RPM package maintenance

May be used for Red Hat Magazine, Issue 5 (Italy); Issue 1 (US)

A large part of my job concerns maintaining RPM packages for Red Hat Linux. In this article I will show you what that means in practice, some tips and tricks I've picked up, and some special tools I've developed along the way to make my job easier.

The majority of my time at work is spent looking at bug reports. We have a public bug tracking system, named Bugzilla[1], which sends the appropriate developer email each time someone reports a new bug or adds a comment. So I get lots of email from Bugzilla.

Bugzilla uses a web interface which provides quite a flexible way of looking through the bug database. When it sends me email, I take a look at the web page for the bug and see if I can help. Sometimes it has been filed against the wrong program, in which case I have to fix that and reassign the bug to the correct person. Very often I will ask for more information about the problem because not enough was provided in the first place. Occasionally the fix is provided by the person who found the bug (very helpful!), and those bug reports are my favourite.

A good bug report will have enough information in it for me to try to get the problem to happen on the machine I'm using. Once I can see the bug in front of me it is generally much easier to fix it. Of course, there are some bugs that disappear as soon as you try to examine them, only to reappear when you aren't paying attention. Reports of those types of bugs are my least favourite.

When the person who found the bug provides the fix, they very often do so by attaching a **patch** to the bug report. This is a file describing in detail a set of changes to some source code (or any file, but in this case it is usually source code). Once I have a correct patch for a problem, it is a very simple matter to build a fixed package. Of course, if the patch is not correct the process is more complicated; but even an incorrect patch which fixes the symptoms of the problem can be helpful, as it often points out the area of source code that requires more attention.

With Red Hat Linux, all of the software in the operating system is provided as RPM packages, which are essentially collections of programs and data files. The programs have been compiled from source code, and that source code is also provided as an RPM package. Take the example of the `tree` program, which shows a directory listing in tree form (try it in a terminal). The program is `/usr/bin/tree`, which comes from the `tree-1.2-22.i386.rpm` RPM package. This RPM package is built from the source RPM package, `tree-1.2-22.src.rpm`.

Building a binary RPM package (like `tree-1.2-22.i386.rpm`) from a source RPM package is similar to building the compiled program, `/usr/bin/tree`, from the source code, `tree.c`, just at a higher level. The source RPM package contains the original source code, as it was released to the Internet community by the "**upstream**" maintainer, plus whatever patches are needed, plus the

1 https://bugzilla.redhat.com/bugzilla

instructions for how to apply the patches, compile it all, and put the binary RPM package together.

In the case of `tree`, the original source code comes as a **tarball** (compressed archive). There are some patches to fix bugs that were reported in Bugzilla, and finally there is file containing the instructions on what to do with it all: the RPM spec file, `tree.spec`. A cut-down idealised version of this file is shown in Figure 1.

```
1  Summary: A utility which displays a tree view of directories.
2  Name: tree
3  Version: 1.2
4  Release: 22
5  Group: Applications/File
6  License: GPL
7  Source: ftp://metalab.unc.edu/pub/Linux/utils/file/tree-1.2.tgz
8  Patch1: tree-1.2-misc.patch
9
10 %description
11 The tree utility recursively displays the contents of
12 directories in a tree-like format.  Tree is basically a UNIX
13 port of the DOS tree utility.
14
15 %prep
16 %setup -q
17 %patch1 -p1 -b .misc
18
19 %build
20 rm -f tree
21 make CFLAGS="$RPM_OPT_FLAGS"
22
23 %install
24 rm -rf $RPM_BUILD_ROOT
25 make install
26
27 %files
28 %defattr(-,root,root)
29 %{_bindir}/tree
30 %{_mandir}/man1/tree.1*
31 %doc README
32
33 %changelog
34 * Wed Jan 22 2003 Tim Powers <timp@redhat.com>
35 - rebuilt
```

Figure 1: tree.spec

Without going into too much detail about what everything means, let me just point out the main things. The original source tarball is referenced in the 'Source:' line (see line 7), with a URL to where it comes from.

The 'Patch1:' line (see line 8) gives the name of a patch file containing some bug fixes. Further down at line 17, information on how to apply this patch is given ("%patch1 ...").

When this package is built, the `tree-1.2.tgz` archive will be unpacked, and the `tree-1.2-misc.patch` patch will be applied to it. Then the patched source code will be compiled and the resulting program put into a binary RPM package.

Very briefly, this is done by using the `rpmbuild` command. Once `tree.spec`, `tree-1.2.tgz` and `tree-1.2-misc.patch` are all in the right place, `rpmbuild` can make a source RPM package (which just contains those files) or a binary RPM package (which contains the compiled program).

The binary RPM package is created as in Figure 2. Again a cut-down idealised version of what actually happens is shown, and much has been omitted. On line 22 you can see the patch program being executed, and this is how bug fixes are applied during the build process.

```
1  $ rpmbuild -bb tree.spec
2  Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.92157
3  + umask 022
4  + cd /home/tim/BUILD
5  + LANG=C
6  + export LANG
7  + unset DISPLAY
8  + cd /home/tim/BUILD
9  + rm -rf tree-1.2
10 + /usr/bin/gzip -dc /home/tim/SOURCES/tree-1.2.tgz
11 + tar -xf -
12 + STATUS=0
13 + '[' 0 -ne 0 ']'
14 + cd tree-1.2
15 ++ /usr/bin/id -u
16 + '[' 2472 = 0 ']'
17 ++ /usr/bin/id -u
18 + '[' 2472 = 0 ']'
19 + /bin/chmod -Rf a+rX,g-w,o-w .
20 + echo 'Patch #1 (tree-1.2-misc.patch):'
21 Patch #1 (tree-1.2-misc.patch):
22 + patch -p1 -b --suffix .misc -s
23 + exit 0
24 Executing(%build): /bin/sh -e /var/tmp/rpm-tmp.92157
25 + umask 022
26 + cd /home/tim/BUILD
27 + cd tree-1.2
28 + LANG=C
29 + export LANG
30 + unset DISPLAY
31 + rm -f tree
32 + make CFLAGS=-g
33 gcc -g -c -o tree.o tree.c
34 gcc  -o tree tree.o
35 + exit 0
36 [... much omitted ...]
37 Wrote: /home/tim/RPMS/i386/tree-1.2-22.i386.rpm
38 Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.40803
39 + umask 022
40 + cd /home/tim/BUILD
41 + cd tree-1.2
42 + rm -rf /var/tmp/tree-root
43 + exit 0
```

Figure 2: Building the package

When you add a patch to a spec file, you have to remember to add it in two places. Firstly, there must be a line starting “Patch” in the main section at the top, mentioning the file name of the patch file. Secondly there must be a line starting “%patch” in the “%prep” section so that the patch actually gets applied. If you forget to do this, as I have done before, the patch will be in the source RPM package but will not actually be applied when you build the binary RPM package!

Another thing to watch out for is the way that macros in the spec file are expanded. It is not as you would expect. Macros can be expanded to several lines; for instance, “%setup” in the spec file is expanded to form lines 3 to 19 of Figure 2 when building. Wherever “%setup” appears in the spec file, it will be replaced in this manner. In particular, even if you write a comment in the spec file (by starting the line with “#”), the macro will still get expanded. Even worse, that means only the first line will be commented out; the rest of the lines resulting from expanding the macro will be executed! This can cause mystifying failures if you don't know how macros get expanded.

Patch files

Obscure details of spec file syntax aside, what I really want to talk about is patches. As I've already explained, a patch is a file detailing a change in, generally, some source code. I would like to explain what they look like, where they come from, how to use them, and then go on to some more advanced things you can do with them.

One simple patch file is shown in Figure 3. This particular patch fixes Bugzilla bug #54298[2], which is that a warning is produced when compiling the `tree` package. This was a nice bug report, because the reporter kindly included the fix (as a patch). At line 7 of the file, the line beginning with a minus sign shows the C source code at fault. The next line, beginning with a plus sign, shows the line of source code that should be used instead. So this patch file just contains a

2 https://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=54298

```
1   --- tree-1.2/tree.c.54298       Fri Oct  5 11:31:05 2001
2   +++ tree-1.2/tree.c     Fri Oct  5 11:31:19 2001
3   @@ -245,7 +245,7 @@
4         if (uflag && (gflag || sflag || Dflag)) printf(" ");
5         if (gflag) printf("%-8.8s",gidtoname((*dir)->gid));
6         if (gflag && (sflag || Dflag)) printf(" ");
7   -     if (sflag) printf("%9d",(*dir)->size);
8   +     if (sflag) printf("%9lu",(*dir)->size);
9         if (sflag && Dflag) printf(" ");
10        if (Dflag) printf("%s",do_date((*dir)->mtime));
11        if (pflag || sflag || uflag || gflag || Dflag) printf("]  ");
```

Figure 3: Patch file

replacement for one line of source code.

That is what lines 7 and 8 are for, and they are the most important part: the change to be made. Lines 1 and 2 identify which source code file needs changing – `tree-1.2/tree.c` – as well as the time and date that the change was originally made. Line 3, the odd-looking one beginning "@@", shows where in the source file the change should be made; in this case it's near line 245. The remaining lines in the patch show some unchanged source code, just to give context.

A patch file can contain several changes to a source file, and can change several source files. An individual change to a source file, complete with the "@@" **offset line**, is sometimes called a **hunk**. There does not seem to be an agreed word for describing a set of changes to a particular source file as opposed to changes to several source files. The word **patch** is used for both.

As you saw earlier, the `patch` program understands how to apply patch files. It was used when building the `tree` package, Figure 2 (line 22):

```
patch -p1 -b --suffix .misc -s
```

When this command was run, we were already in the `tree-1.2` directory. The `-p1` option means that one directory name should be stripped from the name of the file that the patch says is to be changed. For our patch in Figure 3, that means that instead of looking for `tree-1.2/tree.c`, the file to be changed is just `tree.c` in the current directory. A common problem people new to patches run into is forgetting to use the `-p` option. Although patch tries to find the right file to modify, sometimes it needs some help. Best practice is to create the patch so that it has file names which include the top-level directory, as "`tree-1.2/tree.c`" does, and to apply the patch using `patch -p1` after changing into the top-level directory (with `cd tree-1.2` in this example; this happens on line 14 of Figure 2).

The next option, `-b`, means that a backup should be made of any files that are changed. Then `--suffix .misc` means that the backup file should be called (in this case) `tree.c.misc`. Lastly, `-s` causes `patch` to be silent about what it is up to. Normally it tells you which source file it is changing.

```
1  $ ls
2  tree-1.2
3  $ cp -lR tree-1.2 tree-1.2.orig
4  $ rm tree-1.2/tree.c
5  $ cp tree-1.2.orig/tree.c tree-1.2
6  $ vim tree-1.2/tree.c
7  [...]
8  $ diff -durN tree-1.2.orig tree-1.2 > tree-fix.patch
9  $
```

Figure 4: Creating a patch

The counterpart to `patch` is the program that creates patches: `diff`. One way of using it is to make a copy of the original source code directory, make the necessary changes, and use `diff` to show all the changes between the two directories. Figure 4 shows a good way of doing this. At line 3, the source code directory and all its contents are copied using `cp`; the switches are `-l`, which makes **hard links** rather than copies of the files, and `-R` to copy subdirectories. The reason for making hard links rather than copies is that it can greatly increase the speed at which `diff` can do its job. It can spot that the two file names refer to the same file, and so cannot be different from each other. This can make an immense difference in large projects with many files.

Let us imagine we want to change the `tree.c` file in the source code, as in the example. At this point, the file name `tree-1.2/tree.c` refers to the same file on disk as `tree-1.2.orig/tree.c` does: they are hard-linked, and this is the effect of using `cp -l` to make the copy. To edit the file, we first need to break the link and end up with two copies of the file. Lines 4 and 5 do this.

Then in line 6 the change is made using the `vim` editor. I like to use `vim` for small fixes in source code, rather than `emacs` which takes longer to start. However, I generally use `emacs` for writing new code or for making larger changes.

The patch is created at line 8:

```
diff -durN tree-1.2.orig tree-1.2 > tree-fix.patch
```

The options I've used here are:

- `-d` (can also use `--minimal`): this makes `diff` try harder to find a smaller set of changes. I use this option from force of habit really.
- `-u` (can also use `--unified`): this sets the output format to be “unified format”, which is the format shown in Figure 3.
- `-r` (can also use `--recursive`): to look into subdirectories.
- `-N` (can also use `--new-file`): allows files to be added or removed in the patch output.

There are of course many other options, and they are described very well in the documentation (use `info diff` to read it).

When I am making patches for RPM packages I use a program called `gendiff`. This is a marvellous and simple tool. See Figure 5, which shows the same sort of operation as Figure 4 but uses `gendiff`. You no longer need to make a copy of the source code directory first: only of the files you want to change. Pick a short name for the change (here I've just used “fix”), and make backup copies of the relevant source code files by putting a dot and the short name at the end, as shown. Then go ahead and make the changes you want. Afterwards, when you tell `gendiff` the name of the source code directory and the backup suffix you chose, it will find files ending with that suffix and run `diff`

```
$ ls
tree-1.2
$ cp tree-1.2/tree.c tree-1.2/tree.c.fix
$ vim tree-1.2/tree.c
[...]
$ gendiff tree-1.2 .fix > tree-fix.patch
$
```

Figure 5: Using gendiff

for each.

One fairly routine part of my job is upgrading a package to a new version. While the `tree` package has been at version 1.2 for several years, other packages “rev” (change version) very frequently by comparison. As I go through my Bugzilla list fixing bugs by making patches, I try to ensure I send the patches to the upstream maintainer, but inevitably some get forgotten – usually by me – or are rejected. This leads to the situation where the leftover patches need to be adjusted to fit in with the new version of the source code.

The way I tend to deal with this is just to update the source code tarball and see if the RPM package still builds. It will attempt to apply each patch in turn, and will stop if it finds a problem. Often the problem is that the patch has already been applied – because I'd sent it to the upstream maintainer and they accepted it. In that case the solution is easy: take the patch out of the spec file altogether, since it is no longer needed. Other times the `patch` program can't find the right part of the source code to modify, and in this case it writes a reject file containing the part of the patch that failed. Then it's a case of applying the changes by hand in an editor, which is easy if the reject is small.

Interdiff

Before I joined Red Hat my main interest in Linux was kernel work, as one of the maintainers of the parallel port driver. The Linux kernel source code is large, and so when Linus Torvalds announced a new version I would download the patch (compressed, since even the patch was large) rather than the entire tarball. The patches were all stored on an FTP server, each one labelled with the version which would result after applying it. To get from version 2.2.1 to 2.2.3, I would need to get patch-2.2.2 and patch-2.2.3 and apply them in order. Occasionally (it was occasional then, but more common now) Linus would release a “pre” version. So patch-2.2.4pre5 (say) would apply to 2.2.3 and take it to version 2.2.4pre5. When patch-2.2.4pre6 came along, I would have to “back out” (using `patch -R` to reverse the patch) the previous “pre” patch to get back to 2.2.3, and apply the new one.

I realised that even without the source code available, if I just had patch-2.2.4pre5 and patch-2.2.4pre6, there was enough information in the patch files to allow me to construct a patch file that applied straight on top of 2.2.4pre5. I set to work on a tool to do this, which I called `interdiff`. I wrote it for fun, to see if I could get it to do what I wanted. By July 2000 I had something that seemed mostly to work.

By this time I was working for Red Hat, and was soon maintaining RPM packages as part of my job. I found that I often wanted to extract bits of patches, or alter them slightly without having to regenerate them, and out of laziness I enhanced `interdiff`. The first enhancement was to create a program which is now known as `filterdiff`, and this can be used to select the parts of a patch you want. For example, if you want just the makefile changes, you can use:

```
        filterdiff -i'*/Makefile*' fixes.patch
```

Or if you want everything except the makefile changes you can use `-x` (exclude) instead of `-i` (include). You can select individual patch hunks, and you can alter the file names in the patch. This is not the same as opening the patch in an editor and removing lines: `filterdiff` adjusts the offset lines (the ones that begin "@@") so that the resulting patch looks just as if it had been freshly generated.

After a few months, both of these programs were collected into a single package called `patchutils`. It now contains more than a dozen tools for operating on patches, most of which have been written because I have wanted to use them, or because someone else has suggested them. For example, a simple tool I wanted to use but that did not exist was one simply to list the files a patch file alters. So I made a small alteration to `filterdiff` and created `lsdiff` (list diff) from it, to do just that.

One tool resulting from a discussion with a co-worker is `rediff`. The idea is this: you have a patch file that you want to make a few adjustments to in an editor, but you want to make sure the offset lines are automatically adjusted for you. That is what `rediff` does. It reads the patch file you give it, and tries to work out what the offset lines ought to contain. It can do this because each line in a unified format patch file can be identified as part of a diff header, an offset line, some context, or an actual change, by the first character or few characters in the line. About the only thing that confuses it is if you remove the first line of context in a hunk, i.e. the line immediately after the "@@" line – so don't do that.

Some more tools developed from the desire to use them are `unwrapdiff` and `dehtmldiff`. The first of these tries to make a sensible patch file from an email that has been word-wrapped. The premise here is that the original patch contains accurate offset lines, but some of the lines of the hunks may need to be joined together, and it is `unwrapdiff`'s task to decide which ones they are. Since different email programs seem to have different rules for word-wrapping, it turns out that a rule-of-thumb approach produces the best results, but unfortunately accuracy cannot be guaranteed. Perhaps after all the best way of achieving the correct result is asking the sender to re-send the patch as an attachment thereby avoiding the word wrapping problem.

The other tool, `dehtmldiff`, was written when I wanted to use a patch that was only available through a web-based mailing list archive. Unfortunately this particular archive software did not allow raw access to the messages but instead converted them into HTML for display. So any "&" characters, for instance, became "&amp;". I wrote `dehtmldiff` as a quick way to convert the HTML bits back into ASCII text, and successfully used it to apply the patch I wanted.

Tools like that, with a simple task to achieve, are quite easy to write once I have some test cases to try them against. Very often they can build upon some code in `patchutils` which already exists to perform a similar task. Not very long ago our kernel RPM maintainer approached me with an idea for an additional tool: since the kernel RPM contains a great number of patches, he was after a tool that could re-arrange the order they apply in. That way new patches could be created after all the existing patches have been applied (easy, since RPM applies them for you), and then moved to be with all the other patches in that general area – for instance, driver patches. I agreed to take a look at it.

I ended up with a design for a tool, `flipdiff`, to just swap the order of two patches that apply one after the other. I soon realised the whole idea of doing this kind of operation was riddled with difficulties, and whenever I tried to think of a new design for the tool my brain got tied up in knots. However, after discussing it with my co-worker we agreed upon something that would be useful in most common situations, and so `flipdiff` was finally written. That proved to be a hard program to write, and I hope I never have to change it.

By now I think you will have some idea of how I spend a lot of my work time. Bugzilla is a useful mechanism for reporting bugs, and sometimes for finding work-arounds for known bugs. If you know programming you now also know how to use patches if you didn't before, so if you find a bug and want to report it, have a go at fixing the problem and attaching a patch to the bug report in Bugzilla.

If you already knew how to use patches but had not heard of `patchutils`, I hope this brief introduction has been useful. If you think a tool is missing, let me know and (so long as it is easier than `flipdiff` was!) I'll be happy to consider writing it.

Tim Waugh <tim@cyberelk.net>