Fix merge conflicts in Git?

Is there a good way to explain how to resolve merge conflicts in Git?

git | git-merge | merge-conflict-resolution | git-conflict-resolution

edited Aug 1 at 0:57 Cupcake **24.4k** 9 54 85

asked Oct 2 '08 at 11:31 Spoike **33.3k** 30 101 125

6 The following blog post seems to give a very good example on how to handle merge conflict with Git that should get you going in the right direction. Handling and Avoiding Conflicts in Git – mwilliams Oct 2 '08 at 11:40

add a comment

12 Answers

Try: `git mergetool`

It opens a GUI that steps you through each conflict, and you get to choose how to merge. Sometimes it requires a bit of hand editing afterwards, but usually it's enough by itself. It is much better than doing the whole thing by hand certainly.

By @JoshGlover

Well, it doesn't necessarily open a GUI unless you install one. Running `git mergetool` for me resulted in `vimdiff` being used. You can install one of the following tools to use it instead: `meld opendiff kdiff3 tkdiff xxdiff tortoisemerge gvimdiff diffuse ecmerge p4merge araxis vimdiff emerge`

`p4merge` is working great for me. I've not tried other tools.

edited Jul 23 at 17:41 Peter Mortensen **7,541** 8 53 85

answered Oct 2 '08 at 17:50 Peter Burns **18.4k** 5 24 42

22 FYI you can use `git mergetool -y` to save a few keystrokes if you're merging a lot of files at once. – davr Jun 17 '10 at 23:32

242 Well, it doesn't necessarily open a GUI unless you install one. Running `git mergetool` for me resulted in `vimdiff` being used. You can install one of the following tools to use it instead: `meld opendiff kdiff3 tkdiff xxdiff tortoisemerge gvimdiff diffuse ecmerge p4merge araxis vimdiff emerge` . – Josh Glover May 11 '11 at 14:00

17 Good point Josh. On ubuntu I've had the best luck with meld, its three way merge display isn't bad. On OSX git chose a nice default. – Peter Burns May 24 '11 at 5:08

7 This opened KDiff3. Which I have absolutely no clue how to use. – David Murdoch Jun 10 '11 at 18:46

10 I don't understand why this answer received so many upvotes, it isn't really very helpful as it only contains this one command and absolutely no explanation how to use it. As others said, it opened a vimdiff and even if I know how to use vim (switch the windows at least or close them) I don't even know what each window represent neither how to compare or accept the changes. It's nice to know there is such a command, but

with no explanation of how to use it or install other 3rd tools, it's useless answer – Petr Mar 10 at 13:48

show **7** more comments

Here's a probable use-case, from the top:

You're going to pull some changes, but oops, you're not up to date:

```
git fetch origin
git pull origin master

From ssh://gitosis@example.com:22/projectname
 * branch            master     -> FETCH_HEAD
Updating a030c3a..ee25213
error: Entry 'filename.c' not uptodate. Cannot merge.
```

So you get up-to-date and try again, but have a conflict:

```
git add filename.c
git commit -m "made some wild and crazy changes"
git pull origin master

From ssh://gitosis@example.com:22/projectname
 * branch            master     -> FETCH_HEAD
Auto-merging filename.c
CONFLICT (content): Merge conflict in filename.c
Automatic merge failed; fix conflicts and then commit the result.
```

So you decide to take a look at the changes:

```
git mergetool
```

Oh me, oh my, upstream changed some things, but just to use my changes...no...their changes...

```
git checkout --ours filename.c
git checkout --theirs filename.c
git add filename.c
git commit -m "using theirs"
```

And then we try a final time

```
git pull origin master

From ssh://gitosis@example.com:22/projectname
 * branch            master     -> FETCH_HEAD
Already up-to-date.
```

Ta-da!

edited Apr 16 at 23:13
Cupcake
24.4k 9 54 85

answered Aug 4 '10 at 17:04
CoolAJ86
20.2k 6 30 46

10 This was super helpful because I had a lot of merge errors with binary files (art assets) and merging those seems to always fail, so I need to overwrite it with the new file always and not "merge" – petrocket Jun 8 '11 at 17:39

78 Careful! The meaning of --ours and --theirs is reversed. --ours == the remote. --theirs == local. See `git merge --help` – mmell Mar 4 '13 at 22:56

27 In my case, I confirm that --theirs = remote repository, --ours = my own local repository. It is the opposite of @mmell comments. – Aryo Jun 22 '13 at 12:59

9 @mmell Only on a rebase, apparently. See `this question` – Navin Nov 10 '13 at 6:19

7 **Guys, "ours" and "theirs" is relative to whether or not you are merging or rebasing.** If you're *merging*, then "ours" means the branch you're merging into, and "theirs" is the branch you're merging in. When you're *rebasing*, then "ours" means the commits you're rebasing onto, while "theirs" refers to the commits that you want to rebase. – Cupcake May 26 at 4:27

show **6** more comments

I find merge tools rarely help me understand the conflict or the resolution. I'm usually more successful looking at the conflict markers in a text editor and using git log as a supplement.

Here are a few tips:

Tip One

The best thing I have found is to use the "diff3" merge conflict style:

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

This produces conflict markers like this:

```
<<<<<<<
Changes made on the branch that is being merged into. In most cases,
this is the branch that I have currently checked out (i.e. HEAD).
|||||||
The common ancestor version.
=======
Changes made on the branch that is being merged in. This is often a
feature/topic branch.
>>>>>>>
```

The middle section is what the common ancestor looked like. This is useful because you can compare it to the top and bottom versions to get a better sense of what was changed on each branch, which gives you a better idea for what the purpose of each change was.

If the conflict is only a few lines, this generally makes the conflict very obvious. (Knowing how to fix a conflict is very different; you need to be aware of what other people are working on. If you're confused, it's probably best to just call that person into your room so they can see what you're looking at.)

If the conflict is longer, then I will cut and paste each of the three sections into three separate files, such as "mine", "common" and "theirs".

Then I can run the following commands to see the two diff hunks that caused the conflict:

```
diff common mine
diff common theirs
```

This is not the same as using a merge tool, since a merge tool will include all of the non-conflicting diff hunks too. I find that to be distracting.

Tip Two

Somebody already mentioned this, but understanding the intention behind each diff hunk is generally very helpful for understanding where a conflict came from and how to handle it.

```
git log --merge -p <name of file>
```

This shows all of the commits that touched that file in between the common ancestor and the two heads you are merging. (So it doesn't include commits that already exist in both branches before merging.) This helps you ignore diff hunks that clearly are not a factor in your current conflict.

Tip Three

Verify your changes with automated tools.

If you have automated tests, run those. If you have a lint, run that. If it's a buildable project, then build it before you commit, etc. In all cases, you need to do a bit of testing to make sure your changes didn't break anything. (Heck, even a merge without conflicts can break working code.)

Tip Four

Plan ahead; communicate with co-workers.

Planning ahead and being aware of what others are working on can help prevent merge conflicts and/or help resolve them earlier -- while the details are still fresh in mind.

For example, if you know that you and another person are both working on different refactoring that will both affect the same set of files, you should talk to each other ahead of time and get a better sense for what types of changes each of you is making. You might save considerable time and effort if you conduct your planned changes serially rather than in parallel.

For major refactorings that cut across a large swath of code, you should strongly consider working serially: everybody stops working on that area of the code while one person performs the complete refactoring.

If you can't work serially (due to time pressure, maybe), then communicating about expected merge conflicts at least helps you solve the problems sooner while the details are still fresh in mind. For example, if a co-worker is making a disruptive series of commits over the course of a one-week period, you may choose to merge/rebase on that co-workers branch once or twice each day during that week. That way, if you do find merge/rebase conflicts, you can solve them more quickly than if you wait a few weeks to merge everything together in one big lump.

Tip Five

If you're unsure of a merge, don't force it.

Merging can feel overwhelming, especially when there are a lot of conflicting files and the conflict markers cover hundreds of lines. Often times when estimating software projects we don't include enough time for overhead items like handling a gnarly merge, so it feels like a real drag to spend several hours dissecting each conflict.

In the long run, planning ahead and being aware of what others are working on are the best tools for anticipating merge conflicts and prepare yourself to resolve them correctly in less time.

edited Jul 23 at 17:32
Peter Mortensen
7,541 8 53 85

answered Sep 28 '11 at 21:08
mehaase
7,067 1 26 34

The diff3 option is a great feature to have with merges. The only GUI I've come across that shows it is Perforce's `p4merge` , which can be installed and used separately from Perforce's other tools (which I've not used, but heard complaints about). – alxndr May 1 at 22:15

add a comment

---

1. Identify which files are in conflict (Git should tell you this).
2. Open each file and examine the diffs; Git demarcates them. Hopefully it will be obvious which version of each block to keep. You may need to discuss it with fellow developers who committed the code.
3. Once you've resolved the conflict in a file `git add the_file` .
4. Once you've resolved **all** conflicts, do `git rebase --continue` or whatever command Git said to do when you completed.

edited Aug 7 at 17:48
Cupcake
24.4k 9 54 85

answered Oct 2 '08 at 12:41
davetron5000
9,861 2 40 79

20 @Justin Think of Git as tracking *content* rather than tracking files. Then it's easy to see that the content you've updated *isn't* in the repository and needs to be added. This way of thinking also explains why Git doesn't track empty folders: Although they are technically files, there isn't any content to track. – Gareth Oct 12 '10 at 9:17

4 content is there, conflict occurs because there 2 version of content. Therefore "git add" does not sound correct. And it does not work (git add, git commit) if you want commit only that one file after conflict was resolved ("fatal: cannot do a partial commit during a merge.") – Dainius Sep 14 '11 at 9:19

Yes, technically, this answers the question which as asked, but is not a usable answer, in my opinion, sorry. What's the **point** of making one branch the same as another? Of course a merge will have conflicts.. – Thufir Aug 9 '12 at 5:56

2 Thulfir: who said anything about making one branch the same as another? There are different scenarios where you need to merge, without "making one branch the same as another". One is when you're done with a

development branch and want to incorporate its changes into the master branch; after this, the development branch can be deleted. Another one is when you want to rebase your development branch, in order to ease the eventual final merge into the master. – Teemu Leisti Sep 21 '12 at 8:50

2 @JustinGrant `git add` stages files in the index; it does **not** add anything to the repository. `git commit` adds things to the repository. This usage makes sense for merges -- the merge automatically stages all of the changes that can be merged automatically; it is your responsibility to merge the rest of the changes and add those to the index when you are done. – mehaase Oct 17 '12 at 15:13

add a comment

Check out the answers in Stack Overflow question *Aborting a merge in Git*, especially Charles Bailey's answer which shows how to view the different versions of the file with problems, for example,

```
# Common base version of the file.
git show :1:some_file.cpp

# 'Ours' version of the file.
git show :2:some_file.cpp

# 'Theirs' version of the file.
git show :3:some_file.cpp
```

edited Jul 23 at 17:34

Peter Mortensen
7,541 8 53 85

answered Oct 3 '08 at 15:15

Pat Notz
49.1k 18 60 76

add a comment

If you're making frequent small commits, then start by looking at the commit comments with `git log --merge` . Then `git diff` will show you the conflicts.

For conflicts that involve more than a few lines, it's easier to see what's going on in an external GUI tool. I like opendiff -- Git also supports vimdiff, gvimdiff, kdiff3, tkdiff, meld, xxdiff, emerge out of the box and you can install others: `git config merge.tool "your.tool"` will set your chosen tool and then `git mergetool` after a failed merge will show you the diffs in context.

Each time you edit a file to resolve a conflict, `git add filename` will update the index and your diff will no longer show it. When all the conflicts are handled and their files have been `git add` -ed, `git commit` will complete your merge.

edited Jul 23 at 17:43

Peter Mortensen
7,541 8 53 85

answered Oct 2 '08 at 16:11

Paul
13.1k 2 20 21

3 Using "git add" is the real trick here. You may not even want to commit (maybe you want to stash), but you have to do "git add" to complete the merge. I think mergetool does the add for you (although it isn't in the manpage), but if you do the merge manually, you need to use "git add" to complete it (even if you don't want to commit). – nobar Oct 25 '10 at 9:37

add a comment

See How Conflicts Are Presented or, in Git, the `git merge` documentation to understand what merge conflict markers are.

Also, the How to Resolve Conflicts section explains how to resolve the conflicts:

> After seeing a conflict, you can do two things:
>
> - Decide not to merge. The only clean-ups you need are to reset the index file to the `HEAD` commit to reverse 2. and to clean up working tree changes made by 2. and 3.; `git merge --abort` can be used for this.
> - Resolve the conflicts. Git will mark the conflicts in the working tree. Edit the files into shape and `git add` them to the index. Use `git commit` to seal the deal.
>
> You can work through the conflict with a number of tools:
>
> - Use a mergetool. `git mergetool` to launch a graphical mergetool which will work you through the merge.
> - Look at the diffs. `git diff` will show a three-way diff, highlighting changes from both the `HEAD`

and `MERGE_HEAD` versions.

- Look at the diffs from each branch. `git log --merge -p <path>` will show diffs first for the `HEAD` version and then the `MERGE_HEAD` version.
- Look at the originals. `git show :1:filename` shows the common ancestor, `git show :2:filename` shows the `HEAD` version, and `git show :3:filename` shows the `MERGE_HEAD` version.

You can also read about merge conflict markers and how to resolve them in the Pro Git book section Basic Merge Conflicts.

edited Jul 23 at 17:22
Peter Mortensen
7,541 8 53 85

answered Jul 14 '13 at 18:34
Cupcake
24.4k 9 54 85

add a comment

For Emacs users which want to resolve merge conflicts semi-manually:

```
git diff --name-status --diff-filter=U
```

shows all files which require conflict resolution.

Open each of those files one by one, or all at once by:

```
emacs $(git diff --name-only --diff-filter=U)
```

When visiting a buffer requiring edits in Emacs, type

```
ALT+x vc-resolve-conflicts
```

This will open three buffers (mine, theirs, and the output buffer). Navigate by pressing 'n' (next region), 'p' (prevision region). Press 'a' and 'b' to copy mine or theirs region to the output buffer, respectively. And/or edit the output buffer directly.

When finished: Press 'q'. Emacs asks you if you want to save this buffer: yes. After finishing a buffer mark it as resolved by running from the teriminal:

```
git add FILENAME
```

When finished with all buffers type

```
git commit
```

to finish the merge.

edited Jul 23 at 17:24
Peter Mortensen
7,541 8 53 85

answered Feb 22 '13 at 23:04
eci
626 4 13

add a comment

You could fix merge conflicts in a number of ways as other have detailed.

I think the real key is knowing how changes flow with local and remote repositories. The key to this is understanding tracking branches. I have found that I think of the tracking branch as the 'missing piece in the middle' between me my local, actual files directory and the remote defined as origin.

I've personally got into the habit of 2 things to help avoid this.

Instead of:

```
git add .
git commit -m"some msg"
```

Which has two drawbacks -

a) All new/changed files get added and that might include some unwanted changes.
b) You don't get to review the file list first.

So instead I do:

```
git add file,file2,file3...
git commit # Then type the files in the editor and save-quit.
```

This way you are more deliberate about which files get added and you also get to review the list and think a bit more while using the editor for the message. I find it also improves my commit messages when I use a full screen editor rather than the `-m` option.

[Update - as time has passed I've switched more to:

```
git status # Make sure I know whats going on
git add .
git commit # Then use the editor
```

]

Also (and more relevant to your situation), I try to avoid:

```
git pull
```

or

```
git pull origin master.
```

because pull implies a merge and if you have changes locally that you didn't want merged you can easily end up with merged code and/or merge conflicts for code that shouldn't have been merged.

Instead I try to do

```
git checkout master
git fetch
git rebase --hard origin/master # or whatever branch I want.
```

You may also find this helpful:

git branch, fork, fetch, merge, rebase and clone, what are the differences?

edited Dec 12 '13 at 12:22

answered Apr 19 '13 at 1:08

Michael Durrant
31.3k 21 102 193

add a comment

CoolAJ86's answer sums up pretty much everything. In case you have changes in both branches in the same piece of code you will have to do a manual merge. Open the file in conflict in any text editor and you should see following structure.

```
(Code not in Conflict)
>>>>>>>>>>>
(first alternative for conflict starts here)
Multiple code lines here
===========
(second alternative for conflict starts here)
Multiple code lines here too
<<<<<<<<<<<
(Code not in conflict here)
```

Choose one of the alternatives or a combination of both in a way that you want new code to be, while removing equal signs and angle brackets.

```
git commit -a -m "commit message"
git push origin master
```

edited Jul 23 at 17:17

Peter Mortensen
7,541 8 53 85

answered Jan 25 at 16:17

iankit
309 3 19

add a comment

---

if you want to merge from branch(test) to master, you can follow these steps:

Step1: go to the branch

```
git checkout test
```

Step2: `git pull --rebase origin master`

Step3: if there are some conflicts, go to these files to modify it.

Step4: add these changes

```
git add #your_changes_files
```

Step5: `git rebase --continue`

Step6: if there is still conflict, go back to Step3 again. If there is no conflict, do following: `git push origin +test`

Step7: and then there is no conflict between test and master. you can use merge directly.

answered Aug 18 at 19:42

Haimei

408 3 6

add a comment

```
git fetch origin/branch
git reset --hard orgin/branch
git merge master
```

answered Jun 10 at 8:11

trai bui

636 1 19

2 Does that sequence always help? Please explain what and how it does and how that code answers the question. – Oleg Estekhin Jun 10 at 8:32

add a comment

---

protected by Will Dec 17 '10 at 13:56

Thank you for your interest in this question. Because it has attracted low-quality answers, posting an answer now requires 10 reputation on this site.

Would you like to answer one of these unanswered questions instead?

Not the answer you're looking for? Browse other questions tagged git | git-merge | merge-conflict-resolution | git-conflict-resolution **or ask your own question.**