A Small Matter of Programming

Tuesday, September 14, 2010

Painless Merge Conflict Resolution in Git

In my line of work, we often deal with a multitude of git branches - whether we're keeping branches for maintenance releases, supporting deprecated APIs for certain customers, or working with experimental features. Although git's model entices you to create more and more branches, it brings the burden of keeping them up to date through periodic merging of branches.

While merges are important for keeping code up to date, errors in merge commits are more common and more impactful than in normal commits. Firstly, merges have multiple parents, which makes it very hard to see, from history, what the programmer actually did to resolve merge conflicts. Secondly, reverting a bad merge can turn into a headache in itself. Thirdly, a large proportion of merge conflicts happen when dealing with someone else's code because by nature, branching is a multiplayer game. In essense, merge errors are easy to make, hard to fix, and hard to find, so it really does pay to improve the merge process.

Yet, I have found that the tools and interfaces available for performing merges do not equip programmers sufficiently to do them effectively. Often, a programmer will simply run `git merge` and hope that git will take care of the large majority of the hunks. Of the hunks that conflict, the usual merge strategy of the human at the helm is to use the surrounding context to roughly guess what the intended program is supposed to look like.

This article will hopefully demonstrate that there can be a much more measured process to the resolution of merge conflicts which takes the guesswork out of this risky operation.

Roses are Blue

Let's say your team has been assigned the task of writing a repository of poems (what a nightmare!), and your nightmare has just been compounded with the task of merging the recent fixes from the `master` branch into the `beta` branch. So you switch to the `beta` branch and run:

```
$ git merge master
Auto-merging roses.txt
CONFLICT (content): Merge conflict in roses.txt
Automatic merge failed; fix conflicts and then commit the result.
```

A merge conflict. So you have a look at the file in question to see what the conflict is:

```
$ cat roses.txt
<<<<<<< HEAD
roses are #ff0000
violets are #0000ff
all my base
are belong to you
=======
Roses are red,
Violets are blue,
All of my base
Are belong to you.
>>>>>>> master
```

(Listing 1)

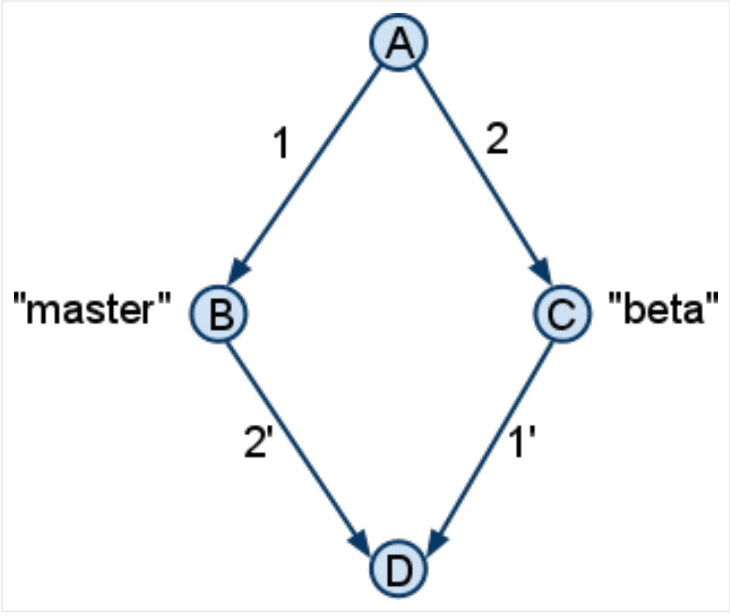
Great! The whole file conflicts. Which is the correct one to take? They both look equally valid. The top one is written in hacker style with HTML-style colour codes and the hardcore, minimalist aura of pure lowercase letters. The bottom one has more correct punctuation and capitalisation, but also seems a bit boring, comparatively.

If this was your project, you could just pick one and be done with it. But the problem is, this isn't your poem, you've never seen the poem before, you weren't responsible for writing or editing it, and basically, you don't want to risk destroying someone's hard work. You have only been assigned the task of merging these branches. So what do you do?

Back to Base

The trick is that Listing 1 doesn't give you all the information you need to complete the merge correctly. In fact, there are four important pieces of information involved in a merge, three of which are necessary to resolve the merge. In this case, git has only given you two of the required pieces.

The following diagram illustrates the four states:



States B and C correspond to the heads of the `master` and `beta` branches respectively, and these are the fragments that git has shown in the failed merge above. D is the state you want to generate (in most cases, git will compute D automatically). State A at the top represents the merge base of the `master` and `beta` branches. The merge base is the last common ancestor of the two branches, and for now, we will assume it is unique. As we will see later, this state plays a crucial role in resolving the merges. In the diagram, I have also marked out the deltas 1 and 2, which represent the change between state A and B, and A and C respectively. Given states A, B and C, deltas 1 and 2 can be derived. Note that deltas 1 and 2 may represent more than 1 commit. But for our purposes, we can just treat them as monolithic deltas.

To understand how to get to D, you need to understand what the merge operation is trying to achieve. D should represent the combination of changes made in the `master` branch and the changes made in the `beta` branch. That is, it is the combination of deltas 1 and 2. The idea is simple - and most of the time, it's so simple that the VCS can combine the deltas

with no human intervention. It is only when the deltas affect proximal parts of the file that it will ask for human assistance. Your job as the human assistant is to continue the job of the machine, by determining what the intents of deltas 1 and 2 are and implementing both intents to produce D, the merged result.

Identifying the Differences

To find the changes that went into each branch, we need to know what the merge base (state A) looks like. The most basic mechanism by which git allows us to do so is to set the `merge.conflictstyle` option to `diff3` by executing the command:

```
$ git config merge.conflictstyle diff3
```

After setting this, retry the merge (`git reset --hard; git merge master`) and examine the conflicting file again:

```
$ cat roses.txt
<<<<<<< HEAD
roses are #ff0000
violets are #0000ff
all my base
are belong to you
|||||||
roses are red
violets are blue
all my base
are belong to you
=======
Roses are red,
Violets are blue,
All of my base
Are belong to you.
>>>>>>> master
```

There is now a third fragment shown in the middle, which corresponds to the merge base. At this point, we can do an eye-diff to find that in HEAD (the `beta` branch) the colour names were changed to the nerdy HTML codes, while the `master` branch introduced capitalisation and punctuation. Based on this knowledge, we now know that the result should include

captalisation, punctuation and HTML-style colours.

It is at this point that this article could end, having reached a solution. However, there is a better way.

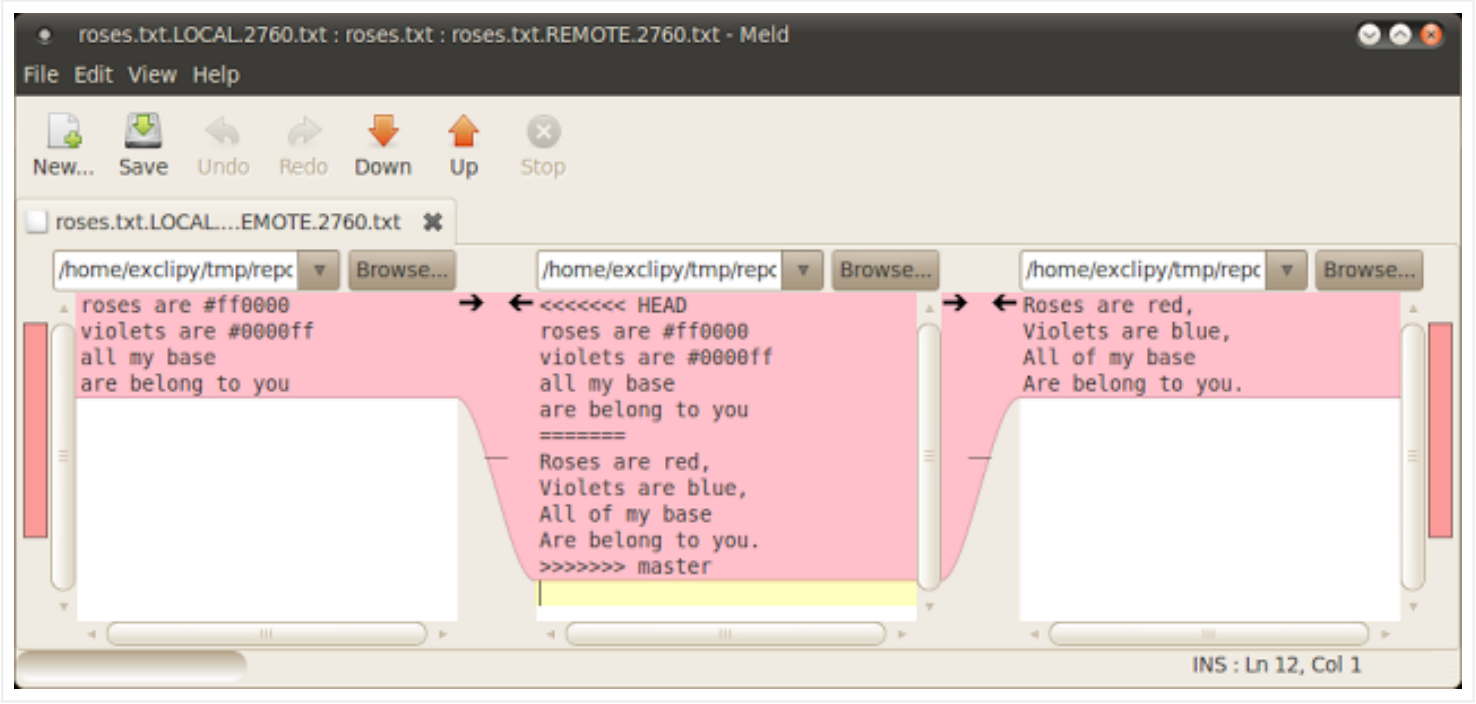
GUI Merging

While staring at plain-text merge output can get the job done in the simple cases, there are times when the conflicts are more drastic and some more sophisticated, graphical tooling can help. My diff and merge tool of choice is a simple Python program called meld, but anything that visualises merges with three panes will do.

To use it for merge resolution, have it installed and, while in an unresolved-merge in git, type:

```
$ git mergetool
```

It will ask you to choose the merge program, so type `meld` and hit Enter. Here is what the window might look like (assuming `merge.conflictstyle` has not been set):



While it is now presented in side-by-side view, this doesn't actually present any more helpful

information than Listing 1. That is because, as before, we are not given the state of the file from the merge base. We are presented with `roses.txt.LOCAL.2760.txt` on the left and `roses.txt.REMOTE.2760.txt` on the right, and the file in the middle is a failed merge. We are given states B, C and an attempt at D, but state A is missing...

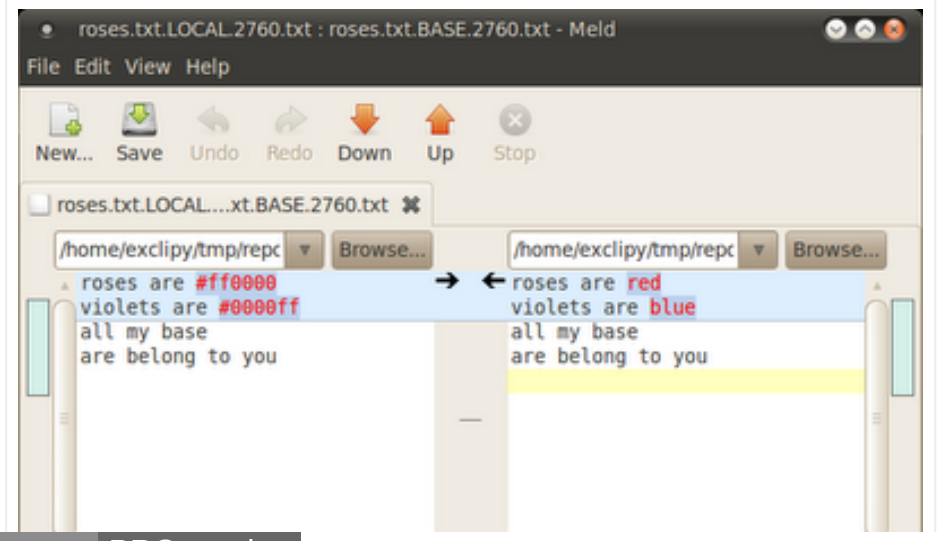
Or is it? If you fire up a new terminal (without exiting meld) and look in the directory, you'll find some extra files:

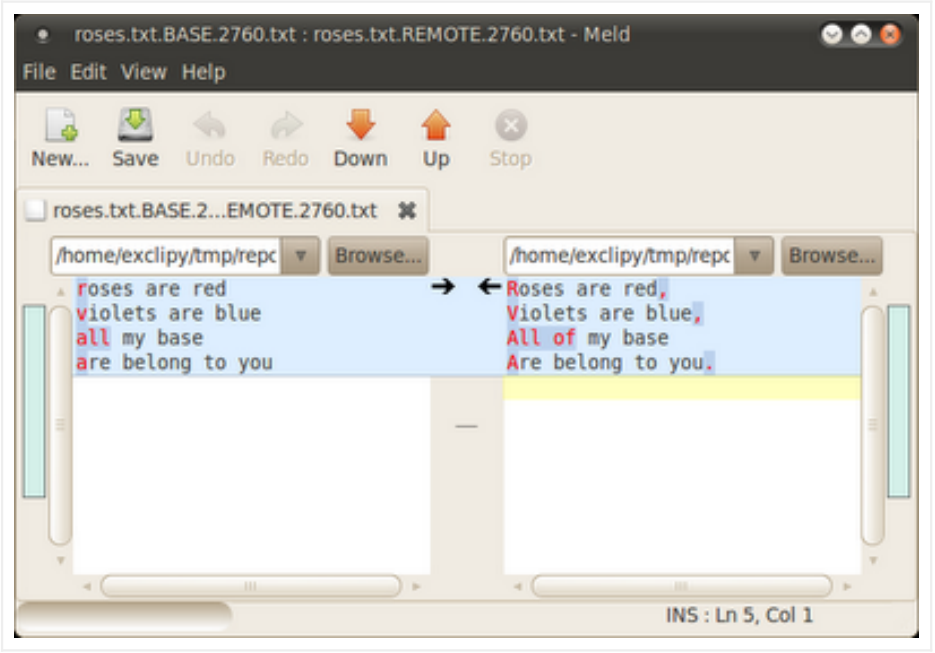
```
$ ls -1
roses.txt
roses.txt.BACKUP.2760.txt
roses.txt.BASE.2760.txt
roses.txt.LOCAL.2760.txt
roses.txt.REMOTE.2760.txt
```

The interesting file here is `roses.txt.BASE.2760.txt`. It is the merge base that we are looking for! The trick now is to find the changes introduced by the `master` and `beta` branches, in relation to the BASE. We can do that with two separate invocations of meld:

```
$ meld roses.txt.LOCAL.2760.txt roses.txt.BASE.2760 &
$ meld roses.txt.BASE.2760 roses.txt.REMOTE.2760.txt &
```

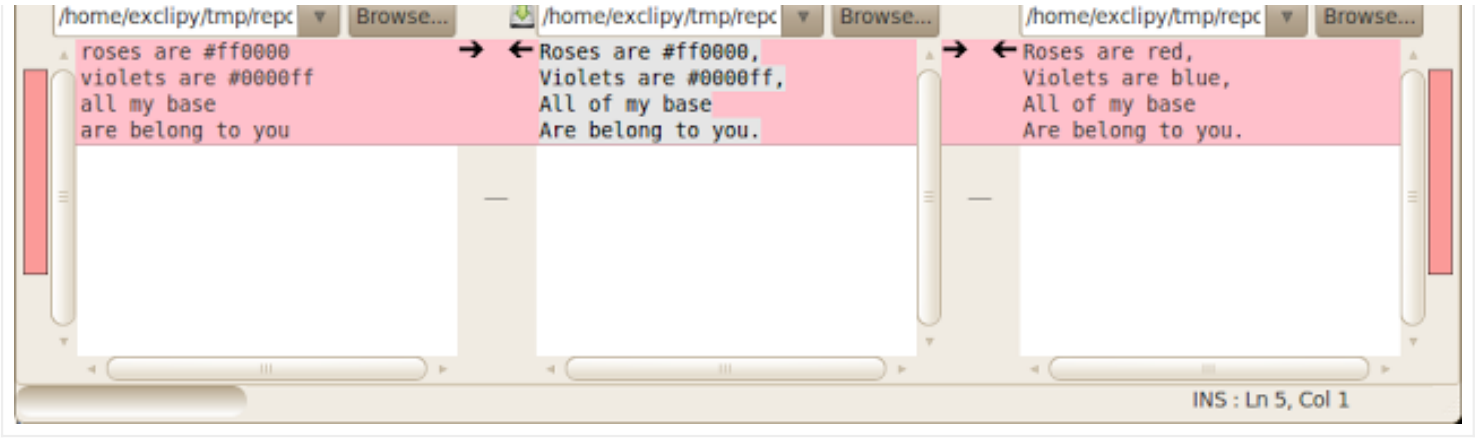
(Some might find it makes more sense to swap the order of the arguments in the first command so that the original file is on the left in both cases, but I like to keep the order the same as in the three-paned window.) The two resultant windows are:





Reading the first window from right to left and the second window from left to right, it is now clear as day what changes occurred in each branch. Because meld has even presented a diff within each line, it's now much easier to spot the differences and with this view, it's likely that you'll find changes that you would not have noticed had you just stared at the text files (eg. did you notice before the addition of the word "of" in the `REMOTE` branch?).

Armed with this knowledge, we can now go back to the three-paned view and make the changes. My edit strategy for performing the manual merge is to take the text from the branch that contained the more extensive changes (in this case, the `master/REMOTE` version), and then insert the edits made in the other branch. Here is the result for this little example:



All Together Now

Hopefully, you'll find the three-window merge conflict resolution strategy demonstrated above to be as useful as I have. But you might, like me, also find it inconvenient to have to manually open up new instances of meld to see the two diff views. The solution to this is to configure git to open up all three windows when the mergetool is requested. To do this, paste the following into the an executable script in the system `PATH` (eg. `$HOME/bin/gitmerge`):

```
#!/bin/sh
meld $2 $1 &
sleep 0.5
meld $1 $3 &
sleep 0.5
meld $2 $4 $3
```

And add the following to your `~/.gitconfig` file:

```
[merge]
    tool = mymeld
[mergetool "mymeld"]
    cmd = $HOME/bin/gitmerge $BASE $LOCAL $REMOTE $MERGED
```

Now, the next time you type git mergetool, for each file to be resolved, you can choose the mymeld option to have all three windows opened - one for the diff from `BASE` to `LOCAL`, one for the diff from `BASE` to `REMOTE` and one for the three-pane view.

After you get used to using these three views to do the merge conflict resolution, you'll find that the process becomes very methodical and mechanical. In most cases, there is no longer the need to read and understand the chunks of code from each branch to understand which elements from which are correct. You'll find that you're employing much less guesswork and you'll be much more confident that what you're commiting is *correct*. And because of this confidence, you might even find that merging becomes *fun*, not dreaded.

Posted by Kevin Wu Won at 13:21

+46 Recommend this on Google

38 comments

Google+

Add a comment

Top comments

Robert Metzger 2 weeks ago - Shared publicly

Thanks alot

+1 +1 · Reply

Deepak Yadav 2 weeks ago - Shared publicly

very helpful

+1 · Reply

Boris Schulz via Google+ 1 year ago - Shared publicly

git merge für Fortgeschrittene

Translate

+1 +1 · Reply

Mahmoud Saada 1 month ago - Shared publicly

Freaking EPIC!!!

+2 +1 · Reply

View all 5 replies

Mahmoud Saada 1 month ago

looooool Google+ is so much better than FB

Ridwan Maassarani 1 month ago

I know !! I am tryin to get ppl to use it

Сергей Горюгин 2 months ago - Shared publicly

The post is awesome! Thnx!

+2 +1 · Reply

Alessandro Re 5 months ago - Shared publicly

Thanks for this useful post :) But, may I suggest the diffuse (http://diffuse.sourceforge.net/) tool, which seems a bit better (to me) than having multiple meld windows opened?

+2 +1 · Reply

Сергей Горюгин 2 months ago
it looks like the diffuse is more ugly than meld

S Ma 3 months ago - Shared publicly

Super! Thank you Kevin for sharing it!

+1 · Reply

Andrea Piccinin via Google+ 5 months ago - Shared publicly
WOW

Alessandro Re originally shared this
Thanks for this useful post :) But, may I suggest the diffuse (http://diffuse.sourceforge.net/) tool, which seems a bit better (to me) than having multiple meld windows opened?

+1 · Reply

Lolla Polla 2 months ago +1
giuro che chi ha inventato git è un masochista intelligentissimo...
Translate

Emmanuel Frenod shared this via Google+ 2 months ago - Shared publicly

+1 · Reply

Subhojit Paul 6 months ago

Thank you. This helped me to understand what file.REMOTE.*, file.BASE.*, file.LOCAL.* actually meant.

Kevin 8 months ago

The centre pane in the 3-pane view is not BASE. It is where you make the edits to form the final result (state D in the diamond). After doing the merge, saving, and exiting Meld (for all conflicts), you must finish it off with a "git commit".

102735678183111951738 9 months ago

Kevin, What should actually happen when you have Meld open with the 3 version? (LOCAL, BASE, REMOTE?) From where to where do you move the changes? Which of these 3 will be the Solution ? What other actions do you need to take after merging and exiting Meld? Thanks!

Eric Drechsel 11 months ago

Thanks Tomek, the 4-argument mode seems great. The center pane uses the auto-merged text except where there is a conflict (there it uses the base text).

Tomek Bury 1 year ago

Meld has hidden 3-way merge feature activated by passing the 4th parameter: [merge] tool = mymeld conflictstyle = diff3 [mergetool "mymeld"] cmd = meld --diff $BASE $LOCAL --diff $BASE $REMOTE --diff $LOCAL $BASE $REMOTE $MERGED This way you don't need a shell

Expand this comment »

rushil 1 year ago

This is such an awesome post. Thanks a lot.

Michael Schober 1 year ago

Kevin, Great post. You completely clarified merging for me. I installed meld and am using your script + smancill's tabbing. Thanks guys

smancill 1 year ago

Hi Great post. You can use tabs too: #!/bin/sh meld --diff $2 $1 --diff $1 $3 --diff $2 $4 $3

Kevin 1 year ago

Awesome stuff rockaholic - thanks!

rockaholic 1 year ago

Dude - your post kicks ass! Great merging techniques, I'm totally adopting it. Check out my blog post on git command line tricks, you might run in to something new :-)
http://rubyglazed.com/post/15772234418/git-ify-your-command-line

TTT 1 year ago

Using vimdiff as the git merge tool allows you to see all the four files you want out of the box.

Show more

Newer Post Home

Subscribe to: Post Comments (Atom)