Java Libraries Optimization:
SmartLinkerConfGenerator user's manual.

Introduction:

SmartLinkerConfGenerator is a command-line Java application, designed to ease the optimization process of Java libraries used by a given Java application. From now on we will call the latter, "user application".

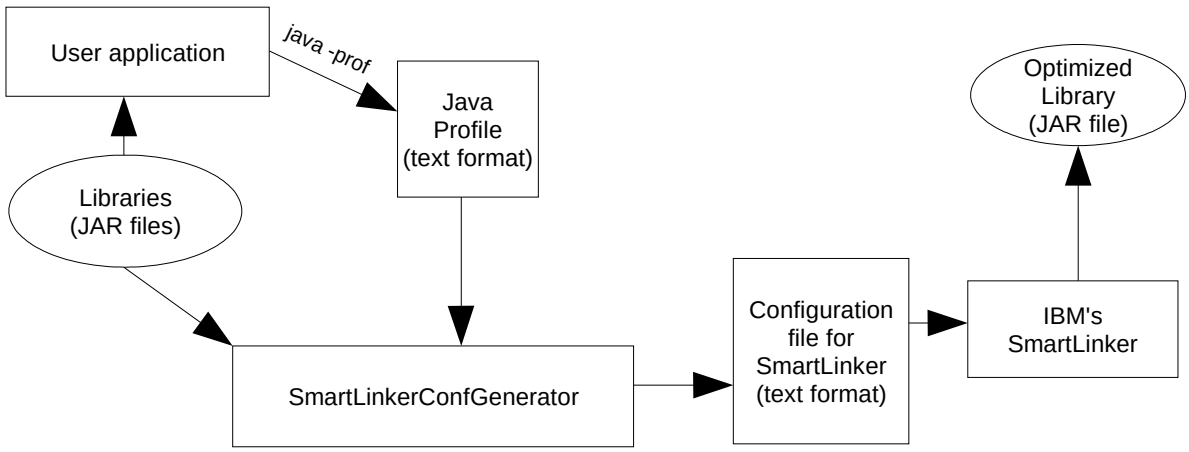
Specifically, SmartLinkerConfGenerator's purpose is, as its name points out, to create configuration files for IBM's SmartLinker, by means of analysis of the libraries to be optimized and user guidance.

SmartLinker is an IBM application used to optimize applications developed for embedded systems, which use, for example J2ME with its different profiles (CLDC, MIDP, etc). You can get a SmartLinker evaluation version on IBM's website (www.ibm.com), integrated in products like IBM WebSphere Device Developer (WSDD).

There are several required elements to optimize Java libraries with SmartLinkerConfGenerator:

- The profile of a particular execution of the user aplication, obtained through "java -prof".
- The libraries that are going to be optimized itself (JAR files).
- IBM SmartLinker application (jxelink), that eventually will be run using the configuration file obtained from SmartLinkerConfGenerator execution.

The following figure shows the general procedure used to optimize Java libraries:



Note that the resulting optimized libraries are intrinsically not 100% reliable. Since the process of optimization is based on a particular execution of the user application, we can only guarantee that the user application will work in that specific execution path and not in other ones. For that reason, as we will see later, it is very important to get the most exhaustive execution path when profiling the user application.

Utilization:

To run SmartLinkerConfGenerator just call the Java interpreter over the main class. It is important to remark that SmartLinkerConfGenerator has to be run with the same *classpath* as the user application had when its profile was obtained.

Its general syntax is:

```
SmartLinkerConfGenerator [-o outputfile] library1.jar [library2.jar]...
                         java.prof_to_parse
```

- *configuration_outputfile* – the full path where the configuration file will be created. By default a file will be created in the current directory, called "`jxelink.jxeLinkOptions`".
- *libraryx.jar* – the full path to the library or libraries to be optimized. It is possible to optimize several libraries at the same time, but note that the resulting optimized library will be a single file with the union of the optimized input libraries.
- *java.prof_to_parse* – the full path where the user application execution profile is.

In the next section we will make a ficticious example (using Linux command syntax) to explain the full process of Java libraries optimization.

Step 1: User application profile

In this step we will get the execution profile of the user application. For this purpose we run it using the "profiling mode" of the java interpreter (java -prof). For example, let's suppose the main class of our user application is called "BarkIDS" and it requires four libraries (which paths now we will see); then the invocation would be like this:

```
java -prof -cp /usr/lib/java/axis.jar:/usr/lib/java/jmf.jar:\
/usr/lib/java/jain-sip.jar:/usr/lib/java/jamon.jar BarkIDS
```

To get the most complete profile possible, it is necessary to make the execution cover the highest number of execution paths, in other words, go through all application's features and use cases. At the end of the execution we will have a file called "`java.prof`" (located in the current directory) which contains the user application's execution profile.

Step 2: SmartLinkerConfGenerator

In this step we will use SmartLinkerConfGenerator to obtain an adapted-for-our-necessities configuration file for SmartLinker. For this purpose, we just need to run SmartLinkerConfGenerator with the file obtained in the previous step.

Let's say we want to optimize the library "`/usr/lib/java/axis.jar`", the profile file we got in the previous step is located at "`/home/yo/java.prof`" and, for example, we want SmartLinkerConfGenerator to create the configuration file at "`/home/yo/axisOpt.jxeLinkOptions`". Thus, the invocation would be:

```
java -cp /usr/lib/java/axis.jar:/usr/lib/java/jmf.jar:/usr/lib/java/jain-sip.jar:\
/usr/lib/java/jamon.jar SmartLinkerConfGenerator \
-o /home/yo/axisOpt.jxeLinkOptions /usr/lib/java/axis.jar \
/home/yo/java.prof
```

Note that the *classpath* given to the Java Virtual Machine has to be exactly the same as the given in the execution of the user application (see step 1).

Once invoked, SmartLinkerConfGenerator will ask some questions:

1) After showing the SmartLinker default basic options, it asks if we want to modify them, and if it is the case, then it will ask for the path of the text file that contains the desired basic options.
In our example we won't change them, so we answer "no".

2) Desired full path where the optimized library (JAR file) will be placed by SmartLinker. For example we enter "`/home/yo/axis-optimizado.jar`".

3) Do you want to automatically add all resources from the library files? This question asks if we want to include inside the final JAR file (optimized library) other resources, contained in the libraries being optimized, for example property files (e.g. org/apache/axis/axis.properties) or manifest files (e.g. META-IN/MANIFEST.MF).

- It's recommended to answer affirmatively, so in our example we enter "y".
- If we answer no, then the program will ask if we want to manually enter any resource, if so we will have to enter the paths of the desired resources (e.g. org/apache/axis/axis.properties).

4) Do you want to explicitly include classes or packages from the library files? This question asks if we want to enforce the inclusion of some classes (or entire packages, asterisk as wildcard is available) in the final library.

SmartLinkerConfGenerator and SmartLinker can leave some neccesary classes out (see Technical Considerations chapter), so we surely need to iterate over step 2 until all neccesary classes are manually added. The names of these classes will be easily identified, because their names will appear in the `ClassNotFoundException` exceptions thrown during the user application executed against the optimized libraries.
In order to make easier the insertion of these names, SmartLinkerConfGenerator give us the possibility to introduce them from an external text file (with one class or package name per line).
In our example we will answer affirmatively this question and enter "`org.apache.axis.*`" so that we force SmartLinker to include all classes from the package named "`org.apache.axis`".

5) After answering these questions, SmartLinkerConfGenerator will begin to work until it creates the required configuration file.

Step 3: SmartLinker (jxelink)

The previously obtained configuration file is used as input for SmartLinker. After the execution it will produce the optimized library we wanted.

In our example, the invocation syntax would be like this:

```
jxelink @/home/yo/axisOpt.jxeLinkOptions
```

After several warning messages, SmartLinker will have generated the optimized library at "`/home/yo/axis-optimizado.jar`".

Step 4: Tests

At this point, we only have to test the user application with the optimized library and check that no necessary classes have been omitted.

For that matter we run the user application as we did in step 1, but turning off profiling mode and replacing in the *classpath* the old library/libraries with the optimized ones.

In the example:

```
java -cp /home/yo/axis-optimizado.jar:/usr/lib/java/jmf.jar:\
/usr/lib/java/jain-sip.jar:/usr/lib/java/jamon.jar BarkIDS
```

If the application runs without problems it means the optimization procedure was successful. Otherwise we will have to repeat the procedure from step 2, entering the names of the missing classes.

TECHNICAL CONSIDERATIONS

SmartLinkerConfGenerator operation:

The implementation of SmartLinkerConfGenerator is based on the concatenation of the texts successively generated through tree stages, in order to create what eventually will be the configuration input file for SmartLinker:

1º SmartLinker basic options: default ones can be used, they are implemented inside the program, but others can be loaded from an external text file.
2º Options dynamically generated on the basis of the given parameters, such as the classpath and output file for SmartLinker (optimized library) and the library's included resources.
3º List of the necessary classes for the execution of the user application. This list is obtained through the user application profile analysis using the ProfileParser application.

ProfileParser operation:

This application extracts all class names that appears in the given profile, provided that they belong to the given package names.

Though this application is intertally used from SmartLinkerConfGenerator, it's also possible to use it as a standalone application. Its syntax is:

```
ProfileParser [-smartlinker] [-o output_file] [-vector] prof_to_parse [package1]..
```

With the options:

- *smartlinker*: this option enables SmartLinker syntax. If activated ProfileParser will return all class names preceded with the string "`-includeWholeClass`".
- *output_file*: specifies the output file. If it's omitted, standard output will be used.
- *vector*: this option tells the program that the data will be internally retrieved from another application (like SmartLinkerConfGenerator) through `returnResult()` method whereby ProfileParser should not give any external output, neither to the screen nor to a file.
- *prof_to_parse*: full path to the profile file to be processed.
- *packageN*: list of the packages to filter the classes names with. ProfileParser will only show the names that belong to any of these packages. If this option is omitted, all class names will be shown.

When run, ProfileParser registers each and every class name that appears in the profile, obtains the names of it's superclass and implemented interfaces, registering them recursively until the profile is read to the end. This is the way it eventually gets the names of all classes that appears in the profile and all classes related by inheritance or implementation.

Necessary classes omission:

Omission of necessary classes for the execution of the user application is an undesirable effect, caused when either the name of a class does not appear in the execution profile (what happens for example when a class is loaded but never instantiated), or it is not related (by inheritance or implementation) with any other that appears, or SmartLinker is not able to include it (in spite of its code analysis).

When this happens, the only solution is to run the user application, writing down the name of the missing class (when `ClassNotFoundException` exceptions are thrown), and make the iterate over the optimization procedure until the user application runs properly.

REAL CASES

Optimization of Apache's AXIS libraries for an user application called "IDS":

Invocation:

```
java -cp $CPATH SmartLinkerConfGenerator $* \
      /usr/java/axis-1_1/lib/axis.jar \
      /usr/java/axis-1_1/lib/commons-discovery.jar \
      /usr/java/axis-1_1/lib/commons-logging.jar \
      /usr/java/axis-1_1/lib/jaxrpc.jar \
      /usr/java/axis-1_1/lib/saaj.jar \
      /usr/java/axis-1_1/lib/wsdl4j.jar \
      /home/rafaelb/PFC/Profiles/ids-oscar-standalone-todosusos.prof
```

With CPATH variable being the same classpath of the user application when the profile was obtained:

"`/home/rafaelb/PFC/Profiles/ids-oscar-standalone-todosusos.prof`"

Class and package names manually entered (initially omitted by SmartLinker):

```
org.apache.axis.encoding.ser.*
org.apache.axis.types.*
org.apache.axis.NoEndPointException
org.apache.axis.transport.java.*
org.apache.axis.transport.http.*
org.apache.axis.transport.local.*
org.apache.axis.components.net.*
```

Results:

Before: total sum: 1.467KB
After: 1.100KB
Saving: 25%

Optimization of NIST's SIP libraries for an user application called "IDS":

Invocation:

```
java -cp $CPATH SmartLinkerConfGenerator $* \
      /usr/java/jain-sip/JainSipApi1.1.jar \
      /usr/java/jain-sip/nist-sip-1.2.jar \
      /usr/java/jain-sip/nist-sdp-1.0.jar \
      /home/rafaelb/PFC/Profiles/ids-oscar-standalone-todosusos.prof
```

Class and package names manually entered (initially omitted by SmartLinker):

```
gov.nist.javax.sip.parser.*
gov.nist.javax.sip.header.*
gov.nist.javax.sdp.parser.*
```

Results:

Before: total sum: 507KB
After: 421KB
Saving: 17%