

Packaging with Javahelper

Matthew Johnson mjj29@debian.org

1st August 2010

Abstract

Programs and libraries written in Java have a lot of special packaging requirements in common. When I started packaging Java programs I was annoyed at the lack of tools to automate some of these tasks and hence Javahelper was born.

Javahelper provides tools to help with the work flow right from processing upstream releases (which can be tricky with some Java upstreams) through building packages where upstream doesn't have a sane build system to generating packaging meta-data and installing files into packages.

This talk gives an overview of the DH7 and CDBS integration, how to use the various helper tools available in javahelper and finishes with example packages of a selection of simple Java programs and libraries.

The aim of Javahelper is to provide tooling to implement Debian Java policy, automate common tasks required by Java packages that aren't provided for by debhelper and to simplify the packaging of Java software.

Upstream

Upstream releases

- Lots of jar or zip releases not tars
- Releases include 3rd party libraries
- Releases include build products (javadoc, class files, etc)
- Repacking needs to be done with every new release
- Solution: `jh_repack`

`jh_repack` will remove any `.class` files, any `.jar` files, the whole directory tree containing javadoc and any empty directories as a result of the above.

Figure 1: jh_repack watch file

```
version=3
http://www.matthew.ath.cx/projects/salliere/ \
(?:.* /)?salliere-?_?([\d+\.]|[\d+)]\.(jar|zip) \
debian jh_repack
```

Figure 2: jh_repack manual

```
jh_repack --upstream-version <version> <archive>
```

Upstream build systems

- Build system expects 3rd party libraries in the build tree
- Build system broken
- Build system non-existent
- Build system doesn't build javadoc
- Solution: jh_linkjars
- Solution: jh_build

Figure 3: jh_linkjars (old-style)

```
jh_linkjars libs
```

Figure 4: jh_linkjars (debian/linkjars)

```
libs
```

jh_linkjars will scan all of the build dependencies and then symlink every jar which the dependency installs to `/usr/share/java` into the target directory. The directory should either be passed on the command line and then again with `-u` in the clean target, or put in the file `debian/linkjars`.

jh_build is designed as a simple point-and-click build system for packages of the form "take all the .java files in that directory, compile them to .class files and put them in a .jar". This is the case for many packages, especially those without their own build system.

Figure 5: jh.build (old-style)

```
JAVA_HOME=/usr/lib/jvm/default-java
CLASSPATH=/usr/share/java/esd.jar:...
jh_build weirdx.jar src
```

Figure 6: jh.build (DH7—debian/javabuild)

```
salliere.jar src
```

Figure 7: jh.build (DH7—debian/rules)

```
export JAVA_HOME=/usr/lib/jvm/default-java
export CLASSPATH=/usr/share/java/csv.jar:...
```

Figure 8: jh.build (CDBS—debian/rules)

```
export JAVA_HOME=/usr/lib/jvm/default-java
export CLASSPATH=/usr/share/java/csv.jar:...
JH_BUILD_JAR=salliere.jar
JH_BUILD_SRC=src
```

More complex packages should use a full build system such as make, ant or maven—and hopefully upstream will already have provided these.

Nevertheless there are still some options which can be provided to change how it works. The CLASSPATH variable will be copied into the jar manifest automatically, but the main-class attribute must be specified on the command line. It can build javadoc automatically for you and both javac and javadoc can have extra parameters passed to them via command line options to jh.build. Lastly, you can specify other non-class files to be included in the jar via an environment variable. Using these command line parameters requires you to put the source and jar files on the command line as well, and not use the debian/javabuild file.

Finally, for both jh.linkjars and jh.build there is a clean option to remove build products, which is automatically called from jh.clean, which you should call from the clean target.

- Upstreams often fail to set manifest entries
- Debian-specific manifest entries may be needed
- Solution: jh_classpath
- Solution: jh_manifest

Figure 9: `jh_classpath` (debian/classpath)

```
src/my.jar /usr/share/java/dep1.jar /usr/share / ...
src/my-other.jar /usr/share/java/dep1.jar / ...
```

Figure 10: `jh_manifest` (debian/manifest)

```
usr/share/weirdx/weirdx.jar :
Main-Class: com.jcraft.weirdx.WeirdX
Debian-Java-Home: /usr/lib/jvm/default-java
```

Both `jh_manifest` and `jh_classpath` can be called directly without a file specifying everything on the command line or `jh_manifest` can be called to update every jar in the package with the classpath taken from the `CLASSPATH` variable. These commands provide a simple, non-intrusive, way of providing common changes to the upstream build products.

As well as being used at runtime by the JVM to load dependent jars (which works transitively through dependencies), the classpath entry can be used to automatically populate the Depends line of your package. The use of the Debian- prefixed entries will be explained later.

Installing

- There are specific requirements for installation of jars and javadoc in policy
- Code duplication to support those requirements is bad
- Being able to make a single source change then rebuild to change the requirements is good
- Solution: `jh_installlibs`
- Solution: `jh_installjavadoc`

Figure 11: `jh_installlibs` (debian/jlibs)

```
foo.jar
build/bar.jar
```

`jh_installlibs` will install a jar into `/usr/share/java/$jar-$version.jar` with a symlink from `/usr/share/java/$jar.jar` as required by policy. It will attempt to strip

Figure 12: `jh_installjavadoc (debian/javadoc)`

```
doc/api
internal
api /usr/share/doc/libfoo-java/api
```

the version number from the jar and any DFSG suffix from the package version before installing it. Other version-mangling rules or the jars themselves can be provided on the command line.

`jh_installjavadoc` has three forms. Calling it with a single path will install that path to `/usr/share/doc/$package/api`. The destination can be overridden so that a `-doc` package can install the doc in the directory for the library package (providing it depends on it). The final form is the string “internal” which will take the javadoc generated by `jh.build` and install that to the normal directory. In all cases it will generate a doc-base file to automatically register the documentation with doc-base

Binary packages

Runtime support

- Every java program needs a wrapper script
- Looking up the path to `libjvm.so` is hard
- Solution: `jarwrapper`
- Solution: `jh_exec`
- Solution: `java-vars.sh`

`jarwrapper` installs a `binfmt-misc` handler for jar files (technically, for zip files containing a manifest) which calls `java -jar $file`. A jar with a correct classpath and main class entries in the manifest can just be made executable and to depend on `jarwrapper` and in can run just like any other executable. There are a few Debian-specific manifest entries which can be set which result in using a specific JVM or specific arguments being passed to the JVM.

In order to make it easier to use the above, `jh_exec` will scan directories in the package which are on path and if any of those contain symlinks to a `.jar` file it will make that file executable. This means that jars can be installed into `/usr/share/java` or `/usr/share/$package` then `dh_link` used to symlink them to `/usr/bin`, then `jh_exec` will automatically set the modes.

Some applications need to load `libjvm.so` or other files from within the JVM directory. These files are in architecture-specific directories within the JVM, but those architectures do not match the debian architecture name. To simplify finding these files for all packages there is a shell script snippet (`java-vars.sh`) which will export a variety of variables based on your `JAVA_HOME` setting, shipped with `jarwrapper` for programs which need it at runtime. This is also available as a makefile snippet in `javahelper` for use in `debian/rules` which need it at build time.

Dependencies

- Autogenerating Depends: lines (a-la `dh_shlibdeps`) is helpful
- Policy specifies a variety of alternate depends on JREs
- `jarwrapper` is needed in the Depends for executable jars
- A single place that generates dependencies makes it easier to change policy
- Solution: `jh_depends`

`dh_shlibdeps` automatically generates dependencies from the libraries required by a C-library. `jh_depends` attempts to do this for jar files as well, using the classpath manifest entry in the jar. `jh_depends` will scan all the jars in your package, work out which jars each depends on and which packages contain those. These will then be added to the `$java:Depends` substvar for replacing in control files. These dependencies will also include other packages built from the same source in which case a strict versioned depends will be used.

If your package contains any executable jar files, or if you explicitly request it, `jh_depends` will also create a dependency on `jarwrapper` in the first case and a JVM in either. The JVM will be taken from the `Debian-Java-Home` variable if it exists in the manifest, or from the command line, defaulting to `default-jre`. Alternate depends are constructed by scanning all the jars for classfile version and depending on a recent enough virtual package to be able to run them.

If there are any policy changes regarding versioned dependencies (see my other talk) then using `jh_depends` will just require a rebuild and not any source changes.

Eclipse

PDE based build

- PDE-based build (auto-generates ant-build files)
- Clean does not clean properly.

- Solution: `jh_setupenvironment`

`jh_setupenvironment` copies (part of) the upstream sources into another directory that can be discarded after the build. Slow, but it works. This is the same method used to ensure a proper clean in the eclipse package.

`jh_clean` removes the copy if it is in the default location.

External (Orbit) dependencies

- Eclipse plug-ins uses non eclipse libraries.
- Requires OSGi-metadata to be loaded (unless embedded)
- Lazy upstreams embeds the jar and use Bundle-Classpath
- Eclipse needs bundles named by their Symbolic Name.
- Eclipse fetches the bundles from Orbit.
- Setting up Orbit is a tedious task.
- Require-Bundle + OSGi-metadata in the Orbit dependency.
- OSGi-metadata must be added manually, but is generally trivial to make.
- For everything else there is `jh_generateorbitdir`.

Given a list of jar files, `jh_generateorbitdir` will setup the Orbit directory. It reads every thing it needs from the Manifest of the jar file.

`jh_clean` removes all temporary files and also the orbitdir if it is in the default location.

OSGi-metadata can usually be copied from Fedora or from Eclipse Orbit.

Building features

- Feature is a set of related plug-ins.
- Java sources are scattered neatly into separate plug-ins.
- Manifest files lists dependencies.
- The PDE handles all of this, but ...
- It takes over 10 arguments to make PDE build one “trivial” feature.
- LinuxTools/eclipse-build provides a wrapper, but it is still a pain.
- Solution: `jh_compilefeatures`

jh_compilefeatures builds list of features given their ID and a list of the features (or “group of features”) they depend on. Unfortunately it does not resolve the feature dependencies itself—most of the information is available, the code to collect it is missing.

Currently it is a debhelper-compatible interface for the eclipse-build wrapper with a little extra intelligence.

Install features

- Features are compiled into zip files.
- Must be unzipped and located so that eclipse finds it.
- The zip file contains a full copy of Orbit Dependencies.
- Solution: jh_installeclipse

jh_installeclipse just needs the ID of the feature that jh_compilefeatures built and a “feature group” to install it under (usually this can be guessed from the package name). It will also replace Orbit Dependencies that jh_generateorbitdir handled with symlinks and populate `{orbit:Depends}` for your package.

Installs in beneath `/usr/share/eclipse` for architecture independent packages and under `/usr/lib/eclipse` for other packages.

Packaging frameworks

DH7

Figure 13: DH7 debian/rules

```
#!/usr/bin/make -f

export JAVA_HOME=/usr/lib/jvm/default-java
export CLASSPATH=/usr/share/java/csv.jar:...

%:
    dh $@ --with javahelper
```

javahelper works with DH7 using `--with javahelper`. All the commands mentioned above will be called in the correct order and will read from files under `debian/` without anything extra in your `debian/rules`. In the case that you need to supply extra arguments or otherwise change the behaviour, then `override_` stanzas can be used as normal.

CDBS

Figure 14: CDBS debian/rules

```
#!/usr/bin/make -f
export JAVA_HOME=/usr/lib/jvm/default-java
export CLASSPATH=/usr/share/java/csv.jar:...
JH_BUILD_JAR=salliere.jar
JH_BUILD_SRC=src

include /usr/share/CDBS/1/class/javahelper.mk
```

javahelper is similarly integrated with CDBS, with the exception that `jh.build` is called using environment variables. Again, the normaly CDBS override methods can be used, but there are also a number of variables which can be used to pass arguments directly to programs instead.

Package autogeneration

`jh.makepkg` can be used to generate most of a debian dir for a simple Java application or a simple library. It will prompt for a few answers it can't work out and infer some others.

References

More information about Javahelper can be found in the following locations:

- <http://packages.debian.org/javahelper>
- `/usr/share/doc/javahelper`
- <http://pkg-java.alioth.debian.org/>
- <http://pkg-java.alioth.debian.org/examples>
- `debian-java@lists.debian.org`