DissectingTheNutchCrawler

Dissecting the Nutch 0.5 Crawler

(10/2004 kangas)

Revisions

1.1	20 Apr 2005	moved to wiki.apache.org
1.0	15 Nov 2004	first published on nutch twiki

Table of Contents

Introduction

The open-source Nutch search engine consists, very roughly, of three components:

- the crawler, which discovers and retrieves web pages
- theWebDB, a custom database that stores knownURLs and fetched page contents
- the indexer, which dissects pages and builds keyword-based indexes from them

This document attempts to describe the operation of the crawler. We begin with theory and drill down to into the details needed to create a customized crawler.

Nutch is implemented in Java, so basic knowledge of the language is assumed.

The "nutch" shell script

http://lucene.apache.org/nutch/tutorial.html

The Nutch tutorial describes a number of operations that can be performed using the "bin/nutch" shell script. Looking inside this script, we see that each command corresponds to a specific Java class.

For an intranet crawl, you will edit some config files and then call "bin/nutch crawl ...". This corresponds to the class net.nutch.tools.CrawlTool.

For a whole-web crawl, you will perform several steps, including:

```
$ bin/nutch admin db -create
$ bin/nutch inject db ...
$ bin/nutch generate db segments
$ bin/nutch fetch ...
$ bin/nutch updatedb ...
$ bin/nutch analyze ...
```

Each command corresponds to a Java class as follows:

- admin: net.nutch.tools.WebDBAdminTool
- inject: net.nutch.db.WebDBInjector
- generate: net.nutch.tools.FetchListTool
- fetch: net.nutch.fetcher.Fetcher
- updatedb: net.nutch.tools.UpdateDatabaseTool
- analyze: net.nutch.tools.LinkAnalysisTool

These commands can be specified using either their nickname, or by their full class name. Thus, the following two commands have the same effect:

```
$ bin/nutch admin db -create
$ bin/nutch net.nutch.tools.WebDBAdminTool db -create
```

The ability to invoke arbitrary Java classes will come in handy when we want to customize the behavior of the basic Nutch operations. Let's see how we might do that by examining the one-step intranet crawler.

Command "crawl": net.nutch.tools.CrawlTool

CrawlTool is a class that does little more than lash together the steps you'd do manually for a whole-web crawl. It consists of two simple static methods, plus a main(). Here is an outline of its operations:

```
- start logger:                        LogFormatter.getLogger(...)
- load "crawl-tool.xml" config file:   NutchConf.addConfResource(...)
- read arguments from command-line
- create a new web db:                   WebDBAdminTool.main(...)
- add rootURLs into the db:             WebDBInjector.main(...)
- for 1 to depth (=5 by default):
    - generate a new segment:            FetchListTool.main(...)
    - fetch the segment:                  Fetcher.main(...)
    - update the db:                     UpdateDatabaseTool.main(...)
- comment:
    "Re-fetch everything to get complete set of incoming anchor texts"
- delete all old segment data:         FileUtil.fullyDelete(...)
- make a single segment with all pages:FetchListTool.main(...)
- re-fetch everything:                    Fetcher.main(...)
- index:                                 IndexSegment.main(...)
- dedup:                                 DeleteDuplicates.main(...)
- merge:                                 IndexMerger.main(...)
```

Translating this into the equivalent "nutch" script commands, we can see how similar this is to the whole-web crawling process:

```
- (start logger, etc)
- bin/nutch admin db -create
- bin/nutch inject db ...
- (for 1 to depth:)
   - bin/nutch generate ...
   - bin/nutch fetch ...
   - bin/nutch updatedb ...
- (call net.nutch.FileUtil.fullyDelete(...))
- bin/nutch generate ...
- bin/nutch index ...
- bin/nutch dedup ...
- bin/nutch merge ...
```

If we wished to customize **CrawlTool**, we could easily copy its contents to another class, edit, compile, then run it via "bin/nutch" using its full class name. But, as you can see, there isn't much here to customize! The actual work of makingHTTP requests is occurs inside Fetcher.main().

Let's examine the steps that occur before Fetcher.main(...), then dive into the crawler itself.

Command "admin -create": net.nutch.tools.WebDBAdminTool

> "admin: database administration, including creation"

> Usage: `java net.nutch.tools.WebDBAdminTool db [-create] [-textdump dumpPrefix] [-scoredump] [-top k]`

The "-create" options is a wrapper around "WebDBWriter.createWebDB(directory)". This in turn instantiates one **WebDBWriter** object with the arguments (dir, true) and then immediately calls ".close()" on the object.

Using "spam" as a directory name, let's run it and see what it creates:

```
$ bin/nutch admin spam -create
$ find spam -type file | xargs ls -l
-rw-r--r--  1 kangas  users   0 Oct 25 18:31 spam/dbreadlock
-rw-r--r--  1 kangas  users   0 Oct 25 18:31 spam/dbwritelock
-rw-r--r--  1 kangas  users  16 Oct 25 18:31 spam/webdb/linksByMD5/data
-rw-r--r--  1 kangas  users  16 Oct 25 18:31 spam/webdb/linksByMD5/index
-rw-r--r--  1 kangas  users  16 Oct 25 18:31 spam/webdb/linksByURL/data
-rw-r--r--  1 kangas  users  16 Oct 25 18:31 spam/webdb/linksByURL/index
-rw-r--r--  1 kangas  users  16 Oct 25 18:31 spam/webdb/pagesByMD5/data
-rw-r--r--  1 kangas  users  16 Oct 25 18:31 spam/webdb/pagesByMD5/index
-rw-r--r--  1 kangas  users  16 Oct 25 18:31 spam/webdb/pagesByURL/data
-rw-r--r--  1 kangas  users  16 Oct 25 18:31 spam/webdb/pagesByURL/index
```

Command "inject": net.nutch.db.WebDBInjector

> "inject: inject new urls into the database"

> Usage: `WebDBInjector <db_dir> (-urlfile <url_file> | -dmozfile <dmoz_file>) [-subset <subsetDenominator>] [-includeAdultMaterial] [-skew skew] [-noDmozDesc] [-topicFile <topic list file>] [-topic <topic> [-topic <topic> [...]]]`

WebDBInjector.main() accepts two input-type options. "-urlfile" parses a simple list ofURLs with oneURL per line. "-dmozfile" is for parsingDMOZRDF files, which is useful for bootstrapping a whole-web database.

Let's see how it works. Create a file with oneURL, then run "bin/nutch inject":

```
$ vi spam_url.txt
$ bin/nutch inject spam -urlfile spam_url.txt
$ find spam -type file | xargs ls -l
-rw-r--r--  1 kangas  users    0 Oct 25 18:57 spam/dbreadlock
-rw-r--r--  1 kangas  users    0 Oct 25 18:57 spam/dbwritelock
-rw-r--r--  1 kangas  users   16 Oct 25 18:57 spam/webdb/linksByMD5/data
-rw-r--r--  1 kangas  users   16 Oct 25 18:57 spam/webdb/linksByMD5/index
-rw-r--r--  1 kangas  users   16 Oct 25 18:57 spam/webdb/linksByURL/data
-rw-r--r--  1 kangas  users   16 Oct 25 18:57 spam/webdb/linksByURL/index
-rw-r--r--  1 kangas  users   89 Oct 25 18:57 spam/webdb/pagesByMD5/data
-rw-r--r--  1 kangas  users   97 Oct 25 18:57 spam/webdb/pagesByMD5/index
-rw-r--r--  1 kangas  users  115 Oct 25 18:57 spam/webdb/pagesByURL/data
-rw-r--r--  1 kangas  users   58 Oct 25 18:57 spam/webdb/pagesByURL/index
-rw-r--r--  1 kangas  users   17 Oct 25 18:57 spam/webdb/stats
```

We can see that a new "stats" file was created, and the data/index files in the "pagesBy..." directories were modified.

Command "generate": net.nutch.tools.FetchListTool

> "generate: generate new segments to fetch"

> Usage: `FetchListTool <db_dir> <segment_dir> [-refetchonly] [-anchoroptimize linkdb] [-topN N] [-cutoff cutoffscore] [-numFetchers numFetchers] [-adddays numDays]`

FetchListTool is used to create one or more "segments". From the tutorial:

<blockquote>
Each segment is a set of pages that are fetched and indexed as a unit.
Segment data consists of the following types:

- a "fetchlist": file that names the pages to be fetched
- the "fetcher output": set of files containing the fetched pages
- the "index" is a Lucene-format index of the fetcher output

</blockquote>

Within **CrawlTool**.main(), **FetchListTool**.main() is invoked once per "depth" value with two arguments: (dir + "/db", dir + "/segments"). After processing args, it creates an instance of itself, calls "flt.emitFetchList()", then returns.

Let's run **FetchListTool** to see what it changes on disk. Note that we have to specify the webdb directory, plus another directory where segments are written to.

```
$ bin/nutch generate spam spam_segments
$ find spam -type file | xargs ls -l
-rw-r--r--  1 kangas  users    0 Oct 25 20:18 spam/dbreadlock
-rw-r--r--  1 kangas  users    0 Oct 25 20:18 spam/dbwritelock
-rw-r--r--  1 kangas  users   16 Oct 25 20:18 spam/webdb/linksByMD5/data
-rw-r--r--  1 kangas  users   16 Oct 25 20:18 spam/webdb/linksByMD5/index
-rw-r--r--  1 kangas  users   16 Oct 25 20:18 spam/webdb/linksByURL/data
-rw-r--r--  1 kangas  users   16 Oct 25 20:18 spam/webdb/linksByURL/index
-rw-r--r--  1 kangas  users   89 Oct 25 20:18 spam/webdb/pagesByMD5/data
-rw-r--r--  1 kangas  users   97 Oct 25 20:18 spam/webdb/pagesByMD5/index
-rw-r--r--  1 kangas  users  115 Oct 25 20:18 spam/webdb/pagesByURL/data
-rw-r--r--  1 kangas  users   58 Oct 25 20:18 spam/webdb/pagesByURL/index
-rw-r--r--  1 kangas  users   17 Oct 25 20:18 spam/webdb/stats
$ find spam_segments/ -type file | xargs ls -l
-rw-r--r--  1 kangas  users  113 Oct 25 20:18 spam_segments/20041026001828/fetchlist/data
-rw-r--r--  1 kangas  users   40 Oct 25 20:18 spam_segments/20041026001828/fetchlist/index
```

Note that no changes occurred under the webdb dir ("spam"), but a new segments directory was created, and data+index files created therein.

Command "fetch": net.nutch.fetcher.Fetcher

> "fetch: fetch a segment's pages"

> Usage: `Fetcher [-logLevel level] [-showThreadID] [-threads n] dir`

So far we've created a webdb, primed it withURLs, and created a segment that a Fetcher can write to. Now let's look at the Fetcher itself, and try running it to see what comes out.

net.nutch.fetcher.Fetcher relies on several other classes:

- **FetcherThread**, an inner class
- net.nutch.parse.ParserFactory
- net.nutch.plugin.PluginRepository
- and, of course, any "plugin" classes loaded by the **PluginRepository**

Fetcher.main() reads arguments, instantiates a new Fetcher object, sets options, then calls run(). The Fetcher constructor is similarly simple; it just instantiates all of the input/output streams:

instance variable	**class**	**arguments**
fetchList	**ArrayFile**.Reader	(dir, "fetchlist")
fetchWriter	**ArrayFile**.Writer	(dir, "fetcher", **FetcherOutput**.class)
contentWriter	**ArrayFile**.Writer	(dir, "content", Content.class)
parseTextWriter	**ArrayFile**.Writer	(dir, "parse_text", **ParseText**.class)
parseDataWriter	**ArrayFile**.Writer	(dir, "parse_data", **ParseData**.class)

Fetcher.run() instantiates 1..threadCount **FetcherThread** objects, calls thread.start() on each, sleeps until all threads are gone or a fatal error is logged, then calls close() on the i/o streams.

FetcherThread is an inner class of net.nutch.fetcher.Fetcher that extends java.lang.Thread. It has one instance method, run(), and three static methods: handleFetch(), handleNoFetch(), and logError().

FetcherThread.run() instantiates a new **FetchListEntry** called "fle", then runs the following in an infinite loop:

1. If a fatal error was logged, break
2. Get the next entry in the **FetchList**, break if none remain
3. Extract url from **FetchListEntry**
4. If the **FetchListEntry** is not tagged "fetch", call this.handleNoFetch() with status=1. This in turn does:
    - Get MD5Hash.digest() of url
    - Build a **FetcherOutput**(fle, hash, status)
    - Build empty Content, **ParseText**, and **ParseData** objects
    - Call Fetcher.outputPage() with all of these objects
5. If is tagged "fetch", call **ProtocolFactory** and get Protocol and Content objects for this url
6. Call this.handleFetch(url, fle, content). This in turn does:
    - Call **ParserFactory**.getParser() for this content type
    - Call parser.getParse(content)
    - Call Fetcher.outputPage() with a new **FetcherOutput**, including url MD5, the populated Content object, and a new **ParseText**
7. On every 100th pass through loop, write a status message to the log
8. Catch any exceptions and log as necessary

As we can see here, the fetcher relies on Factory classes to choose the code it uses for different content types: **ProtocolFactory**() finds a Protocol instance for a given url, and **ParserFactory** finds a Parser for a given contentType.

It should now be apparent that implementing a custom crawler with Nutch will revolve around creating new Protocol/Parser classes, and updating **Protocol Factory**/ParserFactory to load them as needed. Let's examine these classes now.

Factory classes: Overview

> Class net.nutch.parser.ParserFactory

> used by:

> - net.nutch.db.WebDBInjector

> - net.nutch.fetcher.Fetcher

> - net.nutch.parser.ParserChecker

>

> Class net.nutch.protocol.ProtocolFactory

> used by:

> - net.nutch.fetcher.Fetcher

> - net.nutch.parser.ParserChecker

>

> Class net.nutch.net.URLFilterFactory

> used by:

> - net.nutch.db.WebDBInjector

> - net.nutch.tools.UpdateDatabaseTool

>

> Class net.nutch.plugin.PluginRepository: used by (Parser/Protocol)Factory

Nutch's **ParserFactory** and **ProtocolFactory** classes are the key extension points for the crawler. **URLFilterFactory** additionally provides an extension point for other components, including **WebDBInjector** and **UpdateDatabaseTool**. These "Factory" classes can all be reconfigured by editingXML config files. So before we describe the mechanics of any of the Factory classes, we need take a quick look at Nutch's configuration system.

Aside: net.nutch.util.NutchConfig

If you have been reading the code along with our discussion, you may have noticed several "private static final" variables at the start of the "command" class definitions. For example, net.nutch.db.WebDBInjector has these definitions for DEFAULT_INTERVAL and NEW_INJECTED_PAGE_NAME:

```
private static final byte DEFAULT_INTERVAL =
  (byte)NutchConf.getInt("db.default.fetch.interval", 30);

private static final float NEW_INJECTED_PAGE_SCORE =
 NutchConf.getFloat("db.score.injected", 2.0f);
```

The values are loaded by calls to net.nutch.util.NutchConf, which is, intuitively enough, a class that loads configuration files. It has two static variables, "List resourceNames" and "Properties properties".The class has several static methods to manipulate these variables. Here's a summary of its operations:

1. resourceNames is initialized with the strings "nutch-default.xml" and "nutch-site.xml"
2. "properties" is initially null
3. A call to one of the "getXXX" methods results in a call to getProps(). If (properties == null), loadResource() is successively called with the values from "resourceNames".
4. loadResource() loads each file, parses theXML, and sets values in "properties" per the config

Factory classes: **URLFilterFactory**

> Class net.nutch.net.URLFilterFactory

> used by:

> - net.nutch.db.WebDBInjector

> - net.nutch.tools.UpdateDatabaseTool

URLFilterFactory is not strictly part of the crawler, but it is a good extension point within Nutch. Here's how it works:

1. When the class is loaded, URLFILTER_CLASS is set to the value returned by **NutchConf** for the key "urlfilter.class"
2. When getFilter() is called, it checks to see if the filter class has already been loaded. If not, we load it using Class.forName(URLFILTER_CLASS), and the class is returned.

It loads one class, which is configurable via "urlfilter.class". By default, nutch-default.xml specifies this as follows:

```
<!-- urlfilter properties -->

<property>
  <name>urlfilter.class</name>
  <value>net.nutch.net.RegexURLFilter</value>
  <description>Name of the class used to filterURLs.</description>
</property>

<property>
  <name>urlfilter.regex.file</name>
  <value>regex-urlfilter.txt</value>
  <description>Name of file onCLASSPATH containing default regular
  expressions used byRegexURLFilter.</description>
</property>
```

Now let's look at the crawler factories, which are a bit more complex.

Factory classes: **ParserFactory**, **ProtocolFactory**

> Class net.nutch.parser.ParserFactory

> used by:

> - net.nutch.db.WebDBInjector

> - net.nutch.fetcher.Fetcher

> - net.nutch.parser.ParserChecker

>

> Class net.nutch.protocol.ProtocolFactory

> used by:

> - net.nutch.fetcher.Fetcher

> - net.nutch.parser.ParserChecker

>

> Class net.nutch.plugin.PluginRepository: used by all of the above

ParserFactory and **ProtocolFactory** are called directly from net.nutch.fetcher.Fetcher, to get the appropriate Parser and Protocol objects for a given content_type and url. They both use an instance of net.nutch.plugin.PluginRepository to find and load Java classes.

By default, nutch-default.xml tells **PluginRepository** to look for classes in a directory called "plugins" somewhere on the Java classpath. Normally you'll just use the one in your Nutch install directory.

```
<!-- plugin properties -->

<property>
  <name>plugin.folders</name>
  <value>plugins</value>
  <description>Directories where nutch plugins are located.  Each
  element may be a relative or absolute path.  If absolute, it is used
  as is.  If relative, it is searched for on the classpath.</description>
</property>
```

Inside the plugin directory you will find a handful of sub-directories, each containing a file called "plugin.xml" and one or more Java archive (.jar) files. Directories include:

- parse-html
- parse-text
- parse-msword
- parse-pdf
- protocol-file
- protocol-ftp
- protocol-http

One directory, plus the "plugin.xml" and .jar file contents, constitutes one "plugin".

TheXML file is a descriptor that is read by **PluginRepository** to determine two main things:

1. What "extension point" (Java interface) the plugin implements, and b. how to load its contents.

Here is the plugin.xml file for "protocol-file":

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="protocol-file"
   name="File Protocol Plug-in"
   version="1.0.0"
   provider-name="nutch.org">

   <extension-point
      id="net.nutch.protocol.Protocol"
      name="Nutch Protocol"/>

   <runtime>
      <library name="protocol-file.jar">
         <export name="*"/>
      </library>
   </runtime>

   <extension id="net.nutch.protocol.file"
              name="FileProtocol"
              point="net.nutch.protocol.Protocol">

      <implementation id="net.nutch.protocol.file.File"
                      class="net.nutch.protocol.file.File"
                      protocolName="file"/>
   </extension>
</plugin>
```

Since the plugin is named "protocol-file", you probably guessed already that this is a protocol handler for loading files on disk. But this descriptor tells us – and **PluginRepository** – precisely what it does:

- the extension-point (Java interface) name is "net.nutch.protocol.Protocol"
- the protocolName is "file"

Thus, when Nutch sees aURL that starts with "file://", it will know to call this plugin to fetch that page.

Look at the descriptors for "protocol-http" and "protocol-ftp". You should see that the extension-point is exactly the same as for protocol-file, but the protocolName is different: "http" and "ftp", respectively.

Now let's examine the descriptor for parse-text:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="parse-text"
   name="Text Parse Plug-in"
   version="1.0.0"
   provider-name="nutch.org">

   <extension-point
      id="net.nutch.parse.Parser"
      name="Nutch Content Parser"/>

   <runtime>
      <library name="parse-text.jar">
         <export name="*"/>
      </library>
   </runtime>

   <extension id="net.nutch.parse.text"
              name="TextParse"
              point="net.nutch.parse.Parser">

      <implementation id="net.nutch.parse.text.TextParser"
                      class="net.nutch.parse.text.TextParser"
                      contentType="text/plain"
                      pathSuffix="txt"/>
   </extension>
</plugin>
```

Note that the extension-point is now net.nutch.parse.Parser. And this time, `<extension><implementation>` doesn't specify a protocolName. Instead, we see "contentType" and "pathSuffix".

So now we see how **PluginRepository** chooses which plugin to use for a given task:

1. It finds the set of plugins that implement a certain extension-point
2. Then, from that set, it finds one that works for the content at hand (protocolName, contentType, or pathSuffix).

Look at the descriptor for parse-html. You'll see that it follows these rules. It implements the same extension-point as parse-text (net.nutch.parse.Parser), but it has different values for contentType and pathSuffix values:

```
    contentType="text/html"
    pathSuffix=""
```

This entry looks a bit strange with the empty pathSuffix value. But that just means that this plugin doesn't match any pathSuffix value. So, parse-html is only used when we fetch remoteURLs, not anything residing on the local filesystem.

Summary: Nutch crawler extension points

The main ways to configure the Nutch crawler are as follows:

1. Configuration files. Default values are in nutch-default.xml, and you should override them in nutch-site.xml.
2. URLFilter interface. By default, the class `net.nutch.net.RegexURLFilter` is used, which reads regular expression patterns from regex-urlfilter.txt. So, you can:
   - Edit that file to tune its behavior
   - Or, write a new class that implements `net.nutch.net.URLFilter`, and change nutch-site.xml to use it.
3. Protocol interface. To add support for a new protocol, write or add a plugin to the "plugins" directory. To change protocol behavior, modify the appropriate plugin.
4. Parser interface. As for Protocol, you should add/create a plugin for any new content-types. Otherwise, you will need to replace the appropriate plugin if you want to modify its behavior.
5. If you need to make other changes, refer to our discussion of **Fetcher** and **FetchListTool**. Consider subclassing these classes, overriding the appropriate method, then calling your class from the "nutch" script using the full class path.

--MattKangas - 15 Nov 2004