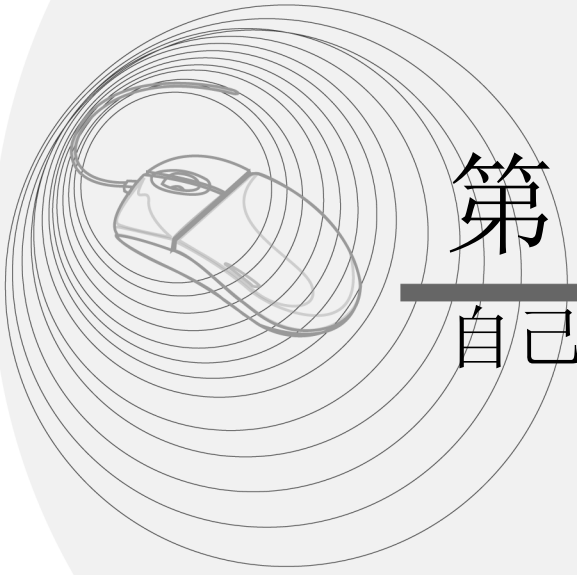
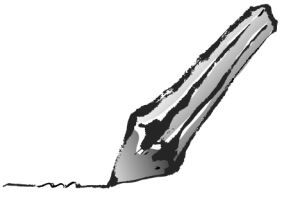
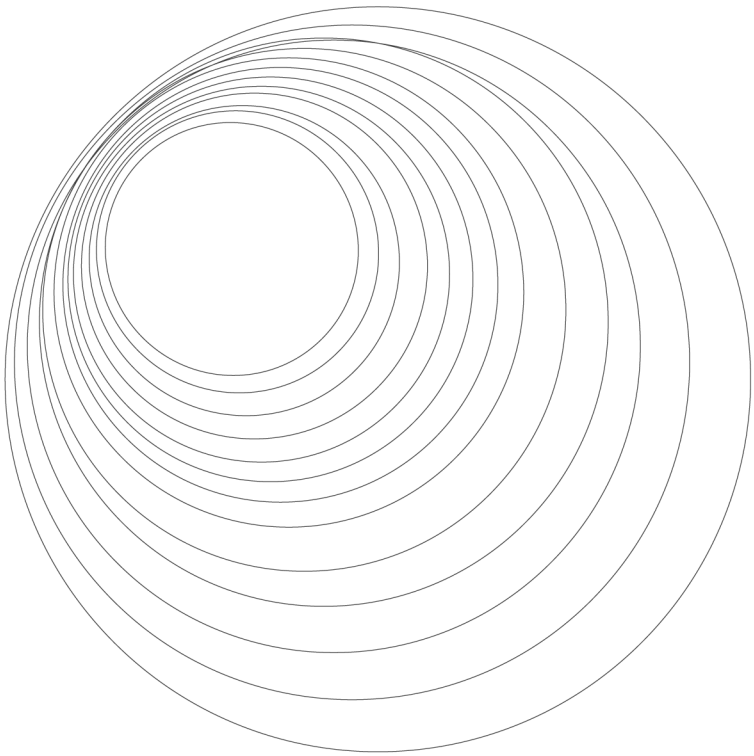


第 1 篇

自己动手抓取数据



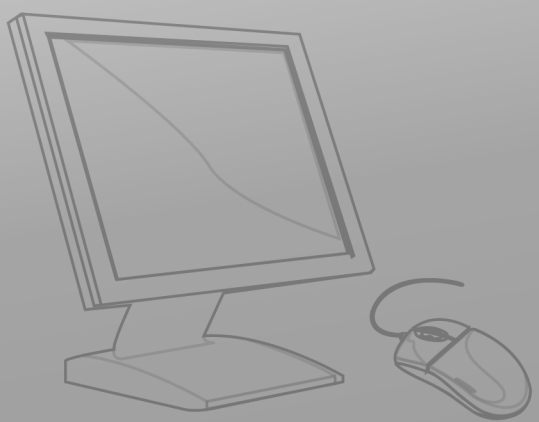


第 1 章

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。

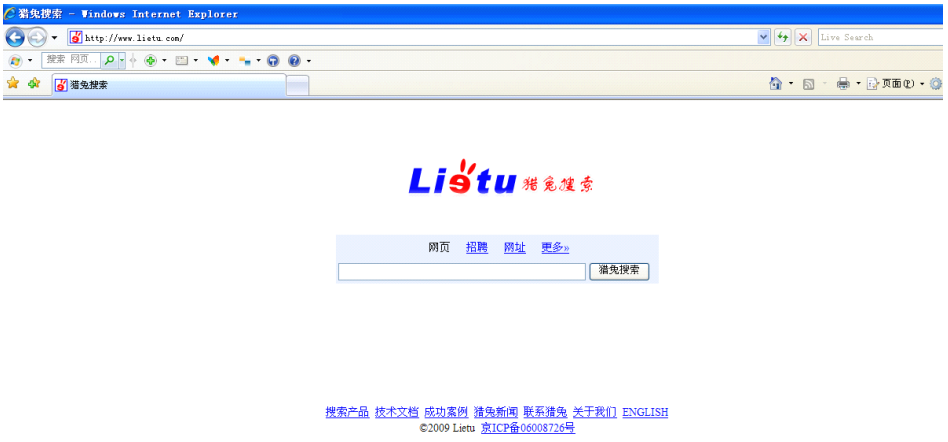


图 1.1 使用浏览器浏览网页

“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/

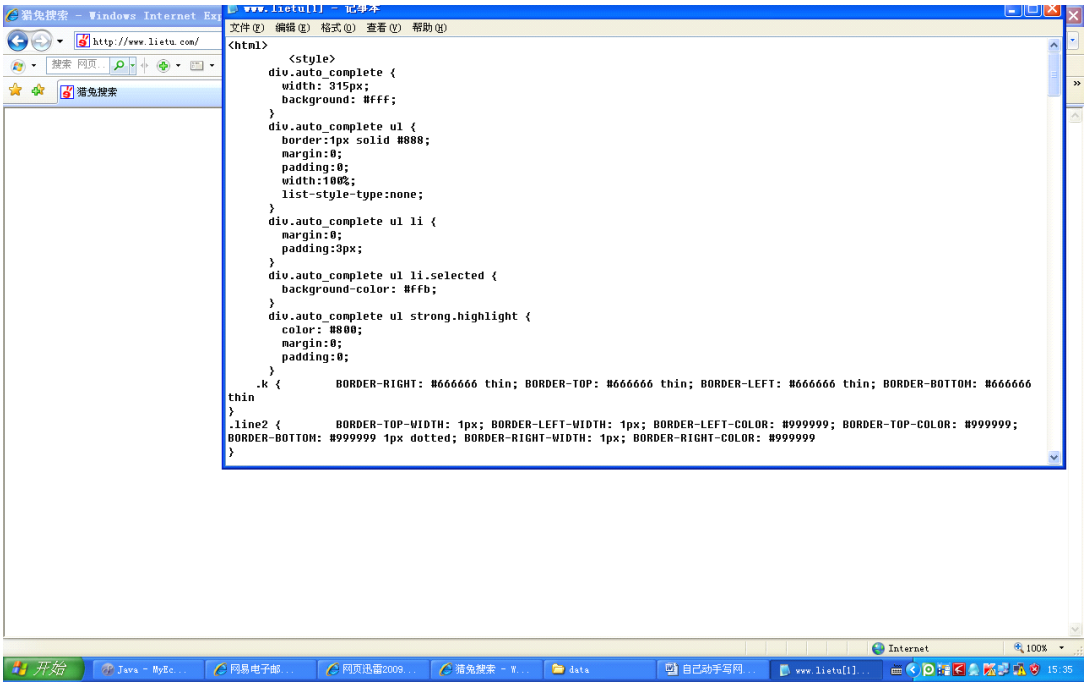


图 1.2　浏览器端源代码

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
```

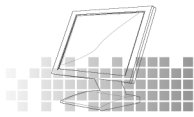
HTTP 状态码表示 HTTP 协议所返回的响应的状态。比如客户端向服务器发送请求，如果成功地获得请求的资源，则返回的状态码为 200，表示响应成功。如果请求的资源不存在，则通常返回 404 错误。

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。

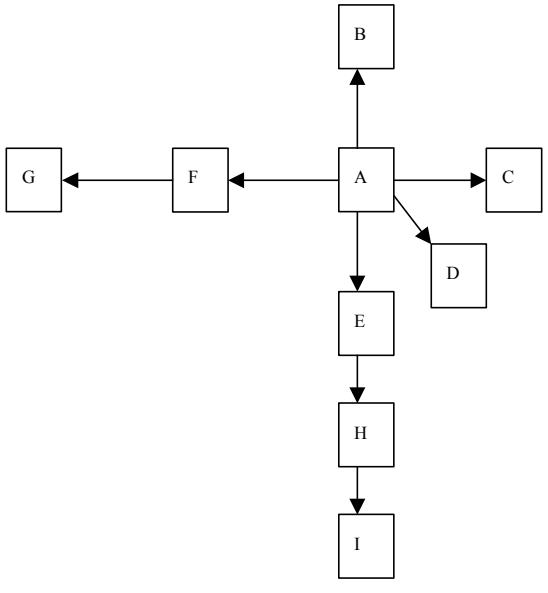


图 1.3 宽度优先遍历过程

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。

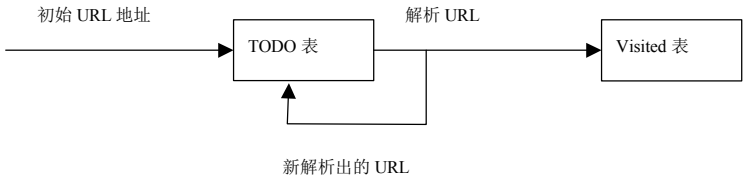


图 1.5　宽度优先爬虫过程

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。

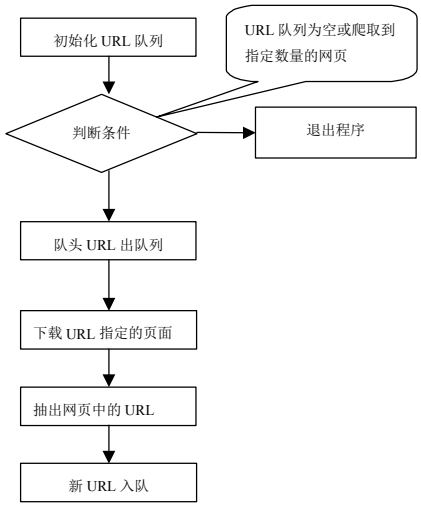


图 1.6　爬虫示例程序结构

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

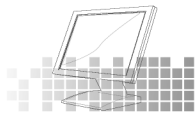
上一节我们讲述了 Berkeley DB 的基础知识，这一节要讲述如何使用 Berkeley DB 来构建一个完整的爬虫队列。

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。

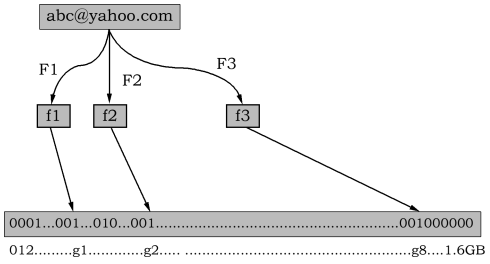


图 1.7　布隆过滤器(Bloom Filter)

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。

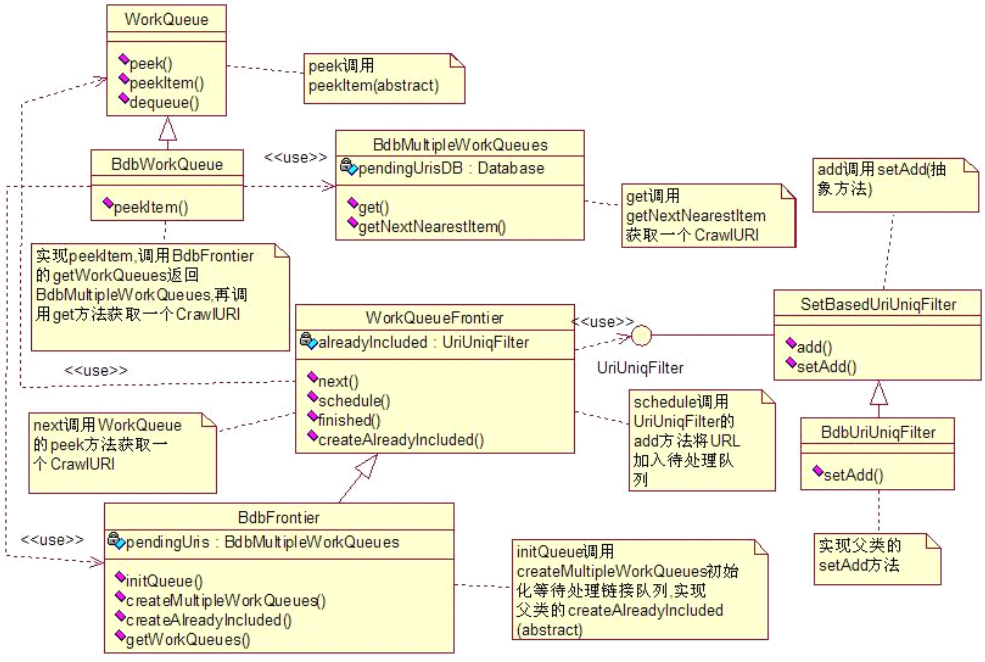


图 1.8　Heritrix 中的 Frontier 架构

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。

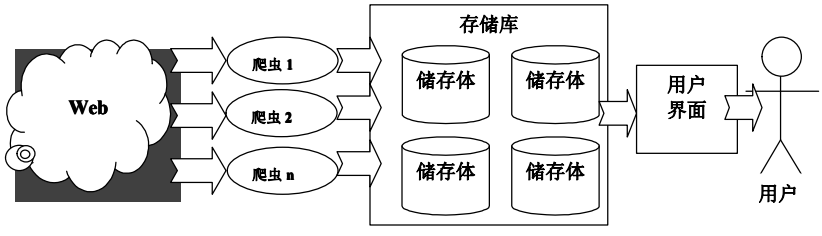


图 1.9　爬虫物理分布简化架构图

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。

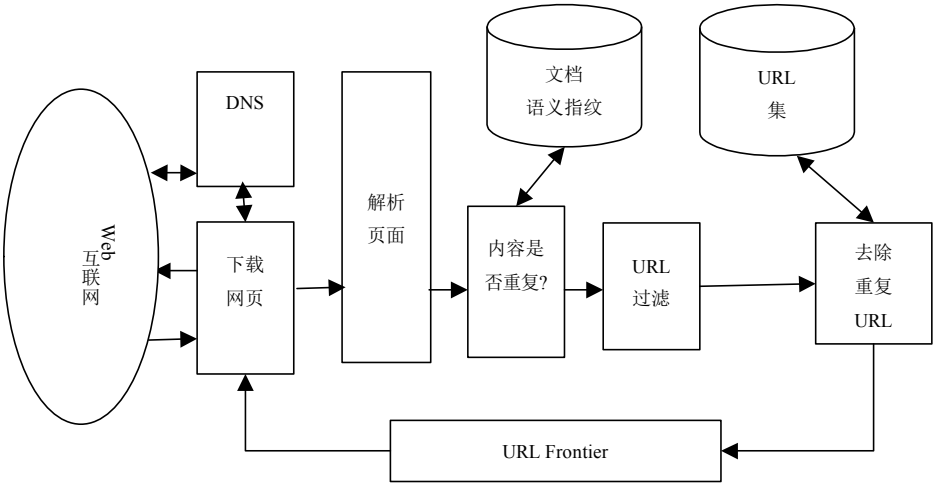


图 1.10　单线程爬虫结构

(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。

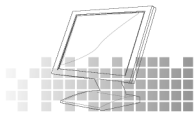
DNS 解析是网络爬虫的瓶颈。由于域名服务的分布式特点，DNS 可能需要多次请求转发，并在互联网上往返，需要几秒有时甚至更长的时间解析出 IP 地址。如果我们的目标是一秒钟抓取数百个文件，这样就达不到性能要求。一个标准的补救措施是引入缓存：最近完成 DNS 查询的网址可能会在 DNS 缓存中找到，避免了访问互联网上的 DNS 服务器。然而，由于抓取礼貌的限制，降低了 DNS 缓存的命中率。

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。

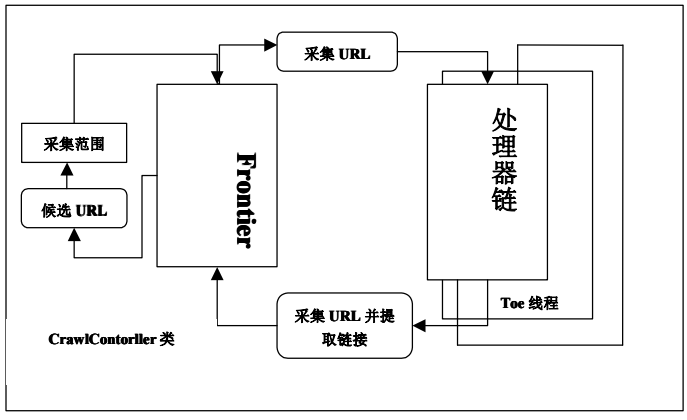


图 1.11　CrawlContorller 类结构

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。

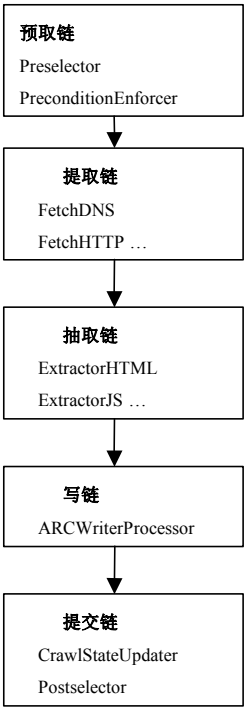


图 1.12 Heritrix 处理链图

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。

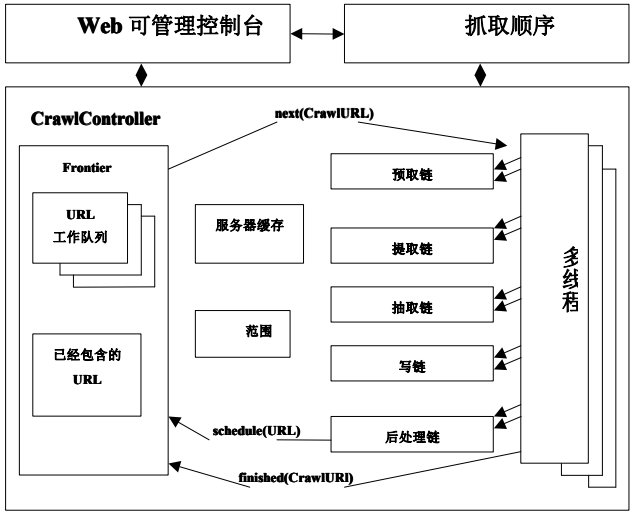


图 1.13　Heritrix 软件的整体结构

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。

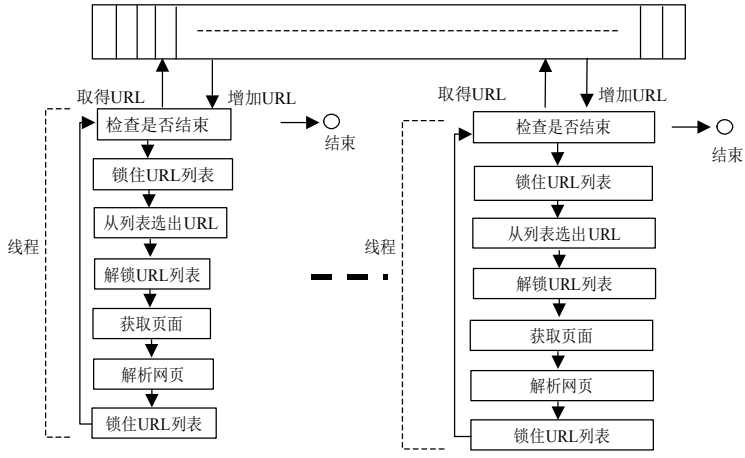


图 1.14　多线程爬虫的结构

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。

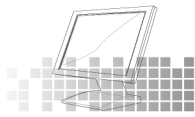
● Ubicrawer 是一个使用 Java 语言编写的分布式爬虫。它没有中央程序，但有一组完全相同的代理组成，分配功能通过主机前后一致的散列计算进行。这里没有重复的页面，除非爬虫崩溃了(然后，另外一个代理就会接替崩溃的代理重新开始抓取)。爬虫设计为高伸缩性。

● 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。