

第 10 章 无比强大的网络爬虫 Heritrix

Lucene 很强大，这点在前面的章节中，已经作了详细介绍。但是，无论多么强大的搜索引擎工具，在其后台，都需要一样东西来支援它，那就是网络爬虫 Spider。

网络爬虫，又被称为蜘蛛 Spider，或是网络机器人、BOT 等，这些都无关紧要，最重要的是要认识到，由于爬虫的存在，才使得搜索引擎有了丰富的资源。

Heritrix 是一个纯由 Java 开发的、开源的 Web 网络爬虫，用户可以使用它从网络上抓取想要的资源。它来自于 www.archive.org。Heritrix 最出色之处在于它的可扩展性，开发者可以扩展它的各个组件，来实现自己的抓取逻辑。本章就来详细介绍一下 Heritrix 和它的各个组件。

10.1 Heritrix 的使用入门

要想学会使用 Heritrix，当然首先得能把它运行起来。然而，运行 Heritrix 并非一件容易的事，需要进行很多配置。在 Heritrix 的文档中对它的运行有详细的介绍，不过尽管如此，笔者仍然花了大量时间，才将其配置好并运行成功。

10.1.1 下载和运行 Heritrix

Heritrix 的下载页面为：<http://crawler.archive.org/downloads.html>。从上面可以链接到 SourceForge 的下载页面。当前 Heritrix 的最新版本为 1.10。

(1) 在下载完 Heritrix 的完整开发包后，解压到本地的一个目录下，如图 10-1 所示。



图10-1 Heritrix的目录结构

其中，Heritrix 所用到的工具类库都存于 lib 下，heritrix-1.10.1.jar 是 Heritrix 的 Jar 包。另外，在 Heritrix 目录下有一个 conf 目录，其中包含了一个很重要的文件：heritrix.properties。

(2) 在 heritrix.properties 中配置了大量与 Heritrix 运行息息相关的参数，这些参数主要是配置了 Heritrix 运行时的一些默认工具类、WebUI 的启动参数，以及 Heritrix 的日志格式等。当第一次运行 Heritrix 时，只需要修改该文件，为其加入 WebUI 的登录名和密码，如图 10-2 所示。

```
# Default commandline startup values.
# Below values are used if unspecified on the command line.
heritrix.cmdline.admin = admin:letmein
heritrix.cmdline.port = 8080
heritrix.cmdline.run = false
heritrix.cmdline.nowui = false
heritrix.cmdline.order =
heritrix.cmdline.jmxserver = false
heritrix.cmdline.jmxserver.port = 8081
```

图10-2 修改Heritrix的WebUI的登录名和密码

其中,用户名和密码是以一个冒号进行分隔,使用者可以指定任何的字符串做为用户名密码,图中所示只不过延续了 Heritrix 以前版本中默认的用户名和密码而已。

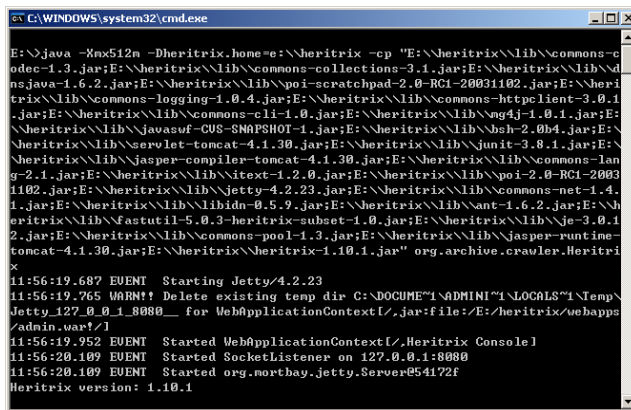
(3) 在设置完登录名和密码后,就可以开始运行 Heritrix 了。Heritrix 有多种方式启动,例如,可以使用 CrawlController,以后台方式加载一个抓取任务,即为程式启动。不过最常见的还是以 WebUI 的方式启动它。

(4) Heritrix 的主类为 org.archive.crawler.Heritrix, 运行它,就可以启动 Heritrix。当然,在运行它的时候,需要为其加上 lib 目录下的所有 jar 包。以下是笔者在命令行中启动 Heritrix 时所使用的批处理文件,此处列出,仅供读者参考(笔者的 Heritrix 目录是位于 E 盘的根目录下,即 E:\heritrix)。

代码 10.1

```
java -Xmx512m -Dheritrix.home=e:\heritrix -cp
"E:\heritrix\lib\commons-codec-1.3.jar;E:\heritrix\lib\commons-collections-3.1.jar;E:\heritrix\lib\commons-logging-1.0.4.jar;E:\heritrix\lib\commons-httpclient-3.0.1.jar;E:\heritrix\lib\commons-cli-1.0.jar;E:\heritrix\lib\mg4j-1.0.1.jar;E:\heritrix\lib\jaswasf-CVS-SNAPSHOT-1.jar;E:\heritrix\lib\bsh-2.0b4.jar;E:\heritrix\lib\servlet-tomcat-4.1.30.jar;E:\heritrix\lib\junit-3.8.1.jar;E:\heritrix\lib\jasper-compiler-tomcat-4.1.30.jar;E:\heritrix\lib\commons-lang-2.1.jar;E:\heritrix\lib\itext-1.2.0.jar;E:\heritrix\lib\poi-2.0-RC1-20031102.jar;E:\heritrix\lib\jetty-4.2.23.jar;E:\heritrix\lib\commons-net-1.4.1.jar;E:\heritrix\lib\libidn-0.5.9.jar;E:\heritrix\lib\ant-1.6.2.jar;E:\heritrix\lib\fastutil-5.0.3-heritrix-subset-1.0.jar;E:\heritrix\lib\je-3.0.12.jar;E:\heritrix\lib\commons-pool-1.3.jar;E:\heritrix\lib\jasper-runtime-tomcat-4.1.30.jar;E:\heritrix\heritrix-1.10.1.jar"
org.archive.crawler.Heritrix
```

(5) 在上面的批处理文件中,将 Heritrix 所用到的所有的第三方 Jar 包都写进了 classpath 中,同时执行了 org.archive.crawler.Heritrix 这个主类。图 10-3 为 Heritrix 启动时的画面。



```
C:\WINDOWS\system32\cmd.exe
E:\>java -Xmx512m -Dheritrix.home=e:\heritrix -cp "E:\heritrix\lib\commons-codec-1.3.jar;E:\heritrix\lib\commons-collections-3.1.jar;E:\heritrix\lib\commons-logging-1.0.4.jar;E:\heritrix\lib\commons-httpclient-3.0.1.jar;E:\heritrix\lib\commons-cli-1.0.jar;E:\heritrix\lib\mg4j-1.0.1.jar;E:\heritrix\lib\jaswasf-CVS-SNAPSHOT-1.jar;E:\heritrix\lib\bsh-2.0b4.jar;E:\heritrix\lib\servlet-tomcat-4.1.30.jar;E:\heritrix\lib\junit-3.8.1.jar;E:\heritrix\lib\jasper-compiler-tomcat-4.1.30.jar;E:\heritrix\lib\commons-lang-2.1.jar;E:\heritrix\lib\itext-1.2.0.jar;E:\heritrix\lib\poi-2.0-RC1-20031102.jar;E:\heritrix\lib\jetty-4.2.23.jar;E:\heritrix\lib\commons-net-1.4.1.jar;E:\heritrix\lib\libidn-0.5.9.jar;E:\heritrix\lib\ant-1.6.2.jar;E:\heritrix\lib\fastutil-5.0.3-heritrix-subset-1.0.jar;E:\heritrix\lib\je-3.0.12.jar;E:\heritrix\lib\commons-pool-1.3.jar;E:\heritrix\lib\jasper-runtime-tomcat-4.1.30.jar;E:\heritrix\heritrix-1.10.1.jar" org.archive.crawler.Heritrix
11:56:19.687 EVENT Starting Jetty/4.2.23
11:56:19.765 WARN!! Delete existing temp dir C:\DOCUMENTS\ADMINISTRATOR\LOCALS~1\Temp\Jetty_127.0.0.1_8080_ for WebApplicationContext[/,jar:file:/E:/heritrix/webapps/admin.war!/]
11:56:19.952 EVENT Started WebApplicationContext[/,Heritrix Console]
11:56:20.109 EVENT Started SocketListener on 127.0.0.1:8080
11:56:20.109 EVENT Started org.morthay.jetty.Server#54172f
Heritrix version: 1.10.1
```

图10-3 Heritrix的启动画面

(6) 在这时，Heritrix 的后台已经对服务器的 8080 端口进行了监听，只需要通过浏览器访问 <http://localhost:8080>，就可以打开 Heritrix 的 WebUI 了。如图 10-4 所示。

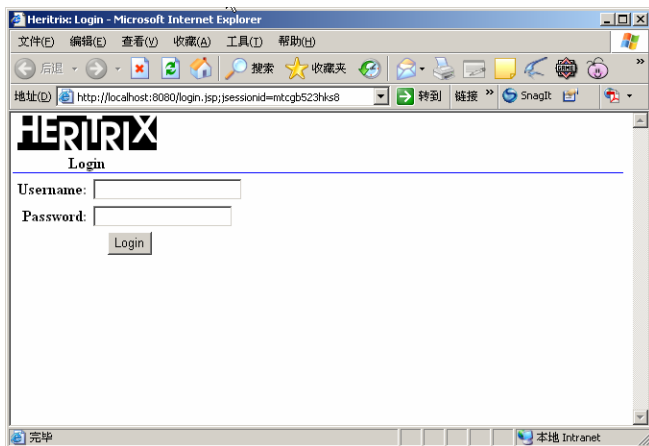


图10-4 Heritrix的WebUI的登录界面

(7) 在这个登录界面，输入刚才在 Heritrix.properties 中预设的 WebUI 的用户名和密码，就可以进入如图 10-5 所示的 Heritrix 的 WebUI 的主界面。

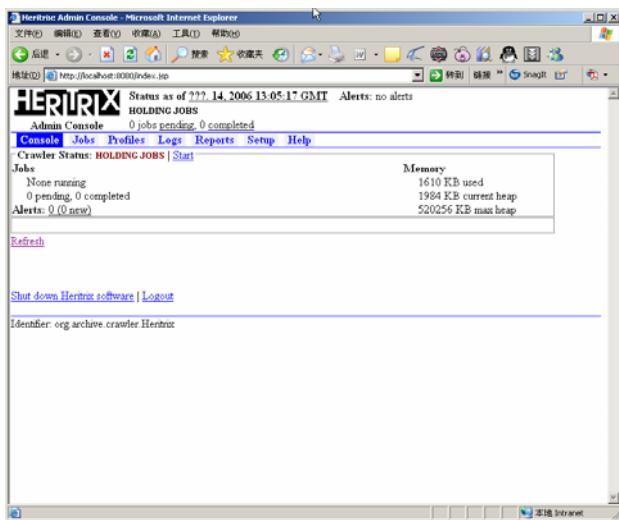


图10-5 登录后的界面

(8) 当看到这个页面的时候，就说明 Heritrix 已经成功的启动了。在页面的中央有一道状态栏，用于标识当前正在运行的抓取任务。如图 10-6 所示：

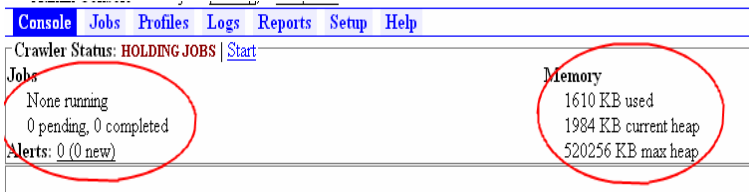


图10-6 抓取任务的状态栏

在这个 WebUI 的帮助下，用户就可以开始使用 Heritrix 来抓取网页了。

10.1.2 在 Eclipse 里配置 Heritrix 的开发环境

讲完了通过命令行方式启动的 Heritrix，当然要讲一下如何在 Eclipse 中配置 Heritrix 的开发环境，因为可能需要对代码进行调试，甚至修改一些它的源代码，来达到所需要的效果。下面来研究一下 Heritrix 的下载包。

(1) webapps 文件夹是用来提供 Servlet 引擎的，也就是提供 Heritrix 的 WebUI 的部分，因此，在构建开发环境时必不可少。conf 文件夹是用来提供配置文件的，因此也需要配置进入工程。Lib 目录下主要是 Heritrix 在运行时需要用到的第三方的软件，因此，需要将其设定到 Eclipse 的 Build Path 下。最后就是 Heritrix 的 jar 包了，将其解压，可以看到其内部的结构如图 10-7 所示。

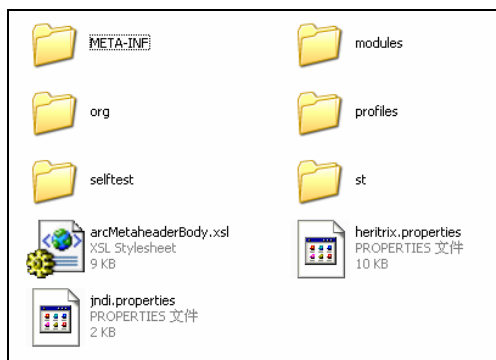


图10-7 Heritrix的Jar包的结构

(2) 根据图 10-7 所示，应该从 Heritrix 的源代码包中把这些内容取出，然后放置到工程中来。Heritrix 的源代码包解压后，只有两个文件夹，如图 10-8 所示。

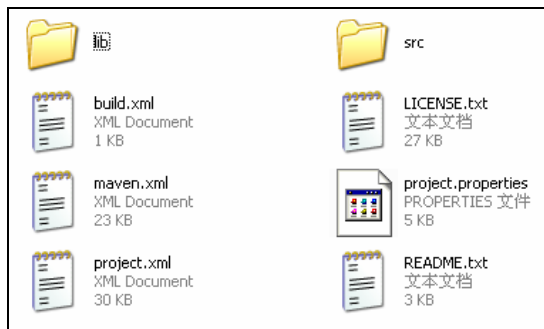


图10-8 Heritrix的源代码包的结构

(3) 只需在 src 目录下，把图 10-7 中的内容配全，就可以将工程的结构完整了。如图 10-9 所示。

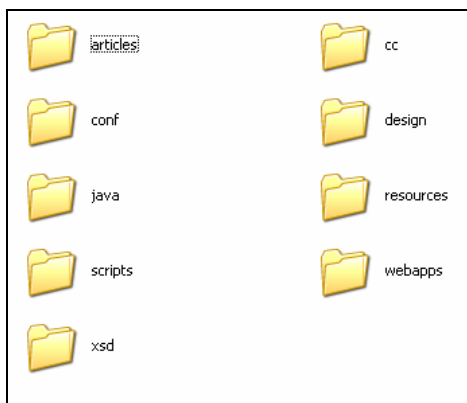


图10-9 src目录下的内容

(4) 图 10-10 和图 10-11 是笔者机器上的 Heritrix 在 Eclipse 中的工程配置好后的截图，以及 workspace 中文件夹的预览。

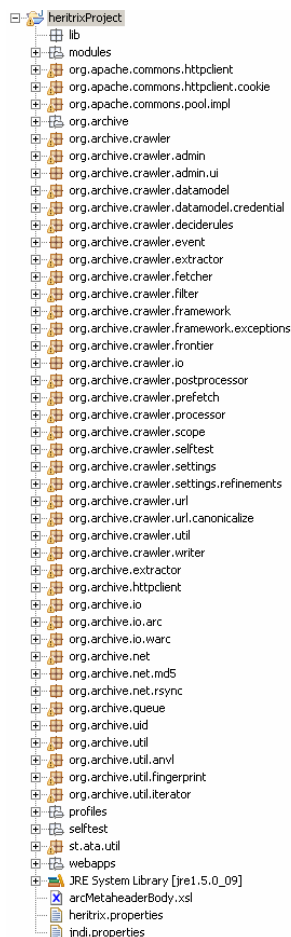


图10-10 Eclipse工程视图下的包结构

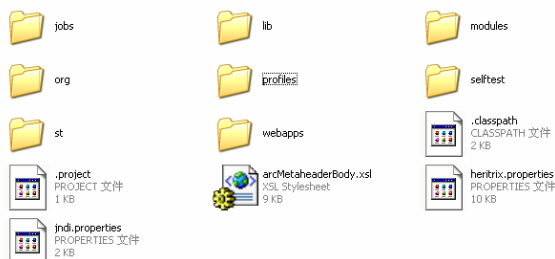


图10-11 文件夹中的工程

其中,org 目录内是 Heritrix 的源代码,另外,笔者将 conf 目录去掉了,直接将 heritrix.properties 文件放在了工程目录下。在图 10-10 中,读者可能没有看到 Heritrix 所使用到的 Jar 包,这是因为在工程视图中,它们被过滤器过滤掉了,实际上,所有 lib 目录下的 jar 包都已经被加进了 build path 中。

(5) 不过,读者很有可能遇到这样的情况,那就是在将所有的 jar 包都导入后,工程编译

```

heritrix:Project
├── lib
├── modules
│   ├── org.apache.commons.httpclient
│   ├── org.apache.commons.httpclient.cookie
│   ├── org.apache.commons.pool.impl
│   ├── org.archive
│   ├── org.archive.crawler
│   ├── org.archive.crawler.admin
│   ├── org.archive.crawler.admin.ui
│   ├── org.archive.crawler.datamodel
│   ├── org.archive.crawler.datamodel.credential
│   ├── org.archive.crawler.deciderules
│   ├── org.archive.crawler.event
│   ├── org.archive.crawler.extractor
│   ├── org.archive.crawler.fetcher
│   ├── org.archive.crawler.filter
│   ├── org.archive.crawler.framework
│   ├── org.archive.crawler.framework.exceptions
│   ├── org.archive.crawler.frontier
│   ├── org.archive.crawler.io
│   ├── org.archive.crawler.postprocessor
│   ├── org.archive.crawler.prefetch
│   ├── org.archive.crawler.processor
│   ├── org.archive.crawler.scope
│   ├── org.archive.crawler.selftest
│   ├── org.archive.crawler.settings
│   ├── org.archive.crawler.settings.refinements
│   ├── org.archive.crawler.url
│   ├── org.archive.crawler.url.canonicalize
│   ├── org.archive.crawler.util
│   └── org.archive.crawler.writer

```

(6) 随便打开一个出错的文件，如图 10-13 所示，会发现大量的错误都来自于“assert”关键字。这种写法似乎 Eclipse 不认识。

```

11         throws IOException {
12             assert (null == host) || ((0 != host.length()) &&
13                 assert 0 == uriPath.length();
14             assert '/' == uriPath.charAt(0) : "uriPath: " + uriPath;
15             -1 == uriPath.indexOf("/") : "uriPath: " + uriPath;
16             assert -1 == uriPath.indexOf("//") : "uriPath: " + uriPath;
17             uriPath.endsWith("/") : "uriPath: " + uriPath;
18             assert (null == query) || (-1 == query.indexOf('/'))
19                 : "query: " + query;
20             assert (null == suffix)
21                 || ((0 != suffix.length()) && (-1 == suffix.indexOf('/')))
22                 : "suffix: " + suffix;
23             assert 0 == baseDir.length();
24             assert maxSegLen >= 1 : "maxSegLen: " + maxSegLen;
25             assert maxPathLen >= 1;
26             assert maxPathLen >= maxSegLen
27                 : "maxSegLen: " + maxSegLen + " maxPathLen: " + maxPathLen;
28             assert 0 == dirFile.length();
29             assert -1 == dirFile.indexOf("/") : "dirFile: " + dirFile;
30             assert null == characterSep;
31             assert (null == dotBegin) || (0 != dotBegin.length());
32             assert (null == dotEnd) || (dotEnd.endsWith(".") && dotEnd
33                 == tooLongDir.length());
34             assert
35                 '/' == tooLongDir.charAt(0) : "tooLongDir: " + tooLongDir;

```

(7) 解决问题的关键在于，Eclipse 的编译器不认识 `assert` 这个关键字。可以在“选项”菜单中将编译器的语法样式改为 5.0，也就是 JDK1.5 兼容的语法，然后重启编译整个工程就可以了。如图 10-14 所示。

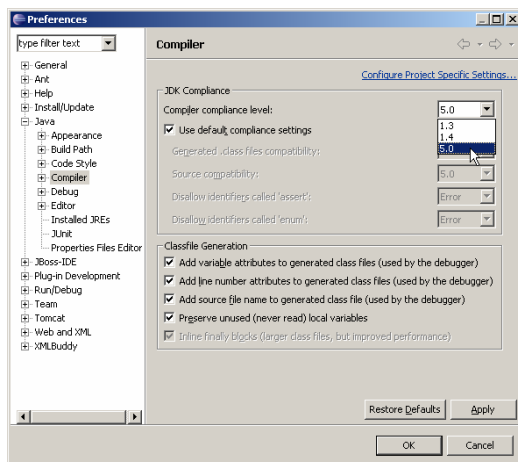


图10-14 改变编译器的语法等级

(8) 在重新编译整个工程后，笔者的 Eclipse 中仍然出现了一个编译错误，那就是在 `org.archive.io.ArchiveRecord` 类中，如图 10-15 所示。

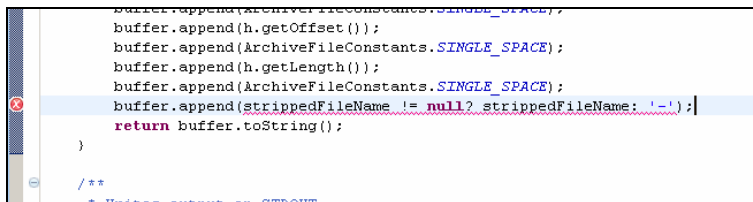


图10-15 一个仍然存在的错误

从代码看来，这是因为在使用条件表达式，对 `strippedFileName` 这个 `String` 类型的对象赋值时，操作符的右则出现了一个 `char` 型的常量，因此影响了编译。暂且不论为什么在 `Heritrix` 的源代码中会出现这样的错误，解决问题的办法就是将 `char` 变成 `String` 类型，即：

```
buffer.append(strippedFileName != null? strippedFileName: "-");
```

(9) 当这样修改完后，整个工程的错误就被全部解决了，也就可以开始运行 `Heritrix` 了。在 Eclipse 下运行 `org.archive.crawler.Heritrix` 类，如图 10-16 所示。

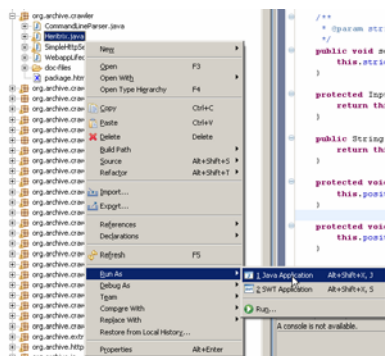


图10-16 在Eclipse中运行Heritrix

(10) 当看到图 10-17 所示的界面时，就说明 `Heritrix` 已经成功的在 Eclipse 中运行，也就意味着可以使用 Eclipse 来对 `Heritrix` 进行断点调试和源码修改了。

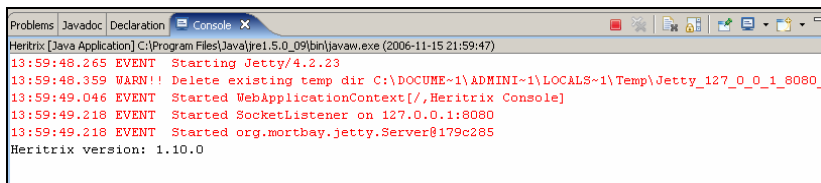


图10-17 在Eclipse中成功的运行

10.1.3 创建一个新的抓取任务

在 10.1.1 小节中，已经看到了 Heritrix 成功运行后的 WebUI，接下来，就要带领读者来创建一个新的抓取任务。

(1) 单击 WebUI 菜单栏上的“Jobs”标签，就可以进入任务创建页面。如图 10-18 所示。



图10-18 菜单栏上的“Jobs”标签

(2) 在任务创建页面中，有 4 种创建任务的方式，如图 10-19 所示，具体含义如下。

- Based on existing job: 以一个已有的抓取任务为模板，创建所有抓取属性和抓取起始URL的列表。
- Based on a recovery: 在以前的某个任务中，可能设置过一些状态点，新的任务将从这个设置的状态点开始。
- Based on a profile: 专门为不同的任务设置了一些模板，新建的任务将按照模板来生成。
- With defaults: 这个最简单，表示按默认的配置来生成一个任务。

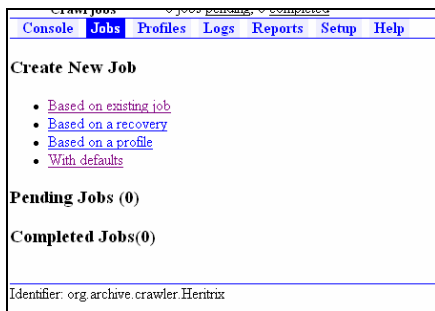


图10-19 “任务”菜单中

在 Heritrix 中，一个任务对应一个描述文件。这个描述文件的默认的名称为 order.xml。每次创建一个新任务时，都相当于生成了一个 order.xml 的文件。文件中详细记录了 Heritrix 在运行时需要的所有信息。例如，它包括该用户所选择的 Processor 类、Frontier 类、Fetcher 类、抓取时线程的最大数量、连接超时的最大等待时间等信息。上面所说的 4 种创建抓取任务的方式，其实都是在生成一个 order.xml 文件。其中，第 4 种 With defaults，则是直接拷贝默认的 order.xml 文件。在所创建的 Eclipse 工程或是命令行启动的 Heritrix 下载包中，该默认的

order.xml 文件均是放于 profiles\default 目录下的。

关于 order.xml 的细节，在此还不必深究。因为它里面所有的内容，都会在 WebUI 上看到。

(3) 单击 With defaults 链接，创建一个新的抓取任务，如图 10-20 所示。

Console Jobs Profiles Logs Reports Setup Help

Create new crawl job based on default profile

Name of new job:

Description:

Seeds:

Modules Submodules Settings Overrides Submit job

Identifier: org.archive.crawler Heritrix

图10-20 新的抓取任务

(4) 在新建任务的名称上，填入“Sohu_news”，表示该抓取任务将抓取搜狐的新闻信息。在 Description 中随意填入字符，然后再在 seeds 框中，填入搜狐新闻的网址。这里需要解释一下 seeds 的含义。所谓 seeds，其实指的是抓取任务的起始点。每次的抓取，总是需要从一个起始点开始，在得到这个起始点网页上的信息后，分析出新的地址加入抓取队列中，然后循环抓取，重复这样的过程，直到所有链接都分析完毕。

(5) 在图 10-20 中，设置了搜狐新闻的首页为种子页面，以此做为起始点。用户在使用时，也可以同时输入多个种子，每个 URL 地址单独写在一行上，如图 10-21 所示。

Name of new job:

Description:

Seeds:

Modules Submodules Settings Overrides Submit job

图10-21 多个种子的情况

当然，凭着目前的设置，还没法开始抓取网页，还需要对这个任务进行详细的设置。

10.1.4 设置抓取时的处理链

在图 10-21 中，seeds 文本框下有一排按钮，单击“Modules”按钮，就进入了配置抓取时的处理链的页面，如图 10-22 所示。

Job Sohu news: [Modules](#) [Submodules](#) [Settings](#) [Overrides](#) [Refinements](#) [Submit job](#)

Select Modules and Add/Remove/Order Processors

Use this page to choose the main modules Heritrix should use for crawling and to add/remove/order process modules and processors.

Select Crawl Scope

Current selection: org.archive.crawler.framework.CrawlScope
Crawl scope

Available alternatives:

Select URI Frontier

Current selection: org.archive.crawler.settings.ModuleType
Frontier

Available alternatives:

Select Pre Processors *Processors that should run before any fetching*

Select Fetchers *Processors that fetch documents using various protocols*

Select Extractors *Processors that extract links from URLs*

Select Writers *Processors that write documents to archive files*

Select Post Processors *Processors that do cleanup and feed the Frontier with new URLs*

图10-22 配置处理链的页面

从上而下，可以看到，需要配置的内容共有 7 项，其中 CrawlScope 和 Frontier 是两个最重要的组件。

CrawlScope 用于配置当前应该在什么范围内抓取网页链接。比如，如果选择 BroadScope，则表示当前抓取的范围不受限制，但如果选择了 HostScope，则表示抓取的范围在当前的 Host 内。

从笔者的经验看来，在抓取时，无论是 HostScope 或 PathScope 都不能真正的限制到抓取的内容。需要对 Scope 内的代码进行一定的修改才可以，因此，暂时选择 BroadScope 来充当示例中的范围限定，其实也就是对范围不做任何的限定。即从 news.sohu.com 开始，抓取任何可以抓取到的信息。如图 10-23 所示。

Select Crawl Scope

Current selection: org.archive.crawler.scope.BroadScope
BroadScope: A scope for broad crawls. Crawls made with this scope will not be limited to the hosts or domains of its seeds. NOTE: BroadScoped crawls will eventually run out of memory (See Release Notes).

Available alternatives:

图10-23 设置Scope

Frontier 则是一个 URL 的处理器，它将决定下一个被处理的 URL 是什么。同时，它还会将经由处理器链所解析出来的 URL 加入到等待处理的队列中去。在例子中，使用 BdbFrontier 类来做为处理器，全权掌管 URL 的分配。如图 10-24 所示。

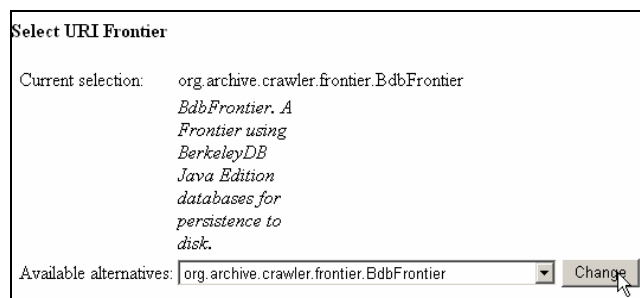


图10-24 设置Frontier

除了这两个组件外，还有 5 个队列要配。这五个队列根据先后的顺序，就依次组成了 Heritrix 的整个处理器链。5 个队列的含义分别如下：

(1) PreProcessor: 这个队列中，所有的处理器都是用来对抓取时的一些先决条件做判断的。比如判断 robot.txt 的信息等，它是整个处理器链的入口。如图 10-25 所示。

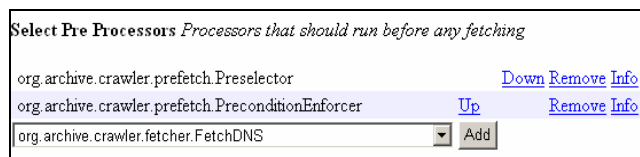


图10-25 设置PreProcessor

(2) Fetcher: 从名称上看，它用于解析网络传输协议，比如解析 DNS、HTTP 或 FTP 等。在演示中，主要使用 FetchDNS 和 FetchHTTP 两个类。如图 10-26 所示。

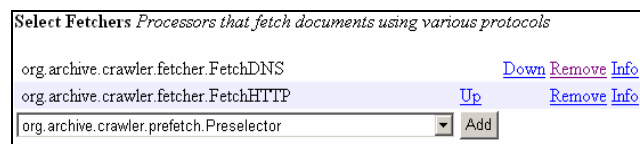


图10-26 设置Fetcher

(3) Extractor: 它的名字就很好的揭示了它的作用。它主要用是于解析当前获取到的服务器返回内容，这些内容通常是以字符串形式缓存的。在这个队列中，包括了一系列的工具，如解析 HTML、CSS 等。在解析完毕，取出页面中的 URL 后，将它们放入队列中，等待下次继续抓取。在演示中，使用两种 Extractor，即 ExtractorHTTP 和 ExtractorHTML。如图 10-27 所示。

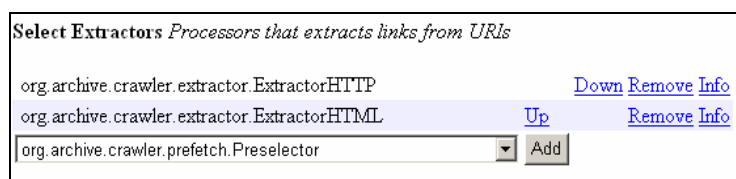


图10-27 设置Extractor

(4) Writer: 主要是用于将所抓取到的信息写入磁盘。通常写入磁盘时有两种形式，一种

是采用压缩的方式写入，在这里被称为 Arc 方式，另一种则采用镜象方式写入。当然处理起来，镜象方式要更为容易一些，因此，在演示中命名用镜象 Mirror 方式。如图 10-28 所示。

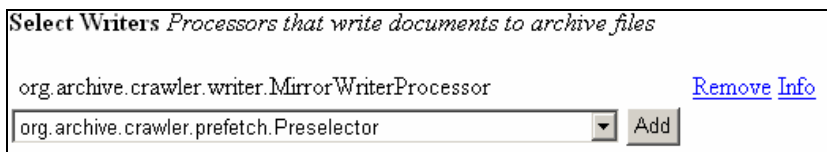


图10-28 设置Writer

(5)PostProcessor: 在整个抓取解析过程结束后，进行一些扫尾的工作，比如将前面 Extractor 解析出来的 URL 有条件的加入到待处理队列中去。如图 10-29 所示。

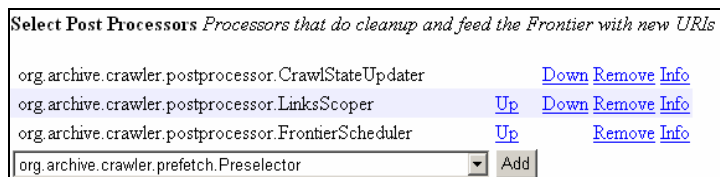


图10-29 设置PostProcessor

值得一提的是，在处理器链的设置过程中，每一个队列中的处理器都是要分先后顺序的，信息的处理流程实际上是不可逆的，因此，在设置时，可以看见在队列的右侧总是有“Up”、“Down”和“Remove”这样的操作，以帮助能够正确的设置其顺序。

在设置完 Hertrix 所需的处理链后，仍然还不能够马上开始抓取任务，还需对默认的运行时参数做一些修改，以适应真正的需要。

10.1.5 设置运行时的参数

在设置完处理链后，在页面顶部或底部都可以找到如图 10-30 所示的菜单项，单击“Settings”链接，就进入了属性设置的页面，如图 10-30 所示。



图10-30 进入“Settings”

在属性设置页面上有非常多的输入域，Heritrix 在抓取网页时，这些域是用来对各个组件的值进行预设，如图 10-31 所示。



图10-31 属性配置页面

由于页面上的内容非常多，使用者可能无法全部了解它们的作用。所以 Heritrix 提供了一个辅助功能，来在最大程度上让使用者了解每个参数的含义。如图 10-32 所示。

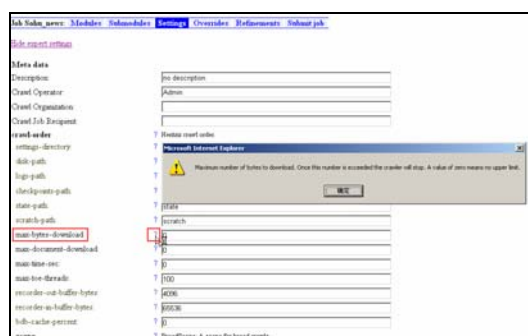


图10-32 属性提示

可以看到，在每个属性的右侧都有一个小问号，当单击问号时，就会弹出一个 Javascript 的 Alert 提示框，上面介绍了当前属性的作用。例如，在上图中单击“max-bytes-download”属性，通过 Alert 的提示可以知道，它表示的是抓取器最大下载的字节数，当下载字节数超过这个属性上所设定的值时，抓取就会自动停止。另外，如果将这个值设为 0，则表示没有限制。

事实上，当在第一次使用 Heritrix 时，所需要设置的参数并不多，以默认设置为主。以下就来介绍一些必须要在第一次使用时就要配置好的参数。

1. max-toe-threads

该参数的含义很容易了解，它表示 Heritrix 在运行该抓取任务时，为任务分配多少个线程进行同步抓取。该参数的默认值为 100，而事实上根据笔者的经验，在机器配置和网络均很好的情况下，设置 50 个线程数就已经足够使用了。

2. HTTP-Header

在 HTTP-Header 这个属性域下面，包括两个属性值“user-agent”和“from”。默认情况下，这两个属性的值如图 10-33 所示。

http-headers	? HTTP headers.
user-agent:	? Mozilla/5.0 (compatible; heritrix/@VERSION@ +PROJECT_URL_HERE)
from:	? CONTACT_EMAIL_ADDRESS_HERE

图10-33 默认的情况

很明显，这样的值是无法完成真实的 HTTP 协议的模拟的，所以，必须要将值改掉。图 10-34 是笔者机器上的一种配置，读者可以借鉴。

http-headers	? HTTP headers.
user-agent	? Mozilla/5.0 (compatible; heritrix/1.10.0 +http://192.168.200.26)
from	? test@luceneheritrix.com

图10-34 一种正确的配置

- “@VERSION@” 字符串需要被替换成Heritrix的版本信息。
- “PROJECT_URL_HERE” 可以被替换成任何一个完整的URL地址。
- “from” 属性中不需要设置真实的E-mail地址，只需是格式正确的邮件地址就可以了。

当正确设置了上述的两个属性后，Heritrix 就具备了运行的条件。单击“Submit”链接，提交这个抓取任务，如图 10-35 所示。

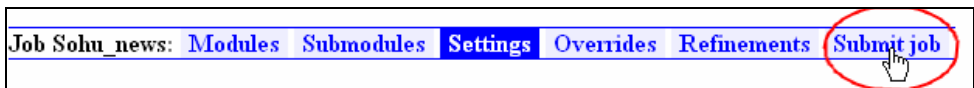


图10-35 提交任务“Submit job”

10.1.6 运行抓取任务

(1) 当单击“Submit job”链接后，会看到图 10-36 所示的页面。图中最上方很清楚的显示了“Job created”，这表示刚才所设置的抓取任务已经被成功的建立。同时，在下面的“Pending Jobs”一栏，可以清楚的看到刚刚被创建的 Job，它的状态目前为“Pending”。

Crawl jobs		1 jobs pending, 4 completed				
Console	Jobs	Profiles	Logs	Reports	Setup	Help
Job created						
Create New Job						
<ul style="list-style-type: none">• Based on existing job• Based on a recovery• Based on a profile• With defaults						
Pending Jobs (1)						
Name	Status	Options				
Sohu_news	Pending	View order	Edit configuration	Journal	Delete	

图10-36 Job提交后的页面

(2) 下面启动这个任务。回到“Console”界面上，可以看到，如图 10-37 所示，刚刚创建的任务已经显示了出来，等待我们开始它。

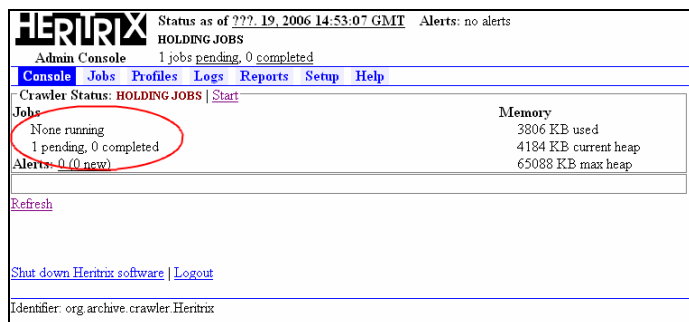


图10-37 Job提交后的Console界面

(3) 在面版的右侧，它显示了当前 Java 虚拟机的一些状态，如图 10-38 所示，可以看到当前的堆大小为 4184KB，而已经被使用了 3806KB，另外，最大的堆内容可以达到 65088KB，也就是在 64M 左右。

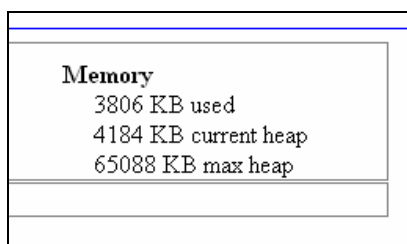


图10-38 内存状态显示

(4) 此时，单击面版中的“Start”链接，就会将此时处于“Pending”状态的抓取任务激活，令其开始抓取

(5) 在图 10-39 中，刚才还处于“Start”状态的链接已经变为了 Hold 状态。这表明，抓取任务已经被激活。

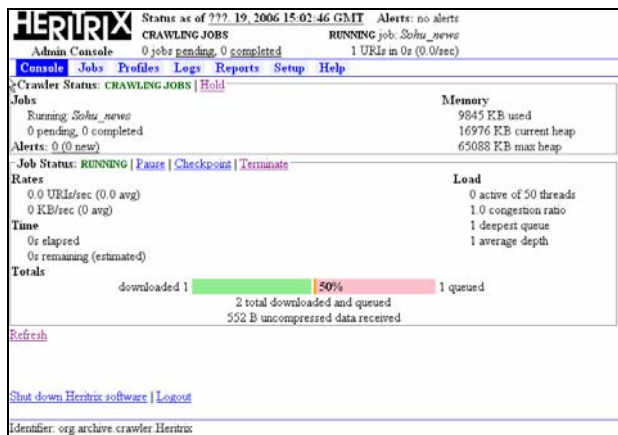


图10-39 抓取开始

(6) 此时，面版中出现了一条抓取状态栏，它清楚的显示了当前已经被抓取的链接数量，另外还有在队列中等待被抓取的链接数量，然后用一个百分比显示出来。

(7) 在绿红相间的长条左侧，是几个实时的运行状态，其中包括抓取的平均速度 (KB/s) 和每秒钟抓取的链接数 (URIs/sec)，另外的统计还包括抓取任务所消耗的时间和剩余的时间，不过这种剩余时间一般都不准，因为 URI 的数量总是在不断变化，每当分析一个网页，就会有新的 URI 加入队列中。如图 10-40 所示。

Rates
0.0 URIs/sec (0.0 avg)
0 KB/sec (0 avg)
Time
1s elapsed
1s remaining (estimated)
Totals

图10-40 抓取的速度和时间

(8) 在绿红相间的长条右侧，是当前的负载，它显示了当前活跃的线程数量，同时，还统计了 Heritrix 内部的所有队列的平均长度。如图 10-41 所示。

Load
0 active of 50 threads
1.0 congestion ratio
1 deepest queue
1 average depth

图10-41 线程和队列负载

(9) 从图 10-40 和图 10-41 中看到，真正的抓取任务还没有开始，队列中的总 URI 数量，以及下载的速率都还基本为 0。这应该还处于接收种子 URL 的网页信息的阶段。让我们再来看一下当 Heritrix 运行一段时间后，整个系统的资源消耗和进度情况。

(10) 在图 10-42 中，清楚的看到系统的资源消耗。其中，每秒下载的速率已经达到了 23KB，另外，平均每秒有 19.3 个 URI 被抓取。在负载方面，初设的 50 个线程均处于工作状态，最长的队列长度已经达到了 415 个 URI，平均长度为 5。从进度条上看，总共有 3771 个 URI 等待抓取，已经完成了 718 个 URI 的抓取，另外，下载的字节总数也已经达到了 1390KB。再观察一下左边，仅用时 32s。可见，多线程抓取的速度还是很快的。

HERITRIX Status as of 777.19.2006.15.03.18 GMT Alerts: no alerts	
Admin Console CRAWLING JOBS RUNNING job: Soku_server	
0 jobs pending, 0 completed 718 URIs in 32s (19.3/sec)	
Console Jobs Profiles Logs Reports Setup Help	
Crawler Status: CRAWLING JOBS Hold	
Jobs	Memory
Running: Soku_server	21154 KB used
0 pending, 0 completed	22708 KB current heap
Alerts: 0 (0 new)	65088 KB max heap
Job Status: RUNNING Pause Checkpoint Terminate	
Rates	Load
19.3 URIs/sec (19.3 avg)	50 active of 50 threads
23 KB/sec (23 avg)	5.74 congestion ratio
Time	415 deepest queue
32s elapsed	5 average depth
2m50s remaining (estimated)	
Totals	
downloaded 718	15% 3771 queued
4539 total downloaded and queued	
1390 KB uncompressed data received	
Refresh	
Shut down Heritrix software Logout	
Identifier: org.archive.crawler.Heritrix	

图10-42 系统运行一段时间后的情况

(11) 不过，当抓取继续进行，观察 Java 虚拟机的内存使用，发现其已达饱和状态。64M 的最大 Heap 显然不够用。如图 10-43 所示。

Memory
47476 KB used
65088 KB current heap
65088 KB max heap

图10-43 Java虚拟机的内存使用

(12) 由于这仅是一次演示，可以忽略内存的影响。但在真正的开发过程中，使用 Heritrix 时，至少应为其分配 512M 的最大 HeapSize，也就是在启动它时，应该设置 -Xmx512m 这个属性。在使用命令行方式启动 Heritrix 的脚本中，笔者已经为其加入了该参数，而如果要在使用 Eclipse 启动 Heritrix 时也设置该参数，具体的设置方法如图 10-44 所示。

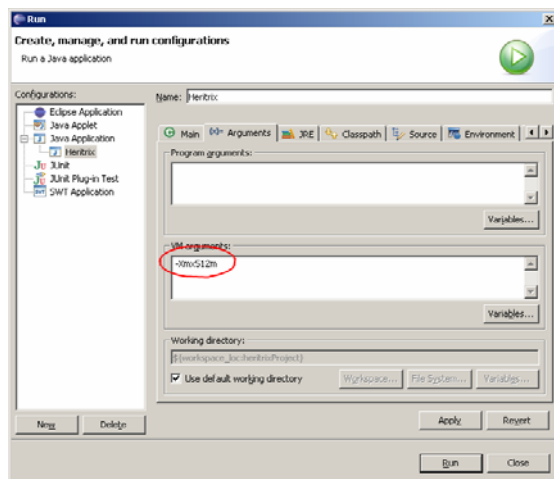


图10-44 在Eclipse中加入启动参数

(13) 按图 10-44 所示，输入 Java 虚拟机的参数，就可以增大 Heritrix 的最大可用内存。如图 10-45 是使用了 -Xmx512m 参数后的 Console 界面。

```
Memory
2106 KB used
2760 KB current heap
520256 KB max heap
```

图10-45 使用了512m的HeapSize

在运行的过程中，值得注意的一点是，进度条的百分比数量并不是准确的。因为这个百分比实际上是已经处理的链接数和总共分析出的链接数的比值。当页面在不断被抓取分析时，链接的数量也会不断的增加，因此，这个百分比的数字也在不断的变化。例如如图 10-46 所示，此时总共抓取到的链接数已经达到了 12280 个，处理了 799 个，它的百分比数量为 6%，这显然比图 10-42 或图 10-39 中的要小。

```
Totals
downloaded 799 6% 11431 queued
12280 total downloaded and queued
3 MB uncompressed data received
```

图10-46 抓取了799的链接

读者可能已经发现，在 Heritrix 中，大量的链接被称为 URI。从理论上说，URL 应该是一个完整的地址，而 URI 应该是去除协议、主机和端口后剩余的部分。Heritrix 中可能有一定程度的混淆，希望读者不要对此感到奇怪。

至此，已经把 Heritrix 成功的运行起来，并且抓取了一定的内容。接下来，看一下它是如何存储抓取下来的信息的。

10.1.7 Heritrix 的镜象存储结构

由于在前面设置了 Writer 的类型为 MirrorWriter。因此，磁盘上应该留有了所抓取到的网页的各种镜象。那么，究竟 Heritrix 是如何存储下镜象信息的呢？

打开 Eclipse 的 workspace 目录，进入 heritrixProject 的工程，里面有一个 jobs 目录。进入后，找到以刚才 job 的名称打头的文件夹，这里面的内容，就是 Heritrix 在运行时实时生成的。其中，有一个 mirror 目录，进入后，如图 10-47 所示。



图10-47 mirror目录下的内容

其实所谓镜象方式存储，就是将 URL 地址按“/”进行切分，进而按切分出来的层次存储，比如一个 URL 地址为：

`http://news.sohu.com/index.html`

那么它在 mirror 目录中的保存位置就应该是 new.sohu.com 目录下的 index.html 文件。为了验证这一说法的准确性，打开 new.sohu.com 目录，可以看到图 10-48。



图10-48 镜象示例

果然，index.html 文件就在这个目录下。另外，Heritrix 也同样将各种图片或脚本信息按路径进行了保存，例如，在 news.sohu.com 目录下有一个 images 目录，其中就保存了 URL 地址如 `http://news.sohu.com/images/xxx.gif` 这样的图片信息。如图 10-49 所示。

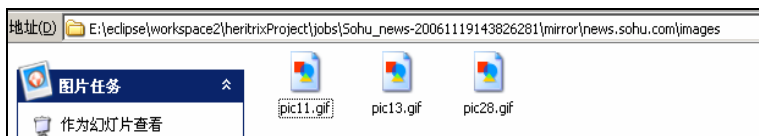


图10-49 抓取下来的图片文件

10.1.8 终止抓取或终止 Heritrix 的运行

当用户进行某个抓取任务时，有两种方法会让任务停止下来。

1. 正常终止

第一种方法当然就是任务的自然结束，其条件为所有队列中的 URI 已经被处理过了。此时，任务将自然终止。在“Jobs”面版上会看到任务已经完成，被加入到“Completed jobs”列表中。

2. 强行终止

当然，任务不可能总是运行完，这可能是对任务的控制不够，结果抓取了太多不相关的信息，进而造成 URL 队列无限制膨胀，无法终止。在这种情况下，就需要强行将任务终止。在 Console 面版上有如图 10-50 所示的一排链接，最后一个“Terminate”链接，就是用来终止当前运行的任务。



图10-50 终止任务的运行

单击“Terminate”链接后，当前在运行的抓取任务就会立即终止，并同样将任务放置到“Jobs”面版上的“Completed jobs”列表中，只不过在“status”上，它会显示“Finished - Ended by operator”这样的提示。

当然，如果用户希望关闭 Heritrix，并终止所有正在运行的任务，也可以单击 Console 面版上的“Shutdown Heritrix software”的链接，此时，Heritrix 会弹出一个警告，告诉你如果关闭 Heritrix，则当前一切正在运行的任务都将被终止。如图 10-51 所示。

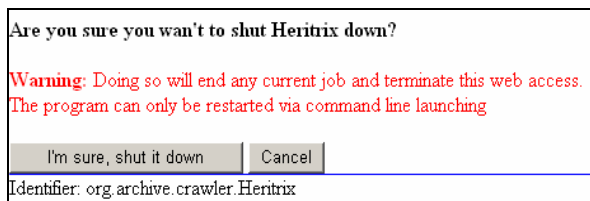


图10-51 关闭前的提示

如果选择“I’m sure, shut it down”，则 Heritrix 的 WebUI 将会终止，虚拟机进程结束。

10.2 Heritrix 的架构

在上一节中，详细介绍了 Heritrix 的使用入门。读者通过上一节的介绍，应该已经能够使用 Heritrix 来进行简单的网页抓取了。那么，Heritrix 的内容究竟是如何工作的呢？它的设计方面有什么突出之处？

本节就将介绍 Heritrix 的几个主要组件，以此让读者了解其主要架构和工作方式。为后续的扩展 Heritrix 做一些铺垫。

10.2.1 抓取任务 CrawlOrder

之所以选择从 CrawlOrder 这个类说起，是因为它是整个抓取工作的起点。在上一节中已经说过，一次抓取任务包括许多的属性，建立一个任务的方式有很多种，最简单的一种就是根据默认的 order.xml 来配置。在内存中，order 使用 CrawlOrder 这个类来进行表示。看一下 API 文档中 CrawlOrder 的继承关系图，如图 10-52 所示。

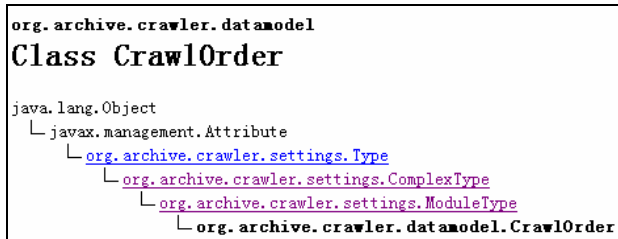


图10-52 CrawlOrder类的继承关系图

从继承关系图中可以看到，CrawlOrder 继承自一系列的与属性设置相关的基类。另外，它的最顶层基类是 javax.management.Attribute，这是一个 JMX 中的类，它可以动态的反映出 Java 容器内某个 MBean 的属性变化。关于这一部分的内容不是我们所要讨论的重点，只需知道，CrawlOrder 中的属性，是需要被随时读取和监测的。

那么究竟使用什么工具来读取 order.xml 文件中的各种属性呢。另外，一个 CrawlOrder 的对象又该如何构建呢？Heritrix 提供了很好的工具支持对于 order.xml 的读取。在 org.archive.crawler.settings 包下有一个 XMLSettingsHandler 类，它可以用来帮助读取 order.xml。

```
public XMLSettingsHandler(File orderFile) throws
InvalidAttributeValueException
```

在 XMLSettingsHandler 的构造函数中，其所传入的参数 orderFile 正是一个经过对象封装的 order.xml 的 File。这样，就可以直接调用其构造函数，来创建一个 XMLSettingsHandler 的实例，以此做为一个读取 order.xml 的工具。

当一个 XMLSettingsHandler 的实例被创建后，可以通过 getOrder()方法来获取 CrawlOrder 的实例，这样也就可以进行下一步的工作了。

10.2.2 中央控制器 CrawlController

中央控制器是一次抓取任务中的核心组件。它将决定整个抓取任务的开始和结束。CrawlController 位于 org.archive.crawler.framework 中，在它的 Field 声明中，看到如下代码片段。

代码 10.2

```
// key subcomponents which define and implement a crawl in progress
private transient CrawlOrder order;
private transient CrawlScope scope;
private transient ProcessorChainList processorChains;

private transient Frontier frontier;

private transient ToePool toePool;
```



```
private transient ServerCache serverCache;  
  
// This gets passed into the initialize method.  
private transient SettingsHandler settingsHandler;
```

可以看到，在 `CrawlController` 类中，定义了以下几个组件：

- **CrawlOrder**：这就不用说了，因为一个抓取工作必须要有一个 `Order` 对象，它保存了对该次抓取任务中，`order.xml` 的属性配置。
- **CrawlScope**：在 10.1.4 节中已经介绍过了，这是决定当前的抓取范围的一个组件。
- **ProcessorChainList**：从名称上很明显就能看出，它表示了处理器链，在这个列表中的每一项都可以和 10.1.4 节中所介绍的处理器链对应上。
- **Frontier**：很明显，一次抓取任务需要设定一个 `Frontier`，以此来不断为其每个线程提供 URI。
- **ToePool**：这是一个线程池，它管理了所有该抓取任务所创建的子线程。
- **ServerCache**：这是一个缓存，它保存了所有在当前任务中，抓取过的 `Host` 名称和 `Server` 名称。

以上组件应该是一次正常的抓取过程中所必需的几项，它们各自的任务很独立，分工明确，但在后台中，它们之间却有着千丝万缕的联系，彼此互相做为构造函数或初始化的参数传入。那么，究竟该如何获得 `CrawlController` 的实例，并且通过自主的编程来使用 `Heritrix` 提供的 API 进行一次抓任务呢？

事实上 `CrawlController` 有一个不带参数的构造函数，开发者可以直接通过它的构造函数来构造一个 `CrawlController` 的实例。但是值得注意的一点，在构造一个实例并进行抓取任务时，有几个步骤需要完成：

- （1）首先构造一个 `XMLSettingsHandler` 对象，将 `order.xml` 内的属性信息装入。
- （2）调用 `CrawlController` 的构造函数，构造一个 `CrawlController` 的实例。
- （3）调用 `CrawlController` 的 `initialize(SettingsHandler)` 方法，初始化 `CrawlController` 实例。

其中，传入的参数是在第一步是构造的 `XMLSettingsHandler` 实例。

（4）当上述 3 步完成后，`CrawlController` 就已经具备运行的条件，可以开始运行了。此时，只需调用它的 `requestCrawlStart()` 方法，就可以启运线程池和 `Frontier`，然后就可以开始不断的抓取网页了。

上述过程可以用图 10-53 所示。

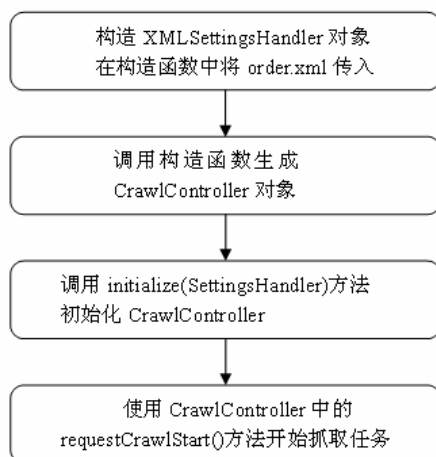


图10-53 使用 `CrawlController` 启运抓取任务



在 `CrawlController` 的 `initialize()` 方法中, `Heritrix` 主要做了以下几件事:

- (1) 从 `XMLSettingsHandler` 中取出 `Order`。
- (2) 检查了用户设定的 `UserAgent` 等信息, 看是否符合格式。
- (3) 设定了开始抓取后保存文件信息的目录结构。
- (4) 初始化了日志信息的记录工具。
- (5) 初始化了使用 `Berkley DB` 的一些工具。
- (6) 初始化了 `Scope`、`Frontier` 以及 `ProcessorChain`。
- (7) 最后实例化了线程池。

在正常情况下, 以上顺序不能够被随意变动, 因为后一项功能的初始化很有可能需要前几项功能初始化的结果。例如线程池的初始化, 必须要在先有了 `Frontier` 的实例的基础上来进行。读者可能对其中的 `Berkeley DB` 感到费解, 在后面的小节将详细说明。

从图 10-53 中看到, 最终启动抓取工作的是 `requestCrawlStart()` 方法。其代码如下。

代码 10.3

```
public void requestCrawlStart() {
    // 初始化处理器链
    runProcessorInitialTasks();

    // 设置一下抓取状态的改变, 以便能够激发一些Listeners
    // 来处理相应的事件
    sendCrawlStateChangeEvent(STARTED, CrawlJob.STATUS_PENDING);
    String jobState;
    state = RUNNING;
    jobState = CrawlJob.STATUS_RUNNING;
    sendCrawlStateChangeEvent(this.state, jobState);

    // A proper exit will change this value.
    this.sExit = CrawlJob.STATUS_FINISHED_ABNORMAL;

    // 开始日志线程
    Thread statLogger = new Thread(statistics);
    statLogger.setName("StatLogger");
    statLogger.start();

    // 启动Frontier, 抓取工作开始
    frontier.start();
}
```

可以看到, 启动抓取工作的核心就是要启动 `Frontier` (通过调用其 `start()` 方法), 以便能够开始向线程池中的工作线程提供 `URI`, 供它们抓取。

下面的代码就是 `BdbFrontier` 的父类 `AbstractFrontier` 中的 `start()` 方法和 `unpause()` 方法:

代码 10.4

```
public void start() {
    if (((Boolean)getUncheckedAttribute(null, ATTR_PAUSE_AT_START))
        .booleanValue()) {
        // 若配置文件中不允许该次抓取开始
        // 则停止
        controller.requestCrawlPause();
    } else {
        // 若允许开始, 则开始
        unpause();
    }
}
```




```
}

synchronized public void unpause() {
    // 去除当前阻塞变量
    shouldPause = false;
    // 唤醒所有阻塞线程，开始抓取任务
    notifyAll();
}
```

在 `start()` 方法中，首先判断配置中的属性是否允许当前线程开始。若不允许，则令 `controller` 停止抓取。若允许开始，则简单的调用 `unpause()` 方法。`unpause()` 方法更为简单，它首先将阻塞线程的信号量设为 `false`，即允许线程开始活动，然后通过 `notifyAll()` 方法，唤醒线程池中所有被阻塞的线程，开始抓取。

10.2.3 Frontier 链接制造工厂

Frontier 在英文中的意思是“前线，领域”，在 **Heritrix** 中，它表示一种为线程提供链接的工具。它通过一些特定的算法来决定哪个链接将接下来被送入处理器链中，同时，它本身也负责一定的日志和状态报告功能。

事实上，要写出一个合格并且真正能够使用的 **Frontier** 绝非一件简单的事情，尽管有了 **Frontier** 接口，其中的方法约束了 **Frontier** 的行为，也给编码带来了一定的指示。但是其中还存在着很多问题，需要很好的设计和处理才可以解决。

在 **Heritrix** 的官方文档上，有一个 **Frontier** 的例子，在此拿出来进行一下讲解，以此来向读者说明一个最简单的 **Frontier** 都能够做什么事。以下就是这个 **Frontier** 的代码。

代码 10.5

```
public class MyFrontier extends ModuleType implements Frontier,
    FetchStatusCodes {

    // 列表中保存了还未被抓取的链接
    List pendingURIs = new ArrayList();

    // 这个列表中保存了一系列的链接，它们的优先级
    // 要高于pendingURIs那个List中的任何一个链接
    // 表中的链接表示一些需要被满足的先决条件
    List prerequisites = new ArrayList();

    // 一个HashMap，用于存储那些已经抓取过的链接
    Map alreadyIncluded = new HashMap();

    // CrawlController对象
    CrawlController controller;

    // 用于标识是否一个链接正在被处理
    boolean uriInProcess = false;

    // 成功下载的数量
    long successCount = 0;
    // 失败的数量
    long failedCount = 0;
    // 抛弃掉链接的数量
    long disregardedCount = 0;
    // 总共下载的字节数
```



```
long totalProcessedBytes = 0;

// 构造函数
public MyFrontier(String name) {
    super(Frontier.ATTR_NAME, "A simple frontier.");
}

// 初始化, 参数为一个CrawlController
public void initialize(CrawlController controller)
    throws FatalConfigurationException, IOException {

    // 注入
    this.controller = controller;

    // 把种子文件中的链接加入到pendingURIs中去
    this.controller.getScope().refreshSeeds();
    List seeds = this.controller.getScope().getSeedlist();
    synchronized(seeds) {
        for (Iterator i = seeds.iterator(); i.hasNext();) {
            UURI u = (UURI) i.next();
            CandidateURI caUri = new CandidateURI(u);
            caUri.setSeed();
            schedule(caUri);
        }
    }
}

// 该方法为给线程池中的线程调用的, 用以取出下一个待处理的链接
public synchronized CrawlURI next(int timeout) throws
InterruptedException {
    if (!uriInProcess && !isEmpty()) {
        uriInProcess = true;
        CrawlURI curi;
        /*
         * 算法很简单, 总是先看prerequisites队列中是否有
         * 要处理的链接, 如果有, 就先处理, 如果没有
         * 再看pendingURIs队列中是否有链接
         * 每次在处理的时候, 总是取出队列中的第一个链接
         */
        if (!prerequisites.isEmpty()) {
            curi = CrawlURI.from((CandidateURI) prerequisites.remove(0));
        } else {
            curi = CrawlURI.from((CandidateURI) pendingURIs.remove(0));
        }

        curi.setServer(controller.getServerCache().getServerFor(curi));
        return curi;
    } else {
        wait(timeout);
        return null;
    }
}

public boolean isEmpty() {
    return pendingURIs.isEmpty() && prerequisites.isEmpty();
}

// 该方法用于将新链接加入到pendingURIs队列中, 等待处理
```

```
public synchronized void schedule(CandidateURI caURI) {
    /*
     * 首先判断要加入的链接是否已经被抓取过
     * 如果已经包含在alreadyIncluded这个HashMap中
     * 则说明处理过了，即可以放弃处理
     */
    if (!alreadyIncluded.containsKey(caURI.getURIStrng())) {
        if(caURI.needsImmediateScheduling()) {
            prerequisites.add(caURI);
        } else {
            pendingURIs.add(caURI);
        }
        // HashMap中使用url的字符串来做为key
        // 而将实际的CadidateURI对象做为value
        alreadyIncluded.put(caURI.getURIStrng(), caURI);
    }
}

public void batchSchedule(CandidateURI caURI) {
    schedule(caURI);
}

public void batchFlush() {
}

// 一次抓取结束后所执行的操作，该操作由线程池
// 中的线程来进行调用
public synchronized void finished(CrawlURI cURI) {
    uriInProgress = false;

    // 成功下载
    if (cURI.isSuccess()) {
        successCount++;
        // 统计下载总数
        totalProcessedBytes += cURI.getContentSize();
        // 如果成功，则触发一个成功事件
        // 比如将Extractor解析出来的新URL加入队列中
        controller.fireCrawledURISuccessfulEvent(cURI);
        cURI.stripToMinimal();
    }
    // 需要推迟下载
    else if (cURI.getFetchStatus() == S_DEFERRED) {
        cURI.processingCleanup();
        alreadyIncluded.remove(cURI.getURIStrng());
        schedule(cURI);
    }
    // 其他状态
    else if (cURI.getFetchStatus() == S_ROBOTS_PRECLUDED
        || cURI.getFetchStatus() == S_OUT_OF_SCOPE
        || cURI.getFetchStatus() == S_BLOCKED_BY_USER
        || cURI.getFetchStatus() == S_TOO_MANY_EMBED_HOPS
        || cURI.getFetchStatus() == S_TOO_MANY_LINK_HOPS
        || cURI.getFetchStatus() == S_DELETED_BY_USER) {
        // 抛弃当前URI
        controller.fireCrawledURIDisregardEvent(cURI);
        disregardedCount++;
        cURI.stripToMinimal();
    } else {
```



```
        controller.fireCrawledURIFailureEvent(cURI);
        failedCount++;
        cURI.stripToMinimal();
    }
    cURI.processingCleanup();
}

// 返回所有已经处理过的链接数量
public long discoveredUriCount() {
    return alreadyIncluded.size();
}

// 返回所有等待处理的链接的数量
public long queuedUriCount() {
    return pendingURIs.size() + prerequisites.size();
}

// 返回所有已经完成的链接数量
public long finishedUriCount() {
    return successCount + failedCount + disregardedCount;
}

// 返回所有成功处理的链接数量
public long successfullyFetchedCount() {
    return successCount;
}

// 返回所有失败的链接数量
public long failedFetchCount() {
    return failedCount;
}

// 返回所有抛弃的链接数量
public long disregardedFetchCount() {
    return disregardedCount;
}

// 返回总共下载的字节数
public long totalBytesWritten() {
    return totalProcessedBytes;
}

public String report() {
    return "This frontier does not return a report.";
}

public void importRecoverLog(String pathToLog) throws IOException {
    throw new UnsupportedOperationException();
}

public FrontierMarker getInitialMarker(String regexpr,
    boolean inCacheOnly) {
    return null;
}

public ArrayList getURIsList(FrontierMarker marker, int numberOfMatches,
    boolean verbose) throws InvalidFrontierMarkerException {
    return null;
}
```



```
}  
  
public long deleteURIs(String match) {  
    return 0;  
}  
  
}
```

在 Frontier 中，根据笔者给出的中文注释，相信读者已经能够了解这个 Frontier 中的大部分玄机。以下给出详细的解释。

首先，Frontier 是用来向线程提供链接的，因此，在上面的代码中，使用了两个 ArrayList 来保存链接。其中，第一个 pendingURIs 保存的是等待处理的链接，第二个 prerequisites 中保存的也是链接，只不过它里面的每个链接的优先级都要高于 pendingURIs 里的链接。通常，在 prerequisites 中保存的都是如 DNS 之类的链接，只有当这些链接被首先解析后，其后续的链接才能够被解析。

除了这两个 ArrayList 外，在上面的 Frontier 还有一个名称为 alreadyIncluded 的 HashMap。它用于记录那些已经被处理过的链接。每当调用 Frontier 的 schedule() 方法来加入一个新的链接时，Frontier 总要先检查这个正要加入到队列中的链接是不是已经被处理过了。

很显然，在分析网页的时候，会出现大量相同的链接，如果没有这种检查，很有可能造成抓取任务永远无法完成的情况。同时，在 schedule() 方法中还加入了一些逻辑，用于判断当前要进入队列的链接是否属于需要优先处理的，如果是，则置入 prerequisites 队列中，否则，就简单的加入 pendingURIs 中即可。

注意：Frontier 中还有两个关键的方法，next() 和 finished()，这两个方法都是要交由抓取的线程来完成的。Next() 方法的主要功能是：从等待队列中取出一个链接并返回，然后抓取线程会在它自己的 run() 方法中完成对这个链接的处理。而 finished() 方法则是在线程完成对链接的抓取和后续的一切动作后（如将链接传递经过处理器链）要执行的。它把整个处理过程中解析出的新的链接加入队列中，并且在处理完当前链接后，将之加入 alreadyIncluded 这个 HashMap 中去。

需要读者记住的是，这仅仅是一个最基础的代码，它有很多的功能缺失和性能问题，甚至可能出现重大的同步问题。不过尽管如此，它应当也起到了抛砖引玉的作用，能够从结构上揭示了一个 Frontier 的作用。

10.2.4 用 Berkeley DB 实现的 BdbFrontier

简单的说，Berkeley DB 就是一个 HashTable，它能够按“key/value”方式来保存数据。它是由美国 Sleepycat 公司开发的一套开放源代码的嵌入式数据库，它为应用程序提供可伸缩的、高性能的、有事务保护功能的数据管理服务。

那么，为什么不使用一个传统的关系型数据库呢？这是因为当使用 BerkeleyDB 时，数据库和应用程序在相同的地址空间中运行，所以数据库操作不需要进程间的通讯。然而，当使用传统关系型数据库时，就需要在一台机器的不同进程间或在网络中不同机器间进行进程通讯，这样所花费的开销，要远远大于函数调用的开销。

另外，Berkeley DB 中的所有操作都使用一组 API 接口。因此，不需要对某种查询语言（比如 SQL）进行解析，也不用生成执行计划，这就大大提高了运行效率。

当然，做为一个数据库，最重要的功能就是事务的支持，Berkeley DB 中的事务子系统就是用来为其提供事务支持的。它允许把一组对数据库的修改看作一个原子单位，这组操作要么全做，要么全不做。在默认的情况下，系统将提供严格的 ACID 事务属性，但是应用程序可

以选择不使用系统所作的隔离保证。该子系统使用两段锁技术和先写日志策略来保证数据的正确性和一致性。这种事务的支持就要比简单的 HashTable 中的 Synchronize 要更加强大。

注意：在 Heritrix 中，使用的是 Berkeley DB 的 Java 版本，这种版本专门为 Java 语言做了优化，提供了 Java 的 API 接口以供开发者使用。

为什么 Heritrix 中要用到 Berkeley DB 呢？这就需要再回过头来看一下 Frontier 了。

在上一小节中，当一个链接被处理后，也即经过处理器链后，会生成很多新的链接，这些新的链接需要被 Frontier 的一个 schedule 方法加入到队列中继续处理。但是，在将这些新链接加入到队列之前，要首先做一个检查，即在 alreadyIncluded 这个 HashMap 中，查看当前要加入到队列中的链接是否在先前已经被处理过了。

当使用 HashMap 来存储那些已经被处理过的链接时，HashMap 中的 key 为 url，而 value 则为一个对 url 封装后的对象。很显然的，这里有几个问题。

- 对这个 HashMap 的读取是多线程的，因为每个线程都需要访问这个 HashMap，以决定当前要加入链接是否已经存在过了。
- 对这个 HashMap 的写入是多线程的，每个线程在处理完毕后，都会访问这个 HashMap，以写入最新处理的链接。
- 这个 HashMap 的容量可能很大，可以试想，一次在广域网范围上的网页抓取，可能会涉及到上十亿个 URL 地址，这种地址包括网页、图片、文件、多媒体对象等，所以，不可能将这么大一张表完全的置放于内存中。

综合考虑以上 3 点，仅用一个 HashMap 来保存所有的链接，显然已经不能满足“大数据量，多并发”这样的要求。因此，需要寻找一个替代的工具来解决问题。Heritrix 中的 BdbFrontier 就采用了 Berkeley DB，来解决这种 URL 存放的问题。事实上，BdbFrontier 就是 Berkeley DB Frontier 的简称。

为了在 BdbFrontier 中使用 Berkeley DB，Heritrix 本身构造了一系列的类来帮助实现这个功能。这些类如下：

- BdbFrontier
- BdbMultipleWorkQueues
- BdbWorkQueue
- BdbUriUniqFilter

上述的 4 个类，都以 Bdb3 个字母开头，这表明它们都是使用到了 Berkeley DB 的功能。其中：

(1) BdbMultipleWorkQueues 代表了一组链接队列，这些队列有各自不同的 key。这样，由 Key 和链接队列可以形成一个“Key/Value”对，也就成为了 Berkeley DB 里的一条记录 (DatabaseEntry) 如图 10-54 所示。

BdbMultipleWorkQueues	
Key1	Queue1 {URI1, URI2, URI3, ...}
Key2	Queue2 {URIx, ...}
Key3	Queue3 {...}
Key4	Queue4 {...}
.....

图10-54 BdbMultipleWorkQueues示意



图 10-54 清楚的显示了 Berkeley DB 中的“key/value”形式。可以说，这就是一张 Berkeley DB 的数据库表。其中，数据库的一条记录包含两个部分，左边是一个由右边的所有 URL 链接计算出来的公共键值，右边则是一个 URL 的队列。

(2) BdbWorkQueue 代表了一个基于 Berkeley DB 的队列，与 BdbMultipleWorkQueues 所不同的是，该队列中的所有的链接都具有相同的键值。事实上，BdbWorkQueue 只是对 BdbMultipleWorkQueues 的封装，在构造一个 BdbWorkQueue 时，需传入一个键值，以此做为该 Queue 在数据库中的标识。事实上，在工作线程从 Frontier 中取出链接时，Heritrix 总是先取出整个 BdbWorkQueue，再从中取出第一个链接，然后将当前这个 BdbWorkQueue 置入一个线程安全的同步容器内，等待线程处理完毕后才将该 Queue 释放，以便该 Queue 内的其他 URI 可以继续被处理。

(3) BdbUriUniqFilter 是一个过滤器，从名称上就能知道，它是专门用来过滤当前要进入等待队列的链接对象是否已经被抓取过。很显然，在 BdbUriUniqFilter 内部嵌入了一个 Berkeley DB 数据库用于存储所有的被抓取过的链接。它对外提供了

```
public void add(String key, CandidateURI value)
```

这样的接口，以供 Frontier 调用。当然，若是参数的 CandidateURI 已经存在于数据库中了，则该方法会禁止它加入到等待队列中去。

(4) BdbFrontier 就是 Heritrix 中使用了 Berkeley DB 的链接制造工厂。它主要使用 BdbUriUniqFilter，做为其判断当前要进入等待队列的链接对象是否已经被抓取过。同时，它还使用了 BdbMultipleWorkQueues 来做为所有等待处理的 URI 的容器。这些 URI 根据各自的内容会生成一个 Hash 值成为它们所在队列的键值。

在 Heritrix1.10 的版本中，可以说 BdbFrontier 是惟一个具有实用意义的链接制造工厂了。虽然 Heritrix 还提供了另外两个 Frontier：

```
org.archive.crawler.frontier.DomainSensitiveFrontier  
org.archive.crawler.frontier.AdaptiveRevisitFrontier
```

但是，DomainSensitiveFrontier 已经被废弃不再推荐使用。而 AdaptiveRevisitFrontier 的算法是不管遇到什么新链接，都义无反顾的再次抓取，这显然是一种很落后的算法。因此，了解 BdbFrontier 的实现原理，对于更好的了解 Heritrix 对链接的处理有实际意义。

BdbFrontier 的代码相对比较复杂，笔者在这里也只能简单将其轮廓进行介绍，读者仍须将代码仔细研读，方能把文中的点点知识串联起来，进而更好的理解 Heritrix 作者们的巧妙匠心。

10.2.5 Heritrix 的多线程 ToeThread 和 ToePool

想要更有效更快速的抓取网页内容，则必须采用多线程。Heritrix 中提供了一个标准的线程池 ToePool，它用于管理所有的抓取线程。

ToePool 和 ToeThread 都位于 org.archive.crawler.framework 包中。前面已经说过，ToePool 的初始化，是在 CrawlController 的 initialize()方法中完成的。来看一下 ToePool 以及 ToeThread 是如何被初始化的。以下代码是在 CrawlController 中用于对 ToePool 进行初始化的。

```
// 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。但是，那些真正的工作线程又是如何建立的呢？

下面来看一下线程池中的 `setSize(int)` 方法。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，它并不是那么简单。

代码 10.6

```
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();
        }
    }
}

// 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，
    // 实际上是将所有该ThreadGroup中开辟的线程放入
    // 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);
    newThread.start();
}
```

通过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，才有可能创建新线程；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所设定的值的大小，来决定池中所管理线程数量的增减。

当线程被启动后，所执行的是其 `run()`方法中的片段。接下来，看一个 `ToeThread` 到底是如何处理从 `Frontier` 中获得的链接的。

代码 10.7

```
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
            // 到等待队列中去
            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;
}
```

在上面的方法中，很清楚的显示了工作线程是如何从 **Frontier** 中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種状态。很显然，以上代码中，最重要的一行语句是 **processCrawlUri()**，它是真正调用处理链来对链接进行处理的代码。其中的内容，放在下一个小节中介绍。

10.2.6 处理链和 Processor

在本章第一节中介绍了设置处理器链相关的内容。从中知道，处理器链包括以下几种：

- PreProcessor
- Fetcher
- Extractor
- Writer
- PostProcessor

为了更好的表示整个处理器链的逻辑结构，以及它们之间的链式调用关系，**Heritrix** 设计了几个 API 来表示这种逻辑结构。

```
org.archive.crawler.framework.Processor
org.archive.crawler.framework.ProcessorChain
org.archive.crawler.framework.ProcessorChainList
```

下面进行详细讲解。



1. Processor类

该类代表着单个的处理器，所有的处理器都是它的子类。在 `Processor` 类中有一个 `process()` 方法，它被标识为 `final` 类型的，也就是说，它不可以被它的子类所覆盖。代码如下。

代码 10.8

```
public final void process(CrawlURI curi) throws InterruptedException
{
    // 设置下一个处理器
    curi.setNextProcessor(getDefaultNextProcessor(curi));

    try
    {
        // 判断当前这个处理器是否为enabled
        if (!((Boolean) getAttribute(ATTR_ENABLED, curi)).booleanValue()) {
            return;
        }
    } catch (AttributeNotFoundException e) {
        logger.severe(e.getMessage());
    }

    // 如果当前的链接能够通过过滤器
    // 则调用innerProcess(curis)方法
    // 来进行处理
    if(filtersAccept(curis)) {
        innerProcess(curis);
    }
    // 如果不能通过过滤器检查，则调
    // 用innerRejectProcess(curis)来处理
    else
    {
        innerRejectProcess(curis);
    }
}
```

方法的含义很简单。即首先检查是否允许这个处理器处理该链接，如果允许，则检查当前处理器所自带的过滤器是否能够接受这个链接。当过滤器的检查也通过后，则调用 `innerProcess(curis)` 方法来处理，如果过滤器的检查没有通过，就使用 `innerRejectProcess(curis)` 方法处理。

其中 `innerProcess(curis)` 和 `innerRejectProcess(curis)` 方法都是 `protected` 类型的，且本身没有实现任何内容。很明显它们是留在子类中，实现具体的处理逻辑。不过大部分的子类都不会重写 `innerRejectProcess(curis)` 方法了，这是因为反正一个链接已经被当前处理器拒绝处理了，就不用再有什么逻辑了，直接跳到下一个处理器继续处理就行了。

2. ProcessorChain类

该类表示一个队列，里面包括了同种类型的几个 `Processor`。例如，可以将一组的 `Extractor` 加入到同一个 `ProcessorChain` 中去。

在一个 `ProcessorChain` 中，有 3 个 `private` 类型的类变量：

```
private final MapType processorMap;
private ProcessorChain nextChain;
private Processor firstProcessor;
```

其中，`processorMap` 中存放的是当前这个 `ProcessorChain` 中所有的 `Processor`。`nextChain` 的类型是 `ProcessorChain`，它表示指向下一个处理器链的指针。而 `firstProcessor` 则是指向当前



队列中的第一个处理器的指针。

3. ProcessorChainList

从名称上看,它保存了 Heritrix 一次抓取任务中所设定的所有处理器链,将之做为一个列表。正常情况下,一个 ProcessorChainList 中,应该包括有 5 个 ProcessorChain,分别为 PreProcessor 链、Fetcher 链、Extractor 链、Writer 链和 PostProcessor 链,而每个链中又包含有多个的 Processor。这样,就将整个处理器结构合理的表示了出来。

那么,在 ToeThread 的 processCrawlUri()方法中,又是如何来将一个链接循环经过这样一组结构的呢?请看下面的代码:

代码 10.9

```
private void processCrawlUri() throws InterruptedException {
    // 设定当前线程的编号
    currentCuri.setThreadNumber(this.serialNumber);
    // 为当前处理的URI设定下一个ProcessorChain

currentCuri.setNextProcessorChain(controller.getFirstProcessorChain());

    // 设定开始时间
    lastStartTime = System.currentTimeMillis();
    try {

        // 如果还有一个处理链没处理完
        while (currentCuri.nextProcessorChain() != null)
        {
            setStep(STEP_ABOUT_TO_BEGIN_CHAIN);

            // 将下个处理链中的第一个处理器设定为
            // 下一个处理当前链接的处理器
            currentCuri.setNextProcessor(currentCuri
                                           .nextProcessorChain().getFirstProcessor()
);

            // 将再下一个处理器链设定为当前链接的
            // 下一个处理器链,因为此时已经相当于
            // 把下一个处理器链置为当前处理器链了
            currentCuri.setNextProcessorChain(currentCuri
                                           .nextProcessorChain().getNextProcessorCha
in());

            // 开始循环处理当前处理器链中的每一个Processor
            while (currentCuri.nextProcessor() != null)
            {
                setStep(STEP_ABOUT_TO_BEGIN_PROCESSOR);
                Processor currentProcessor =
getProcessor(currentCuri.nextProcessor());
                currentProcessorName = currentProcessor.getName();
                continueCheck();
                // 调用Process方法
                currentProcessor.process(currentCuri);
            }
        }
        setStep(STEP_DONE_WITH_PROCESSORS);
        currentProcessorName = "";
    }
}
```



```
catch (RuntimeExceptionWrapper e) {
    // 如果是Berkeley DB的异常
    if(e.getCause() == null) {
        e.initCause(e.getDetail());
    }
    recoverableProblem(e);
} catch (AssertionError ae) {
    recoverableProblem(ae);
} catch (RuntimeException e) {
    recoverableProblem(e);
} catch (StackOverflowError err) {
    recoverableProblem(err);
} catch (Error err) {
    seriousError(err);
}
}
```

代码使用了双重循环来遍历整个处理器链的结构，第一重循环首先遍历所有的处理器链，第二重循环则在链内部遍历每个 `Processor`，然后调用它的 `process()` 方法来执行处理逻辑。

10.3 扩展和定制 Heritrix

在前面两节中，向读者介绍了 Heritrix 的启动、创建任务、抓取网页、组件结构。但是，读者应该也可以明显的看出，如果不用 Heritrix 抓取和分析网页的行为进行一定的控制，它是无法达到要求的。

对 Heritrix 的行为进行控制，是要建立在对其架构充分了解的基础之上的，因此，本节的内容完全是基于上一节中所讨论的基础。

10.3.1 向 Heritrix 中添加自己的 Extractor

很明显，Heritrix 内嵌的 Extractor 并不能够很好的完成所需要的工作，这不是说它不够强大，而是因为在解析一个网页时，常常有特定的需要。比如，可能只想抓取某种格式的链接，或是抓取某一特定格式中的文本片断。Heritrix 所提供的大众化的 Extractor 只能够将所有信息全部抓取下来。在这种情况下，就无法控制 Heritrix 到底该抓哪些内容，不该抓哪些内容，进而造成镜像信息太复杂，不好建立索引。

下面就使用一个实例，来讲解该如何定制和使用 Extractor。这个实例其实很简单，主要功能就是抓取所有在 Sohu 的新闻主页上出现的新闻，并且 URL 格式如下所示。

```
http://news.sohu.com/20061122/n246553333.shtml
```

(1) 分析一下这个 URL 可以知道，其中的主机部分是 `http://news.sohu.com`，这是搜狐新闻的域名，“20061122”应该表示的是新闻的日期，而最后的“n246553333.shtml”应该是一个新闻的编号，该编号全部以“n”打头。

(2) 有了这样的分析，就可以根据 URL 的特点，来定出一个正则表达式，凡是当链接符合该正则表达式，就认为它是一个潜在的值得抓取的链接，将其收藏，以待抓取。正则表达式如下：

```
http://news.sohu.com/[\\d]+/n[\\d]+.shtml
```

(3) 事实上所有的 Extractor 均继承自 `org.archive.crawler.extractor.Extractor` 这个抽象基类，在它的内部实现了 `innerProcess` 方法，以下便是 `innerProcess` 的实现：



代码 10.10

```
public void innerProcess(CrawlURI curi) {
    try {
        /*
         * 处理链接
         */
        extract(curi);
    } catch (NullPointerException npe) {
        curi.addAnnotation("err=" + npe.getClass().getName());
        curi.addLocalizedError(getName(), npe, "");
        logger.log(Level.WARNING, getName() + ": NullPointerException",
npe);
    } catch (StackOverflowError soe) {
        curi.addAnnotation("err=" + soe.getClass().getName());
        curi.addLocalizedError(getName(), soe, "");
        logger.log(Level.WARNING, getName() + ": StackOverflowError", soe);
    } catch (java.nio.charset.CoderMalfunctionError cme) {
        curi.addAnnotation("err=" + cme.getClass().getName());
        curi.addLocalizedError(getName(), cme, "");
        logger.log(Level.WARNING, getName() + ": CoderMalfunctionError",
cme);
    }
}
```

这个方法中，大部分代码都用于处理在解析过程中发生的各种异常和日志写入，不过，它为所有的 `Extractor` 定义了新的一个接口 `extract(CrawlURI)`，也就是说，所有的 `Extractor` 继承自它后，只需实现 `extract` 方法就可以了。以下是扩展 `Extractor` 时要做的几件事：

- (1) 写一个类，继承 `Extractor` 的基类。
- (2) 在构造函数中，调用父类的构造函数，以形成完整的家族对象。
- (3) 继承 `extract(cur)` 方法。

为了实现抓取 `news.sohu.com` 首页上所有新闻的链接，所开发的 `Extractor` 的完整源代码如下所示。

代码 10.11

```
package my;

import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.apache.commons.httpclient.URIException;
import org.archive.crawler.datamodel.CrawlURI;
import org.archive.crawler.extractor.Extractor;
import org.archive.crawler.extractor.Link;
import org.archive.io.ReplayCharSequence;
import org.archive.util.HttpRecorder;

public class SohuNewsExtractor extends Extractor {
    private static Logger logger = Logger.getLogger(SohuNewsExtractor.class
.getName());

    // 构造函数
    public SohuNewsExtractor(String name) {
```




```
this(name, "Sohu News Extractor");
}
// 构造函数
public SohuNewsExtractor(String name, String description) {
    super(name, description);
}

// 第一个正则式, 用于匹配SOHU新闻的格式
public static final String PATTERN_SOHU_NEWS =
"http://news.sohu.com/[\\d]+/n[\\d]+.shtml";

// 第二个正则式, 用于匹配所有的<a href="xxx">
public static final String PATTERN_A_HREF =
"<a\\s+href\\s*=\\s*(\\\"([^\\\"]*)\\\"|\\\"([^\s>])\\\")\\s*>";

// 继承的方法
protected void extract(CrawlURI curi) {

    // 将链接对象转为字符串
    String url = curi.toString();

    /*
     * 下面一段代码主要用于取得当前链接的返回 字符串, 以便对内容进行分析时使用
     */
    ReplayCharSequence cs = null;
    try {
        HttpRecorder hr = curi.getHttpRecorder();
        if (hr == null) {
            throw new IOException("Why is recorder null here?");
        }
        cs = hr.getReplayCharSequence();
    } catch (IOException e) {
        curi.addLocalizedError(this.getName(), e,
            "Failed get of replay char sequence " + curi.toString()
            + " " + e.getMessage());
        logger.log(Level.SEVERE, "Failed get of replay char sequence in
"
            + Thread.currentThread().getName(), e);
    }

    // 如果什么也没抓取到, 就返回
    if (cs == null) {
        return;
    }

    // 将链接返回的内容转成字符串
    String content = cs.toString();
    try {

        // 将字符串内容进行正则匹配
        // 取出其中的链接信息
        Pattern pattern = Pattern.compile(PATTERN_A_HREF,
            Pattern.CASE_INSENSITIVE);
        Matcher matcher = pattern.matcher(content);
```

```
// 若找到了一个链接
while (matcher.find()) {
    String newUrl = matcher.group(2);
    // 查看其是否为SOHU新闻的格式
    if (newUrl.matches(PATTERN_SOHU_NEWS)) {
        // 若是, 则将链接加入到队列中
        // 以备后续处理
        addLinkFromString(curi, newUrl, "", Link.NAVLINK_HOP);
    }
}

} catch (Exception e) {
    e.printStackTrace();
}
}

// 将链接保存记录下来, 以备后续处理
private void addLinkFromString(CrawlURI curi, String uri,
    CharSequence context, char hopType) {
    try {
        curi.createAndAddLinkRelativeToBase(uri, context.toString(),
            hopType);
    } catch (URISyntaxException e) {
        if (getController() != null) {
            getController().logUriError(e, curi.getUURI(), uri);
        } else {
            logger.info("Failed createAndAddLinkRelativeToBase "
                + curi + ", " + uri + ", " + context + ", "
                + hopType + ": " + e);
        }
    }
}
}
```

在上面代码的 `extract()` 方法中:

(1) 首先是将 `Fetcher` 所获得的链接的 `HTML` 响应取得, 并转成字符串, 这样, 才有可能在后面对页面中的链接做处理。

(2) 从页面内容中, 使用正则式取出所有链接的内容。判断链接是否符合 `Sohu` 的新闻格式, 倘若符合, 则调用 `addLinkFromString()` 方法, 来将这个链接加入到某个队列缓存中, 以备后续的处理。

在 `Extractor` 类开发完毕后, 如果使用 `WebUI` 的方式启动 `Heritrix`, 并让它出现在下拉选项中, 则需要修改 `Eclipse` 工程中的 `modules` 目录下的 `Processor.options` 文件, 如图 10-55 所示。

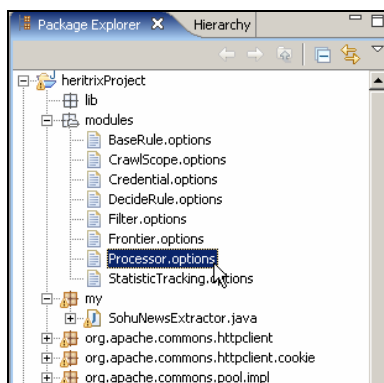


图10-55 修改Processor.options文件

打开 Processor.options 文件可以看到，所有在 WebUI 中设置处理器链时，页面上的下拉列表中的数据都保存在了其中，为了加入我们开发的 SohuNewsExtractor，只需在其中合适的位置上加入一行，内容如下所示：

```
my.SohuNewsExtractor | SohuNewsExtractor
```

接下来，再次启动 Heritrix，创建一个任务，进入处理器链设置的页面，就可以看到自己开发的 Extractor 了，如图 10-56 所示。

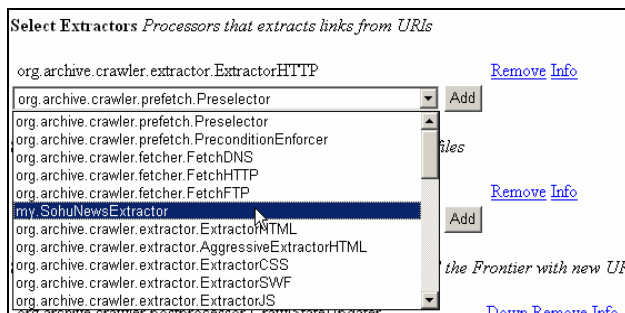


图10-56 新加入的Extractor已经在下拉菜单中显示出来

选择后，单击“Add”按钮，就可以将其加入到队列中，如图 10-57 所示。

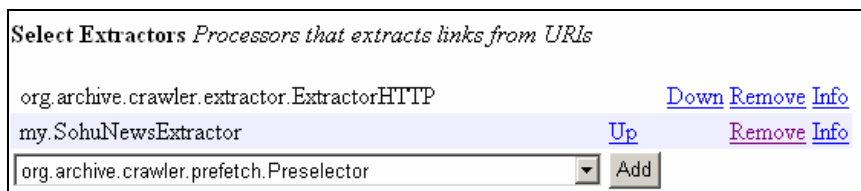


图10-57 已经加入到处理器队列中

需要注意的是，一定要将其置于 ExtractorHTTP 的后面，以保证 Heritrix 能够先行处理 HTTP 协议中的相关内容。与加入自己定制的 Extractor 的过程类似，开发者们也可以定制其他几种处理器。同样，只需要在 modules 目录下找到相应的.options 文件，然后将类全名加入即可。

10.3.2 定制 Queue-assignment-policy 两个问题

首先提出两个问题：

- 什么是Queue-assignment-policy
- 为什么要改变Queue-assignment-policy

在 10.2 节中，向读者介绍过了 Heritrix 的架构。其中，讲解了 Heritrix 使用了 Berkeley DB 来构建链接队列。这些队列被置放于 BdbMultipleWorkQueues 中时，总是先给予一个 Key，然后将那些 Key 值相同的链接放在一起，成为一个队列，也就是一个 Queue。

这里就出现了一个问题，这个 Key 值到底该如何计算呢？事实上，这里也说的 Key 值，应该是做为一种标识符的形式存在。也就是说，它要与 URL 之间有一种内在的联系。

在 Heritrix 中，为每个队列赋上 Key 值的策略，也就是它的 queue-assignment-policy。这就解答了第一个问题。

在默认的情况下，Heritrix 使用 HostnameQueueAssignmentPolicy 来解决 Key 值生成的问题。



仔细看一下这个策略的名称就知道，这种策略其实是以链接的 Host 名称为 Key 值来解决这个问题的。换句话也就是说，相同 Host 名称的所有 URL 都会被置放于同一个队列中间。这种方式在很大程度上可以解决广域网中信息抓取时队列的键值问题。但是，它对于某个单独网站的网页抓取，就出现了很大的问题。以 Sohu 的新闻网页为例，其中大部分的 URL 都来自于 sohu 网站的内部，因此，如果使用了 HostnameQueueAssignmentPolicy，则会造成有一个队列的长度非常长的情况。

在 Heritrix 中，一个线程从一个队列中取 URL 链接时，总是会先从队列的头部取出第一个链接，在这之后，这个被取出链接的队列会进入阻塞状态，直到待该链接处理完，它才会从阻塞状态中恢复。

假如使用 HostnameQueueAssignmentPolicy 策略来应对抓取一个网站中内容的情况，很有可能造成仅有一个线程在工作，而其他所有线程都在等待。这是因为那个装有绝大多数 URL 链接的队列几乎会永远处于阻塞状态，因此，别的线程根本获取不到其中的 URI，在这种情况下，抓取工作会进入一种类似于休眠的状态。因此，需要改变 queue-assignment-policy 来避免发生这种情况，这也就回答了第二个问题。

10.3.3 定制 Queue-assignment-policy 继承 QueueAssignmentPolicy 类

那么，被改变的 Key 值的生成方式，应该具有什么样的要求呢？从上面的叙述中可以知道，这个 Key 值最重要的一点就是应该能够有效的将所有的 URL 散列到不同的队列中，最终能使所有的队列的长度的方差较小，在这种情况下，才能保证工作线程的最大效率。

任何扩展 queue-assignment-policy 的默认实现的类，均继承自 QueueAssignmentPolicy 并覆写了其 getClassKey()方法，getClassKey 方法的参数为一个链接对象，而我们的散列算法，正是要根据这个链接对象来返回一个值。

具体的算法就不说了，有许多种方法可以实现的。比如使用字符串的长度等，在百度上搜索 URL 散列算法，最为出名的就要算是 ELFHash 法了。关于它的实现，有兴趣的读者可以自行研究。

10.3.4 扩展 FrontierScheduler 来抓取特定的内容

FrontierScheduler 是一个 PostProcessor，它的作用是将 Extractor 中所分析得出的链接加入到 Frontier 中，以待继续处理。先来看一下 FrontierScheduler 的 innerProcess()方法，代码如下。

代码 10.12

```
protected void innerProcess(final CrawlURI curi) {
    if (LOGGER.isLoggable(Level.FINEST)) {
        LOGGER.finest(getName() + " processing " + curi);
    }

    // 如果当前链接的处理结果中，有一些高优
    // 先级的链接要被处理
    if (curi.hasPrerequisiteUri() && curi.getFetchStatus() == S_DEFERRED) {
        handlePrerequisites(curi);
    }
}
```

```
        return;
    }

    // 对当前这个Processor进行同步
    synchronized(this) {
        // 从处理结果中，取出所有链接进行循环
        for (final Iterator iter = curi.getOutLinks().iterator();
             iter.hasNext();) {
            Object obj = iter.next();
            CandidateURI cauri = null;
            // 转型为CandidateURI
            if (obj instanceof CandidateURI) {
                cauri = (CandidateURI)obj;
            } else {
                LOGGER.severe("Unexpected type: " + obj);
            }

            // 调用schedule()方法
            if (cauri != null) {
                schedule(cauri);
            }
        }
    }
}

protected void schedule(CandidateURI caUri) {
    // 调用Frontier中的schedule()方法
    // 将传入的链接加入到等待队列中
    getController().getFrontier().schedule(caUri);
}
```

上面的代码中，首先检查当前链接处理后的结果集中是否有一些属于高优先级的链接，如果是，则立刻转走进行处理。如果没有，则对所有的结果集进行遍历，然后调用 `Frontier` 中的 `schedule` 方法加入队列进行处理。

在这里，`innerProcess()`中并未立刻使用 `Frontier` 中的 `schedule()`方法，而是增加了一层封装，先调用了一个类内部的 `protected` 类型的 `schedule()`方法，进而在这个方法中再调用 `Frontier` 的 `schedule` 方法。这种方式对 `FrontierScheduler` 进行扩展留出了很好的接口。

例如，当有某个任务在抓取时，可能希望人为的去掉符合某种条件的 `URL` 链接，使得其内容不会保存到本地。比如，要去除所有的扩展名为 `.zip`、`.exe`、`.rar`、`.pdf` 和 `.doc` 的链接（其实也就是不想下载这类文件）。可以通过继承 `FrontierScheduler`，并重写内部的 `schedule` 方法来达到我们的需要。以下是一个示例。

```
protected void schedule(CandidateURI caUri) {

    String url = caUri.toString();

    if (url.endsWith(".zip")
        || url.endsWith(".rar")
        || url.endsWith(".exe")
        || url.endsWith(".pdf")
        || url.endsWith(".doc")
        || url.endsWith(".xls")) {
        return;
    }

    getController().getFrontier().schedule(caUri);
}
```



```
}
```

这样，每当 Heritrix 在执行任务时，遇到这样的文件，就会跳过抓取，从而达到了对 URL 链接进行筛选的目的。

10.3.5 在 Prefetcher 中取消 robots.txt 的限制

Robots.txt 是一种专门用于搜索引擎网络爬虫的文件，当构造一个网站时，如果作者希望该网站的内容被搜索引擎收录，就可以在网站中创建一个纯文本文件 robots.txt，在这个文件中，声明该网站不想被 robot 访问的部分。这样，该网站的部分或全部内容就可以不被搜索引擎收录了，或者指定搜索引擎只收录指定的内容。

Heritrix 在其说明文档中，表明它是一个完全遵守 robots.txt 协议的网络爬虫。这一点固然在宣传上起到了一定的作用。但是，在实际的网页采集过程中，这并不是最好的作法。因为大部分的网站并不会放置一个 robots.txt 文件以供搜索引擎读取，在互联网信息以几何级数增长的今天，网站总是在希望自己的内容不被人所利用的同时，又希望自己能够被更多的用户从搜索引擎上检索到。

不过幸好，robots.txt 协议本身只是一种附加的协议，网站本身并不能了解究竟哪些 Socket 联接属于爬虫哪些属于正常的浏览器连接。所以，不遵守 robots.txt 协议成为了更多搜索引擎的首选。

使用过 Heritrix 的朋友就会发现这样一个问题，如果当一个网站没有放置 robots.txt 文件时，Heritrix 总是要花上大量的时间试图去访问这样一个文件，甚至可能 retry 很多次。这无疑很大的降低了抓取效率。因此，为了提高抓取的效率，可以试着将对 robots.txt 的访问部分去除。

在 Heritrix 中，对 robots.txt 文件的处理是处于 PreconditionEnforcer 这个 Processor 中的。PreconditionEnforcer 是一个 Prefetcher，当处理时，总是需要考虑一下当前这个链接是否有什么先决条件要先被满足的，而对 robots.txt 的访问则正好是其中之一。在 PreconditionEnforcer 中，有一个 private 类型的方法，它的方法签名为：

```
private boolean considerRobotsPreconditions(CrawlURI curi)
```

该方法的含义为：在进行对参数所表示的链接的抓取前，看一下是否存在一个由 robots.txt 所决定的先决条件。很显然，如果对每个链接都有这样的处理。那么，很有可能导致整个抓取任务的失败。因此，需要对它进行调整。

这个方法返回 true 时的含义为需要考虑 robots.txt 文件，返回 false 时则表示不需要考虑 robots.txt 文件，可以继续将链接传递给后面的处理器。所以，最简单的修改办法就是将这个方法整个注释掉，只留下一个 false 的返回值。经过笔者的试验，这种方法完全可行，抓取的速度提高了至少一半以上！

10.4 小结

本章对一款使用纯 Java 语言开发的、功能强大的网络爬虫进行了介绍。从它的使用入门至系统结构，以至最后的扩展和定制，旨在使读者用最快的速度了解一款优秀的开源爬虫。然而，由于篇幅所限，本章中的内容只能算是一个简单的入门。Heritrix 本身的功能极其强大，且扩展性良好。但它的缺点是配置较为复杂，且源码不好理解。希望有能力的读者可以下载它的源码并且阅读，相信通过这样的努力，一定可以令自己阅读代码的能力有很大的增强。在一个搜索引擎的开发过程中，使用一个合适的爬虫来获得所需要的网页信息是第一步，这



一步是整个系统成功的基础。因为搜索引擎事实上是一个巨大的资源库，如果从资源角度无法解决用户的需要。那么它也一定不会成功。相信 Heritrix 在今后的版本中会更加完善，功能更为丰富。