能多年在搜索领域保持霸主的地位。本节以 Google 的架构为例，看一下云计算是如何应用在

爬虫之中的。

Google 的三大核心技术构成了实现云计算服务的基础：GFS(Google 文件系统)、MapReduce(分布式计算系统)和 BigTable(分布式存储系统)。

GFS 位于这三项技术的最底层，负责许多服务器、机器数据的存储工作。它将一个“大体积数据(通常在百兆甚至千兆级别)分隔成固定大小的数据块放到两到三个服务器上。这样做的目的是当一个服务器发生故障时，可以将数据迅速地从另外一个服务器上恢复过来。在存储层面，机器故障的处理由 Google 文件系统来完成。

MapReduce (分布式计算系统)是 Google 开发的编程工具，用于 1TB 数据的大规模数据集并行运算。这项技术的意义在于，实现跨越大量数据结点分割任务，使得某项任务可以被同时分拆在多台机器上执行。例如把一项搜索任务拆分成一两百个小的子任务，经并行处理后，将运算结果在后台合并，最后把最终结果返回到客户端。

BigTable (分布式存储系统)作为 Google 的一种对于半结构化数据进行分布存储与访问的接口或服务，它是建立在 GFS 和 MapReduce 之上的结构化分布式存储系统，可以帮助 Google 最大限度地利用已有的数据存储能力和计算能力，在提供服务时降低运行成本。

## 2.2 分布式存储

分布式存储是网络爬虫中的一个重要问题，抓取下来的 URL 要存放在分布式环境中。在云计算热潮风起云涌的今天，存储云的概念也被炒得沸沸扬扬。因此，如何在分布式网络环境中存储数据，也是分布式爬虫的重要课题。

### 2.2.1 从 Ralation_DB 到 key/value 存储

前几年，key/value 这个词还是和 hash 表联系在一起的。而现在，程序员看见 key/value 这个词时，马上联想到的就是 BigTable、SimpleDB 和云计算。当下，key/value 存储(或者叫 key/value Database、云存储等)是个非常时髦的词汇，越来越多的开发人员(特别是互联网企业)开始关注和尝试 key/value 的存储形式。

key/value 形式的存储并不是凭空想象出来的。有两个原因导致了 key/value 存储方式的崛起。

#### 1. 大规模的互联网应用

对于 Google 和 eBay 这样的互联网企业，每时每刻都有无数的用户在使用它们提供的互联网服务，这些服务带来的就是大量的数据吞吐量。同一时间，会并发出成千上万的连接对数据库进行操作。在这种情况下，单台服务器或者几台服务器远远不能满足这些数据处理的需求，简单的升级服务器性能的方式也不行，所以唯一可以采用的办法就是使用集群了。使用集群的方法有很多种，但大致分为两类：一类仍然采用关系数据库管理系统(RDBMS)，然后通过对数据库的垂直和水平切割将整个数据库部署到一个集群上，这种方

法的优点在于可以采用 RDBMS 这种熟悉的技术，但缺点在于它是针对特定应用的。由于应用的不同，切割的方法是不一样的。关于数据库的垂直和水平切割的具体细节可以查看相关资料。

还有一类就是 Google 所采用的方法，抛弃 RDBMS，采用 key/value 形式的存储，这样可以极大地增强系统的可扩展性(scalability)，如果要处理的数据量持续增大，多加机器就可以了。事实上，key/value 的存储就是由于 BigTable 等相关论文的发表慢慢进入人们的视野的。

**2. 云存储**

如果说上一个问题还有可以替代的解决方案(切割数据库)的话，那么对于云存储来说，也许 key/value 的存储就是唯一的解决方案了。云存储简单点说就是构建一个大型的存储平台给别人用，这也就意味着在这上面运行的应用其实是不可控的。如果其中某个客户的应用随着用户的增长而不断增长时，云存储供应商是没有办法通过数据库的切割来达到扩展的，因为这个数据是客户的，供应商不了解这个数据自然就没法作出切割。在这种情况下，key/value 的存储就是唯一的选择了，因为这种条件下的可扩展性必须是自动完成的，不能有人工干预。这也是为什么目前几乎所有的云存储都是 key/value 形式的，例如 Amazon 的 smipleDB，底层实现就是 key/value，还有 Google 的 GoogleAppEngine，采用的是 BigTable 的存储形式。

key/value 存储与 RDBMS 相比，一个很大的区别就是它没有模式的概念。在 RDBMS 中，模式所代表的其实就是对数据的约束，包括数据之间的关系(relationship)和数据的完整性(integrity)，比如 RDBMS 中对于某个数据属性会要求它的数据类型是确定的(整数或者字符串等)，数据的范围也是确定的(0～255)，而这些在 key/value 存储中都没有。在 key/value 存储中，对于某个 key，value 可以是任意的数据类型。

在所有的 RDBMS 中，都是采用 SQL 语言对数据进行访问。一方面，SQL 对于数据的查询功能非常强大；另一方面，由于所有的 RDBMS 都支持 SQL 查询，所以可移植性很强。而在 key/value 存储中，对于数据的操作使用的都是自定义的一些 API，而且支持的查询也比较简单。

正如前面反复提及的，key/value 存储最大的特点就是它的可扩展性(scalability)，这也是它最大的优势。所谓的可扩展性，其实包括两方面内容。一方面，是指 key/value 存储可以支持极大的数据存储。它的分布式的架构决定了只要有更多的机器，就能够保证存储更多的数据。另一方面，是指它可以支持数量很多的并发的查询。对于 RDBMS，一般几百个并发的查询就可以让它很吃力了，而一个 key/value 存储，可以很轻松地支持上千个并发查询。

key/value 存储的缺陷主要有两点：

- 由于 key/value 存储中没有 schema，所以它是不提供数据之间的关系和数据的完备性的，所有的这些东西都落到了应用程序一端，其实也就是开发人员的头上。这无疑加重了开发人员的负担。

- 在 RDBMS 中，需要设定各表之间的关系，这其实是一个数据建模的过程(data modeling process)。当数据建模完成后，这个数据库对于应用程序就是独立的了，这就意味着其他程序可以在不改变数据模型的前提下使用相同的数据集。但在 key/value 存储中，由于没有这样一个数据模型，不同的应用程序需要重复进行这个过程。
- key/value 存储最大的一个缺点在于它的接口是不熟悉的。这阻碍了开发人员可以快速而顺利地用上它。当然，现在有种做法，就是在 key/value 存储上再加上一个类 SQL 语句的抽象接口层，从而使得开发人员可以用他们熟悉的方式(SQL)来操作 key/value 存储。但由于 RDBMS 和 key/value 存储的底层实现有着很大的不同，这种抽象接口层或多或少还是受到了限制。

### 2.2.2 Consistent Hash 算法

分布式存储常常会涉及负载平衡的问题，由于有多个存储介质，分布在不同的结点上。因此，当一个对象被保存的时候，它究竟应该保存在哪个存储介质上呢(存储介质可以是数据库、Berkeley DB 等，甚至可以是内存数据结构)？这就是负载均衡的问题，如图 2.1 所示。

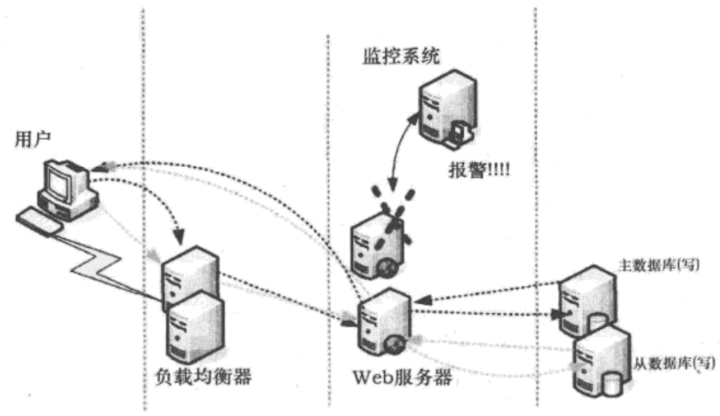



图 2.1 负载均衡示意图

面对云计算的热潮，如何很好地分布存储数据，是一个非常重要的话题。在分布式网络爬虫中，抓取的页面非常多(通常是数十亿级别)，因此，分布式存储就非常有意义。那么，如果抓取下来一个页面，究竟要存放到哪个数据库中呢？这就涉及负载均衡的问题。

如果你有 N 个数据存储服务器，那么如何将一个对象(object)映射到 N 个服务器上呢？你很可能会采用类似下面的通用方法计算对象的 hash 值，然后均匀地映射到 N 个服务器：

```
hash(object)%N
```

一切都运行正常，但是要考虑以下两种情况：

(1) 一个服务器 m 挂掉了(在实际应用中必须要考虑这种情况)，则所有映射到服务器 m 的对象都会失效。怎么办，需要把服务器 m 移除，这时候服务器为 N-1 台，映射公式变成了 hash(object)%(N-1)。

(2) 由于访问加重，需要添加服务器，这时候服务器是 N+1 台，映射公式变成了 hash(object)%(N+1)。

在上面两种情况下，突然之间几乎所有的服务器都失效了(请看下面单调性的解释)。对于服务器而言，这是一场灾难。

再来考虑第三个问题，由于硬件能力越来越强，你可能会想让后面添加的节点多干点活，显然上面的 hash 算法也做不到。

有什么方法可以改变这个状况呢，这就要用到 Consistent Hashing 算法。

hash 算法的一个衡量指标是单调性(Monotonicity)，定义如下：

单调性是指如果已经有一些内容通过哈希分配到了相应的缓冲中，而又有新的缓冲加入到系统中。哈希的结果应能够保证原有已分配的内容可以被映射到新的缓冲中去，而不会被映射到旧的缓冲集合中的其他缓冲区。

上面的简单 hash 算法 hash(object)%N 难以满足单调性要求。

Consistent Hashing 是一种 hash 算法，简单地说，在移除/添加一个服务器时，它能够尽可能小地改变已存在的 key 映射关系，尽可能地满足单调性的要求。下面就按照 5 个步骤简单讲讲 Consistent Hashing 算法的基本原理。

**步骤一**：环形 hash 空间。

考虑通常的 hash 算法都是将 value 映射到一个 32 位的 key 值，即 $0\sim2^{32}-1$ 的数值空间。我们可以将这个空间想象成一个首(0)尾($2^{32}-1$)相接的圆环，如图 2.2 所示。

**步骤二**：把对象映射到 hash 空间。

接下来考虑 4 个对象 object1～object4，通过 hash 函数计算出的 hash 值 key 在环上的分布如图 2.2 所示。

```
hash(object1) = key1;
…
hash(object4) = key4;
```

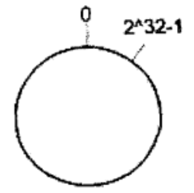



图 2.2　环形 hash 空间

**步骤三**：把服务器映射到 hash 空间。

Consistent Hashing 的基本思想就是将对象和服务器都映射到同一个 hash 数值空间，并且使用相同的 hash 算法。

假设当前有 A、B 和 C 共 3 台服务器，那么其映射结果将如图 2.3 所示，它们在 hash 空

间中，以对应的 hash 值排列。

```
hash(服务器 A) = key A;
…
hash(服务器 C) = key C;
```

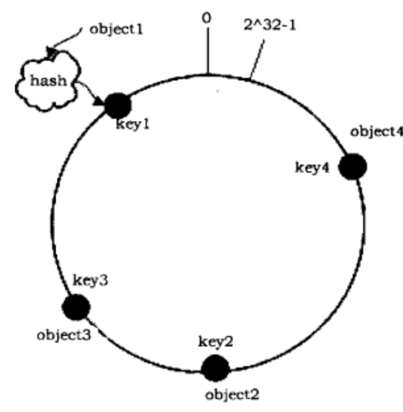



图 2.3　4 个对象的 key 值分布

**步骤四：** 把对象映射到服务器。

现在 cache 和对象都已经通过同一个 hash 算法映射到 hash 数值空间中了，接下来要考虑的就是如何将对象映射到 cache 上面。

在这个环形空间中，如果沿着顺时针方向从对象的 key 值出发，直到遇见一个服务器，那么就将该对象存储在这个服务器上，因为对象和服务器的 hash 值是固定的，因此这个服务器必然是唯一和确定的。这样不就找到了对象和服务器的映射方法了吗？

依然继续上面的例子(见图 2.4)，那么根据上面的方法，对象 object1 将被存储到服务器 A 上，object2 和 object3 对应到服务器 C，object4 对应到服务器 B。

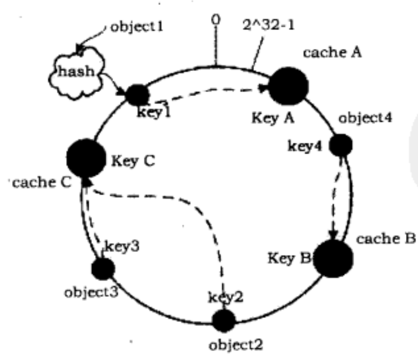



图 2.4　服务器对象的 key 值表示

**步骤五**：考察服务器的变动。

前面讲过，通过 hash 算法然后求余的方法带来的最大问题就在于不能满足单调性，当服务器有所变动时，服务器会失效，进而对后台服务器造成巨大的冲击，现在就来分析 Consistent Hashing 算法。

(1) 移除服务器。

考虑假设服务器 B 挂掉了，根据上面讲到的映射方法，这时受影响的将只是那些沿 cache B 逆时针遍历直到下一个服务器(服务器 C )之间的对象，也即是本来映射到服务器 B 上的那些对象。

因此这里仅需要变动对象 object4，将其重新映射到服务器 C 上即可，如图 2.5 所示。

(2) 添加服务器。

再考虑添加一台新的服务器 D 的情况，假设在这个环形 hash 空间中，服务器 D 被映射在对象 object2 和 object3 之间。这时受影响的仅是那些沿 cache D 逆时针遍历直到下一个服务器(服务器 B )之间的对象，将这些对象重新映射到服务器 D 上即可。 因此这里仅需要变动对象 object2，将其重新映射到服务器 D 上，如图 2.6 所示。

考量 hash 算法的另一个指标是平衡性(Balance)，定义如下：

平衡性是指哈希的结果能够尽可能分布到所有的缓冲中，这样可以使所有的缓冲空间都得到利用。

hash 算法并不能保证绝对的平衡，如果服务器较少，对象并不能被均匀地映射到服务器上，比如在上面的例子中，仅部署服务器 A 和服务器 C 的情况下，在 4 个对象中， 服务器 A 仅存储了 object1，而服务器 C 则存储了 object2、object3 和 object4，分布是很不均衡的。

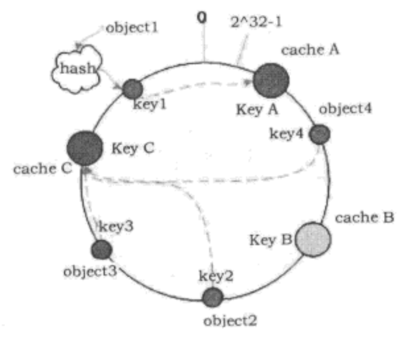



图 2.5　服务器 B 被移除后的映射

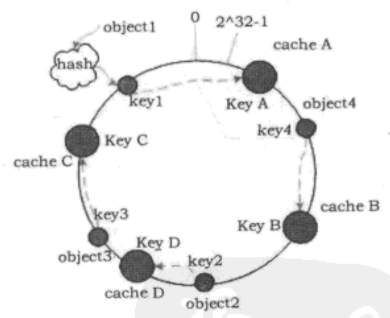



图 2.6　添加服务器 D 后的映射关系

为了解决这种情况，Consistent Hashing 引入了“虚拟节点”的概念，它可以如下定义：

“虚拟节点”( virtual node )是实际节点在 hash 空间的复制品( replica )，一个实际节点对应若干个“虚拟节点”，这个对应个数也称为“复制个数”，“虚拟节点”在 hash 空间中以 hash 值排列。

仍以仅部署服务器 A 和 服务器 C 的情况为例，在图 2.5 中我们已经看到，服务器分布并不均匀。现在我们引入虚拟节点，并设置“复制个数”为 2，这就意味着一共会存在 4

个“虚拟节点”，服务器 A1 和服务器 A2 代表服务器 A；服务器 C1 和服务器 C2 代表服务器 C，假设一种比较理想的情况如图 2.7 所示。

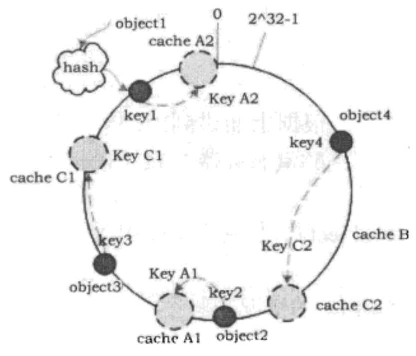



图 2.7 引入“虚拟节点”后的映射关系

此时，对象到“虚拟节点”的映射关系为：

objec1->服务器 A2；objec2->服务器 A1；objec3->服务器 C1；objec4->服务器 C2。

因此对象 object1 和 object2 都被映射到服务器 A 上，而 object3 和 object4 映射到服务器 C 上，平衡性有了很大提高。

引入“虚拟节点”后，映射关系就从 { 对象 -> 节点 } 转换到了 { 对象 -> 虚拟节点 } 。查询对象所在 cache 时的映射关系如图 2.8 所示。

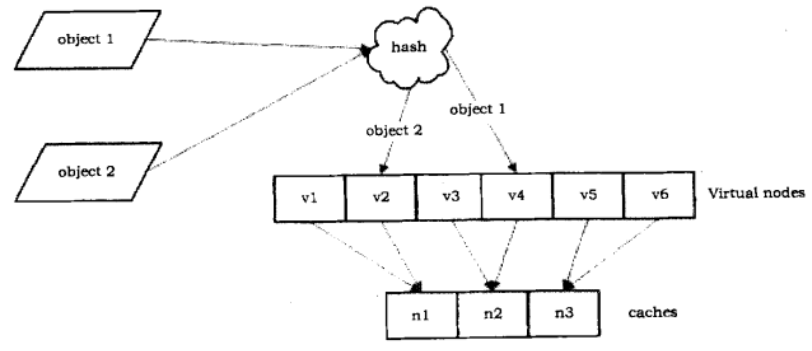



图 2.8 查询对象所在的 cache

“虚拟节点”的 hash 计算可以采用对应节点的 IP 地址加数字后缀的方式。例如假设服务器 A 的 IP 地址为 202.168.14.241。引入“虚拟节点”前，计算服务器 A 的 hash 值：

```
Hash("202.168.14.241");
```

引入“虚拟节点”后，计算“虚拟节点”服务器 A1 和服务器 A2 的 hash 值：

```
hash("202.168.14.241#1");  // cache A1
hash("202.168.14.241#2");  // cache A2
```

### 2.2.3 Consistent Hash 代码实现

上一节，讲述了 Consistent Hash 算法的原理。本节，我们实现一个简单的 Consistent Hash 算法。

```
public class ConsistentHash<T> {
 private final HashFunction hashFunction;//hash 算法
 private final int numberOfReplicas;//虚拟节点数目
 private final SortedMap<Integer, T> circle = new TreeMap<Integer, T>();
 public ConsistentHash(HashFunction hashFunction, int numberOfReplicas,
     Collection<T> nodes //物理节点) {
   this.hashFunction = hashFunction;
   this.numberOfReplicas = numberOfReplicas;
   for (T node : nodes) {
     add(node);
   }
 }

 public void add(T node) {
   for (int i = 0; i < numberOfReplicas; i++) {
     circle.put(hashFunction.hash(node.toString() + i), node);
   }
 }

 public void remove(T node) {
   for (int i = 0; i < numberOfReplicas; i++) {
     circle.remove(hashFunction.hash(node.toString() + i));
   }
 }

 public T get(Object key) {//关键算法
   if (circle.isEmpty()) {
     return null;
   }
   //计算 hash 值
   int hash = hashFunction.hash(key);
   //如果不包括这个 hash 值
   if (!circle.containsKey(hash)) {
     SortedMap<Integer, T> tailMap = circle.tailMap(hash);
     hash = tailMap.isEmpty() ? circle.firstKey() : tailMap.firstKey();
   }
   return circle.get(hash);
 }
```

本节内容详细讲述了有关分布式和云计算存储的相关问题，下面几节，我们将讲述 Google 的分布式和云计算技术以及它们在网络爬虫中的应用。

## 2.3 Google 的成功之道——GFS

Google 之所以能成功，很大程度上是应用了 GFS+BigTable+MapReduce 的架构。由于这种架构，使得 Google 在当前风起云涌的云计算热潮中始终保持领先地位。Google 的爬虫也是采用 GFS 作为存储网页的底层数据结构。为何 GFS 能如此受 Google 的青睐？本节将为你揭示这个谜团。

### 2.3.1 GFS 详解

GFS(Google File System)是 Google 自己研发的一个适用于大规模分布式数据处理相关应用的、可扩展的分布式文件系统。它基于普通的不算昂贵的硬件设备，实现了容错的设计，并且为大量客户端提供了极高的聚合处理性能。GFS 正好与 Google 的存储要求相匹配，因此在 Google 内部广泛用作存储平台，适用于 Google 的服务产生和处理数据应用的要求，以及 Google 的海量数据的要求。Google 最大的集群通过上千个计算机的数千个硬盘，提供了数百 TB 的存储，并且这些数据被数百个客户端并行操作。Google 之所以要研发自己的分布式文件系统，是因为在分布式存储环境中，常常会产生以下一些问题。

#### 1. 在分布式存储中，经常会出现节点失效的情况

因为在分布式存储中，文件系统包含了几百个或者几千个廉价的普通机器，而且这些机器要被巨大数量的客户端访问。节点失效可能是由于程序的 bug，操作系统的 bug，人工操作的失误，以及硬盘坏掉，内存、网络、插板的损坏，电源的坏掉等原因造成的。因此，持续监视，错误检测，容错处理，自动恢复必须集成到这个文件系统的设计中。

#### 2. 分布式存储的文件都是非常巨大的

GB 数量级的文件是常事。每一个文件都包含了很多应用程序对象，比如 Web 文档等。搜索引擎的数据量是迅速增长的，它通常包含数十亿数据对象。如果使用一般的文件系统，就需要管理数十亿个 KB 数量级大小的文件。而且，每次 I/O 只能读出几个字节也不能满足搜索引擎吞吐量的要求。因此，Google 决定设计自己的文件系统，重新规定每次 I/O 的块的大小。

#### 3. 对于搜索引擎的业务而言，大部分文件都只会在文件尾新增加数据，而少见修改已有数据的

对一个文件的随机写操作在实际上几乎是不存在的。一旦写完，文件就是只读的，并且一般都是顺序读取。比如，在网络爬虫中，把网页抓取下来之后不会做修改，而只是简单地存储，作为搜索结果的快照。在实际的系统中，许多数据都有这样的特性。有些数据可能组成很大的数据仓库，并且数据分析程序从头扫描到尾。有些可能是运行应用而不断地产生数据流。对于这些巨型文件的访问模式来说，增加模式是最重要的，所以我们首要优化性能的就是它。

**4. 与应用一起设计的文件系统 API 对于增加整个系统的弹性和适用性有很大的好处**

为了满足以上几点需要，Google 遵照下面几条原则设计了它的分布式文件系统(GFS)：

(1) 系统建立在大量廉价的普通计算机上，这些计算机经常出故障。则必须对这些计算机进行持续检测，并且在系统的基础上进行检查、容错，以及从故障中进行恢复。

(2) 系统存储了大量的超大文件。数 GB 的文件经常出现并且应当对大文件进行有效的管理。同时必须支持小型文件，但是不必为小型文件进行特别的优化。

(3) 一般的工作都是由两类读取组成：大的流式读取和小规模的随机读取。在大的流式读取中，每个读操作通常一次就要读取几百字节以上的数据，每次读取 1MB 或者以上的数据也很常见。因此，在大的流式读取中，对于同一个客户端来说，往往会发起连续的读取操作顺序读取一个文件。而小规模的随机读取通常在文件的不同位置，每次读取几字节数据。对于性能有过特别考虑的应用通常会做批处理并且对它们读取的内容进行排序，这样可以使得它们的读取始终是单向顺序读取，而不需要往回读取数据。

(4) 通常基于 GFS 的操作都有很多超大的、顺序写入的文件操作。通常写入操作的数据量和读入的数据量相当。一旦完成写入，文件就很少会更改。应支持文件的随机小规模写入，但是不需要为此做特别的优化。

**5. 系统必须非常有效地支持多个客户端并行添加同一个文件**

GFC 文件经常使用生产者/消费者队列模式，或者以多路合并模式进行操作。好几百个运行在不同机器上的生产者，将会并行增加一个文件。

**6. 高性能的稳定带宽的网络要比低延时更加重要**

GFS 目标应用程序一般会大量操作处理比较大块的数据。

基于以上几点考虑，GFS 采用了如图 2.9 所示的架构。

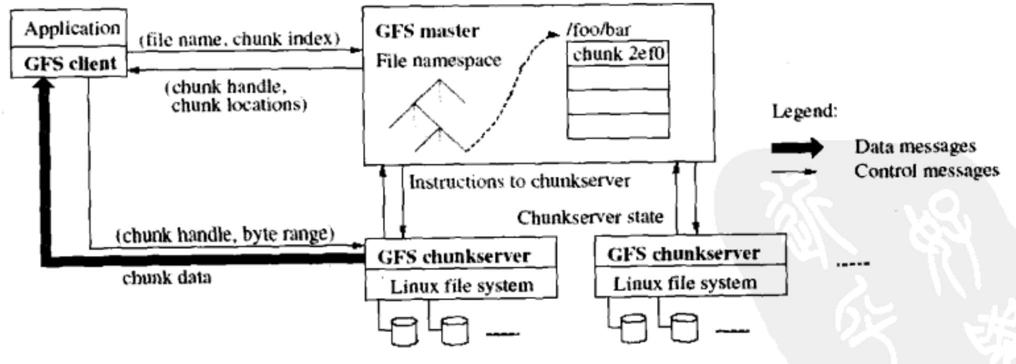



图 2.9　Google File System 架构

如图 2.9 所示，GFS 集群由一个主服务器(master)和多个块服务器(chunkserver)组成，GFS 集群会有很多客户端访问。每一个节点都是一个普通的 Linux 计算机，运行的是一个用户级别的服务器进程。

在 GFS 下，每个文件都被拆成固定大小的块(chunk)。每一个块都由主服务器根据块创建的时间产生一个全局唯一的以后不会改变的 64 位的块处理(chunk handle)标志。块服务器在本地磁盘上用 Linux 文件系统保存这些块，并且根据块处理标志和字节区间，通过 Linux 文件系统读写这些块的数据。出于可靠性的考虑，每一个块都会在不同的块处理器上保存备份。

主服务器负责管理所有的文件系统的元数据，包括命名空间、访问控制信息、文件到块的映射关系、当前块的位置等信息。主服务器也同样控制系统级别的活动，比如块的分配管理，孤点块的垃圾回收机制、块服务器之间的块镜像管理。

连接到各个应用系统的 GFS 客户端代码包含了文件系统的 API，并且会和主服务器及块服务器进行通信处理，代表应用程序进行读写数据的操作。客户端和主服务器进行元数据的操作，但是所有的数据相关的通信是直接和块服务器进行的。

由于在流式读取中，每次都要读取非常多的文件内容，并且读取动作是顺序读取，因此，在客户端没有设计缓存。没有设计缓存系统使得客户端以及整个系统都大大简化了(少了缓存的同步机制)。块服务器不需要缓存文件数据，因为块文件就像本地文件一样被保存，所以 Linux 的缓存已经把常用的数据缓存到了内存里。

下面简单介绍图 2.9 中的读取操作分段。首先，客户端把应用要读取的文件名和偏移量，根据固定的块大小，转换为文件的块索引。然后向主服务器发送这个包含了文件名和块索引的请求。主服务器返回相关的块处理标志以及对应的位置。客户端缓存这些信息，把文件名和块索引作为缓存的关键索引字。

于是这个客户端就向对应位置的块服务器发起请求，通常这个块服务器是离这个客户端最近的一个。请求给定了块处理标志以及需要在所请求的块内读取的字节区间。在这个块内，再次操作数据将不用再通过客户端-主服务器的交互，除非这个客户端本身的缓存信息过期了，或者这个文件重新打开了。实际上，客户端通常都会在请求中附加向主服务器询问多个块的信息，主服务器会立刻给这个客户端回应这些块的信息。这个附加信息是通过几个几乎没有任何代价的客户端-主服务器的交互完成的。

块的大小是设计的关键参数。Google 选择的块大小为 64MB，远远大于典型的文件系统的块大小。每一个块的实例(复制品)都是作为在主服务器上的 Linux 文件格式存放的，并且只有当需要的情况下才会增长。滞后分配空间的机制可以通过文件内部分段来避免空间浪费，对于这样大的块大小来说，内部分段可能是一个最大的缺陷了。

选择一个很大的块可以提供一些重要的好处。首先，它减少了客户端和主服务器的交互，因为在同一个块内的读写操作只需要客户端初始询问一次主服务器关于块的位置信息就可以了。对主服务器访问的减少可以显著提高系统性能，因为使用 GFS 的应用大部分是顺序读写超大文件的。即使是对小范围的随机读，客户端也可以很容易地缓存许多大的数据文件的位置信息。其次，由于是使用一个大的块，客户端可以在一个块上完成更多的操作，它可以通过维持一个到主服务器的 TCP 持久连接来减少网络管理量。第三，它减少了元数据在主服务器上的大小，使得 Google 应用程序可以把元数据保存在内存中。

下面简单介绍图 2.9 中的主服务器：

主服务器节点保存了三个主要的数据类型：文件和块的命名空间，文件到块的映射关

系，每一个块的副本的位置。所有的元数据都保存在主服务器的内存里。头两个类型(namepspaces 和文件到块的映射)同时也保存在主服务器本地磁盘的日志中。通过日志，在主服务器宕机的时候，我们可以简单、可靠地恢复主服务器的状态。主服务器并不持久化保存块位置信息。相反，它在启动的时候以及主服务器加入集群的时候，向每一个主服务器询问它的块信息。

因为元数据都是在内存保存的，所以在主服务器上操作很快。另外主服务器也很容易定时扫描后台所有的内部状态。定时扫描内部状态可以用来实现块的垃圾回收，当主服务器失效的时候重新复制，还可以作为服务器之间的块镜像，在执行负载均衡和磁盘空间均衡任务时使用。

因为我们采用内存保存元数据的方式，如果需要支持更大的文件系统，我们可以简单、可靠、高效、灵活地通过增加主服务器的内存来实现。

主服务器并不持久化保存块服务器上的块记录。它只是在启动的时候简单地从块服务器取得这些信息。主服务器可以在启动之后一直保持自己的这些信息是最新的，因为它控制所有的块的位置。

上文提到的主服务器的日志信息保存了关键的元数据变化历史记录，它是 GFS 的核心。不仅仅因为它是唯一持久化的元数据记录，而且也是因为日志记录也是作为逻辑时间基线，定义了并行操作的顺序。块以及文件，都是用它们创建时刻的逻辑时间基线来作为唯一的并且永远唯一的标志。

由于日志记录是极关键的，因此必须可靠保存，在元数据改变并且持久化之前，对于客户端来说都是不可见的(也就是说保证原子性)。否则，就算是在块服务器完好的情况下，也可能会丢失整个文件系统，或者最近的客户端操作。因此，把这个文件保存在多个不同的主机上，并且只有当刷新这个相关的日志记录到本地和远程磁盘之后，才会给客户端操作应答。主服务器可以每次刷新一批日志记录，以减少刷新和复制这个日志导致的系统吞吐量。

主服务器通过自己的日志记录进行自身文件系统状态的反演。为了减少启动时间，我们必须尽量减少操作日志的大小。主服务器在日志增长超过某一个大小的时候，执行检查点动作，这样可以使下次启动的时候从本地硬盘读出这个最新的检查点，然后反演有限记录数。检查点是一个类似 B-树的格式，可以直接映射到内存，而不需要额外的分析。这更进一步加快了恢复的速度，提高了可用性。

对于主服务器的恢复，只需要最新的检查点以及后续的日志文件。旧的检查点及其日志文件可以删掉了，但我们还是要保存几个检查点以及日志文件，用来防止发生比较大的故障。

GFS 是一个松散的一致性检查的模型，通过简单高效的实现，来支持高度分布式计算的应用。下面详细讲解 GFS 的一致性模型。

文件名字空间的改变(如文件的创建)是原子操作，由主服务器来专门处理。名字空间的锁定保证了操作的原子性以及正确性，主服务器的操作日志定义了这些操作的全局顺序。

什么是文件区，文件区就是在文件中的一小块内容。

不论对文件进行何种操作，文件区所处的状态都包括三种：一般成功、并发成功和失

败。表 2.1 列出了这些结果。当所有的客户端看到的都是相同的数据的时候，并且与这些客户端从哪个数据的副本读取无关的时候，这个文件区是一致性的。当一个更改操作成功完成，而且没有并发写冲突时，那么受影响的区就是确定的(并且潜在一致性)：所有客户端都可以看到这个变化是什么。并发成功操作使得文件区是不确定的，但是是一致性的：所有客户端都看到了相同的数据，但是并不能确定到底什么变化发生了。通常，这种变化由好多个变动混合片断组成。一个失败的改变会使得一个文件区不一致(因此也不确定)：不同的用户可能在不同时间看到不同的数据。

如果表 2.1 中数据更改可能是写一个记录或者是一个记录增加。写操作会导致一个应用指定的文件位置的数据写入动作。记录增加会导致数据(记录)增加，这个增加即使是在并发操作中也至少是一个原子操作，但是在并发记录增加中，GFS 选择一个偏移量增加(与之对应的是，一个“普通”增加操作是类似写到当前文件最底部的一个操作)。我们把偏移量返回给客户端，并且标志包含这个记录的确定的区域的开始位置。另外，GFS 可以在这些记录之间增加填充，或者仅仅是记录的重复。这些确定区间之间的填充或者记录的重复是不一致的，并且通常是因为用户记录数据比较小造成的。

表 2.1　文件区所处状态

| | 写 记 录 | 增加记录 |
|---|---|---|
| 一般成功 | 定义 | 定义 |
| 并发成功 | 一致定义 | |
| 失败 | 非一致 | |

在一系列成功的改动之后，改动后的文件区是确定的，并且包含了最后一个改动所写入的数据。GFS 通过对所有的数据副本，按照相同顺序对块进行提交数据的改动来保证这样的一致性，并且采用块的版本号码控制机制来检查是否有过期的块改动，这种检查通常在主服务器宕机的情况下使用。

另外，由于客户端会缓存这个块的位置，因此可能会在信息刷新之前读到这个过期的数据副本。这个故障潜在发生的区间受块位置缓存的有效期限制，并且也受到下次重新打开文件的限制，重新打开文件会把这个文件所有的块相关的缓存信息全部丢弃而重新设置。此外，由于多数文件都只是追加数据，过期的数据副本通常返回一个较早的块尾部(也就是说这种模式下，过期的块返回的仅仅是说，这个块它以为是最后一个块，其实不是)，而不是返回一个过期的数据。

### 2.3.2　开源 GFS——HDFS

Google 的文件系统究竟是如何写的，我们不得而知。但是根据 Google 发表的论文以及 GFS 的相关资料，可以知道 apache 下有一个开源实现——HDFS。这一小节，我们来介绍 HDFS 的架构与设计。HDFS 的架构如图 2.10 所示。

根据 GFS 中主服务器/块服务器的设计，HDFS 采用了主服务器/从属服务器架构。一个 HDFS 集群是由一个名称节点和一定数目的数据节点组成的。名称节点是一个中心服务器，

负责管理文件系统的名称空间和客户端对文件的访问。数据节点在集群中一般是一个节点一个，负责管理节点上附带的存储。在内部，一个文件会分成一个或多个块，这些块存储在数据节点集合里。名称节点执行文件系统的名称空间操作，例如打开、关闭、重命名文件和目录，同时决定块到具体数据节点的映射。数据节点在名称节点的指挥下进行块的创建、删除和复制。名称节点和数据节点都是设计成可以运行在普通的、廉价的机器上。HDFS 采用 Java 语言开发，因此可以部署在不同的操作系统平台上。一个典型的部署场景是一台机器运行一个单独的名称节点，集群中的其他机器各自运行一个数据节点实例。这个架构并不排除一台机器上运行多个数据节点，不过这比较少见。

HDFS Architecture

Metadata ops　Namenode　Metadata(Name, replicas, ...): /home/foo/data.3...

Client　Block ops　Read　Datanodes　Datanodes

Replication　Blocks

Rack 1　Write　Rack 2

Client

图 2.10　HDFS 架构

名称节点运行在单一节点上大大简化了系统的架构。名称节点负责保管和管理所有的 HDFS 元数据，而用户与数据节点的通信就不需要通过名称节点(也就是说文件数据直接在数据节点上读写)。

HDFS 支持传统的层次型文件组织，与大多数其他文件系统类似，用户可以创建目录，并在其中创建、删除、移动和重命名文件。名称节点维护文件系统的名称空间，任何对文件系统的名称空间和文件属性的修改都将被名称节点记录下来。用户可以设置 HDFS 保存的文件的副本数目，文件副本的数目称为文件的复制因子，这个信息也是由名称节点保存。

HDFS 被设计成在一个大集群中可靠地存储海量文件的系统。它将每个文件存储成块序列，除了最后一个块，所有的块大小都是相同的。文件的所有块都会被复制。每个文件的块大小和复制因子都是可配置的。复制因子可以在文件创建的时候配置，而且以后也可以改变。HDFS 中的文件是单用户写模式，并且严格要求在任何时候都只能有一个用户写入。名称节点全权管理块的复制，它周期性地从集群中的每个数据节点接收心跳包和一个数据块报告(Blockreport)。心跳包的接收表示该数据节点正常工作，而数据块报告包括该数据节点上所有的块组成的列表。

名称节点存储 HDFS 的元数据。对于任何修改文件元数据的操作，名称节点都用一个名为 Editlog 的事务日志记录下来。例如，在 HDFS 中创建一个文件，名称节点就会在 Editlog 中插入一条记录来表示；同样，修改文件的复制因子也会在 Editlog 中插入一条记录。名称

节点在本地 OS 的文件系统中存储这个 Editlog。整个文件系统的名称空间，包括块到文件的映射、文件的属性，都存储在名为 FsImage 的文件中，这个文件也放在名称节点所在系统的文件系统中。

名称节点在内存中保存着整个文件系统的名称空间和文件块的映像。这个关键的元数据设计得很紧凑，一个带有 4GB 内存的名称节点足以支撑海量的文件和目录。当名称节点启动时，它从硬盘中读取 Editlog 和 FsImage，将 Editlog 中的所有事务作用在内存中的 FsImage，并将这个新版本的 FsImage 从内存中创新到硬盘上。这个过程称为检查点(checkpoint)。在当前实现中，检查点只在名称节点启动时发生。

所有的 HDFS 通信协议都是构建在 TCP/IP 协议上的。客户端通过一个可配置的端口连接到名称节点，通过各个端协议组件(Client Protocol)与名称节点交互。而数据节点是使用数据节点协议组件(Datanode Protocol)与名称节点交互。

使用 HDFS 的应用都是处理大数据集合的。这些应用都是写数据一次，而读是一次到多次，并且读的速度要满足流式读。HDFS 支持文件的一次写入多次读取。一个典型的块大小是 64MB，因而，文件总是按照 64MB 大小切分成块，每个块存储于不同的数据节点中。

客户端创建文件的请求其实并没有立即发给名称节点，事实上，HDFS 客户端会将文件数据缓存到本地的一个临时文件。应用的写操作被透明地重定向到这个临时文件。当这个临时文件累积的数据超过一个块的大小(默认为 64MB)，客户端才会联系名称节点。名称节点将文件名插入文件系统的层次结构中，并且给它分配一个数据块，然后返回数据节点的标识符和目标数据块给客户端。客户端将本地临时文件刷新到指定的数据节点上。当文件关闭时，在临时文件中剩余的没有刷新的数据也会传输到指定的数据节点，然后客户端告诉名称节点文件已经关闭。此时名称节点才将文件创建操作提交到持久存储。如果名称节点在文件关闭前挂了，则该文件将丢失。

当某个客户端向 HDFS 文件写数据的时候，一开始是写入本地临时文件，假设该文件的复制因子设置为 3，那么客户端会从名称节点获取一张数据节点列表来存放副本。然后客户端开始向第一个数据节点传输数据，第一个数据节点一小部分一小部分(4kb)地接收数据，将每个部分写入本地仓库，并且同时将该部分传输到第二个数据节点。第二个数据节点也是这样边收边传，一小部分一小部分地收，存储在本地仓库，同时传给第三个数据节点，第三个数据节点就仅仅是接收并存储了。这就是流水线式的复制。

HDFS 给应用提供了多种访问方式，可以通过 DFSShell 命令行与 HDFS 数据进行交互，也可以通过 Java API 调用，还可以通过 C 语言的封装 API 访问，并且提供了浏览器访问的方式。

最后，通过一个简单的小例子，展示一下如何使用 Java 访问 HDFS。

```
import java.io.InputStream;
import java.net.URL;
import org.apache.hadoop.fs.FsUrlStreamHandlerFactory;
import org.apache.hadoop.io.IOUtils;

public class DataReadByURL {
static{
```

```
URL.setURLStreamHandlerFactory(new FsUrlStreamHandlerFactory());
}
public static void main(String[] args) throws Exception{
InputStream in = null;
try{
in = new URL("hdfs://127.0.0.1:9000/data/mydata").openStream();
IOUtils.copyBytes(in,System.out,2048,false);
}finally{
IOUtils.closeStream(in);
}
}
}
```

上面讨论了利用 HDFS 的 URL 方式读取 HDFS 内文件内容的方法，下面讨论如何使用 HDFS 中的 API 读取 HDFS 内的文件。

HDFS 主要通过 FileSystem 类来完成对文件的打开操作。和 Java 使用 java.io.File 来表示文件不同，HDFS 文件系统中的文件是通过 Hadoop 的 Path 类来表示的。

FileSystem 通过静态方法 get(Configuration conf)获得 FileSystem 的实例。通过 FileSystem 的 open()、seek()等方法可以实现对 HDFS 的访问，具体的方法如下所示：

```
public FSDataInputStream open(Path f) throws IOException
public abstract FSDataInputStream open(Path
f, int bufferSize) throws IOException;
```

下面来看一个通过 HDFS 的 API 访问文件系统的例子：

```
import org.apache.hadoop.fs.*;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;

public class HDFSCatWithAPI {
public static void main(String[] args) throws Exception{
// 指定 Configuration
Configuration conf = new Configuration();
//定义一个 DataInputStream
FSDataInputStream in = null;
try{
//得到文件系统的实例
FileSystem fs = FileSystem.get(conf);
//通过 FileSystem 的 open 方法打开一个指定的文件
in = fs.open(new
Path("hdfs://localhost:9000/user/myname/input/fixFontsPath.sh"));
//将 InputStream 中的内容通过 IOUtils 的 copyBytes 方法复制到 System.out 中
IOUtils.copyBytes(in,System.out,4096,false);
//seek 到 position 1
in.seek(1);
//执行一边复制一边输出工作
```

```
IOUtils.copyBytes(in,System.out,4096,false);
}finally{
IOUtils.closeStream(in);
}
}

}
```

输出如下：

```
#!/bin/sh

# Licensed to the Apache Software Foundation (ASF) under one or more
# contributor license agreements.  See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version
 ...(中间内容略去)
</map:sitemap>
EOF
!/bin/sh

# Licensed to the Apache Software Foundation (ASF) under one or more
# contributor license agreements.  See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version
 ...(中间内容略去)
</map:sitemap>
EOF
```

## 2.4 Google 网页存储秘诀——BigTable

在前面的章节里，提到过分布式系统通常都采用 key/value 的形式存储数据。比如爬虫抓取页面后，页面的存储就是采用 key/value 形式。针对这一特点，Google 在 GFS 文件系统的基础上，设计了一种名为 BigTable 的 key/value 型分布式数据库系统。应用程序通常都不会直接操作 GFS 文件系统，而直接操作它的上一级存储结构——BigTable。这正如一般文件系统和关系数据库的道理一样。这一节，我们将详细讲述 BigTable 的相关知识。

### 2.4.1 详解 BigTable

Bigtable(下文中简称 BT)是用来存储大规模结构化数据的，它最多可以存储 $2^{50}$ 个字节，分布在几千个普通的服务器上。Google 的很多项目都使用 BT 存储数据，包括网页查询、Google 地图和 Google 金融。这些应用对 BT 的要求各不相同：数据大小(从 URL 到网页到卫星图像)不同，反应速度不同(从后端的大批处理到实时数据服务)。对于不同的项目需求，

BT 都提供了灵活高效的服务。

设计 BT 的目标是建立一个可以广泛应用的、高度扩缩的、高可靠性和高可用性的分布式数据库系统。现在，Google 至少有 60 多个产品和应用都采用 BT 作为存储结构。下面我们从几个方面介绍 BT。

### 1. BT 与关系数据库

BT 在很多地方和关系数据库类似：它采用了许多关系数据库的实现策略。但和它们不同的是，BT 采用了不同的用户接口。BT 不支持完全的关系数据模型，而是为客户提供了简单的数据模型，让客户来动态控制数据的分布和格式(就是只存储字符串，格式由客户来解释)，这样能大幅度地提高访问速度。数据的下标是行和列的名字，数据本身可以是任意字符串。BT 的数据是字符串，没有具体的类型。客户会把各种结构化或者半结构化的数据(比如说日期串)序列化成字符串。最后，可以使用配置文件来控制数据是放在内存里还是在硬盘上。

### 2. BT 逻辑存储结构

BT 的本质是一个稀疏的、分布式的、长期存储的、多维度的和排序的 Map。Map 的 key 是行关键字(Row)、列关键字(Column)和时间戳(Timestamp)。Value 是一个普通的 bytes 数组。如下所示：

```
(row:string, column:string,time:int64)->string
```

图 2.11 是 BT 存储网页的底层数据结构示意图(webtable)。其中，每个网页的内容与相关信息作为一行，无论它有多少列。

如图 2.11 所示，以反转的 URL 作为 Row，列关键字是“contents”、“anchor:my.look.ca”和“anchor:cnnsi.com”，时间戳是 t3, t5, t6, t8, t9 等。如果要查询 t5 时间 URL 为 www.cnn.com 的页面内容，可以使用{com.cnn.www,contents,t5}作为 key 去 BT 中查询相应的 value。

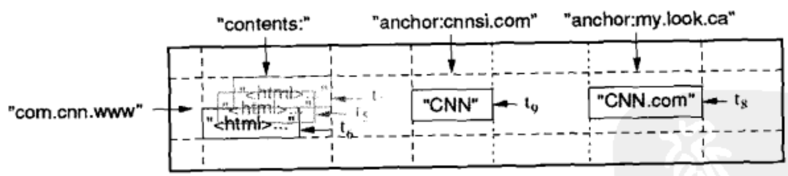



图 2.11　BT 存储示意图

表中的行(Row)可以是任意长度的字符串(目前最多支持 64KB，多数情况下 10～100 个字节就足够了)。在同一行下的每一个读写操作都是原子操作(不管读写这一行里多少个不同列)，这使得在对同一行进行并发操作时，用户对于系统行为更容易理解和掌控。

BT 通过行关键字在字典中的顺序来维护数据。一张表可以动态划分成多个连续“子表”(tablet)。这些“子表”由一些连续行组成，它是数据分布和负载均衡的单位。这使得读取较少的连续行比较有效率，通常只需要少量机器之间的通信即可。用户可以利用这个属性来

选择行关键字，从而达到较好的数据访问“局部性”。举例来说，在 webtable 中，通过反转 URL 中主机名的方式，可以把同一个域名下的网页组织成连续行。具体而言，可以把站点 maps.google.com/index.html 中的数据存放在关键字 com.google.maps/index.html 所对应的数据中。这种存放方式可以让基于主机和基于域名的分析更加有效。

一组列关键字组成了“列族”(column famliy)，这是访问控制的基本单位。同一列族下存放的所有数据通常都是同一类型的。“列族”必须先创建，然后才能在其中的“列关键字”下存放数据。“列族”创建后，其中任何一个“列关键字”都可使用。

“列关键字”用如下语法命名：“列族”：限定词。“列族”名必须是看得懂的字符串，而限定词可以是任意字符串。比如，webtable 可以有个“列族”叫 language，存放撰写网页的语言。我们在 language“列族”中只用一个“列关键字”，用来存放网页的语言标识符。该表的另一个有用的“列族”是 anchor。“列族”的每一个“列关键字”代表一个锚链接，访问控制、磁盘使用统计和内存使用统计，均可在“列族”这个层面进行。在图 2.11 的例子中，可以使用这些功能来管理不同应用：有的应用添加新的基本数据，有的读取基本数据并创建引申的“列族”，有的则只能浏览数据(甚至可能因为隐私权的原因不能浏览所有数据)。

BT 表中的每一个表项都可以包含同一数据的多个版本，由时间戳来索引。BT 的时间戳是 64 位整型，表示准确到毫秒的“实时”。需要避免冲突的应用程序必须自己产生具有唯一性的时间戳。不同版本的表项内容按时间戳倒序排列，即最新的排在前面。在图 2.11 中，“contents:”列存放一个网页被抓取的时间戳。

BT 使用 Google 分布式文件系统(GFS)来存储日志和数据文件。一个 BT 集群通常在一个共享的机器池中工作，池中的机器还运行着其他分布式应用，BT 和其他程序共享机器(BT 的瓶颈是 I/O 内存，可以和 CPU 要求高的程序并存)。BT 依赖集群管理系统来安排工作，在共享的机器上管理资源，处理失效机器并监视机器状态。

### 3. BT 内部存储格式——SSTable

BT 内部采用 SSTable 格式存储数据。SSTable 提供了一个从关键字到值的映射，关键字和值都可以是任意字符串，如图 2.12 所示。

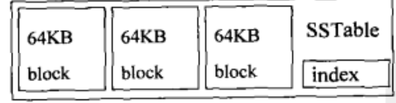



**图 2.12 SSTable 示意图**

在 SSTable 格式中，映射是排序的、存储的(不会因为掉电而丢失)、不可更改的。并且可以进行如下操作：

(1) 通过关键字查询相关的值。

(2) 根据给出的关键字范围遍历所有的关键字和值。

SSTable 内部包含一列数据块，通常每个块的大小是 64KB，但是大小是可以配置的。SSTable 中块索引(index)大小是 16 位。块索引(存储在 SSTable 的最后)用来定位数据块。当

打开 SSTable 的时候，块索引被读入内存。每次查找都可以用一次硬盘搜索完成，首先在内存中的索引里进行二分查找，获得数据块的位置，然后根据位置信息直接到硬盘读取数据块。最佳情况是：整个 SSTable 可以被放在内存里，这样一来就不必访问硬盘了。

#### 4. BT 的锁

BT 还依赖一个高度可用的分布式数据锁服务(Chubby)。一个 Chubby 由 5 个“活跃”的备份构成，其中一个被这些备份选成主备份，并且处理请求。这个服务只有在大多数备份都是“活跃”的并且互相通信的时候，才是“活跃”的。当有机器失效的时候，Chubby 使用一定的算法来保证备份的一致性。Chubby 提供了一个名字空间，里面包括目录和一系列文件。每个目录或者文件可以当成一个锁来用，读写文件操作都是原子操作。Chubby 客户端的程序库提供了对 Chubby 文件的一致性缓存。每个 Chubby 客户维护一个和 Chubby 通信的会话。如果客户不能在一定时间内更新自己的会话，会话就失效了。当一个会话失效时，其拥有的锁和打开的文件句柄都失效。Chubby 客户可以在文件和目录上登记回调函数，以获得改变或者会话过期的通知。

BT 使用锁服务来完成以下几个任务：

(1) 保证任何时间最多只有一个活跃的主备份。

(2) 存储 BT 数据的启动位置。

(3) 发现“子表”服务器，并处理 tablet 服务器失效的情况。

(4) 存储 BT 数据的“模式”信息(每张表的列信息)。

(5) 存储访问权限列表。在 Chubby 中，存储了 BT 的访问权限，如果 Chubby 不能访问，那么由于获取不到访问权限，因此 BT 就也不能访问了。

#### 5. BT 的主要组成部件

BT 主要由三个构件组成：

(1) 一个客户端的链接库。

(2) 一个主服务器。

(3) 许多“子表”服务器。“子表”服务器可以动态地从群组中被添加和删除，以适应流量的改变。

主服务器的作用是给“子表”服务器分配“子表”、探测“子表”服务器的增加和缩减、平衡“子表”服务器负载，以及回收 GFS 系统中文件的碎片。此外，它还可以创建模式表。

一个“子表”服务器管理许多子表(一般每个“子表”服务器可以管理 10 到 1000 个子表)。“子表”服务器处理它所管理的“子表”的读写请求，还可以将那些变得很大的“子表”分割。

像许多单主机的分布式存储系统一样，客户端数据不是通过主服务器来传输的：客户端要读写时直接与“子表”服务器通信。因为 BT 客户端并不依赖主服务器来请求“子表”本地信息，大多数客户端从不与主服务器通信。因此，实际上主机的负载往往很小。

一个 BT 群组可以存储大量的表。每一个表有许多“子表”，并且每个“子表”包含一

行上所有相关的数据。最初，每个表只包含一个子表。随着表的增长，自动分成了许多的“子表”，每个子表的默认大小为 100～200MB。

BT 用三层体系的 B+树来存储子表的地址信息，如图 2.13 所示。

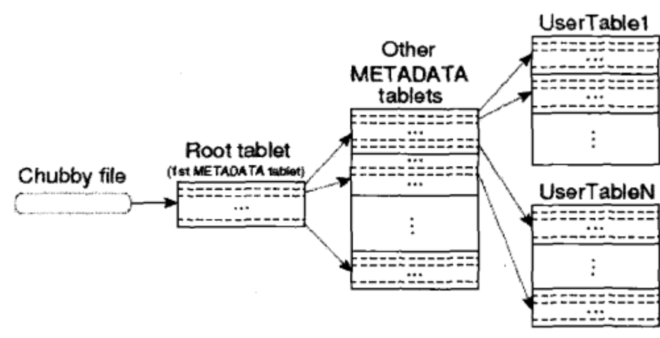



图 2.13　BT 的 B+树的体系结构

第一层是一个存储在 Chubby 中的文件，它包含“根子表”(root tablet)的地址。如图 2.13 所示，“根子表”包含一些“元数据表”(MetaData tablets)的地址信息。这些“元数据表”包含用户“子表”的地址信息。“根子表”是“元数据表”中的第一个“子表”，但它从不会被分割。

客户端缓存“子表”地址。如果客户端发现缓存的地址信息是错误的，那么它会递归地提升“子表”地址等级。如果客户端缓存是空的，寻址算法需要三个网络往返过程，包括一次从 Chubby 的读取。如果客户端缓存是过期的，那么寻址算法可能要用 6 个往返过程。尽管“子表”地址缓存在内存里，不需要 GFS 访问，但还是可以通过客户端预提取“子表地址”进一步降低性能损耗。

“子表”一次被分配给一个子表服务器。主服务器跟踪“活跃”的“子表”服务器集合以及当前“子表”对“子表服务器”的分配状况。当一个“子表”还没有被分配，并且有一个“子表”服务器是可用的，主机就通过传输一个“子表”装载请求到“子表”服务器来分配“子表”。

BT 使用 Chubby 来跟踪“子表”服务器。当“子表”服务器启动时，它在一个特别的 Chubby 目录中创建一个文件，并且获得一个互斥锁。主机通过监听这个目录(服务器目录)来发现“子表”服务器。如果“子表”服务器丢失了自己的互斥锁，就会停止为它的“子表”服务。

主机负责探测何时“子表”服务器不再为它的“子表”服务，以便可以尽快地分配那些“子表”。为了达到这个目的，主机会周期性地询问每个“子表”服务器的锁的状态。如果一个“子表”服务器报告它丢失了锁，主机会尝试在服务器的文件中获取一把互斥锁。如果主机能获得这把锁，则表示 Chubby 是可用的并且“子表”服务器已经失效。因此主机通过删除“子表”服务器的服务文件来确保它不会再工作。一旦服务文件被删除，主机可以把先前分配给这台服务器的所有“子表”移到未分配的“子表”集合中。

“子表”的持久化状态存储在 GFS 文件里，如图 2.14 所示。

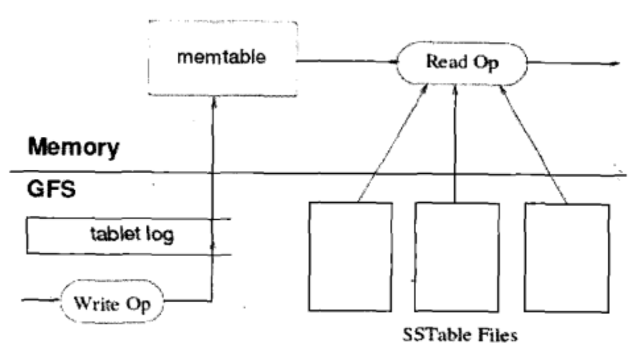



图 2.14　子表持久化

每次更新“子表”前都要更新“子表”的重做日志(redo log)。并且最近更新的内容(已经提交的但还没有写入到磁盘的内容)会存放在内存中，称为 memtable。之前的更新(已经提交的并且固化在磁盘的内容)会被持久化到一系列的 SSTable 中。

当一个写操作请求过来时，“子表”服务器会先写日志，当提交的时候，就把这些更新写入 memtable 中。之后等系统不繁忙的时候，就写入 SSTable 中(这个过程和 Oracle 数据库写操作基本一致)。

如果请求是读操作，则可以根据当前的 memtable 和 SSTable 中的内容进行合并，然后对请求返回结果。因为 memtable 和 SSTable 有相同的结构，因此，合并是一个非常快的操作。

### 2.4.2　开源 BigTable——HBase

Google 提出“云计算”理论之后，很多团队都开始根据这一理论，开发自己的云计算存储模型。其中，HBase 是一个比较成功的实例。本小节就讲述 HBase 的相关知识。

HBase 是一个 Apache 开源项目，它的目标是提供一个在 Hadoop 分布式环境中运行的类似于 BT 的存储系统。正如同 Google 将 BT 架设在自己的分布式存储系统 GFS 中一样，HBase 是基于 HDFS 的。

在 HBase 中，数据在逻辑上被组织成为表、行和列。客户可以使用类似于 iterator 的接口来遍历行数据，同时也可以通过行值来获取列值。同一个行对应多个不同版本的列值。

#### 1. 数据模型

HBase 使用的数据模型和 BT 类似。应用程序将数据行存储在“打上标签”的表中。一个数据行包含一个排好序的行键值和任意长度的列值。

列名的格式为“<family>:<label>”，其中<family> 与 <label> 可以为任意的字节数组。表的<family>集合是固定的。如果希望修改表的<family>字段，需要管理员来操作；但可以随时增加<label>，而无需事先声明它。HBase 在磁盘上按照<family>储存数据，所以<family>里的所有项应该有相同的读/写方式。

默认每次将一行数据进行锁定，行数据的写入是原子操作，写入时这行数据将被锁定，

同时进行读和写的操作。

### 2. 概念模型

从概念上来看，一个表由多个数据行组成，每个数据行中的列不一定有值。表 2.2 展示了一个来自 BigTable 的数据。

表 2.2　BigTable 中的数据

| Row Key | Time Stamp | Column "contents:" | Column "anchor:" | | Column "mime:" |
|---|---|---|---|---|---|
| "com.cnn.www" | t9 | | "anchor:cnnsi.com" | "CNN" | |
| | t8 | | "anchor:my.look.ca" | "CNN.com" | |
| "com.cnn.www" | t6 | "<html>..." | | | "text/html" |
| | t5 | "<html>..." | | | |
| | t3 | "<html>..." | | | |

### 3. 物理存储模型

从概念上看，表数据是按照稀疏的行来存储的，但是物理上，它们是按照列来存储的。理解这个概念对考虑表的设计和程序设计是非常重要的。

回到先前的那个例子，物理存储结构如表 2.3～表 2.5 所示。

表 2.3　物理存储结构(1)

| Row Key | Time Stamp | Column "contents:" |
|---|---|---|
| "com.cnn.www" | t6 | "<html>..." |
| | t5 | "<html>..." |
| | t3 | "<html>..." |

表 2.4　物理存储结构(2)

| Row Key | Time Stamp | Column "anchor:" | |
|---|---|---|---|
| "com.cnn.www" | t9 | "anchor:cnnsi.com" | "CNN" |
| | t8 | "anchor:my.look.ca" | "CNN.com" |

表2.5 物理存储结构(3)

| Row Key | Time Stamp | Column "mime:" |
|---|---|---|
| "com.cnn.www" | t6 | "text/html" |

从上面的表中可以看出：空的列并没有被存储。所以当请求列值为"contents:"的时候，会返回空值。

如果请求数据的时候，没有提供时间戳(time stamp)的值，那么请求的数据将返回最新的值(时间戳最近的值)，并且在实际的存储中，所有的值都是按照时间戳降序排列的。

### 4. 行的范围：区域

数据表是由一系列按升序排序的数据行组成的，在同一行中，数据列按名字的升序排列，时间戳按降序排列。物理上，数据表被拆分成由多个数据行组成的区域(这个概念在BT中就是tablet)。区域包含从开始行(包含)到终止行(排除)。所有区域构成一个数据表。与BT不同的是，在BT中，数据行区域由表名和结束行值决定，而HBase的区域由表名和开始行值决定。

区域中的每一个“列族”都由一个名为HStore的对象管理。每个HStore由一个或多个MapFiles(Hadoop中的一个文件类型)组成。MapFiles的概念类似于Google的SSTable。MapFiles一旦被关闭，就是不可修改的类型了。MapFiles被存储在Hadoop的HDFS中，其与SSTable的不同之处如下：

- MapFiles目前还无法进行内存的映射。
- MapFiles由一些稀疏的索引文件来维护，而SSTable在其文件的最后。
- HBase扩展了MapFiles，所以使用布隆过滤器的时候可以提高查找的性能。

### 5. 架构与实现

HBase由以下三个主要构件组成。

1) HBaseMaster(类似于BT主服务器)

HBaseMaster负责给HRegionServer分配区域。分配的第一个区域就是ROOT区域，它用于定位所有的META区域。一旦所有的META区域被分配好，master便开始分配用户区域给HReginServer，同时维护每个HReginServer的负载平衡。

HBaseMaster还含有一个指向包含有ROOT区域的HReginServer地址。

同时，HBaseMaster会监控每台HReginServer的健康状况，如果某台HReginServer不可用，它将会把不可用的HReginServer替换成其他HReginServer。

另外，HBaseMaster还负责对数据表进行管理，比如让表处于有效(online)或失效(offline)的状态，改变表的结构(增加或者减少列族)，等等。

与BT不同的是，如果HBaseMaster失效了，整个集群就会关闭。在BT中，即使主服务器失效了，Tablet服务器还是可以提供服务的。而HBase使用单点访问模式：所有的HReginServer都访问HBaseMaster。

2) HRegionServer(类似于 BT tablet 服务器)

HReginServer 负责处理用户读和写的操作。HReginServer 通过与 HBaseMaster 通信获取需要服务的数据表，并向 HBaseMaster 反馈自己的运行状况。HReginServer 的任务主要包括以下几类：

(1) 写请求(Write Request)

当一个写的请求到来时，它首先会写到一个称为 HLog 的日志模块中。HLog 被缓存在内存中，称为 Memcache，每一个 HStore 只能有一个 Memcache。

(2) 读请求(Read Request)

当读取的请求到来时，HReginServer 会先在 Memcache 中寻找该数据，当找不到时，才会去 MapFiles 中寻找。

(3) 刷新缓存(Cache Flushe)

当 Memcache 到达配置的大小以后，会创建一个 MapFile，将其写到磁盘中。这将减少 HReginServer 的内存压力。

在读和写的过程中，Cache Flushe 会经常发生。当创建一个新的 MapFile 时，读和写的操作会被挂起，直到新的 MapFile 创建好，并被加入 HStroe 的管理中才可以使用。

(4) 压缩(Compaction)

当一定数量的 MapFiles 超过一个配置的阈值之后，压缩操作就开始执行。压缩操作的主要工作就是周期性地将一些 MapFiles 合并成一个 MapFile。

在执行压缩操作的过程中，HReginServer 的读和写操作将被挂起，直到操作执行完毕。

(5) 区域切分(Region Split)

当一个 HStore 所管理的 MapFiles 超过一个配置(当前是 256MB)的值以后，将会执行区域的切分操作。区域的切分操作将原先的区域对半分割为两个新的区域。

在进行区域切分的操作过程中，读和写的操作将被挂起，直到完成为止。

3) HBase 客户端(由 org.apache.hadoop.hbase.client.HTable 定义)

HBase 客户端负责寻找提供需求数据的 HRegin Server。在这个过程中，HBase 客户端将首先与 HBase 主服务器通信，找到 ROOT 区域。这个操作是客户端和主服务器之间仅有的通信操作。

一旦 ROOT 区域找到以后，客户端就可以通过扫描 ROOT 定位实际提供数据的 HRegin Server。

当定位到提供数据的 HRegin Server 以后，客户端就可以通过这个 HRegin Server 找到需要的数据了。

这些信息将会被客户端缓存起来，当下次请求的时候，就不需要再走上面的这个流程了。

当这些区域中的某个区域不可用时，客户端将会逆向执行上面的过程，直到找到实际提供数据的 HRegin Server 为止。

接下来，简单介绍如何操作 HBase。HBase 与我们常用数据库的最大差别就是列存储和无数据类型，所有数据都以 string 类型存储。并且如果在 HBase 中存储了 5 个字段，但实际只有 4 个字段有值，那么为空的那个字段不占用空间。具体使用请参考下面代码：

```
/**
```

```
 * 定义几个常量
 */
public static HBaseConfiguration conf = new HBaseConfiguration();
static HTable table = null;
/**
 * 创建hbase table
 * @param table
 * @throws IOException
 */
public static void creatTable(String tablename) throws IOException {
    HBaseAdmin admin = new HBaseAdmin(conf);
    if (!admin.tableExists(new Text(tablename))) {
        HTableDescriptor tableDesc = new HTableDescriptor(tablename);
        tableDesc.addFamily(new HColumnDescriptor("ip:"));
        tableDesc.addFamily(new HColumnDescriptor("time:"));
        tableDesc.addFamily(new HColumnDescriptor("type:"));
        tableDesc.addFamily(new HColumnDescriptor("cookie:"));
        //注意这个C列，下面简单以此列来说明列存储
        tableDesc.addFamily(new HColumnDescriptor("c:"));
        admin.createTable(tableDesc);
        System.out.println("table create ok!!!");
    } else {
        System.out.println("table Already exists");
    }
}
/**
 * 录入数据
 * @throws Exception
 */
public static void insertData() throws Exception{
    //读取日志文件
    BufferedReader reader = new BufferedReader(new FileReader("log file
name"));
    if(table==null)
        table = new HTable(conf, new Text(tablename));
    String line;
    while((line = reader.readLine()) != null){
        //这里就不介绍了，先前有说明
        LogAccess log = new LogAccess(line);
        //这里使用time+cookie为row关键字，确保不重复，如果cookie记录有重复，
        //将区别对待，这里暂不多做说明
        String row = createRow(log.getTime(),log.getCookie());

        long lockid = table.startUpdate(new Text(row));
        if(!log.getIp().equals("") && log.getIp()!=null)
            table.put(lockid, new Text("ip:"), log.getIp().getBytes());
        if(!log.getTime().equals("") && log.getTime()!=null)
```

```
                table.put(lockid, new Text("time:"),
log.getTime().getBytes());
            if(!log.getType().equals("") && log.getType()!=null)
                table.put(lockid, new Text("type:"),
log.getType().getBytes());
            if(!log.getCookie().equals("") && log.getCookie()!=null)
                table.put(lockid, new Text("cookie:"),
log.getCookie().getBytes());
            if(!log.getRegmark().equals("") && log.getRegmark()!=null)
                table.put(lockid, new Text("c:_regmark"),
log.getRegmark().getBytes());
            if(!log.getRegmark2().equals("") && log.getRegmark2()!=null)
                table.put(lockid, new Text("c:_regmark2"),
log.getRegmark2().getBytes());
            if(!log.getSendshow().equals("") && log.getSendshow()!=null)
                table.put(lockid, new Text("c:_sendshow"),
log.getSendshow().getBytes());
            if(!log.getCurrenturl().equals("") && log.getCurrenturl()!=null)
                table.put(lockid, new Text("c:_currenturl"),
log.getCurrenturl().getBytes());
            if(!log.getAgent().equals("") && log.getAgent()!=null)
                table.put(lockid, new Text("c:_agent"),
log.getAgent().getBytes());
            //存入数据
            table.commit(lockid);
        }
}
```

本小节是针对 HBase 的简单介绍，如果想深入了解的话，请参考相关的内容。

## 2.5 Google 的成功之道——MapReduce 算法

MapReduce 算法是处理/产生海量数据集的编程模型。在 MapReduce 算法中，用户指定一个 map()函数，通过这个 map()函数处理键/值(key/value)对，产生一系列的中间键/值(key/value)对，并且使用一个 reduce()函数来合并具有相同键 (key)的中间键值(key/value)对中的值(value)。现实生活中，很多任务的实现都是基于这个模式的。

使用 MapReduce 算法的程序可以将任务自动分布到一个由普通机器组成的超大规模集群上并发执行。大多数实现 MapReduce 算法的框架(比如 Hadoop)都会解决输入数据的分布细节，跨越机器集群的程序执行调度，处理机器的失效和机器之间的通信等问题。使用框架来实现 MapReduce 算法，程序员可以不需要有什么并发处理或者分布式系统的经验，就可以处理超大规模的分布式系统的资源。

考虑这样一个例子，在很大的文档集合中统计每个单词出现的次数。写出如下类似的伪代码：

```
map(String key, String value):
```

```
    // key: 文档名
    // value:文档内容
    for each word w in value:
        EmitIntermediate(w, "1");
reduce(String key, Iterator values):
    // key: a word
    // values: 一个针对这个 word 的计数列表(每个列表记录文档中 word 出现的次数)
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

map()函数检查每个单词，并且每检查一次，就将当前检查的单词的计数器加 1。Reduce()函数将特定单词出现的次数进行合并。

上面的例子是用字符串作为输入和输出的，从概念上讲，map()和 reduce()函数的输入/输出类型遵循以下原则：

```
map (k1,v1)              →    list(k2,v2)
reduce (k2,list(v2))     →    list(v2)
```

也就是说，输入的键(key)值和输出的键(key)值属于不同的域。而中间的键值和输出的键值属于相同的域。

MapReduce 算法在实际的系统中使用非常广泛，下面是一些简单有趣的例子，它们都可以使用 MapReduce 算法来进行计算。

- 分布式 Grep：如果 map()函数检查输入行，满足条件的时候，map()函数就把本行输出。Reduce()函数就是一个直通函数，简单地把中间数据输出就可以了。
- URL 访问频率统计：map()函数处理请求和应答(URL, 1)的 log。Reduce()函数把所有相同的 URL 的值合并，并且输出一个成对的(URL, 总个数)。
- 逆向 Web-Link 图：map()函数输出所有包含指向目标 URL 的网页，用(目标 URL, 源 URL)这样的结构对输出。Reduce()函数聚合所有关联相同目标 URL 的列表、源 URL 并且输出一个(目标 URL,list(源 URL))的结构。
- 主机关键向量指标(Term-Vector per Hosts)：关键词向量指标简而言之就是指在一个文档或者一组文档中的重点词出现的频率，用(word,frequency)表达。map()函数计算每一个输入文档(主机名字是从文档的 URL 取出的)的关键词向量，然后输出(hostname，关键词向量(Term-Vector))。Reduce()函数处理所有相同主机的所有文档关键词向量。去掉不常用的关键词，并且输出最终的(hostname, 关键词向量)对。
- 逆序索引：map()函数分析每一个文档，并且产生一个序列(word, document ID)组。Reduce()函数处理指定 word 的所有序列组，并且对相关的 document ID 进行排序，输出一个(word,list(document ID))组。所有的输出组，组成一个简单的逆序索引。通过这种方法可以很容易地保持关键词在文档库中的位置。
- 分布式排序：map()函数从每条记录中抽取关键字，并且产生(key,record)对。Reduce()函数原样输出所有的关键字对。

### 2.5.1 详解 MapReduce 算法

map()函数把输入数据进行切割(比如分为 M 块)之后，分布到不同的机器上执行(例如前面介绍的单词统计例子，可以把每一个文件分配到一台机器上执行)。Reduce()函数通过产生的键 key(例如可以根据某种分区函数(比如 hash(key) mod R)，R 的值和分区函数都是由用户指定)将 map()的结果集分成 R 块，然后分别在 R 台机器上执行。

图 2.15 是 MapReduce 算法示意图。当用户程序调用 MapReduce 函数时，就会引起如下的操作。

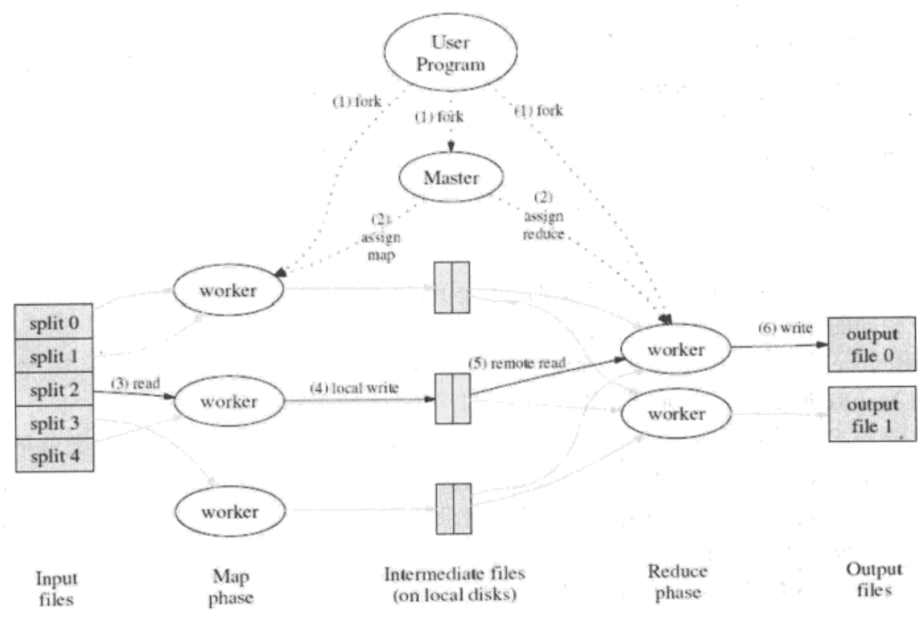



图 2.15 MapReduce 运行机制

(1) MapReduce 函数库首先把输入文件分成 M 块，每块大概 16MB 到 64MB。接着在集群的机器上执行处理程序。

如图 2.14 所示，MapReduce 算法运行过程中有一个主控程序，称为 master。主控程序会产生很多作业程序，称为 worker。并且把 M 个 map 任务和 R 个 reduce 任务分配给这些 worker，让它们去完成。

(2) 被分配了 map 任务的 worker 读取并处理相关的输入(这里的输入是指已经被切割的输入小块 splite)。它处理输入的数据，并且将分析出的键/值(key/value)对传递给用户定义的 reduce()函数。map()函数产生的中间结果键/值(key/value)对暂时缓冲到内存。

(3) map()函数缓冲到内存的中间结果将被定时刷写到本地硬盘，这些数据通过分区函数分成 R 个区。这些中间结果在本地硬盘的位置信息将被发送回 master，然后这个 master 负责把这些位置信息传送给 reduce()函数的 worker。

(4) 当 master 通知了 reduce()函数的 worker 关于中间键/值(key/value)对的位置时，worker 调用远程方法从 map()函数的 worker 机器的本地硬盘上读取缓冲的中间数据。当 reduce()函数的 worker 读取到了所有的中间数据，它就使用这些中间数据的键(key)进行排序，这样可以使得相同键(key)的值都在一起。如果中间结果集太大了，那么就需要使用外排序。

(5) reduce()函数的 worker 根据每一个中间结果的键(key)来遍历排序后的数据，并且把键(key)和相关的中间结果值(value)集合传递给 reduce()函数。reduce()函数的 worker 最终把输出结果存放在 master 机器的一个输出文件中。

(6) 当所有的 map 任务和 reduce 任务都已经完成后，master 激活用户程序。在这时，MapReduce 返回用户程序的调用点。

(7) 当以上步骤成功结束以后，MapReduce 的执行数据存放在总计 R 个输出文件中(每个输出文件都是由 reduce 任务产生的，这些文件名是用户指定的)。通常，用户不需要将这 R 个输出文件合并到一个文件，他们通常把这些文件作为输入传递给另一个 MapReduce 调用，或者用另一个分布式应用来处理这些文件，并且这些分布式应用把这些文件看成为输入文件由于分区(partition)成为的多个块文件。

### 2.5.2 MapReduce 容错处理

由于 MapReduce 函数库是设计用于在成百上千台机器上处理海量数据的，所以这个函数库必须考虑到机器故障的容错处理。下面就详细介绍 MapReduce 的容错处理机制。

如图 2.14 所示，master 会定期发送命令轮询每一台 worker 机器。如果在一定时间内有一台 worker 机器一直没有响应，master 就认为这个 worker 失效了。所有这台 worker 完成的 map 任务都被设置成为它们的初始空闲状态，并且因此可以被其他 worker 调度执行。类似的，所有这个机器上正在处理的 map 任务或者 reduce 任务都被设置成为空闲状态，被其他 worker 重新执行。

在失效机器上的已经完成的 map 任务还需要再次重新执行，这是因为中间结果存放在这个失效的机器上，所以导致中间结果无法访问。已经完成的 recude 任务无需再次执行，因为它们的结果已经保存在全局的文件系统中了。

当 map 任务首先由 A worker 执行，随后被 B worker 执行的时候(因为 A 机器失效了)，所有执行 reduce 任务的 worker 都会被通知。所有还没有来得及从 A 上读取数据的 worker 都会从 B 上读取数据。

大多数 MapReduce 函数库都能够有效地支持很多 worker 失效的情况。比如，在一个网络例行维护时，可能会导致每次大约有 80 台机器在几分钟之内不能访问。master 会简单地使这些不能访问的 worker 上的工作再执行一次，并且继续调度进程，直到所有任务都完成。

在 master 中，定期会设定检查点(checkpoint)。如果 master 任务失效了，可以从上次最后一个检查点开始启动另一个 master 进程。

map 和 reduce 任务可靠性是由输出进行原子提交来完成的。每一个正在进行的任务把输出写到一个私有的临时文件中。当全部写完之后，进行提交操作，并把这些临时文件变为永久保存的文件。

### 2.5.3 MapReduce 实现架构

当前，针对 MapReduce 算法的实现架构如图 2.16 所示。主应用程序(Main Application)协调其他实例进行 map()或者 reduce()操作，然后从每个 reduce 操作中收集结果。

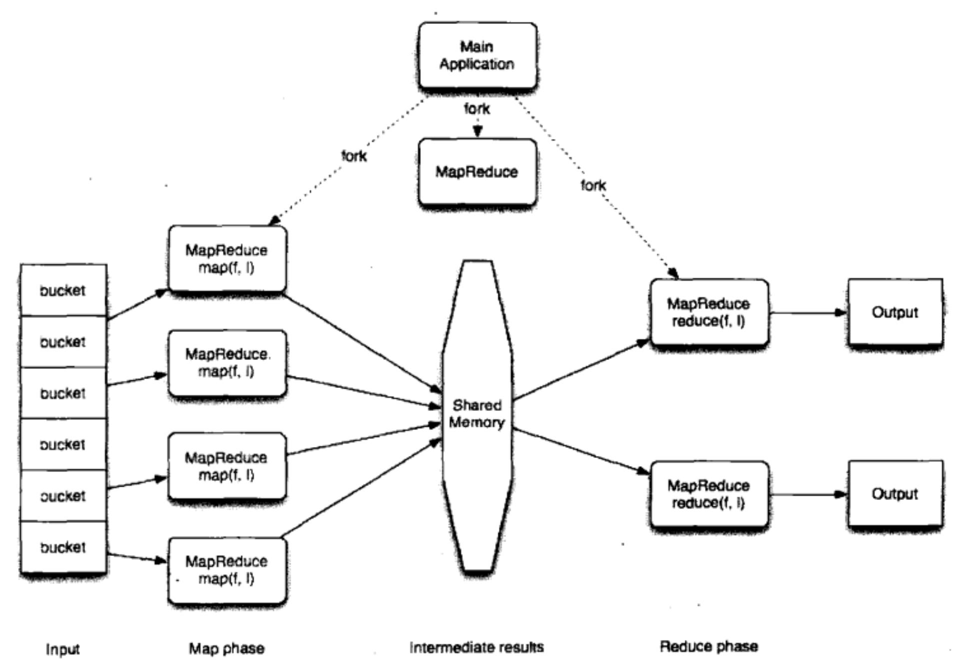


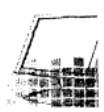

图 2.16　通用 MapReduce 实现架构

主应用程序负责把基础的数据集分解到“桶”(bucket)中。桶的最佳大小依赖于应用、结点的数量和可用的 I/O 带宽。这些“桶”通常存储在磁盘，如果有必要也可能分散到主存中，这依赖于具体的应用。“桶”将作为 map()函数的输入。

主应用程序也负责调度和分散几个 MapReduce 的核心备份，这几个备份是完全一致的。每个核心备份中有一个控制者，这个控制者会持续跟踪每个 map()和 reduce()任务的状态，并且可以作为 map()和 reduce()任务之间路由中间结果的管道。每个 map() 任务处理器完全指派给“桶”，然后产生一个保存到共享存储区域(Shared Memery)的中间结果集。共享存储可以设计成分布缓存、磁盘或其他设备等形式。当一个新的中间结果写入共享存储区域后，任务就向控制者发出通知，并提供指向其共享存储位置的句柄。

当新的中间结果可用时，控制者分配 reduce()任务。这个任务通过应用独立的中间键值(key/value)来实现排序，使相同的数据能聚集在一起，以提供更快的检索。大块的结果集可以进行外部排序，reduce()任务遍历整个排序的数据，把唯一的键和分类的结果传递到用户的 reduce()函数进行处理。

经过 map()和 reduce()的过程，当所有的“桶”都用完时，全部的 reduce()任务都会通知





控制者，以说明它们的结果产生了。控制者就向主应用程序发出检索这个结果的信号。主应用程序可能就直接操作这些结果，或者重新分配到不同的 MapReduce 控制者和任务进行进一步的处理。

当前，很多企业应用系统都是建立在 Java 技术上，它们依赖于已有的文件系统、通信协议和应用栈。

一个基于 Java 的 MapReduce 实现应该考虑到已存在的数据存储设备，将来部署的结构里面支持哪种协议，有哪些内部 API 和支持部署哪种第三方产品(开源的或商业的)。图 2.17 显示了通常的架构是如何通过映射到已有的、健壮的 Java 开源架构来实现的。

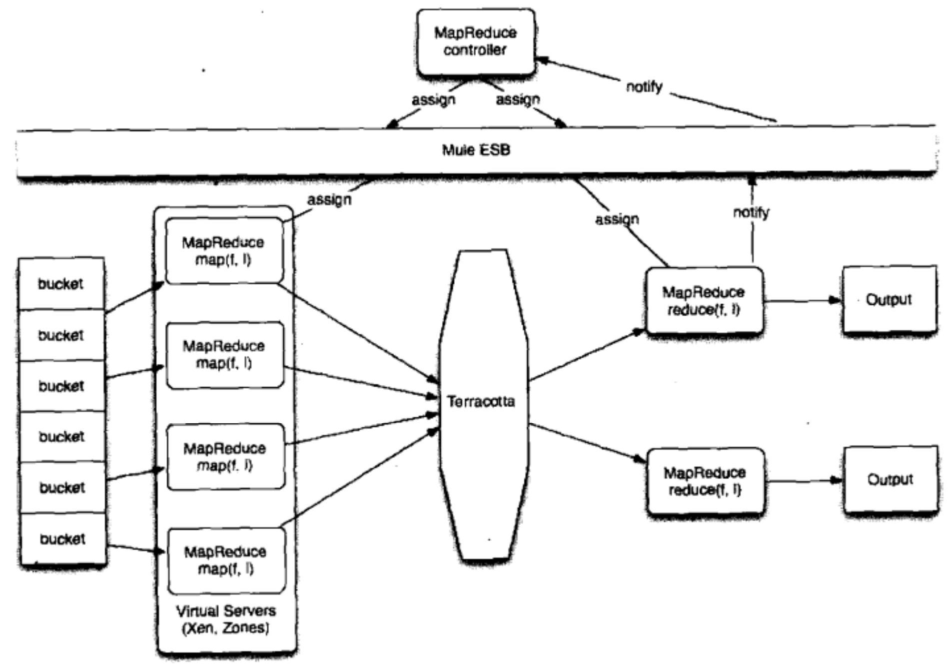



图 2.17 MapReduce Java 开源实现架构

这个架构采用了已有的工具，比如 Terracotta 和 Mule，它们经常用来架构企业级应用系统。以物理或虚拟系统形式存在的“白盒子”通过简单的配置和部署，设计成 MapReduce 群组中的一部分。为了提高效率，一个庞大的系统可以分解到多个虚拟机器上，如果需要可以分配更多的结点。

Terracotta 集群技术是 map 和 reduce 任务之间共享数据的良好选择，因为它把 map()函数和 reduce()函数之间的通信过程，包括共享文件或者使用 RPC 调用以及初始处理结构都做了良好的封装。

从前面的描述可知，Map 和 reduce 任务是在同一个核心应用中实现的。用来共享中间结构集的数据结构可以保持在内存的数据结构中，通过 Terracotta 透明地共享交换。

由跨域集群的 MapReduce 产生的进程内通信问题，自从 Terracotta 在运行时掌管着这

些共享数据结构后就不存在了。在 Terracotta 中，所有的 map()任务都需要标记内存中的中间结果集，然后 reduce()任务就根据标记直接在内存中提取它们。

控制者和主应用程序都是通过 Mule 的 ESB 传递信号的。通过主流的企业应用协议或者完全的原始 TCP/IP Sockets、Mule 支持在内存中进行同步和异步的数据传输。Mule 可用于在同一台机器执行的应用系统、跨越不同的数据中心或者完全不同的地方且被程序员分开标识的本地终端结点之间传递输出结构集。

大多数 Java 语言的 MapReduce 算法的实现都是基于 Hadoop 的。Hadoop 是一个开源的点对点、通用的 MapReduce 实现。有关它的介绍，我们在下一节详细讲解。

### 2.5.4 Hadoop 中的 MapReduce 简介

Hadoop 是 Apache 下的一个分布式并行计算框架。Hadoop 的核心设计思想是 MapReduce 和 HDFS，MapReduce 在前文中已经做了详细的介绍；而 HDFS 是 Hadoop Distributed File System 的缩写，即 Hadoop 的分布式文件系统，它也是基于我们前面介绍的 Google File System 的原理开发的，它为分布式计算存储提供底层支持。

在 Hadoop 官方文档介绍了 Hadoop 中 MapReduce 的三个步骤：map(主要是分解并行的任务)、combine(主要是为了提高 reduce 的效率)和 reduce(把处理后的结果再汇总起来)。

#### 1. map

由于 map 是并行地对输入的文件集进行操作，所以它的第一步(FileSplit)就是把文件集分割成一些子集。如果单个的文件大到影响查找效率时，它会被分割成一些小的文件。要指出的是，分割这一步是不知道输入文件的内部逻辑结构的。比如，以行为逻辑分割的文本文件会被以任意的字节界限分割，所以这个具体分割要由用户自己指定。然后每个文件分割体都会对应地有一个新的 map 任务。

当单个 map 任务开始时，它会对每个配置过的 reduce 任务开启一个新的输出流(writer)，这个输出流会读取文件分割体。Hadoop 中的类 InputFormat 用于分析输入文件并产生键值(key/value)对。

Hadoop 中的 Mapper 类是一个可以由用户实现的类，经过 InputFormat 类分析的键值(key/value)对都传给 Mapper 类，这样，用户提供的 Mapper 类就可以进行真正的 map 操作。

当 map 操作的输出被收集后，它们会被 Hadoop 中的 Partitioner 类以指定的方式区分地写入输出文件里。

#### 2. combine

当 map 操作输出它的键值(key/value)对时，出于性能和效率的考虑，Hadhoop 框架提供了一个合成器(combine)。有了这个合成器，map 操作所产生的键值(key/value)对就不会马上写入输出文件，它们会被收集在一些 list 中，一个 key 值对应一个 list，当写入一定数量的键值(key/value)对时，这部分 list 会被合成器处理。

比如，hadoop 案例中的 word count 程序，它的 map 操作输出是(word,1)键值对，在 map 操作的输入中，词的计数可以使用合成器来加速。合成操作会在内存中收集处理 list，一个

词一个 list。当一定数量的键值对输出到内存中时，就调用合成操作的 reduce 方法，每次都以一个唯一的词为 key，values 是 list 的迭代器，然后合成器输出(word, count-in-this-part-of-the-input)键值对。

3．reduce

当一个 reduce 任务开始时，它的输入分散在各个节点上的 map 的输出文件里。如果在分布式的模式下，需要先把这些文件拷贝到本地文件系统上。

一旦所有的数据都被拷贝到 reduce 任务所在的机器上时，reduce 任务会把这些文件合并到一个文件中。然后这个文件会被合并分类，使得相同的 key 的键值对可以排在一起。接下来的 reduce 操作就很简单了，顺序地读入这个文件，将键(key)所对应的值(values)传给 reduce 方法完成之后再读取一个键(key)。

最后，输出由每个 reduce 任务的输出文件组成。而它们的格式可以由 JobConf.setOutputFormat 类指定。

### 2.5.5　wordCount 例子的实现

还记得我们在 2.5 节介绍的单词统计的例子么，它是学习 map/reduce 最好的例子。本节我们将使用 Java 语言来实现这个例子，其中用到了 hadoop 开源云计算包(hadoop 相关 API 请参见 hadoop 站点)。好了，先让我们来看看代码吧。

```
package org.myorg;
import java.io.*;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.filecache.DistributedCache;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;
public class WordCount extends Configured implements Tool {
   public static class Map extends MapReduceBase implements
   Mapper<LongWritable, Text, Text, IntWritable> {
     static enum Counters { INPUT_WORDS }
     private final static IntWritable one = new IntWritable(1);
     private Text word = new Text();
     private boolean caseSensitive = true;
     private Set<String> patternsToSkip = new HashSet<String>();
     private long numRecords = 0;
     private String inputFile;
     public void configure(JobConf job) {
       caseSensitive = job.getBoolean("wordcount.case.sensitive", true);
       inputFile = job.get("map.input.file");
       if (job.getBoolean("wordcount.skip.patterns", false)) {
         Path[] patternsFiles = new Path[0];
         try {
```

```
      patternsFiles = DistributedCache.getLocalCacheFiles(job);
    } catch (IOException ioe) {
      System.err.println("Caught exception while getting cached files: "
      + StringUtils.stringifyException(ioe));
    }
    for (Path patternsFile : patternsFiles) {
      parseSkipFile(patternsFile);
    }
  }
}
private void parseSkipFile(Path patternsFile) {
  try {
    BufferedReader fis = new BufferedReader(new
    FileReader(patternsFile.toString()));
    String pattern = null;
    while ((pattern = fis.readLine()) != null) {
      patternsToSkip.add(pattern);
    }
  } catch (IOException ioe) {
    System.err.println("Caught exception while parsing the cached file '"
    + patternsFile + "' : " + StringUtils.stringifyException(ioe));
  }
}
public void map(LongWritable key, Text value, OutputCollector<Text,
IntWritable> output, Reporter reporter) throws IOException {
  String line = (caseSensitive) ? value.toString():
  value.toString().toLowerCase();
  for (String pattern : patternsToSkip) {
    line = line.replaceAll(pattern, "");
  }
  StringTokenizer tokenizer = new StringTokenizer(line);
  while (tokenizer.hasMoreTokens()) {
    word.set(tokenizer.nextToken());
    output.collect(word, one);
    reporter.incrCounter(Counters.INPUT_WORDS, 1);
  }
  if ((++numRecords % 100) == 0) {
    reporter.setStatus("Finished processing " + numRecords + " records "
    + "from the input file: " + inputFile);
  }
}
}
public static class Reduce extends MapReduceBase implements Reducer<Text,
IntWritable, Text, IntWritable> {
  public void reduce(Text key, Iterator<IntWritable> values,
  OutputCollector<Text, IntWritable> output, Reporter reporter) throws
  IOException {
```

```
      int sum = 0;
      while (values.hasNext()) {
        sum += values.next().get();
      }
      output.collect(key, new IntWritable(sum));
    }
  }
  public int run(String[] args) throws Exception {
    JobConf conf = new JobConf(getConf(), WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    List<String> other_args = new ArrayList<String>();
    for (int i=0; i < args.length; ++i) {
      if ("-skip".equals(args[i])) {
        DistributedCache.addCacheFile(new Path(args[++i]).toUri(), conf);
        conf.setBoolean("wordcount.skip.patterns", true);
      } else {
        other_args.add(args[i]);
      }
    }
    FileInputFormat.setInputPaths(conf, new Path(other_args.get(0)));
    FileOutputFormat.setOutputPath(conf, new Path(other_args.get(1)));
    JobClient.runJob(conf);
    return 0;
  }
  public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new Configuration(), new WordCount(), args);
    System.exit(res);
  }
}
```

下面是 WordCount 的运行样例及结果。

输入样例:

```
$ bin/hadoop dfs -ls /usr/joe/wordcount/input/
/usr/joe/wordcount/input/file01
/usr/joe/wordcount/input/file02

$ bin/hadoop dfs -cat /usr/joe/wordcount/input/file01
Hello World, Bye World!

$ bin/hadoop dfs -cat /usr/joe/wordcount/input/file02
```

```
Hello Hadoop, Goodbye to hadoop.
```

运行程序：

```
$ bin/hadoop jar /usr/joe/wordcount.jar org.myorg.WordCount
/usr/joe/wordcount/input /usr/joe/wordcount/output
```

输出：

```
$ bin/hadoop dfs -cat /usr/joe/wordcount/output/part-00000
Bye 1
Goodbye 1
Hadoop, 1
Hello 2
World! 1
World, 1
hadoop. 1
to 1
```

现在通过 DistributedCache 插入一个模式文件，文件中保存了要被忽略的单词模式。

```
$ hadoop dfs -cat /user/joe/wordcount/patterns.txt
\.
\,
\!
to
```

再运行一次，这次使用更多的选项：

```
$ bin/hadoop jar /usr/joe/wordcount.jar org.myorg.WordCount
-Dwordcount.case.sensitive=true /usr/joe/wordcount/input
/usr/joe/wordcount/output -skip /user/joe/wordcount/patterns.txt
```

应该得到这样的输出：

```
$ bin/hadoop dfs -cat /usr/joe/wordcount/output/part-00000
Bye 1
Goodbye 1
Hadoop 1
Hello 2
World 2
hadoop 1
```

再运行一次，这一次关闭大小写敏感性(case sensitivity)：

```
$ bin/hadoop jar /usr/joe/wordcount.jar org.myorg.WordCount
-Dwordcount.case.sensitive=false /usr/joe/wordcount/input
/usr/joe/wordcount/output -skip /user/joe/wordcount/patterns.txt
```

输出：

```
$ bin/hadoop dfs -cat /usr/joe/wordcount/output/part-00000
bye 1
```

```
goodbye 1
hadoop 2
hello 2
world 2
```

# 2.6　Nutch 中的分布式

前面介绍了很多分布式的内容，具体在网络爬虫中如何应用这些知识呢？下面以一款开源的搜索引擎为例。介绍如何设计一个分布式爬虫。

和 Heritrix 一样，Nutch 是一个用 Java 语言开发的开源搜索引擎。现在，越来越多的人从 Heritrix 转向了 Nutch 的学习与开发。与 Heritrix 相比，Nutch 的爬虫具有以下特点：

- 简单。Nutch 的代码简单易懂、容易修改，核心的爬虫类不超过 10 个，而 Heritrix 比较复杂。读懂 Nutch 的代码轻而易举，而要想读懂 Heritrix 的代码则要费很大的功夫。
- Nutch 支持分布式爬虫。Nutch 最新版本的底层实现使用 Hadoop。在云计算风起云涌的今天，确实得到众多开发者的青睐。

## 2.6.1　Nutch 爬虫详解

Nutch 爬虫使用宽度优先搜索的技术进行抓取。Nutch 爬虫的设计着重两个方面：存储与爬虫过程。

首先，看一下 Nutch 的存储。Nutch 存储主要使用数据文件。它的数据文件有三类：分别是 Web database、segment 和 index。

Web database，也叫 WebDB，用于存储爬虫抓取的网页之间的链接结构信息，WebDB 只在爬虫中使用。WebDB 内存储了两种实体的信息：Page 和 Link。Page 实体描述互联网中网页的特征信息，主要包括网页内的链接数目，抓取此网页的时间等相关抓取信息，对此网页的重要度评分等。Link 实体描述的是两个 Page 实体之间的链接关系。WebDB 中存储了一个所抓取网页的链接结构图，在链接结构图中，Page 实体是图的节点，而 Link 实体则代表图的边。

一次爬行会产生很多个段(segment)，段存储的是爬虫在一次抓取过程中抓到的网页以及这些网页的索引。爬虫爬行时会根据 WebDB 中的链接关系按照一定的爬行策略生成每次抓取循环所需的预取列表(fetch list)，然后 Fetcher 类通过预取列表中的 URL 抓取这些网页并索引，然后将其存入段中。段是有时效的，网页被爬虫重新抓取后，先前抓取产生的段就作废了。存储时，段文件夹是以产生时间命名的，方便用户删除作废的 segments 以节省存储空间。

index 是爬虫抓取的所有网页的索引，它是将所有 segment 中的索引合并处理后得到的。Nutch 利用 Lucene 技术进行索引。但是需要注意的是，Lucene 中的段和 Nutch 中的段不同，Lucene 中的段是索引的一部分，但 Nutch 中的段和索引是各自独立的。

在分析了爬虫工作中设计的文件之后，接下来研究 Nutch 爬虫的抓取流程以及这些文

件在抓取中扮演的角色。Nutch 爬虫的工作原理主要是：首先根据 WebDB 生成一个待抓取网页的 URL 集合——预取列表，接着下载线程 Fetcher 类开始根据预取列表进行网页抓取。如果下载线程有很多个，那么就生成很多个预取列表，也就是一个 Fetcher 类的线程对应一个预取列表。爬虫根据抓取回来的网页更新 WebDB，根据更新后的 WebDB 生成新的预取列表。接着下一轮抓取循环重新开始。这个循环过程可以叫做“产生/抓取/更新”循环。

指向同一个主机上 Web 资源的 URL 通常被分配到同一个预取列表中，这样可以防止过多的 Fetcher 线程对一个主机同时进行抓取而导致主机负担过重。另外 Nutch 遵守 Robots 协议，网站可以通过自定义 Robots.txt 控制 Nutch 爬虫的抓取。

在 Nutch 中，抓取操作的实现是通过实现一系列子操作来完成的。Nutch 提供了子命令行可以单独调用这些子操作。下面就是这些子操作的功能描述以及对应的命令行，其中命令行写在括号中。

(1) 创建一个新的 WebDB (admin db -create)，并且将起始 URL 写入 WebDB (inject)。

(2) 根据 WebDB 生成预取列表并写入相应的 segment(generate)。

(3) 根据预取列表中的 URL 抓取网页(fetch)。

(4) 解析(parse)获得的网页。

(5) 根据网页内的 URL 更新 WebDB(updateDB)。

(6) 循环进行(2)～(5)步直至预先设定的抓取深度。

(7) 根据 WebDB 得到的网页评分和链接更新 segments (updatesegs)。

(8) 对所抓取的网页进行索引(index)。

(9) 在索引中丢弃有重复内容的网页和重复的 URL (dedup)。

(10) 将 segments 中的索引进行合并生成用于检索的最终 index(merge)。

Nutch 爬虫的详细工作流程是：在创建一个 WebDB 之后(步骤 1)， 根据一些种子 URL 开始启动“产生/抓取/更新”循环(步骤(2)～(6))。当这个循环彻底结束，爬虫根据抓取中生成的 segments 创建索引(步骤(7)～(10))。在重复清除 URL(步骤(9))之前，每个 segment 的索引都是独立的(步骤(8))。最终，各个独立的 segment 索引被合并为一个最终的索引 index(步骤(10))。整个 Nutch 的流程图如图 2.18 所示。

其中，标出来的(1)到(5)步是 Nutch 爬虫的过程。

下面，结合 Nutch 的源代码，看一下 Nutch 爬虫的实现。Nutch 的爬虫代码部分主要集中在 package org.apache.nutch.fetcher 和插件 protocol-file、Protocol-ftp、protocol-http、protocol-httpclient 以及相应的 Parser 插件中。最主要的类是 Fetcher 类，它控制了整个爬虫的工作流程，我们从它入手一步步跟踪整个代码。

Fetcher 类是一个线程类，它在 run()中采用多线程运行 FetcherThread 类，并调用恰当的 Protocol 插件(支持 http、ftp 等协议)获取内容，当内容被获取之后，调用恰当的 Parser 插件将内容分析为文本，然后把这些文本内容放到 FetcherOutput 类里，最后由 FetcherOutputFormat 类写到段中。下面从 run()函数开始分析，它的关键代码如下：

```
for (int i = 0; i < threadCount; i++) {
     FetcherThread thread = new FetcherThread(THREAD_GROUP_NAME+i);
     thread.start();
}
```

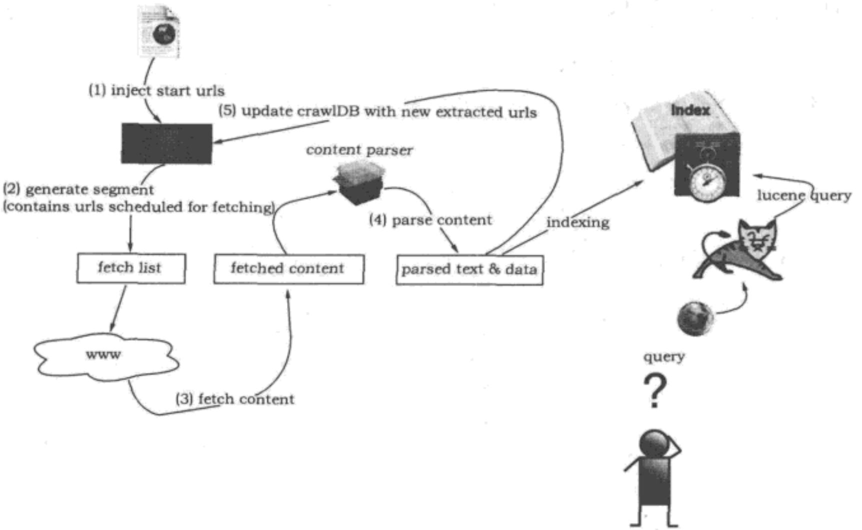



图 2.18 Nutch 工作流程

在上面的 run()函数的关键代码中，建立了多个 FetcherThread 线程来抓取网页，threadCount 可以配置或者使用默认值。在这段代码之后是一个 while(true)的循环：

```
int n = group.activeCount();
Thread[] list = new Thread[n];
group.enumerate(list);
boolean noMoreFetcherThread = true;
for (int i = 0; i < n; i++) {

    if (list[i] == null) continue;
    String tname = list[i].getName();
    if (tname.startsWith(THREAD_GROUP_NAME))
 noMoreFetcherThread = false;
    if (LOG.isLoggable(Level.FINE))
 LOG.fine(list[i].toString());
}
     if (noMoreFetcherThread) {
    if (LOG.isLoggable(Level.FINE))
LOG.fine("number of active threads: "+n);
    if (pages == pages0 && errors == errors0 && bytes == bytes0)      break;
    status();
    pages0 = pages;
    errors0 = errors;
    bytes0 = bytes;
}
```

上面这段代码相当于维护一个线程池，并在日志中输入抓取页面的速度、状态之类的

信息。下面看看抓取的线程 FetcherThread 是如何工作的：

```
FetchListEntry fle = new FetchListEntry();
```

首先，建立一个抓取列表类，然后又是一个 while (true)循环：

```
if (fetchList.next(fle) == null)
            break;
url = fle.getPage().getURL().toString();
```

从当前的 FetchListEntry 中获得一个要抓取的 URL，然后进行抓取：

```
if (!fle.getFetch()) {
            if (LOG.isLoggable(Level.FINE))
                 LOG.fine("not fetching " + url);
             handleFetch(fle, new ProtocolOutput(null,
             ProtocolStatus.STATUS_NOTFETCHING));
            continue;
}
if (fetchList.next(fle) == null)
          break;
url = fle.getPage().getURL().toString();
```

上面代码表明，如果不需要抓取，则在 handleFetch 类中进行相应的处理。接着又是一个 do…while 循环，用来处理抓取过程中重定向指定的次数：整个循环的条件是需要重新抓取并且重定向次数没有超出最大次数。

接下来，使用 ProtocolFactory 工厂类创建 Protocol 类实例：

```
Protocol protocol = ProtocolFactory.getProtocol(url);
```

Protocol 的实现是以插件的形式提供的，可以从 Protocol 中获取 Fetch 的输出流：

```
ProtocolOutput output = protocol.getProtocolOutput(fle);
```

通过输出流可以获取抓取的状态 ProtocolStatus 和抓取的内容 Content：

```
ProtocolStatus pstat = output.getStatus();
Content content = output.getContent();
ProtocolStatus pstat = output.getStatus();
Content content = output.getContent();
```

然后判断抓取的状态：

```
switch(pstat.getCode())
```

如果抓取成功，

```
case ProtocolStatus.SUCCESS:

if (content != null)
```

如果抓取到的内容不为空则修改抓取的页数，抓取的字节数，并且每抓取 100 页，就在日志中记录抓取的速度等信息。

```
synchronized (Fetcher.this) {
    pages++;
    bytes += content.getContent().length;
   if ((pages % 100) == 0) {
      status();
    }
}
```

在 handleFetch 中进行相应的处理：

```
ParseStatus ps = handleFetch(fle, output);
```

如果处理返回的状态不为空，并且成功地重定向：

```
if (ps != null && ps.getMinorCode() == ParseStatus.SUCCESS_REDIRECT)
```

获取重定向的链接并进行过滤：

```
String newurl = ps.getMessage();
newurl = URLFilters.filter(newurl);
```

如果重定向的链接 newurl 不为空并且和现在的 URL 不同：

```
if (newurl != null && !newurl.equals(url))

refetch = true;
url = newurl;
redirCnt++;
```

创建当前页面的 FetchListEntry：

```
fle = new FetchListEntry(true, new Page(url, NEW_INJECTED_PAGE_SCORE), new
String[0]);
```

如果链接页面已经转移或者临时转移：

```
case ProtocolStatus.MOVE D:
case ProtocolStatus.TEMP_MOVED:
```

立即重定向，处理抓取的结果：

```
handleFetch(fle, output);
```

获取重定向的 URL：

```
String newurl = pstat.getMessage();
      newurl = URLFilters.filter(newurl);
     if (newurl != null && !newurl.equals(url)) {
              refetch = true;
              url = newurl;
              redirCnt++;
              fle = new FetchListEntry(true, new Page(url,
              NEW_INJECTED_PAGE_SCORE), new String[0]);
}
```

整个获取重定向的URL过程和上面的重定向类似。

如果获得的状态是以下几种状态之一，直接交由handleFetch类来处理。

```
case ProtocolStatus.GONE:
case ProtocolStatus.NOTFOUND:
case ProtocolStatus.ACCESS_DENIED:
case ProtocolStatus.ROBOTS_DENIED:
case ProtocolStatus.RETRY:
case ProtocolStatus.NOTMODIFIED:
case ProtocolStatus.GONE:
case ProtocolStatus.NOTFOUND:
case ProtocolStatus.ACCESS_DENIED:
case ProtocolStatus.ROBOTS_DENIED:
case ProtocolStatus.RETRY:
case ProtocolStatus.NOTMODIFIED:
```

如果发生异常，则在日志中记录异常信息，然后交给handleFetch类处理：

```
case ProtocolStatus.EXCEPTION:
logError(url, fle, new Exception(pstat.getMessage()));
handleFetch(fle, output);
```

其他情况为未知状态，在日志中记录当前的状态，然后交给handleFetch处理：

```
default:
LOG.warning("Unknown ProtocolStatus: " + pstat.getCode());
handleFetch(fle, output);
```

循环结束。

最后如果完成的线程数等于threadCount，则关闭所有的插件：

```
synchronized (Fetcher.this) {
          atCompletion++;
         if (atCompletion == threadCount) {
           try {
              PluginRepository.getInstance().finalize();
            } catch (java.lang.Throwable t) {
             // do nothing
            }
          }
        }
```

从上面的代码分析中可以看到，获取页面后大多数的处理都交给了handleFetch类。下面，就来看看private ParseStatus handleFetch(FetchListEntry fle, ProtocolOutput output)的代码：

根据output获取到内容和URL：

```
Content content = output.getContent();
MD5Hash hash = null;
```

```
String url = fle.getPage().getURL().toString();
```

如果 content 为 null，新建 content：

```
if (content == null) {
        content = new Content(url, url, new byte[0], "", new
Properties());
        hash = MD5Hash.digest(url);
} else {
        hash = MD5Hash.digest(content.getContent());
}
```

在获取 ProtocolStatus 时：

```
ProtocolStatus protocolStatus = output.getStatus();
```

如果 Fetcher 不进行解析(parse)，直接把抓取的页面写入磁盘。

```
if (!Fetcher.this.parsing) {
        outputPage(new FetcherOutput(fle, hash, protocolStatus),
                content, null, null);
       return null;
      }
```

如果 Fetcher 需要进行解析，则首先获取页面 contentType，以便根据正确编码进行解析：

```
String contentType = content.getContentType();
```

下面是使用 Parser 类进行页面解析的过程：

```
Parse parse = null;
ParseStatus status = null;
try {
  parser = ParserFactory.getParser(contentType, url);
  parse = parser.getParse(content);
  status = parse.getData().getStatus();
} catch (Exception e) {
  e.printStackTrace();
  status = new ParseStatus(e);
}
```

如果提取页面成功：

```
if (status.isSuccess())
```

将 FetcherOutput 提取的内容以及状态保存：

```
outputPage(new FetcherOutput(fle, hash, protocolStatus),
      content, new ParseText(parse.getText()), parse.getData());
```

否则将 FetcherOutput 和空的内容保存：

```
LOG.info("fetch okay, but can't parse " + url + ", reason: "
          + status.toString());
```

```
outputPage(new FetcherOutput(fle, hash, protocolStatus),
           content, new ParseText(""),
           new ParseData(status, "", new Outlink[0], new Properties()));
```

在抓取过程中使用的 Protocol 采用了 Nutch 的插件机制，任何实现 Protocol 接口的实现类都可以负责抓取数据。这使得 Nutch 可以使用更多的网络协议获得数据。例如：Http、FTP 等。有关插件机制的代码，这里不再一一讲述，有兴趣的读者可以参考相关资料。

### 2.6.2 Nutch 中的分布式

本节将讲述 Nutch 中的分布式机制。首先，Nutch 对于抓取的数据和索引是采用分布式存储的；其次，在爬虫和建立索引的许多环节，Nutch 都采用了 Map/Reduce 算法进行分布式计算。本节将从这两个方面对 Nutch 的分布式进行讲解。

#### 1. Nutch 的分布式文件系统

Nutch 抓取内容之后就要开始对文件进行管理。Nutch 分布式文件系统的基础架构是 Hadoop 文件系统 NDFS。

Nutch 的整个分布式文件系统工作架构如图 2.19 所示。

#### 2. Nutch 中的 MapReduce 算法

在介绍 Nutch 中的 MapReduce 算法应用之前，先介绍 Nutch 中的 CrawlDB 目录。

Nutch 中的 CrawlDB 目录中存储着一系列的文件，这些文件中每一行都存储 <URL,CrawlDatum>数据结构。其中，CrawlDatum 数据结构保存了对应 URL 的一系列属性。包括抓取时间、状态、抓取时间间隔、链接数目等。

上一节讲述的 Nutch 运行过程的 10 个步骤中有 6 个步骤，包括插入 URL 列表(inject)，生成抓取列表(generate)，抓取内容(fetch)，分析处理内容(parse)，更新 Crawl DB 库(updatedb)和建立索引(index)都是采用 MapReduce 算法来完成的。具体技术实现细节如下：

1) 插入 URL 列表(inject)

MapReduce 程序 1：

目标：转换 input 输入为 CrawlDatum 格式

输入：URL 文件

步骤：

(1) map(line) → <url, CrawlDatum>

(2) reduce()合并多重的 URL

输出：临时的 CrawlDatum 文件

MapReduce 程序 2：

目标：合并上一步产生的临时文件到新的 CrawlDB

输入：上次 MapReduce 输出的 CrawlDatum

步骤：

(1) map()过滤重复的 URL

(2) reduce：合并两个 CrawlDatum 到一个新的 CrawlDB

输出：CrawlDatum

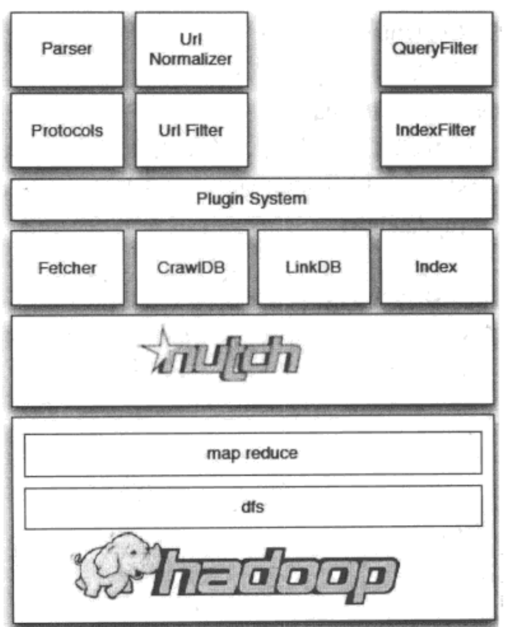



图 2.19　Nutch 分布式文件系统架构

2) 生成抓取列表(Generate)

MapReduce 程序：

目标：选择抓取列表

输入：CrawlDB 文件

步骤：

(1) map() → 如果抓取当前时间大于现在时间，转换成 <CrawlDatum,URL>格式

(2) reduce：取最顶部的 N 个链接

输出：<URL,CrawlDatum>文件

3) 抓取内容(Fetch)

MapReduce 程序：

目标：抓取内容

输入：<URL,CrawlDatum>，按主机划分，按 hash 排序

步骤：

(1) map(URL,CrawlDatum) → 输出<URL,FetcherOutput>

(2) 多线程，调用 Nutch 的抓取协议插件，抓取输出<CrawlDatum, Content>

输出：<URL,CrawlDatum>和<URL,Content>两个文件

4) 分析处理内容(Parse)

MapReduce 程序：

目标：处理抓取的内容

输入：抓取的< URL, Content>

步骤：

(1) map(URL,Content) → <URL,Parse>

(2) raduce()函数调用 Nutch 的解析插件，输出处理完的格式是<ParseText,ParseData>

输出：< URL,ParseText>、<URL,ParseData>、<URL,CrawlDatum>

5) 更新 CrawlDB 库(updateDB)

MapReduce 程序：

目标：将 fetch 和 parse 整合到 DB 中

输入：<URL, CrawlDatum> 现有的 DB 加上 fetch 和 parse 的输出，合并上面 3 个 DB 为一个新的 DB

输出：新的抓取 DB

6) 建立索引(index)

MapReduce 程序：

目标：生成 Lucene 索引

输入：多种文件格式

步骤：

(1) parse 处理完的<URL,ParseData> 提取 title、metadata 信息等

(2) parse 处理完的<URL,ParseText> 提取 text 内容

(3) 转换链接处理完的<URL,Inlinks> 提取 anchors

(4) 抓取内容处理完的<URL,CrawlDatum> 提取抓取时间

(5) map()函数用 ObjectWritable 包裹上面的内容

(6) reduce()函数调用 Nutch 的索引插件，生成 Lucene Document 文档

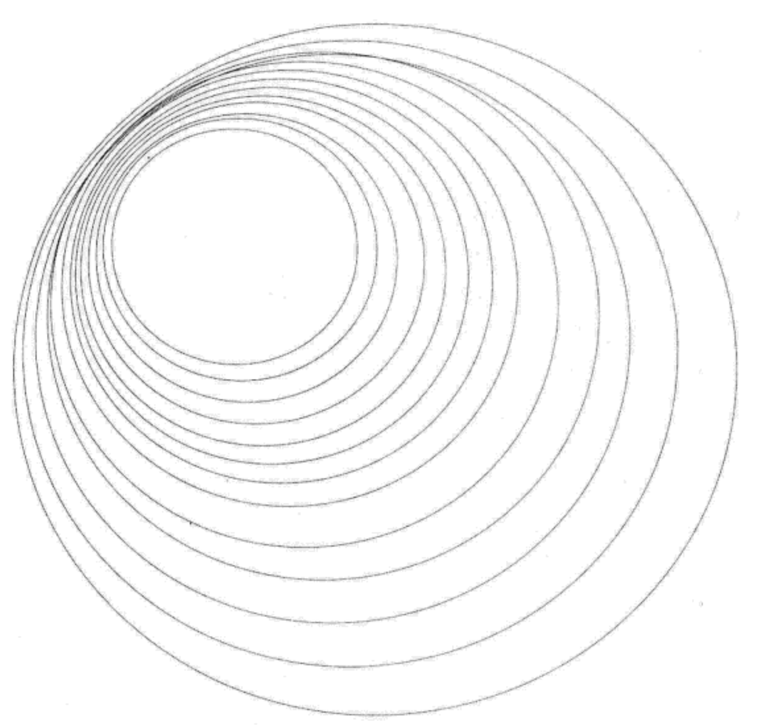
输出：输出 Lucene 索引

## 2.7 本章小结

本章介绍了分布式爬虫的基本原理以及开源分布式爬虫 Nutch 的实现。主要内容包括当前比较流行的开源分布式架构：Map/Reduce+BigTable+GFS 以及负载平衡的 ConsistentsHash 算法。最后，分析了开源分布式爬虫架构 Nutch 的实现。下面是一些相关的资料。

- http://labs.google.com/zh-CN//papers/mapreduce-osdi04.pdf 是 Google 介绍 MapReduce 的权威论文。
- http://labs.google.com/papers/bigtable.html 是 Google 介绍 BigTable 的权威论文。

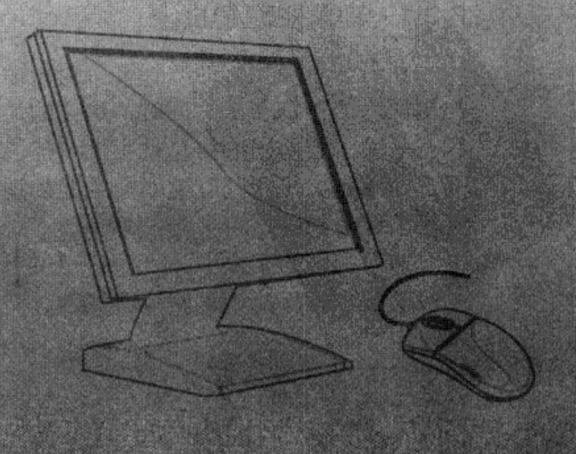
- http://labs.google.com/papers/gfs.html 是 Google 介绍 GFS 的权威论文。
- http://en.wikipedia.org/wiki/Consistent_hashing 网站详细讲述了 Consistent Hash 的相关内容。
- http://lucene.apache.org/nutch/是 Apache 下的 Nutch 项目的站点，网上有关 Nutch 的资料不是很多，这个站点是不错的一个。



# 第3章

## 爬虫的“方方面面”

在网络爬虫中，有一些“特殊情况”需要处理。比如，爬虫可能会在运行过程中掉进“陷阱”，或者有些网站不希望爬虫去获取某些内容，这时候，一个良好的爬虫应该遵守爬虫“道德”。而且有些时候我们只需要某一主题的爬虫，因此，要对抓取的内容进行过滤。还有的时候，我们需要对爬取的站点做一定的限制，比如限制IP等。本章将详细介绍如何解决这些问题。



# 3.1 爬虫中的“黑洞”

在网络爬虫中，有一种被称为“黑洞”的情况，即在抓取一张网页的链接时，链接本身是一个无限循环。因此，爬虫抓取的时候也会循环，以至资源白白被浪费掉而没有任何效果。并且有些 URL 看起来不同，但实际指向同一张网页，也会使爬虫陷入重复抓取的境地。

设想如下的例子，一个搜索引擎爬虫来到一个网站，被指定了一个 Session ID，然后这个 ID 被嵌入以后在该网站中被爬行的所有页面。另一个搜索引擎爬虫又来到这个页面时，会得到一个全新的 Session ID，因为网站服务器不能探测到这是同一个搜索引擎爬虫，而且它在几分钟以前来过。这就导致了同一个页面被索引了很多次，这无论对于搜索引擎自身，或是对于搜索引擎的用户来说，这都是很坏的结果，因为这些都是重复的。

那如何在爬虫的设计中防止“黑洞”的发生呢？

设计爬虫时，通常都回避动态网页。因为动态网页常常会把爬虫带入蜘蛛陷阱(也就是我们所说的黑洞)。识别动态网页时，只需要看 URL 中是否出现问号，含问号的就是动态网页。在线日历就是一个容易被忽视的蜘蛛陷阱，它生成的动态网页中可以标上任何日期，并包含指向后一天网页的链接。一个爬虫从这个日历中找到一个网页后，便会无止境地请求后一天的网页。

商业搜索引擎的爬虫通常回避这些带问号的URL，因为这些URL可能会导致蜘蛛陷阱。Google 已经放宽了对这些站点的限制。Google 爬虫似乎有另一种避免蜘蛛陷阱的方法。

之前介绍的 Visited 表其实也起到了一种防止爬虫“黑洞”的作用。因为如果没有 Visited 表，遇到网络中许多链接的环路，例如：a->b->c->d->b->e 这样的环路链接，爬虫就会掉进去，反复抓取 b、c、d 这几个 URL 针对的页面。而有了 Visited 表，就能避免这个问题。

在过去的几年中，爬虫的“黑洞”问题曾经一度困扰搜索引擎开发者。但是最近几年，它已经显得不那么重要了。因为基本所有的网站都在 SEO(搜索引擎优化)。本书只是让大家了解“黑洞”的概念以及基本的解决方法。因此不再举例说明。

# 3.2 限定爬虫和主题爬虫

## 3.2.1 理解主题爬虫

随着信息多元化的增长，千篇一律地给所有用户同一个入口显然已经不能满足特定用户更深入的查询需求。同时，这样的通用搜索引擎在目前的硬件条件下，要及时更新以得到互联网上较全面的信息是不太可能的。针对这种情况，我们需要一个分类细致精确、数据全面深入、更新及时的面向主题的搜索引擎。像新闻搜索、生活搜索等主题搜索引擎已经越来越受到用户的欢迎。

面向主题的爬虫称为“主题爬虫”。主题爬虫和一般的爬虫有何区别呢？简单地说，一般的爬虫程序是“见网页就抓”。而主题爬虫只抓取和主题“相关”的页面。比如，新

闻搜索引擎就只抓取新闻。

如何控制抓取的网页是和主题相关的呢？主要的解决思路有四种，第一种最简单，通常指一些行业搜索。比如说机票搜索等，抓取的是各大航空公司网站和代理人网站上面的数据。由于航空公司的数量有限(只有 20 多家)，代理人网站的数量也比较有限，因此，抓取的时候可以根据这些网站做定制抓取。也就是根据每个网站的页面内容，分析它们的源代码的结构，获得自己需要的信息，存入数据库中。这种方法适合小型的行业搜索引擎。

下面是定制抓取机票价格的例子：

```
public class RetrivePage {
    private static HttpClient httpClient = new HttpClient();
    // 设置代理服务器
    static {
        // 设置代理服务器的 IP 地址和端口
        // httpClient.getHostConfiguration().setProxy("172.17.18.84", 8080);
    }

    public static boolean downloadPage(String path) throws HttpException,
            IOException, Exception {
        InputStream input = null;
        OutputStream output = null;
        // 得到 get 方法
        GetMethod getMethod = new GetMethod(path);
        // 设置 get 方法的参数

        NameValuePair[] postData = new NameValuePair[8];
        postData[0] = new NameValuePair("DCity1", "BJS");
        postData[1] = new NameValuePair("ACity1", "SHA");
        postData[2] = new NameValuePair("DDate1", "2010-1-31");
        postData[3] = new NameValuePair("ClassType", "");
        postData[4] = new NameValuePair("PassengerQuantity", "1");
        postData[5] = new NameValuePair("SendTicketCity", "%u5317%u4EAC");
        postData[6] = new NameValuePair("Airline", "");
        postData[7] = new NameValuePair("PassengerType", "ADU");
        getMethod.setQueryString(postData);
        // 执行，返回状态码
        int statusCode = httpClient.executeMethod(getMethod);
        // 针对状态码进行处理 (简单起见，只处理返回值为 200 的状态码)
        if (statusCode == HttpStatus.SC_OK) {
            // 获得返回的内容
            input = getMethod.getResponseBodyAsStream();
            String charset = getMethod.getResponseCharSet();
            // 得到文件名
            String filename = path.substring(path.lastIndexOf('/') + 1);
            filename += System.currentTimeMillis();
            // 获得文件输出流
            File tempFile = new File(filename);
            if (!tempFile.exists()) {
                tempFile.createNewFile();
```

```
}
output = new FileOutputStream(tempFile);
// 输出到文件
int tempByte = -1;
while ((tempByte = input.read()) > 0) {
    output.write(tempByte);
}
// 关闭输入输出流
if (input != null) {
    input.close();
}
if (output != null) {
    output.close();
}
// 使用 htmlpaser 解析
Parser parser = new Parser(filename);
parser.setEncoding(charset);
NodeVisitor nodeVisitor = new NodeVisitor(){
    private boolean flag =false;
    private int index = -1;
    public void visitTag (Tag tag){
        if(tag.getTagName().equals("TBODY")){
            System.out.println("begin........");
            flag = true;
            index = 0;
        }
        if(tag.getTagName().equals("TD")&&flag){

            switch(index){
            case 0:
                System.out.println("from_to:"+tag.
                toPlainTextString().trim());
                break;
            case 1:
                System.out.println("carrier:"+tag.
                toPlainTextString().trim());
            break;
            case 2:
                System.out.println("type:"+tag.
                toPlainTextString().trim());
            break;
            case 3:
                System.out.println("number:"+tag.
                toPlainTextString().trim());
                break;
            case 4:
                System.out.println("dicount:"+tag.
                toPlainTextString().trim());
            break;
```

```
                    case 5:
                        System.out.println("price:"+tag.
                        toPlainTextString().trim());
                    break;
                    }
                    index++;
                    }
                }

                public void visitEndTag (Tag tag){
                    if(tag.getTagName().equals("TBODY")){

                        System.out.println("end........");
                        flag = false;
                    }
                }
            };
            parser.visitAllNodesWith(nodeVisitor);
            return true;
        }
        return false;
    }

    /**
     * 测试代码
     */
    public static void main(String[] args) {
        // 抓取 lietu 首页，输出
        try {
            RetrivePage
                    .downloadPage("http://flights.ctrip.com/
                    Domestic/ShowFareFirst.aspx");
        } catch (HttpException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

第二种思路是根据得到的网页的内容，判断网页的内容和主题是否相关。如果一个网页是和主题相关的，在网页中的标题、正文、超链接中，通常会有一些与主题相关的关键词。在面向主题的搜索中，这种词叫做导向词，给每个导向词一个权重，就能够优先访问和主题相关的 URL。

导向词设置权重的方法有两种，第一种是根据管理员的经验手工设置；第二种是给定一个跟主题相关的网页集合，由程序自动提取这些网页中共同的特征，在这些网页中都出

现的关键词，就被选作导向词，我们把第二种方法称为“特征提取”。手工设置的好处是实现简单，同时人的经验一般比较准确，跟实际情况不会出现太大的偏差，缺点是导向词可能有缺漏，权值的量化定义不够精确；特征提取的优点是权值量化定义精确，但是它要求用来提取特征的网页集合必须是非常有代表性而且是全面概括的，否则导向词就可能出现很大的偏差。系统可以结合使用这两种方法：

(1) 手工设置好一组导向词和它们对应的权值。

(2) 用这组导向词到原搜索引擎中查找出对应的网页。

(3) 按权值的比例选取一定数量的网页(比如若权值是 10，则可以选 10n 个网页)。

(4) 用这些网页组成的集合作为特征提取网页内容，得到一组新的导向词。

以下是以“电影”为主题的一个例子的导向词及权值的配置：

| 导向词 | 权值 |
| --- | --- |
| 电影 | 10 |
| 影视 | 10 |
| 导演 | 10 |
| 编剧 | 10 |
| 监制 | 10 |
| 主角 | 8 |
| 配角 | 8 |
| 奥斯卡 | 5 |
| 影星 | 9 |

下面是 URL 优先级的一个计算公式，有了这个公式，我们就可以把 URL 赋予一定的权重，然后根据权重的大小来确定先扩展哪个 URL。

$$Pirority(URL) = a1 \times domain_weight + a2 \times link_popularity + a3 \times priority(parent_url) + a4 \times directory_depth$$

其中，domain_weight 表示 URL 中各个域的权值。link_popularity 表示到目前为止，这个超链接被其他网页引用的次数。parent_url 表示它的上一级双亲 URL 的权值。directory_depth 表示这个 URL 中目录的深度，越深的目录，这个 URL 的权值越低。ai 表示每部分所占的比例。

现在，有了导向词这个设置，公式也应该随之修改：

$$Pirority(URL) = a1 \times domain_weight + a2 \times link_popularity + a3 \times priority(parent_url) + a4 \times directory_depth + a5 \times \sum Oi$$

其中，Oi 表示导向词的权重乘以导向词出现的次数；a5 表示导向词在整个评分系统中所占的比例。

导向词算法在对一个 URL 进行评分的时候，总是要先下载 URL 指向的页面，然后对 URL 进行评分，放入 TODO 表中进行处理。当扩展这个 URL 的时候，还要进行一次下载，这样对网络资源的消耗非常大。这时可以采用一种阈值的办法进行处理。即通过经验设定一个阈值，如果得出的评分大于这个阈值，就立刻进行下一步处理，并且先不对从当前页

面中提取出的 URL 进行评分计算，待到扩展的时候再进行评分，否则，简单丢弃这个页面即可。

本章我们暂时先不给出评分程序，而是介绍一种思路，我们将在第 8 章讲述 PageRank 算法时，给出网页评分的示例代码。

第三种方法是针对网页链接进行评分(这种方法只是根据之前爬虫爬取的信息对当前的 URL 进行评分，而不涉及当前网页的内容)。在爬虫过程中，针对每一个 URL，都有一个指向该网页的其他网页的总数，称为 HitNumber。爬虫每访问一个新的网页，都会逐一检查这个网页的所有超链接，如果发现这些超链接里面有指向已经访问过的网页，那么这个已经访问过的网页的 HitNumber 将会被相应地加 1。等到爬虫已经访问过的网页集合足够大的时候(理想情况是整个网络)，HitNumber 越大，表示这个网页被别人引用得越多，由此可以估计这个网页也越重要。这种网页不论是在抓取网页方面，还是在检索器最终给用户返回结构方面，都应该放在优先处理的位置。

但是，单纯比较两个网页的 HitNumber，有时候可能无法估计出这两个网页正确的重要性排序。比如有两个网页，它们在 Internet 上同样只是被引用了一次。一个是一份已经过时的个人简历，被一个个人网站所引用，除了作者以外基本没有人关心；另外一个是当天发生的重大国际新闻，被 Yahoo！所引用，每一秒钟都被世界各地不同肤色的数以百计的人浏览。这时你不能由两个网页的 HitNumber 一样(都是 1)就得出两个网页在 Internet 上一样重要的结论。这时，我们需要一个更加深入的指标来评测，这就是下面提到的网页评分(PageRank)(在本章，我们只对 PageRank 的内容做简单的介绍，而不涉及过多深入的讲解和代码实现，对 PageRank 的原理和代码实现，我们将在第 8 章详细讲述)。

在考虑一个网页被另一个网页引用的时候，不是单纯地将被引用网页的 HitNumber 加 1，而是将引用网页的链接数作为权，同时将该引用网页的重要性也考虑进来(看看上面提到的例子，Yahoo！引用的网页显然比个人网站引用的网页重要，因为 Yahoo！本身很重要)，就可以得到扩展后的网页评分。最早提出网页评分的计算方法是 Google。它们提出了一个“随机冲浪”模型来描述网络用户对网页的访问行为。模型假设如下：

(1)　用户随机选择一个网页作为上网的起始网页。

(2)　看完这个网页后，从该网页内所含的超链接内随机地选择一个页面继续进行浏览。

(3)　沿着超链接前进一定数目的网页后，用户对这个主题感到厌倦，重新随机选择一个网页进行浏览，并重复第(2)步和第(3)步。按照以上的用户行为模型，每个网页可能被访问到的次数就是该网页的链接权值。PageRank 采用以下公式计算链接权值：

$$W_j = (1-d) + d\sum_{i=1,i\neq j}^{N} l_{i,j}\frac{W_i}{n_i}$$

其中，$W_j$代表第 $j$ 个网页的权值；$l_{i,j}$，只取 0、1 值，代表从网页 $i$ 到网页 $j$ 是否存在链接；$n_i$代表网页 $i$ 有多少个与其他网页的链接；$d$ 代表“随机冲浪”中沿着链接访问网页的平均次数。选择合适的数值，递归使用以上公式，即可得到理想的网页链接权值。

该方法能够大幅度地提高简单检索返回结果的质量，同时能够有效地防止网页编写者对搜索引擎的欺骗。因此可以将其广泛地应用在检索器提供给用户的网页排序上，对于网页评分越高的网页，就排得越往前。

最后一种策略称为链接描述文本分析，互联网的魅力在于“互联”，从互联网上任何一个地方出发，都可以轻而易举地到达世界上任何地方。互联网是通过一系列的“超级文本链接”(Hypertext Link)实现互联的，可以这样说，没有“超级文本链接”，互联网会变得不名一文。每一个超级链接都有一个描述文本(Anchor Text)，这个文本反映了该网页与该链接所至网页的某种关系，是互联的关键所在。通过分析这个描述文本，就可以得到网页之间重要的关系。

当爬虫在处理当前网页的时候，会遇到许多描述文本。由于描述文本通常与所指向的网页相关联。因此，处理描述文本需要频繁切换当前处理页面，从而影响到爬虫速度。然而，这样做也有自身好处。首先，描述文本通常比网页上的文本要更加精确地概括这个网页；其次，基于文本切词的搜索引擎并不能处理网络上的所有格式，比如一些图像、程序、数据库和多媒体等格式的文件，但是依靠超链接里面的描述文本与其关联，即使搜索器本身并没有真的抓取这些格式的文件，也可以索引这些搜索引擎无法直接索引的文件。但是这时候需要注意一个问题，就是通常说的“死链”问题。因为爬虫本身并没有真地抓取这些格式的文件，而只是记录了超链接描述文本，所以这些链接有可能只有描述文本，而它指向的文件根本不存在，这就是“死链”。

### 3.2.2 Java 主题爬虫

这一节介绍 Java 主题爬虫的具体实现，如图 3.1 所示为主题爬虫 URL 处理流程。

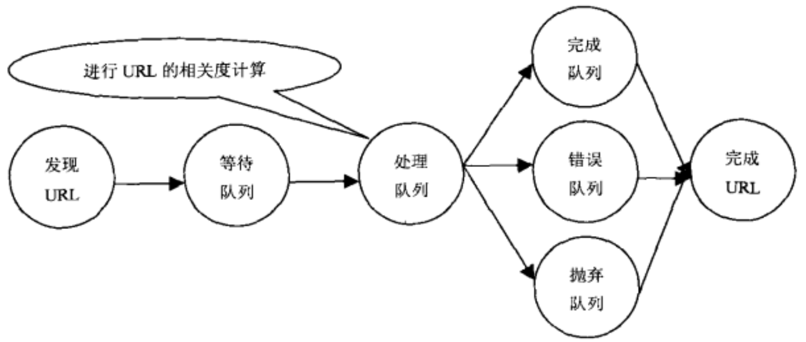



**图 3.1 主题爬虫 URL 处理流程**

由此可见，把一般爬虫修改为主题爬虫相对来说还是比较简单的，就是要在处理的时候(也就是下载网页内容之后)计算出网页的相关度，如果是主题相关的，就做进一步处理。否则，就简单地丢弃网页。

下面针对第 1 章写的爬虫做修改，可以实现主题爬虫：

```
public interface ComputeUrl {
    public boolean accept(String url,String pageContent);
}
public class PageRankComputeUrl implements ComputeUrl {
    //针对 URL 进行链接分析，这里暂时不实现，等到第 8 章再实现 PageRank 算法
    public boolean accept(String url,String pageContent) {
```

```
        return false;
    }

}
public class MyCrawler {
    private ComputeUrl computeUrl = null;
    public MyCrawler(){
        computeUrl = new  PageRankComputeUrl();
    }
    /**
     * 使用种子初始化 URL 队列
     * @return
     * @param seeds 种子 URL
     */
    private void initCrawlerWithSeeds(String[] seeds)
    {
        for(int i=0;i<seeds.length;i++)
            LinkQueue.addUnvisitedUrl(seeds[i]);
    }
    /**
     * 抓取过程
     * @return
     * @param seeds
     */
    public void crawling(String[] seeds)
    {   //定义过滤器，提取以 http://www.lietu.com 开头的链接
        LinkFilter filter = new LinkFilter(){
            public boolean accept(String url) {
                if(url.startsWith("http://www.lietu.com"))
                    return true;
                else
                    return false;
            }
        };
        //初始化 URL 队列
        initCrawlerWithSeeds(seeds);
        //循环条件：待抓取的链接不空且抓取的网页不多于 1000

    while(!LinkQueue.unVisitedUrlsEmpty()&&LinkQueue.getVisitedUrlNum()<
=1000)
        {
            //队头 URL 出队列
            String visitUrl=(String)LinkQueue.unVisitedUrlDeQueue();
            //如果 URL 为空或者没有通过链接分析的计算
            if(visitUrl==null)
                continue;
            DownLoadFile downLoader=new DownLoadFile();
```

使用 PageRank 算法进行分析，也可以换成其他算法，只要实现 ComputeUrl 接口

```
            //下载网页
            String content  =  downLoader.downloadFile(visitUrl);
            if(computeUrl.accept(visitUrl,content)){
               continue;
            }
            //该 URL 放入到已访问的 URL 中
            LinkQueue.addVisitedUrl(visitUrl);
            //提取出下载网页中的 URL

            Set<String> links=HtmlParserTool.extracLinks(visitUrl,filter);
            //新的未访问的 URL 入队
            for(String link:links)
            {
                LinkQueue.addUnvisitedUrl(link);
            }
        }
    }
    //main 方法入口
    public static void main(String[]args)
    {
        MyCrawler crawler = new MyCrawler();
        crawler.crawling(new String[]{"http://www.twt.edu.cn"});
    }
}
```



有关 PageRankComputeUrl 的实现，我们留待第 8 章详细讲解 PageRank 算法的时候再介绍。

### 3.2.3 理解限定爬虫

在日常的爬虫过程中，除了主题爬虫之外，有时候还会涉及“限定”爬虫。那么，什么是限定爬虫呢？

限定爬虫就是对爬虫所爬取的主机的范围做一些限制，通常，限定爬虫包含以下几个方面。

(1) 限定域名的爬虫。比如，只抓取 edu.cn 结尾的域名。

(2) 限定爬取层数的爬虫。比如，限定只抓取 2 层的数据。

(3) 限定 IP 的抓取。比如，只抓取中国范围内的 IP。

(4) 限定语言的抓取。比如，只抓取中文的页面。

对于限定域名的抓取，是一种最简单的限定抓取。只要判断当前 URL 字符串的值，使得主机部分以 edu.cn 结尾即可。

限定爬虫爬取的层次，比限定域名复杂。之前章节对于 URL，都以一个字符串表示。现在，重新对 URL 进行建模，建立一个完整的 URL 模型，并且能够应用在第 8 章的链接分析等章节。下面是我们对 URL 进行重新建模的代码：

```
public class Url {
```

```
    // 原始URL的值，主机部分是域名
    private String oriUrl;
    // URL的值，主机部分是IP，为了防止重复主机的出现
    private String url;
    //URL NUM
    private int urlNo;
    // 获取URL返回的结果码
    private int statusCode;
    // 此URL被其他文章引用的次数
    private int hitNum;
    // 此URL对应文章的汉字编码
    private String charSet;
    // 文章摘要
    private String abstractText;
    // 作者
    private String author;
    // 文章的权重(包含导向词的信息)
    private int weight;
    // 文章的描述
    private String description;
    // 文章大小
    private int fileSize;
    // 最后修改时间
    private Timestamp lastUpdateTime;
    // 过期时间
    private Date timeToLive;
    // 文章名称
    private String title;
    // 文章类型
    private String type;
    // 引用的链接
    private Url[] urlRefrences;
    //爬取的层次，从种子开始，依次为第0层，第1层……
    private int layer;
    //get set方法等
    …
}
```

爬取的层次为父URL加1

爬虫开始的时候，种子节点层次设定为 0。当抓取种子节点对应的页面时，把其中的超链接放入 urlRefrences 中，并且把这些超链接的 layer 层次加 1，然后再放入 TODO 表中。之后再抓取的时候，首先应该取得 layer 数据，判断是否小于限制的层次，如果小于的话，就继续爬取，否则丢弃。

限定 IP 的抓取是限定抓取中最难的部分，所谓限定 IP，通常分为限定特定 IP 和限定某一地区的 IP。限定特定 IP 抓取通常较容易实现，通过 URL 主机部分字符串，可以获得主机 IP 地址，如果主机 IP 地址在被限制的列表中，则不进行抓取，否则，进行正常的抓取工作。

那如何根据主机字符串获得主机 IP 地址呢？下面的程序展示了这个过程：

```
public class IP {
    public static void main(String[] args) throws IOException {
        String hostname;
        BufferedReader input = new BufferedReader(new InputStreamReader(
                System.in));
        System.out.print("\n");
        System.out.print("Host name: ");
        hostname = input.readLine();
        try {
            InetAddress ipaddress = InetAddress.getByName(hostname);
            System.out.println("IP address: " + ipaddress.getHostAddress());
        } catch (UnknownHostException e) {
            System.out.println("Could not find IP address for: " + hostname);
        }
    }
}
```

执行程序，首先要求输入主机名：

```
Host name:
```

输入“www.163.com”后，按 Enter 键，可以得到网易的主机 IP：

```
Host name: www.163.com
IP address: 61.135.253.10
```

如果要实现对限定地区的 IP 的抓取，则需要一个 IP 与地区对应的数据库。网络上这种数据库大多都是付费的，腾讯有一款自己做的数据库“纯真 IP 数据库”，可以提供地址和 IP 的对应关系。

“纯真 IP 数据库”是一个名为 QQWry.dat 的文件，它有自己的文件结构。通过了解它的文件结构，可以实现 IP 地址和区域的映射程序。

QQWry.dat 文件在结构上分为三块：文件头、记录区和索引区。一般我们要查找 IP 时，先在索引区查找记录偏移，之后到记录区读出信息。由于记录区的记录是不定长的，所以直接在记录区中搜索是不可能的。由于记录数比较多，全部遍历索引区也会比较慢，一般来说，我们可以用二分查找法搜索索引区，其速度比遍历索引区快若干数量级。图 3.2 所示是纯真数据库的文件结构。

要注意的是，QQWry.dat 文件全部采用了 Little-endian 字节序。

QQWry.dat 的文件头只有 8 个字节，其结构非常简单，前四个字节是第一条索引的绝对偏移，后四个字节是最后一条索引的绝对偏移。每条 IP 记录都由国家和地区名组成，国家地区在这里并不是太确切，因为可能会查出 “清华大学计算机系”之类的内容，这里清华大学就成了国家名了，因此记录的格式有点像 QName，即由全局部分和局部部分组成，但本书还是沿用国家名和地区名的说法。

最简单情况下，一条记录的格式应该是 [IP 地址][国家名][地区名]。国家名和地区名可能会有很多的重复，如果每条记录都保存一个完整的名称拷贝是非常不理想的，因此需要

重定向以节省空间。为了得到一个国家名或者地区名，就有了两个可能：第一就是直接用字符串表示国家名；第二就是采用一个 4 字节的结构，第一个字节表明了重定向的模式，后面 3 个字节是国家名或者地区名的实际偏移位置。对于国家名来说，情况还可能更复杂些，因为这样的重定向最多可能有两次。

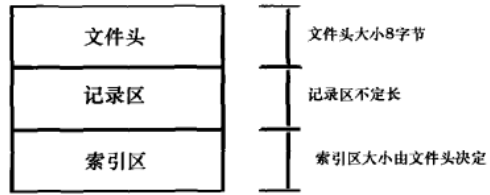



图 3.2　纯真 IP 数据库文件结构

那么什么是重定向模式？根据上面所述，一条记录的格式是[IP 地址][国家记录][地区记录]，如果国家记录是重定向的话，那么地区记录有可能没有，于是就有了两种情况，我管它叫做模式 1 和模式 2。我们用图来说明这些格式的情况。

图 3.3 表示了最简单的 IP 记录格式。

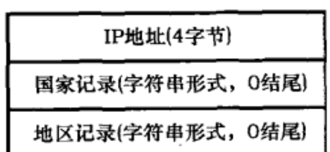



图 3.3　IP 记录的最简单形式

图 3.4 演示了重定向模式 1 的情况。可以看到在模式 1 的情况下，地区记录和国家记录一同移走了，后面 3 个字节构成了一个指针，指向了实际的国家名，然后又跟着地区名。模式 1 的标识字节是 0x01。

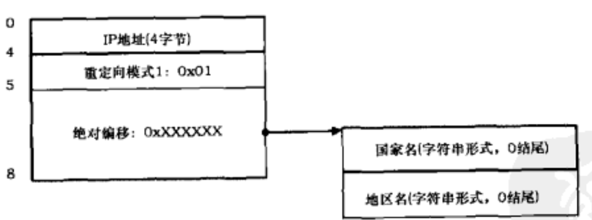



图 3.4　重定向模式 1

图 3.5 演示了重定向模式 2 的情况。在模式 2 的情况下(其标识字节是 0x02)，地区记录没有跟着国家记录移走，因此在国家记录之后 4 个字节之后还是有地区记录。我想你已经明白了模式 1 和模式 2 的区别，即：模式 1 的国家记录后面不会再有地区记录，模式 2 的国家记录后会有地区记录。下面来看一下更复杂的情况。

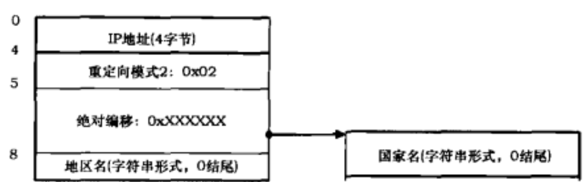



图 3.5　重定向模式 2

图 3.6 演示了当国家记录为模式 1 的时候可能出现的更复杂情况，在这种情况下，重定向指向的位置仍然是个重定向，不过第二次重定向为模式 2。这个重定向最多只有两次，并且如果发生了第二次重定向，则其一定为模式 2，而且这种情况只会发生在国家记录上，对于地区记录，模式 1 和模式 2 是一样的，地区记录也不会发生 2 次重定向。不过，图 3.5 还可以更复杂，变成图 3.6 和图 3.7 两种情况。

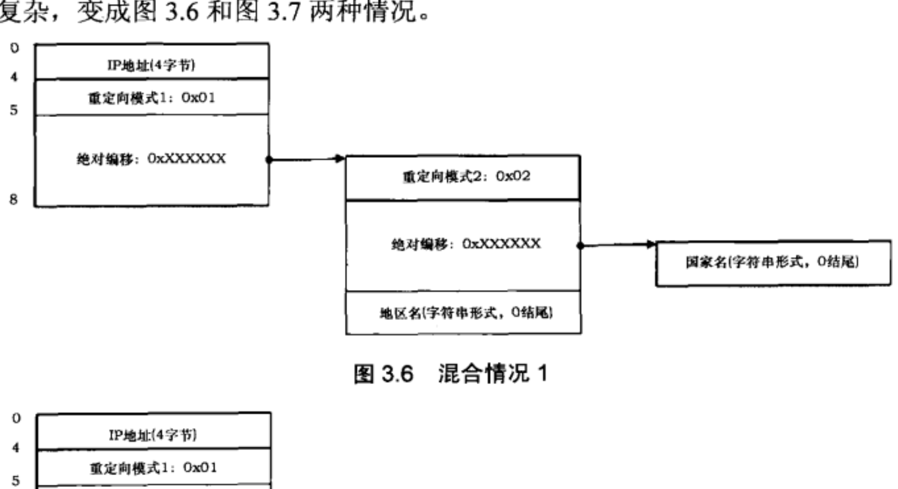



图 3.6　混合情况 1

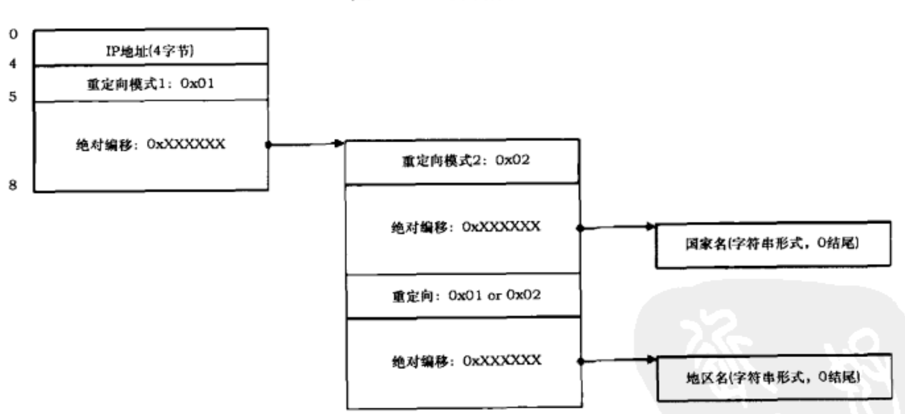



图 3.7　混合情况 2

总结如下：一条 IP 记录由[IP 地址][国家记录][地区记录]组成。对于国家记录，可以有三种表示方式：字符串形式、重定向模式 1 和重定向模式 2。对于地区记录，可以有两种表示方式：字符串形式和重定向。另外有一条规则：重定向模式 1 的国家记录后不能跟地区记录。按照这个总结，在这些方式中合理组合，就构成了 IP 记录的所有可能情况。

上文说明了文件头实际上是两个指针，分别指向了第一条索引和最后一条索引的绝对偏移，如图 3.8 所示。

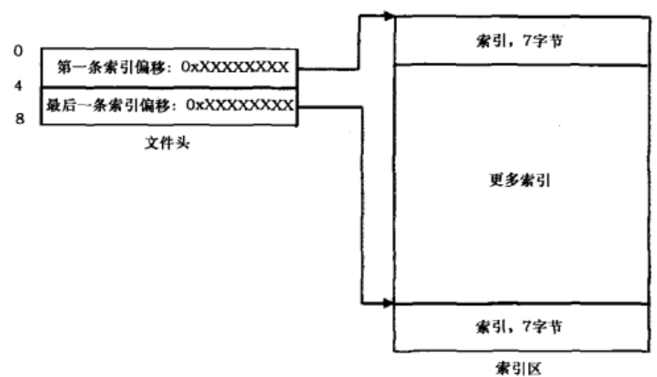



图 3.8　文件头指向索引区图示

实在是很简单，不是吗？从文件头就可以定位到索引区，然后就可以开始搜索 IP 了！每条索引长度为 7 个字节，前 4 个字节是起始 IP 地址，后三个字节就指向了 IP 记录。这里有些概念需要说明一下，什么是起始 IP，那么有没有结束 IP？ 假设有这么一条记录：166.111.0.0 - 166.111.255.255，那么 166.111.0.0 就是起始 IP，166.111.255.255 就是结束 IP，结束 IP 就是 IP 记录中的头 4 个字节。于是，每条索引配合一条记录，就构成了一个 IP 范围，如果要查找 166.111.138.138 所在的位置，就会发现 166.111.138.138 落在了 166.111.0.0 - 166.111.255.255 这个范围内，那么你就可以顺着这条索引去读取国家和地区名了。下面给出一个最详细的图解，如图 3.9 所示。

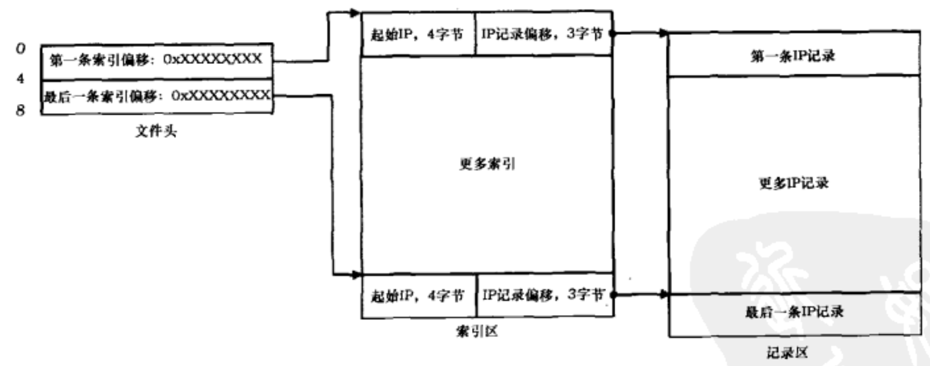



图 3.9　文件详细结构

现在一切都清楚了是不是？也许还有一点你不清楚，QQWry.dat 的版本信息保存在哪里了呢？答案是：最后一条 IP 记录实际上就是版本信息，最后一条记录显示的样子是：255.255.255.0 255.255.255.255 纯真网络 2004 年 6 月 25 日 IP 数据。好了，到现在你应该全部清楚了。

知道了纯真数据库文件的结构，那么，如何解析文件以及查找 IP 对应的地区呢？我们下一节会给出完整的程序，在做限定区域爬虫时，可以直接应用书中的程序。

最后介绍一下如何限制抓取特定语言的页面的问题，通常，这种爬虫有两个策略可以使用。第一是在下载一个页面的时候，先获得编码信息，通过编码来确定语言。但是，这样的抓取通常不准确。因为编码通常不能准确地表述语言信息。比如，UTF8 的编码，可能表示中文，也可能表示韩文。所以，通常采用另一种办法来限制，比如，中国境内的网站，通常都使用中文。而日本的网站，通常都是用日文。因此，通过限制 IP 的办法同样可以做到限制语言。

### 3.2.4 Java 限定爬虫示例

根据纯真数据库查找 IP 所对应的区域的代码如下：

```
public class IPSeeker {
    //纯真 IP 数据库名
    private String IP_FILE="QQWry.Dat";
    //保存的文件夹
    private String INSTALL_DIR="c:\\qqwry";


    // 一些固定常量，比如记录长度等
    private static final int IP_RECORD_LENGTH = 7;
    private static final byte REDIRECT_MODE_1 = 0x01;
    private static final byte REDIRECT_MODE_2 = 0x02;

    // 用来作为 cache，查询一个 IP 时首先查看 cache，以减少不必要的重复查找
    private Map<String, IPLocation> ipCache;
    // 随机文件访问类
    private RandomAccessFile ipFile;
    // 内存映射文件
    private MappedByteBuffer mbb;
    // 起始地区的开始和结束的绝对偏移
    private long ipBegin, ipEnd;
    // 为提高效率而采用的临时变量
    private IPLocation loc;
    private byte[] buf;
    private byte[] b4;
    private byte[] b3;

    public IPSeeker(String fileName,String dir)  {
        this.INSTALL_DIR=dir;
        this.IP_FILE=fileName;
        ipCache = new HashMap<String, IPLocation>();
        loc = new IPLocation();
        buf = new byte[100];
        b4 = new byte[4];
```

```
    b3 = new byte[3];
    try {
        ipFile = new RandomAccessFile(IP_FILE, "r");
    } catch (FileNotFoundException e) {
        //如果找不到这个文件，再尝试在当前目录下搜索，这次全部改用小写文件名
        //因为有些系统可能会区分大小写，导致找不到 IP 地址信息文件
        String filename = new File(IP_FILE).getName().toLowerCase();
        File[] files = new File(INSTALL_DIR).listFiles();
        for(int i = 0; i < files.length; i++) {
            if(files[i].isFile()) {
                if(files[i].getName().toLowerCase().equals(filename)) {
                    try {
                        ipFile = new RandomAccessFile(files[i], "r");
                    } catch (FileNotFoundException e1) {
                        LogFactory.log("IP 地址信息文件没有找到，IP 显示功能
                        将无法使用",Level.ERROR,e1);
                        ipFile = null;
                    }
                    break;
                }
            }
        }
    }
    // 如果打开文件成功，读取文件头信息
    if(ipFile != null) {
        try {
            ipBegin = readLong4(0);
            ipEnd = readLong4(4);
            if(ipBegin == -1 || ipEnd == -1) {
                ipFile.close();
                ipFile = null;
            }
        } catch (IOException e) {
            LogFactory.log("IP 地址信息文件格式有错误，IP 显示功能将无法使用",
            Level.ERROR,e);
            ipFile = null;
        }
    }
}

/**
 * 给定一个地点的不完全名字，得到一系列包含 s 子串的 IP 范围记录
 * @param s 地点子串
 * @return 包含 IPEntry 类型的 List
 */
public List getIPEntriesDebug(String s) {
```

```
    List<IPEntry> ret = new ArrayList<IPEntry>();
    long endOffset = ipEnd + 4;
    for(long offset = ipBegin + 4; offset <= endOffset; offset +=
    IP_RECORD_LENGTH) {
        // 读取结束 IP 偏移
        long temp = readLong3(offset);
        // 如果 temp 不等于-1，读取 IP 的地点信息
        if(temp != -1) {
            IPLocation ipLoc = getIPLocation(temp);
            // 判断这个地点里面是否包含 s 子串，如果包含，添加这个记录到 List 中，
            // 如果没有，继续
            if(ipLoc.getCountry().indexOf(s) != -1 ||
            ipLoc.getArea().indexOf(s) != -1) {
                IPEntry entry = new IPEntry();
                entry.country = ipLoc.getCountry();
                entry.area = ipLoc.getArea();
                // 得到起始 IP
             readIP(offset - 4, b4);
                entry.beginIp = Util.getIpStringFromBytes(b4);
                // 得到结束 IP
                readIP(temp, b4);
                entry.endIp = Util.getIpStringFromBytes(b4);
                // 添加该记录
                ret.add(entry);
            }
        }
    }
    return ret;
}

public IPLocation getIPLocation(String ip){
    IPLocation location=new IPLocation();
    location.setArea(this.getArea(ip));
    location.setCountry(this.getCountry(ip));
    return location;
}

/**
 * 给定一个地点的不完全名字，得到一系列包含 s 子串的 IP 范围记录
 * @param s 地点子串
 * @return 包含 IPEntry 类型的 List
 */
public List<IPEntry> getIPEntries(String s) {
    List<IPEntry> ret = new ArrayList<IPEntry>();
    try {
        // 映射 IP 信息文件到内存中
        if(mbb == null) {
```

```
            FileChannel fc = ipFile.getChannel();
            mbb = fc.map(FileChannel.MapMode.READ_ONLY, 0,
            ipFile.length());
            mbb.order(ByteOrder.LITTLE_ENDIAN);
        }

        int endOffset = (int)ipEnd;
        for(int offset = (int)ipBegin + 4; offset <= endOffset; offset +=
        IP_RECORD_LENGTH) {
            int temp = readInt3(offset);
            if(temp != -1) {
            IPLocation ipLoc = getIPLocation(temp);
            // 判断这个地点里面是否包含 s 子串，如果包含，在 List 中添加这个记录，
            // 如果没有，继续
            if(ipLoc.getCountry().indexOf(s) != -1 ||
            ipLoc.getArea().indexOf(s) != -1) {
                IPEntry entry = new IPEntry();
                entry.country = ipLoc.getCountry();
                entry.area = ipLoc.getArea();
                // 得到起始 IP
                readIP(offset - 4, b4);
                entry.beginIp = Util.getIpStringFromBytes(b4);
                // 得到结束 IP
                readIP(temp, b4);
                entry.endIp = Util.getIpStringFromBytes(b4);
                // 添加该记录
                ret.add(entry);
            }
            }
        }
    } catch (IOException e) {
        LogFactory.log("",Level.ERROR,e);
    }
    return ret;
}

/**
 * 从内存映射文件 offset 位置开始的第 3 个字节起读取一个 int
 * @param offset
 * @return
 */
private int readInt3(int offset) {
    mbb.position(offset);
    return mbb.getInt() & 0x00FFFFFF;
}

/**
```

```
 * 从内存映射文件当前位置开始的第 3 个字节起读取一个 int
 * @return
 */
private int readInt3() {
    return mbb.getInt() & 0x00FFFFFF;
}

/**
 * 根据 IP 得到国家名
 * @param ip IP 的字节数组形式
 * @return 国家名字符串
 */
public String getCountry(byte[] ip) {
     // 检查 IP 地址文件是否正常
     if(ipFile == null)
         return Message.bad_ip_file;
     // 保存 IP，转换 IP 字节数组为字符串形式
     String ipStr = Util.getIpStringFromBytes(ip);
     // 先检查 cache 中是否已经包含这个 IP 的结果，没有再搜索文件
     if(ipCache.containsKey(ipStr)) {
         IPLocation ipLoc = ipCache.get(ipStr);
         return ipLoc.getCountry();
     } else {
         IPLocation ipLoc = getIPLocation(ip);
         ipCache.put(ipStr, ipLoc.getCopy());
         return ipLoc.getCountry();
     }
}

/**
 * 根据 IP 得到国家名
 * @param ip IP 的字符串形式
 * @return 国家名字符串
 */
public String getCountry(String ip) {
    return getCountry(Util.getIpByteArrayFromString(ip));
}

/**
 * 根据 IP 得到地区名
 * @param ip IP 的字节数组形式
 * @return 地区名字符串
 */
public String getArea(byte[] ip) {
     // 检查 IP 地址文件是否正常
     if(ipFile == null)
         return Message.bad_ip_file;
```

```
    // 保存 IP，转换 IP 字节数组为字符串形式
    String ipStr = Util.getIpStringFromBytes(ip);
    // 先检查 cache 中是否已经包含这个 IP 的结果，没有再搜索文件
    if(ipCache.containsKey(ipStr)) {
        IPLocation ipLoc = ipCache.get(ipStr);
        return ipLoc.getArea();
    } else {
        IPLocation ipLoc = getIPLocation(ip);
        ipCache.put(ipStr, ipLoc.getCopy());
        return ipLoc.getArea();
    }
}

/**
 * 根据 IP 得到地区名
 * @param ip IP 的字符串形式
 * @return 地区名字符串
 */
public String getArea(String ip) {
    return getArea(Util.getIpByteArrayFromString(ip));
}

/**
 * 根据 ip 搜索 IP 信息文件，得到 IPLocation 结构，所搜索的 ip 参数从类成员 ip 中得到
 * @param ip 要查询的 IP
 * @return IPLocation 结构
 */
private IPLocation getIPLocation(byte[] ip) {
    IPLocation info = null;
    long offset = locateIP(ip);
    if(offset != -1)
        info = getIPLocation(offset);
    if(info == null) {
        info = new IPLocation();
        info.setCountry (Message.unknown_country);
        info.setArea(Message.unknown_area);
    }
    return info;
}

/**
 * @param offset
 * @return 读取的 long 值，返回-1 表示读取文件失败
 */
private long readLong4(long offset) {
    long ret = 0;
    try {
```

```
            ipFile.seek(offset);
            ret |= (ipFile.readByte() & 0xFF);
            ret |= ((ipFile.readByte() << 8) & 0xFF00);
            ret |= ((ipFile.readByte() << 16) & 0xFF0000);
            ret |= ((ipFile.readByte() << 24) & 0xFF000000);
            return ret;
        } catch (IOException e) {
            return -1;
        }
    }

    /**
     * @param offset 整数的起始偏移
     * @return 读取的 long 值，返回-1 表示读取文件失败
     */
    private long readLong3(long offset) {
        long ret = 0;
        try {
            ipFile.seek(offset);
            ipFile.readFully(b3);
            ret |= (b3[0] & 0xFF);
            ret |= ((b3[1] << 8) & 0xFF00);
            ret |= ((b3[2] << 16) & 0xFF0000);
            return ret;
        } catch (IOException e) {
            return -1;
        }
    }

    /**
     * 从当前位置读取 3 个字节转换成 long
     * @return 读取的 long 值，返回-1 表示读取文件失败
     */
    private long readLong3() {
        long ret = 0;
        try {
            ipFile.readFully(b3);
            ret |= (b3[0] & 0xFF);
            ret |= ((b3[1] << 8) & 0xFF00);
            ret |= ((b3[2] << 16) & 0xFF0000);
            return ret;
        } catch (IOException e) {
            return -1;
        }
    }

    /**
```

```
 * 从 offset 位置读取四个字节的 IP 地址放入 ip 数组中，读取后的 ip 为 Big-endian 格
 * 式，但是文件中是 Little-endian 形式，将会进行转换
 * @param offset
 * @param ip
 */
private void readIP(long offset, byte[] ip) {
    try {
        ipFile.seek(offset);
        ipFile.readFully(ip);
        byte temp = ip[0];
        ip[0] = ip[3];
        ip[3] = temp;
        temp = ip[1];
        ip[1] = ip[2];
        ip[2] = temp;
    } catch (IOException e) {
       LogFactory.log("",Level.ERROR,e);
    }
}

/**
 * 从 offset 位置读取四个字节的 IP 地址放入 ip 数组中，读取后的 ip 为 Big-endian 格
 * 式，但是文件中是 little-endian 形式，将会进行转换
 * @param offset
 * @param ip
 */
private void readIP(int offset, byte[] ip) {
    mbb.position(offset);
    mbb.get(ip);
    byte temp = ip[0];
    ip[0] = ip[3];
    ip[3] = temp;
    temp = ip[1];
    ip[1] = ip[2];
    ip[2] = temp;
}

/**
 * 把类成员 ip 和 beginIp 比较，注意这个 beginIp 是 Big-endian 格式的
 * @param ip 要查询的 IP
 * @param beginIp 和被查询 IP 相比较的 IP
 * @return 若 ip 和 begin 相等返回 0，ip 大于 beginIp 则返回 1，小于返回-1
 */
private int compareIP(byte[] ip, byte[] beginIp) {
    for(int i = 0; i < 4; i++) {
        int r = compareByte(ip[i], beginIp[i]);
        if(r != 0)
```

```
                return r;
        }
        return 0;
    }

    /**
     * 把两个 byte 当作无符号数进行比较
     * @param b1
     * @param b2
     * @return 若 b1 大于 b2 则返回 1，相等返回 0，小于返回-1
     */
    private int compareByte(byte b1, byte b2) {
        if((b1 & 0xFF) > (b2 & 0xFF)) // 比较是否大于
            return 1;
        else if((b1 ^ b2) == 0)// 判断是否相等
            return 0;
        else
            return -1;
    }

    /**
     * 这个方法将根据 IP 的内容，定位到包含这个 IP 国家地区的记录处，返回一个绝对偏移
     * 使用二分法查找
     * @param ip 要查询的 IP
     * @return 如果找到了，返回结束 IP 的偏移，如果没有找到，返回-1
     */
    private long locateIP(byte[] ip) {
        long m = 0;
        int r;
        // 比较第一个 ip 项
        readIP(ipBegin, b4);
        r = compareIP(ip, b4);
        if(r == 0) return ipBegin;
        else if(r < 0) return -1;
        // 开始二分搜索
        for(long i = ipBegin, j = ipEnd; i < j; ) {
            m = getMiddleOffset(i, j);
            readIP(m, b4);
            r = compareIP(ip, b4);
            // log.debug(Utils.getIpStringFromBytes(b));
            if(r > 0)
                i = m;
            else if(r < 0) {
                if(m == j) {
                    j -= IP_RECORD_LENGTH;
                    m = j;
                } else
```

```
            j = m;
        } else
            return readLong3(m + 4);
    }
    // 如果循环结束了，那么 i 和 j 必定是相等的，这个记录为最可能的记录，但是并非
    // 肯定就是，还要检查一下，如果是，就返回结束地址区的绝对偏移
    m = readLong3(m + 4);
    readIP(m, b4);
    r = compareIP(ip, b4);
    if(r <= 0) return m;
    else return -1;
}

/**
 * 得到 begin 偏移和 end 偏移中间位置记录的偏移
 * @param begin
 * @param end
 * @return
 */
private long getMiddleOffset(long begin, long end) {
    long records = (end - begin) / IP_RECORD_LENGTH;
    records >>= 1;
    if(records == 0) records = 1;
    return begin + records * IP_RECORD_LENGTH;
}

/**
 * 给定一个 IP 国家地区记录的偏移，返回一个 IPLocation 结构
 * @param offset 国家记录的起始偏移
 * @return IPLocation 对象
 */
private IPLocation getIPLocation(long offset) {
    try {
        // 跳过 4 字节 ip
        ipFile.seek(offset + 4);
        // 读取第一个字节判断是否标志字节
        byte b = ipFile.readByte();
        if(b == REDIRECT_MODE_1) {
            // 读取国家偏移
            long countryOffset = readLong3();
            // 跳转至偏移处
            ipFile.seek(countryOffset);
            // 再检查一次标志字节，因为这个时候这个地方仍然可能是个重定向
            b = ipFile.readByte();
            if(b == REDIRECT_MODE_2) {
                loc.setCountry (  readString(readLong3()));
                ipFile.seek(countryOffset + 4);
```

```
            } else
                loc.setCountry ( readString(countryOffset));
            // 读取地区标志
            loc.setArea( readArea(ipFile.getFilePointer()));
        } else if(b == REDIRECT_MODE_2) {
            loc.setCountry ( readString(readLong3()));
            loc.setArea( readArea(offset + 8));
        } else {
            loc.setCountry (  readString(ipFile.getFilePointer() - 1));
            loc.setArea( readArea(ipFile.getFilePointer()));
        }
        return loc;
    } catch (IOException e) {
        return null;
    }
}

/**
 * 给定一个 IP 国家地区记录的偏移，返回一个 IPLocation 结构，此方法应用于内存映射文
 * 件方式
 * @param offset 国家记录的起始偏移
 * @return IPLocation 对象
 */
private IPLocation getIPLocation(int offset) {
    // 跳过 4 字节 IP
    mbb.position(offset + 4);
    //判断第一个字节是否为标志字节
    byte b = mbb.get();
    if(b == REDIRECT_MODE_1) {
        // 读取国家偏移
        int countryOffset = readInt3();
        // 跳转至偏移处
        mbb.position(countryOffset);
        // 再检查一次标志字节，因为这个时候这个地方仍然可能是个重定向
        b = mbb.get();
        if(b == REDIRECT_MODE_2) {
            loc.setCountry (  readString(readInt3()));
            mbb.position(countryOffset + 4);
        } else
            loc.setCountry(readString(countryOffset));
        // 读取地区标志
        loc.setArea(readArea(mbb.position()));
    } else if(b == REDIRECT_MODE_2) {
        loc.setCountry(readString(readInt3()));
        loc.setArea(readArea(offset + 8));
    } else {
        loc.setCountry(readString(mbb.position() - 1));
```

```
            loc.setArea(readArea(mbb.position()));
        }
        return loc;
    }

    /**
     * 从offset偏移开始解析后面的字节，读出一个地区名
     * @param offset 地区记录的起始偏移
     * @return 地区名字符串
     * @throws IOException
     */
    private String readArea(long offset) throws IOException {
        ipFile.seek(offset);
        byte b = ipFile.readByte();
        if(b == REDIRECT_MODE_1 || b == REDIRECT_MODE_2) {
            long areaOffset = readLong3(offset + 1);
            if(areaOffset == 0)
                return Message.unknown_area;
            else
                return readString(areaOffset);
        } else
            return readString(offset);
    }

    /**
     * @param offset 地区记录的起始偏移
     * @return 地区名字符串
     */
    private String readArea(int offset) {
        mbb.position(offset);
        byte b = mbb.get();
        if(b == REDIRECT_MODE_1 || b == REDIRECT_MODE_2) {
            int areaOffset = readInt3();
            if(areaOffset == 0)
                return Message.unknown_area;
            else
                return readString(areaOffset);
        } else
            return readString(offset);
    }

    /**
     * 从offset偏移处读取一个以0结束的字符串
     * @param offset 字符串起始偏移
     * @return 读取的字符串，出错返回空字符串
     */
    private String readString(long offset) {
```

```
        try {
            ipFile.seek(offset);
            int i;
            for(i = 0, buf[i] = ipFile.readByte(); buf[i] != 0; buf[++i] =
            ipFile.readByte());
            if(i != 0)
                return Util.getString(buf, 0, i, "GBK");
        } catch (IOException e) {
            LogFactory.log("",Level.ERROR,e);
        }
        return "";
    }

    /**
     * 从内存映射文件的offset位置得到一个0结尾字符串
     * @param offset 字符串起始偏移
     * @return 读取的字符串，出错返回空字符串
     */
    private String readString(int offset) {
        try {
            mbb.position(offset);
            int i;
            for(i = 0, buf[i] = mbb.get(); buf[i] != 0; buf[++i] = mbb.get());
            if(i != 0)
                return Util.getString(buf, 0, i, "GBK");
        } catch (IllegalArgumentException e) {
            LogFactory.log("",Level.ERROR,e);
        }
        return "";
    }
}

/**
 * 工具类，提供一些方便的方法
 */
public class Util {

    private static StringBuilder sb = new StringBuilder();

    /**
     * 从IP的字符串形式得到字节数组形式
     * @param ip 字符串形式的IP
     * @return 字节数组形式的IP
     */
    public static byte[] getIpByteArrayFromString(String ip) {
        byte[] ret = new byte[4];
        StringTokenizer st = new StringTokenizer(ip, ".");
```

```
        try {
            ret[0] = (byte) (Integer.parseInt(st.nextToken()) & 0xFF);
            ret[1] = (byte) (Integer.parseInt(st.nextToken()) & 0xFF);
            ret[2] = (byte) (Integer.parseInt(st.nextToken()) & 0xFF);
            ret[3] = (byte) (Integer.parseInt(st.nextToken()) & 0xFF);
        } catch (Exception e) {
            LogFactory.log("从IP的字符串形式得到字节数组形式报错", Level.ERROR, e);
        }
        return ret;
    }

    /**
     * @param ip
     *IP的字节数组形式
     * @return 字符串形式的IP
     */
    public static String getIpStringFromBytes(byte[] ip) {
        sb.delete(0, sb.length());
        sb.append(ip[0] & 0xFF);
        sb.append('.');
        sb.append(ip[1] & 0xFF);
        sb.append('.');
        sb.append(ip[2] & 0xFF);
        sb.append('.');
        sb.append(ip[3] & 0xFF);
        return sb.toString();
    }

    /**
     * 根据某种编码方式将字节数组转换成字符串
     * @param b 字节数组
     * @param offset 要转换的起始位置
     * @param len 要转换的长度
     * @param encoding 编码方式
     * @return 如果encoding不支持，返回一个默认编码的字符串
     */
    public static String getString(byte[] b, int offset, int len,
            String encoding) {
        try {
            return new String(b, offset, len, encoding);
        } catch (UnsupportedEncodingException e) {
            return new String(b, offset, len);
        }
    }
}

public interface Message {
```

```
    String bad_ip_file="IP 地址库文件错误";
    String unknown_country="未知国家";
    String unknown_area="未知地区";
}

/**
 * <pre>
 * 一条 IP 范围记录，不仅包括国家和区域，也包括起始 IP 和结束 IP
 * </pre>
 */
public class IPEntry {
   public String beginIp;
   public String endIp;
   public String country;
   public String area;

   /**
    * 构造函数
    */
   public IPEntry() {
      beginIp = endIp = country = area = "";
   }
}

/**
 * @category 用来封装 IP 相关信息，目前只有两个字段，IP 所在的国家和地区
 */

public class IPLocation {
    private String country;
    private String area;

    public IPLocation() {
        country = area = "";
    }

    public IPLocation getCopy() {
        IPLocation ret = new IPLocation();
        ret.country = country;
        ret.area = area;
        return ret;
    }

    public String getCountry() {
        return country;
    }
```

```
    public void setCountry(String country) {
        this.country = country;
    }

    public String getArea() {
        return area;
    }

    public void setArea(String area) {
              //如果为局域网，纯真 IP 地址库的地区会显示 CZ88.NET，这里把它去掉
        if(area.trim().equals("CZ88.NET")){
            this.area="本机或本网络";
           }else{
            this.area = area;
        }
    }
}

/**
 * 日志工厂
 */
public class LogFactory {
    private static final Logger logger;
    static {
        logger = Logger.getLogger("stdout");
        logger.setLevel(Level.DEBUG);
    }

    public static void log(String info, Level level, Throwable ex) {
        logger.log(level, info, ex);
    }

    public static Level getLogLevel() {
        return logger.getLevel();
    }
}

public class IPTest {

    public static void main(String[] args) {
        // 指定纯真数据库的文件名和所在文件夹
        IPSeeker ip = new IPSeeker("QQWry.Dat", "c:\\qqwry");
        // 测试 IP 58.20.43.13
        System.out.println(ip.getIPLocation("58.20.43.13").getCountry() + ":"
                + ip.getIPLocation("58.20.43.13").getArea());
    }
}
```

运行程序，输出：

```
湖南省长沙市:联通
```

说明 IP 地址为 58.20.43.13 的区域属于湖南省长沙市。如果你想限制不抓取湖南省长沙市的 IP 的话，那就不能抓取这个主机的文件。

## 3.3 有“道德”的爬虫

作为一个爬虫程序，在网上应该遵守爬虫的“道德”。何谓爬虫的“道德”？因为爬虫会访问网站资源，并下载其中的资源。尤其是多线程爬虫，可能会在访问一个网站的时候开启多个线程，然后使用很多 Session 进行连接，爬取网页。这样，很容易造成网站瘫痪、不能访问等后果。还有一种情况是网站有很多东西属于自己的“秘密”，本身就不想让爬虫抓取，如果爬虫随意抓取，就相当于侵犯网站的“隐私”。

为了避免这种情况的发生，互联网行业采用了两种规定，第一种是在网站的根目录下放置一个文件，起名为 robots.txt，其中规定了哪些内容不想被抓取。另一种办法是设置 Robots Meta 标签。

robots.txt 文件必须放置在站点的根目录下，而且文件名必须小写。该文件包含一条或更多的记录，这些记录用空行分开(以 CR、CR/NL 或者 NL 作为结束符)，在该文件中可以用#进行注解，具体使用方法和 UNIX 中的惯例一样。该文件中的记录通常以一行或多行 User-agent 开始，后面加上若干 Disallow 行，详细情况如下。

User-agent：该项的值用于描述搜索引擎 robots 的名字，在 robots.txt 文件中，如果有多条 User-agent 记录，说明有多个 robots 会受到该协议的限制，对该文件来说，至少要有一条 User-agent 记录。如果 User-agent 的值设为*，则该协议对任何机器人均有效，在 robots.txt 文件中，“User-agent:*”这样的记录只能有一条。

Disallow：该项的值用于描述不希望被访问到的 URL，这个 URL 可以是一条完整的路径，也可以是部分路径，任何以 Disallow 开头的 URL 均不会被 robots 访问到。例如“Disallow:/help”对/help.html 和/help/index.html 都不允许搜索引擎访问，而“Disallow:/help/”则允许 robots 访问/help.html，而不能访问/help/index.html。

任何一条 Disallow 记录为空，说明该网站的所有部分都允许被访问，在 robots.txt 文件中，至少要有一条 Disallow 记录。如果 robots.txt 是一个空文件，则该网站对于所有的搜索引擎 robots，都是开放的。一个 robots.txt 的例子如下：

```
User-agent: *
Disallow: /cgi-bin/
Disallow: /tmp/
Disallow: /private/
```

以上例子表明这个网站的 robots 协议对每个爬虫都适用。并且不允许爬取/cgi-bin/、/tmp/和/private/下的文件。

下面的例子是 Heritrix 中处理 robots.txt 协议的代码：

```
//是否允许访问
public boolean disallows(CrawlURI curi, String userAgent) {
        if (this == ALLOWALL)
            return false;
        if (this == DENYALL)
            return true;
        // 同一台服务器上
        if((honoringPolicy.isType(curi, RobotsHonoringPolicy.CLASSIC)
                || honoringPolicy.isType(curi, RobotsHonoringPolicy.CUSTOM))
            && (lastUsedUserAgent == null
            || !lastUsedUserAgent.equals(userAgent))) {

            lastUsedUserAgent = userAgent;
            userAgentsToTest = new ArrayList();
            Iterator iter = userAgents.iterator();
            while ( iter.hasNext() ) {
                String ua = (String)iter.next();
                if (userAgent.indexOf(ua)>-1) {
                    userAgentsToTest.add(ua);
                    break;
                }
            }
        }
        //返回值，初始值为 true
        boolean disallow = false;
        boolean examined = false;
        String ua = null;

        //得到不可访问的爬虫程序名的列表
        Iterator uas = userAgentsToTest.iterator();
        while(uas.hasNext() && examined == false) {
            disallow = false;
            ua = (String)uas.next();
            Iterator dis = ((List) disallows.get(ua)).iterator();

            // 检查当前的爬虫是否可以访问
            while(dis.hasNext() && examined == false && disallow == false) {
                String disallowedPath = (String) dis.next();
                if(disallowedPath.length() == 0) {
                    examined = true;
                    disallow = false;
                    break;
                }
                try {
                    String p = curi.getUURI().getPathQuery();
                    if (p != null && p.startsWith(disallowedPath) ) {
```

```
                    disallow = true;
                }
            }
            catch (URIException e) {
                logger.log(Level.SEVERE,"Failed getPathQuery from " + curi, e);
            }
        }
        if(disallow == false) {
            examined = true;
        }
    }

    if(honoringPolicy.shouldMasquerade(curi) && ua != null
&& !ua.equals("")) {
        curi.setUserAgent(ua);
    }
    return disallow;
}
```

在 HttpClient 包中，当执行 get 或者 post 方法时，会默认地提供对 robots.txt 的支持，不抓取 Disallow 规定的目录下的网页。当然，HttpClient 也设置了默认选项，可以让编写的爬虫不受 robots.txt 协议的限制，但是，作为一个网络爬虫作者，我们还是提倡有“道德”的抓取，以保障互联网行业健康发展。

现在我们讨论前面第二种方法，即通过设置 Robots Meta 的值来限制爬虫的访问。这种方法是一种细粒度的方法，能够把限制细化到每个网页。和其他的 META 标签(如使用的语言、页面的描述、关键词等)一样，Robots Meta 标签也放在页面的<head></head>中，专门用来告诉搜索引擎 Robots 如何抓取该页的内容。

Robots Meta 标签中没有大小写之分，name="Robots"表示所有的搜索引擎，可以针对某个具体搜索引擎写为 name="BaiduSpider"。Content 部分有 4 个指令选项：INDEX、NOINDEX、FOLLOW 和 NOFOLLOW，指令间以“,”分隔。

INDEX 指令告诉搜索机器人抓取该页面；

FOLLOW 指令表示搜索机器人可以沿着该页面上的链接继续抓取下去；

Robots Meta 标签的默认值是 INDEX 和 FOLLOW，只有名为 inktomi 的标签除外，对于它，默认值是 INDEX 和 NOFOLLOW。

这样，一共有 4 种组合：

```
<META NAME="ROBOTS" CONTENT="INDEX,FOLLOW">
<META NAME="ROBOTS" CONTENT="NOINDEX,FOLLOW">
<META NAME="ROBOTS" CONTENT="INDEX,NOFOLLOW">
<META NAME="ROBOTS" CONTENT="NOINDEX,NOFOLLOW">
```

其中：

<META NAME="ROBOTS" CONTENT="INDEX,FOLLOW">可以写成<META

NAME="ROBOTS" CONTENT="ALL">

<META NAME="ROBOTS" CONTENT="NOINDEX,NOFOLLOW">可以写成<META NAME="ROBOTS" CONTENT="NONE">

目前看来，绝大多数的搜索引擎机器人都遵守 robots.txt 的规则，而对于 Robots Meta 标签，目前支持的并不多，但是正在逐渐增加，如著名搜索引擎 Google 就完全支持，而且 Google 还增加了一个指令 archive，可以限制 Google 是否保留网页快照。例如：

```
<META NAME="googlebot" CONTENT="index,follow,noarchive">
```

表示抓取该站点中的页面并沿着页面中的链接继续抓取，但是不在 Goolge 上保留该页面的网页快照。

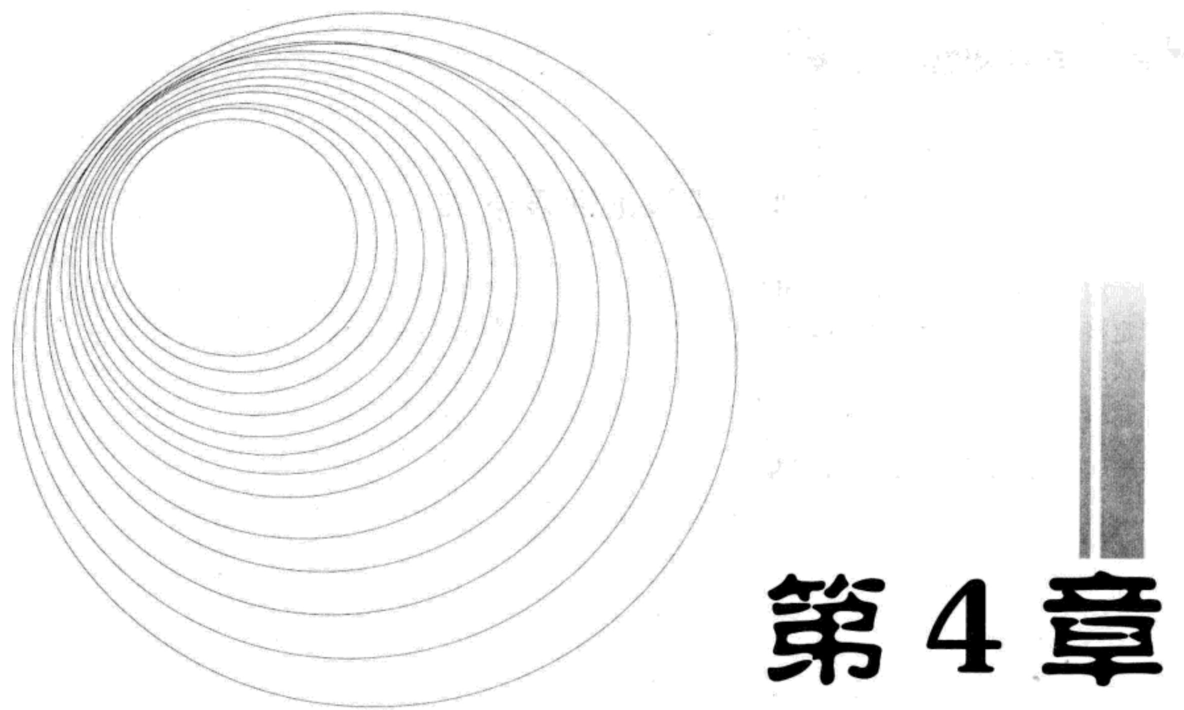
## 3.4 本章小结

本章主要讲述了在开发一个爬虫时，可能会涉及的一些问题以及解决方案。首先，讲解了爬虫的“黑洞”问题，但是由于这几年互联网的发展，遇到的黑洞问题越来越少。因此，本章没有展开讲解它的解决方案。接着，详细地讲述了主题爬虫和限定爬虫。给出了主题爬虫和限定爬虫的概念和相关的解决方案。还提到了主题相关度的算法以及 QQWry.dat 的纯真数据库。最后一节讲述了爬虫的道德，详细解释了 Robots.txt 文件。下面是一些相关的资源，希望能对大家进一步学习有所帮助。

- http://www.xd-tech.com.cn/blog/attachments/month_0610/g200610110544.pdf——此论文讲述了主题爬虫的一些概念、解决方案以及处理的核心代码。
- http://www.doc88.com/p-5780383971.html——基于遗传算法的主题爬虫，把人工智能应用在爬虫中，是一个不错的启发式方法。
- http://www.qikan.com.cn/Article/jsjy/jsjy200909/jsjy20090961.html——基于贝叶斯分类的主题爬虫，对机器学习有兴趣的朋友可以好好看看。
- http://hi.baidu.com/anspider/blog/item/665ef4447cd27b20cefca37a.html——有关限定爬虫的一点资料。

# 第2篇

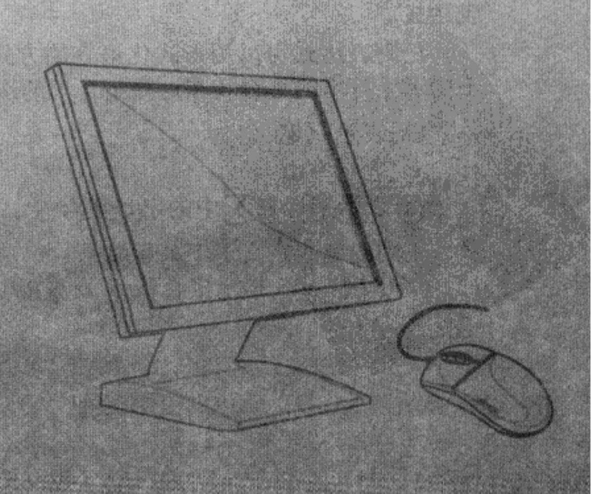
## 自己动手抽取 Web 内容



# 第4章

## “处理”HTML页面

在网络爬虫中，我们抓取最多的内容就是HTML页面，但是，许多搜索引擎感兴趣的内容是隐藏于HTML中的，而不是页面本身。这时候，如何处理抓取的HTML页面，以期得到真正需要的内容，就成为爬虫中一个重要的主题。这一章，我们就为读者解决这个难题。



## 4.1 征服正则表达式

在 HTML 处理中，经常会用到正则表达式。比如，在抽取网页链接时，只想抽取某种特定格式的链接加入 TODO 表中，而其他不符合格式的链接直接丢弃。如下面的正则表达式：

```
<[aA]\s+.*?[hH][rR][eE][fF]=\s*("|\')?(.*)(\1)(\s[^>]*)*?>(.*?)<\/[aA]>
```

这个正则表达式支持下面的格式：

```
<a href="http://www.baidu.com">Baidu</a>
<a href='http://www.baidu.com'>Baidu</a>
<a href=http://www.baidu.com>Baidu</a>
<a class="123" title="abc" href="http://www.baidu.com">Baidu</a>
<a href="http://www.baidu.com">Baidu</a>
<a href=http://www.baidu.com     >Baidu</a>
```

因此，若把表达式<[aA]\s+.*?[hH][rR][eE][fF]=\s*("|\')?(.*)(\1)(\s[^>]*)*?>(.*?)<\/[aA]>应用于爬虫项目，则这个项目中所扩展的 URL 就类似于上面列举出来的 URL，而其他模式的 URL 则被抛弃，不进行抓取。

正则表达式在网络爬虫中，主要有以下两方面的应用：

- 对 URL 链接进行过滤，只提取符合特定格式的链接。
- 提取网页内容。

这一节，我们先从正则表达式的基础讲起，并配合一些小的示例，介绍正则表达式在 Java 中的应用。

### 4.1.1 学习正则表达式

正则表达式(Regular Expression)在计算机科学中，是指一个用来描述或者匹配一系列符合某个句法规则的字符串的单个字符串。在很多文本编辑器或其他工具里，正则表达式通常被用来检索和/或替换那些符合某个模式的文本内容。许多程序设计语言都支持利用正则表达式进行字符串操作。比如 Java 语言，在 JDK 1.4 之后，就内建了一个功能强大的正则表达式。

正则表达式可以用形式化语言的方式来表达。正则表达式由常量和算子组成，分别用于指示字符串的集合和在这些集合上的运算。给定有限字母表$\sum$定义了下列常量：

⊙ 表示空集

$\varepsilon$ 表示空串

{a}表示字母表$\sum$中的字母集合

正则表达式定义了下列运算：

(“串接”)RS 指示集合{$\alpha\beta \mid \alpha \in R \wedge \beta \in S$}，例如，{"ab"|"c"}{"d"|"ef"}={"abd","abef", "cd", "cef"}。

R|S 表示 R 和 S 的并集。

R*表示包含 ε 并且闭包在字符串串接下的 R 的最小超集。可以通过 R 中的零或多个字符串的串接得到所有字符串的集合，例如，{"ab", "c"}* = { ε , "ab", "c", "abab", "abc", "cab", "cc", "ababab", ... }。

为了避免括号，假定 Kleene 星号有最高优先级，然后是串接，接着是并集。如果没有歧义则可以省略括号。例如，(ab)c 可以写为 abc，而 a|(b(c*)) 可以写为 a|bc*。

例子：

a|b* 表示 { ε , a, b, bb, bbb, ...}。

(a|b)* 表示由空串、任意个 a 或 b 字符组成的所有字符串的集合。

ab*(c| ε ) 表示开始于一个 a，接着是 0 或多个 b，最后接一个可能包含 c 的字符串。

正则表达式有多种不同的风格。表 4.1 列出了常用的正则表达式及其定义。

表 4.1 正则表达式列表

| 字 符 | 描 述 |
|---|---|
| \ | 将下一个字符标记为一个特殊字符、一个原义字符、一个向后引用或一个八进制转义符。例如，“n”匹配字符“n”，“\n”匹配一个换行符，序列“\\”匹配“\”而“\(”则匹配“(”。 |
| ^ | 匹配输入字符串的开始位置。如果设置了正则表达式对象的 Multiline 属性，“^”也匹配“\n”或“\r”之后的位置。 |
| $ | 匹配输入字符串的结束位置。如果设置了正则表达式对象的 Multiline 属性，“$”也匹配“\n”或“\r”之前的位置。 |
| * | 匹配前面的子表达式零次或多次。例如，“zo*”能匹配“z”以及“zoo”，“*”等价于“{0,}”。 |
| + | 匹配前面的子表达式一次或多次。例如，“zo+”能匹配“zo”以及“zoo”，但不能匹配“z”；“+”等价于“{1,}”。 |
| ? | 匹配前面的子表达式零次或一次。例如，“do(es)?”可以匹配“do”或“does”中的“do”，“?”等价于“{0,1}”。 |
| {n} | n 是一个非负整数。匹配确定的 n 次。例如，“o{2}”不能匹配“Bob”中的“o”，但是能匹配“food”中的两个“o”.。 |
| {n,} | n 是一个非负整数。至少匹配 n 次。例如，“o{2,}”不能匹配“Bob”中的“o”，但能匹配“foooood”中的所有“o”。“o{1,}”等价于“o+”，“o{0,}”则等价于“o*”。 |
| {n,m} | m 和 n 均为非负整数，其中 n≤m。最少匹配 n 次且最多匹配 m 次。例如，“o{1,3}”将匹配“fooooood”中的前三个“o”，“o{0,1}”等价于“o?”。请注意在逗号和两个数之间不能有空格。 |

续表

<table>
<tr><th>字　符</th><th>描　述</th></tr>
<tr><td>?</td><td>当该字符紧跟在任何一个其他限制符(*,+,?, {n}, {n,}, {n,m})后面时，匹配模式是非贪婪的。非贪婪模式尽可能少地匹配所搜索的字符串，而默认的贪婪模式则尽可能多地匹配所搜索的字符串。例如，对于字符串“oooo”，“o+?”将匹配单个“o”，而“o+”将匹配所有“o”。</td></tr>
<tr><td></td><td>匹配除“\n”之外的任何单个字符。要匹配包括“\n”在内的任何字符，请使用类似“[.\n]”的模式。</td></tr>
<tr><td>(pattern)</td><td>匹配 pattern 并获取这一匹配。所获取的匹配可以从产生的 Matches 集合得到，在 VBScript 中使用 SubMatches 集合，在 JScript 中则使用$0…$9 属性。要匹配圆括号字符，请使用“\()”或“\”。</td></tr>
<tr><td>(?:pattern)</td><td>匹配 pattern 但不获取匹配结果，也就是说这是一个非获取匹配，不进行存储供以后使用。例如“industr(?:y|ies)”就是一个比“industry|industries”更简略的表达式。</td></tr>
<tr><td>(?=pattern)</td><td>正向预查，在任何匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如，“Windows(?=95|98|NT|2000)”能匹配 Windows 2000 中的 Windows，但不能匹配 Windows 3.1 中的 Windows。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。</td></tr>
<tr><td>(?!pattern)</td><td>负向预查，在任何不匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如 Windows(?!95|98|NT|2000)能匹配 Windows 3.1 中的 Windows，但不能匹配 Windows 2000 中的 Windows。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。</td></tr>
<tr><td>x|y</td><td>匹配 x 或 y。例如，“z|food”能匹配“z”或“food”，“(z|f)ood”则匹配“zood”或“food”。</td></tr>
<tr><td>[xyz]</td><td>字符集合。匹配所包含的任意字符。例如，“[abc]”可以匹配“plain”中的“a”。</td></tr>
<tr><td>[^xyz]</td><td>负值字符集合。匹配未包含的任意字符。例如，“[^abc]”可以匹配“plain”中的“p”。</td></tr>
<tr><td>[a-z]</td><td>字符范围。匹配指定范围内的任意字符。例如，“[a-z]”可以匹配 a 到 z 范围内的任意小写字母字符。</td></tr>
<tr><td>[^a-z]</td><td>负值字符范围。匹配不在指定范围内的任意字符。例如，“[^a-z]”可以匹配不在 a 到 z 范围内的任意字符。</td></tr>
<tr><td>\b</td><td>匹配一个单词边界，也就是指单词和空格间的位置。例如，“er\b”可以匹配“never”中的“er”，但不能匹配“verb”中的“er”。</td></tr>
</table>

PDG

续表

| 字 符 | 描 述 |
|---|---|
| \B | 匹配非单词边界。“er\B”能匹配“verb”中的“er”，但不能匹配“never”中的“er”。 |
| \cx | 匹配由 x 指明的控制字符。例如，“\cM”匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的“c”字符。 |
| \d | 匹配一个数字字符。等价于“[0-9]”。 |
| \D | 匹配一个非数字字符。等价于“[^0-9]”。 |
| \f | 匹配一个换页符。等价于“\x0c”和“\cL”。 |
| \n | 匹配一个换行符。等价于“\x0a”和“\cJ”。 |
| \r | 匹配一个回车符。等价于“\x0d”和“\cM”。 |
| \s | 匹配任何空白字符，包括空格、制表符、换页符等，等价于“[\f\n\r\t\v]”。 |
| \S | 匹配任何非空白字符。等价于“[^\f\n\r\t\v]”。 |
| \t | 匹配一个制表符。等价于“\x09”和“\cI”。 |
| \v | 匹配一个垂直制表符。等价于“\x0b”和“\cK”。 |
| \w | 匹配包括下划线的任何单词字符。等价于“[A-Za-z0-9_]”。 |
| \W | 匹配任何非单词字符。等价于“[^A-Za-z0-9_]”。 |
| \xn | 匹配 n，其中 n 为十六进制转义值。十六进制转义值必须为确定的两个数字长。例如，“\x41”匹配“A”，“\x041”则等价于“\x04&1”。正则表达式中可以使用 ASCII 编码。 |
| \num | 匹配 num，其中 num 是一个正整数。例如，“(.)\1”匹配两个连续的相同字符。 |
| \n | 标识一个八进制转义值或一个向后引用。如果“\n”之前至少 n 个获取的子表达式，则“n”为向后引用。否则，如果 n 为八进制数字(0～7)，则 n 为一个八进制转义值。 |
| \nm | 标识一个八进制转义值或一个向后引用。如果“\nm”之前至少有 m 个获取，则“nm”为向后引用。如果“\nm”之前至少有 n 个获取，则“n”为一个后跟文字 m 的向后引用。如果前面的条件都不满足，若 n 和 m 均为八进制数字(0～7)，则\nm 将匹配八进制转义值 nm。 |
| \nml | 如果 n 为八进制数字(0～3)，且 m 和 l 均为八进制数字(0～7)，则匹配八进制转义值 nml。 |
| \un | 匹配 n，其中 n 是一个用四个十六进制数字表示的 Unicode 字符。例如，“\u00A9”匹配版权符号(©)。 |

### 4.1.2 Java 正则表达式

从 JDK 1.4 开始 Java.util.regex 包提供了对正则表达式的支持。下面的几个例子详细描述了用 Java 语言处理正则表达式的方法。

例：查找以 Java 开头，任意结尾的字符串。

```
Pattern pattern = Pattern.compile("^Java.*");
Matcher matcher = pattern.matcher("Java是一门编程语言");
boolean b= matcher.matches();
//当条件满足时，将返回 true，否则返回 false
System.out.println(b);
```

例：以多个条件分割字符串。

```
Pattern pattern = Pattern.compile("[, |]+");
String[] strs = pattern.split("Java Hello World  Java,Hello,,World|Sun");
for (int i=0;i<strs.length;i++) {
    System.out.println(strs[i]);
}
```

例：文字替换(首次出现字符)。

```
Pattern pattern = Pattern.compile("正则表达式");
Matcher matcher = pattern.matcher("正则表达式 Hello World,正则表达式 Hello
World");
//替换第一个符合正则表达式的数据
System.out.println(matcher.replaceFirst("Java"));
```

例：文字替换(全部)。

```
Pattern pattern = Pattern.compile("正则表达式");
Matcher matcher = pattern.matcher("正则表达式 Hello World,正则表达式 Hello
World");
//替换第一个符合正则表达式的数据
System.out.println(matcher.replaceAll("Java"));
```

例：文字替换(置换字符)。

```
Pattern pattern = Pattern.compile("正则表达式");
Matcher matcher = pattern.matcher("正则表达式 Hello World,正则表达式 Hello
World ");
StringBuffer sbr = new StringBuffer();
while (matcher.find()) {
    matcher.appendReplacement(sbr, "Java");
}
matcher.appendTail(sbr);
```

```
System.out.println(sbr.toString());
```

例：验证是否为邮箱地址。

```
String str="ceponline@yahoo.com.cn";
Pattern pattern = Pattern.compile("[\\w\\.\\-]+@([\\w\\-]+\\.)+[\\w\\-]+",
Pattern.CASE_INSENSITIVE);
Matcher matcher = pattern.matcher(str);
System.out.println(matcher.matches());
```

例：去除 html 标记。

```
Pattern pattern = Pattern.compile("<.+?>", Pattern.DOTALL);
Matcher matcher = pattern.matcher("<a href=\"index.html\">主页</a>");
String string = matcher.replaceAll("");
System.out.println(string);
```

例：查找 html 中对应条件字符串。

```
Pattern pattern = Pattern.compile("href=\"(.+?)\"");
Matcher matcher = pattern.matcher("<a href=\"index.html\">主页</a>");
if(matcher.find())
  System.out.println(matcher.group(1));
}
```

例：截取 http://地址。

```
//截取 URL
Pattern pattern = Pattern.compile("(http://|https://){1}[\\w\\.\\-/:]+");
Matcher matcher = pattern.matcher("dsdsds<http://dsds//gfgffdfd>fdf");
StringBuffer buffer = new StringBuffer();
while(matcher.find()){
    buffer.append(matcher.group());
    buffer.append("\r\n");
System.out.println(buffer.toString());
}
```

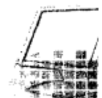
例：替换{}中的文字。

```
String str = "Java 目前的发展史是由{0}年-{1}年";
String[][] object={new String[]{"\\{0\\}","1995"},new
String[]{"\\{1\\}","2007"}};
System.out.println(replace(str,object));
public static String replace(final String sourceString,Object[] object) {
          String temp=sourceString;
          for(int i=0;i<object.length;i++){
                  String[] result=(String[])object[i];
            Pattern   pattern = Pattern.compile(result[0]);
            Matcher matcher = pattern.matcher(temp);
            temp=matcher.replaceAll(result[1]);
          }
```

```
        return temp;
}
```

例：以正则条件查询指定目录下的文件。

```
//用于缓存文件列表
private ArrayList files = new ArrayList();
//表示文件路径
private String _path;
//表示未合并的正则表达式
private String _regexp;

class MyFileFilter implements FileFilter {
    /**
     * 匹配文件名称
     */
    public boolean accept(File file) {
      try {
        Pattern pattern = Pattern.compile(_regexp);
        Matcher match = pattern.matcher(file.getName());
        return match.matches();
      } catch (Exception e) {
        return true;
      }
    }
  }

/**
 * 解析输入流
 * @param inputs
 */
FilesAnalyze (String path,String regexp){
    getFileName(path,regexp);
}

/**
 * 分析文件名并加入 files
 * @param input
 */
private void getFileName(String path,String regexp) {
    //目录
      _path=path;
      _regexp=regexp;
      File directory = new File(_path);
      File[] filesFile = directory.listFiles(new MyFileFilter());
      if (filesFile == null) return;
      for (int j = 0; j < filesFile.length; j++) {
        files.add(filesFile[j]);
```



```
        }
        return;
      }

    /**
     * 显示输出信息
     * @param out
     */
    public void print (PrintStream out) {
        Iterator elements = files.iterator();
        while (elements.hasNext()) {
            File file=(File) elements.next();
                out.println(file.getPath());
        }
    }

    public static void output(String path,String regexp) {

        FilesAnalyze fileGroup1 = new FilesAnalyze(path,regexp);
        fileGroup1.print(System.out);
    }

    public static void main (String[] args) {
        output("C:\\","[A-z|.]*");
    }
```

上面的例子深入浅出地介绍了 Java 处理正则表达式的方法。在本节的最后，结合本书内容，给出一个和爬虫相关的例子——使用 Java 正则表达式提取 URL：

```
    public class TestString {
     /**多次使用的话，不需要重新编译正则表达式，对于频繁调用能提高效率*/
    public static final String patternString1="[^\\s]*((<\\s*[aA]\\s+(href\\
    s*=[^>]+\\s*)>)(.*)</[aA]>).*";
    public static   final String patternString2=".*(<\\s*[aA]\\s+(href\\s*=[
    ^>]+\\s*)>(.*)</[aA]>).*";
    public static final String patternString3=".*href\\s*=\\s*(\"|'|)http://
    .*";
    public static Pattern pattern1 =Pattern.compile(patternString1,Pattern.D
    OTALL);
    public static Pattern pattern2 =Pattern.compile(patternString2,Pattern.D
    OTALL);
    public static Pattern pattern3 =Pattern.compile(patternString3,Pattern.D
    OTALL);
      /**
       * @param args
       */
     public static void main(String[] args) {
     /**测试的数据*/
```

```
  String ss="这是测试<a href=http://www.google.cn>www.google.cn</a>真的是测
试了";
 /**保存提取出来的 url，用 Set 从字面上去重*/
 Set<String> set=new HashSet<String>();
/**解析 URL 并保存在 set 里*/
parseUrl(set,ss);
/**针对解析出来的 URL 做处理*/
System.out.println(replaceHtml(set,ss));

 }
/**给每个 URL 加上 target 属性*/
  public static String replaceHtml(Set<String> set,String var)
{
  String result=null;
         /**最好不要修改参数*/
         result=var;
         Iterator<String> ite=set.iterator();
         while(ite.hasNext())
         {
            String url=ite.next();
            if(url!=null)
            {
                result=result.replaceAll(url,url+"  target=\"_blank\"");

            }
         }
        return result;
    }
    public static void parseUrl(Set<String> set,String var)
    {
        Matcher matcher=null;
        String result=null;
        //假设最短的 a 标签链接为 <a href=http://www.a.cn></a>，则计算它的长度为 28
        if(var!=null  && var.length()>28)
        {
             matcher=pattern3.matcher(var);
             //确定句子里包含有链接
            if(matcher!=null && matcher.matches())
            {
                matcher=pattern1.matcher(var);
                String aString=null;
                String bString=null;

                while(matcher!=null && matcher.find())
                {
                    if(matcher.groupCount()>3)
                    {
```

```
                    bString=matcher.group(matcher.groupCount()-3);
                    //这个 group 包含所有符合正则表达式的字符串
                    aString=matcher.group(matcher.groupCount()-2);
                    //这个 group 包含 URL 的 HTML 标签
                    String url1=matcher.group(matcher.groupCount()-1);
                    //最后一个 group 就是 URL
                    set.add(url1);//将找到的 URL 保存起来
                    bString=bString.replaceAll(aString, "");
                    //去掉找到的 URL 的 HTML 标签
                  }

              }
              if(bString!=null)
             {
                  parseUrl(set,bString);//继续循环提取下一个 url
              }

          }
      }

   }

}
```

正则表达式是搜索引擎开发中的强大利器。本节详细讲述了正则表达式的基本知识和用 Java 开发正则表达式的基础知识。接下来的几节将详细讲述如何在爬虫项目中使用正则表达式。

## 4.2 抽取 HTML 正文

网页展现给用户的主要内容是它的文本。因此，在获得网页源代码时，针对网页抽取出它的特定文本内容，也是爬虫的一个主题。利用正则表达式和一定的抽取工具(比如 HtmlParser)，能够很好地抽取这些内容。本节我们结合 HtmlParser 和上一节讲的正则表达式，来详细讲述抽取 HTML 文本的内容。

### 4.2.1 了解 HtmlParser

HtmlParser 是一个用来解析 HTML 文件的 Java 包，主要用于转换、抽取两个方面。利用 HtmlParser，可以实现以下内容的抽取：

- 文本抽取。作为一些垂直搜索引擎的检索内容放入数据库中。
- 链接抽取。比如可以从当前页面中抽取链接放入 TODO 表中。
- 资源抽取。可以搜集到图像和声音等资源。
- 链接检查。保证链接是有用的。
- 站点检查。可以查看页面不同版本之间的差异(在爬虫过程中，防止重复抓取同一

页面)。

HtmlParser 的转换功能通常用在以下几个方面：

- URL 重写。能够修正页面中的错误链接。
- 广告清除。清除页面中的广告内容和指向广告的链接。
- 将 HTML 页面转化成 XML 页面。
- HTML 页面清理。

下面我们使用图 4.1 来说明 HtmlParser 是如何解析 HTML 页面的。

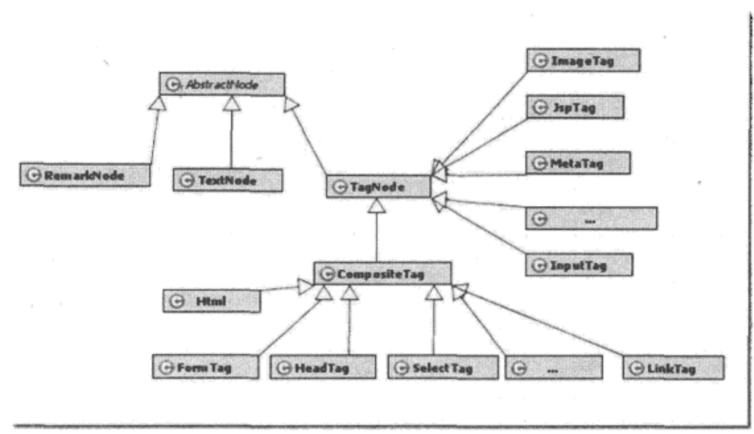



图 4.1　HtmlParser 类图

如图 4.1 所示，HtmlParser 采用了经典的 Composite 模式，通过 RemarkNode、TextNode、TagNode、AbstractNode 和 Tag 来描述 HTML 页面中的各元素。下面介绍 HtmlParser 中重要的类。

### 1. org.htmlparser.Node

Node 接口定义了进行树形结构节点操作的各种典型操作方法，包括：节点到 html 文本、text 文本的方法：toPlainTextString、toHtml。

典型树形结构遍历的方法有：getParent、getChildren、getFirstChild、getLastChild、getPreviousSibling、getNextSibling、getText 等，下面按照功能依次讲解。

获取节点对应的树形结构的顶级节点 Page 对象的方法：getPage

获取节点起始位置的方法：getStartPosition、getEndPosition

遍历节点的方法：accept (NodeVisitor visitor)

Filter 方法：collectInto (NodeList list, NodeFilter filter)

### 2. org.htmlparser.nodes.AbstractNode

AbstractNode 是形成 HTML 树形结构的抽象基类，实现了 Node 接口。

在 HtmlParser 中，Node 分成三类。

- RemarkNode：代表 HTML 中的注释。

- TagNode：标签节点。
- TextNode：文本节点。

这三类节点都继承 AbstractNode。

### 3. org.htmlparser.nodes.TagNode

TagNode 包含了对 HTML 处理的核心的各个类，是所有 TAG 的基类，其中又分为包含其他 TAG 的复合节点 ComositeTag 和不包含其他 TAG 的叶子节点 Tag。

复合节点 CompositeTag 包括：AppletTag, BodyTag, Bullet, BulletList, DefinitionList, DefinitionListBullet, Div, FormTag, FrameSetTag, HeadingTag, HeadTag, Html, LabelTag, LinkTag, ObjectTag, ParagraphTag, ScriptTag, SelectTag, Span, StyleTag, TableColumn, TableHeader, TableRow, TableTag, TextareaTag, TitleTag。

叶子节点 TAG 包括：BaseHrefTag, DoctypeTag, FrameTag, ImageTag, InputTag, JspTag, MetaTag, ProcessingInstructionTag。

HtmlParser 有两种方式可以访问互联网：Visitor 方式和 Filter 方式。其中，采用 Visitor 方式访问 HTML 的代码如下：

```
try {
Parser parser = new Parser();
parser.setURL("http://www.google.com")/;
parser.setEncoding(parser.getEncoding());
NodeVisitor visitor = new NodeVisitor() {
public void visitTag(Tag tag) {
logger.fatal("testVisitorAll()  Tag name is : "
+ tag.getTagName() + " \n Class is : "
+ tag.getClass());
}
};
parser.visitAllNodesWith(visitor);
} catch (ParserException e) {
e.printStackTrace();
}
```

采用 Filter 方式访问 HTML 的代码如下：

```
try {
NodeFilter filter = new NodeClassFilter(LinkTag.class);
Parser parser = new Parser();
parser.setURL("http://www.google.com")/;
parser.setEncoding(parser.getEncoding());
NodeList list = parser.extractAllNodesThatMatch(filter);
for (int i = 0; i < list.size(); i++) {
LinkTag node = (LinkTag) list.elementAt(i);
logger.fatal("testLinkTag() Link is :" + node.extractLink());
}
} catch (Exception e) {
```

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

## 4.2.2 使用正则表达式抽取示例

了解 HtmlParser 的基本知识之后，我们看一看如何把 HtmlParser 应用到网络爬虫中，也就是如何使用 HtmlParser 来抽取我们所需要的内容。

在网络爬虫中，当获得一个页面后，抽取链接是首要的工作，那么如何使用 HtmlParser 来抽取页面中的链接呢？其实非常简单，就是用 Visitor 方式来访问 Html 中的每一个节点，当节点类型是 LinkTag 的时候，保存链接：

```
public void testTagVisitor(String url) {
       Parser parser = null;
       NodeVisitor visitor = null;
       try {
           parser = new Parser(url);
           visitor = new NodeVisitor() {
              @Override
              public void visitTag(Tag tag) {
                  if (tag instanceof LinkTag) {
                     LinkTag link = (LinkTag)tag;//处理 tag
                     String linkString = link.getLink();//获得链接
                     // 对链接做进一步处理
                  }
              }

           };
             parser.visitAllNodesWith(visitor);
       } catch (ParserException e) {
       }

    }
```

结合上一节学习的正则表达式的内容，可以实现结合使用 HtmlParser 和正则表达式的抽取程序。大多数爬虫程序都采用这种方式，尤其是一些垂直搜索引擎，经常会定向抓取网站中的特定频道或者内容。下面就给出了一个利用正则表达式和 HtmlParser 的代码做定向抓取的例子。请大家重点关注 public List extractHtml(Node nodeP) 函数，该函数利用递归的方法定位频道。下面的代码利用了 HtmlParser 的抓取分析功能。

```
public class TestTabDivSerial {
     /**
     * 新行
     */
     private static final String NEWLINE = System.getProperty("line.separator");
     /**
      * 字符串长度
      */
```

```
private static final int NEWLINE_SIZE = NEWLINE.length();
private String url;
private final String oriEncode = "gb2312,utf-8,gbk,iso-8859-1";
private ArrayList htmlContext = new ArrayList();
private String urlEncode;
private int tableNumber;
private int channelNumber;
private int totalNumber;
//URL 正则表达
private String domain;
private String urlDomaiPattern;
private String urlPattern;
private Pattern pattern;
private Pattern patternPost;
public void channelParseProcess() {
    /**提取本站信息的正则表达式**/
    urlDomaiPattern = "(http://[^/]*?" + domain + "/)(.*?)";
    urlPattern = "(http://[^/]*?" + domain + "/[^.]*?).(shtml|html|h
    tm|shtm|php|asp#|asp|cgi|jsp|aspx)";
      pattern = Pattern.compile(urlDomaiPattern,
      Pattern.CASE_INSENSITIVE + Pattern.DOTALL);
      patternPost = Pattern.compile(urlPattern,
      Pattern.CASE_INSENSITIVE + Pattern.DOTALL);
      /**收集表单集合**/
      SplitManager splitManager = (SplitManager) ExtractLinkConsole.co
      ntext.getBean("splitManager");
    urlEncode = dectedEncode(url);
    if (urlEncode == null) {
        return;
    }
  singContext(url);
 Iterator hi = htmlContext.iterator();
if (htmlContext.size() == 0) {
        return;
    }
  totalNumber = htmlContext.size();
//分析表单集合
    while (hi.hasNext()) {
        TableContext tc = (TableContext) hi.next();
       this.totalNumber = tc.getTableRow();
       if ((tc.getTableRow() == this.channelNumber) ||
               (this.channelNumber == -1)) {
           System.out.println("********************表单" +
               tc.getTableRow() + "****************");
               List linkList = tc.getLinkList();
               //如果没有任何连接
           if ((linkList == null) || (linkList.size() == 0)) {
```

```
                continue;
            }
        Iterator hl = linkList.iterator();
        /**分析单个表单**/
            while (hl.hasNext()) {
                LinkTag lt = (LinkTag) hl.next();
                /**过滤非法 link**/
                if (isValidLink(lt.getLink()) == SpiderConstant.OUTD
                OMAINLINKTYPE) {
                    continue;
                }
                if (lt.getLinkText().length() < 8) {
                    continue;
                }
                /**过滤无效 link**/
                if (splitManager.isChannelLink(lt.getLinkText()) !=
                SpiderConstant.COMMONCHANNEL) {
                    continue;
                }
              /**生成 link 的 hashcode**/
                System.out.println("URL:" + lt.getLinkText() + "    " +
                    lt.getLink());
            }
        }
    }
}
/**
*判断是否为有效连接
*/
public int isValidLink(String link) {
    Matcher matcher = pattern.matcher(link);
    while (matcher.find()) {

        int start = matcher.start(2);
        int end = matcher.end(2);
        String postUrl = link.substring(end).trim();
        if ((postUrl.length() == 0) || (postUrl.indexOf(".") < 0)) {
            return SpiderConstant.CHANNELLINKTYPE;
        } else {
            Matcher matcherPost = patternPost.matcher(link);
            if (matcherPost.find()) {
                return SpiderConstant.COMMONLINKTYPE;
            } else {
                return SpiderConstant.OUTDOMAINLINKTYPE;
            }
        }
    }
```

```
        return SpiderConstant.OUTDOMAINLINKTYPE;
    }
    /**
     * 收集 HTML 页面信息
     */
    public void singContext(String url) {
        try {
            Parser parser = new Parser(url);
            parser.setEncoding(urlEncode);
            tableNumber = 0;
            for (NodeIterator e = parser.elements(); e.hasMoreNodes();) {
                Node node = (Node) e.nextNode();
                if (node instanceof Html) {
                    extractHtml(node);
                }
            }
        } catch (Exception e) {
        }
    }
    /**
    * 递归钻取信息
    */
    public List extractHtml(Node nodeP) {
        NodeList nodeList = nodeP.getChildren();
        boolean bl = false;
        if ((nodeList == null) || (nodeList.size() == 0)) {
            return null;
        }
        if ((nodeP instanceof TableTag) || (nodeP instanceof Div)) {
            bl = true;
        }
        ArrayList tableList = new ArrayList();
        try {
            for (NodeIterator e = nodeList.elements(); e.hasMoreNodes();) {
                Node node = (Node) e.nextNode();
                if (node instanceof LinkTag) {
                tableList.add(node);
                } else if (node instanceof ScriptTag ||
                        node instanceof StyleTag || node instanceof SelectTag) {
                } else if (node instanceof TextNode) {
                    if (node.getText().trim().length() > 0) {
                        tableList.add(node);
                    }
                } else {
                    List tempList = extractHtml(node);
                    if ((tempList != null) && (tempList.size() > 0)) {
                        Iterator ti = tempList.iterator();
```

```
                    while (ti.hasNext()) {
                        tableList.add(ti.next());
                    }
                }
            }
        }
    } catch (Exception e) {
    }
    if ((tableList != null) && (tableList.size() > 0)) {
        if (bl) {
            TableContext tc = new TableContext();
            tc.setLinkList(new ArrayList());
            tc.setTextBuffer(new StringBuffer());
            tableNumber++;
            tc.setTableRow(tableNumber);
            Iterator ti = tableList.iterator();
            while (ti.hasNext()) {
                Node node = (Node) ti.next();
                if (node instanceof LinkTag) {
                    tc.getLinkList().add(node);
                } else {
                    tc.getTextBuffer()
                      .append(collapse(node.getText()
                                        .replaceAll(" ", "")));
                }
            }
            htmlContext.add(tc);
            return null;
        } else {
            return tableList;
        }
    }
    return null;
}
/**
 * 去除无效字符
 */
protected String collapse(String string) {
    int chars;
    int length;
    int state;
    char character;
    StringBuffer buffer = new StringBuffer();
    chars = string.length();
    if (0 != chars) {
      length = buffer.length();
      state = ((0 == length) || (buffer.charAt(length - 1) == ' ') ||
```

```
            ((NEWLINE_SIZE <= length) &&
            buffer.substring(length - NEWLINE_SIZE, length).equals(N
            EWLINE))) ? 0 : 1;
        for (int i = 0; i < chars; i++) {
            character = string.charAt(i);
            switch (character) {
            case '\u0020':
            case '\u0009':
            case '\u000C':
            case '\u200B':
            case '\u00a0':
            case '\r':
            case '\n':
                if (0 != state) {
                    state = 1;
                }
                break;
            default:
                if (1 == state) {
                    buffer.append(' ');
                }
                state = 2;
                buffer.append(character);
            }
        }
    }
    return buffer.toString();
}
/**
 * 检测字符级
 */
private String dectedEncode(String url) {
    String[] encodes = oriEncode.split(",");
    for (int i = 0; i < encodes.length; i++) {
        if (dectedCode(url, encodes)) {
            return encodes;
        }
    }
    return null;
}
public boolean dectedCode(String url, String encode) {
    try {
      Parser parser = new Parser(url);
      parser.setEncoding(encode);
      for (NodeIterator e = parser.elements(); e.hasMoreNodes();) {
           Node node = (Node) e.nextNode();
            if (node instanceof Html) {
```

```
                    return true;
                }
            }
        } catch (Exception e) {
        }
        return false;
    }
    public String getDomain() {
        return domain;
    }
    public void setDomain(String domain) {
        this.domain = domain;
    }
    public Pattern getPattern() {
        return pattern;
    }
    public void setPattern(Pattern pattern) {
        this.pattern = pattern;
    }
    public Pattern getPatternPost() {
        return patternPost;
    }
    public void setPatternPost(Pattern patternPost) {
        this.patternPost = patternPost;
    }
    public String getUrlDomaiPattern() {
        return urlDomaiPattern;
    }
    public void setUrlDomaiPattern(String urlDomaiPattern) {
        this.urlDomaiPattern = urlDomaiPattern;
    }
    public String getUrlPattern() {
        return urlPattern;
    }
    public void setUrlPattern(String urlPattern) {
        this.urlPattern = urlPattern;
    }
    public int getChannelNumber() {
        return channelNumber;
    }
    public void setChannelNumber(int channelNumber) {
        this.channelNumber = channelNumber;
    }
    public int getTotalNumber() {
        return totalNumber;
    }
```

```
    public void setTotalNumber(int totalNumber) {
        this.totalNumber = totalNumber;
    }
    public String getUrlEncode() {
        return urlEncode;
    }
    public void setUrlEncode(String urlEncode) {
        this.urlEncode = urlEncode;
    }
    public String getUrl() {
        return url;
    }
    public void setUrl(String url) {
        this.url = url;
    }
```

上面的例子结合 HtmlParser 和正则表达式对网页的内容进行了简单抽取。下一节将详细地讲述如何抽取网页正文。

## 4.3 抽 取 正 文

抽取正文在网络爬虫中具有重要意义。如果不能很好地提取原有文章的内容和样式，那么搜索出来的东西就会惨不忍睹，根本就没有使用价值。

正文抽取有很多方法：配置模板、视觉匹配、关键字识别等。

首先配置模板是不太现实的。因为在网络爬虫应用中，根本不知道会搜索到哪个网站，也根本没有精力去配置模板。

基于视觉效果的分析难度比较大，而且只适用于规范的网站。现在很多网站不规范，广告链接漫天飞。网站常常把最好的位置留给了广告。

那是否有种简单的方法呢？

所有的正文应该有个共同的特点，那就是正文的长度应该超过其他文字组合的长度，很少会有一句话的正文，很少会有长度短于标题的正文，所以这个应该成为一个突破口。

接下来，有一个很重要的问题，那段最长的文本(即可能是正文的文本)在哪里呢？

包含这段文本的标签通常是在一个 TABLE、DIV 或者 ParagraphTag 里。因此，找到那个包含文字最多的 DIV 或者 TABLE，通常就找到了正文。

对于不少 HTML 页面，HTML 元素的长度超过了正文的长度，有时候会混入很多 JavaScript。HtmlParser 经常会将这些元素误认为是正文加以识别，导致很多正文竟然是一段 JavaScript。因此祛除杂质是关键，这里面要把那些 HTML 中常用的标签，以及连接从正文中去除(去除杂质有一个专业的名称，叫网页去噪，我们在第 7 章将详细地介绍网页去噪)。

下面是参照上述方法实现的正文抽取代码：

```
//table 有效性的记录
```

```
public class TableValid {
    private int trnum;
    private int tdnum;
    private int linknum;
    private int textnum;
    private int scriptnum;

    public int getScriptnum() {
        return scriptnum;
    }
    public void setScriptnum(int scriptnum) {
        this.scriptnum = scriptnum;
    }
    public int getLinknum() {
        return linknum;
    }
    public void setLinknum(int linknum) {
        this.linknum = linknum;
    }
    public int getTdnum() {
        return tdnum;
    }
    public void setTdnum(int tdnum) {
        this.tdnum = tdnum;
    }
    public int getTextnum() {
        return textnum;
    }
    public void setTextnum(int textnum) {
        this.textnum = textnum;
    }
    public int getTrnum() {
        return trnum;
    }
    public void setTrnum(int trnum) {
        this.trnum = trnum;
    }
}

//table中的内容
public class TableContext {
    private List linkList;
    private StringBuffer textBuffer;
    private int tableRow;
    private int totalRow;
    private String sign;
```

```
    public String getSign() {
        return sign;
    }
    public void setSign(String sign) {
        this.sign = sign;
    }
    public int getTotalRow() {
        return totalRow;
    }
    public void setTotalRow(int totalRow) {
        this.totalRow = totalRow;
    }
    public int getTableRow() {
        return tableRow;
    }
    public void setTableRow(int tableRow) {
        this.tableRow = tableRow;
    }
    public List getLinkList() {
        return linkList;
    }
    public void setLinkList(List linkList) {
        this.linkList = linkList;
    }
    public StringBuffer getTextBuffer() {
        return textBuffer;
    }
    public void setTextBuffer(StringBuffer textBuffer) {
        this.textBuffer = textBuffer;
    }
}

//column 有效性的记录
public class TableColumnValid {
    int tdNum;
    boolean valid;
    public int getTdNum() {
        return tdNum;
    }
    public void setTdNum(int tdNum) {
        this.tdNum = tdNum;
    }
    public boolean isValid() {
        return valid;
    }
    public void setValid(boolean valid) {
        this.valid = valid;
```

```
    }
}
//页面内容
public class PageContext {
    private StringBuffer textBuffer;
    private int number;
    private Node node;

    public Node getNode() {
        return node;
    }
    public void setNode(Node node) {
        this.node = node;
    }
    public int getNumber() {
        return number;
    }
    public void setNumber(int number) {
        this.number = number;
    }
    public StringBuffer getTextBuffer() {
        return textBuffer;
    }
    public void setTextBuffer(StringBuffer textBuffer) {
        this.textBuffer = textBuffer;
    }
}

//正文抽取主程序
public class ExtractContext {
    protected static final String lineSign = System.getProperty(
            "line.separator");
    protected static final int lineSign_size = lineSign.length();

    /**定义系统上下文**/
    public static final ApplicationContext context = new
    ClassPathXmlApplicationContext(new String[] {
                "newwatch/persistence.xml", "newwatch/biz-util.xml",
                "newwatch/biz-dao.xml"
            });

    /**
    * @param args
    */
    public static void main(String[] args) {
        ExtractContextConsole console = new ExtractContextConsole();
```

```
    ChannelLinkDO c = new ChannelLinkDO();
    c.setEncode("gb2312");
    c.setLink("http://www.qiche.com.cn/files/200712/12016.shtml");
    c.setLinktext("test");
    console.makeContext(c);
}

/**
* 收集HTML页面信息
* @param url
* @param urlEncode
*/
public void makeContext(ChannelLinkDO c) {
    String metakeywords = "<META content={0} name=keywords>";
    String metatitle = "<TITLE>{0}</TITLE>";
    String metadesc = "<META content={0} name=description>";
    String netshap = "<p> 正文快照: 时间{0}</p> ";

    String tempLeate = "<LI class=active><A href=\"{0}\"
    target=_blank>{1}</A></LI>";
    String crop = "<p><A href=\"{0}\" target=_blank>{1}</A></p> ";

    try {
        String siteUrl = getLinkUrl(c.getLink());
        Parser parser = new Parser(c.getLink());
        parser.setEncoding(c.getEncode());
        for (NodeIterator e = parser.elements(); e.hasMoreNodes();) {
            Node node = (Node) e.nextNode();
            if (node instanceof Html) {
                PageContext context = new PageContext();
                context.setNumber(0);
                context.setTextBuffer(new StringBuffer());
                //抓取出内容
                extractHtml(node, context, siteUrl);
                StringBuffer testContext = context.getTextBuffer();
                String srcfilePath = "D:/kuaiso/site/templeate/context.vm";
                String destfilePath = "D:/kuaiso/site/test/test.htm";
                BufferedReader reader = new BufferedReader(new InputStreamReader(
                        new FileInputStream(srcfilePath), "gbk"));
                BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(
                        new FileOutputStream(destfilePath), "gbk"));
                String lineContext = context.getTextBuffer().toString();
                String line;
                while ((line = reader.readLine()) != null) {
                    int start = line.indexOf("#context");
                    if (start >= 0) {
                        String tempCrop = StringUtil.replace(crop, "{0}",
```

```
                c.getLink());
            tempCrop = StringUtil.replace(tempCrop, "{1}",
                    "原文链接: " + c.getLink());
            writer.write(tempCrop + lineSign);
            writer.write(netshap + lineSign);
            writer.write(lineContext + lineSign);
            continue;
        }
        int start1 = line.indexOf("#titledesc");
        if (start1 >= 0) {
            String tempLine = StringUtil.replace(tempLeate,
                    "{0}", "test.htm");
            tempLine = StringUtil.replace(tempLine, "{1}",
                    "标题: " + c.getLinktext());

            writer.write(tempLine + lineSign);
            continue;
        }
        int start2 = line.indexOf("#metatitle");
        if (start2 >= 0) {
            metatitle = StringUtil.replace(metatitle, "{0}",
                    c.getLinktext());
            writer.write(metatitle + lineSign);
            continue;
        }
        int start3 = line.indexOf("#metadesc");
        if (start3 >= 0) {
            metadesc = StringUtil.replace(metadesc, "{0}",
                    c.getLinktext());
            writer.write(metadesc + lineSign);
            continue;
        }
        writer.write(line + lineSign);
    }
    writer.flush();
    writer.close();
    reader.close();
  }
 }
} catch (Exception e) {
    System.out.println(e);
}
}

//从一个字符串中提取出链接
private String getLinkUrl(String link) {
    String urlDomaiPattern = "(http://[^/]*?" + "/)(.*?)";
```

```
    Pattern pattern = Pattern.compile(urlDomaiPattern,
    Pattern.CASE_INSENSITIVE + Pattern.DOTALL);
    Matcher matcher = pattern.matcher(link);
    String url = "";
    while (matcher.find()) {
        int start = matcher.start(1);
        int end = matcher.end(1);
        url = link.substring(start, end - 1).trim();
    }
    return url;
}

/**
* 递归钻取正文信息
* @param nodeP
* @return
*/
protected List extractHtml(Node nodeP, PageContext context, String
                                siteUrl)
    throws Exception {
    NodeList nodeList = nodeP.getChildren();
    boolean bl = false;
    if ((nodeList == null) || (nodeList.size() == 0)) {
        if (nodeP instanceof ParagraphTag) {
            ArrayList tableList = new ArrayList();
            StringBuffer temp = new StringBuffer();
            temp.append("<p style=\"TEXT-INDENT: 2em\">");
            tableList.add(temp);
            temp = new StringBuffer();
            temp.append("</p>").append(lineSign);
            tableList.add(temp);
            return tableList;
        }
        return null;
    }
    if ((nodeP instanceof TableTag) || (nodeP instanceof Div)) {
        bl = true;
    }
    if (nodeP instanceof ParagraphTag) {
        ArrayList tableList = new ArrayList();
        StringBuffer temp = new StringBuffer();
        temp.append("<p style=\"TEXT-INDENT: 2em\">");
        tableList.add(temp);
        extractParagraph(nodeP, siteUrl, tableList);
        temp = new StringBuffer();
        temp.append("</p>").append(lineSign);
        tableList.add(temp);
```

```
    return tableList;
}
ArrayList tableList = new ArrayList();
try {
    for (NodeIterator e = nodeList.elements(); e.hasMoreNodes();) {
        Node node = (Node) e.nextNode();
        if (node instanceof LinkTag) {
            tableList.add(node);
            setLinkImg(node, siteUrl);
        } else if (node instanceof ImageTag) {
            ImageTag img = (ImageTag) node;
            if (img.getImageURL().toLowerCase().indexOf("http://") < 0) {
                img.setImageURL(siteUrl + img.getImageURL());
            } else {
                img.setImageURL(img.getImageURL());
            }
            tableList.add(node);
        } else if (node instanceof ScriptTag ||
                node instanceof StyleTag || node instanceof SelectTag) {
        } else if (node instanceof TextNode) {
            if (node.getText().length() > 0) {
                StringBuffer temp = new StringBuffer();
                String text = collapse(node.getText()
                                .replaceAll(" ", "")
                                .replaceAll("  ", ""));
                temp.append(text.trim());
                tableList.add(temp);
            }
        } else {
            if (node instanceof TableTag || node instanceof Div) {
                TableValid tableValid = new TableValid();
                isValidTable(node, tableValid);
                if (tableValid.getTrnum() > 2) {
                    tableList.add(node);
                    continue;
                }
            }
            List tempList = extractHtml(node, context, siteUrl);
            if ((tempList != null) && (tempList.size() > 0)) {
                Iterator ti = tempList.iterator();
                while (ti.hasNext()) {
                    tableList.add(ti.next());
                }
            }
        }
    }
} catch (Exception e) {
```

```
        return null;
    }
    if ((tableList != null) && (tableList.size() > 0)) {
        if (bl) {
            StringBuffer temp = new StringBuffer();
            Iterator ti = tableList.iterator();
            int wordSize = 0;
            StringBuffer node;
            int status = 0;
            StringBuffer lineStart = new StringBuffer(
                    "<p style=\"TEXT-INDENT: 2em\">");
            StringBuffer lineEnd = new StringBuffer("</p>" + lineSign);
            while (ti.hasNext()) {
                Object k = ti.next();
                if (k instanceof LinkTag) {
                    if (status == 0) {
                        temp.append(lineStart);
                        status = 1;
                    }
                    node = new StringBuffer(((LinkTag) k).toHtml());
                    temp.append(node);
                } else if (k instanceof ImageTag) {
                    if (status == 0) {
                        temp.append(lineStart);
                        status = 1;
                    }
                    node = new StringBuffer(((ImageTag) k).toHtml());
                    temp.append(node);
                } else if (k instanceof TableTag) {
                    if (status == 0) {
                        temp.append(lineStart);
                        status = 1;
                    }

                    node = new StringBuffer(((TableTag) k).toHtml());
                    temp.append(node);
                } else if (k instanceof Div) {
                    if (status == 0) {
                        temp.append(lineStart);
                        status = 1;
                    }
                    node = new StringBuffer(((Div) k).toHtml());
                    temp.append(node);
                } else {
                    node = (StringBuffer) k;
                    if (status == 0) {
                        if (node.indexOf("<p") < 0) {
```

```
                                temp.append(lineStart);
                                temp.append(node);
                                wordSize = wordSize + node.length();
                                status = 1;
                            } else {
                                temp.append(node);
                                status = 1;
                            }
                        } else if (status == 1) {
                            if (node.indexOf("</p") < 0) {
                                if (node.indexOf("<p") < 0) {
                                    temp.append(node);
                                    wordSize = wordSize + node.length();
                                } else {
                                    temp.append(lineEnd);
                                    temp.append(node);
                                    status = 1;
                                }
                            } else {
                                temp.append(node);
                                status = 0;
                            }
                        }
                    }
                }
                if (status == 1) {
                    temp.append(lineEnd);
                }

                if (wordSize > context.getNumber()) {
                    context.setNumber(wordSize);
                    context.setTextBuffer(temp);
                }
                return null;
            } else {
                return tableList;
            }
        }
        return null;
    }

    /**
    * 设置图像连接
    * @param nodeP
    * @param siteUrl
    */
    private void setLinkImg(Node nodeP, String siteUrl) {
```

```
    NodeList nodeList = nodeP.getChildren();
    try {
        for (NodeIterator e = nodeList.elements(); e.hasMoreNodes();) {
            Node node = (Node) e.nextNode();
            if (node instanceof ImageTag) {
                ImageTag img = (ImageTag) node;
                if (img.getImageURL().toLowerCase().indexOf("http://") < 0) {
                    img.setImageURL(siteUrl + img.getImageURL());
                } else {
                    img.setImageURL(img.getImageURL());
                }
            }
        }
    } catch (Exception e) {
        return;
    }
    return;
}
/**
* 钻取段落中的内容
* @param nodeP
* @param siteUrl
* @param tableList
* @return
*/
private List extractParagraph(Node nodeP, String siteUrl, List tableList) {
    NodeList nodeList = nodeP.getChildren();
    if ((nodeList == null) || (nodeList.size() == 0)) {
        if (nodeP instanceof ParagraphTag) {
            StringBuffer temp = new StringBuffer();
            temp.append("<p style=\"TEXT-INDENT: 2em\">");
            tableList.add(temp);
            temp = new StringBuffer();
            temp.append("</p>").append(lineSign);
            tableList.add(temp);
            return tableList;
        }
        return null;
    }
    try {
        for (NodeIterator e = nodeList.elements(); e.hasMoreNodes();) {
            Node node = (Node) e.nextNode();
            if (node instanceof ScriptTag || node instanceof StyleTag ||
                    node instanceof SelectTag) {
            } else if (node instanceof LinkTag) {
                tableList.add(node);
                setLinkImg(node, siteUrl);
```

```
} else if (node instanceof ImageTag) {
    ImageTag img = (ImageTag) node;
    if (img.getImageURL().toLowerCase().indexOf("http://") < 0) {
        img.setImageURL(siteUrl + img.getImageURL());
    } else {
        img.setImageURL(img.getImageURL());
    }
    tableList.add(node);
} else if (node instanceof TextNode) {
    if (node.getText().trim().length() > 0) {
        String text = collapse(node.getText()
                            .replaceAll(" ", "")
                            .replaceAll("  ", ""));
        StringBuffer temp = new StringBuffer();
        temp.append(text);
        tableList.add(temp);
    }
} else if (node instanceof Span) {
    StringBuffer spanWord = new StringBuffer();
    getSpanWord(node, spanWord);
    if ((spanWord != null) && (spanWord.length() > 0)) {
        String text = collapse(spanWord.toString()
                                .replaceAll(" ", "")
                                .replaceAll("  ", ""));
        StringBuffer temp = new StringBuffer();
        temp.append(text);
        tableList.add(temp);
    }
} else if (node instanceof TagNode) {
    String tag = node.toHtml();
    if (tag.length() <= 10) {
        tag = tag.toLowerCase();
        if ((tag.indexOf("strong") >= 0) ||
                (tag.indexOf("b") >= 0)) {
            StringBuffer temp = new StringBuffer();
            temp.append(tag);
            tableList.add(temp);
        }
    } else {
        if (node instanceof TableTag || node instanceof Div) {
            TableValid tableValid = new TableValid();
            isValidTable(node, tableValid);
            if (tableValid.getTrnum() > 2) {
                tableList.add(node);
                continue;
            }
        }
```

```
                extractParagraph(node, siteUrl, tableList);
            }
        }
    }
    } catch (Exception e) {
        return null;
    }
    return tableList;
}

protected void getSpanWord(Node nodeP, StringBuffer spanWord) {
    NodeList nodeList = nodeP.getChildren();
    try {
        for (NodeIterator e = nodeList.elements(); e.hasMoreNodes();) {
            Node node = (Node) e.nextNode();
            if (node instanceof ScriptTag || node instanceof StyleTag ||
                    node instanceof SelectTag) {
            } else if (node instanceof TextNode) {
                spanWord.append(node.getText());
            } else if (node instanceof Span) {
                getSpanWord(node, spanWord);
            } else if (node instanceof ParagraphTag) {
                getSpanWord(node, spanWord);
            } else if (node instanceof TagNode) {
                String tag = node.toHtml().toLowerCase();
                if (tag.length() <= 10) {
                    if ((tag.indexOf("strong") >= 0) ||
                            (tag.indexOf("b") >= 0)) {
                        spanWord.append(tag);
                    }
                }
            }
        }
    } catch (Exception e) {
    }
    return;
}

/**
* 判断 TABLE 是否是表单
* @param nodeP
* @return
*/
private void isValidTable(Node nodeP, TableValid tableValid) {
    NodeList nodeList = nodeP.getChildren();
    /**如果该表单没有子节点则返回**/
    if ((nodeList == null) || (nodeList.size() == 0)) {
```

```
            return;
        }
        try {
            for (NodeIterator e = nodeList.elements(); e.hasMoreNodes();) {
                Node node = (Node) e.nextNode();
                /**如果子节点本身也是表单则返回**/
                if (node instanceof TableTag || node instanceof Div) {
                    return;
                } else if (node instanceof ScriptTag ||
                     node instanceof StyleTag || node instanceof SelectTag) {
                    return;
                } else if (node instanceof TableColumn) {
                    return;
                } else if (node instanceof TableRow) {
                    TableColumnValid tcValid = new TableColumnValid();
                    tcValid.setValid(true);
                    findTD(node, tcValid);
                    if (tcValid.isValid()) {
                        if (tcValid.getTdNum() < 2) {
                            if (tableValid.getTdnum() > 0) {
                                return;
                            } else {
                                continue;
                            }
                        } else {
                            if (tableValid.getTdnum() == 0) {
                                tableValid.setTdnum(tcValid.getTdNum());
                                tableValid.setTrnum(tableValid.getTrnum() + 1);
                            } else {
                                if (tableValid.getTdnum() == tcValid.getTdNum()) {
                                    tableValid.setTrnum(tableValid.getTrnum() +
                                        1);
                                } else {
                                    return;
                                }
                            }
                        }
                    }
                } else {
                    isValidTable(node, tableValid);
                }
            }
        } catch (Exception e) {
            return;
        }
        return;
    }
```

```
/**
* 判断是否有效TR
* @param nodeP
* @param TcValid
* @return
*/
private void findTD(Node nodeP, TableColumnValid tcValid) {
    NodeList nodeList = nodeP.getChildren();
    /**如果该表单没有子节点则返回**/
    if ((nodeList == null) || (nodeList.size() == 0)) {
        return;
    }
    try {
        for (NodeIterator e = nodeList.elements(); e.hasMoreNodes();) {
            Node node = (Node) e.nextNode();
            if (node instanceof TableTag || node instanceof Div ||
                    node instanceof TableRow ||
                    node instanceof TableHeader) {
                tcValid.setValid(false);
                return;
            } else if (node instanceof ScriptTag ||
                 node instanceof StyleTag || node instanceof SelectTag) {
                tcValid.setValid(false);
                return;
            } else if (node instanceof TableColumn) {
                tcValid.setTdNum(tcValid.getTdNum() + 1);
            } else {
                findTD(node, tcValid);
            }
        }
    } catch (Exception e) {
        tcValid.setValid(false);
        return;
    }
    return;
}

protected String collapse(String string) {
    int chars;
    int length;
    int state;
    char character;
    StringBuffer buffer = new StringBuffer();
    chars = string.length();
    if (0 != chars) {
        length = buffer.length();
```

```
            state = ((0 == length) || (buffer.charAt(length - 1) == ' ') ||
                ((lineSign_size <= length) &&
                buffer.substring(length - lineSign_size,
                length).equals(lineSign)))
                ? 0 : 1;
            for (int i = 0; i < chars; i++) {
                character = string.charAt(i);
                switch (character) {
                case '\u0020':
                case '\u0009':
                case '\u000C':
                case '\u200B':
                case '\u00a0':
                case '\r':
                case '\n':
                    if (0 != state) {
                        state = 1;
                    }
                    break;
                default:
                    if (1 == state) {
                        buffer.append(' ');
                    }
                    state = 2;
                    buffer.append(character);
                }
            }
        }
        return buffer.toString();
    }
}
```

本章简单地介绍了一些抽取正文的方法，像其他一些方法，比如标记窗算法、VIPS 网页分割算法、基于双层决策的算法等，在抽取正文中都有较好的效果。有兴趣的读者可以查看相关资料。

## 4.4 从 JavaScript 中抽取信息

在网页中，充斥着大量的 JavaScript 代码。通常情况下，JavaScript 代码对我们没有什么作用。但是，有的 JavaScript 代码中，变量也定义了许多正文的内容，比如文字、链接等。因此，有时候从 JavaScript 代码中提取有用的信息也是非常必要的。

下面是两个 JavaScript 代码的例子。

例 1：在浏览器中打开网页 http://work.cat898.com/list.asp ? boardid = 1 (凯迪社区-猫眼看人-帖子列表)，右键单击查看源代码，得到其中某个帖子的源代码：

```
< Script Language = JavaScript > document . write ( dvbbs - top2ic - list
( TempStr , '1590974 ',' 1 ',' 储户在银行被运钞警卫开枪打死。 ',' Susan 3 3 3 ','
gun $17167040 $2007 - 4 - 12 20 :23 :16 $必须枪毙! $ $53304 $1590974 $1
',' face1. gif ',' 68004 ',' 2007 - 4 - 9 13 :04 :17 ',' 1252 ',' 88272 ','
0 ',' 2007 -4 - 12 20 :21 :15 ',' 0 ',' 0 ',' 0 ',' 0 ',' 0 ',' 5 ', '1' ) ) ;
hiddentr( 'follow1733888' ) ; < / Script >
```

例 2：在浏览器中打开网页 http://club. book. so2hu. com/ r - zz0090 - 59022 - 0 - 5 - 0. html，右键单击查看源代码，在源代码中查询帖子中的某个句子，比如“青春不再，红颜不再”，将发现其在源代码中不存在。

由以上的例子可知，传统的 HTML 标记识别方法无法完成动态页面主体内容及其所含超链接网络地址的提取工作。然而在互联网页面总数中，动态网页占 50%以上，并以 JavaScript 语言编写的动态页面最为流行。

本节就讲述如何从 JavaScript 中抽取信息。

### 4.4.1 JavaScript 抽取方法

要想从 JavaScript 中抽取信息，基本思路是解析 JavaScript 代码，从而获得动态页面主体内容与超链接网络地址。这种解决办法的工作针对性更强，执行效率更快，实现过程中使用的资源空间更少。正因如此，选取合适的脚本解释引擎，构建动态页面核心脚本所需要的解析环境，已经成为当前动态页面解析工作的主流技术。

Rhino是一个由Java实现的JavaScript语言解析引擎，Rhino 的主要功能是管理脚本执行时的运行环境。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，调用相应的API 建立脚本语言的内置对象。例如，下面是一段执行JavaScript代码的Rhino程序：

```
public class RhinoTest
{
    public static void main(String[] args)
    {
        /* 创建一个 JavaScript 的上下文环境，用来存储 JavaScript 的环境信息 */
        Context cx = Context.enter();
        try {
            /* 初始化 JavaScript 标准对象(例如 Object, Function, Array 等) */
            Scriptable scope = cx.initStandardObjects();

            /* 读取一个 js 文件 */
            String script = "";
            File file = null;
            if(args.length > 0)
            {
                file = new File(args[0]);
                // 如果有参数，则读入第一个参数中指定的 js 文件
            }
```

```
            else
            {
                file = new File("script.js"); // 如果没有参数，则读入script.js
            }
            BufferedReader in = new BufferedReader(new FileReader(file));
            String s = "";
            while((s = in.readLine()) != null)
            {
                script += s + "\n";
            }

            /* 执行代码 */
            cx.evaluateString(scope, script, "[" + file.getName() + "]", 1, null);
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
        finally
        {
            Context.exit();
        }
    }
}
```

测试的 script.js 文件为：

```
var lang = new JavaImporter();
lang.importPackage(java.lang);
lang.System.out.println("Hello, World!~");

var swing = new JavaImporter();
swing.importPackage(javax.swing);
with(swing)
{
    var frame = new JFrame("Swing Application");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    var button = new JButton("Hello, Rhino!");
    button.addActionListener(function() {
        lang.System.out.println("This is a text from swing button");
        });

    frame.add(button);
    frame.pack();
    frame.setResizable(false);
    frame.setVisible(true);
}
```

Rhino 会分析 JavaScript 语法并加以执行，执行后的结果可以通过它提供的 API 接口取得。对于需要使用自定义脚本的应用，Rhino 不愧是一个好的选择。有如下原因：

(1) 相对于自定义脚本而言，自定义解释器 Rhino 可以解释现成的 JavaScript 语法并加以执行，为我们节省了大量的时间和精力。

(2) JavaScript 本身比较成熟稳定。

(3) Rhino 支持 JavaScript 1.6 标准。

(4) 允许直接在 Java 代码中嵌入 Script。

(5) J2SE 6 将 Rhino 作为默认的 JavaScript 引擎。

### 4.4.2 JavaScript 抽取示例

前面介绍了开源的 JavaScript 解析引擎，下面我们来看看如何利用这个强大的工具实现 JavaScript 抽取功能。

由于脚本解释引擎 Rhino 无法识别 JavaScript 脚本片段中包含的 HTML DOM，在把动态页面脚本片段传递给 Rhino 前，需要先对脚本片段中的 HTML DOM 实现本地创建，给出每个 HTML DOM 的方法和属性描述，这是构建脚本运行环境最为核心的工作。

JavaScript 动态页面脚本片段主要有以下三种存储方式：

(1) 最为常见的是位于< script > 和< / script > 标签之间。

(2) 使用“JavaScript :”脚本描述方式，置于某个 HTML 标记中。

(3) 位于< script>标签的 SRC/ARCHIVE 属性指向的外部 js 文件中。

对于前两种可以使用传统的 HTML 标记识别方法实现脚本片段提取，分别匹配<script > 和</script > 标记对和“JavaScript:”字符串。对于后一种，要先获得网页源文件内 SRC/ARCHIVE 属性中以.js 结束的字符串，再结合当前网页的基地址构造该 js 文件的网络绝对地址，最后对其单独实现网络获取，即从动态页面所属主机上获取该 js 文件。

在 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，可以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 JavaScript 语言内置 String 类型)是一个静态网页源文件。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 和文本信息等方式直接嵌入 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 在执行脚本片断中的 Window.open()与 Document.write()时，实际上是分别调用 Rhino 代码中与上述两方法同名的 Java 函数，执行函数体中关于函数行为的描述。

在完成脚本片断提取和 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了。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 对象方法的 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 对象方法的 write 参数所指向的初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，就能够提取得到的静态网页源文件中的所有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，如图 4.2 所示。

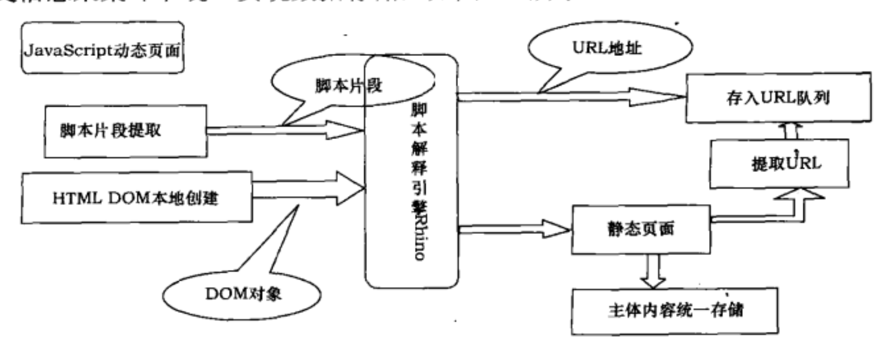



图 4.2　基于 Rhino 实现 JavaScript 动态页面解析的过程

比如，针对本节开始时讲的示例，解析前帖子在源代码中的存在形式：

```
< Script Language = JavaScript > document . write ( dvbbs - top2ic - list
( TempStr ,'1590974', '1', '储户在银行被运钞警卫开枪打死。','Susan 3 3 3','gun
$17167040 $2007 - 4 - 12 20 :23 :16 $必须枪毙！ $ $53304 $1590974 $1','face1.
gif','68004','2007 - 4 - 9 13 :04 :17','1252','88272','0','2007 -4 - 12
20 :21 :15','0','0','0','0','0','5','1') ) ; hiddentr('follow1733888') ; <
/ Script >
```

解析后帖子在源代码中的存在形式：

```
< a href ="dispbbs. asp ? boradID = 1 &ID = 159097 &page =1"tilte = "《储
户在银行被运钞警卫开枪打死。》& # 13 ; & # 10 ;作者: Susan 3 3 3 & # 13 ; & # 10 ;
发表于:2007 - 4 - 9 13 :04 :17 & # 13 & # 10 ;最后发帖:必须枪毙!. . . "target
= "- blank"> 储户在银行被运钞警卫开枪打死。< / a >
```

下面是 4.4 节例 2 的实验结果：解析前，在源代码中找不到帖子的主题内容，其源代码如下：

```
< script language = javascript > document . write ( body -59022) ; < / script >
< font color = 73A2A0 > < script laguange =javascript > document . write (body1
- 59022) ; < / script >
```

解析后，在源代码中将得到如下内容(只摘取了小部分)：

```
< br >   ;   ;   ;时间从指尖一分一秒悄悄流逝。< br >   ;   ;
  ;青春不再，红颜不再。< br >  ;   ;   ;有人在说话，空气喧嚣了起
```

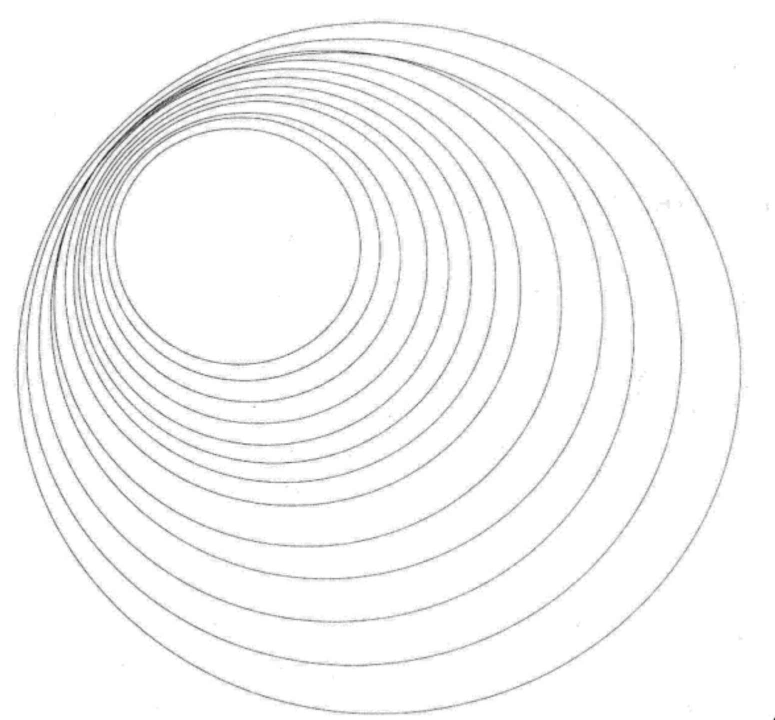
来，有人喊收衣服，有人挪花盆，风来了。好大风，吹得人如此舒服。如此迷人醉人的夜晚。<br>
  ;   ;   ; 风还在肆意地吹。

本节详细地讲述了在 JavaScript 中抽取信息的过程。JavaScript 抽取是正文抽取的一个难点，也是重点。但是有了 Rhino 的帮助，就会变得简单方便。

## 4.5 本章小结

本章是 Web 内容抽取的第一章，主要讲述了一些与 Web 信息抽取相关的基础知识，例如正则表达式的基础知识，以及使用 Java 处理正则表达式的方法。接着，介绍了 HtmlParser 的内容。并且演示了如何抽取 HTML 页面。最后，还介绍了 JavaScript 的抽取方案。

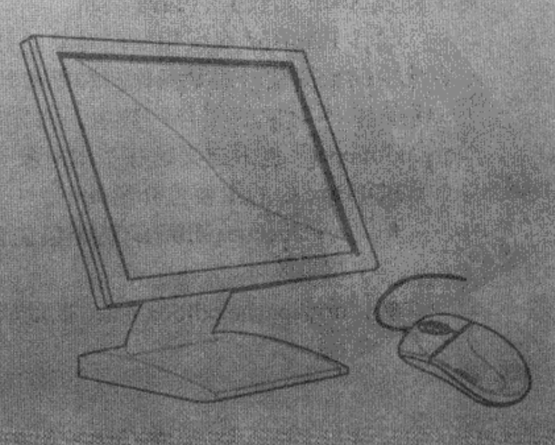
本节的内容都比较基础，就不再给大家列出参考资料了。



# 第5章

# 非HTML正文抽取

为了方便查找抓取下来的文件，需要把文件转换成半结构化的数据形式，抽取标题、时间等关键要素。本章介绍一些常用文件的特点与处理方法。



## 5.1 抽取 PDF 文件

便携文档格式(PDF)是 Adobe 公司在 1993 年开发的电子文件格式。这种文件以独立于操作系统和硬件的方式来表示二维文档格式。在许多操作系统下都有阅读器可以浏览和打印 PDF 格式的文件，例如 Adobe 公司的 Acrobat Reader。PDF 文件格式的完整说明可以从 http://www.adobe.com/devnet/pdf/pdf_reference.html 网页下载。

PDF 文件格式的前身是 PostScript，属于描述型语言(Adobe 公司推出的一套打印标准)。PDF 文件格式中的指令集和 PostScript 中的指令类似，都是操作数在前，操作符在后，形如：

```
操作数 1 操作数 2  操作数 3  ...  操作符
```

PDF 规范定义了 5 种图形对象：

- 路径对象是由直线、矩形和曲线组成的任意形状。路径对象可以用一个或多个操作符结束，用来指明是仅对路径对象边框着色还是全部填充等。
- 文本对象由一个或多个字符串数组组成。和路径对象类似，可以用一个或多个操作符来修饰文本对象，指明对文本对象边框着色还是全部填充等。
- 外部对象是在内容流外部定义的对象。
- 内部图形对象是在内容流内部直接表示小图形的一种方法。
- 底纹对象是一个几何形状，它的颜色是和形状内的位置相关的一个函数。

不同的操作符应用于不同的对象。例如，和文本对象操作符相关的操作符有 BT 和 ET。BT 表示开始一个文本对象，而 ET 表示结束一个文本对象。

有时候会遇到这种情况，就是想复制 PDF 文件中的文字，可是经常发现不能复制，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。PDFBox 就是专门用来解析 PDF 文件的 Java 项目。除 PDFBox 外，还有一个主要用于创建 PDF 文件的 iText 项目(http://itextpdf.com/)可供参考。

### 5.1.1 学习 PDFBox

PDFBox 是一个开源软件，可以从 http://pdfbox.apache.org 下载。PDFBox 项目中有两个子项目：FontBox 和 JempBox。FontBox 是一个处理 PDF 字体的 Java 类库。JempBox 是一个处理 XMP 元数据的 Java 类库。

设计 PDFBox 时采用面向对象的方式来描述 PDF 文档。从逻辑角度来看，PDF 文档内容流中的数据被看成是操作符和操作数组成的序列。从实现角度来看，PDF 文档的数据是一系列基本对象的集合：数组、布尔型、字典、数字、字符串和二进制流。PDFBox 在 org.pdfbox.cos 包中定义这些基本对象类型。

PDFBox 中的主要包介绍如下。

- org.apache.pdfbox：这个包包含一些可执行的类，例如，ExtractText 类从 PDF 文档提取文本。
- org.apache.pdfbox.ant：批量从 PDF 文档提取文本。

- org.apache.pdfbox.cos：组成 PDF 文档的底层对象，所有的类都继承自 COSBase 抽象类。包括数组(COSArray)、布尔型(COSBoolean)、名称(COSName)、字符串(COSString)、字典(COSDictionary)、数字(COSNumber)、对象(COSObject)的实现。COSNumber 的子类包括：浮点数(COSFloat)和整数(COSInteger)。COSDocument 是 PDF 文档的内存表示。
- org.apache.pdfbox.encoding：用于文档的所有编码的实现，也包括 CMap 转换的实现。
- org.apache.pdfbox.encryption：用于文档的加密算法处理。
- org.apache.pdfbox.examples：示范如何使用 PDFBox 中的 API 的例子程序。其中，org.apache.pdfbox.examples.fdf 展示如何使用 PDF 特征。org.apache.pdfbox.examples.pdmodel 展示如何使用 pdmodel 包中的类；org.apache.pdfbox.examples.persistence 展示如何使用 PDFBox 的持久性特征；org.apache.pdfbox.examples.signature 展示如何获得 PDF 的数字签名。
- org.apache.pdfbox.filter：用于处理 PDF 文档中的字节流。Filter 接口包括压缩和解压缩两个方法。
- org.apache.pdfbox.pdfparser：解析 PDF 文档和文档中的对象。
- org.apache.pdfbox.pdfviewer：图形化显示 PDF 文档。
- org.apache.pdfbox.pdmodel：创建和操作 PDF 文档的高层的 API。其中，PDDocument 封装了 COSDocument，提供 PDF 文档的内存表示。PDPage 表示 PDF 文档中的单个页面。PDF 文件中可以包括交互式表单，PDDocumentCatalog 表示文档中的表单。
- org.apache.pdfbox.searchengine.lucene：将 pdfbox 和 Lucene 集成起来。LucenePDFDocument 类把单个 PDF 文件中需要搜索的信息提取成 Lucene 中的 Document 对象。

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。

PDFBox 中包含一个把 PDF 文件输出到可视化窗口的应用程序 PageDrawer。例如，Resources/PageDrawer.properties 中有一行：

```
BT=org.apache.pdfbox.util.operator.BeginText
```

表示 BT 操作的处理类是 org.apache.pdfbox.util.operator.BeginText。

PageDrawer 有完整的对于文字显示方面的处理功能，例如为文本确定显示位置和颜色。但是，如果只是提取文件的标题以及相关的要素信息用于搜索，并不需要把 PDF 文件完整地显示出来，就不要直接使用 PageDrawer，而要创建一个继承 PDFStreamEngine 的 TextPageDrawer 类。TextPageDrawer 模仿了 PageDrawer 的实现，但是简化了一些与 Path 和 Line 相关的操作符的处理，只把文本的位置和颜色等信息提取出来。

```
public TextPageDrawer() throws IOException
{
    Properties properties =
ResourceLoader.loadProperties( "Resources/PageDrawer.properties", true );
```

```
    if( properties == null )
    {
        throw new NullPointerException( "properties cannot be null" );
    }
    try
    {
        Iterator keys = properties.keySet().iterator();
        while( keys.hasNext() )
        {
            String operator = (String)keys.next();

            if(operators.contains(operator))
            {
                //忽略的操作符
                continue;
            }

            //取得操作符对应的类名
            String operatorClass = properties.getProperty( operator );
            //创建一个类的实例
            OperatorProcessor op =
(OperatorProcessor)Class.forName( operatorClass ).newInstance();
            //注册类的实例到对应的操作符
            registerOperatorProcessor(operator, op);
        }
    }
    catch( Exception e )
    {
        throw new WrappedIOException( e );
    }
}
```

可以通过扩展 org.pdfbox.util.PDFTextStripper 类，重写 processTextPosition 方法来遍历 PDF 文件中的 TextPosition 对象。下面的代码可以简单地提取最大字体的文字。

```
        float currentFontSize = text.getFontSizeInPt();
        if(currentFontSize < bigestFontSize) //字体不一致，则不是标题文字
        {
         consistent = false;
        }
        else if (currentFontSize > bigestFontSize) //字体最大，是标题文字
        {
         titleGuess = text.getCharacter();
             bigestFontSize = currentFontSize;
         consistent = true;
        }
        else if(currentFontSize == bigestFontSize
             && 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();
    }
```

文字的颜色信息并不能从文本对象直接得到，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()
      );
}
```

使用 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 内部默认支持一些中文字体，但 PDFBox 内部却没有对 GBK-EUC-H 字体的支持。CMap 文件中定义了 CID 数字和字符编码之间的对应关系。PDFReader 会查找 cmap 路径下的 CMap 文件中的“GBK-EUC-H”，但是在 PDFBox 中却没法查找。可以在 PDFReader 的文档属性中看到 PDF 文件编码用到的字体，如图 5.1 所示。

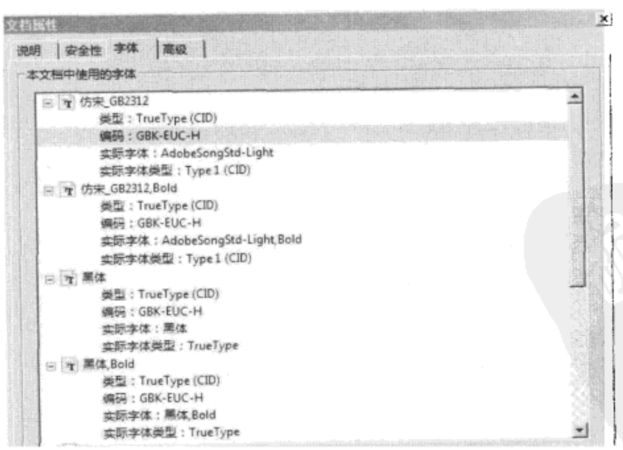



图 5.1　文档属性窗口中显示的字体

在 PDFBox 0.8 版本中，会查找操作系统本身的字体。

```
java.awt.Font[] allFonts =
java.awt.GraphicsEnvironment.getLocalGraphicsEnvironment().getAllFonts()
;
int numberOfFonts = allFonts.length;
for (int i=0;i<numberOfFonts;i++)
{
          java.awt.Font font = allFonts[i];
          System.out.println(font);
}
```

但是 GBK-EUC-H 是重新排序后生成的字体，不是操作系统直接支持的全真(true type)字体，所以这种方法仍然找不 GBK-EUC-H 字体。GBK-EUC-H 字体在 Adobe 公司的 cmap 字体文件中定义。因为涉及 Adobe 公司的专利问题，PDFBox 0.8 以后的版本不再包括 Adobe 公司的 cmap 字体文件。使用编译工具 Ant 编译该项目会自动从 Adobe 公司下载 cmap 字体文件，这样就避免了因为直接包含字体文件所带来的法律问题。

### 5.1.2 使用 PDFBox 抽取示例

使用 PDFTextStripper 提取文本的代码如下：

```
//输入 pdf 文件，返回提取结果
public static String getText(File file) throws Exception {
    // 是否按位置排序
    boolean sort = false;
    // 开始提取页数
    int startPage = 1;
    // 结束提取页数
    int endPage = 10;
    // 内存中存储的 PDF 文档
    PDDocument document = null;
    try {
        try {
            // 当作一个 URL 来装载文件
            document = PDDocument.load(file);
        } catch (MalformedURLException e) {
        }
        // 用 PDFTextStripper 来提取文本
        PDFTextStripper stripper = new PDFTextStripper();
        // 设置文本是否按位置排序
        stripper.setSortByPosition(sort);
        // 设置起始页
        stripper.setStartPage(startPage);
        // 设置结束页
        stripper.setEndPage(endPage);

        // 返回文本
        return stripper.getText(document);
    } catch (Exception e) {
```

```
            return "";
        } finally {
            if (document != null) {
                document.close();
            }
        }
    }
```

### 5.1.3　提取 PDF 文件标题

很多 PDF 文件命名不规范，从文件命名上无法看出文件标题。因此，搜索引擎需要想办法提取出 PDF 文件标题。例如，从 Google 搜索“filetype:pdf”可以看到 Google 提取的 PDF 文件标题。

设计提取标题的流程如图 5.2 所示。

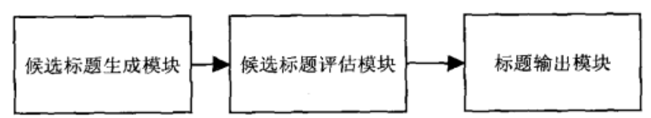



图 5.2　PDF 文件标题提取流程图

- 候选标题生成模块：首先将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，从原子节点开始构造；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，文字在同一行，并且字体、字号大小、颜色都一致，则视为不可拆分，然后根据字体、字号大小、颜色、位置等信息再次合并文字，在文档结构树中给出几个可能的候选标题。
- 候选标题评估模块：对于每个候选标题，按照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，可以按照词频-逆文档频率法(TF*IDF)选取重要性较高的词作为关键词，然后根据关键词给出每个候选标题的可能性权重。还可以将候选标题与首页中的其他文字进行比较，看候选标题相对其他文字的代表度或者说是相对其他文字的可替代性，也就是说候选标题对其他文字的覆盖度。从通顺性与意义完整性方面考虑，可以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出一个 HMM 模型。
- 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。

如果 PDF 元数据中已经存储了文档标题，可以用元数据中的标题作为文档候选标题：

```
PDDocumentInformation info = document.getDocumentInformation();
this.title = info.getTitle();
```

但是很多 PDF 文件中的元数据并没有存储任何内容，而且有些元数据中存储显示的标题并不一定准确。

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。

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。

候选标题可以单独组成一个段落，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间隔。

(2) 根据最大垂直间隔拆分 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。

在 PDFBox 0.8 版本中，可以通过 TextPosition 对象的 getFontSizeInPt 方法返回文字字体大小。可以使用 SVN 的客户端工具 TortoiseSVN 从 SVN 导出 PDFBox 0.8 开发版本。

政府公文中往往包含一些主题词，我们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，利用标题语料库可以保证提取出来的标题的完整性。

文件名本身也可能是标题。

```
public static String getFirstName(String fileName)
{
    int pos = fileName.indexOf('.');
    return fileName.substring(0,pos);
}
```

### 5.1.4 处理 PDF 格式的公文

经常需要对抓取下来的红头文件做结构化信息提取。例如，提取“发文机关”和“收文机关”以及“文号”和“发文时间”等。

例如图 5.3 所示文件的“发文单位”是“广东省交通厅”，“文号”是“粤交运函〔2009〕1356 号”，“接收单位”是“各地级以上市交通局(委)”。

广 东 省 交 通 厅

粤交运函〔2009〕1356 号

关于做好市际客运班线调整工作的通知

各地级以上市交通局（委）：

为及早应对轨道交通对我省道路客运市场的影响，优化运力资源配置，提高运输组织水平与运力利用效率，促进我省道路运输的持续健康发展，根据《关于印发<关于促进我省道路运输业又好又快发展的实施意见>的通知》（粤交运〔2009〕190 号）要求，厅决定开展市际客运班线节点调整工作，现将有关事项通知如下：

图 5.3 PDF 公文图

判断一段文字是否为发文单位的代码如下：

```
//输入文本和文本的字体颜色，如果是发文单位则返回true
public static boolean isProSendGovUnit(String s,Color tcolor)
{
        double pro = 0;//判断是否为发文单位的概率
        //如果是红色则增加概率
        if(tcolor.equals(Color.RED))
        {
            pro+=0.2;
        }

        //如果以文件结尾则增加概率
        if(s.endsWith("文件"))
        {
            pro+=0.4;
        }

        int numBlank = 0;   //计算空格数量
        int numGovUnit = 0; //计算单位特征词数量
        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());
        }
        //发文单位的概率大于一个阈值，则认为是发文单位
        if(pro > 0.6)
            return true;
        return false;
    }
```

判断是否为收文机关的代码如下：

```
//输入文本和文本的字体颜色，如果是收文单位则返回 true
public static boolean isProReceiGovUnit(String s,Color tcolor)
{
        //如果不是黑色则不是收文机关
        if(!tcolor.equals(Color.BLACK))
        {
            return false;
        }

        float pro =0 ;
        //如果以冒号结尾则可能是收文机关
        if(s.endsWith(": ") || s.endsWith(":"))
        {
            pro += 0.5;
        }

        for(int i=0; i<s.length(); i++)
        {
            char c = s.charAt(i);
            //如果包含某些标点或者关键字则可能是收文机关
            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，粤交运函〔2009〕1356 号)的代码如下：

```
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'))
            {}//如果匹配上指定数字或空格则跳过
            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;
}
```

# 5.2 抽取 Office 文档

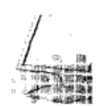
用 Apache 的 POI 项目来处理 Office 系列文档是最流行的方式。http://npoi.codeplex.com/是 POI 项目的.net 移植版本。除此以外，http://code.google.com/p/text-mining/包含了另外一个 Word 解析包。

## 5.2.1 学习 POI

Apache 的 POI 项目(http://poi.apache.org)可以用于在 Windows 或 Linux 平台下提取 Word 文档。POI 项目的目标是创建和维护操作各种基于 OOXML 和 OLE2 的文件格式的 Java API。可以使用 POI 来读写 Excel、Word 或 PowerPoint 文件。大多数 Office 文件都是 OLE2 格式的，例如：XLS、DOC 和 PPT 以及基于 MFC 序列化 API 的文件格式。POIFS 子项目用 Java 实现了对 OLE2 文件系统的读写功能。HPSF 子项目实现了读取 OLE2 文档属性功能。

Office 的 OpenXML 格式是 Microsoft Office 2007/2008 中的基于 XML 文件格式的新标准，包括以 XLSX、DOCX 和 PPTX 为扩展名的文件。POI 项目使用 openXML4J 提供底层接口 API 来支持开放打包约定(OPC)。开放打包约定定义了一种通过标准 Zip 文件存储应用程序数据及其相关资源的结构化方法。

POI 项目通过 HSMF 子项目支持 Outlook，通过 HDGF 子项目支持 Visio，通过 HPBF 子项目支持 Publisher。



### 5.2.2　使用 POI 抽取 Word 示例

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。基本的内容提取方法如下：

```
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();
}
```

一个 Word 文档包含一个或者多个章节(Section)，每个章节下面包含一个或者多个段落(Paragraph)，每个段落(Paragrah)下面包含一个或者多个字符串(CharacterRun)，如图 5.4 所示。

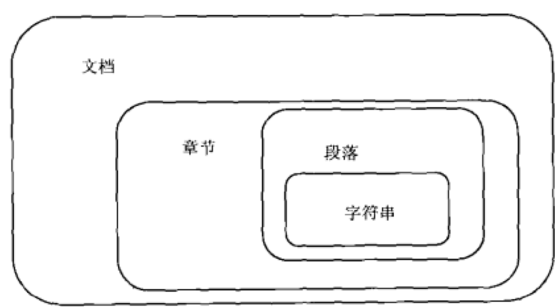



图 5.4　Word 文档结构图

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 的返回值为 0 表示左对齐，为 1 表示居中对齐，为 2 表示右对齐，为 3

表示两端对齐。

可以通过 CharacterRun 对象取得文字内容、字号大小、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
```

提取标题的流程与 PDF 文件的提取过程类似，但是提取候选标题大多依赖于所能提取出的特征，例如，Word 中的段落特征容易获得，但是很难得到 PDF 文本所在的段落信息，所以与 PDF 文件处理略有不同。提取候选标题的实现代码如下：

```
public static ArrayList<TitleInfo> getCandidates(String fileName,
        StringBuffer content) throws Exception {
    ArrayList<TitleInfo> titles = new ArrayList<TitleInfo>();
    InputStream is = new FileInputStream(fileName);
    HWPFDocument doc = new HWPFDocument(is);
    Range r = doc.getRange();
    Section s = r.getSection(0);

    int maxFontSize = 0;
    for (int i = 0; i < s.numParagraphs(); i++)
    {
        maxFontSize = Math.max(s.getParagraph(i).getCharacterRun(0)
                .getFontSize(), maxFontSize);
    }

    int position = 0;
    for (int i = 0; i < s.numParagraphs(); i++) {
        Paragraph para = s.getParagraph(i);
        int maxFont = para.getCharacterRun(0).getFontSize();
        int justification = para.getJustification();

        if (justification == 1 || maxFont == maxFontSize)
        {
            //居中或字体最大的
            String title = "";

            for (int j = 0; j < para.numCharacterRuns(); j++) {
                CharacterRun run = para.getCharacterRun(j);

                title = title + run.text();
                maxFont = Math.max(maxFont, run.getFontSize());
                if (j == para.numCharacterRuns() - 1)
                    titles.add(new TitleInfo(title, run.getColor(),
                            maxFont, position, run.getFontName()));
            }
        } else //主体内容部分
        {
```

```
            for (int j = 0; j < para.numCharacterRuns(); j++) {
                content.append(para.getCharacterRun(j).text().trim() + " ");
            }
        }
        position ++;
    }
    is.close();

    return titles;
}
```

### 5.2.3 使用 POI 抽取 PPT 示例

PPT 文件由一个或多个幻灯片(Slide)组成。调用 Apache 的 POI 提取 PPT 中的文本的代码如下：

```
public static String readDoc(InputStream is) throws IOException{
    //创建 PowerPointExtractor
    PowerPointExtractor extractor=new PowerPointExtractor(is);
    //对 PPT 文件进行提取
    return extractor.getText();
}
```

把第一张幻灯片的标题作为标题：

```
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();
```

### 5.2.4 使用 POI 抽取 Excel 示例

Excel 文件由一个工作簿(Workbook)组成。工作簿由一个或多个有名称的工作表(sheet)组成。每个工作表又包含多个单元格(cell)。除了 POI 项目外，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：

```
public static String readDoc(InputStream is) throws IOException{
        HSSFWorkbook wb = new HSSFWorkbook(new POIFSFileSystem(is));
        ExcelExtractor extractor = new ExcelExtractor(wb);
        extractor.setFormulasNotResults(true);
        extractor.setIncludeSheetNames(false);
        return extractor.getText();
}
```

为了提取 Excel 文件的标题，首先从每个工作表寻找最可能的标题，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，定义的代码如下，其中包含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;
    }
}
```

标题类的属性包括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;
    }
}
```

通过遍历工作表中的每个字符型单元格来估计每个工作表的标题：

```
private TitleInf 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();
```

```
        Iterator<HSSFCell> citer = row.cellIterator();//每行再按列遍历

        while(citer.hasNext())
        {
            HSSFCell cell = citer.next();
            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
            {
                //取得单元格内的文本
                String cellString =  cell.toString().trim();
                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);
}
```

估计整个 Excel 文件的标题：

```
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>();
        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;
    }
```

## 5.3 抽取 RTF

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：{\rtf1 foobar}。用写字板打开这个文件，显示：foobar。

### 5.3.1　开源 RTF 文件解析器

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)，不过这个项目是由 C#语言实现的，下节介绍如何用 Java 语言实现一个 RTF 文件解析器。

### 5.3.2　实现一个 RTF 文件解析器

RTF 文件解析器要满足如下设计目标：

- 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 5.5 所示的这个开放式架构。

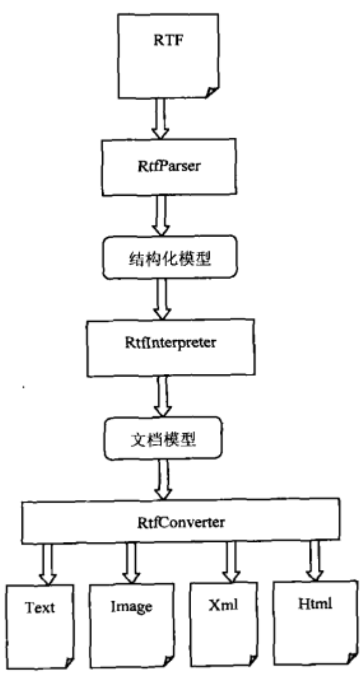



图 5.5　RTF 转换程序总体结构图

RtfParser 类用于完成实际的数据解析。除了识别标签(tag)，也处理字符编码和 Unicode 支持。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.Write((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形式的 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。

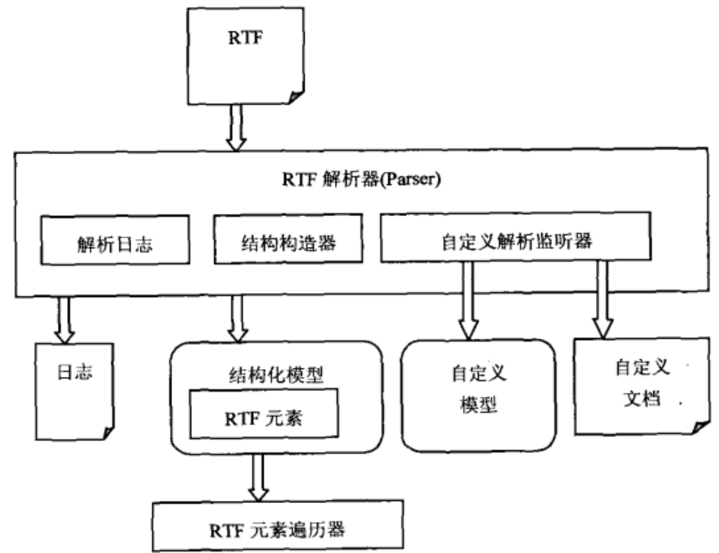



图 5.6　RTF 解析器结构图

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，如标题、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，如线、段落、节、页。
- 特殊字符，如制表符、段落开始/结束、下划线、空格、圆点、引号、连字符。

● 图片。

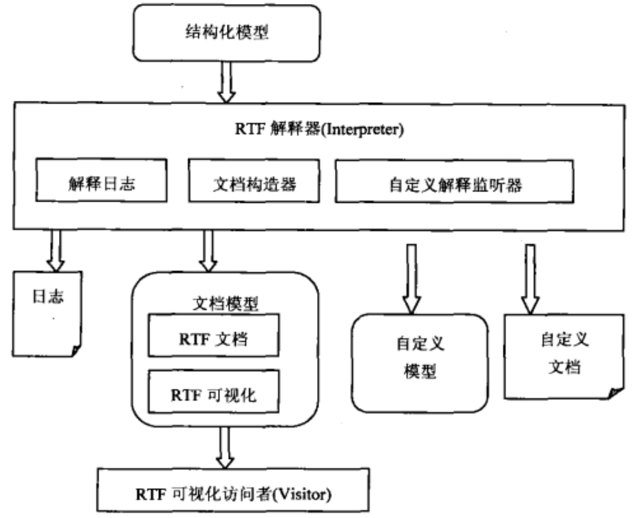



**图 5.7　RTF 解释器结构图**

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) {
            System.out.println("Text: "
                    + ((IRtfVisualText) visual).getText());
        } else if (visualKind == RtfVisualKind.Break) {
            System.out.println("Tag: "
                    + ((IRtfVisualBreak)visual).
                      getBreakKind().toString());
        } else if (visualKind == RtfVisualKind.Special) {
            System.out.println("Text: "
                    + ((IRtfVisualSpecialChar) visual).getCharKind()
                            .toString());
        } else if (visualKind == RtfVisualKind.Image) {
            IRtfVisualImage image = (IRtfVisualImage) visual;
            System.out.println("Image: " + image.getFormat().toString()
                    + " " + image.getWidth() + "x" + image.getHeight());
        }
    }
}
```

### 5.3.3 解析 RTF 示例

下面调用我们自己实现的 RTF 文件格式解析器：

```
URL u = new URL("d:/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
}
```

提取标题的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);
    }
}
```

取得候选标题的部分实现代码如下：

```
public static ArrayList<TitleInfo> getCandidates(String fileName,
    StringBuffer content) throws Exception {
    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
{
    try {
        maxFontSize = Math.max(((IRtfVisualText) (rtfv)).getFormat()
                .getFontSize(), maxFontSize);
    } catch (Exception e) {
    }
}

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，除了字号大小
        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
            try {// Special、Image 类无法强制转换成 Text
                oneRow += ((IRtfVisualText) (rtfv)).getText();
            } catch (Exception e) {
            }
            i++;
            rtfv = rtfVisuals.get(i);
        }

        //公文属性判断
```

```
            if (isProSendGovUnit(oneRow, tc)
                    || isProReceiGovUnit(oneRow, tc) ||
                    isFileNum(oneRow)) {
                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 // 字号大小一致
                && ti.position + 1 == tj.position // 上下连贯
                && titles.get(i).text.length() > 1
                && titles.get(j).text.length() > 1)// 字号大小一样
        {
            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;// 去除废词

            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;
}
// 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;
        }
    }
}
```

最后简单地获取最大分值的标题：

```
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;
```

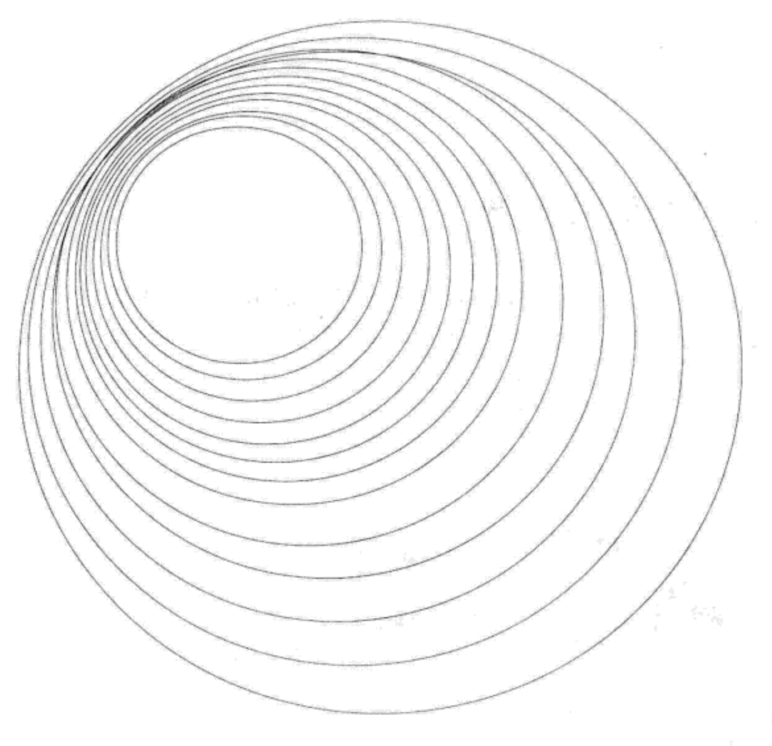
```
            bestTitle = t.text;
        }
    }
    return bestTitle;
}
```

## 5.4　本 章 小 结

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。各种文档格式的处理方式参见表 5.1。

表 5.1　各种文档格式与对应的解析包

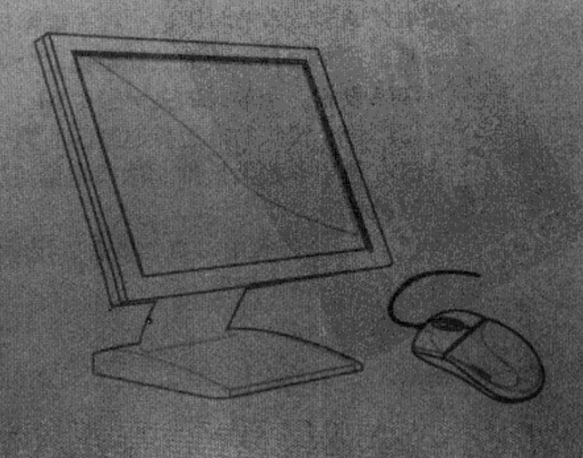
| 格　式 | 解 析 包 |
|---|---|
| Microsoft Office OLE2 Compound Document Format (Excel, Word, PowerPoint, Visio,Outlook) | Apache POI |
| Microsoft Office 2007 OOXML | Apache POI |
| Adobe Portable Document Format (PDF) | PDFBox |
| Rich Text Format (RTF) | RtfParser |
| 英文文本 | ICU4J |
| HTML | NekoHTML |
| 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.* |



# 第6章

## 多媒体抽取

在网络爬虫中，视频信息和音频信息的抽取也是比较重要的内容，很多垂直搜索网站都专门做视频和音频信息的抽取工作。因此，视频和音频的抽取就成为这些垂直搜索引擎的重要技术手段。本章我们将详细讲述如何抽取视频和音频内容。



# 6.1 抽取视频

视频的内容一般分为四部分：帧、镜头、情节和节目。帧的抽取，尤其是关键帧的抽取，在视频检索中占有重要地位。例如，在 Google 的视频检索结果中，都有关键帧的播放，得以让用户先浏览一下内容，如图 6.1 所示。

图 6.1 视频关键帧

镜头是由若干静态画面组成的一段连续的动态影像流程，是电视语言的基本表意单元和叙事单元。它既有两维平面表现三维立体的空间特性，又有影像连续运动的时间特性。简单点说，镜头就是由一系列连续的帧组成的，其中可能包括关键帧。

情节是叙事性文艺作品中人物的生活和斗争的演变过程。由一组以上能显示人物和人物、人物和环境之间的错综复杂关系的具体事件和矛盾冲突所构成，用以展示人物性格和表现主题。简单点说，情节就是由一系列的镜头组成，并且能够描述一定事件的过程。

节目由一系列情节组成，是一个完整的视频内容。

## 6.1.1 抽取视频关键帧

视频检索中最重要的步骤就是抽取视频关键帧，如图 6.1 所示，视频关键帧可以作为检索结果的 URL 指向关键帧所在的镜头、情节、节目等。当你点击关键帧图片时，就可以通过这个 URL 链接到相关的内容。因此，这个关键帧要能非常好地描述它所链接的视频内容。

关键帧提取策略有很多种，比如可以使用动态规划策略，基于视觉模型的自适应关键帧提取策略等。本节将讲述一种关键帧提取策略——基于镜头边界系数的关键帧提取策略。

基于镜头边界系数的关键帧提取分为 3 个步骤：

(1) 设置一个最大的关键帧数 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。

以上 3 个步骤中，第二步需要找出镜头的边界，第三步需要了解非极大值抑制法。下面详细描述这两个问题。

找出镜头边界的方法可以根据镜头边界系数来确定。具体的方法有以下三种：

- 基于帧差的镜头边界检测方法。
- 基于模型的镜头边界检测方法。
- 基于学习的镜头边界检测方法。

这三种方法中，最简单的是基于帧差的边界检测算法。帧差，就是帧的视觉特征之间的差值，它可以表示各帧之间的连续程度。不同的镜头，若在镜头内部，各帧之间比较连续，则帧差就比较小。而在不同的镜头之间，各帧不连续，帧差就比较大。而镜头的边界，可以采用比较第 n 帧和第 n+1 帧的帧差大小来决定。

### 6.1.2 Java 视频处理框架

采用 Java 媒体处理框架(JMF)可以编写出功能强大的多媒体程序，却不用关心底层复杂的实现细节。JMF API 的使用相对比较简单，但是几乎能够满足所有多媒体编程的需求。本节将介绍如何用很少的代码编写出多媒体程序。

JMF 目前的最新版本是 2.1，Sun 公司通过它向 Java 中引入处理多媒体的能力。下面是 JMF 所支持功能的概述：

- 可以在 Java Applet 和应用程序中播放各种媒体文件，例如 AU、AVI、MIDI、MPEG、QuickTime 和 WAV 等文件。
- 可以播放从互联网上下载的媒体流。
- 可以利用麦克风和摄像机一类的设备截取音频和视频，并保存成多媒体文件。
- 处理多媒体文件，转换文件格式。
- 向互联网上传音频和视频数据流。
- 在互联网上播放音频和视频数据。

为了更好地说明 JMF 的结构，让我们用立体声音响做一个简单的比喻。当 CD 机播放 CD 唱片的时候，CD 唱片向系统提供音乐信号。这些数据是在录音棚中用麦克风和其他类似的设备录制的。CD 播放机将音乐信号传送到系统的音箱。在这个例子中，麦克风就是一个音频截取设备，CD 唱片是数据源，而音箱是输出设备。

JMF 的结构和立体声音响系统非常相似，它由以下几个设备组成。

#### 1. 数据源

就像 CD 唱片中保存歌曲一样，数据源中包含了媒体数据流。在 JMF 中，DataSource 对象就是数据源，它可以是一个多媒体文件，也可以是从互联网上下载的数据流。

#### 2. 截取设备

截取设备指的是可以截取到音频或视频数据的硬件，如麦克风、摄像机等。截取到的数据可以被送入 Player 对象中进行处理。

#### 3. 播放器

在 JMF 中对应播放器的接口是 Player。Player 对象将音频/视频数据流作为输入，然后将数据流输出到音箱或屏幕上。Player 对象有多种状态，JMF 中定义了 Player 的 6 种状态，

在正常情况下，Player 对象需要经历每个状态，然后才能播放多媒体。下面是对这些状态的说明。

- Unrealized：在这种状态下，Player 对象已经被实例化，但是并不知道它需要播放的多媒体的任何信息。
- Realizing：调用 realize()方法时，Player 对象的状态从 Unrealized 转变为 Realizing。在这种状态下，Player 对象正在确定它需要占用哪些资源。
- Realized：在这种状态下，Player 对象已经确定了它需要哪些资源，并且也知道了需要播放的多媒体的类型。
- Prefetching：调用 prefecth()方法时，Player 对象的状态从 Realized 变为 Prefetching。在该状态下的 Player 对象正在为播放多媒体做一些准备工作，其中包括加载多媒体数据，获得需要独占的资源等。这个过程被称为预取(Prefetch)。
- Prefetched：Player 对象完成预取操作后就进入该状态。
- Started：调用 start()方法后，Player 对象就进入该状态并播放多媒体。

#### 4. 处理器

Processor 接口是一种播放器，在 JMF API 中，Processor 接口继承了 Player 接口。Processor 对象除了支持 Player 对象支持的所有功能外，还可以控制对于输入的多媒体数据流进行何种处理以及通过数据源向其他的 Player 对象或 Processor 对象输出数据。

除了在播放器中提到的 6 种状态外，Processor 对象还包括两种新的状态，这两种状态在 Unrealized 状态之后，Realizing 状态之前。

- Configuring：调用 configure()方法后，Processor 对象进入该状态。在该状态下，Processor 对象连接到数据源并获取输入数据的格式信息。
- Configured：当完成数据源连接，获得输入数据格式的信息后，Processor 对象就处于 Configured 状态。

#### 5. 数据格式

Format 对象中保存了多媒体的格式信息。该对象本身没有记录多媒体编码的相关信息，但是它保存了编码的名称。Format 的子类包括 AudioFormat 和 VideoFormat 类，ViedeoFomat 又有 6 个子类：H261Format、H263Format、IndexedColorFormat、JPEGFormat、RGBFormat 和 YUVFormat 类。

#### 6. 管理器

JMF 提供了 4 种管理器。

- Manager：Manager 相当于两个类之间的接口。例如要播放一个 DataSource 对象，可以通过使用 Manager 对象创建一个 Player 对象来完成。使用 Manager 对象可以创建 Player、Processor、DataSource 和 DataSink 对象。
- PackageManager：该管理器中保存了 JMF 类注册信息。
- CaptureDeviceManager：该管理器中保存了截取设备的注册信息。
- PlugInManager：该管理器中保存了 JMF 插件的注册信息。

使用 JMF 进行编程以前，需要先安装 JMF。同时在硬件上也有一些要求。由于本文的代码是在 Windows XP 下编写和测试的，因此文章中提到的操作系统需要的软件都是与 Windows 有关的。虽然 Java 是跨平台的，但 JMF 是个例外——并不是所有的平台上都实现了 JMF。

下面示例利用 JMF 技术，在一个 Applet 中直接抓取摄像头的影像：

```
import java.awt.BorderLayout;
import java.awt.Choice;
import java.awt.Component;
import java.util.Vector;
// JMF 相关的类
import javax.media.CaptureDeviceInfo;
import javax.media.CaptureDeviceManager;
import javax.media.Format;
import javax.media.Manager;
import javax.media.MediaLocator;
import javax.media.Player;
import javax.media.format.VideoFormat;
import javax.swing.JPanel;
import javax.swing.JApplet;

public class VApplet extends JApplet {
    private JPanel jContentPane = null;
    private Choice choice = null;
    public VApplet() {
        super();
    }

    public void init() {
        this.setSize(320, 240);
        this.setContentPane(getJContentPane());
        this.setName("VApplet");
}
    // 取系统所有可采集的硬件设备列表
    private CaptureDeviceInfo[] getDevices() {
        Vector devices = CaptureDeviceManager.getDeviceList(null);
        CaptureDeviceInfo[] info = new CaptureDeviceInfo[devices.size()];
        for (int i = 0; i < devices.size(); i++) {
            info [i] = (CaptureDeviceInfo) devices.get(i);
        }
        return info;
}
    // 从已知设备中取所有视频设备的列表
    private CaptureDeviceInfo[] getVideoDevices() {
        CaptureDeviceInfo[] info = getDevices();
        CaptureDeviceInfo[] videoDevInfo;
        Vector vc = new Vector();
```

```
            for (int i = 0; i < info.length; i++) {
            // 取设备支持的格式，如果有一个是视频格式，则认为此设备为视频设备
            Format[] fmt = info[i].getFormats();
                for (int j = 0; j < fmt.length; j++) {
                    if (fmt[j] instanceof VideoFormat) {
                        vc.add(info[i]);
                    }
                    break;
                }
            }
            videoDevInfo = new CaptureDeviceInfo[vc.size()];
            for (int i = 0; i < vc.size(); i++) {
                videoDevInfo[i] = (CaptureDeviceInfo) vc.get(i);
            }
            return videoDevInfo;
        }

        private JPanel getJContentPane() {
            if (jContentPane == null) {
                BorderLayout borderLayout = new BorderLayout();
                jContentPane = new JPanel();
                jContentPane.setLayout(borderLayout);
                MediaLocator ml = null;
                Player player = null;
                try {
                    // 这里只有一个视频设备，直接取第一个
                    // 取得当前设备的 MediaLocator
                    ml = getVideoDevices()[0].getLocator();
                    // 用已经取得的 MediaLocator 得到一个 Player
                    player = Manager.createRealizedPlayer(ml);
                    player.start();
                    // 取得 Player 的 AWT Component
                    Component comp = player.getVisualComponent();
                    // 如果是音频设备，这个方法将返回 null，这里要再判断一次
                    if (comp != null) {
                        // 将 Component 加到窗体
                        jContentPane.add(comp, BorderLayout.EAST);
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
            return jContentPane;
        }
    }
```

### 6.1.3 Java 视频抽取示例

上面讲述了 JMF 的基本原理与抓取影像的示例，这一节，我们使用 JMF 进行视频抽取，也就是抽取视频中所有的帧。关键代码如下：

```
import java.io.*;
import java.awt.*;
import javax.media.*;
import javax.media.control.*;
import javax.media.format.*;
import javax.media.protocol.*;
import java.awt.image.*;
import com.sun.media.codec.video.jpeg.NativeEncoder;
import com.sun.image.codec.jpeg.*;

public class vid2jpg extends Frame implements ControllerListener
{
    Processor p;
    Object waitObj = new Object();
    boolean stateOK = true;
    DataSourceHandler handler;
    imgPanel currPanel;int imgWidth;int imgHeight;
    DirectColorModel dcm = new DirectColorModel(32, 0x00FF0000, 0x0000FF00,
    0x000000FF);
    MemoryImageSource sourceImage;Image outputImage;
    String sep = System.getProperty("file.separator");
    NativeEncoder e;
    int[] outvid;
    int startFr = 1;int endFr = 1000;int countFr = 0;
    boolean sunjava=true;

    public static void main(String[] args)
    {
        if(args.length == 0)
        {
            System.out.println("No media address.");
            new vid2jpg("file:testcam04.avi"); // or alternative "vfw://0"
        }
        else
        {
            String path = args[0].trim();
            System.out.println(path);
            new vid2jpg(path);
        }
    }

    public vid2jpg(String path)
```

```
{
    MediaLocator ml;String args = path;

    if((ml = new MediaLocator(args)) == null)
    {
        System.out.println("Cannot build media locator from: " + args);
    }

    if(!open(ml))
    {
        System.out.println("Failed to open media source");
    }
}

/**
* 根据传入的参数，创建处理器并开始处理
*/
private boolean open(MediaLocator ml)
{
    System.out.println("Create processor for: " + ml);

    try
    {
        p = Manager.createProcessor(ml);
    }
    catch (Exception e)
    {
        System.out.println("Failed to create a processor from the given
        media source: " + e);
        return false;
    }

    p.addControllerListener(this);

    p.configure();
    if(!waitForState(p.Configured))
    {
        System.out.println("Failed to configure the processor.");
        return false;
    }

    p.setContentDescriptor(new ContentDescriptor(ContentDescriptor.RAW));

    TrackControl tc[] = p.getTrackControls();
    if(tc == null)
    {
```

```
    System.out.println("Failed to obtain track controls from the
    processor.");
    return false;
}

TrackControl videoTrack = null;
for(int i = 0; i < tc.length; i++)
{
    if(tc[i].getFormat() instanceof VideoFormat)
    {
        tc[i].setFormat(new RGBFormat(null, -1, Format.byteArray,
        -1.0F, 24, 3, 2, 1));
        videoTrack = tc[i];
    }
    else
    tc[i].setEnabled(false);
}
if(videoTrack == null)
{
    System.out.println("The input media does not contain a video
    track.");
    return false;
}
System.out.println("Video format: " + videoTrack.getFormat());

p.realize();
if(!waitForState(p.Realized))
{
    System.out.println("Failed to realize the processor.");
    return false;
}

// 获取数据源
DataSource ods = p.getDataOutput();
handler = new DataSourceHandler();
try
{
    handler.setSource(ods);
}
catch(IncompatibleSourceException e)
{
    System.out.println("Cannot handle the output DataSource from the
    processor: " + ods);
    return false;
}

setLayout(new FlowLayout(FlowLayout.LEFT));
```

```
        currPanel = new imgPanel(new Dimension(imgWidth,imgHeight));
        add(currPanel);
        pack();
        setVisible(true);

        handler.start();

        // 预取
        p.prefetch();

        if(!waitForState(p.Prefetched))
        {
            System.out.println("Failed to prefetch the processor.");
            return false;
        }

        // 处理器开始工作
        p.start();

        return true;
    }

    /**
    * 设置图像大小
    */
    private void imageProfile(VideoFormat vidFormat)
    {
        System.out.println("Push Format "+vidFormat);
        Dimension d = (vidFormat).getSize();
        System.out.println("Video frame size: "+ d.width+"x"+d.height);
        imgWidth=d.width;
        imgHeight=d.height;
    }

    private void useFrameData(Buffer inBuffer)
    {
        countFr++;
        if(countFr<startFr || countFr>endFr)return;

        try
        {
            printDataInfo(inBuffer);

            if(inBuffer.getData()!=null)
            {
                if(outvid==null)outvid = new int[imgWidth*imgHeight];
                outdataBuffer(outvid,(byte[])inBuffer.getData());
```

```
                setImage(outvid);

                String paddedname = "00000000000000000000"
                +inBuffer.getTimeStamp();
                String sizedname = paddedname.substring
                (paddedname.length()-20);

                if(sunjava)
                {
                    saveJpeg(outputImage,"image_"+sizedname+".jpg");

                }
                else
                {
                    if(e==null)initJpeg((RGBFormat)inBuffer.getFormat());
                    byte[] b = fetchJpeg(inBuffer);
                    String filename = "image_"+sizedname+".jpg";
                    makeFile(filename, b);
                }
            }
        }
        catch(Exception e){System.out.println(e);}
    }

    /**
    * 关闭
    */
    public void tidyClose()
    {
        handler.close();
        p.close();
        if(e!=null)e.close();
        //dispose();    // frame
        System.out.println("Sources closed");
    }

    /**
    * 绘制图像
    */
    private void setImage(int[] outpix)
    {
        if(sourceImage==null)sourceImage = new MemoryImageSource(imgWidth,
        imgHeight, dcm, outpix, 0, imgWidth);
        outputImage = createImage(sourceImage);
        currPanel.setImage(outputImage);
    }
```

```
/**
* 在处理器未进入状态前阻塞
*/
private boolean waitForState(int state)
{
    synchronized(waitObj)
    {
        try
        {
            while(p.getState() < state && stateOK)
            waitObj.wait();
        }
        catch (Exception e)
        {
        }
    }
    return stateOK;
}

/**
* 监听器
*/
public void controllerUpdate(ControllerEvent evt)
{
    if(evt instanceof ConfigureCompleteEvent ||evt instanceof
    RealizeCompleteEvent || evt instanceof PrefetchCompleteEvent)
    {
        synchronized(waitObj)
        {
            stateOK = true;
            waitObj.notifyAll();
        }
    }
    else
    if(evt instanceof ResourceUnavailableEvent)
    {
        synchronized(waitObj)
        {
            stateOK = false;
            waitObj.notifyAll();
        }
    }
    else
    if(evt instanceof EndOfMediaEvent || evt instanceof StopAtTimeEvent)
    {
        tidyClose();
    }
```

```
    }

    /**
    * 打印
    */
    private void printDataInfo(Buffer buffer)
    {
        System.out.println(" Time stamp: " + buffer.getTimeStamp());
        System.out.println(" Time: " +
        (buffer.getTimeStamp()/10000000)/100f+"secs");
        System.out.println(" Sequence #: " + buffer.getSequenceNumber());
        System.out.println(" Data length: " + buffer.getLength());
        System.out.println(" Key Frame: " +
        (buffer.getFlags()==Buffer.FLAG_KEY_FRAME)+"
        "+buffer.getFlags());
    }

    /**
    * 把缓冲内的数据转化为像素
    */
    public void outdataBuffer(int[] outpix, byte[] inData) // could use
JavaRGBConverter
    {
        boolean flip=false;
        {
            int srcPtr = 0;
            int dstPtr = 0;
            int dstInc = 0;
            if(flip)
            {
                dstPtr = imgWidth * (imgHeight - 1);
                dstInc = -2 * imgWidth;
            }

            for(int y = 0; y < imgHeight; y++)
            {
                for(int x = 0; x < imgWidth; x++)
                {
                    byte red = inData[srcPtr + 2];
                    byte green = inData[srcPtr + 1];
                    byte blue = inData[srcPtr];

                    int pixel = (red & 0xff) << 16 | (green & 0xff) << 8 |
                    (blue & 0xff) << 0;
                    outpix[dstPtr] = pixel;
                    srcPtr += 3;
                    dstPtr += 1;
```

```
                }
                dstPtr += dstInc;
            }
        }
        Thread.yield();
    }

    /**
    * JPEG 编码
    */
    public void saveJpeg(Image img, String filename)
    {
        BufferedImage bi = new BufferedImage(img.getWidth(null),
        img.getHeight(null), BufferedImage.TYPE_INT_RGB);
        Graphics g = bi.getGraphics();
        g.drawImage(img, 0, 0, this);

        BufferedOutputStream fw=null;
        try
        {
            fw = new BufferedOutputStream(new
            FileOutputStream("images"+sep+filename));
        }
        catch(IOException e ){System.out.println("makeFile "+e);}

        JPEGImageEncoder encoder = JPEGCodec.createJPEGEncoder(fw);
        JPEGEncodeParam param = encoder.getDefaultJPEGEncodeParam(bi);
        param.setQuality(0.6f,false);
        encoder.setJPEGEncodeParam(param);

        try
        {
            encoder.encode(bi);
            fw.close();

            System.out.println(" *** Created file "+filename);
        }
        catch (java.io.IOException io)
        {
            System.out.println("IOException");
        }
    }

    /**
    * 初始化 JPEG 编码
    */
```

```
private void initJpeg(RGBFormat vfin) throws Exception
{
    float val=0.6F;

    int widpx=imgWidth;
    int hgtpx=imgHeight;

    if(widpx % 8 != 0 || hgtpx % 8 != 0)
    {
        System.out.println("Width = "+imgWidth+" "+"Height =
        "+imgHeight);
        throw new Exception("Image sizes not /8");
    }

    VideoFormat vfout = new VideoFormat("jpeg", new
    Dimension(widpx,hgtpx), widpx * hgtpx * 3, Format.byteArray, -1F);

    e = new NativeEncoder();
    e.setInputFormat(vfin);
    e.setOutputFormat(vfout);
    e.open();

    Control cs[] = (Control[])e.getControls();
    for (int i = 0; i < cs.length; i++)
    {
        if (cs[i] instanceof QualityControl)
        {
            QualityControl qc = (QualityControl)cs[i];
            qc.setQuality(val);
            break;
        }
    }
}

/**
* 从缓冲中获得 JPEG 数据
*/
private byte[] fetchJpeg(Buffer inBuffer) throws Exception
{
    Buffer outBuffer=new Buffer();
    int result = e.process(inBuffer, outBuffer);
    int lengthF = outBuffer.getLength();
    byte[] b = new byte[lengthF];
    System.arraycopy(outBuffer.getData(), 0, b, 0, lengthF);
    return b;
}
```

```
/**
* 保存
*/
public void makeFile(String filename, byte[] b)
{
    BufferedOutputStream fw=null;
    try
    {
        fw = new BufferedOutputStream(new
        FileOutputStream("images"+sep+filename));
        fw.write(b, 0, b.length);fw.close();

        System.out.println(" *** Created file "+filename);
    }
    catch(IOException e ){System.out.println("makeFile "+e);}
}

/***************************************************
* 内部类
***************************************************/

/**
* 数据源处理类，用于读取数据源并且展示获得的每帧的信息
*/
class DataSourceHandler implements BufferTransferHandler
{
    DataSource source;
    PullBufferStream pullStrms[] = null;
    PushBufferStream pushStrms[] = null;
    Buffer readBuffer;

    /**
    *设置数据源
    */
    private void setSource(DataSource source) throws
    IncompatibleSourceException
    {
        if(source instanceof PushBufferDataSource)
        {
            pushStrms = ((PushBufferDataSource) source).getStreams();

            pushStrms[0].setTransferHandler(this);

            imageProfile((VideoFormat)pushStrms[0].getFormat());
        }
```

```
    else
    if(source instanceof PullBufferDataSource)
    {
        System.out.println("PullBufferDataSource!");

        throw new IncompatibleSourceException();
    }

    this.source = source;
    readBuffer = new Buffer();
}

/**
* 数据转换
*/
public void transferData(PushBufferStream stream)
{
    try
    {
        stream.read(readBuffer);
    }
    catch(Exception e)
    {
        System.out.println(e);
        return;
    }

    Buffer inBuffer = (Buffer)(readBuffer.clone());

    if(readBuffer.isEOM())
    {
        System.out.println("End of stream");
        return;
    }

    useFrameData(inBuffer);
}

public void start()
{
    try{source.start();}catch(Exception e){System.out.println(e);}
}

public void stop()
{
    try{source.stop();}catch(Exception e){System.out.println(e);}
```

```
    }

    public void close(){stop();}

    public Object[] getControls()
    {
        return new Object[0];
    }

    public Object getControl(String name)
    {
        return null;
    }
}

/**
* Panel 扩展
*/
class imgPanel extends Panel
{
    Dimension size;
    public Image myimg = null;

    public imgPanel(Dimension size)
    {
        super();
        this.size = size;
    }

    public Dimension getPreferredSize()
    {
        return size;
    }

    public void update(Graphics g)
    {
        paint(g);
    }

    public void paint(Graphics g)
    {
        if (myimg != null)
        {
            g.drawImage(myimg, 0, 0, this);
        }
    }
```

```
        public void setImage(Image img)
        {
            if(img!=null)
            {
                this.myimg = img;
                update(getGraphics());
            }
        }
    }

}
```

# 6.2　音 频 抽 取

在抓取网络中的音乐文件时，通常会一并提出音乐文件中的歌手名、歌曲名等元信息。这些信息和音乐文件一起构成了我们检索的源材料(如图 6.2 所示)。如何从类似于 MP3 等的音乐文件中抽取出歌手名、歌曲名等元信息呢？本节主要讲述这些内容。

| 歌曲TOP500 | 经典老歌 | | | 歌手TOP200 | |
|---|---|---|---|---|---|
| 1 爱 | 小虎队 | 11 朋友 | 谭咏麟 | 1 S.H.E | 11 梁静茹 |
| 2 说谎 | 温兆伦 | 12 约定 | 王菲 | 2 周杰伦 | 12 王力宏 |
| 3 至少还有你 | 林忆莲 | 13 彩虹 | 羽泉 | 3 凤凰传奇 | 13 弦子 |
| 4 红豆 | 王菲 | 14 我只在乎你 | 邓丽君 | 4 王菲 | 14 张靓颖 |
| 5 大海 | 张雨生 | 15 情人 | 杜德伟 | 5 刘德华 | 15 刘若英 |
| 6 后来 | 刘若英 | 16 真的爱你 | Beyond | 6 郑源 | 16 邓丽君 |
| 7 海阔天空 | beyond | 17 童年 | 罗大佑 | 7 张学友 | 17 小虎队 |
| 8 勇气 | 梁静茹 | 18 新不了情 | 万芳 | 8 林俊杰 | 18 蔡依林 |
| 9 听海 | 张惠妹 | 19 黄昏 | 周传雄 | 9 BY2 | 19 陈奕迅 |
| 10 再回首 | 姜育恒 | 20 水手 | 郑智化 | 10 许嵩 | 20 五月天 |
| 更多>> | | | 试听全部 | 更多>> | 歌手列表 |

图 6.2　百度的 MP3 搜索

## 6.2.1　抽取音频

要想构建音频搜索引擎，就需要知道这种音乐文件的元信息，比如音乐名、作者等。这些元信息通常都隐藏在文件中。下面我们以 MP3 文件为例，来看看 MP3 文件的结构。

MP3 文件大体分为三部分：TAG_V2(ID3V2)，Frame 和 TAG_V1(ID3V1)。其中，ID3V2 包含了作者、作曲、专辑等信息，长度不固定，扩展了 ID3V1 的信息量。Frame 是指一系列的帧，个数由文件大小和帧长决定，每个 Frame 的长度可能不固定。每个 Frame 又分为帧头和数据实体两部分，帧头记录了 MP3 的位率、采样率、版本等信息，每个帧之间相互独立。ID3V1 包含了作者，作曲，专辑等信息，长度为 128 字节。

下面，首先来看一下普通的 MP3 的帧格式，也就是上面提到的 Frame 的格式。每个帧都有一个四字节长的帧头，接下来可能有两个字节的 CRC 校验，其存在由帧头中的具体信

息决定。接着就是帧的实体数据，也就是 MAIN_DATA 了。

(1) 帧头结构。

| 位置 (BIT) | 长度 (BITS) | 描述 |
|---|---|---|
| 31－19 | 12 | Frame sync(0xFFF) |
| 18/17 | 2 | Layer, 00 – reserved, 01 – Layer III<br>10 – Layer II, 11 － Layer I |
| 16 | 1 | protection_bit, 0 意味着受 CRC 保护，帧头后面跟 16 位的 CRC |
| 15-12 | 4 | bitrate_index，比特率 |
| 11-10 | 2 | sampling_frequency, 00 – 44.1kHz, 01 – 48kHz<br>10 – 32 kHz, 11 – 保留 |
| 9 | 1 | padding_bit,1 意味着帧里包含 padding 位，仅当采样频率为 44.1kHz 时发生 |
| 8 | 1 | private_bit |
| 7－6 | 2 | mode, 00－stereo, 01-joint stereo(intensity stereo and/or ms_stereo)<br>11- dual_channel, 11 – single_channel |
| 5-4 | 2 | mode_extension,在 Layer III 中表示使用了哪种 joint stereo 编码方式。<br>Intensity_stereo ms_stereo<br>00 off off<br>01 on off<br>10 off on<br>11 on on |
| 3 | 1 | copyright，1 表示受版权保护。 |
| 2 | 1 | original，0 表示该 bitstream 是一个 copy，1 表示是 original. |
| 1-0 | 2 | emphasis，表示会使用哪种 de-emphasis<br>00 － no emphasis, 01 – 50/15 microsec. Emphasis<br>10 – reserved, 11 – CCITT J.17 |

① 无论帧长是多少，每帧的播放时间都是 26ms。

② 数据帧大小：

FrameSize = 144 * Bitrate / SamplingRate + PaddingBit

当 144 * Bitrate / SamplingRate 不能被 8 整除，则加上相应的 paddingBit.

(2) MAIN_DATA。

MP3 的 granule 包含 18 × 32 个 subband 采样。每个数据帧含有两个 granule 的数据，其内容如下：

```
- main_data_end pointer
- side info for both granules (scfsi)
- side info granule 1
```

```
- side info granule 2
- scalefactors and Huffman code data granule 1
- scalefactors and Huffman code data granule 2
```

主要数据里包含了扩展参数(scalefactors)，哈夫曼编码的数据(Huffman encoded data)和补充信息(ancillary information)。其内容不再详述，可以参考 MP3 SPEC－IS0 11172-3 AUDIO PART。现在一般用的都是立体声，scfsi 的长度为 32 个字节。 这里要解释的一个概念就是位流——bitstream。通常我们接触到的数据都是整数，最小的单位就是 byte。虽然有时也会用一个字节里的不同位来表示不同的含义，但在解析数据的时候还是把它当作一个个字节看待。但对 MP3 这种数据格式来说，这是行不通的。在解码时，它的数据输入就是一个个比特流。其中一个或几个比特是采样数据或者信息编码。采样需要从整个 MAIN_DATA 里提取所需要的以 bit 为单位的参数和输入信号，从而进行解码。因此采样时需要一个子程序，getbit(n)，从缓冲中提取所需要的位，并形成一个新的整数，作为采样的输出。

了解了数据帧的格式，下面重点来了解一下 MP3 的 ID3V2 和 ID3V1 信息。其实，讲解数据帧是为了让读者对 MP3 的结构有个较完整的了解。对于抽取作者、歌曲名、专辑等信息来说，ID3 才是真正有价值的内容。

MP3 帧头中除了存储一些像隐私(private)、版权(copyright)、原始信息(original)等的简单音乐说明信息以外，没有考虑存放歌名、作者、专辑名、年份等复杂信息，而这些信息在 MP3 应用中非常必要。1996 年，FricKemp 在 Studio 3 项目中提出在 MP3 文件尾增加一块用于存放歌曲的说明信息，即形成了 ID3 标准，至今已制定出 ID3 V1.0，V1.1，V2.0，V2.3 和 V2.4 标准。版本越高，记录的相关信息就越详尽。

ID3 V1.0 标准并不周全，存放的信息少，无法存放歌词，无法录入专辑封面、图片等。V2.0 是一个相当完备的标准，但编写软件比较困难，虽然赞成此格式的人很多，在软件中真正实现的却极少。绝大多数 MP3 仍使用 ID3 V1.0 标准。此标准是将 MP3 文件尾的最后 128 个字节用来存放 ID3 信息，这 128 个字节的使用说明如表 6.1 所示。

表 6.1 ID3 V1.0 文件尾说明

| 字 节 | 长 度 | 说 明 |
|---|---|---|
| 1～3 | 3 | 存放 TAG 字符，表示 ID3 V1.0 标准，紧接其后的是歌曲信息 |
| 4～33 | 30 | 歌名 |
| 34～63 | 30 | 作者 |
| 64～93 | 30 | 专辑名 |
| 94～97 | 4 | 年份 |
| 98～127 | 30 | 附注 |
| 128 | 1 | MP3 音乐类别(共 147 种) |

ID3 V2 到现在一共有 4 个版本，但流行的播放软件一般只支持第 3 版，既 ID3 V2.3。由于 ID3 V1 记录在 MP3 文件的末尾，ID3 V2 就只好记录在 MP3 文件的首部了。也正是这个原因，对 ID3 V2 的操作比 ID3 V1 要慢。而且 ID3 V2 结构比 ID3 V1 的结构要

复杂得多，但比前者全面且可以伸缩和扩展。

下面就介绍一下 ID3 V2.3。

每个 ID3 V2.3 标签都由一个标签头和若干个标签帧或一个扩展标签头组成。关于曲目的信息如标题、作者等都存放在不同的标签帧中，扩展标签头和标签帧并不是必要的，但每个标签至少要有一个标签帧。标签头和标签帧一起顺序存放在 MP3 文件的首部。

### 1. 标签头

在文件的首部顺序记录 10 个字节的 ID3 V2.3 的头部。数据结构如下：

```
char Header[3];     //存放的值必须为"ID3"，否则认为标签不存在
char Ver;           //版本号
char Revision;      //副版本号
char Flag;          //标志字节
char Size[4];       //标签大小，包括标签头的 10 个字节和所有的标签帧的大小
```

1) 标志字节

标志字节一般为 0，定义如下：

```
abc00000
```

a 表示是否使用 Unsynchronisation。

b 表示是否有扩展头部，一般没有，所以一般也不设置。

c 表示是否为测试标签(99.99%的标签都不用于测试，所以一般也不设置)。

2) 标签大小

一共四个字节，但每个字节只用 7 位，最高位不使用恒为 0。格式如下：

```
0xxxxxxx 0xxxxxxx 0xxxxxxx 0xxxxxxx
```

### 2. 标签帧

每个标签帧都由一个 10 个字节的帧头和至少一个字节的不固定长度的内容组成。它们是顺序存放在文件中，和标签头及其他的标签帧没有特殊的字符分隔。得到一个完整的帧的内容只有先从帧头中得到内容大小，之后才能顺序读出。

帧头的定义如下：

```
char FrameID[4]; //四个字符的帧标识
char Size[4]; //帧内容的大小，不包括帧头，不得小于 1
char Flags[2]; //存放标志，只定义了 6 位
```

1) 帧标识

用四个字符标识一个帧，说明一个帧的内容含义，常用的对照如下：

```
TIT2=标题
TPE1=作者
TALB=专集
TRCK=音轨(格式：N/M：其中 N 为专集中的第 N 首，M 为专集中共 M 首，N 和 M 为 ASCII 码表示的数字)
```

```
TYER=年代（是用 ASCII 码表示的数字）
TCON=类型（直接用字符串表示）
COMM=备注（格式："eng\0 备注内容"，其中 eng 表示备注所使用的自然语言）
```

2) 大小

这个字段由 4 个字节组成，每个字节的 8 位全都使用，格式如下

```
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
```

算法如下：

```
int FSize;
FSize = Size[0]*0x100000000
+Size[1]*0x10000
+Size[2]*0x100
+Size[3];
```

3) 标志

只定义了 6 位，另外的 10 位为 0，但大部分情况下，16 位都为 0 就可以了。格式如下：

```
abc00000 ijk00000
```

- a：标签保护标志，设置时认为此帧作废。
- b：文件保护标志，设置时认为此帧作废。
- c：只读标志，设置时认为此帧不能修改。
- i：压缩标志，设置时一个字节存放两个 BCD 码表示数字。
- j：加密标志。
- k：组标志，设置时说明此帧和其他的某帧同组。

以上解析了 MP3 文件的结构。如果获得的文件是其他类型的音乐文件，也可以根据文件的结构进行解析。现在搜索引擎大多数都只支持 MP3 格式文件的搜索。在下一节，我们讲解在 Java 中如何解析 MP3 音乐文件。

### 6.2.2 学习 Java 音频抽取技术

Sourceforge 提供了一个方便的解析 MP3 文件的开源库：org.farng.mp3。它可以有效地帮助我们读取 MP3 中的各种信息。例如：

```
import java.io.IOException;
import org.farng.mp3.MP3File;
import org.farng.mp3.TagException;
import org.farng.mp3.id3.AbstractID3v2;
import org.farng.mp3.id3.ID3v1;
import org.farng.mp3.lyrics3.AbstractLyrics3;
public class TestMP3 {
    /**
     * @param args
     */
```

```
    public static void main(String[] args) {
        try {
            //MP3File file = new
MP3File("c:\\TDDOWNLOAD\\shuangjiegun.mp3");//1,2
            MP3File file = new MP3File("c:\\TDDOWNLOAD\\1.mp3");//1,lyrics
            // MP3File file = new
MP3File("/home/zhubin/Music/1.mp3");//1,lyrics
            AbstractID3v2 id3v2 = file.getID3v2Tag();
            ID3v1 id3v1 = file.getID3v1Tag();
            if (id3v2 != null) {
                System.out.println("id3v2");
                System.out.println(id3v2.getAlbumTitle());//专辑名
                System.out.println(id3v2.getSongTitle());//歌曲名
                System.out.println(id3v2.getLeadArtist());//歌手
            } else {
                System.out.println("id3v1");
                System.out.println(id3v1.getAlbumTitle());
                System.out.println(id3v1.getSongTitle());
                System.out.println(id3v1.getLeadArtist());
            }
            AbstractLyrics3 lrc3Tag = file.getLyrics3Tag();
            if (lrc3Tag != null) {
                String lyrics = lrc3Tag.getSongLyric();
                System.out.println(lyrics);
            }
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (TagException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println("over");
    }
}
```

输出结果如下：

```
id3v1
依然范特西
听妈妈的话
周杰伦
[ti:听妈妈的话]
[ar:周杰伦]
[al:依然范特西]
[by:Hybert Kwok]

[00:00.11]听妈妈的话
[00:01.11]词：周杰伦 曲：周杰伦
```

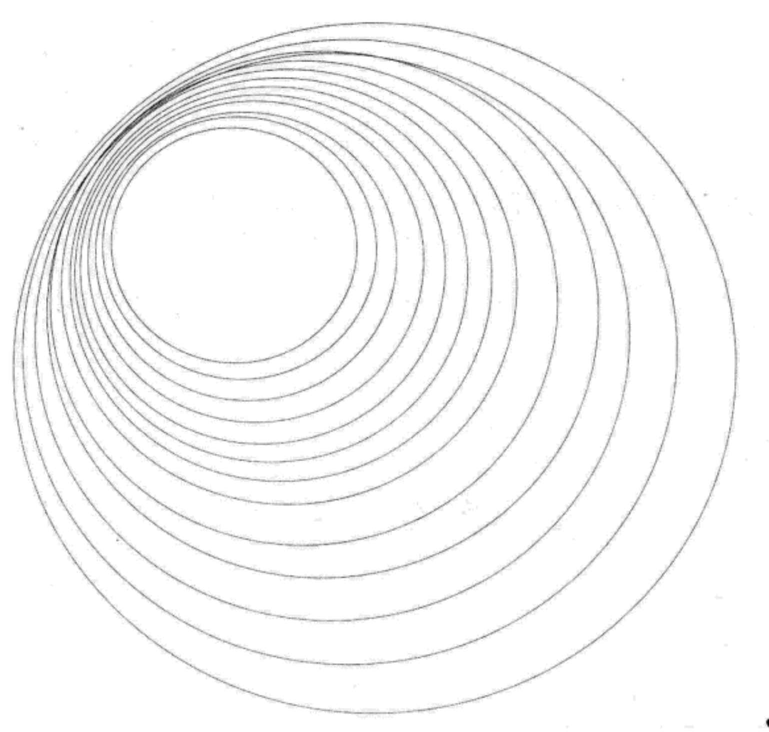
```
[00:02.11]编曲：林迈可 洪敬饶 制作人：周杰伦
[00:04.11]和声编写：周杰伦 和声：周杰伦
[00:06.11]录音室：阿尔发录音室
[00:08.11]歌词录入：Hybert Kwok
[00:09.00]
[00:10.14]小朋友 你是否有很多问号
[00:12.76]为什么 别人在那看漫画
[00:15.39]我却在学画画 对着钢琴说话
[00:17.87]别人在玩游戏 我却靠在墙壁背我的 ABC
[00:20.88]我说我要一台大大的飞机
[00:23.41]但却得到一台旧旧录音机
[00:25.98]为什么要听妈妈的话
[00:28.29]长大后你就会开始懂了这段话 哼
[03:15.77][00:30.76]长大后我开始明白
[03:18.74][00:33.73]为什么我 跑得比别人快
[03:20.15][00:35.14]飞得比别人高
[03:21.06][00:36.38]将来大家看的都是我画的漫画
[03:24.02][00:38.73]大家唱的都是 我写的歌
[03:26.81][00:41.45]妈妈的辛苦 不让你看见
[03:29.22][00:44.07]温暖的食谱在她心里面
[03:31.81][00:46.51]有空就多多握握她的手
[03:34.65][00:49.20]把手牵着一起梦游
[03:36.84][02:55.18][02:13.93][00:51.60]听妈妈的话 别让她受伤
[03:47.22][03:05.96][02:24.72][01:01.64]想快快长大 才能保护她
[03:57.60][02:34.66][01:11.43]美丽的白发 幸福中发芽
[04:07.82][02:45.16][01:22.46]天使的魔法 温暖中慈祥
[01:32.93]在你的未来 音乐是你的王牌
[01:34.71]拿王牌谈个恋爱
[01:36.18]唉！我不想把你教坏
[01:37.71]还是听妈妈的话吧
[01:39.02]晚点再恋爱吧
[01:40.46]我知道你未来的路
[01:42.59]但妈比我更清楚
[01:43.90]你会开始学其他同学
[01:45.03]在书包写东写西
[01:46.00]但我建议最好写妈妈
[01:47.60]我会用功读书
[01:48.73]用功读书 怎么会从我嘴巴说出
[01:50.84]不想你输 所以要叫你用功读书
[01:53.97]妈妈织给你的毛衣 你要好好的收着
[01:56.06]因为母亲节到的时候我要告诉她我还留着
[01:58.41]对了！我会遇到了周润发
[02:00.81]所以你可以跟同学炫耀
[02:01.88]赌神未来是你爸爸
[02:02.97]我找不到 童年写的情书
[02:05.44]你写完不要送人
[02:06.52]因为过两天你会在操场上捡到
[02:07.90]你会开始喜欢上流行歌
[02:11.31]因为张学友开始准备唱《吻别》
```

PDG

```
[04:22.93]END
over
```

## 6.3 本章小结

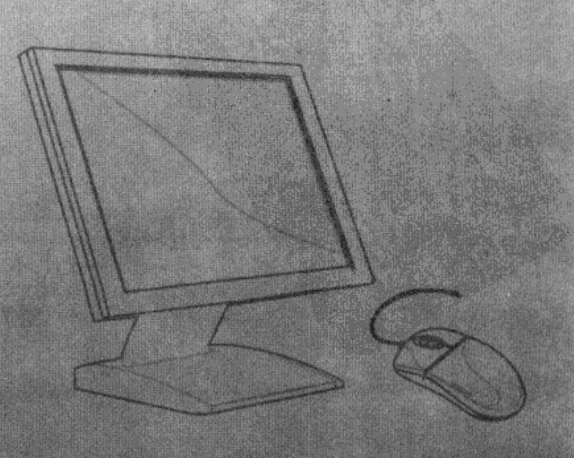
本章介绍了多媒体的抽取技术，内容包括从视频抽取到音频抽取，重点介绍了关键帧抽取和音频文件中元数据的抽取。如果想了解更多的内容，请参考相关文献。



# 第7章

## 去掉网页中的“噪声”

互联网爬虫处理的主要文档格式是网页，而去除“噪声”信息以及从网页中提取出正文和标题等信息是从互联网提取有效信息的第一步。



## 7.1 “噪声”对网页的影响

许多网页中都包含与主要内容无关的文本、链接、图片和 Flash 动画等。如图 7.1 所示的新闻网页正文块以外的部分都可以认为是噪声。

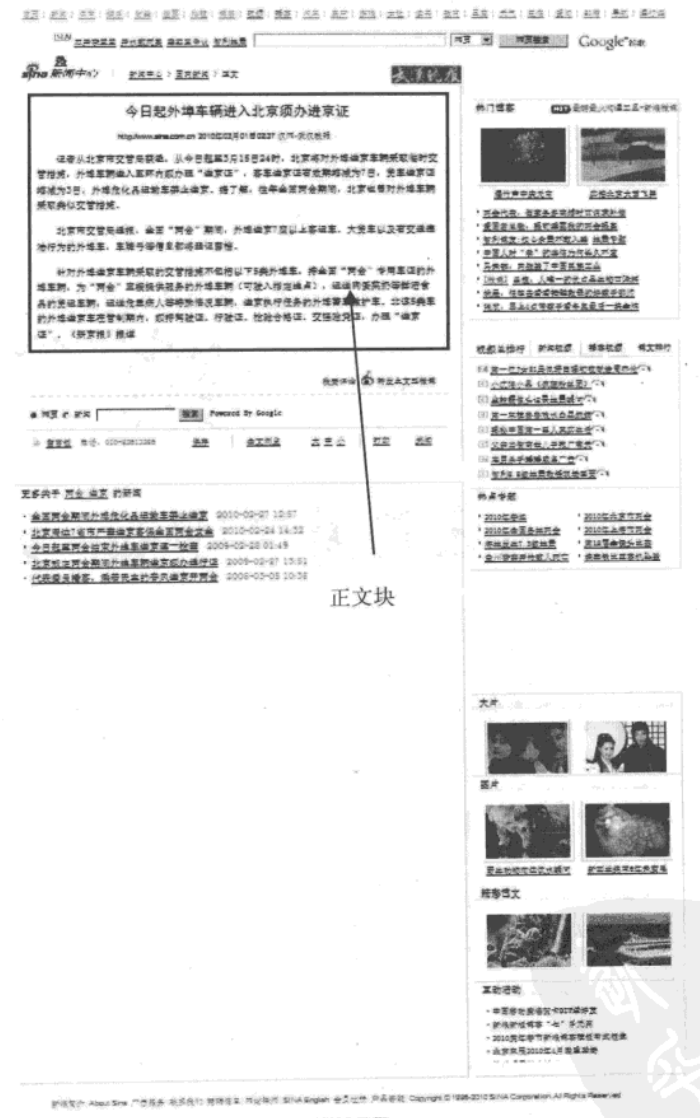



图 7.1 网页的正文块

可以看出正文块在这个网页中所占的显示区域小于 20%，其余部分由广告、分割条、导航链接、搜索服务和版权信息等组成。

在主题搜索领域，大量的广告、导航条等噪声内容会导致主题漂移(topic drift)。这说明传统的主题搜索算法中以网页为粒度构造的 Web 图不够准确，必须深入到网页内部将处理单元的粒度缩小，才能提高内容分析的准确性。

在 Web 信息检索领域，检索结果的相关性和检索的速度是评价 Web 检索系统的两个指标。如果不去除原始网页中的噪声内容，检索系统对噪声内容也建立索引，从而导致仅仅因为查询词在某张网页的噪声内容中出现，而把该网页作为结果返回，而网页的主题内容可能和这个查询词完全无关。可以看出，噪声内容不仅会使索引结构的规模变大，而且还会导致检索系统准确性的下降。对网页排重计算来说，最好不要计入噪声内容，否则有区别的内容块容易被噪声内容淹没，也不能让噪声内容出现在网页摘要中。

为了去除噪音，简单的方法是针对某一类具体网页，人工提取该类网页的内容组织模式。然后，信息提取系统根据该模式从属于该类的网页中提取相应内容。例如，针对每个网站的同类模板的网页编写特定的提取规则。在具体实现上，可以为每类网页配置提取模板，或者手工编写正则表达式提取规则。但这些方法有一个共同局限性，那就是需要人工提取内容组织模式，这对于内容组织繁多的互联网来说显然代价太高。因此在很多应用中，一般采用统计的方法实现通用的网页去噪。

## 7.2　利用“统计学”消除“噪声”

判断网页中的片断是“噪声”还是“有效信息”，这是一个两类分类问题。在缺少训练集的情况下，可以采用自定义规则来分类，或者把它看成聚成两类的聚类问题。如果有训练集，可以采用支持向量机来分类。

从流程上，可以先对网页自动分类，然后对每类网页应用不同的分类方式或者使用不同的分类参数。

可以简单地把网页分成目录导航式页面(或者说列表页 List Page)、详细页面(Detail Page)和未知类型(Unknow Page)页面。此外，还可以分出图片页面、论坛页面、博客页面等。例如，图 7.1 所示的页面就是一个详细页。一个列表页往往指向多个详细页，如图 7.2 所示。

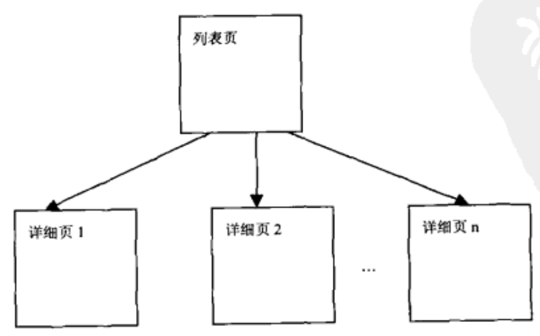
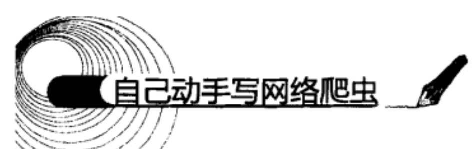



图 7.2　列表页和详细页之间的关系




网页自动分类的基础是网页相似度计算。网页本身可以抽象成串行的节点或者 DOM 树。

对于串行序列，可以采用最长公共子串来衡量相似度。对于树型结构，可以通过树相似度来衡量网页相似度。

HtmlParser(http://htmlparser.sourceforge.net/)是一个 Html 文件解析程序，可以把网页解析成串行的节点。

首先把网页分成由 HTML 标签组成的节点。例如：

```
<font class="nrbt">广告业务</font>
```

由三个节点组成："<font class="nrbt">"是标签节点(TagNode)，"广告业务"是文本节点(Text Node)，"</font>"也是标签节点(TagNode)，不过它的 isEndTag()方法返回真。

可以通过 Firefox 浏览器的"DOM 查看器"查看网页 DOM 树，如图 7.3 所示。

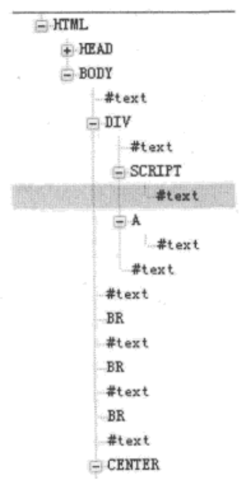



**图 7.3　网页 DOM 树**

可以使用 NekoHTML(http://nekohtml.sourceforge.net/)解析 HTML 文档并形成标准的 DOM 树。NekoHTML 首先通过扫描器把 HTML 转换成串行的 XNI(Xerces Native Interface)文档"事件"，然后用标签补偿器(tag balancer)进一步加工成支持 DOM 接口的树。

下面介绍用最长公共子序列(Longest Common Subsequence，LCS)来衡量网页相似度的方法。举例说明两个序列 s1 和 s2 的最长公共子序列。

```
s1 = { a , b , c , b , d , a , b }
s2 = { b , d , c , a , b , a }
```

则 s1 和 s2 的最长公共子序列为 LSC(s1, s2) = { b , c , b , a }。

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]。

计算两个序列的最长公共子序列的实现如下：

```
public static <E> List<E> longestCommonSubsequence(E[] s1, E[] s2)
{
        //二维数组，初始化为 0
        int[][] num = new int[s1.length+1][s2.length+1];

        //动态规划计算
        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如下：

- 把网页抽象成一个节点数组。
- 计算两个网页对应的节点数组的最长公共子序列长度 LCS-Length。

- 计算两个网页的相似度 sim=(2*LCS-Length)/(Length1+Length2)。

输入两个 URL 地址，计算两个网页之间的相似度，实现代码如下：

```
public static double getPageSim(String urlStr1,String urlStr2) {
    ArrayList<Node> pageNodes1 = new ArrayList<Node>();

    URL url1 = 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);
    }

    double distance = PageDistance. longestCommonSubsequence
                            (pageNodes1,
                            pageNodes2);
    return (2.0*distance)/
            ((double)pageNodes1.size() + (double)pageNodes2.size());
}
```

对网页进行分类可以利用的特征有：链接密度、背景颜色、高宽比、在页面上的位置、平均句子长度、不在锚点上的文字。

### 7.2.1 网站风格树

需要对 DOM 树的每个节点分类，而不是对节点下的整个网页段落分类，因为段落可能混合了“噪声节点”和“有效节点”。本节介绍用网站风格树的方法为 DOM 树中的每个节点设置权重特征。

每个 HTML 页面对应于一个 DOM 树，其中标签是 DOM 树内部的节点，而详细的文本、图像或者超链接则是叶节点。图 7.4 展示了 HTML 的一部分以及与它相应的 DOM 树。在 DOM 树中，长方形框代表标签节点，阴影方块代表节点的实际内容，例如：标签 IMG 的实际内容是“src=image.gif”。每个节点都包含了宽度或颜色等显示属性。注意，这里只从 BODY 标签开始研究 HTML 网页，因为所有的可视部分都在 BODY 范围内。为便于分析，在该 DOM 树内，增加了一个虚拟根目录节点(root)，根节点没有任何属性，作为 BODY

的父亲节点。图 7.4 是一个 DOM 树的例子，其中较低水平的标签没有画出来：

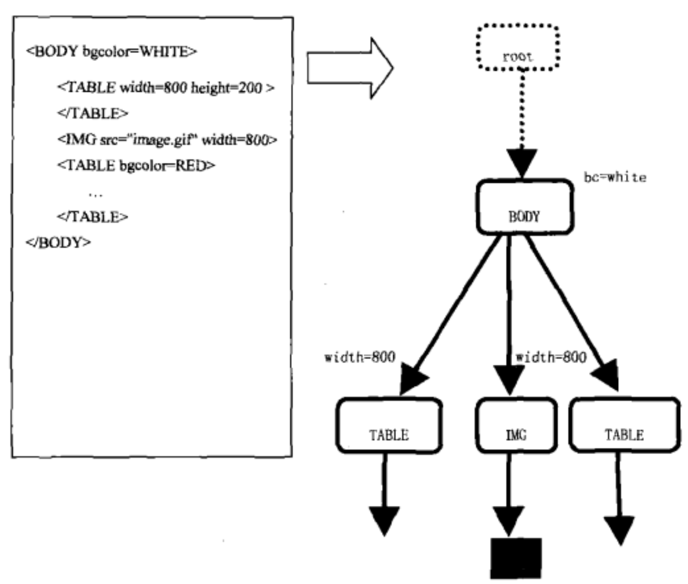



图 7.4　从网页到 DOM 树

虽然 DOM 树已经足以描述 HTML 页面的布局和呈现风格，但是很难基于单个的 HTML 页面了解一组 HTML 页面所有的呈现风格并且对它们整体去噪。因此，在同时考虑呈现风格和网页的实际内容时，DOM 树已不再适用于去噪工作。为此，我们需要一个更强有力的结构。这个结构至关重要，因为我们的算法需要从一个站点发现网页共同的风格，以便消除噪声。现在引入一种称为风格树(Style Tree)的新的树结构。该树结构能压缩一组相关网页的共同呈现风格。

图 7.5 给出了一个风格树的例子，这个风格树合并了 DOM 树 $d_1$ 和 $d_2$。除了位于底层的四个标签(P、IMG、P 和 A)，$d_1$ 中的所有标签在 $d_2$ 中都有相应的标签。因此，$d_1$ 和 $d_2$ 是可以被压缩的。

可以使用计数器来指示在风格树的一个特定层上具有一种特定风格的网页有多少。在图 7.5 中，我们能看到两个网页都以 BODY 开始，并且因此 BODY 的计数为 2。

BODY 节点往下，两个网页有同样的 TABLE-IMG-TABLE 风格。把整个 TABLE-IMG-TABLE 标签序列叫做风格节点，这个节点包围在图 7.6 所示的虚线长方形框内。此时，它代表一种特定的呈现风格。因此，在 DOM 树中，一个风格节点是一个标签节点序列。我们把这些标签节点称为元素节点，以便在 DOM 树内把它们从标签节点中鉴别出来。例如，TABLE-IMG-TABLE 风格节点具有三个元素节点：TABLE、IMG 和 TABLE。

元素节点和来自 DOM 树的标签节点不同，后面将给出元素节点的定义。

在图 7.6 中，可以看到右侧的 TABLE 标签下，$d_1$ 和 $d_2$ 分岔，在风格树中由两个不同风格的节点来反映。两个风格节点分别是 P-IMG-P-A 和 P-BR-P。这意味着在右侧 TABLE 节点下，有两种不同的呈现风格。这两个风格节点的网页计数器都是 1。风格树是两个 DOM

树的压缩表现形式。通过风格树，可以看到 DOM 树的哪些部分是相同的，哪些部分是不同的。

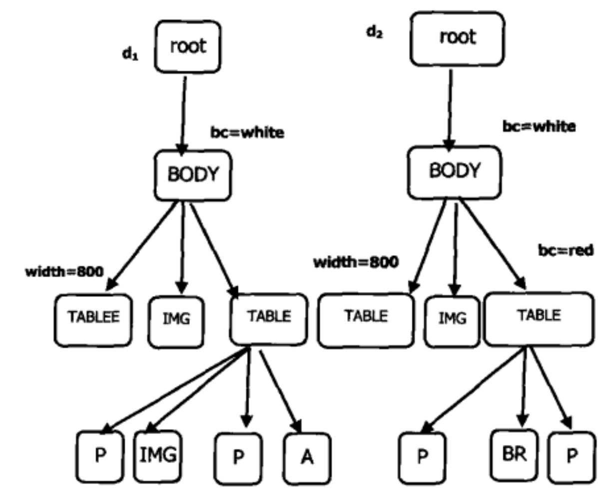



图 7.5　风格树示例

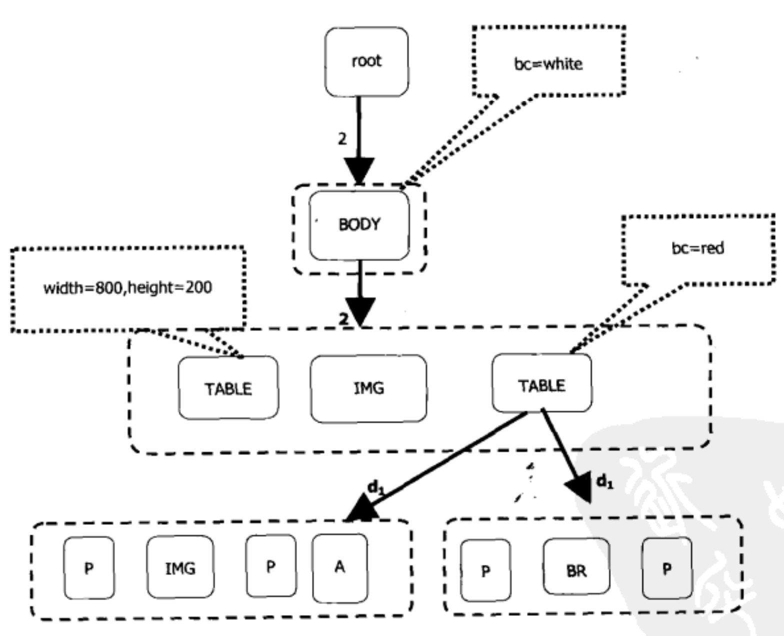



图 7.6　两个网页的风格树

1. 风格树(Style Tree)

我们现在定义的风格树包括两种类型的节点，即风格节点和元素节点。

一个风格节点代表了一个布局或呈现风格，它有两个组成部分，记作(ES, *n*)，其中 ES

是元素节点的一个序列，$n$ 是有该风格的网页的数量。

在图 7.6 中，风格节点(在一虚线方框内)的 P-IMG-P-A 有 4 个元素节点：P、IMG、P 和 A，并且 $n$=1。实现风格节点的数据结构如下：

```
public class StyleNode {
    private int count;
    private ArrayList<ElementNode> nodeList =
     new ArrayList<ElementNode>();
...
}
```

一个元素节点 E 有三个组成部分，记作(TAG, Attr, Ss)，其中：

- TAG 是标签的名称，例如，TABLE 和 IMG。
- Attr 是 TAG 的一套显示属性，例如，bgcolor =RED、weidth=100 等。
- Ss 是 E 下面的一套风格节点。

请注意，在 DOM 树内，一个元素节点对应一个标签节点，但是指向一组孩子风格节点 Ss(见图 7.6)。为了方便起见，我们通常用标签的名称标记元素节点，用元素节点序列标记风格节点。实现元素节点的数据结构如下：

```
public class ElementNode {
    private String tagName;
    private String nodeValue;
    private boolean isText;
    private HashMap<String,String> atrrMap = new HashMap<String,String>();
    private ArrayList<StyleNode> children = new ArrayList<StyleNode>();
    private ArrayList<String> contents = new ArrayList<String>();
    ...
}
```

为网站中的网页创建一个风格树(称为网站风格树或 SST)非常简单。首先我们给每个网页创建一个 DOM 树，然后自上而下合并成风格树。在风格树的一个特定元素节点 E 处(E 在 DOM 树中对应的标签节点是 T)，检查一下 DOM 树中 T 的子标签节点序列是否与位于 E 下的风格节点 S 中的元素节点的序列相同。如果相同，则只是增加了风格节点 S 的网页计数器，然后合并风格树和 DOM 树中剩下的节点。如果不相同，风格树中的元素节点 E 的下方就可以创建一个新的风格节点。在转换为风格树的风格节点和元素节点后，把 DOM 树中标签节点 T 的子树复制到风格树内。

### 2. 在风格树内确定噪声元素

噪声的定义基于以下假设：①一个元素节点具有的呈现风格越多，该元素节点越重要。②某个元素节点的实际内容具有的分岔越多，该元素节点越重要。这两个重要性的值用于评估元素节点的重要性。计算呈现风格重要度的目的是用呈现风格的数量来检测噪声，与此同时，计算内容重要性的目的是鉴别这些以类似的呈现风格呈现的网页的主要内容。最后，通过结合呈现重要性和内容重要性，给出元素节点的重要性。一个元素节点的结合重要性越大，就越有可能是主要的网页内容。

在图 7.7 所示的网站风格树的例子中，SST 的阴影部分很可能是噪声，因为它们的呈现风格(连同其实际内容，在图 7.7 中未标示出)是很规则和固定的，因此不那么重要。图 7.7 中的双线 Table 元素节点有许多子风格节点，也就是说，该元素节点很可能是重要的。也就是说，双线 Table 更可能包含网页的主要内容。图 7.7 中的双线 Text 也有意义，因为它的呈现风格虽然是固定的，但其对应网页的内容不同。SST 可以用一个网站的所有网页创建出来。

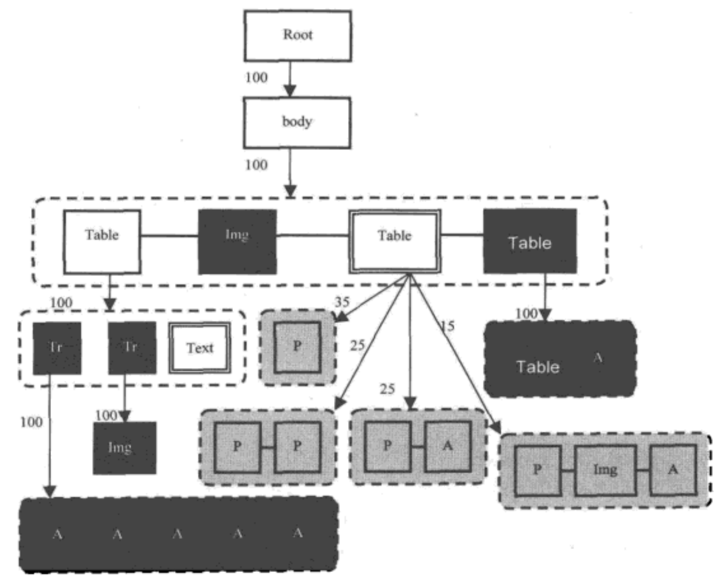



**图 7.7　网站风格树示例**

可以用信息理论中的熵来衡量呈现风格的重要性。

**节点重要性计算方法**：在 SST 中，对于某个元素节点 $E$，设 $m$ 是包含 $E$ 的网页数，$l$ 是 $E$ 的子风格节点的数量(即 $l=|E.\mathrm{Ss}|$)，$E$ 的节点重要性由 NodeImp($E$)表示，定义为

$$\mathrm{NodeImp}(E)=\begin{cases}-\sum_{i=1}^{l} p_i \log_m p_i & , m>1 \\ 1 & , m=1\end{cases} \tag{1}$$

其中，$p_i$ 表示在 $E$.Ss 中，网页使用第 $i$ 个风格节点的可能性。直观上，如果 $l$ 小，E 在不同的风格中被呈现出来的可能性就小，因此 NodeImp($E$)的值小。如果 $E$ 包含多种呈现风格，那么 NodeImp($E$)的值就大。例如，在图 7.7 展示的 SST 中，元素节点 BODY 的重要性是 0($l$ log 100 $l$=0)，因为 $l$=1。即，在 BODY 下，仅有一种呈现风格 Table-Img-Table-Table。双线 Table 的重要性是：

```
NodeImp(TABLE) = -0.35log1000.35 - 2×0.25log1000.25-0.15log1000.15 = 0.292 > 0
```

计算内部节点重要性的代码如下：

```
public double getNodeImportance() {
        double sum = 0.0;
        int numOfStyles = 0;

        for (StyleNode child : this.children) {
                numOfStyles += child.getCount();
        }

        for (StyleNode child : this.children) {
                sum += child.getImportance(numOfStyles);
        }

        return sum;
}
```

其中 child.getImportance 的实现如下：

```
public double getImportance(int numOfStyles) {
        if (numOfStyles == 1) {
                return 1.0;
        }
        double ratio = (double)count / numOfStyles;
        return -ratio * Math.log(ratio) / Math.log(numOfStyles);
}
```

然而，不能只考虑节点重要性，因为 BODY 是一个噪声，所以还没有考虑其后代的重要性。我们使用合成重要性去测量一个元素节点及其后代的重要性。

**综合重要性计算方法**：对叶子节点和非叶子节点(也就是内部节点)可以用两种不同的方法计算。这里首先介绍内部节点的计算方法，然后再介绍叶子节点的计算方法。

在 SST 内，对于一个内部元素节点 $E$，设 $l = |E.\mathrm{Ss}|$。$E$ 的综合重要性由 CompImp($E$)表示。定义为

$$\mathrm{CompImp}(E) = (1-\gamma^{l})\mathrm{NodeImp}(E) + \gamma^{l}\sum_{i=1}^{l}(p_i \mathrm{CompImp}(s_i)) \tag{2}$$

其中 $p_i$ 表示在 $E$.Ss 中 $E$ 具有第 $i$ 个子风格节点的可能性。在上述等式中，CompImp($S_i$)是一个风格节点 $S_i$ ($\in E$.Ss)的综合重要性，定义为

$$\mathrm{CompImp}(S_i) = \frac{\sum_{j=1}^{k}\mathrm{CompImp}(E_i)}{k} \tag{3}$$

其中 $E_j$ 是 $S_i.E$ 中的一个元素节点，并且 $k = |S_i.\mathrm{Es}|$，表示在 $S_i$ 中元素节点的数目。

在公式(2)中，$\gamma$是衰减因子，设置为 0.9。当 $l$ 较大时，它增加了 NodeImp($E$)的权重；当 $l$ 较小时，它减少了 NodeImp($E$)的权重。这意味着，一个风格节点具有的子风格节点越多，其本身的综合重要性越大，所具有的子风格节点数越少，其子代的综合重要性越大。

计算内部节点的综合重要性的实现代码如下：

```
public double getCompositeImportance() {
```

```
        double sum = 0.0;

        for (StyleNode child : this.children) {
               sum += child.getCompositeImportance();
        }
        return (1 - ElementNode.RAMDA) * this.getNodeImportance()
                   + ElementNode.RAMDA * sum;
}
```

叶子节点与内部节点不同，因为它们只有实际内容没有标签。我们基于其实际内容(即文本、图片、连接等)中的信息定义一个叶元素节点的综合重要性。

在 SST 中的一个叶元素节点 $E$，设 $l$ 是出现在 $E$ 中的特征的数目(即文字、图片文件、连接参考等)，并且设 $m$ 是包含 $E$ 的网页数目，$E$ 的综合重要性定义为

$$\text{CompImp}(E)=\begin{cases}1 & ,\quad m=1\\ 1-\dfrac{\sum_{i=1}^{l}H(a_i)}{l} & ,\quad m>1\end{cases} \tag{4}$$

其中，$a_i$ 是 $E$ 中内容的实际特征。$H(a_i)$是在 $E$ 中 $a_i$ 的熵，定义为

$$H(a_i)=\begin{cases}0 & ,m=1\\ -\sum_{j=1}^{m}p_{ij}\log_m p_{ij} & ,m=1\end{cases} \tag{5}$$

其中，$p_{ij}$ 是网页 $j$ 的元素节点 $E$ 中出现 $a_i$ 的可能性。

如果 $m$=1，则意味着只有一个网页含有 $E$，那么 $E$ 是一个非常重要的节点，并且它的综合重要性 CompImp 是 1(CompImp 的值归一化成 0～1 之间的数值)。

例如计算图 7.8 这个风格树中节点 $E$ 的综合重要性：

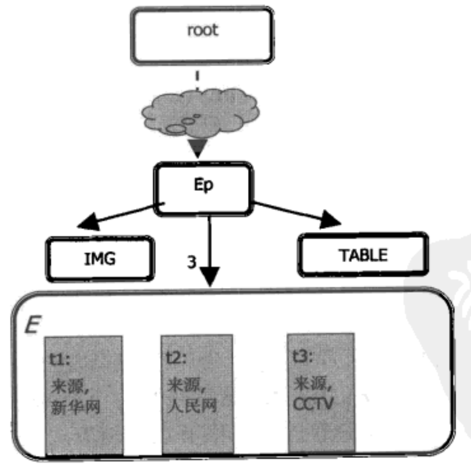



图 7.8　风格树示例

包含 $E$ 的网页数目 $m = |E.\text{TAGs}| = 3$
特征个数 $N = |\{$来源，新华网，人民网，CCTV $\}| = 4$
熵值 $H_E(\text{来源}) = -3 * (1/3 \log_3 1/3) = 1$
熵值 $H_E(\text{新华网}) = H_E(\text{人民网}) = H_E(\text{CCTV}) = -(0+0+1 \log_3 1) = 0$

$$\text{NodeImp}(E) = 1 - \frac{\sum_{i=1}^{l} H(a_i)}{l}$$

$$= 1-(H_E(\text{来源}) + H_E(\text{新华网}) + H_E(\text{人民网}) + H_E(\text{CCTV}))/4$$

$$= 1-(1+0+0+0)/4 = ((1-1)+3*(1-0))/4 = 0.75$$

计算叶元素节点综合重要性的代码如下：

```
public double getLeafCompositeImportance() {
   boolean isText = true;
   for (ElementNode styleNode : nodeList) {
          if(!styleNode.isLeaf()) {
                isText = false;
                break;
             }
   }

   if(!isText){
         return 0;
   }
   int m = count;
   if(m==0){
         return 0;
   }
   HashMap<String,Integer> atts = new HashMap<String,Integer>();
   for (ElementNode styleNode : nodeList) {
         ArrayList<String> contents = styleNode.getContents();
         for(String a:contents)
         {
             Integer count = atts.get(a);
             if(count == null)
             {
                 atts.put(a, 1);
             }
             else
             {
                 ++count;
                 atts.put(a, count);
             }
         }
   }
```

```
        double sumHE =0;

        for (Entry<String, Integer> attEntry : atts.entrySet()) {
             String att = attEntry.getKey();
             for (ElementNode styleNode : nodeList) {
                 ArrayList<String> contents = styleNode.getContents();
                 double heAtt = 0;
                 if(contents.contains(att)) {
                     double p = 1/(double)(attEntry.getValue());
                     heAtt = p*( Math.log(p) / Math.log(m) );
                 }
                 sumHE += heAtt;
             }
        }
        int n = atts.size();
        if(n==0)
             return 0;
        return (1+sumHE/n);
}
```

所有的元素节点与风格节点的综合重要性计算都可以通过遍历 SST 来实现，遍历树可以用递归调用的方法实现。可以把网页去噪看成是网页上的标签分类成“噪声”还是“有效信息”的分类问题。SST 贡献了一个对网页上的标签分类的特征。以前的特征是基于单个页面的，但这个特征是基于多个页面的。

### 7.2.2 “统计学去噪”Java 实现

用统计的方法去噪并提取网页正文的流程如下。

(1) 删除噪音块：网页去噪的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。

(2) 划分段落。可以把 HTML 页面划分成多个段落(Paragraph)。简单的实现方法是，根据<td><p><br><div><table>这些标签来划分段落。

(3) 评估段落。每个段落内的文字因其在视觉上或者是主题上对文档的贡献程度而具有不同的权重。选取分值最大段落作为正文块。

下面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;
    }
```

下面是计算一个文本中大概包括的正文长度的方法：

```
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;
}
```

根据链接文字比率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。

下面是其实现代码：

```
/**
 * 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;
    double words;

    //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) {
        iNode.getParentNode().removeChild(iNode);
    }
}
```

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 7.9 显示了来源于新华网的两个页面之间的对应关系。

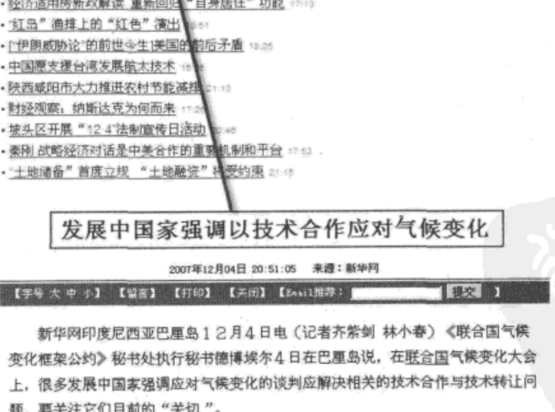



图 7.9　网页标题和锚点文本的对应关系

取得网页标题以后，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;
}
```

# 7.3 利用“视觉”消除“噪声”

## 7.3.1 “视觉”与“噪声”

基于视觉的网页分块算法的原理来自于用户的实际观察和体验，即用户并不关心 Web 页面的内部结构，而是使用一些视觉因素，比如背景颜色、字体颜色和字号大小、边框、逻辑块和逻辑块之间的间距等来识别页面中的语义块。因此如果充分使用 Web 页面的视觉信息，模拟人眼识别语义块的过程，并结合 DOM 树结构分析进行页面分块，则可以达到更好的效果。

微软亚洲研究院的蔡登等人在 2003 年首次提出了基于视觉的网页分块算法 VIPS(Vision-based page segmentation)。VIPS 算法充分利用了 Web 页面的布局特征(见图 7.10)，它有三个主要步骤：首先在 DOM 树中以较小的粒度提取出所有可视标签块，并且给每个可视标签块计算出一个“一致性程度”(Degree of Coherence，DoC)值来描述该块内部内容的相关性。DoC 的值越大，则表明该块内部内容之间的联系越紧密，反之越松散。第二步利用每个可视标签块的绝对位置和相对位置信息，检测出它们之间所有的分割条，包括水平和垂直方向。最后基于这些分割条，利用更多的诸如颜色等视觉信息，重新构建 Web 页面的内容结构(见图 7.11)。

VIPS 算法的优点十分明显，它充分利用了网页的视觉信息和结构信息，相对于传统的基于规则的分块算法来说，大大提高了分块的精确度。但提取网页视觉信息的代价很高。因为 HTML 语言本身并没有包含足够的视觉信息，所以网页真正显示出来的效果会因浏览器、操作系统，甚至硬件而异。为了得到网页的完整视觉信息，必须完全下载该网页所链接的 CSS 文件、JavaScript 文件、图片文件等，然后调用浏览器内核代码渲染这些网页文件，最后从浏览器内核代码的接口中得到每个 HTML 标签的视觉信息。整个步骤不仅耗时，而且十分依赖于浏览器内核代码。有的 VIPS 算法实现了调用 IE 浏览器的 COM 接口。在 Linux 编程环境下，可以利用的只有 Mozilla(Firefox)浏览器的开源代码。但 Mozilla 代码并没有针

对网页视觉信息提取这一需求给出方便的使用接口，只有在其渲染完网页之后才能截取视觉信息。

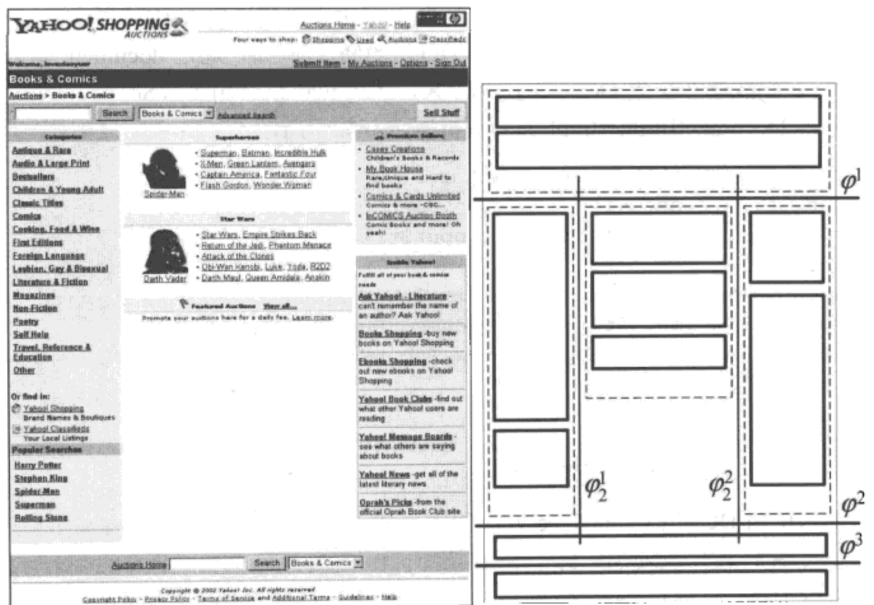



图 7.10　从网页到布局特征

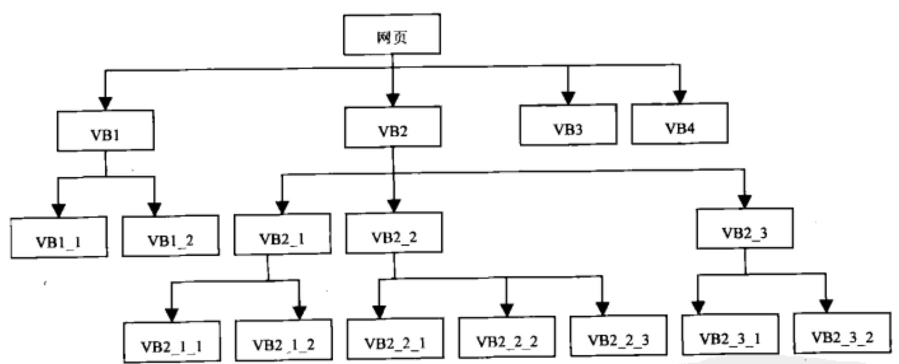



图 7.11　基于视觉的内容结构

## 7.3.2　“视觉去噪”Java 实现

Cobra(http://lobobrowser.org/cobra)提供了纯 Java 实现的浏览器。Cobra 是一个 HTML 网页渲染引擎，支持 CCS2 和 JavaScript。Cobra 的一些特性如下：

- 实现了 W3C 的 HTML DOM Level 2 接口。
- 像浏览器一样解析现实的网页。

- 可以在没有头(body)的模式中使用。
- 当解析文档时，提供 DOM 修改的增量的通知。
- 可以通过设置元素的 innerHTML 属性等方法增量式地修改 DOM。
- 会识别 JavaScript。解析过程中对 DOM 的修改(例如，通过 document.write 修改网页内容)能够在结果 DOM 中反映出来，另外也可以禁用 JavaScript。

  通过 DocumentBuilderImpl 类来解析网页，如下所示：

```
URL urlObj = new URL(url); //输入url 地址
URLConnection connection = urlObj.openConnection();
InputStream in = connection.getInputStream();

UserAgentContext context = new SimpleUserAgentContext();
DocumentBuilderImpl dbi = new DocumentBuilderImpl(context);
Document document = dbi.parse(new InputSourceImpl(in, url,
        "UTF-8"));

Element ex = document.getDocumentElement();
doTree(ex);//从根节点递归遍历 DOM 树
```

递归遍历访问 DOM 树的 doTree 方法的实现如下：

```
public void doTree(Node node) {
    if (node instanceof Element) {//标签
        Element element = (Element) node;

        //访问标签
        doElement(element);

        //访问这个标签下所有的孩子，也就是这个标签所包括的标签
        NodeList nl = element.getChildNodes();
        if (nl == null)
            return;
        int num = nl.getLength();
        for (int i = 0; i < num; i++)
            doTree(nl.item(i));

        //标签结束
        doTagEnd(element);
    }
    else //非标签节点，例如文本
    {
        System.out.println(node.getNodeName()+":"+node.getNodeValue());
    }
}
```

在显示过程中，DOM 树中的标准的节点封装成了 ModelNode 对象。HtmlBlockPanel 的 paint(Graphics g)方法可以得到渲染的节点大小。

```
Rectangle clipBounds = g.getClipBounds();
g.setColor(this.getBackground());
g.fillRect(clipBounds.x, clipBounds.y, clipBounds.width,
clipBounds.height);
```

取得位置信息后，就可以开始实现 VIPS 算法了。定义内容块类型的代码如下：

```
public enum BlockType {
    Link, //链接类型
    Text, //文本类型
    Img, //图片类型
    Form //表单类型
}
```

分割条类的代码如下：

```
public class Splitter {
    // 描述该分割条的左上角的坐标
    public long left;
    public long top;
    public long right;
    public long buttom;

    public long width;
    public long height;
    // 当前分割条的权重，权重越大，成为真正分割条的可能性也就越大
    public int weight;
    //是水平分割条还是垂直分割条，true 表示垂直；false 表示水平
    public boolean vertical;

    public NodePool leftUpBlock;//对于水平分割条，保存该分割条左侧的 block
    // 对于垂直分割条，保存该分割条上面的 block
    public NodePool rightButtomBlock;
    // 对于水平分割条，保存该分割条右侧的 block
    // 对于垂直分割条，保存该分割条下侧的 block

    // 该标记用于判断当前分隔条是否是显式分隔条
    // 比如 HR，空的 Table，TR，TD 以及宽度和高度高于 20 的分割条都属于显式分隔条
    // 其余的分隔条属于隐式分隔条
    public boolean isExplicit;
}
```

VisionBlock 是 VIPS 算法中最重要的类。它用以描述最终的视觉内容块。最初的整个页面属于一个大的语义块。进行第一次迭代后，该语义块被分割为多个新的小的语义块，因此语义块的结构应该是自包含的。

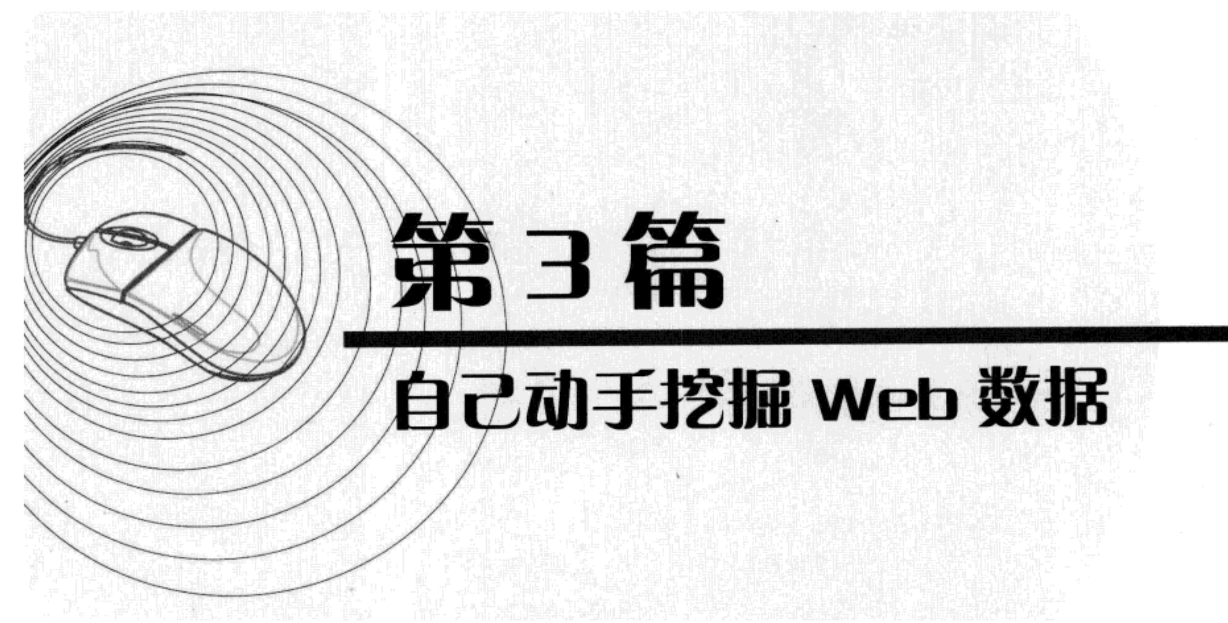
```
public class VisionBlock {
    // 位置信息
    public long blockLeft; // 当前内容块的左边缘位置
```

```
    public long blockTop;        // 当前内容块的上边缘位置
    public long blockRight;      // 当前内容块的右边缘位置
    public long blockButtom;     // 当前内容块的下边缘位置
    public long blockWidth;      // 当前内容块的宽度
    public long blockHeight;     // 当前内容块的高度

    // 分割条信息
    public Splitter upSplitter;
    public Splitter downSplitter;
    public Splitter leftSplitter;
    public Splitter rightSplitter;

    // 邻居信息
    private ArrayList<CHTMLNode> leftVisionBlock;   // 当前内容块的左侧语义块
    private ArrayList<CHTMLNode> topVisionBlock;    // 当前内容块的上侧语义块
    private ArrayList<CHTMLNode> rightVisionBlock;  // 当前内容块的右侧语义块
    private ArrayList<CHTMLNode> buttomVisionBlock; // 当前内容块的下侧语义块

    // 结构信息
    //该 Block 内部包含的 Block
    private ArrayList<VisionBlock> containedVisionBlockList;
    //当前内容块所包含的节点
    private ArrayList<CHTMLNode> containedHTMLNodeList;

    // 当前 Block 中包含的节点分隔条
    public ArrayList<CHTMLNode> nodeSplitterList;

    // 当前逻辑块的 doc 值，用以描述逻辑块内部的关联程度的强弱
    // DOC 用于描述当前 VisionBlock 内部的关联性，DOC 的值越大，关联性越大
    // DOC 的值越小，关联性越小。DOC 的值介于 1~10 之间
    public int DOC;
    private String blockName;

    // 当前 Block 的类型
    public BlockType blockType;

    // 当前语义块的父亲语义块
    public VisionBlock parentBlock;

    // 内部的 Block 是垂直 Block 还是水平 Block
    public int blockDirection;
    public int divideDirection;// 当前 block 被分割的方向
    // 判断当前的 Block 是否是文本 Block
    public int isTextBlock;
    // 子节点的数目
    public int childNums;
}
```

## 7.4 本 章 小 结

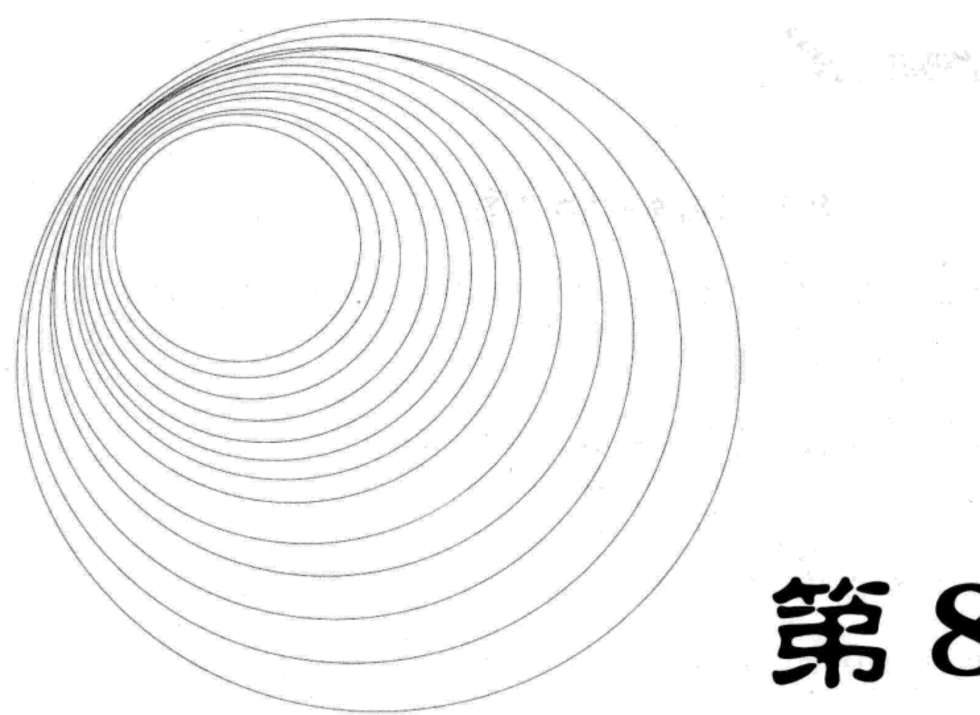
本章主要介绍了利用“统计学”方法和基于视觉的方法去除网页噪声的方法与实现。此外，在实际开发爬虫时，还可以和网站开发者建立合作关系，根据约定的格式直接获取有效信息。

PDG



# 第3篇

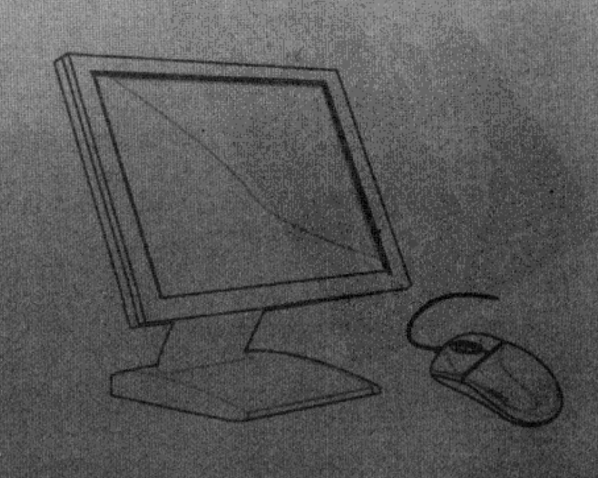
## 自己动手挖掘 Web 数据



# 第8章

## 分析Web图

在第1章我们就讲过，互联网的本质是一些超链接形成的Web图。在用网络爬虫进行处理的时候，需要把这个图放入内存中。比如，在分析链接的时候，我们需要知道每个URL所表示的网页中有多少链接，都是哪些链接。或者，针对一个网页，有哪些链接是指向它的。指向网页的链接称为网页的“入度”，而网页指向的链接(即网页中的超链接)称为网页的“出度”。这一章将通过分析Web图的存储，以及利用Web图对链接进行分析向读者展示实际应用中如何处理和利用互联网这张“超图”。



## 8.1 存储 Web“图”

对于小型的爬虫应用，存储链接之间关系的时候，可以把它们放在内存中。而对于大型爬虫而言，必须在内存数据库中建立这些链接关系。

下面是一段在内存中存储 Web 图的代码。

```
public class WebGraphMemory {
        //把每个 URL 映射为一个整数，存储在 Web 图中
    private Map<Integer,String> IdentifyerToURL;

    /** 存储 Web 图的 Hash 表 */
    private Map<String,Map<String,Integer>> URLToIdentifyer;

    /**
     *  存储入度,其中整数的第一个参数是 URL 的 ID,第二个参数存放指向这个 URL 链接的 Map,
     * Double 表示权重
     */
    private Map<Integer,Map<Integer,Double>> InLinks;

    /**
     * 存储出度，其中第一个参数是 URL 的 ID，第二个参数存放网页中的超链接，Double 表示
     * 权重
     */
    private Map<Integer,Map<Integer,Double>> OutLinks;
    /** 图中节点的数目*/
    private int nodeCount;
    /**
     * 构造函数，0 个节点的构造函数
     */
    public WebGraphMemory () {
        IdentifyerToURL = new HashMap<Integer,String>();
        URLToIdentifyer = new HashMap<String,Map<String,Integer>>();
        InLinks = new HashMap<Integer,Map<Integer,Double>>();
        OutLinks = new HashMap<Integer,Map<Integer,Double>>();
        nodeCount = 0;
    }

    /**
     * 从一个文本文件中取得节点的构造函数。 每行包含一个指向关系。例如:
     *    http://url1.com -> http://url2.com 1.0
     * 表示“http://url1.com”包含一个超链接“http://url2.com”，并且这个超链接的
     *权重是 1.0
     */
    public WebGraphMemory (File file) throws IOException,
FileNotFoundException {
```

```
    this();
    BufferedReader reader = new BufferedReader(new FileReader(file));
    String line;
    while((line=reader.readLine())!=null) {
        int index1 = line.indexOf("->");
        if(index1==-1) addLink(line.trim()); else {
            String url1 = line.substring(0,index1).trim();
            String url2 = line.substring(index1+2).trim();
            Double strength = new Double(1.0);
            index1 = url2.indexOf(" ");
            if(index1!=-1) try {
                strength = new Double(url2.substring(index1+1).trim());
                url2 = url2.substring(0,index1).trim();
            } catch ( Exception e ) {}
            addLink (url1,url2,strength);
        }
    }
}

/**
 * 根据 URL 制订它的 ID
 */
public Integer URLToIdentifyer ( String URL ) {
    String host;
    String name;
    int index = 0 , index2 = 0;
    if(URL.startsWith("http://")) index = 7;
    else if(URL.startsWith("ftp://")) index = 6;
    index2 = URL.substring(index).indexOf("/");
    if(index2!=-1) {
        name = URL.substring(index+index2+1);
        host = URL.substring(0,index+index2);
    } else {
        host = URL;
        name = "";
    }
    Map<String,Integer> map = (URLToIdentifyer.get(host));
    if(map==null)
    {
        return null;
    }
    return (map.get(name));
}

/**
 * 根据 ID 获得 URL
 */
```

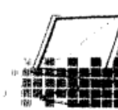
```
public String IdentifyerToURL ( Integer id ) {
    return (IdentifyerToURL.get(id));
}

/**
 *  在图中增加一个节点
 */
public Integer addLink (String link) {
    Integer id = URLToIdentifyer(link);
    if(id==null) {
        id = new Integer(++nodeCount);
        String host;
        String name;
        int index = 0 , index2 = 0;
        if(link.startsWith("http://")) index = 7;
        else if(link.startsWith("ftp://")) index = 6;
        index2 = link.substring(index).indexOf("/");
        if(index2!=-1) {
            name = link.substring(index+index2+1);
            host = link.substring(0,index+index2);
        } else {
            host = link;
            name = "";
        }
        Map<String,Integer> map = (URLToIdentifyer.get(host));
        if(map==null)
        {
            map = new HashMap<String,Integer>();
            URLToIdentifyer.put(host,map);
        }
        map.put(name,id);

        IdentifyerToURL.put(id,link);
        InLinks.put(id,new HashMap<Integer,Double>());
        OutLinks.put(id,new HashMap<Integer,Double>());
    }
    return id;
}

/**
 *在两个节点中增加一个对应关系。如果节点不存在，就新创建节点
 */
public Double addLink (String fromLink, String toLink, Double weight)
{
    Integer id1 = addLink(fromLink);
    Integer id2 = addLink(toLink);
    return addLink(id1,id2,weight);
```

```
}

/**
 *在两个节点中增加一个对应关系。如果节点不存在，就新创建节点
 */
private Double addLink ( Integer fromLink, Integer toLink, Double weight )
{
    Double aux;
    Map<Integer,Double> map1 = (InLinks.get(toLink));
    Map<Integer,Double> map2 = (OutLinks.get(fromLink));
    aux = (Double)(map1.get(fromLink));
    if(aux==null) map1.put(fromLink,weight);
    else if(aux.doubleValue()<weight.doubleValue())
    map1.put(fromLink,weight);
    else weight = new Double(aux.doubleValue());

    aux = (map2.get(toLink));
    if(aux==null) map2.put(toLink,weight);
    else if(aux.doubleValue()<weight.doubleValue())
    map2.put(toLink,weight);
    else {
        weight = new Double(aux.doubleValue());
        map1.put(fromLink,weight);
    }
    InLinks.put(toLink,map1);
    OutLinks.put(fromLink,map2);
    return weight;
}

/**
 * 针对指定的 URL 返回包含它的所有入度链接的 Map
 */
public Map inLinks ( String URL ) {
    Integer id = URLToIdentifyer(URL);
    return inLinks(id);
}

/**
 *针对指定的 URL 返回包含它的所有入度链接的 Map
 */
public Map<Integer,Double> inLinks ( Integer link ) {
    if(link==null) return null;
    Map<Integer,Double> aux = (InLinks.get(link));
    return aux;
}

/**
```

```
 *针对指定的 URL 返回包含它的出度的链接的 Map
 */
public Map<Integer,Double> outLinks ( String URL ) {
    Integer id = URLToIdentifyer(URL);
    return outLinks(id);
}

/**
 *针对指定的 URL 返回包含它的出度链接的 Map
 */
public Map<Integer,Double> outLinks ( Integer link ) {
    if(link==null) return null;
    Map<Integer,Double> aux = OutLinks.get(link);
    return aux;
}

/**
 *返回两个节点之间的权重，如果节点没有链接，就返回 0
 */
public Double inLink ( String fromLink, String toLink ) {
    Integer id1 = URLToIdentifyer(fromLink);
    Integer id2 = URLToIdentifyer(toLink);
    return inLink(id1,id2);
}

/**
 *返回两个节点之间的权重，如果节点没有链接，就返回 0
 */
public Double outLink ( String fromLink, String toLink ) {
    Integer id1 = URLToIdentifyer(fromLink);
    Integer id2 = URLToIdentifyer(toLink);
    return outLink(id1,id2);
}

/**
 *返回两个节点之间的权重，如果节点没有链接，就返回 0
 */
public Double inLink ( Integer fromLink, Integer toLink ) {
    Map<Integer,Double> aux = inLinks(toLink);
    if(aux==null) return new Double(0);
    Double weight = (aux.get(fromLink));
    return (weight == null) ? new Double(0) : weight;
}

/**
 *返回两个节点之间的权重，如果节点没有链接，就返回 0
 */
```

```
public Double outLink ( Integer fromLink, Integer toLink ) {
    Map<Integer,Double> aux = outLinks(fromLink);
    if(aux==null) return new Double(0);
    Double weight = (aux.get(toLink));
    return (weight == null) ? new Double(0) : weight;
}

/**
 * 把有向图变为无向图
 */
public void transformUnidirectional () {
    Iterator it = OutLinks.keySet().iterator();
    while (it.hasNext()) {
        Integer link1 = (Integer)(it.next());
        Map auxMap = (Map)(OutLinks.get(link1));
        Iterator it2 = auxMap.keySet().iterator();
        while (it2.hasNext()) {
            Integer link2 = (Integer)(it.next());
            Double weight = (Double)(auxMap.get(link2));
            addLink(link2,link1,weight);
        }
    }
}

/**
 *删除内部链接，内部链接就是指在同一主机上的链接
 */
public void removeInternalLinks () {
    int index1;
    Iterator it = OutLinks.keySet().iterator();
    while (it.hasNext()) {
        Integer link1 = (Integer)(it.next());
        Map<Integer,Double> auxMap = (OutLinks.get(link1));
        Iterator it2 = auxMap.keySet().iterator();
        if(it2.hasNext()) {
            String URL1 = (String)(IdentifyerToURL.get(link1));
            index1 = URL1.indexOf("://");
            if(index1!=-1) URL1=URL1.substring(index1+3);
            index1 = URL1.indexOf("/");
            if(index1!=-1) URL1=URL1.substring(0,index1);
            while (it2.hasNext()) {
                Integer link2 = (Integer)(it.next());
                String URL2 = (String)(IdentifyerToURL.get(link2));
                index1 = URL2.indexOf("://");
                if(index1!=-1) URL2=URL1.substring(index1+3);
                index1 = URL2.indexOf("/");
                if(index1!=-1) URL2=URL1.substring(0,index1);
```

```
                    if(URL1.equals(URL2)) {
                        auxMap.remove(link2);
                        OutLinks.put(link1,auxMap);
                        auxMap = (InLinks.get(link2));
                        auxMap.remove(link1);
                        InLinks.put(link2,auxMap);
                    }
                }
            }
        }
    }

    /**
     * 删除内部导航链接
     */
    public void removeNepotistic() {
        removeInternalLinks();
    }

    /**
     * 删除终止 URL
     */
    public void removeStopLinks(String stopURLs[]) {
        HashMap aux = new HashMap();
        for (int i=0; i<stopURLs.length; i++) aux.put(stopURLs[i],null);
        removeStopLinks(aux);
    }

    /**
     *删除终止 URL
     */
    public void removeStopLinks(Map stopURLs) {
        int index1;
        Iterator it = OutLinks.keySet().iterator();
        while (it.hasNext()) {
            Integer link1 = (Integer)(it.next());
            String URL1 = (String)(IdentifyerToURL.get(link1));
            index1 = URL1.indexOf("://");
            if(index1!=-1) URL1=URL1.substring(index1+3);
            index1 = URL1.indexOf("/");
            if(index1!=-1) URL1=URL1.substring(0,index1);
            if(stopURLs.containsKey(URL1)) {
                OutLinks.put(link1,new HashMap());
                InLinks.put(link1,new HashMap());
            }
        }
    }
```



```
    public int numNodes() {
        return nodeCount;
    }
}
```

前面讲过，互联网是一张超图。那么如何存储这张超图呢？在大型应用中，使用上面的内存数据结构来存储和处理显然是不行的。因此，链接数据应该存放在内存数据库或者真正的数据库中。下面代码是以 Berkeley DB 内存数据库来存储 Web 图的，有关 Berkeley DB 的知识，请参见第 1 章的内容。

```
public class WebGraph {
    //出度
    private PrimaryIndex<String,Link> outLinkIndex;
   //入度
    private SecondaryIndex<String,String,Link> inLinkIndex;

    private EntityStore store;
    /**
     * 构造函数
     */
    public WebGraph (String dbDir) throws DatabaseException {
        File envDir = new File(dbDir);
        EnvironmentConfig envConfig = new EnvironmentConfig();
       envConfig.setTransactional(false);
       envConfig.setAllowCreate(true);
       Environment env = new Environment(envDir, envConfig);

        StoreConfig storeConfig = new StoreConfig();
        storeConfig.setAllowCreate(true);
        storeConfig.setTransactional(false);

       store = new EntityStore(env, "classDb", storeConfig);
       outLinkIndex = store.getPrimaryIndex(String.class, Link.class);
       inLinkIndex = store.getSecondaryIndex(outLinkIndex, String.class,
       "toURL");
    }

    /**
     *构造 Web 图，从文件内读入。每一行为一个对应关系，例如
     *    http://url1.com -> http://url2.com 1.0
     * 表示在链接 http://url1.com 所表示的网页上面，有一个超链接 http://url2.com
     *并且它们之间的权重为 1.0
     */
    public void load (File file) throws IOException, FileNotFoundException,
DatabaseException {
        BufferedReader reader = new BufferedReader(new FileReader(file));
```

```
        String line;
        while((line=reader.readLine())!=null) {
            int index1 = line.indexOf("->");
            if(index1==-1)
            {
                continue;
            }
            else {
                String url1 = line.substring(0,index1).trim();
                String url2 = line.substring(index1+2).trim();
                index1 = url2.indexOf(" ");
                if(index1!=-1) try {
                    url2 = url2.substring(0,index1).trim();
                } catch ( Exception e ) {}
                addLink (url1,url2);
            }
        }
    }

    /**
    * 加入节点之间的对应关系，如果节点不存在就创建，如果已经存在对应关系，就更新权重
     */
    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);
        }
    }
   //根据制定的URL，获得指向它的入度链接
    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;
    }
}
```

## 8.2　利用 Web“图”分析链接

上一节讲述了如何存储互联网这张巨大的 Web 图。那么，存储下来的 Web 图到底有何意义呢？

对于网络爬虫而言，存储的 Web 图中包括每张网页的出度和入度的信息。运用这些信息能够进行链接分析。比如，根据链接的出度和入度的相关信息，可以对链接进行排序。在后期检索的时候，可以把这种排序的结果作为检索结果呈现给用户。

链接分析的方法有很多种，比较流行的也是比较基础的两种方法是 PageRank 算法和 HITS 算法。在下面的两节中，我们将分别介绍这两种算法。

## 8.3　Google 的秘密——PageRank

在过去几年中，Google 成为全世界使用最多的搜索引擎。与其他搜索引擎相比，除高性能和高易用性以外，一个决定性的因素是它的优秀的搜索结果。搜索结果的高质量来源于 PageRank——一个精密的排序网页文件等级的方式。

### 8.3.1　深入理解 PageRank 算法

从万维网的早期，搜索引擎都在开发不同的排序网页的方法。但实际上，直到今天，所有搜索引擎对网页的排序，都是根据搜索的词组短语在页面中出现的次数，并用页面长度和 html 标签的重要性提示等进行权重修订。

为了得到更好的搜索结果，尤其是使搜索引擎自动抵制那些基于对详细等级标准页面(入口页)内容的分析而自动生成的网页，出现了链接人气值的概念。根据这个概念，一个网页文件的入链数量通常表示此文件的重要程度。因此，一般的，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。采用链接人气值的概念通常可以避免使那些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提高网站在搜索引擎中的排名，会从其他没有意义的网页创建大量入站链接，从而使网站的链接人气升高。

与链接人气值相比较，PageRank 算法并不是简单地根据入站链接的总数来计算排名。PageRank 基本的原理是，一个网页的重要程度依赖与它的入链，如果高等级的文件链接到网页，那么根据 PageRank 的规则，此网页的等级也高。

如此，在 PageRank 概念中，文件的等级由与它链接的文件的等级决定，而所链接的这些文件的等级再由与它们链接文件的等级决定。因此，一个文件的 PageRank 由其他文件的

PageRank 总递归之和确定。概言之，PageRank 的等级由整个网的链接结构决定。虽然这种方法似乎非常宽泛和复杂，但 Page 和 Brin 已经能够通过一个简单的运算法则将它付诸实践了。

最初的 PageRank 算法是：

$$PR(A) = (1-d) + d\,(PR(T1)/C(T1) + \ldots + PR(Tn)/C(Tn))$$

式中：

PR(A)——网页 A 的 PageRank 值；

PR(Ti)——链接到 A 的 Ti 网页的 PageRank 值；

C(Ti)——网页 Ti 的出站链接数量；

d——阻尼系数，$0<d<1$。

可见，首先 PageRank 并不是为整个网站排级，而是以单个页面计算的。其次，页面 A 的 PageRank 值取决于那些链接到 A 页面的网页的 PageRank 的递归值。

PR(Ti)值并不是均等影响页面 PR(A)的。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 这个链接的影响就越少。

PR(A)由所有它的入链接的 PR(Ti)与 C(Ti)组成。之所以选择 P(Ti)/C(Ti)的形式，就是为了保证重要的网页(也就是 PageRank 值比较高，但是出站链接相对较少的网页)所指向的页面，比非重要的网页指向的页面 PageRank 值要高。对于页面 A 而言，每多增加一个入链接都能增加 PR(A)的值。因为每增加一个入链接，PR(A)的组成中就会增加一项 PR(Ti)/C(Ti)。而新添的这个入链接的 PangRank 值越大，PR(A)增加的也就越多。

在 PR(A)的组成中，所有 PR(Ti)/C(Ti)之和还要乘以一个阻尼系数 d，它的值在 0 到 1 之间。阻尼系数的使用，可以调节其他页面对当前页面 A 的排序贡献。

PageRank 的特性如图 8.1 所示。

假设一个小网站由三个页面 A、B、C 组成，A 链接到 B 和 C，B 链接到 C，C 链接到 A。Page 和 Brin 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 的精确值无疑会影响到 PageRank 值，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：

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

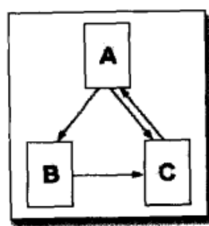
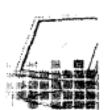



图 8.1　PageRank 的特性



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

很明显所有页面的 PageRank 值的和为 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，如表 8.1 所示。

表 8.1　PageRank 迭代计算方法

| 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 |
| 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 |

实际计算中，通常需要上百次计算才能得到 PageRank 的值。这里只给出了 12 次迭代计算作为示例。

关于 PageRank 的实现，重要的是网页的 PageRank 值怎样被 Google 综合考虑进网页的排序。最初，Google 搜索引擎对于网页的排序由三个因素决定：

- 页面的特定因素
- 入链锚的文字内容
- PageRank

页面的特定因素值是指：网页内容、标题内容和文档的 URL。根据 Page 和 Brin 公开发表的文章，很有可能会有更多的因素影响 Google 的排序方式。但是这里我们不讨论那些因素。

Google 根据页面的特定因素和入链锚的文字内容计算出网页的 IR 值，确定文档和搜索

语句的相关性。然后此 IR 值结合 PageRank 值表示网页的重要程度。为了结合 IR 值和 PageRank 值，这两个值被相乘。很明显不可能是相加的，否则的话，如果页面拥有一个很高的 PageRank 值，即使和搜索语句无关，也会在搜索结果中排在前面。

对于两条或更多的关键词所构成的搜索语句，IR 值对于评级标准的影响更大；相反的，PageRank 主要会对用单个词作为搜索语句时的评级造成显著的影响。

多数人是通过 Google 工具栏开始了解 PageRank 的。Google 工具栏是 Microsoft Internet Explorer 的一个浏览器插件，可以在 Google 网站下载。图 8.2 是 Google 搜集工具栏的 PageRank 值。

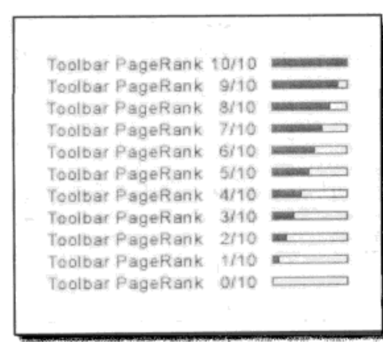



图 8.2 Google 工具栏的 PageRank 值

Google 工具栏以 0～10 的刻度显示 PageRank 值。从显示栏中绿色部分的长度可以估计出当前正在访问的页面的 PageRank 值。如果用户将鼠标放于显示栏上，就是显示 PageRank 值。

注意：显示 PageRank 值是 Google 工具栏的一个高级功能。如果高级功能被打开，Google 就会收集用户信息。另外，工具栏会自动升级，用户无需关心。所以，Google 可以访问用户的硬盘。

如果我们进行一下计算，PageRank 理论上拥有最大值($Nd+(1-d)$)，这里的 $N$ 为互联网网页总数，$d$ 通常被设为 0.85，计算出的 PageRank 值和工具栏显示出的数值成一定的比例。普遍认同的是，它们之间的比例并非线性的，而是成对数关系。如果设阻尼系数 $d$ 为 0.85，而 PageRank 的最低值为 0.15，并且对数的基数为 6，则可以得到如表 8.2 所示的比例关系。

表 8.2 搜索工具栏显示的 PageRank 值与实际 PageRank 值比较

| 工具栏 PageRank | 阻尼系数为 0.15 时的对应值 | 实际 PageRank |
|---|---|---|
| 0/10 | 0.15 | 0.9 |
| 1/10 | 0.9 | 5.4 |
| 2/10 | 5.4 | 32.4 |
| 3/10 | 32.4 | 194.4 |
| 4/10 | 194.4 | 1 166.4 |

续表

| 工具栏 PageRank | 阻尼系数为 0.15 时的对应值 | 实际 PageRank |
|---|---|---|
| 5/10 | 1 166.4 | 6 998.4 |
| 6/10 | 6 998.4 | 41 990.4 |
| 7/10 | 41 990.4 | 251 942.4 |
| 8/10 | 251 942.4 | 1 511 654.4 |
| 9/10 | 1 511 654.4 | 9 069 926.4 |
| 10/10 | 9 069 926.4 | $0.85 \times N + 0.15$ |

## 8.3.2　PageRank 算法的 Java 实现

本节的 PageRank 算法使用了一种名为 BigMatrix 的数据结构来进行迭代计算。

```
public class Google {
    private double[] rank;
    Hashtable<String,Integer> hashedPages;
    String[] sortedRank;
    public Google() {
    }
    private void rankFilter(BigMatrix dataMatrix) {
        String[] tempRank = new String[sortedRank.length];
        Boolean isEqual = true;
        //迭代计算，直到数据收敛或者次数达到 50 次
        for (int i = 0; i < 50; i++) {
            rank = dataMatrix.multiply(rank);
            // 拷贝当前的数组值到临时数组
            for (int j = 0; j < sortedRank.length; j++) {
                tempRank[j] = sortedRank[j];
            }
            //排序
            Arrays.sort(sortedRank, new compareByRank());
            // 计算是否收敛
            for (int j = 0; j < sortedRank.length; j++) {
                if (sortedRank[j].compareTo(tempRank[j]) != 0) {
                    isEqual = false;
                    break;
                }
            }

            if (isEqual == true) {
                break;
            } else {
                isEqual = true;
            }
        }
    }
```

```
class compareByRank implements Comparator<String> {
    public int compare(String a, String b) {
        int indexA = hashedPages.get(a);
        int indexB = hashedPages.get(b);
        if (rank[indexA] == rank[indexB]) {
            return(0);
        } else if (rank[indexA] > rank[indexB]) {
            return(-1);
        } else {
            return(1);
        }
    }
}

public java.lang.String[] pageRank(java.lang.String[] s) {
    // height of data
    int theSize = Math.max(4 * s.length/3 + 1, 16);

    // 初始化
    hashedPages = new Hashtable<String,Integer>(theSize);
    String[] pages = new String[s.length];
    int[] nLinks = new int[s.length];
    rank = new double[s.length];
    sortedRank = new String[s.length];
    String[] dataEntry = new String[s.length];

    // 获取数据
    for (int i = 0; i < s.length; i++) {
        String[] temp = s[i].split(" ");
        pages[i] = temp[0];
        nLinks[i] = temp.length - 1;
        sortedRank[i] = temp[0];
        rank[i] = 1;
        dataEntry[i] = "";
        hashedPages.put(pages[i], i);
    }

    int tRow, tCol;
    //初始化矩阵
    for (int i = 0; i < s.length; i++) {
        String[] temp = s[i].split(" ");
        for (int j = 1; j < temp.length; j++) {
            tCol = hashedPages.get(temp[0]);
            tRow = hashedPages.get(temp[j]);
            dataEntry[tRow] += "{" + tCol + "," + (1 / (double)nLinks[i]) + "};";
        }
    }
    // 创建矩阵数据
```

```
        BigMatrix dataMatrix = new BigMatrix(dataEntry);
        // 排序
        rankFilter(dataMatrix);
        //返回排序后的URL列表
        return(sortedRank);
    }
}

//矩阵
class BigMatrix {
    public int nCols, nRows;
    EntryList[] theRows;
//构造函数采用String的数组作为输入，例如{"(1,1); (4,3); (5,8)", "(2,5);
//(3,4)","(3,8);(4,5)"}
//每个字符串能够初始化一行数据。例如，(2, 5)表示在第二行的第二列值为5
    public BigMatrix(java.lang.String[] x) {
        nRows = x.length;
        nCols = 0;
        theRows = new EntryList[nRows];
        for (int i = 0; i < nRows; i++) {
           theRows[i] = new EntryList();
           if (x[i] != null) {
               String[] tempArr = x[i].split(";");
               if (tempArr[0] != null) {
                   for (int j = 0; j < tempArr.length; j++) {
                       Entry instance = new Entry(tempArr[j]);
                       theRows[i].add(instance);
                       if (nCols <= instance.col) {
                           nCols = instance.col + 1;
                       }
                   }
               }
           }
        }
    }

    //乘以1维向量
    public double[] multiply(double[] x) {
        double[] result = new double[nRows];
        int count;
        for (int i = 0; i < nRows; i++) {
            EntryList temp = theRows[i];
            count = 0;
            while ((temp != null) && (temp.data != null)) {
                result[i]+= (temp.data.value * x[temp.data.col]);
                temp = temp.next;
                count++;
            }
        }
```

```
            return(result);
        }
    }
    //矩阵的元素
    class Entry {
        int col;//元素所在列
        double value;//元素值
        public Entry(java.lang.String x) {
            String[] temp = x.split(",");
            if (temp[0].compareTo("") != 0) {
                col = Integer.parseInt(temp[0].trim().substring(1));
                value = Double.parseDouble(temp[1].trim().substring(0,
temp[1].trim().length() -1));
            }
        }
    }
    //元素列表，对行进行建模
    class EntryList {
        Entry data;
        EntryList next, tail;
        public EntryList() {
            next = null;
            tail = null;
            data = null;
        }
       //添加数据
        void add(Entry x) {
            if (tail == null) {
                data = x;
                tail = this;
            } else {
                tail.next = new EntryList();
                tail.next.data = x;
                tail = tail.next;
            }
        }
    }
```

### 8.3.3 应用 PageRank 进行链接分析

在前文中我们讲解了链接分析的概念，链接分析从本质上讲，就是在 Web 图中，计算网页的重要性。因此，凡是能够计算出网页重要性的算法，都能够应用于链接分析。PageRank也不例外。

在执行 PageRank 算法的时候，需要用到网页的出度和入度以及网页之间的关系。这些数据是如何得到的呢？是由爬虫抓取网页时，把网页中的出度记录下来，并且去更新这些超链接所对应的入度。因此，整体的链接分析过程如图 8.3 所示。

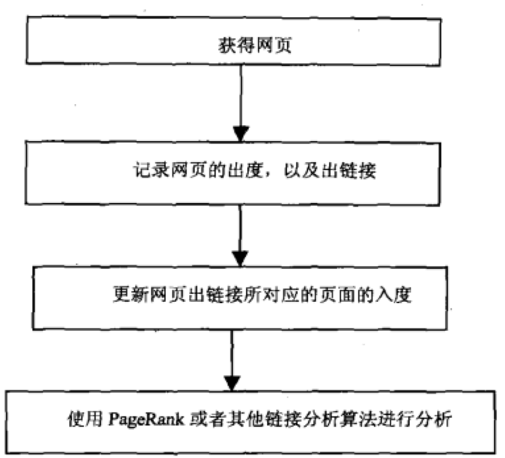



图 8.3 链接分析过程

前面讲的 PageRank 算法，是理论上的算法，也是比较基础的算法。但是，由于互联网的复杂性，往往要对 PageRank 做一定的修改才能用于链接分析。

## 8.4 PageRank 的兄弟 HITS

HITS 算法是由康奈尔大学(Cornell University)的 Jon Kleinberg 博士于 1998 年首先提出的，HITS 的英文全称为 Hypertext-Induced Topic Search。目前，它是 IBM 公司阿尔马登研究中心(IBM Almaden Research Center)的名为 CLEVER 的研究项目中的一部分。

### 8.4.1 深入理解 HITS 算法

Kleinberg 将每个网页赋两个值，即 hub 和 authority。网页的集合组成 hubs 和 authorities 向量。HITS 算法的目标就是通过一定的计算方法(迭代计算)以得到针对某个检索提问的最有价值的网页，即排名最高的 authorities。

为便于理解，Kleinberg 用图来表示链接关系，他认为超链页面的集合 V 为一个有向图 GE(V, E)，图中的每个节点对应一个网页，有向边(p, q)∈E 表示网页 p 链接指向网页 q，节点 p 的出度(out-degree)指节点 p 链出的网页数量，而节点 p 的入度(in-degree)则指的是指向节点 p 的链接数量。如果集合 W 是 V 的一个子集(即 W 包含 V 中的一部分节点)，则用 G[W]来表示由 W 中的节点组成的有向图，而有向图边对应于 W 中的所有链接。现在假设给定一个泛指主题检索提问$\sigma$，需要通过链接分析确定该提问的权威页。最先是确定 HITS 算法作用的 WWW 子集。理想地，Kleinberg 希望得到的集合 $S_\sigma$具有以下特点：

- $S_\sigma$相对较小；
- $S_\sigma$中相关网页丰富；
- $S_\sigma$包含多数最有价值的 authorities 页面。

针对具体的检索提问，构建关于该提问的 WWW 聚集子图的具体做法如下：

(1) 用基于文本的搜索引擎如 AltaVista 或 Hotbot 来得到σ的查询结果集，取排名最高的前 t(t 值通常设为 200)个结果的结果集 $R_σ$(称为 RootSet)，Kleinberg 认为 $R_σ$满足特点 a 和 b，但远不能满足 c，因此需要扩充 $R_σ$。

(2) 扩充 $R_σ$分为两个方面，一是将所有 $R_σ$中页面所指向的页面扩充进去，该扩充在数量上没有限制；二是将指向 $R_σ$中的每一页面的链接页面取其中任意 d(d 值通常设定为 50，如果 d 不大于 50，则取其所有页面)个页面扩充到原来的 $R_σ$中形成 $S_σ$ (称为 BaseSet)。通过实验表明，这样的集合 $S_σ$能够较好地满足上述三个特点，$S_σ$的数量范围一般为 1000～5000。

(3) 为了排除干扰，提高计算效果，Kleinberg 还将 $S_σ$作了进一步的处理，他将链接分为两种情况：一是指有链接关系的两个页面处在不同域名之间，这样的链接称为横向链接；还有一种情况是指有链接关系的两个页面处于同一域名之下，这样的链接称为内在链接。Kleinberg 认为内在链接只具有网站内部的导航功能，它几乎不能传递网页间的 authority，因此需要将这种内在链接从 $S_σ$中删去，形成 $G_σ$。

Kleinberg 认为 hubs 和 authority 是相互增强的关系。一个具有较高 hub 值的页指向许多具有较高 authority 值的页，在许多实例中，它们之间是一种环状的关系，因此需要一定的计算方法来打破这种环状结构。对于每一个页面 p，用 x<p>表示页面 p 的权威权重(authority weight)，用 y<p>表示页面 p 的中心权重(hub weight)，满足规范化条件：$\sum p \in s_σ$(x<p>)2E1 且 Σ p∈$s_σ$(y<p>)2E1。Kleinberg 将网页权重的传递分为两种方式，即 I 操作和 O 操作。I 操作为 hub 到 authority 的传递，表示为 x<p>← Σ q:(q,p)∈Ey<q>；O 操作为 authority 到 hub 的传递，表示为 y<p>← Σ q:(q,p)∈Ex<q>。预先设定迭代次数 k，算法表示如下：

```
Iterate(G,k)
G: a collection of n linked pages
k: a natural number
Let z denote the vector(1,1,1,...,1) • Rn.
Set x0: Ez.
Set y0: Ez.
For I E1,2,…,k
    Apply the I operation to(xi.1,yi.1),obtaining new x-weights x-i.
    Apply the O operation to(x-i,yi.1),obtaining new y-weights y.
    Normalize x-i,obtaining xi.
    Normalize y-i,obtaining yi.
End
Return(xk,yk).
```

根据矩阵计算知识容易证明，对于给定一个初始向量 x0 和 y0，迭代过程收敛，其最终结果 x*为 ATA 的主特征向量，y*为 AAT 的主特征向量。

### 8.4.2 HITS 算法的 Java 实现

根据以上的分析，本小节我们给出 HITS 的算法代码。

```
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 的分值
 */
public void computeHITS() {
    computeHITS(25);
}

/**
 *计算网页的 Hub 和 Authority scores
 *
 */
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);
    }
}

/**
 * 归一化数据集
 */
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 评分
 */
public Double hubScore(String link) {
    return hubScore(graph.URLToIdentifyer(link));
}

/**
 *返回与给定链接关联的 Hub 评分

*/
private Double hubScore(Integer id) {
    return (Double)(hubScores.get(id));
}

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

/**
 *初始化与给定链接关联的 Hub 评分
 */
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 评分
 */
public Double authorityScore(String link) {
    return authorityScore(graph.URLToIdentifyer(link));
```

```
    }

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

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

}
```

以下是存储 Web 图的数据结构。

```
public class WebGraphMemory {
        //把每个 URL 映射为一个整数，存储在 Web 图中
    private Map<Integer,String> IdentifyerToURL;

    /**存储 Web 图中 URL 之间关系的 MAP*/
    private Map<String,Map<String,Integer>> URLToIdentifyer;

    /**
     *  存储入度，其中第一个参数是 URL 的 ID，第二个参数存放指向这个 URL 链接的 Map，
     *  Double 表示权重
     */
    private Map<Integer,Map<Integer,Double>> InLinks;

    /**
     * 存储出度，其中第一个参数是 URL 的 ID，第二个参数存放网页中的超链接，Double 表示
     * 权重
     */
    private Map<Integer,Map<Integer,Double>> OutLinks;
    /** 图中节点的数目*/
    private int nodeCount;
```

```
/**
 * 构造函数，0 个节点的构造函数
 */
public WebGraphMemory () {
    IdentifyerToURL = new HashMap<Integer,String>();
    URLToIdentifyer = new HashMap<String,Map<String,Integer>>();
    InLinks = new HashMap<Integer,Map<Integer,Double>>();
    OutLinks = new HashMap<Integer,Map<Integer,Double>>();
    nodeCount = 0;
}

/**
 * 从一个文本文件中取得节点的构造函数。 每行包含一个指向关系。例如：
 *   http://url1.com -> http://url2.com 1.0
 * 表示“http://url1.com”包含一个超链接 http://url2.com，并且这个超链接
 * 的权重是 1.0
 */
public WebGraphMemory (File file) throws IOException,
FileNotFoundException {
    this();
    BufferedReader reader = new BufferedReader(new FileReader(file));
    String line;
    while((line=reader.readLine())!=null) {
        int index1 = line.indexOf("->");
        if(index1==-1) addLink(line.trim()); else {
            String url1 = line.substring(0,index1).trim();
            String url2 = line.substring(index1+2).trim();
            Double strength = new Double(1.0);
            index1 = url2.indexOf(" ");
            if(index1!=-1) try {
                strength = new Double(url2.substring(index1+1).trim());
                url2 = url2.substring(0,index1).trim();
            } catch ( Exception e ) {}
            addLink (url1,url2,strength);
        }
    }
}

/**
 * 根据 URL 制订它的 ID
 */
public Integer URLToIdentifyer ( String URL ) {
    String host;
    String name;
    int index = 0 , index2 = 0;
    if(URL.startsWith("http://")) index = 7;
    else if(URL.startsWith("ftp://")) index = 6;
```

```
        index2 = URL.substring(index).indexOf("/");
        if(index2!=-1) {
            name = URL.substring(index+index2+1);
            host = URL.substring(0,index+index2);
        } else {
            host = URL;
            name = "";
        }
        Map<String,Integer> map = (URLToIdentifyer.get(host));
        if(map==null)
        {
            return null;
        }
        return (map.get(name));
    }

    /**
     * 根据 ID 获得 URL
     */
    public String IdentifyerToURL ( Integer id ) {
        return (IdentifyerToURL.get(id));
    }

    /**
     *  在图中增加一个节点
     */
    public Integer addLink (String link) {
        Integer id = URLToIdentifyer(link);
        if(id==null) {
            id = new Integer(++nodeCount);
            String host;
            String name;
            int index = 0 , index2 = 0;
            if(link.startsWith("http://")) index = 7;
            else if(link.startsWith("ftp://")) index = 6;
            index2 = link.substring(index).indexOf("/");
            if(index2!=-1) {
                name = link.substring(index+index2+1);
                host = link.substring(0,index+index2);
            } else {
                host = link;
                name = "";
            }
            //System.out.println("HOST:"+host + " name:"+name);
            Map<String,Integer> map = (URLToIdentifyer.get(host));
            if(map==null)
            {
```

```
            map = new HashMap<String,Integer>();
            URLToIdentifyer.put(host,map);
        }
        map.put(name,id);

        IdentifyerToURL.put(id,link);
        InLinks.put(id,new HashMap<Integer,Double>());
        OutLinks.put(id,new HashMap<Integer,Double>());
    }
    return id;
}

/**
 *在两个节点中增加一个对应关系。如果节点不存在，就新建节点
 */
public Double addLink (String fromLink, String toLink, Double weight) {
    Integer id1 = addLink(fromLink);
    Integer id2 = addLink(toLink);
    return addLink(id1,id2,weight);
}

/**
 *在两个节点中增加一个对应关系。如果节点不存在，就新建节点
 */
private Double addLink ( Integer fromLink, Integer toLink, Double weight ) {
    Double aux;
    Map<Integer,Double> map1 = (InLinks.get(toLink));
    Map<Integer,Double> map2 = (OutLinks.get(fromLink));
    aux = (Double)(map1.get(fromLink));
    if(aux==null) map1.put(fromLink,weight);
    else if(aux.doubleValue()<weight.doubleValue()) map1.put
    (fromLink,weight);
    else weight = new Double(aux.doubleValue());

    aux = (map2.get(toLink));
    if(aux==null) map2.put(toLink,weight);
    else if(aux.doubleValue()<weight.doubleValue()) map2.put
    (toLink,weight);
    else {
        weight = new Double(aux.doubleValue());
        map1.put(fromLink,weight);
    }
    InLinks.put(toLink,map1);
    OutLinks.put(fromLink,map2);
    return weight;
}
```

```
/**
 * 针对指定的 URL 返回包含它的入度的链接的 Map
 */
public Map inLinks ( String URL ) {
    Integer id = URLToIdentifyer(URL);
    return inLinks(id);
}

/**
 *针对指定的 URL 返回包含它的入度的链接的 Map
 */
public Map<Integer,Double> inLinks ( Integer link ) {
    if(link==null) return null;
    Map<Integer,Double> aux = (InLinks.get(link));
    return aux;
}

/**
 *针对指定的 URL 返回包含它的出度的链接的 Map
 */
public Map<Integer,Double> outLinks ( String URL ) {
    Integer id = URLToIdentifyer(URL);
    return outLinks(id);
}

/**
 *针对指定的 URL 返回包含它的出度的链接的 Map
 */
public Map<Integer,Double> outLinks ( Integer link ) {
    if(link==null) return null;
    Map<Integer,Double> aux = OutLinks.get(link);
    return aux;
}

/**
 *返回两个节点之间的权重，如果节点没有链接，就返回 0
 */
public Double inLink ( String fromLink, String toLink ) {
    Integer id1 = URLToIdentifyer(fromLink);
    Integer id2 = URLToIdentifyer(toLink);
    return inLink(id1,id2);
}

/**
 *返回两个节点之间的权重，如果节点没有链接，就返回 0
 */
public Double outLink ( String fromLink, String toLink ) {
```

```
    Integer id1 = URLToIdentifyer(fromLink);
    Integer id2 = URLToIdentifyer(toLink);
    return outLink(id1,id2);
}

/**
 *返回两个节点之间的权重，如果节点没有链接，就返回 0
 */
public Double inLink ( Integer fromLink, Integer toLink ) {
    Map<Integer,Double> aux = inLinks(toLink);
    if(aux==null) return new Double(0);
    Double weight = (aux.get(fromLink));
    return (weight == null) ? new Double(0) : weight;
}

/**
 *返回两个节点之间的权重，如果节点没有链接，就返回 0
 */
public Double outLink ( Integer fromLink, Integer toLink ) {
    Map<Integer,Double> aux = outLinks(fromLink);
    if(aux==null) return new Double(0);
    Double weight = (aux.get(toLink));
    return (weight == null) ? new Double(0) : weight;
}

/**
 * 把有向图变为无向图
 */
public void transformUnidirectional () {
    Iterator it = OutLinks.keySet().iterator();
    while (it.hasNext()) {
        Integer link1 = (Integer)(it.next());
        Map auxMap = (Map)(OutLinks.get(link1));
        Iterator it2 = auxMap.keySet().iterator();
        while (it2.hasNext()) {
            Integer link2 = (Integer)(it.next());
            Double weight = (Double)(auxMap.get(link2));
            addLink(link2,link1,weight);
        }
    }
}

/**
 *删除内部链接，内部链接就是指在同一主机上的链接
 */
public void removeInternalLinks () {
    int index1;
```

```
    Iterator it = OutLinks.keySet().iterator();
    while (it.hasNext()) {
        Integer link1 = (Integer)(it.next());
        Map<Integer,Double> auxMap = (OutLinks.get(link1));
        Iterator it2 = auxMap.keySet().iterator();
        if(it2.hasNext()) {
            String URL1 = (String)(IdentifyerToURL.get(link1));
            index1 = URL1.indexOf("://");
            if(index1!=-1) URL1=URL1.substring(index1+3);
            index1 = URL1.indexOf("/");
            if(index1!=-1) URL1=URL1.substring(0,index1);
            while (it2.hasNext()) {
                Integer link2 = (Integer)(it.next());
                String URL2 = (String)(IdentifyerToURL.get(link2));
                index1 = URL2.indexOf("://");
                if(index1!=-1) URL2=URL1.substring(index1+3);
                index1 = URL2.indexOf("/");
                if(index1!=-1) URL2=URL1.substring(0,index1);
                if(URL1.equals(URL2)) {
                    auxMap.remove(link2);
                    OutLinks.put(link1,auxMap);
                    auxMap = (InLinks.get(link2));
                    auxMap.remove(link1);
                    InLinks.put(link2,auxMap);
                }
            }
        }
    }
}

/**
 *  删除内部导航链接
 */
public void removeNepotistic() {
    removeInternalLinks();
}

/**
 * 删除 stop URL
 */
public void removeStopLinks(String stopURLs[]) {
    HashMap aux = new HashMap();
    for (int i=0; i<stopURLs.length; i++) aux.put(stopURLs[i],null);
    removeStopLinks(aux);
}

/**
```

```
 *删除 stop URL
 */
public void removeStopLinks(Map stopURLs) {
    int index1;
    Iterator it = OutLinks.keySet().iterator();
    while (it.hasNext()) {
        Integer link1 = (Integer)(it.next());
        String URL1 = (String)(IdentifyerToURL.get(link1));
        index1 = URL1.indexOf("://");
        if(index1!=-1) URL1=URL1.substring(index1+3);
        index1 = URL1.indexOf("/");
        if(index1!=-1) URL1=URL1.substring(0,index1);
        if(stopURLs.containsKey(URL1)) {
            OutLinks.put(link1,new HashMap());
            InLinks.put(link1,new HashMap());
        }
    }
}

public int numNodes() {
    return nodeCount;
}
}
```

### 8.4.3 应用 HITS 进行链接分析

与 PageRank 一样，HTIS 也可用于链接分析。但是 HTIS 用于链接分析，不能够结合该链接所代表的网页的文本内容。因此，针对 THIS 有一些变种的算法，很值得我们学习。

#### 1. ARC 算法(Automatic Resource Compilation)

IBM Almaden 研究中心的 Clever 工程组提出了 ARC 算法，对原始的 HITS 做了改进，在赋予网页集对应的链接矩阵初值时，结合了链接的锚(anchor)文本，适应不同的链接具有不同的权值的情况。

ARC 算法与 HITS 的不同之处主要有以下 3 点：

(1) 由根集 S 扩展为 T 时，HITS 只扩展与根集中网页链接路径长度为 1 的网页，也就是只扩展与 S 相邻的网页，而 ARC 中把扩展的链接长度增加到 2，扩展后的网页集称为增集(Augment Set)。

(2) 在 HITS 算法中，将每个链接对应的矩阵值设为 1，但实际上每个链接的重要性是不同的，ARC 算法可以根据链接周围的文本来确定链接的重要性。在 ARC 算法中考虑链接 $p \rightarrow q$，p 中有若干链接标记，P 的格式是“文本 1<a href="q">锚文本</a>文本 2”，设查询项 t 在“文本 1”处，“锚文本”处，“文本 2”处，出现的次数为 n(t)，则 $w(p, q)=1/n(t)$。文本 1 和文本 2 的长度经过试验设为 50 字节。构造矩阵 W，如果有网页 $i \rightarrow j$，$W_{i,j}=w(i, j)$，否则 $W_{i,j}=0$，H 值设为 1，Z 为 W 的转置矩阵，迭代执行下面 3 个操作：

①A＝WH；②H＝ZA；③规范化 A。

以上三个步骤都是针对矩阵的操作。ARC 算法的目标是找到前 15 个最重要的网页，只需要 A/H 的前 15 个值相对大小保持稳定即可，不需要 A/H 整个收敛，这样②中迭代次数很小就能满足，实验表明迭代 5 次就可以，所以 ARC 算法有很高的计算效率，开销主要是在扩展根集上。

**2. Hub 平均算法(Hub Averaging Kleinberg)**

科学家 Allan Borodin 等指出了一种现象，设有 M+1 个 Hub 网页，M+1 个权威网页，前 M 个 Hub 指向第一个权威网页，第 M+1 个 Hub 网页指向所有 M+1 个权威网页。显然根据 HITS 算法，第一个权威网页最重要，有最高的 Authority 值，这是我们希望的。但是，根据 HITS 算法，第 M+1 个 Hub 网页有最高的 Hub 值，事实上，第 M+1 个 Hub 网页既指向了权威值很高的第一个权威网页，同时也指向了其他权威值不高的网页，它的 Hub 值不应该比前 M 个网页的 Hub 值高。因此，Allan Borodin 修改了 HITS 的算法，使得仅指向权威值高的网页的 Hub 值比既指向权威值高又指向权威值低的网页的 Hub 值高，此算法称为 Hub 平均算法。

**3. 阈值算法(Threshhold Kleinberg)**

Allan Borodin 等同时提出了 3 种阈值控制的算法，分别是 Hub 阈值算法，Authority 阈值算法，以及二者结合的全阈值算法。

计算网页(p)的 Authority 时，不考虑指向它的所有网页的 Hub 值对它的贡献，只考虑 Hub 值超过平均值的网页的贡献，这就是 Hub 阈值方法。

Authority 阈值算法和 Hub 阈值算法类似，不考虑所有 p 指向的网页的 Authority 对 p 的 Hub 值贡献，只计算前 K 个权威网页对 Hub 值的贡献。

在做链接分析的时候，使用 HITS 的改进算法，往往能取得更好的效果。

## 8.5 PageRank 与 HITS 的比较

PageBank 和 HITS 均是基于链接分析的搜索引擎排序算法，并且在算法中二者均利用特征向量作为理论基础和收敛性依据。但这两种算法的不同点也非常明显，下面就主要谈谈其不同点：

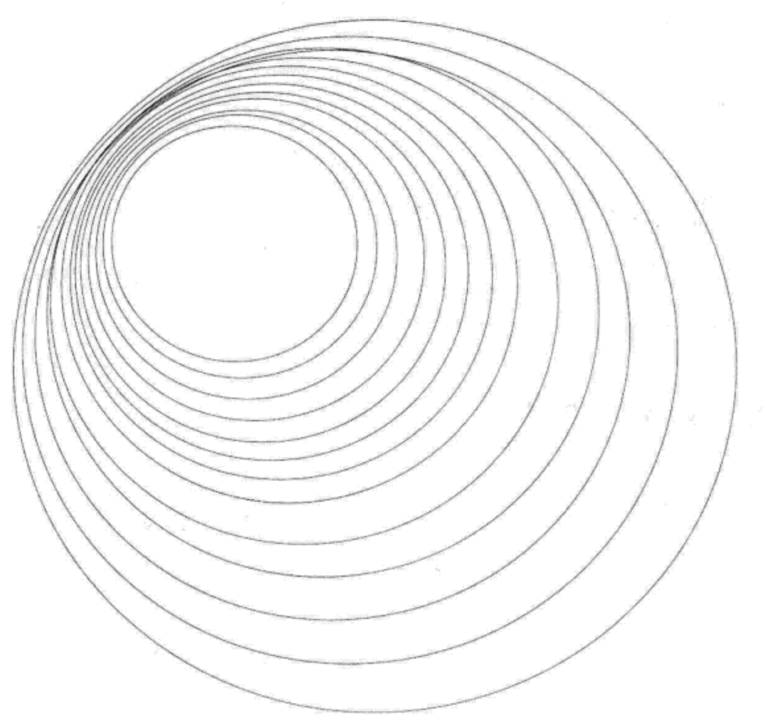
- 从算法思想上看，虽然同为链接分析算法，但二者之间还是有一定区别的。HITS 的原理如前所述，其 authority 值只是相对于某个检索主题的权重，因此 HITS 算法也常被称为 query dependent 算法。而 PageRank 算法独立于检索主题，因此也常被称为 query independent 算法。PageRank 的发明者(Page 和 Brin)和网络文档重要性的计算中借鉴了引文分析思想，利用网络自身的超链接结构给所有的网页确定一个重要性的等级数。当然 PageRank 并不是引文分析的完全翻版，根据因特网自身的性质等，它不仅考虑了网页引用的数量，还特别考虑了网页本身的重要性。
- 从权重的传播模型上来看，HITS 首先通过基于文本的搜索引擎来获得最初的处理数据，网页重要性的传播是通过 hub 页向 authority 页传递。而且 Kleinberg 认为，

hub 与 authority 之间是相互增强的关系；而 PageRank 基于随机冲浪(random surfer)模型，可以认为它将网页的重要性从一个 authority 页传递给另一个 authority 页。

- 从处理的数据量及用户端等待时间来分析。表面上看，当需排序的网页数量较小时，采用 HITS 算法，所计算的网页数量一般为 1000 至 5000 个，因为 HITS 算法需要从基于内容分析的搜索引擎中提取根集并扩充基本集，这个过程需要耗费相当长的时间。而 PageRank 算法从表面上看，处理的数据量远远超过了 HITS 算法。据 Google 介绍，目前收录的中文网页已达 33 亿以上，但由于其计算量在用户查询时已由服务器端独立完成，不需要用户端等待，基于该原因，从用户端等待时间来看，PageRank 算法应该比 HITS 要短。

## 8.6 本章小结

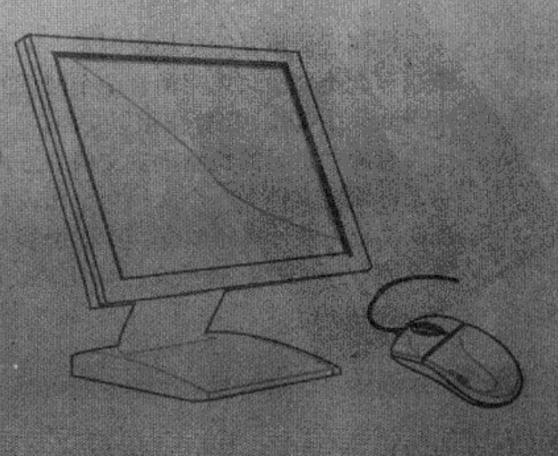
本章详细介绍了 Web 图的存储，包括内存存储和内存数据库存储两种情况。之后，还阐述了如何利用 PageRank 算法和 HITS 算法来分析 Web 图，并对两种算法进行了简单的比较。Web 图在网络爬虫中是非常重要的数据结构，它直接决定了搜索引擎中检索的质量，比如检索结果的排序等。有关 PageRank 和 HITS 的相关算法，如果读者还有不明白的，可以参考相关资料。



# 第9章

## 去掉重复的“文档”

互联网存在大量的重复内容，有研究显示，其中有 30%的网页内容重复。重复的网页降低了爬虫抓取有效信息的速度。与文本去重类似的技术还可以用在抄袭检测上。



## 9.1 何为“重复”的文档

有些网页是完全一样的，例如镜像网站和原网站中的内容。还有些文本大部分内容是一样的，而只有少许差别，例如：广告、计数器、时间戳等。

## 9.2 去除“重复”文档——排重

为什么需要去除重复文档？因为这样可以节省空间，提高搜索质量，从而节省用户的时间。

可以用比较 checksum 值的方法来判断完全相同的文档。checksum 是一个代表文档内容的值。如果两个文档的 checksum 值不匹配，则认为这两个文档不相同。当然，事实并不一定完全如此，但如果选择合适的 checksum 计算过程，则不同的文档产生相同的 checksum 值的概率很小。

checksum 计算过程叫做 checksum 算法。最简单的算法是把文本中的每个字符按编码求和。MD5 算法是一种流行的 checksum 算法，返回 128 位的字节数组。下面的代码返回给定文本的 MD5 值。

```
public static byte[] getMD5(String text) {
    MessageDigest md = null;
    byte[] encryptMsg = null;
    try {
        md = MessageDigest.getInstance("MD5"); // 取得 MD5-Instance
        encryptMsg = md.digest(text.getBytes()); // 求 MD5-Hash
    } catch (NoSuchAlgorithmException e) {
        System.out.println("No Such Algorithm Exception!");
    }
    return encryptMsg;
}
```

## 9.3 利用“语义指纹”排重

检测近似重复的文档很困难，因为很难对近似文档给出一个确切的定义。检测转载文章也与此类似。可以用衡量两个网页的相似度的阈值来定义近似文档。例如向量余弦夹角大于 0.9 的两篇文档算作相似文档。

有两种常见的检测重复文档的方式：在一个给定的文档集合内部检测，叫做自查重；将某一给定文档和一个文档集合比较，叫做单条查重。自查重最容易想到的方法是集合内部的文档两两计算相似度，但采用这种方法计算的时间复杂度是 $O(n^2)$，太高。为了提高比较效率，不直接比较原文，而比较文档的缩略表示。文档的语义缩略表示叫做语义指纹(FingerPrint)。

也可以把文档提取成结构化的形式，然后再生成语义指纹。为了提高准确性，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，即把“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

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：

```
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)
        {
            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();
}
```

### 9.3.1 理解“语义指纹”

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。

### 9.3.2　“语义指纹”排重的 Java 实现

语义指纹生成算法如下：

(1) 将每个文档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。

(2) 将特征项按照词权值排序。

(3) 选取前 5 个特征项，然后重新按照字符排序。

(4) 调用 MD5 算法，将每个特征项串转化为一个 128 位的串，作为该文档的指纹。

输入文档的标题和内容，返回代表语义指纹的字符串：

```
public static String getFingerPrint(String name, String content) {
    int min = Math.min(content.length(), 20000);
    content = content.substring(0, min);//取正文的前 20000 个字
    StringBuilder fingerPrint = new StringBuilder();
    //提取 FEATURE_NUM 个关键词
    String[] ret = getTag(name, content, FEATURE_NUM);
    for (int k = 0; k < ret.length; k++)
        fingerPrint.append(ret[k]);

    return fingerPrint.toString();
}
```

## 9.4　SimHash 排重

如果两个相似文档的语义指纹只相差几个位(bit)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位数。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 XOR B = 1 0 1 0

计算 1 0 1 0 中 1 的个数 = 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
}
```

### 9.4.1 理解 SimHash

文档向量是高维的，把维度削减技术用在文档排重上，就产生了 SimHash。已有的输入是文档的一系列特征，每个特征有不同的重要度。为了实现快速排重，需要计算文档对应的 SimHash 值(假定 SimHash 的长度为 64 位)。计算文档的 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 9.1 所示。

可以把特征权重看成特征在 Hash 结果的每一位上的投票权。权重大的特征的投票权更大，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 值中的更多位，而权重小的特征影响文档的 SimHash 值位数要少一些。

文档的 SimHash 值计算过程：

(1) 初始化长度为 64 位的向量，该向量的每个维度都是 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，则将向量的第 i 个数加上特征的权重；反之，如果 hash 值的第 i 位是 0，则将向量的第 i 个数减去特征的权重。

(3) 完成所有特征的处理，向量中的某些数为正，某些数为负。SimHash 值的每一位与向量中的每个数对应，将正数对应的位设为 1，负数对应的位设为 0，就得到了 64 位的 0/1 值的位数组，即最终的 SimHash。

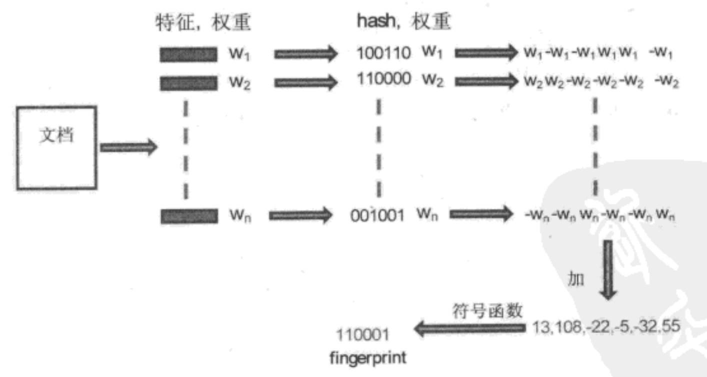



图 9.1 语义指纹计算过程

### 9.4.2 SimHash 排重的 Java 实现

设计排重接口：

首先生成一个语义指纹的集合(FingerPrint Set)，然后通过一个 SimHash 的 key 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);
```

计算 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 = GeneralHashFunctionLibrary.DJBHash(features[i]);
        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;
}
```

SimHash 的海明距离计算问题描述如下：给出一个 f 位的语义指纹集合 F 和一个语义指纹 fg，在 F 中寻找与 fg 只有 k 位以内差异的语义指纹。

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。首先借助组合数生成器 CombinationGenerator 来生成组合数：

```
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);
```

```java
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 count=2016。因为是从 64 位中选有差别的 2 位，所以计算公式是 $C_{64}^{2}=64*63/2=2016$。

完整的查找过程如下：

```java
//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)
    {
        //查找差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。例如 k=1 时，可以生成指纹集合 S 中每个元素的所有可能差 1 位的元素，然后再把所有这些新生成的元素排序，如图 9.2 所示。例如，对于长整型的元素，差别在 1 位以内的元素只有 65 种可能。

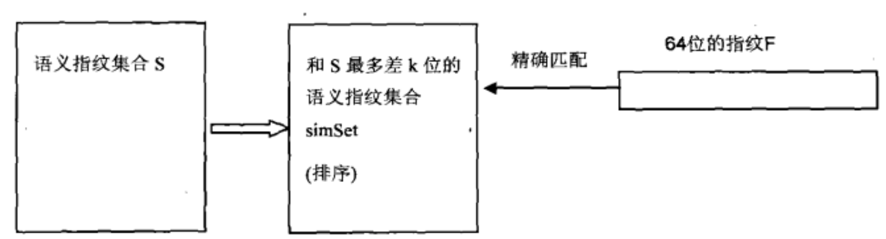



图 9.2　查找语义指纹

对给定 SimHash 值生成差 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
}
```

当 k 值较大时，会导致新集合过大而不容易存储。这时候可以考虑压缩存储语义编码。

在实际系统中，SimHash 数组是一个很大的数组，写入到文件中也会是一个很大的文件，比如 100 万的记录保存成 500MB 左右大小的文件。对于记录编号也会有类似压缩存储的需要。

数据压缩，通俗地说，就是用最少的字节来表示大量的数据。其作用是：把写入数据的文件减小，减少文件在硬盘上占用的存储空间，同时在读取数据时，减少对内存的占用。在 Java 中，1 个长整型数据占 8 个字节，但是可以使出现较多的数字使用较短的编码，如数字“1”的有用编码在最低的字节上，也就是说 1 个字节就能存，可是因为“1”是长整型，所以占了 8 个字节。如果把“1”用 1 个字节存，就能节约 7 个字节的空间。所以说如果较小的数值用较短的编码可以取得不错的压缩效果。

取出一个已经从小到大排好的数组的第一个数字，通过 longToBytes 方法把此数值转换成一个字节数组，用 BufferedOutputStream 实例把它写入文件，然后通过一个循环产生一个压缩的数组，用一个变量记录数组中后一个数减前一个数的差，如压缩数组中的第二个数等于原数组的第二个数减去原数组的第一个数，再通过 writeVLong 方法将这个变量写入文件中。这种压缩方式叫做差分编码(Differential Encoding)，或者增量编码。差分编码转换示例如下：

$$X_1, X_2, \cdots, X_n \Rightarrow X_1, X_2 - X_1, \cdots, X_n - X_{n-1}$$

用 DataInputStream 实例的 readLong 方法读出文件的第一个数字写入数组，再通过 readVLong 方法把该文件剩下的数字读取出来，同时把此数组剩下的数字都变成后一个数字和前一个数字的和，还原成原来的数据。还原数据的过程叫做解码，解码转换示例如下：

$$Y_1, Y_2, \cdots, Y_n \Rightarrow Y_1, Y_2 + Y_1, \cdots, Y_n + Y_{n-1}$$

写入数据并用差分编码压缩的实现代码如下：

```
//把一个长整型的数据变成二进制
public static byte[] longToBytes(long n)   {
    byte[] buf=new byte[8];//新建一个字节数组
    for(int i=buf.length-1;i>=0;i--){
       buf[i]=(byte)(n&0x00000000000000ff);//取低 8 位的值
       n>>>=8;//右移 8 位
    }
    return buf;
}
//压缩一个长整型的数据
public static void writeVLong(long i,BufferedOutputStream dos) throws
IOException{
     while ((i & ~0x7F) != 0) {
         dos.write((byte)((i & 0x7f) | 0x80)); //写入低位字节
         i >>>= 7; //右移 7 位
     }
     dos.write((byte)i);
}
//把长整型数组 simHashSet 写入 fileName 指定的文件中
private static int write(long[] simHashSet,String fileName) {
    int j=0;
    try {
        BufferedOutputStream dos =
                 new BufferedOutputStream(new FileOutputStream(fileName));
        //把数组中的第一个数字转换成二进制表示
        byte[] b = longToBytes(simHashSet[0]);
        dos.write(b);//把它写到文件中
        for (int i = 1; i < simHashSet.length; i++) {
            //数组中后一个数减前一个数的差
            long deta=simHashSet[i]-simHashSet[i-1];
            writeVLong(deta, dos);// 把这个差值写入文件
        }
        dos.close();
        j=simHashSet.length;
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return j;
}
```

读出数据并解压缩的实现代码如下：

```
//把一个压缩后的长整型的数据读取出来
```

```
private static long readVLong(DataInputStream dis) throws IOException {
    byte b = dis.readByte(); //读入一个字节
    int i = b & 0x7F; //取低 7 位的值
    //每个高位的字节多乘个 2 的 7 次方，也就是 128
    for (int shift = 7; (b & 0x80) != 0; shift += 7) {
        if(dis.available()!=0){
            b = dis.readByte();
            i |= (b & 0x7F) << shift; //当前字节表示的位乘 2 的 shift 次方
        }
    }
    return i;//返回最终结果 i
}
//从 fileName 指定的文件中把长整型数组读出来
private static void read(int len,String fileName) {
    try {
        DataInputStream dis = new DataInputStream(new BufferedInputStream(
                new FileInputStream(fileName)));
        long[] simHashSet = new long[len];
        simHashSet[0] = dis.readLong();//从文件读取第一个长整型数字放入数组
        for (int i = 1; i < len; i++) {
            simHashSet[i] = readVLong(dis);//读取文件剩下的元素
            //将元素都变成数组中后一个数字和前一个数字的和
            simHashSet[i] = simHashSet[i] + simHashSet[i - 1];
        }
        dis.close();
        for(int i=0;i<len;i++){   //将长整型数组的数据显示出来
            System.out.println(simHashSet[i]);
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

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：

- 整个表中排列组合的部分很少，不太可能出现，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。
- 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。

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。海明距离的比较在 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。

总的思想：先把要检索的集合缩小，然后在小集合中检索 f-d'位的海明距离。

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 9.3 所示。

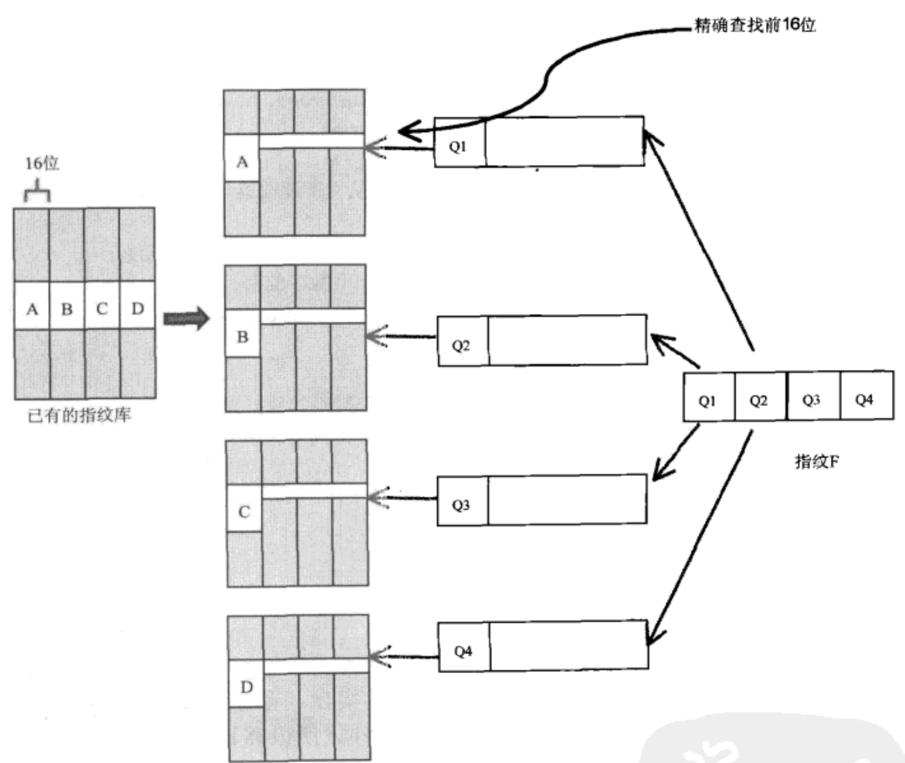
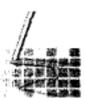



图 9.3 分 4 块查找语义指纹

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;
        if(o1.equals(o2)) return 0;
        return (isLessThanUnsigned(o1,o2)) ? 1: -1;
    }
}; // 比较无符号 64 位中的高 16 位
```

对表 $T_1$、$T_2$、$T_3$、$T_4$ 按高位排序的代码如下：

```
public void sort()
{
    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);
}
```

下面介绍两种更复杂的分块方法：

**第一种策略**：语义指纹 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 位不同，把这三块限制到低位，换言之，把精确匹配的三块选到高位，有 $C_6^3=20$ 种选法，因此需要复制 20 个语义指纹集合 T 表。对每个表高位做精确匹配，需要匹配 31~33 位(11*3=33、11*2+10=32、11+10*2=31)，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 $2^{|34-28|}=64$ 个需要算海明距离的 SimHash。

海明距离算法步骤：

(1) 先复制原表 T 为 t 份：$T_1,T_2,\cdots,T_t$。

(2) 每个 $T_i$ 都关联一个 $p_i$ 和一个$\pi_i$，其中 $p_i$ 是一个整数，$\pi_i$ 是一个置换函数，负责把 $p_i$ 的个位换到高位上。

(3) 把置换函数$\pi_i$ 应用到相应的表 $T_i$ 上，然后对 Ti 排序。

(4) 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 $T_i$ 中与高 $p_i$ 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) 合并 t 个表检索出的结果。

举个例子，假设有 100 亿左右($2^{34}$)的语义指纹。SimHash 有 64 位，所以 f＝64，d＝34。海明距离是 3。

算法的本质就是采用分治法，降低问题规模。利用内存空间的增加和并行计算换取查找时间的减少。当然这个算法即使不并行也比逐次探查法快。

## 9.5 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。假设有一批 1MB 大小的语义指纹，因此文件 Q 的大小是 8MB。Google 把文件 F 和 Q 存放在 GFS 分布式文件系统中。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。并且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)。

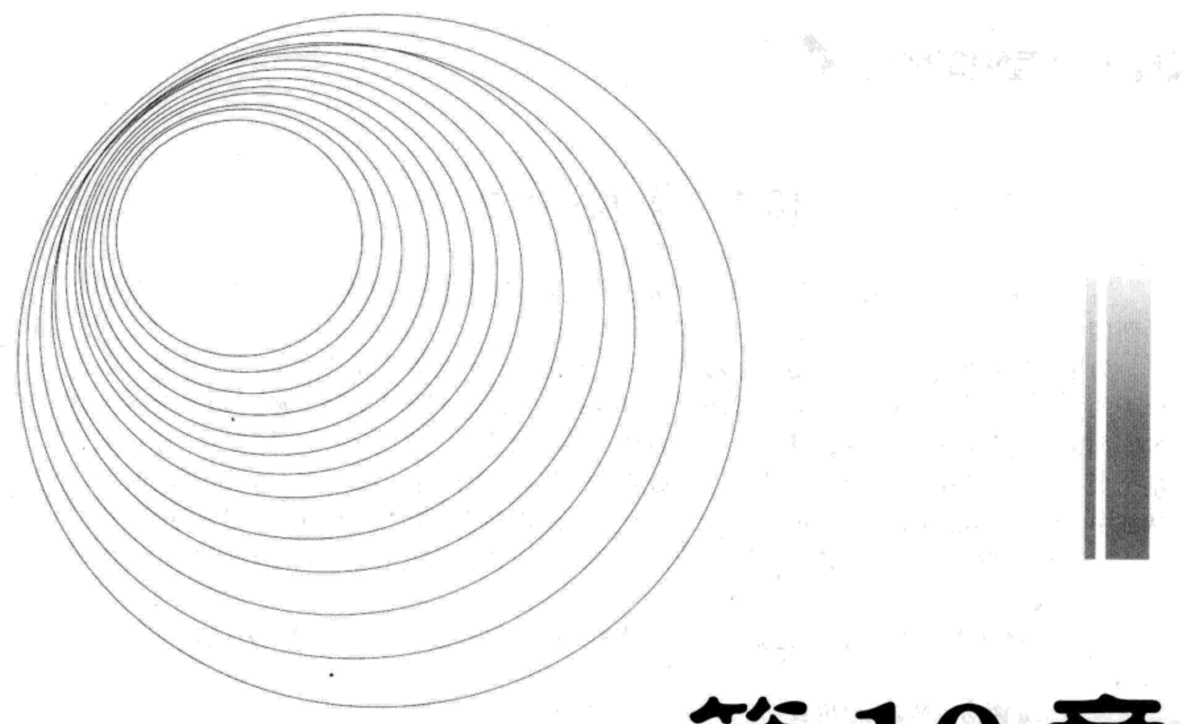
每个任务以整个文件 Q 作为输入在某个 64MB 的组块上求解海明距离。

一个任务以它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。

Google 用 200 个任务(mapper)，扫描组块的合并速度在每秒 1GB 以上。压缩版本的文件 Q 大小约为 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。

## 9.6 本章小结

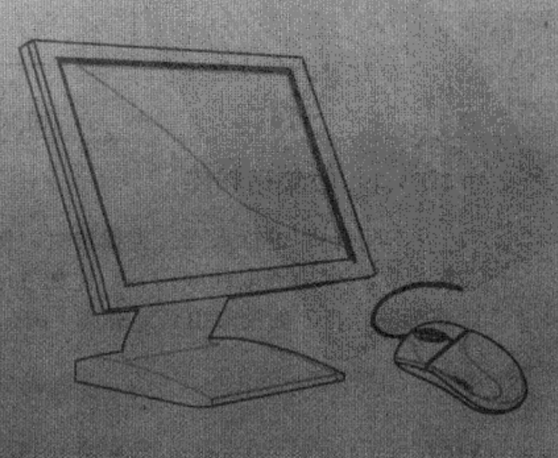
Broder 提出 shingling 算法用于文档内容相似性检测。Charikar 在论文 Similarity Estimation Techniques from Rounding Algorithms 中提出了用 SimHash 实现维度约减。Google 公司的论文 Detecting NearDuplicates for Web Crawling 介绍了把 SimHash 用于爬虫抓取过程中的网页去重。本章介绍了用语义指纹排重的基本方法，以及一种特殊的语义指纹 SimHash。详解在 Java 中实现 SimHash 与 SimHash 的压缩存储方法。



# 第10章

## 分类与聚类的应用

抓取的网页或文档需要区分出类别以方便浏览或检索。处理互联网上的海量信息需要能够自动分类或聚类。分类和聚类有很多技术基础是相同的，例如分类和聚类的依据一般是从文本中提取的特征。这两个任务都是经典的机器学习问题。



# 10.1 网 页 分 类

可以把网页归为一类，或者分为多个类别。例如可以把新闻分成“国内新闻”或“国际新闻”。

为了理解机器学习的算法是如何对网页进行分类的，首先看一下人是如何对事物进行分类的。为了判断食物是否为健康食品，可参考食品中的饱和脂肪、胆固醇、糖和钠的含量。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”。首先从整个项目集合中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进行分类。

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，可以通过食品包装上的营养成分表来取得。

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。

$$H(食物)= W_{脂肪}脂肪(食物) + W_{胆固醇}胆固醇(食物) + W_{糖}糖(食物) + W_{钠}钠(食物)$$

这里 $W_{脂肪}$、$W_{胆固醇}$ 等是和每个特征关联的重要度。在这个公式里，这些值可能是负数。

对网页中的文本进行分类是网页分类的重要基础。除文本外，还可以利用网页的 URL 地址和网页输入链接的锚点文字等信息来对网页分类。图 10.1 展示了文本分类的框架。

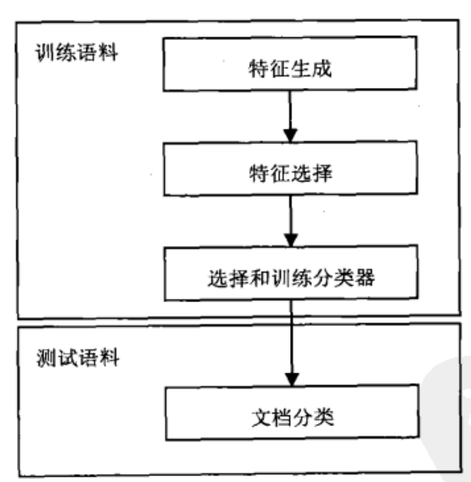



图 10.1 文本分类的步骤

## 10.1.1 收集语料库

对语料库进行人工分类很费时。例如要取得分好类的“国际新闻”和“国内新闻”。可以利用爬虫抓取新闻网站，比如 http://news.sina.com.cn，这个网站上有国际新闻和国内新闻的栏目。如果索引页能分成“国际新闻”或“国内新闻”，“详细页”的类别参考“索

引页”的分类，那么抓取下来的语料就可以分别归在“国际新闻”或“国内新闻”类别下。

### 10.1.2　选取网页的“特征”

对于文本分类来说，可以选择文本中出现的字、词或者词组作为分类特征。根据实验结果，普遍认为选取词作为特征优于选取字或者词组作为特征。所以实际的中文文本分类系统中，特征一般是对文本分词后得到的词。

词的数量可能很多，例如中文分词词表可能包括几十万个词。特征空间维度太大会影响分类器的执行效率和有效性，因此需要降低特征空间维度，这个技术叫做特征选择。

可以按词性过滤，只选择某些词性作为分类特征，比如，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。一般至少可以到 2000 个词。

特征选择的常用方法还有：CHI 方法和信息增益(Information Gain)方法等。下面介绍特征选择的 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。

用 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 10.1 计算。

表 10.1　CHI 统计变量定义表

| | 属于 class 类 | 不属于 class 类 | 合计 |
|---|---|---|---|
| 包含单词 term | a | b | a+b |
| 不含单词 term | c | d | c+d |
| 合计 | a+c | b+d | a+b+ c+d=n |

这里，a 代表属于类别 class 的文档集合中出现单词 term 的文档数；b 代表不属于类别 class 的文档集合中出现单词 term 的文档数；c 代表属于类别 class 的文档集合中未出现单词 term 的文档数；d 代表不属于类别 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 10.1 中的单词 term 的 CHI 统计公式如下：

chi_statistics(term, class) = n*(ad-cb)^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。

以表 10.2 中的垃圾邮件分类为例，在这个例子中有 10 个文档(每个文档有一个唯一的 ID 编号)，要把文档分到两个类别(spam 和 not spam)中的一个，对文档分类的候选分类特征词是 cheap、buy、banking、dinner 和 the。

表 10.2 垃圾邮件分类表

| document id | cheap | buy | banking | dinner | the | class |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 1 | not spam |
| 2 | 1 | 0 | 1 | 0 | 1 | spam |
| 3 | 0 | 0 | 0 | 0 | 1 | not spam |
| 4 | 1 | 0 | 1 | 0 | 1 | spam |
| 5 | 1 | 1 | 0 | 0 | 1 | spam |
| 6 | 0 | 0 | 1 | 0 | 1 | not spam |
| 7 | 0 | 1 | 1 | 0 | 1 | not spam |
| 8 | 0 | 1 | 0 | 0 | 1 | not spam |
| 9 | 0 | 0 | 0 | 0 | 1 | not spam |
| 10 | 1 | 0 | 0 | 1 | 1 | not spam |

对于 chi_statistics(dinner, spam) 来说，a = 0, b=1,c=3,d=6, n =10

chi_statistics(dinner,spam) = 10 *(0*6 – 3*1)^2 / (0+3)*(1+6)*(0+1)*(3+6) = 10*9/3*7*1*9
=0.11

对于 chi_statistics(the, spam) 来说，a = 3, b=7,c=0,d=0, n =10

chi_statistics(the,spam) = 10 *(3*0 – 0*7)^2 / (3+0)*(7+0)*(3+7)*(0+0)，除零溢出，因此“the”不作为分类特征。

对于 chi_statistics(cheap, spam) 来说，a = 3, b=1,c=0,d=6, n =10。

chi_statistics(dinner,spam) = 10 *(3*6 – 0*1)^2/(3+0)*(1+6)*(3+1)*(0+6) = 10*324/3*7*4*6
=6.43

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。

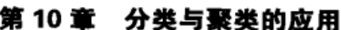
信息增益的计算方法是：把文档集合 $D$ 看成一个符合某种概率分布的信息源，依靠文档集合的信息熵和文档中词语的条件熵之间信息量的增益关系确定该词语在文本分类中所能提供的信息量。

词语 $w$ 的信息量的计算公式为：

$$\mathrm{IG}(w)=H(D)-H(D|w)$$
$$=-\sum_{d_i\in D}P(d_i)\times\log_2 P(d_i)+\sum_{w\in\{0,1\}}P(w)\sum_{d_i\in D}P(d_i\,|\,w)\times\log_2 P(d_i\,|\,w)$$

这里 $H(D)$是 $P(C)$的熵，而 $H(D|w)$是条件熵。因为涉及到取对数，因为这里的 IG 只有相对大小的意义。为了在 Java 中方便计算，可以简单地以 e 为底计算。计算 cheap 的信息增益：

$$\mathrm{IG(cheap)}=-P(\mathrm{spam})\log P(\mathrm{spam})-P(\overline{\mathrm{spam}})\log P(\overline{\mathrm{spam}})+$$
$$P(\mathrm{cheap})P(\mathrm{spam|cheap})\log P(\mathrm{spam|cheap})+$$
$$P(\mathrm{cheap})P(\overline{\mathrm{spam}}\,|\,\mathrm{cheap})\log P(\overline{\mathrm{spam}}\,|\mathrm{cheap})+$$



$$P(\overline{cheap})P(spam|\overline{cheap})\log P(spam|\overline{cheap})+$$
$$P(\overline{cheap})P(\overline{spam}|\overline{cheap})\log P(\overline{spam}|\overline{cheap})$$
$$=-\frac{3}{10}\log\frac{3}{10}-\frac{7}{10}\log\frac{7}{10}+\frac{4}{10}*\frac{3}{4}\log\frac{3}{4}$$
$$+\frac{4}{10}*\frac{1}{4}\log\frac{1}{4}+\frac{6}{10}*\frac{0}{6}\log\frac{0}{6}+\frac{6}{10}*\frac{6}{6}\log\frac{6}{6}$$
$$=0.2749$$

计算程序如下：

```
double ig = -(3.0 / 10.0) * (Math.log(3.0 / 10.0))
        - (7.0 / 10.0) * (Math.log(7.0 / 10.0))
        + (4.0 / 10.0) * (3.0 / 4.0) * (Math.log(3.0 / 4.0))
        + (4.0 / 10.0) * (1.0 / 4.0) * (Math.log(1.0 / 4.0))
        + (6.0 / 10) * (0.0 / 6) * (0.0)
        + (6.0 / 10.0) * (6.0 / 6.0) * (Math.log(6.0 / 6.0));
System.out.println(ig);
```

分类器采用特征选择来降维的过程是：先计算所有候选特征的类别区分度，再选出区分度最大的 5000 个，然后重新计算特征词的权重。

### 1. 计算候选特征集中每个特征的类别区分度

wordNode.m_dWeight = 各类别下文本区分度之和

概率统计有基于词的统计和基于文档的统计两种方法。以基于词的统计方法为例，对每个文本类别，令：

P_c 表示类别中的词占所有类别的词的比率。

P_ft 表示含 f_t 的词占词总数的比率。

P_c_ft 表示类别中含 f_t 的词占词数的比率。

```
P_n_c = 1 - P_c;
P_n_ft = 1 - P_ft;
P_n_c_ft = P_ft - P_c_ft;
P_c_n_ft = P_c - P_c_ft;
P_n_c_n_ft = P_n_ft - P_c_n_ft;
```

可以用不同的方法计算区分度：

方法一：右半信息增益(Right half of IG)

$$Weight += P_c_i_n_ft * \log\left(\frac{P_c_i_n_ft}{P_c * P_n_ft}\right)$$

方法 2：信息增益 (IG)

$$Weight += P_c_i_n_ft * \log\left(\frac{P_c_i_n_ft}{P_c_i * P_n_ft}\right);$$

$$Weight += P_c_i_ft * \log\left(\frac{P_c_i_ft}{(P_c_i * P_ft}\right)$$

方法 3：互信息(MI)

$$\text{Weight}+=\text{P_c}_i*\log\left(\frac{\text{P_c}_i\text{_ft}}{\text{P_c}_i*\text{P_ft}}\right)$$

方法 4：X2 统计量(CHI)

$$\text{Weight}+=\text{P_c}_i\text{_ft}*\text{P_n_c}_i\text{_n_ft-P_n_c}_i\text{_ft}*\text{P_c}_i\text{_n_ft}$$
$$*\frac{\text{P_c}_i\text{_ft}*\text{P_n_c}_i\text{_n_ft}-\text{P_n_c}_i\text{_ft}*\text{P_c}_i\text{_n_ft}}{\text{P_ft}*\text{P_n_c}_i*\text{P_n_ft}}$$

方法 5：期望交叉熵(ECE)

$$\text{Weight}+=\text{P_c}_i\text{_ft}*\log\left(\frac{\text{P_c}_i\text{_ft}}{\text{P_c}_i*\text{P_ft}}\right)$$

方法 6：文本证据权重(WET)

默认的概率统计是基于词的统计，特征选择算法用 X2 统计量。

**2. 生成新的特征词和权重。特征词的权重有两种计算方法**

TF*IDF：特征区分度* log(文档数/特征的文档频率)

TF_IDF_DIFF：log(文档数/特征的文档频率)

### 10.1.3 使用支持向量机进行网页分类

支持向量机(SVM)是 Vipnik 于 1995 年根据统计学习理论(Statistical Learning Theory)提出的一种机器学习方法。

支持向量机的最初思想是对于线性可分问题如何寻求最优分类面。图 10.1 显示了一个两类分类问题，其中“+”是特征空间的一类点，而“-”是特征空间中的另一类点。图 10.1 中左边的点是线性可分的，右边的点是线性不可分的。对于特征空间中线性可分问题，最优分类面就是间隔 $\gamma$ 最大的分界面。

线性判别函数(discriminant function)是指由分类向量 x 的各个分量的线性组合而成的函数：g(x)=w*x +b

分成两类的情况：把分类向量 x 分成 C1 类或 C2 类，对于两类问题的决策规则如图 10.2 所示。如果 g(x)≥0，则判定 x 属于 C1 类；如果 g(x)<0，则判定 x 属于 C2 类。

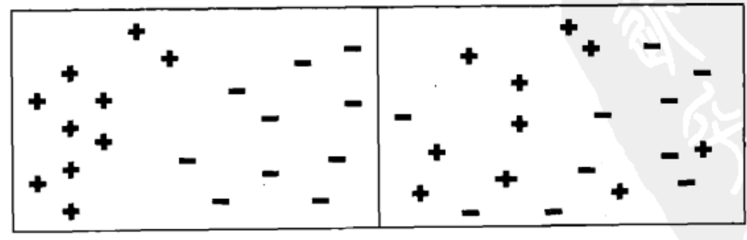


图 10.2 两类分类问题

当 g(x)是线性函数时，这个平面被称为“超平面”(hyperplane)。

在一维空间中，任何一个线性函数能不能解决图 10.3 所示的划分问题(粗线和细线各代表一类数据)，可见线性判别函数有一定的局限性。

a　b

图 10.3　线性函数不能分类的问题

如果建立一个二次判别函数 g(x)=(x−a)(x−b)。如图 10.4 所示，则可以很好地解决图 10.3 所示的分类问题。决策规则仍是：如果 g(x)≥0，则判定 x 属于 C1；如果 g(x)<0，则判定 x 属于 C2。

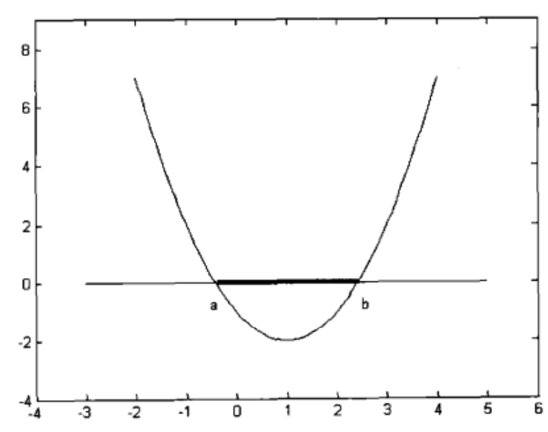



图 10.4　二次判别函数

SVM 是从线性可分情况下的最优分类面发展而来的。基本思想可用图 10.5 所示的两维情况说明。

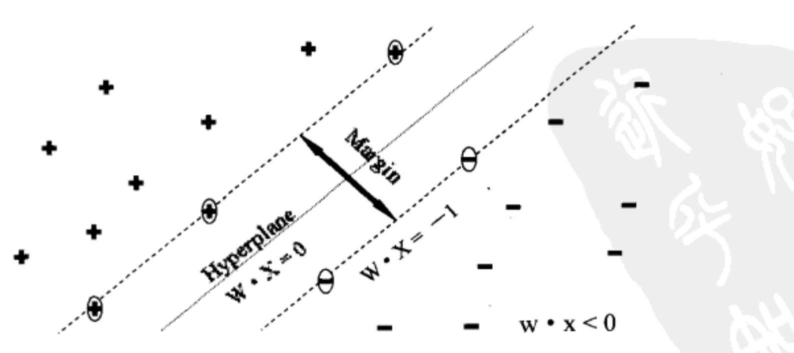



图 10.5　分类间隔

最优分类超平面是指该分类面不但能正确分类而且使各类别的分类间隔最大。

SVM 文档分类实现流程如下：

(1) 生成所有候选特征项，将其保存在 candidateWordList 中。

```
WordList candidateWordList = genDic();
```

(2) 为候选特征项列表 candidateWordList 中的每个特征设置权重。

```
featherWeight(candidateWordList);
```

(3) 从特征项列表 candidateWordList 中选出最优特征。

```
WordList trainWordList = featherSelection(candidateWordList);
```

(4) 生成文档向量，将其保存在 trainWordList 中。

```
genModel(trainWordList);
```

(5) 训练 SVM 分类器。

```
trainSVM();
```

(6) 保存分类模型。

```
trainWordList.writeModel(m_paramClassifier.m_txtResultDir +
"\\model.prj");
```

### 10.1.4 利用 URL 地址进行网页分类

从 http://bj.58.com/job.shtml 网址就可以知道它可能是和招聘相关的网页，而且可能是北京地区的网页。因此，使用 URL 地址可以快速分类网页。

每次搜索 Google 都会看见一个编码后的 URL 地址，例如：http://www.google.com/search?client=opera&rls=en&q=%22rabbits%22%2BEaster+eggs&sourceid=opera&ie=utf-8&oe=utf-8 实际的查询词是：

```
"rabbits"+Easter eggs
```

使用下面的 Java 语句对编码后的 URL 地址解码：

```
String reconstituted = URLDecoder.decode( received, "UTF-8" );
```

为了提取“http://bj.58.com/job.shtml”的特征词“bj”、“58”、“job”，首先按“:”、“/”、“.”切分，然后通过词干化和小写化处理，去除停用词“http”，“com”，“shtml”，最后剩下“bj”、“58”、“job”三个分类特征词。

### 10.1.5 使用 AdaBoost 进行网页分类

Boosting 是近年来发现的最强大的机器学习方法之一。它的基本原理就是“三个臭皮匠，顶个诸葛亮”。可以将弱的分类器组合成为高精度的分类器。用来组合的弱分类器必须在

训练实例上的错误少于 50%。假设有 $t$ 个弱分类器，分类结果是 $h_t(x)$，可以采用如下的公式线性组合强分类器：$f(x)=\sum_{t=1}^{T}\alpha_t h_t(x)$

AdaBoost 使用投票的方法来综合 $t$ 个弱分类器的分类结果。根据 MALLET(http://mallet.cs.umass.edu)中的 AdaBoost 分类过程实现代码说明如下：

```
FeatureVector fv = (FeatureVector) inst.getData(); //取得特征向量

int numClasses = getLabelAlphabet().size();//取得类别数量
double[] scores = new double[numClasses];
int bestIndex;
double sum = 0;
// 收集所有弱分类器的分值
for (int round = 0; round < numWeakClassifiersToUse; round++) {
    bestIndex =
weakClassifiers[round].classify(inst).getLabeling().getBestIndex();
    scores[bestIndex] += alphas[round];
    sum += scores[bestIndex];
}
// 归一化分值
for (int i = 0; i < scores.length; i++)
    scores[i] /= sum;
return new Classification (inst, this, new LabelVector (getLabelAlphabet(),
scores));
```

AdaBoost 的训练过程是：首先，使用原始的训练集生成一个分类器。这时候每个弱分类器的投票权一样。然后，给被错误分类的实例赋予较大的权重(把这些实例错误分类会使得错误更大)。接下来，通过使用上一步带权重的训练集取得新的分类器。计算训练错误，根据弱分类器在训练集上的表现给这个弱分类器分配权重。因此，新的分类器在错误的实例上应该有更好的表现。这个过程可以重复多次，最终，在最后的分类器中组合若干个弱分类器。

```
FeatureSelection selectedFeatures = trainingList.getFeatureSelection();

java.util.Random random = new java.util.Random();
//设置训练实例的初始权重为均匀分布
double w = 1.0 / trainingList.size();
InstanceList trainingInsts = new InstanceList(trainingList.getPipe(),
trainingList.size());
for (int i = 0; i < trainingList.size(); i++)
    trainingInsts.add(trainingList.get(i), w);

boolean[] correct = new boolean[trainingInsts.size()];
Classifier[] weakLearners = new Classifier[numRounds]; //弱分类器
double[] alphas = new double[numRounds];
```

```
InstanceList roundTrainingInsts = new
InstanceList(trainingInsts.getPipe());

// Boosting 叠迭，每轮叠迭学习出一个弱分类器
for (int round = 0; round < numRounds; round++) {
    // 保持重新采样训练实例(使用实例权重的分布)，在这些实例上训练弱分类器
    // 直到达到预定的最大叠迭次数，或者弱分类器在训练集中学习到非零的错误
    // 这样可以保证至少采样到一些难的实例
    int resamplingIterations = 0; //重复采样叠迭计数器
    double err;
    do {
        err = 0;
        roundTrainingInsts =
                trainingInsts.sampleWithInstanceWeights(random);
        weakLearners[round] = weakLearner.train (roundTrainingInsts);

        // 计算错误
        for (int i = 0; i < trainingInsts.size(); i++) {
            Instance inst = trainingInsts.get(i);
            if (weakLearners[round].classify(inst).bestLabelIsCorrect())
                correct[i] = true;
            else {
                correct[i] = false;
                err += trainingInsts.getInstanceWeight(i);
            }
        }
        resamplingIterations++;
    }
    while (Maths.almostEquals(err, 0) &&
           resamplingIterations < MAX_NUM_RESAMPLING_ITERATIONS);

    // 计算分配给弱分类器的权重
    alphas[round] = Math.log((1 - err) / err);
    double reweightFactor = err / (1 - err);
    double sum = 0;
    //降低正确分类的实例的权重
    for (int i = 0; i < trainingInsts.size(); i++) {
        w = trainingInsts.getInstanceWeight(i);
        if (correct[i])
            w *= reweightFactor;
        trainingInsts.setInstanceWeight (i, w);
        sum += w;
    }
    // 归一化训练实例的权重
    for (int i = 0; i < trainingInsts.size(); i++) {
        trainingInsts.setInstanceWeight (i, trainingInsts.
        getInstanceWeight(i) / sum);
```

```
    }
    logger.info("==========  AdaBoostTrainer round " + (round+1)
            + " finished, weak classifier training error = " + err);
}
//打印训练出来的 alpha 值
for (int i = 0; i < alphas.length; i++)
    logger.info("AdaBoostTrainer weight[weakLearner[" + i + "]]=" +
                alphas[i]);
    this.classifier = new AdaBoost (roundTrainingInsts.getPipe(),
    weakLearners, alphas);
```

# 10.2 网页聚类

分类算法需要训练数据，也就是分好类的数据。和分类算法不同，聚类是无监督的学习。正如其他一切无监督学习问题一样，处理过程要在没有标记的数据中寻找相应的结构。

聚类就是“把相似的物体组织到一起”。相似的物体分在一起，叫做“簇”，因此同簇中的对象相似度大，不同簇的对象相似度小。图 10.6 是一个聚类简单的图形化例子。

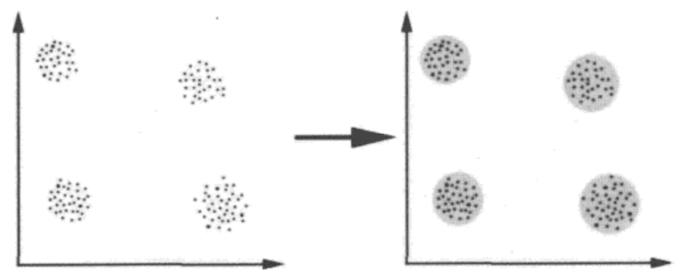


图 10.6　聚类

卡耐基-梅隆大学的研究人员把聚类技术和网页搜索相结合，创建了 http://vivisimo.com/ 聚类搜索技术商业公司。vivisimo 并不是开源的，而 http://project.carrot2.org/则是一个开源的聚类搜索引擎。

## 10.2.1　深入理解 DBScan 算法

DBScan 是 Martin Ester 等人在 1996 年提出的一个数据聚类算法。DBScan 算法是基于密度的方法。它从对应点的估计的密度分布发现一些簇，能够发现任意形状的簇并且可以有效地处理噪声。

DBScan 基于密度可达性的概念来定义簇。如果满足两个条件就认为从一个点 $p$ 直接密度可达点 $q$：如果它们之间的距离在一个给定的距离 $\varepsilon$ 内(也就是说 $q$ 是 $p$ 的一个 $\varepsilon$ 邻居)，并且如果 $p$ 周围围绕了足够多的点。这样就可以认为 $p$ 和 $q$ 是属于同一个簇中的点。

如果存在一个点的序列 $p_1,\cdots,p_n$，其中 $p_1=p$，$p_n=q$，$p_i$ 直接密度可达 $p_{i+1}$，则认为从

$p$ 密度可达 $q$。注意，密度可达关系不是对称的，也就是说 $p$ 密度可达 $q$ 并不一定保证 $q$ 密度可达 $p$，因为 $q$ 可能位于簇的边界，没有足够多的邻居点来让它真正成为簇元素。因此引入了密度连通的概念：如果存在一个中间点 $o$，$o$ 与 $p$ 是密度可达的，同时 $o$ 与 $q$ 也是密度可达的，则点 $p$ 和 $q$ 是密度连通的。一个簇是文档库中的点的子集，簇满足两个属性：簇中的所有点是互相密度连通的；如果某个点和簇中的任意某个点是密度连通的，则这个点也是簇的一部分。

在一个簇中有两种不同的点：$\varepsilon$ 邻居数量多于指定阈值 MinPts 的称为核心点，否则为非核心点。核心点可以把非核心点"拉"到簇里面。

DBScan 需要两个参数：Eps 和 MinPts 来形成一个簇。从任意的未访问过的点开始，如果它包含足够多的 $\varepsilon$ 邻居，则形成一个簇，否则把这个点标示成噪声。注意，这个点后来可能在另外某个点的 $\varepsilon$ 环境中，因此成为一个簇中的一部分。

如果一个点是簇的一部分，则它的 $\varepsilon$ 邻居也是簇的一部分。因此，把它的 $\varepsilon$ 邻居中的所有点都加入到簇中。重复这个扩展点的过程，直到完全发现簇。然后取一个新的没访问过的点同样处理，从而发现另外一个簇或者噪音。

发现一个簇的步骤基于这样一个原理：一个簇能够由它的任一核心对象唯一地确定。伪代码如下：

```
DBSCAN(D, eps, MinPts)
   C = 0
   对数据集 D 中每个未访问过的点 P
      把 P 标志成已经访问过
      N = getNeighbors (P, eps)
      if sizeof(N) < MinPts
         把 P 标志成噪音
      else
         C = 下一个簇
         expandCluster(P, N, C, eps, MinPts)

expandCluster(P, N, C, eps, MinPts)
   把 P 加到簇 C
   对 N 中的每个点 P'
      if 没有访问过 P'
         把 P'标志成已访问
         N' = getNeighbors(P', eps)
         if sizeof(N') >= MinPts
            N = N 与 N'合并
      if P' 不是任何簇的成员
         把 P'加到簇 C
```

### 10.2.2 使用 DBScan 算法聚类实例

Weka 中有个 DBScan 算法的实现。源代码在 weka.clusterers 包中，文件名为 DBScan.java。其中 buildClusterer 和 expandCluster 这两个方法是最核心的方法。buildClusterer 是所有聚类方法实现的接口，而 expandCluster 用于扩展样本对象集合的高密度连接区域。另外还有一

个用于查询指定点的 ε 邻居的方法，叫 epsilonRangeQuery，这个方法在 Database 类中，调用例子如下：

```
seedList = database.epsilonRangeQuery(getEpsilon(),dataObject);
```

在 buildClusterer 方法中，通过对每个未聚类的样本对象调用 expandCluster 方法，查找由这个对象开始的密度连通的最大样本对象集合。在这个方法中处理的主要代码如下：

```
while (iterator.hasNext()) {
    DataObject dataObject = (DataObject) iterator.next();
    if (dataObject.getClusterLabel() == DataObject.UNCLASSIFIED) {
        if (expandCluster(dataObject)) {//一个簇已经形成
            clusterID++;//取下一个聚类标号
            numberOfGeneratedClusters++;
        }
    }
}
```

下面再来看 expandCluster 方法，这个方法的输入参数是样本对象 dataObject。这个方法首先判断样本对象是不是核心对象，如果是核心对象再判断这个样本对象的 ε 邻域中的每一个对象，检查它们是不是核心对象，如果是核心对象则将其合并到当前的聚类中。源代码分析如下：

```
//查找输入的 dataObject 样本对象的 ε 邻域中的所有样本对象
List seedList = database.epsilonRangeQuery(getEpsilon(), dataObject);
//判断 dataObject 是不是核心对象
if (seedList.size() < getMinPoints()) {
    //如果不是核心对象则将其设置为噪声点
    dataObject.setClusterLabel(DataObject.NOISE);
    //没有发现新的簇，所以返回 false
    return false;
}

//如果样本对象 dataObject 是核心对象，则对其邻域中的每一个对象进行处理
for (int i = 0; i < seedList.size(); i++) {
    DataObject seedListDataObject = (DataObject) seedList.get(i);

    //设置 dataObject 邻域中的每个样本对象的聚类标识，将其归为一个簇
    seedListDataObject.setClusterLabel(clusterID);

    //如果邻域中的样本对象与当前 dataObject 是同一个对象，那么将其删除
    if (seedListDataObject.equals(dataObject)) {
        seedList.remove(i);
        i--;
    }
}

//对 dataObject 的 ε 邻域中的每一个样本对象进行处理
```

```
for (int j = 0; j < seedList.size(); j++) {
    //从邻域中取出一个样本对象 seedListDataObject
    DataObject seedListDataObject = (DataObject) seedList.get(j);

    //查找 seedListDataObject 的 epsilon 邻域并取得其中所有的样本对象
    List seedListDataObject_Neighbourhood =
    database.epsilonRangeQuery(getEpsilon(), seedListDataObject);

    //判断 seedListDataObject 是不是核心对象
    if (seedListDataObject_Neighbourhood.size() >= getMinPoints()) {
        for (int i = 0; i < seedListDataObject_Neighbourhood.size(); i++){
            DataObject p =  (DataObject)
                               seedListDataObject_Neighbourhood.get(i);
        //如果 seedListDataObject 样本对象是一个核心对象
        //则将这个样本对象邻域中的所有未被聚类的对象添加到 seedList 中
            //并且设置其中未聚类对象或噪声对象的聚类标号为当前聚类标号
         if (p.getClusterLabel() == DataObject.UNCLASSIFIED ||
             p.getClusterLabel() == DataObject.NOISE) {
             if (p.getClusterLabel() == DataObject.UNCLASSIFIED)  {
                 //在这里将样本对象 p 添加到 seedList 列表中
                 seedList.add(p);
             }
             p.setClusterLabel(clusterID);
        }
     }
}
    //去除当前处理过的样本点
    seedList.remove(j);
    j--;
}

//发现新的簇，所以返回 true
return true;
```

## 10.3 本章小结

本章介绍了网页的特征选择方法和常用的几种分类、聚类算法。分类算法除了已经介绍的 SVM 和 AdaBoost 算法，还有朴素贝叶斯和 K 近邻算法、EM 算法、最大熵等。聚类算法除了已经介绍的 DBScan，还有层次聚类和 KMeans 聚类方法、数据流聚类算法等。