CollabNet

Deconstructing a Branch with Git

Blog Article Series

Table of Contents

Deconstructing a Branch with Git

Everyone knows — it's a basic Best Practice! — that you should do parallel development on parallelbranches, not cascaded branches: base them both on trunk or master or whatever your favorite version control system calls it. Don't branch one off the other, lest plans change, release schedules diverge, and you suddenly have to disentangle them.

Right? You knew that. I knew that. Everyone knows that.

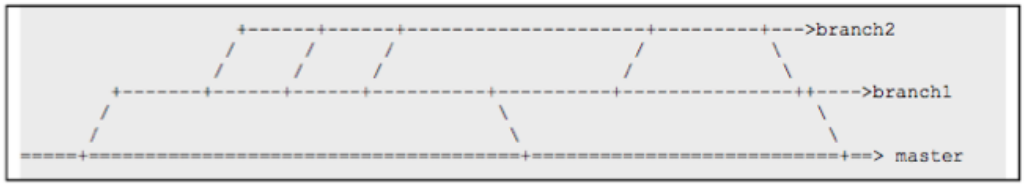
You know where this is going.

<Insert favorite snarky gallows-humor quip>

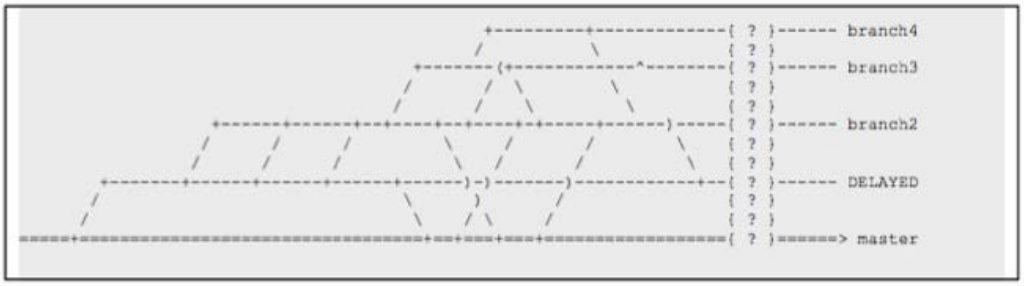
The Problem — With Pictures

During development of our new CloudForge product, we also developed an API for our partners to use to integrate with us.

The Plan:



What Really Happened:



Or something like that.

More branches were created than we'd expected. Intoxicated, perhaps, by Git's vaunted branch magic, merges were done among branches that should never have been merged. Scope changed. Schedules changed. Priorities shifted. Brilliant new ideas appeared. The luster of old ideas sometimes faded.

You know: life happened.

Bottom line, the work I've been calling "branch 1" (CloudForge) increased significantly in scope, became far more wonderful in our eyes than it had ever before been … and was granted an Agile schedule extension to become all that it could be. Meanwhile, the work I've been calling "branch 2" (the API) remained prosaically un-replanned, basically on schedule, and steadfastly desirable in its own right and on its original schedule. When it came time to release branch2, there was no branch1 release vehicle to carry it.

Worse, much of the branch1 stuff had drifted over into branch2, and though we wanted to release branch2, we definitely did not want to release these bits of branch1—not yet, anyway. We were faced with the need to merge branch2 to master without the branch1 stuff, and to do it in a way that wouldn't interfere, later, with delivering all this work as part of the branch1 release.

Around 1200 commits to assess, after discounting the around 600 merge commits.

In the end, after much trial and learning, we were able to use Git, Gitk, and cherry-picking to construct the branch weshould have built in the first place. In my next few blogs, I'll tell you how.

Deconstructing a Git Branch: Tools Fail

In our last episode, I confessed to making an unholy mess of a branch snarl by ignoring standard software process 'Best Practices', and running head-on into an especially concentrated cluster of all those things that make those Practices the Best ones.

But, hey, we're using Git! Git is great! Git is all about branches! No worries, right?

Starting Simple – Rebase

So, I hear there's a rebase command in Git, that allows you to "edit history" so it looks like what it should have looked like. Maybe that's what I want? Let's ask Google:

google: git rebase edit history

OK, loads of pages there. Peruse a few. Lots of info, lots of different perspectives. One common thread, though, that runs through them all:

Do not include any commit you have already pushed to a central server – it will mess other people up.

OK, that lets me out: my messy changes have been pushed, and pulled, and built upon, repushed, merged to other branches, folded, spindled, and mutilated. That's the whole problem. So, let's drop rebase.

More simplicities – cherry-pick

How about this: we leave the current, confused branch alone, and build up a new one to be what the original should have been. Git has a cherry-pick sub-command that applies one change to a branch. We just have to find the last moment when the branch was as desired, and roll forward from there cherry-picking only the desired changes.

git help cherry-pick

Hmm … four pages long. Hmmm …. third sentence gives me a few qualms:

When it is not obvious how to apply a change, the following happens:

But, let's give it a go.

Picking cherries

Since the exercise here is to select some changes and not others, I'm not going to be able to just tell cherry-pick to do them all at once: I need to figure out which commits are wanted. Figure there are really three kinds of commits:

1. The ones I want: changes made explicitly to my branch, or to working branches off my branch (there's some of that going on).
2. Merges, in any direction (the merge itself isn't cherry-pick material anyway: we'll deal with the actual changes instead)
3. Changes originally made to the delayed branch (and then merged into mine, or into a working branch off mine)

Now git log (from within a check-out of my branch) will tell me every commit that has contributed to my branch, so far so good.

At this point, I ran into a problem I was unable to solve. Note that git log tells you about each commit regardless of what branch the change was originally made to, regardless of how convoluted the path from original commit to my branch. Ordinarily, that's a good thing, knowing the change itself regardless of the path. But in this case,

the paths are important to me: in some cases, the only way I can identify whether I want the change is to know where it was first made (or to constantly bug the author for that info). I suspect git can tell me this. In fact, as you'll see in subsequent chapters, the graphical tool gitktells me this quite cleanly, and I have a strong suspicion that it's merely wrapping command-line functions, or something darned close. But I couldn't find how to do this from the command line, and furthermore my efforts to do that were digging me deeper and deeper into "learning opportunities" (that is, things I didn't understand, like advanced Git Revision Expressions).

[Interesting story: at least at the time of this writing, Google can't find any documentation on Git Revision Expressions. Very odd, since there's so much good info on so many other git-topics. The best googlable reference is this Stack Overflow article, which points out that the answer is in GIT-REV-PARSE(1) (the man page, or git help page, for Git's rev-parse command). Which is highly googlable. But you apparently have to know the answer before you're allowed to ask the question. If you head down this path, I can recommend The Horse's Mouth, and also a blog that, if nothing else, proves just how complicated all this stuff really is!]

Well, we could dig down that rat-hole for several more miles. But fortunately, we don't have to. Stay tuned for my next installment (scheduled for September 18) so I can show how you the Easy Way!

Deconstructing a Git Branch — The Tools…Seriously?

As I confessed last time around, I mismanaged some Git-branch evolution and had to reconstruct what the branch would have looked like, if I wasn't an idiot. I tried to fix things up with the Git command-line, which I'm sure can be done … but I sure couldn't find all the pieces to do the job.

So I turned to GUI tools, hoping for extra insight. After all, Git has GUI tools, right? I read that somewhere, recently.

What I found was that many of the Git GUI tools are designed to provide GUI assistance to the basic, common operations. That's not a bad thing, but not really what I was looking for. I did some research, but eliminated most offerings (for my purposes) rather quickly. I'll talk a little about those, anyway, in case you're in the market for "easy-to-use git-basics," but will focus on the search for a tool so clever it can even clean up *my* messes.

As I do, and more so once I roll up my sleeves and get down to work, I'll be working with some sample data I ginned up for illustration. If you're curious, you can download the repo and play along. Here's the repository. And here's its GPG signature.

Git GUI tools

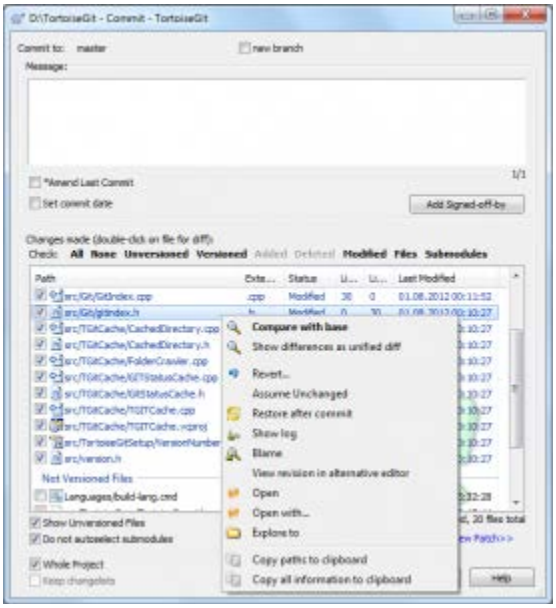
gitk

First out of the gate is gitk. This is the most often-mentioned, most developed of the stable. It comes with git itself, so you probably already have it. It should be available on whatever git-enabled platform you have. Unfortunately, though it's very powerful, it's also quite overwhelming: there's a lot of power here, but more investment in making it available than making it accessible. I need some power, but I also need some guidance. Let's set this one aside, and see what else we can find.

git-gui

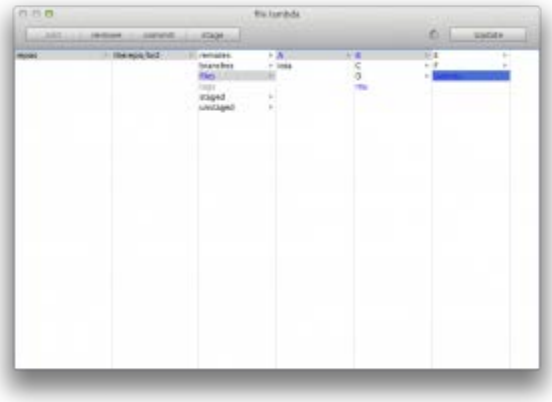
Next up is git-gui, which also comes with Git, and is probably already installed on your system (type git gui and see). This is a bit more presentable, and provides a bit of guidance, too. For example, it was this GUI that first suggested to me that “Sign Off” was important enough to live among the “basic, main-line operations.” But right there is the problem: this is one of those GUIs intended to provide a GUI wrapper around the basic, common operations. Not a bad implementation of that, but not the problem I need solved. Moving on.

TortoiseGit

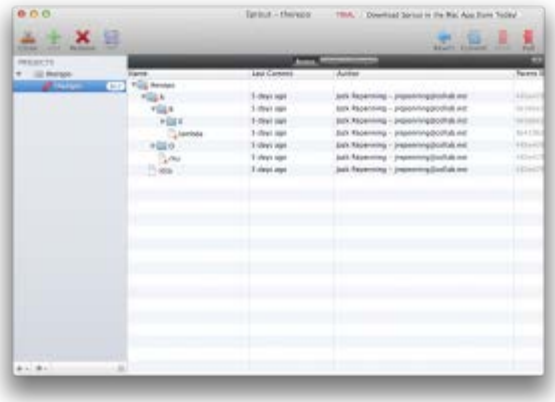


This is Windows-only, which kills it for me, but if you’re a Windows type, you might want to check it out. It also requires the msysgit version of git itself.

Katana



Katana is a Mac-only tool (which is fine with me). Katana provides several ways to browse several ways (files, branches, stagedness). But it only provides the basic operations.



Sprout. Sprout is a Mac only pay-ware. It has a better browser than any of the above, and a good Mac look-and-feel. Somewhat more than just “basic” operations, but still no help for me.

QGit.

Linux only (experimentally Windows). No help for a Macophile like me.

Giggle.

Linux only.

Conclusions

So the only one of these that offers me any hope of help is gitk, the scary one.

Next episode, we roll up our sleeves, bite the bullet, knuckle down, and crack out a good platitude!

Deconstructing a Git Branch — A Guided Tour

Today, we continue the saga of Deconstructing a Branch in Git. Last time out, I took a survey of graphical tools, and concluded that the only one likely to help in this particular situation is gitk. As I summarized,

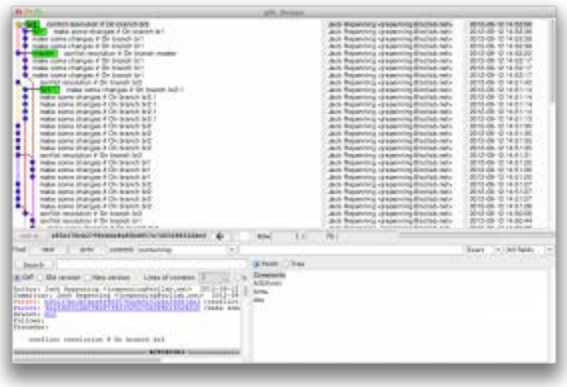
Unfortunately, though it’s very powerful, it’s also quite overwhelming: there’s a lot of power here, but more investment in making it available than making it accessible.

So before I dove into the actual work, I spent some time figuring out how to find my way around. Also, as mentioned last week, these examples will use a repository full of nothing in particular, and you might find int useful to download the same repo (along with its GPG signature), so you can play along.

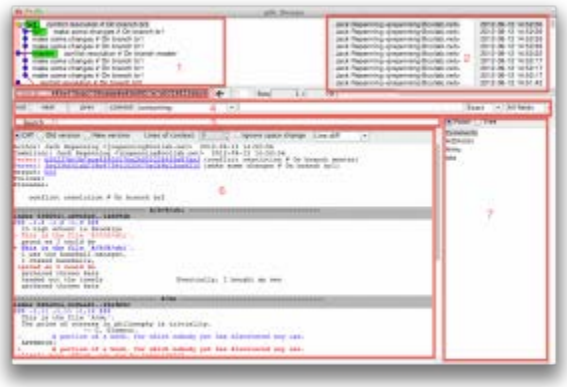
Where to begin?

As with many tools, you begin with gitk by establishing some context: a repository, a checkout, and a working directory. Most Git commands assume, at least by default, that they should operate on “the repo/checkout/working-directory right here.” If you’re using a general GUI tool, like say TortoiseGIT, “right here” means the directory contents shown on your screen. If you’re using the command-line tools, then you use your shell to “cd” into the appropriate spot.

Having established your “right here,” launch gitk. From the shell command line, just type “gitk” and press return. What you get is something like this:



You might first want to play around a bit with the window controls, and in particular notice that both the main horizontal divider, and all the vertical ones, can be dragged to relay the window. Here's the same window again, but with what might be a more accessible allocation of the space:



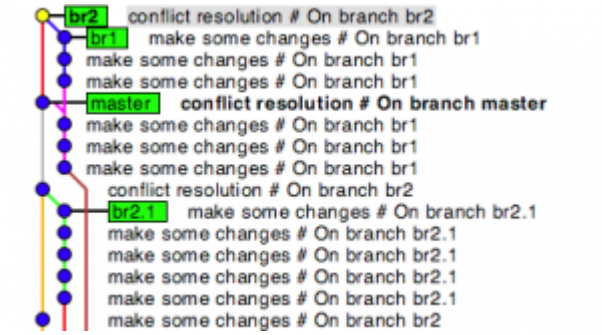
Now, what are all these sections?

1. The upper-left corner displays the version tree (what I call the "tree view"), including branches (places where the tree splits) and merges (places where two branches come together). It's organized newest-on-top, and it contains all the history that led up to the particular files-and-versions that were "right here" when the tool was launched–and only that history: changes that haven't contributed to your chosen "right here" aren't there. This area will be the primary focus for the real work of branch deconstruction, and we'll get back to here soon, but first let's look the rest over.
2. The upper-right area is line-for-line matched with the versions listed in the upper left, showing the author and date of each change. You can use this as a supplemental control, by clicking on an author or date to select a version in the tree view.
3. Area 3 shows the SHA1 hash ID of the selected version. When gitk is launched, that will l be the HEAD of the branch you have checked out, but if you click around in the tree view a bit you'll see this field change to follow (in fact, nearly everything in the display tracks the tree view in this way). If you've worked with Git at all, you've noticed these hashes are central to the user interface (and guessed that they're central to the implementation as well).
4. Area 4 is a tool that searches the commit logs. Try typing "conflict" into the large text field there. You'll see that several rows in the tree view turn bold–the ones with "conflict" in their description. You can now click one of those to focus on that change, or use the "next" and "prev" buttons to select among the matching ones. So if you're trying to find the commit that broke the log in feature, then you can just search for "break log in," right? Well, maybe not … but you get the general drift. In my data set, the commit logs aren't very informative, but try searching for "merge" or "conflict".
5. Area 5 is another search widget, but this one searches the text currently displayed in area 6. This won't switch you to new versions, only navigate around in the display of a particular change. So, for example, once you've used the Area-4 search to find the change containing the log text "break log in," you can use the Area-5 search to find the comment that was inserted into the text, "This breaks log in." Or, in my data set, you might search each change to see what changes it made in a particular file. The files, in this repo, have names which are Greek letters spelled out in English:alpha, beta, gamma, delta, and so on, so try searching various changes for iota or mu or omega.
6. Area 6 displays the patch that constitutes a particular change. There are some controls for how to display it, and how much of it to show. There's also…

7. Area 7, which chooses between having Area 6 do what I just said, or just having it display the contents of each file in the working copy, regardless of whether it happens to be affected by this change.

Where'd everybody go????

Let's go back to Area 1, the tree view:



I'll include the SHA1 hashes as annotations; if you click around in your own gitk and get lost, just paste these into the Area 3 ID field.

Line 1, SHA1 485e478cb2798cbde8e85b8857e7d0328832ddc6

Recalling that this is listed newest-first, and from a branch br2 "right here," we can immediately understand

- Why the first line is first (it's the last change contained in the history)
- Why it's labelled br2 (that's the branch where the change was made, as you can see from the comment)

Line 2, SHA1 fe4336f41dd739e973941525470e3f9b13ce9210

The second change adds a slight nuance: it's second because it's the second-latest change contained in the history (it was merged to br2, our "right here"), and it's labelled br1 because that's where the change was made. But the nuance: this is not necessarily the latest change that has been made on br1: there may be later changes that have not yet made it to br2. They aren't shown, because our "right here" is the HEAD of br2. Only changes that are contained in "right here" show up in this display.

Line 5, SHA1 b26237b43b7aca9f905576a2b925328450d87da3

Skipping ahead just a bit, look at the line with the "master" label. That's a change that was made on the master branch, and it's the latest master-branch change that's present in our "right here" anchor point, which all makes it very familiar. But what is that little dot doing on the left-most line? I thought that was the br2 line! Is gitk confused?

Well, no. But it's telling you either a bit more, or a bit less, of the truth than you were expecting to hear. There are two things contributing to this surprise display.

The "Branch" fantasy

We tend to think of a "branch" as something solid and persistent, a steady presence we can rely on. The truth, however, in any versioning system, is a bit different: the contents of "the branch" at any moment may consist of work that was originally done on many branches. This is the goal and point of merging, after all. There's a conventional fiction inherent in ideas like "the changes contained in a branch." While at any moment in time it's unambiguous to talk about "the branch," digging back into its past will lead you to a complex tree of changes (actually, a "Directed Acyclic Graph," or DAG, but let's not quibble). It may lead you into and out of many branches. It's completely answerable "which changewas made on which branch," but it becomes very mystical to talk about "which change is on which branch": a given change will often be on many branches. In Git terms: at any moment in time, the HEAD of any given branch is clear, but the branch-based identity of any given change may be elusive.

Most version control systems make some effort to represent the branch in some persistent way, which is nice for the humans trying to make sense of it all. But, in any version control system, the notion of a linear, stable, solid "branch" is a synthetic fantasy. And worse than that: in order to provide this comforting fantasy, branch-

perserving versioning systems have to downplay or obscure some information, such as the original authorship of a given change: during a merge, a new "version" is created for the notion of "add this to the branch," and the change comes to be "owned" by whoever did the merge, rather than whoever did the original work.

Git doesn't work that way. A given change, regardless of its peregrinations through the branches, retains its original identity, authorship, date, and so on. This is very nice, from the perspective of properly attributing contributions to a large open-source project like Linux (the design target for Git). But from the standpoint of "people who think in terms of branches," it gets a bit awkward. In this case, gitk is telling you the whole, unvarnished truth: There Ain't No Such Things As Branches. But I could wish it would take the time to pretend there were ….

The Missing Merges

But gitk is also telling you a bit less than it could. In the Area 1 display, you can see two kinds of changes: plain old changes (with commentary like "make some changes"), and merges that resulted in conflicts that had to be fixed ("conflict resolution"). But if you'd been watching over my shoulder as I did all this, you'd notice that there were often changes of a third type: conflict-free merges (which Git calls "fast-forward" merges). A fast-forward merge occurs when the two branches to be merged are identical except for some number of additions on only one branch. For example,

1. Sprout branch-one off of master
2. Change branch-one (and don't change master)
3. Merge branch-one back into master

Since no changes were made to master, this is really easy to do: just apply those step-2 changes to master in the same way as they were applied to branch-one. This is a fast-forward merge.

Gitk is not showing the fast-forward merges, but they're hiding there, wherever the branch picture gets confusing.

Coming up next

All this logic-chopping and nuanced reading of surprising displays will become crucial to the process we came here to do: deconstructing a polluted branch and reconstructing the branch as it ought to be.

Next time!

Deconstructing a Branch — Rolling Up Our Sleeves, Battling the Beast

This is the final installment in the continuing saga of cleaning up the feature branch I put through the Osterizer. In previous episodes, we've decided a GUI tool would be handy, surveyed some available GUI tools, and explored the power and quirks of our likely choice. It's now time to roll up our sleeves and get some work done!

As I've mentioned before, this will be a detailed, step-by-step walk through of some actual (artificial) data. You really might want to download the repo I'lll be using, so you can play along. Here's the repo,and here's its GPG signature.

The Set-Up

Within this repository, you'll find four branches:

- master, used as the actual release for the product
- br1, a major feature branch begun some time ago
- br2, a minor feature branch, related to the br1 work
- br2.1, a small branch for some side work related to br2

The Problem

At the time br2 was created, it was based on existing work in br1 and was planned for release along with br1. But then, Reality happened: br1 was delayed for a massive growth in scope, becoming far more wonderful, but also far later. Meanwhile, customer demand for the features of br2 was becoming strong. The decision was made to reorder the releases, shipping br2 first, without the br1 stuff. By this time, however, br2 contained substantial amounts of br1 work, which ought not to be shipped And thereby hangs the tail.

The Solution

We created a new branch, containing all the br2 and br2.1 work, but excluding (as much as possible) the br1 work. We did this primarily through git's cherry-pick feature, applied and managed through the git GUI tool. Along the way, we ran into several problems that forced us to work around, or even restart the process again. My principle goal in this blog is to point out these pit-falls, so you don't run into them yourself.

The Process

At this point, you should unpack the sample repository. If you're on Unix, Linux, or OS X, you should be able simply to double-click therepo.tgz, to produce a directory tree named therepo. I suggest you open two shell windows (xterm, Terminal, or whatever you like).

In the first shell window, run git branch. You'll see the four branches I've mentioned, and the branch br2 will have a "bullet", indicating that it's your current branch. This is the branch with all the stuff we want, and some stuff we don't.

```
> git branch
  br1
* br2
  br2.1
  master
```

In this same window, run gitk. A new window will open up. (You may get some inscrutable messages in this shell window, from time to time; you can ignore them so long as the gitk window behaves properly.)

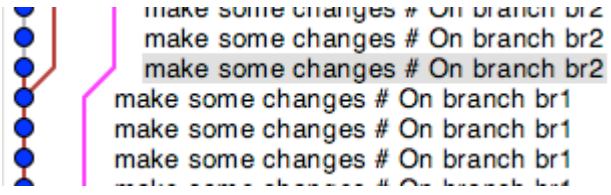
What you see

Look first at the panel in the upper left. This is the primary control point. There's a graphical representation of the history of this repository, with each change represented by a dot. There's also a list of the comments associated with each change.

Caution: you'll notice that every comment is annotated to show that the change was made "On branch br2" or whatever. I did this to make this blog easier. If you're ever doing this for real, you won't have this!

Find the starting point

We want the latest (nearest the top) version of branch br2 that is not polluted by undesirable br1 work. Scroll around a bit, and you'll find that the very earliest reference to br2 is on row 50 (notice the "Row 50 / 71" indicator right below the column of author information).

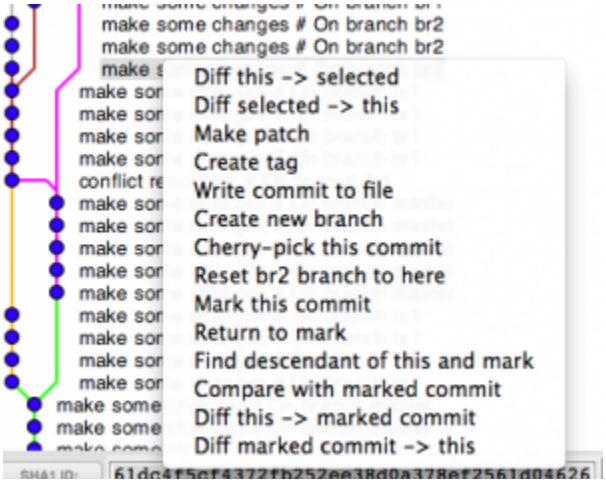


Surprise #1: Notice that the changes on br1 and the changes on br2 appear to be in the same line, even though they're different branches. In general, gitk doesn't keep a given branch on the same line. Not only do branches wander among lines, as we see here, but there can be stretches when no line on screen represents a particular branch, even though the branch existed at that time, and has changes both earlier and later than that. So be very careful in determining which branch a change applies to!

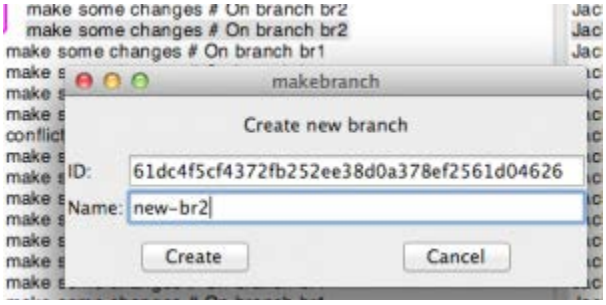
OK, so, we want "br2 without any br1 stuff," yet even the very first work on br2 was based on, and contains, br1 stuff. In our case, we were able to convince ourselves that the early br1 stuff was OK, wouldn't actually appear on screen, so we ignored it. So we will take this line here, the first one that was actually made on br2, as the base of our new branch.

Click on that line, line 50 (if you haven't already), so it's grayed as I show here.

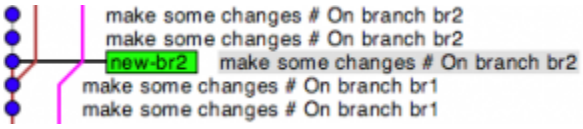
Right-click (or control-click, or whatever your favorite window system does to mean "open the contextual menu"). You should get this:



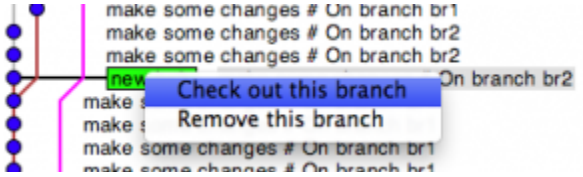
Pick "Create new branch," and name it **new-br2**.



A new notation appears on the screen, showing the location of your **new-br2**:



Point your cursor at the "new-br2" marker (not the "make some changes" text), and right-click yourself a contextual menu again. It will be different from before:



Pick "Check out this branch." This will switch your repository to looking at new-br2.

Surprise #2: At this point, your repository is looking at new-br2, but your gitk window is more or less still looking at (old) br2. This is a somewhat delicate situation. Don't refresh the gitk window, or quit-and-restart, or you'll lose this "straddling the branches" view, and it may be hard to get back.

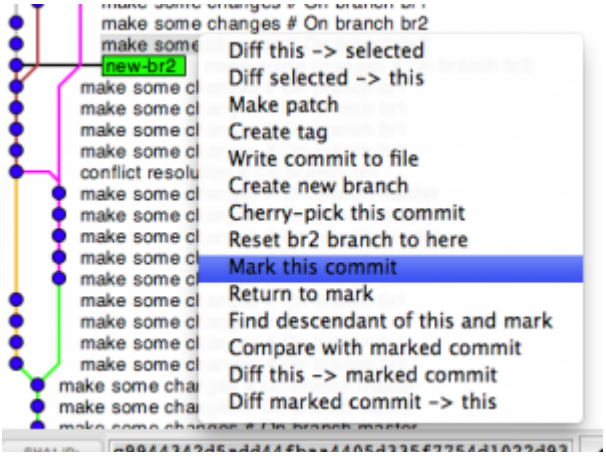
What comes next?

What we're going to do now is work our way upwards through the display, looking at successively more recent changes, and deciding, one by one, whether we want them in new-br2. For this blog, the answer's easy — anything with the comment "On branch br2" or "On branch br2.1" is good; anything that says "On branch br1" or "master," we skip. Basically. But don't tune out just yet.

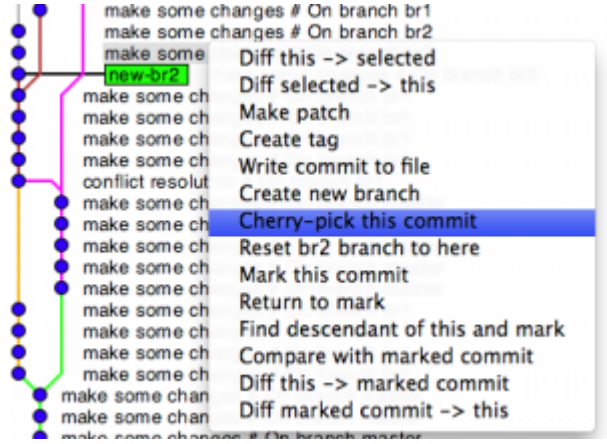
Use the keyboard up arrow, or click with the mouse, to select line #49, another "On branch br2." Do we want this? Yes, we do: it's br2. So context-menu on that line and cherry#####

Hold on.

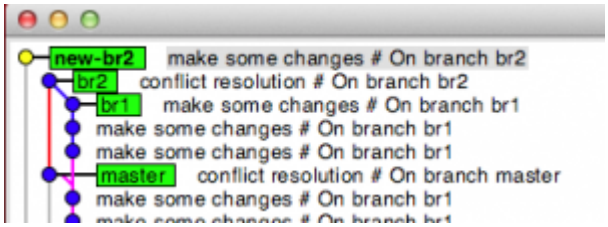
Surprise #3: Before you cherry-pick, you should mark the current line:



Why? Well, here, let me show you: context menu again, and now do pick "Cherry-pick":



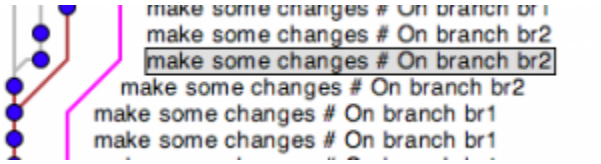
Whoa! What happened? The graph scrolled up to the top, and started new-br2 up there.



Meanwhile, the gray marker showing which commit we considered last is gone. That would make it much harder to figure out which commit to do next!

Cherry-picking means plucking one commit from one branch, and attaching it to another. I guess it should be called "cherry-transplanting." But, it's not.

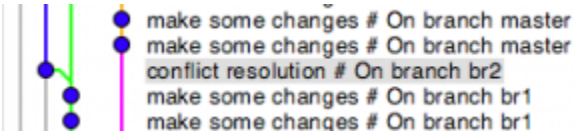
Good thing we marked it! So now, either use the context menu "Return to mark," or just scroll around until you spot the commit with a box around it:



So, remember: mark first, then cherry-pick!

Up arrow or click to the next one. Do we want it? Sure do: it's br2. Mark it. Cherry-pick it. Back to the mark. Up-arrow through a few br1 changes–we don't want those!

This brings us to line 44:

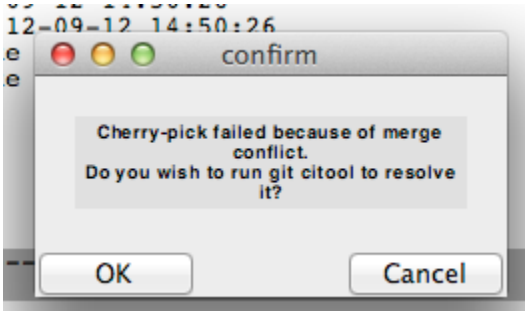


Now this one says "br2," but do we want it? Actually not. Notice the graph: this change represents merging some stuff from br1 to br2. We just carefully skipped over those br1 changes, it would be a shame to drag them in via this merge! So, don't.

Or, actually, just for drill, do cherry-pick this merge. What you'll see is this:

OK, that's a relief: it refuses to do the merges. So if you ever git caught up in the rhythms of the process and do try to cherry-pick a merge, you're safe. (You really are "safe." Yes, I know the alert a chock-full of words like "fatal" and "abnormal," but in this case that's exactly what we wanted.) OK that, and on we go.

Next (line #43) is some work done "on branch master." Skip those, too. Carry on skipping br1, and mark-and-picking br2 a bit, until you get to line #39, and this:



What this means is that some changes in this commit attempt to change the same lines as some other changes. In context, what it really means is that, originally, these changes built on top of some br1 stuff we've been skipping. Well, now the hens come home to roost: we have to straighten this out.

I recommend you not use “git citool” here (just cancel it), but instead go to your second shell window. You remember that second shell window I had you o pen? Now you know why.

In the second shell window (if you haven’t already), cd into the repository. Just for confirmation, try a git branch here, and see the new “new-br2” branch, and the bullet indicating it’s the current one.

```
> git branch
  br1
  br2
  br2.1
  master
* new-br2
```

Now, use git status to see what the problem is:

```
> git status
# On branch new-br2
# You are currently cherry-picking.
#   (fix conflicts and run "git commit")
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#
#       both modified:      A/B/E/alpha
#
no changes added to commit (use "git add" and/or "git commit -a")
```

OK, so notice that “both” branches modified the file A/B/E/alpha. Pull that file into your favorite text editor, and you’ll see a region like this:

```
innovations have resulted in tremendous cost savings which have been
over water.  They can ram competing planes in mid-air.  These
<<<<<<< HEAD
=======
apply, the main one being that all these flights take you to Newark,
>>>>>>> c0e0dfa... make some changes # On branch br2
passed along to you, the consumer, in the form of flights with
amazingly low fares, such as $29.  Of course, certain restrictions do
```

The part between “<<<<<< HEAD” and “======” (i.e., the nothing) indicates that the new-br2 branch we’re building up doesn’t have anything at that point. Maybe something was deleted. Correspondingly, the part between “======” and “>>>>>>>>c0e0dfa” is the stuff we’re bringing in from br2 into new-br2.

This is not the place to fully explore git merge conflicts. For purposes of this blog, let’s just say that we want the line that begins “apply, the main one”. Remove the marker lines and the nothing between “<<<<” and “=====”, save that out, git add A/B/E/alpha, and git commit.

The trick that isn’t

If you proceed in this way, you’ll eventually come to some changes that were made “On branch br2.1.” I said, above, that we do want these changes, so we must “mark and cherry-pick” them somehow or other, right? Right. But they’re on Some Other Branch, that must introduce some new complications, right?

Nope. Just “mark-and-pick” like always. Cherry-picking can take individual changes from nearly anywhere, in just the same way.

Repeat until done

That’s about it. Keep advancing from one commit to the next, deciding whether you want it, marking, cherry-picking, and occasionally resolving merge conflicts.

Remember: If you can keep your head about you when all around are losing theirs, … Yours is the Earth and everything that’s in it!

CONTACT US

Corporate Headquarters
8000 Marina Blvd, Suite 600
Brisbane, CA 94005
United States
Phone: +1 (650) 228-2500
Toll Free: +1 (888) 778-9793

Topics trending now

Many of the latest technology announcements have implications for PaaS and cloud development that will serve agile businesses everywhere.

- Enterprise Cloud Development, www.collab.net/ecd
- Continuous Integration, www.collab.net/getci
- 5 Things your Development Team need to be doing now, www.collab.net/5things

For More Information

CollabNet is your one-stop shop for enterprise-grade Git management.

Subscribe to Git Blogs: http://blogs.collab.net/email-subscribe?catName=Git

24/7 Git Support: http://www.collab.net/downloads/git-enterprise

Enterprise Git Management: http://www.collab.net/downloads/git-enterprise

Git Toolkit: www.collab.net/gotgit<http://www.collab.net/gotgit>

About CollabNet

CollabNet is a leading provider of Enterprise Cloud Development and Agile ALM products and services for software-driven organizations. With more than 10,000 global customers, the company provides a suite of platforms and services to address three major trends disrupting the software industry: Agile, DevOps and hybrid cloud development. Its CloudForge™ development-Platform-as-a-Service (dPaaS) enables cloud development through a flexible platform that is team friendly, enterprise ready and integrated to support leading third party tools. The CollabNet TeamForge® ALM, ScrumWorks® Pro project management and SubversionEdge source code management platforms can be deployed separately or together, in the cloud or on-premise. CollabNet complements its technical offerings with industry leading consulting and training services for Agile and cloud development transformations. Many CollabNet customers improve productivity by as much as 70 percent, while reducing costs by 80 percent.

For more information, please visit (www.collab.net).

CollabNet, Inc.
8000 Marina Blvd.,
Suite 600
CA 94005

Tel +1 650 228 2500
Fax +1 650 228 2501
www.collab.net
info@collab.net

Blog blogs.collab.net
Twitter twitter.com/collabnet
Facebook www.facebook.com/collabnet
LinkedIn www.linkedin.com/company/collabnet-inc