Debian Packaging

Matthew Palmer

`mpalmer@debian.org`

Overview

- Structure of Binary Packages
- Structure of Source Packages
- Turning one into the other
- Supporting Multiple Distributions

Binary Packages

Every `.deb` is actually an `ar` archive, containing `data.tar.gz` (files for the filesystem), `control.tar.gz` (maintainer scripts and other metadata), and `debian-binary` (containing the packaging version, currently 2.0).
You can manually create or manipulate Debian packages using standard Unix tools – one of the major advantages of the format.
Far better, though, is to use the tools designed for the purpose...

- `dpkg --info x.deb` – Package metadata
- `dpkg --contents x.deb` – File listing
- `dpkg --unpack x.deb` – Extract the package, but don't run the configure scripts
- `dpkg --install x.deb` – Extract, configure, etc

Package Metadata

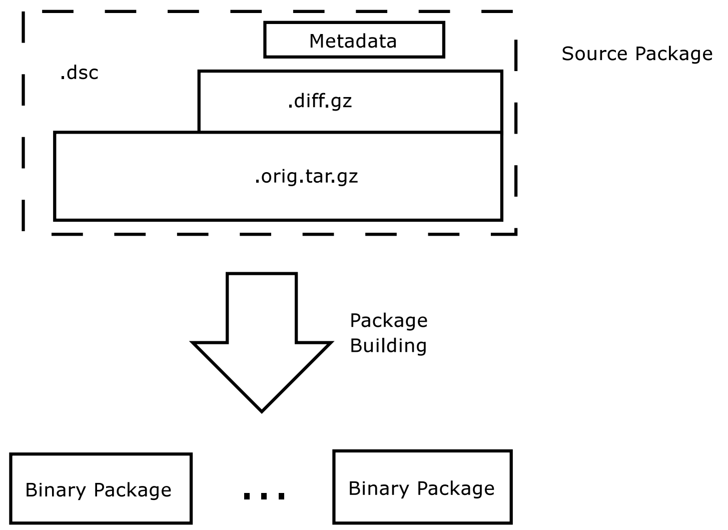
Binary packages have several pieces of metadata associated with them, viewable with `dpkg --info`. Of particular interest:

- `Package:` the actual package name. The filename of the package is totally unimportant.
- `Version:` Oddly enough, the version of the package. Usually comprised of upstream version (portion before the hyphen) and Debian version (portion after the hyphen).
- `Architecture:` 'nuff said.
- `Depends, Recommends, Suggests, Replaces, Conflicts, Enhances:` Fields describing various relationships with other packages.

Source Packages

- The overall control file is the `.dsc` – a description of the source package and appropriate fields to describe build parameters.
- Usually an `.orig.tar.gz`, which should be the original source code as provided by upstream, or close to it; and a `.diff.gz` which contains all of the changes to the upstream source made for Debian.
- “Native” packages have no orig and diff split, but simply put everything into a single tarball. Do not use this package format for most packages.

A Package Diagram



A source package unpacked

Most of the changes in a "Debianised" source package are localised in the `debian` directory. This directory contains the files which control both how the package gets built, what the binary packages are called, and the maintainer scripts used during binary package installation.
Other changes are often needed to source packages, to make them policy conformant or to fix bugs, but these should be minimised by passing changes upstream for integration.

`debian/rules`

It sure does!
More specifically, `debian/rules` is the file which controls how a Debian package is built. It is typically a makefile, defining several targets, corresponding to the various stages of the build process. Some of these are required by policy to be present, although they may do nothing:

- `build` compiles the package
- `binary`, `binary-arch`, and `binary-indep` create the binary packages. `binary` typically invokes `binary-arch` and `binary-indep`.
- `clean` returns the package to it’s original (pre-build) state.

debian/control

This file describes the source package and all binary packages, including their section, architecture, and build and installation dependencies and conflicts.

The .changes file

This is an important file, but only for the upload stage of package development. It describes the changes represented by a given upload – the package affected, files supplied, version, bugs closed, the maintainer, the uploader, and so on.

Maintainer Scripts

The package installation and removal scripts. Typically written in Bourne Shell or Perl, they perform any required configuration and deconfiguration on package installation and removal. The four standard scripts are `preinst`, `postinst`, `prerm`, and `postrm`. There is also a debconf pre-installation script, called `config`, which is supposed to ask the user all sorts of questions, whose answers are used in the other maintainer scripts.

Building a package

The minimum requirement is that the `binary` target of `debian/rules` produce the binary packages specified in `debian/control`. So, in principle, all that someone needs to do is change into the root of the package source tree and run `debian/rules binary`.
How exactly the package is built is left up to the package maintainer when writing the rules file. So, you can have a lot of shell script fragments doing sick and twisted things to produce a package. There are, however, several much better ways of writing `debian/rules` files.

Helpers and their Usefulness

The most commonly used build help system is debhelper, in the package of the same name and documented in debhelper(7) and the great many dh_* commands available.
Each dh_ script performs a particular small, well defined action, such as “install files related to init scripts” and “create defined symlinks”. You can assemble the appropriate dh_ scripts to suit your particular package.

Automatic Packaging

dh_make (package: dh-make) is often used as an initial packaging tool. It creates a debian/ directory full of useful examples and almost-ready-to-go scripts. It is an excellent starting point for most debianisation efforts.
dh-make-perl and dh-make-php (I think that’s the name of the package, anyway) are dh_make alternatives tuned for the particular requirements of packaging Perl modules and PEAR/PECL modules, respectively.

More Advanced Helpers

In the spirit of Free Software, there are several advanced build helpers:

- dpatch;
- dbs;
- cdbs.

And probably several others I’ve forgotten.

Build Helpers

On the package building side, there are several very useful scripts you can use to automate part or all of the build process – the debian/rules binary call, and associated scaffolding.

- dpkg-buildpackage – Cleans the source, builds the binary packages, build description (.dsc) and .changes files, and (by default) signs the package ready for upload.
- debuild – Wraps dpkg-buildpackage, adding extra useful bits like automatic lintian/linda checks.
- pbuilder – Builds a package in a clean chroot environment.
- cvs-buildpackage – Pulls a debian release out of CVS and automatically builds it. There is also an analogous svn-buildpackage (and probably a tla-buildpackage by now, too).

The sid dilemma

You don’t want to run sid on your workstation, but you need to build your packages in sid before uploading them. Or, you’re running sid on your workstation but need to backport some packages to work on your woody-based server. Oh, the humanity!
`pbuilder` to the rescue! Takes a source package description and runs the build process in a pre-built chroot of whatever release you wish to target.

Calling `pbuilder`

- Create the chroot: `pbuilder create --basetgz /var/chroots/woody.tgz --distribution woody`
- Build your package in the chroot: `pbuilder build --basetgz /var/chroots/woody.tgz mypackage_1.1-1.dsc --buildresult ‘pwd‘`
- Periodically, you should update your chroot (especially important when building for sid): `pbuilder update --basetgz /var/chroots/woody.tgz`

Packages of Interest

- `dh-make`
- `devscripts`
- `debhelper`
- `lintian`
- `linda`
- `dpkg-dev`
- `pbuilder`
- `build-essential`