A gentle introduction to quilt

Or, patch management for software.

Table of Contents

Speaker Intro

Hi, I'm bjb. I've been programming in C on unix-y operating systems for about 25 years.

Motivation and topic intro

People collaborating on a project must edit the same set of source files. When one person commits some changes, then the other people must rebase their own changes on the new version of the shared files before they can push their own changes.

One might want to fine-tune one's changes before submitting them. A minor fix for some old typo should not be in the same patch as a new feature; a comment correction should also be in its own patch. Especially, two new features and some bug fixes should not be all smushed together in one patch. Each feature should be in its own patch (or patch series), and each bug fix should also be in its own patch. This allows others to be able to review the code easily, and even lets others pick and choose which patches they want to apply. It becomes a chore to manage all these patches. That's where quilt comes in.

Sadly, I hadn't learned quilt till this weekend ... well one way to ensure I learn it fairly well is to write a HPR episode about it! Here goes.

I have written this episode to be understandable by anyone - you do not have to be a coder. You could use this tool to keep track of any plain-text files - recipes, todo lists, html, hpr show notes, poetry, what-have-you.

Introduction

First let's describe what a patch is. No, first let's describe what source code looks like. Source code is a plain text file full of computer instructions. It is a plain text file, as opposed to a word processing file. Plain text files do not have any formatting codes or styles (such as which font should be used, or what colour, etc) in them. They just contain the characters that make up words of the content.

A key feature of these source code files is that a new section of the file starts on a new line. The source code is almost never "reflowed" like prose might be. It is sort of like poetry - the more formal poetry, not prose poetry. There are a lot of really small sections in source code files (called "statements" and "expressions"). Most of these sections fit on one line. This is useful for the tools we are going to discuss because when one line changes, it does not affect the following lines as it might when text is reflowed after a change.

People have been coding with plain text files in various languages for decades. Thus a large set of tooling has grown around this format. One of those tools is called "diff" and another one is called "patch".

Diff is a way to compare two text files. Typically it would be used to compare the "before" and "after" of a source code file undergoing changes. So you could find out what was done to the source code file by running diff on the before and after versions of that file.

A diff file is a series of excerpts from the original and changed files. There are various kinds of diffs. Some of them show only the changed lines. Some of them show a few lines before and after the changed lines in addition to the changed lines themselves. That second kind is called a "context diff" and helps the automated machinery (and humans too) find the correct part of the file to which the change must be applied.

By default there are 3 lines of context before and after the changed lines.

The changed part is represented by including the old AND new code. In order to distinguish which lines are old and which are the replacements, all the lines (context lines, removed lines and added lines) are shifted over to the right by one character. The context lines start with a space in the extra left-most character, the original removed lines have a minus sign in the left-most character and the new added lines have a plus sign.

Thus if any character on a line in the source file has changed, been added or removed, then the whole line will be replaced with a new line in the new file. The diff will show both the removed line and the new one.

The patch utility takes the "diff" output and applies it to the original file to produce the later version of that file. You can apply it in reverse mode to the later version to get the original version. So patch is also a really useful program, and these two tools, diff and patch, are the basis of most of the version control systems out there. It is the existence of these text-based diff and patch tools that makes revision control systems work really well on plain-text files that are naturally structured in a line-by-line format.

A note about terminology: the diff program produces a diff. This diff is also called a patch. The patch program takes the diff (aka patch) and applies it to the original file to produce the changed file.

So if you have a timeline of adding a few features and making a few fixes on a code-base, it can be fully described by the original file plus a set of patches that had been produced with diff. You can get the final source code by taking the original file, applying the patches one by one, and voila, the final version of the file has been recreated.

Now we know enough to give a concise description of quilt:

Quilt lets you work with patches, creating them, applying them, un-applying them, and moving some things from one patch to another with a minimum of effort.

How to use quilt

Now a tutorial on how to get started using quilt.

This tutorial will start with a buggy program, create a few bad patches, and fix them up into good patches. I make no claims as to the quality of the final code though. The reason for starting with bad code and patches is to illustrate how to use quilt.

Starting to use quilt on a project

To start using quilt, create a directory called "patches" at the top of your code or just above.

```
$ mkdir patches
```

If you don't do this, quilt will create it for you. However, first it will look for a directory called "patches" in the current working directory, its parent, and all the way up ... if it finds one, it will use it. If not, it will create one in the current directory.

So, to keep it from finding some unrelated directory with the name "patches", just create a patches directory yourself in the right place.

Quilt first patch, including a new file!

You must tell quilt before you make any changes to your source code. Then it can store the original versions of the files that will change, so it can produce the diffs that will become that patch once you change the files.

Create a directory called example, and create a file in it like this, called hello.c (don't fix the errors):

```
#include "stdio.h"

int main (int argc, char *argv[], char *env[])
{
    print ("Hello, world!\n")
    return 0;
}
```

Now create a new patch - that is, give it a name - before you change any code. This will create (or find) a couple of directories, "patches" and ".pc", and populate them with some files to start.

```
$ quilt new fix-typo
```

And now you can fix the typo and generate the patch. First start by telling quilt that you want hello.c to be in the patch. Quilt saves a copy of it aside for comparing with the later versions:

```
$ quilt add hello.c
```

You can get quilt to tell you what files it knows about:

```
$ quilt files
```

Edit the file - add a semicolon at the end of the print line, and change the double-quotes on the #include line to angle brackets:

```
#include <stdio.h>
print ("Hello, world!\n");
```

Save the file and exit the editor. Next generate the patch:

```
$ quilt refresh
```

The oddly named "refresh" command creates the patch itself. It is called "refresh" because it can also be used to update the patch.

Now you can see the current set of patches by giving the command:

```
$ quilt series
```

The single patch is called fix-typo, and its name in the list is coloured brownish. That is because it is the "current" patch, and it is the one that will be updated if you "quilt refresh" again with more changes.

One thing I did not find in the quilt documentation is how to add a new file. When adding a new file, there is no existing file that you can name in the quilt add command. Of course, the very first patch I wanted to manage with quilt, I had introduced a new file. It turns out that the quilt edit command can be used to add a file to the patch, even if the file does not yet exist:

```
$ quilt edit header.h
```

Add content to header.h (see below) using the plain-text editor that quilt has started up for you. Save the file.

```
#ifndef HEADER_HH__
#define HEADER_HH__

#define NAME "bjb"

#endif
```

Regenerate the patch with the new changes:

```
$ quilt refresh
```

Now you can list the patch series again with quilt series. So far there is one patch. You can see what the patch consists of with the

```
$ quilt diff
```

command.

```
$ quilt diff
Index: hello/hello.c
===================================================================
--- hello.orig/hello.c
+++ hello/hello.c
@@ -1,8 +1,8 @@
-#include "stdio.h"
+#include <stdio.h>

 int main (int argc, char *argv[], *env[])
 {
-    print ("Hello, world!\n")
+    print ("Hello, world!\n");
     return 0;
 }

Index: hello/header.h
===================================================================
--- /dev/null
+++ hello/header.h
@@ -0,0 +1,7 @@
+#ifndef HEADER_HH__
+#define HEADER_HH__
+
+#define NAME "bjb
+
+#endif
+
$
```

Quilt second patch

Now it is time to make a second patch. First we tell quilt we are moving to a new patch:

```
$ quilt new prototype
$ quilt edit header.h
```

Edit this file again - add a function prototype.

```
int do_output(const char *name);
```

Create the patch and look at the list of patches:

```
$ quilt refresh
$ quilt series
```

Now when we give the quilt series command, we see two patches. The first one is green, meaning it has been applied, and the second one is brown, meaning this is the one that quilt refresh will change if you call it.

Again you can see what latest diff looks like by giving the quilt diff command.

```
$ quilt diff
```

Now let's unapply the latest diff:

```
$ quilt pop
$ quilt series
```

We see that the list of patches has the same patches in it, but now the second patch is white (meaning unapplied) and the first patch is brown (meaning it is the one that would change if we edited a file and typed quilt refresh.

```
$ quilt files
```

That first patch has two files in it, hello.c and header.h.

Now unapply the first diff:

```
$ quilt pop
$ quilt series
```

Both patches are listed, and both are shown as white.

We can see what files quilt knows about before any patches are applied:

```
$ quilt files
```

No files.

Apply all the patches at once:

```
$ quilt push -a
$ quilt series
```

And look at what files quilt knows about:

```
$ quilt files
```

Now quilt reports on only one file, while in the first patch it knew about two files. You must be careful to "add" each file to each patch, or it will not put the changes in those files into the patch. Luckily, quilt edit will put the files in the patch for you, so if you always start your editor with quilt edit fname, then you will have your changed files added to your patches without having to take any other action. But, if you are adding an existing file to the patch, you can add it without having to open your editor with the quilt add command:

```
$ quilt add fname
```

In order to avoid forgetting to add a file in a patch as I was editing, I just added all the files in the directory each time I created a new patch, whether I edited them or not.

Split a patch in two parts

We are going to split the first patch in two parts. We had fixed a typo and added a new file in one patch. They should be two separate patches.

First make the first patch current:

```
$ quilt pop
```

Then make a copy of that patch:

```
$ quilt fork
```

This makes a copy of the first patch called fix-typo-2. But, it removes the first patch fix-typo and puts fix-typo-2 in the series. We need to put the first patch back, and then edit each of the two fix-typo patches so each one contains one part of the original patch.

```
# edit patches/series file and put the first patch back
# The file should contain:

fix-typo
```

```
fix-typo-2
prototype
```

Now edit the first patch using a plain-text editor. It is in patches/fix-typo. Remove the part about the new file, header.h. It should now look like:

```
Index: hello/hello.c
===================================================================
--- hello.orig/hello.c
+++ hello/hello.c
@@ -1,4 +1,4 @@
-#include "stdio.h"
+#include <stdio.h>

 int main (int argc, char *argv[], *env[])
 {
```

Save this file. Now edit the second patch patches/fix-typo-2 using a plain-text editor. Remove the part about the file hello.c. It should now look like:

```
Index: hello/header.h
===================================================================
--- /dev/null
+++ hello/header.h
@@ -0,0 +1,7 @@
+#ifndef HEADER_HH__
+#define HEADER_HH__
+
+#define NAME "bjb
+
+#endif
+
```

If you give a quilt series command now, you will see that fix-typo-2 is the current patch and quilt thinks fix-typo has been applied.

We have to fix up quilts idea of reality.

Pop the current patch. Things have changed under quilts feet so we have to force this with the -f option:

```
$ quilt pop -f
```

Now, because quilt thought the original state of fix-typo-2 is the unchanged file, quilt shows the series as being completely un-applied.

```
$ quilt series
patches/fix-typo
patches/fix-typo-2
patches/prototype
```

Now we can push the patches:

```
$ quilt push -a
```

Rename a patch

Here we rename a patch from fix-typo-2 to add-header. The quilt rename command acts on the current patch, so make fix-typo-2 current first:

```
$ quilt pop fix-typo-2
$ quilt rename add-header
$ quilt series
$ quilt push -a
```

Reorder the patch series

We will make a new patch, then move it earlier in the series:

First make the new patch:

```
$ quilt new printf
$ quilt edit hello.c
```

And change the print statement to:

```
printf("Hello, world!\n");
```

Save the patch:

```
$ quilt refresh
```

Now to demonstrate the reordering.

Unapply all the patches, edit the patches series file patches/series so the patches are in the order you like, and then re-apply the patches. If you are lucky, they will re-apply with no conflicts.

```
$ quilt pop -a
$ vi patches/series
# move "printf" between fix-typo and add-header.
# now all the bug-fixes are at the beginning of the series
$ quilt push -a
```

Merge two patches into one

Make another new patch:

```
$ quilt new output-function
$ quilt edit hello.c
```

Change the c file to this:

```
#include <stdio.h>

int do_output(const char *name)
{
    return printf("Hello, %s!\n", name);
}
```

```
int main (int argc, char *argv[], char *env[])
{
    /* ignoring the return code for do_output */
    do_output(NAME);
    return 0;
}
```

```
$ quilt refresh
```

Now, to merge two patches into one:

```
$ quilt pop prototype
$ quilt fold < patches/output-function
```

We have merged the prototype and output-function patches, because they describe a related change.

Save the patch.

```
$ quilt refresh
```

Throw away a patch

Now we no longer need the last patch, output-function, as it has been included into the prototype patch. But we might want to rename the prototype patch.

```
$ quilt delete output-function
# we have to clean up a bit for quilt or the rename won't work
$ rm patches/output-function
$ quilt rename output-function
```

Deleting will not work on a patch that has been applied before the current patch.

You are ready to contribute your patches ... go forth and code.

Summary

We have seen that quilt can help you manage your contributions to any project that is written in plain-text files. It can generate patch files (usually needed for contributions to open source projects) and can help you manage and update them as the tip of the development branch moves forward with other peoples' contributions.

To use quilt successfully, you need to remember to add files to each patch with quilt add/or quilt edit before editing, and to generate the patch with quilt refresh once all the editing of each patch is done. The rest is easy.

Commands that edit the patches:

```
$ quilt new patch-name
$ quilt add fname
$ quilt edit fname
$ quilt refresh
$ quilt pop [-a]
$ quilt push [-a]
$ quilt rename [-P oldname] newname
$ quilt delete [-P patchname]
$ quilt fold < patch_to_merge
```

Commands that view the state of the patches:

```
$ quilt series
$ quilt files
$ quilt diff [-P patchname]
$ quilt graph [--all]
$ quilt patches fname
$ quilt annotate fname
$ quilt applied
$ quilt unapplied
```

HPR exhortation

You've been listening to Hacker Public Radio. Anyone can make a show - if I can do it, so can you.