Volume Number: 26
Issue Number: 04
Column Tag: Code Resources

Branching Out with Git

Learn how to use Git to manage code branches

Introduction

Last we looked at Git, we learned how to use this tool to manage a simple set of project files. Then, we used the same tool to setup and manage another project as part of a distributed source-control management (dSCM) scheme.

Today, we will look into the tricky concept of code branches. First, we learn the benefits of branching and what issues to expect. Then, we learn how branches are merge and how conflicts are resolved. Next, we will use Git to create and manage branches of a simple text file. We will also use the same tool to merge some of those branches.

Readers must know their way around a Terminal window. Also, they must have a working knowledge of Git's basic features. For a good introduction to Git, consult one of the references listed at the end of this article.

The Project Cycle

A typical project cycle goes through four basic phases (Figure 1). In the development phase, the project's core features are in flux, subject to change or removal. The source code is rough, with minimal factoring, and it tends to be un-optimized. The user interface is sketchy and mostly incomplete. Development builds are often too unstable for widespread testing. But, they are useful as proof-of-concepts. Version numbers for these builds get either a 'd' or 'e' tag.

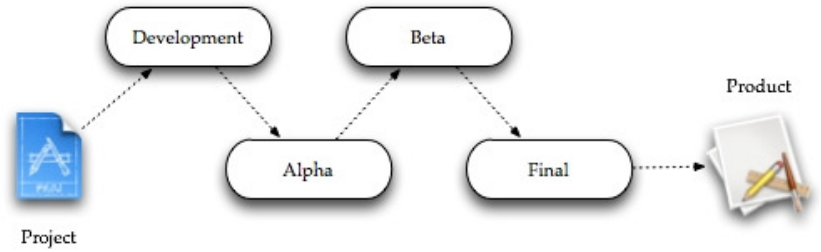

Figure 1. Phases of a typical project.

Once the project enters the alpha phase, its core features are well defined, though its auxiliary features are not. The core code is still in flux, yet stable enough for internal testing. Its user interfaces will be mostly stable, but still subject to change. Alpha builds always has an 'a' tag in their version numbers.

Next, the project reaches the beta phase. At this phase, it has most, if not all, of its features implemented and working. All user interfaces are stable, subject only to some cosmetic changes. This is also the stage where the project enters active testing by third-party groups. Any bugs found during testing are then evaluated and fixed. Version

numbers for beta builds always use a 'b' tag.

At long last, the project goes into its final phase. By now, the project must be feature complete, its user interface polished. Any incomplete or buggy feature must be disabled and documented. Any remaining bugs must also be documented. Plans for the next project release are often done at this phase.

Branches in a project

Suppose you have a new and unplanned feature to add to the project. But you are unsure if this feature will even work, let alone be stable. One way to do this is to first tag all affected project files, and then proceed writing the feature code (Figure 2). If the feature failed your expectations, you can revert the files back to their previous state. If it works, you then commit your changes back to the repository. This approach, however, disrupts the project cycle by putting one or more files back to an earlier phase. It does not guarantee that the added feature code plays well with the rest of the project. And, if the affected files revert to their previous state, all records of your revisions will be lost.

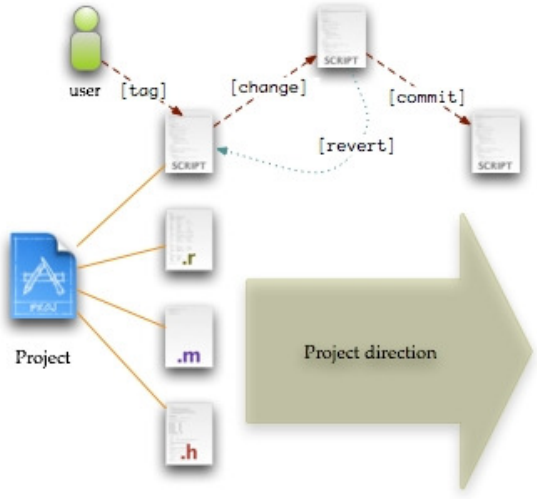

Figure 2. Adding a new feature.

Next, suppose you want to create two variants of the same project: a PRO variant, which has features that most users are willing to pay; and a FREE variant, which has a subset of those features. One way to handle this is to create two copies of the project (Figure 3), one for each variant. But this approach is harder to manage when both project variants have a lot of files. You have to track which file belongs to which project variant. You have to keep all common files in synch. And, if you decided to add a PRO feature to the FREE variant, or vice versa, you have to copy or move the right files to the latter.

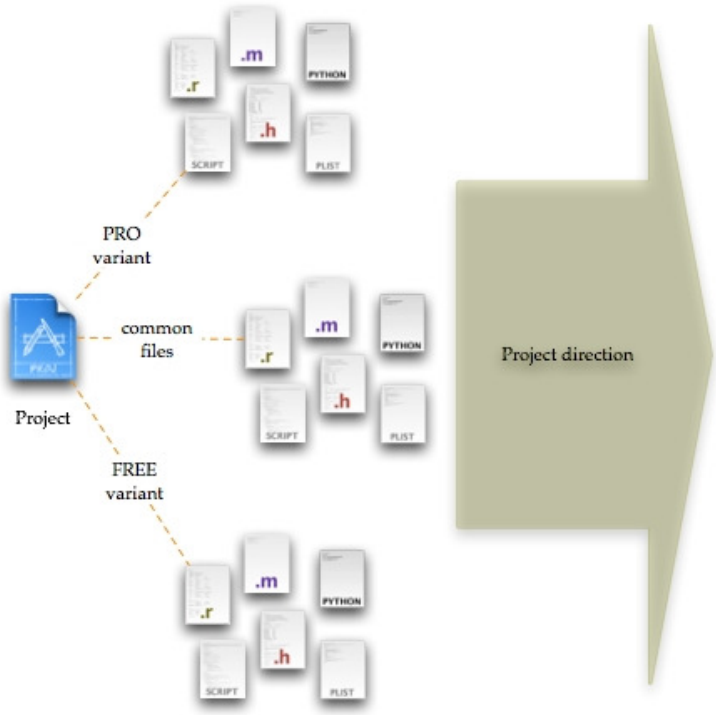

Figure 3. Working on two project variants.

A better way to do either one is to branch out specific parts of the project (Figure 4). Each branch has the files that implement a unique feature or set of features. The rest of the project, called the trunk, has files common to the project and files shared by the branches. Changes made in the branch files remain in that branch. Likewise, changes made in the trunk files remain in that trunk. A branch can also spawn other branches, each with its own set of files. It can also merge with other branches or with the trunk itself.

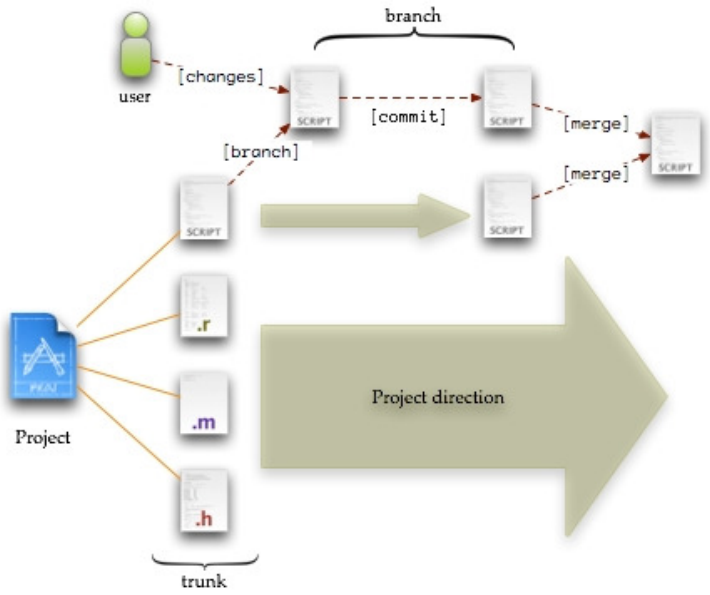

Figure 4. Branches and merges in a project.

Branching helps preserve the project's working state. It lets you isolate any unplanned changes, leaving the main trunk unaffected and viable. It lets you work on one or more project variants in parallel, while keeping common project aspects in synch. And it lets you track your activities in a specific branch or in the whole project.

Merges in the project

As stated earlier, you can merge one branch with another in the same project. Or you can merge the branch back to the main trunk. Merges combine the files in the first branch with those in the second branch or in the trunk. If a file is absent in the second branch or

trunk, the merge then makes a copy of said file. But if the same file exists, the merge can take one of two possible actions.

The first merge action uses two files: A and B (Figure 5), A being the source file and B the target. Here, the merge finds out which lines in file A are present or absent in file B. It also notes down which lines in A are different in B. Next, the merge gathers its results into a data set called a diff. It then uses this set to create file C, which is a modified copy of B. Optionally, it can write its diff data set into a separate file. This merge action is called a two-way merge.

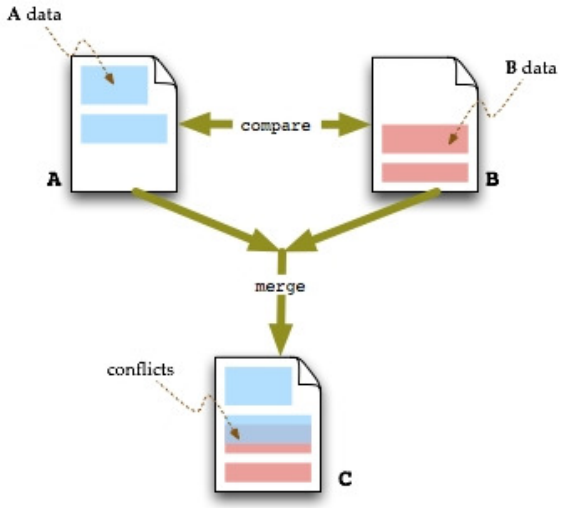

Figure 5. A two-way merge.

The second merge action uses three files: A, B, and C (Figure 6). Here, files A and B are revisions of the same file C. First, the merge compares files A and C, then B and C. As before, it keeps the results of both comparisons as two sets of diff data. But when it creates a revised copy of file C (called D), the merge uses both diff data sets as a guide. This action is known as a three-way merge.

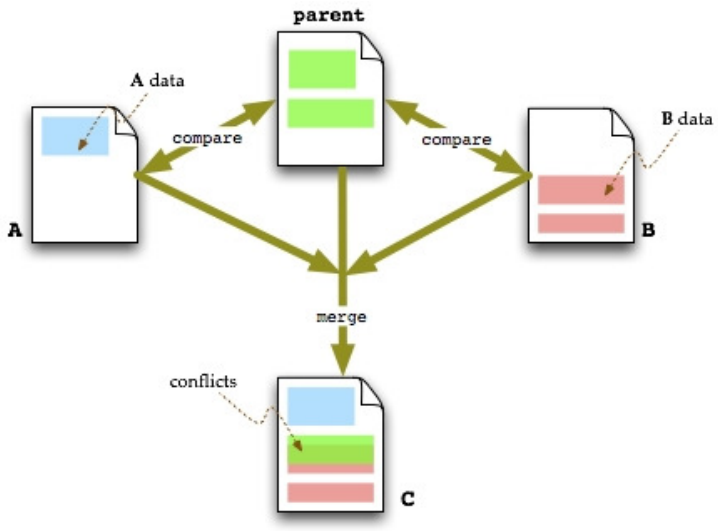

Figure 6. A three-way merge.

A two-way merge uses fewer resources than a three-way one. But a three-way merge is more reliable, requiring less user action to resolve any errors. Both merge actions will flag any overlaps (or conflicts) in the merged files with special tags. These tags aid us in resolving those conflicts manually.

Working With Branches

You will now learn how to create and manage branches with Git. Most branch operations are done using the command git-branch. This command can have up to three arguments, which are arranged as follows.

```
git-branch [branch-options] [branch-name] [branch-parent]
```

Here, branch-name is the name of the branch. That name is a unique string of alphanumeric characters. The argument branch-parent is the source of the branch. It can be another branch or the trunk itself. The argument branch-options sets the behavior of the command. It consists of one or more flags.

Building the trunk

Begin by creating a repository with a single file. First, go to the Finder and make a new directory named Foobar in your home account. Then launch your favorite text editor and create a new file with the contents shown in Listing 1. Save this file as Foobar.text in your new directory.

Listing 1. Initial contents of the test file.

```
Foobar.text
   Pease porridge hot, pease porridge cold,
   Pease porridge in the pot, nine days old;
```

Next, start a Terminal session and use the cd command to navigate to the Foobar directory. Type git nit to create an empty repository. Then type git add . (note the period at the end), and then gitcommit -m "Original test file". These add the file Foobar.text to the repository.

Next, switch to your text editor and append the following line to Foobar.text.

```
   Some like it hot, some like it cold,
```

Save the change and switch back to the Terminal session. Type git-commit -m "First revision at the trunk" Foobar.text at the prompt. When done, your repository has the structure shown in Figure 7. On the left is the original Foobar.text; on the right is the revised Foobar.text. These two files comprise the project's trunk.

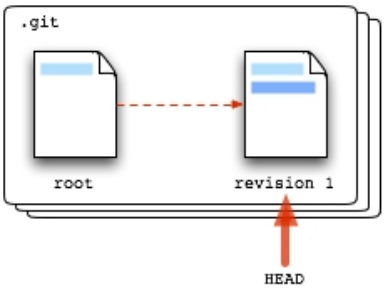

Figure 7. The project trunk.

Making branches

It is time to make your first branch. At the Terminal prompt, type git-branch foo. This command creates a new branch off the trunk and names that branch as foo. Then, it copies all the files from the trunk into foo. But which part of the trunk is the branch attached to? Recall the repository structure in Figure 7. After you changed, saved, and committed Foobar.text, Git places the HEAD pointer to that revised file. Then, when you create branch foo, foo gets the latest revision of Foobar.text (Figure 8). Note the original Foobar.text is still in the repository, available as a fallback option. Note also the HEAD pointer still refers to revision 1of Foobar.text on the trunk.

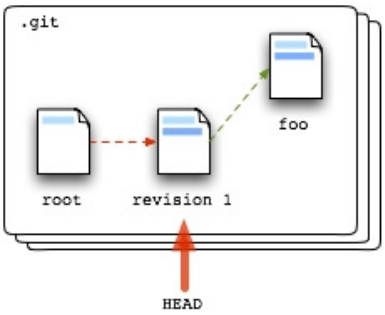

Figure 8. Making branch foo.

Let us create a second branch. At the Terminal prompt, type git-checkout -b bar. Git creates a new branch off the trunk and names that branch as bar. It copies the latest

revisions from the trunk to the branch. And it moves the HEAD pointer to that branch's files (Figure 9). Thus, any changes you make to Foobar.text will go to branch bar of your repository.

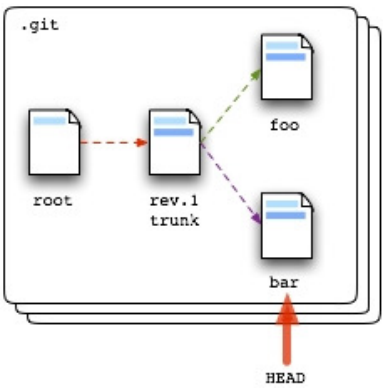

Figure 9. Making branch bar.

To demonstrate, add the following line to the file Foobar.text.

```
Some like it in the pot, nine days old.
```

Save your changes, and then commit by typing git-commit -m "Revision 1 at branch bar" Foobar.text. When done, your repository now appears as shown in Figure 10. Also, HEAD points at the revised Foobar.text on branch bar.

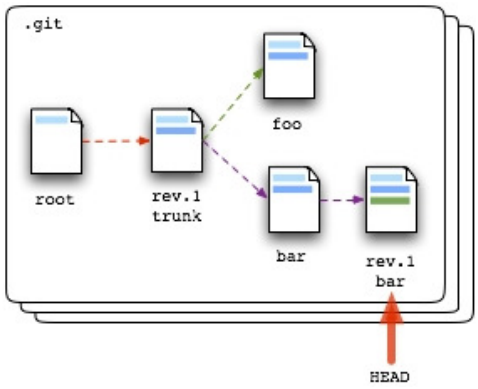

Figure 10. Revised branch bar.

Switching branches

Suppose you want to revise a file in branch foo. To switch to that branch, type git-checkout foo at the Terminal prompt. Git updates your working copy with the latest files from branch foo. And it moves the HEAD pointer to that branch. Now open Foobar.text and add the following line.

```
Hickory Dickory Dock
```

Save your changes and commit them by typing git-commit -m "Revision 1 at branch foo" Foobar.text. After this, your repository takes the form in Figure 11.

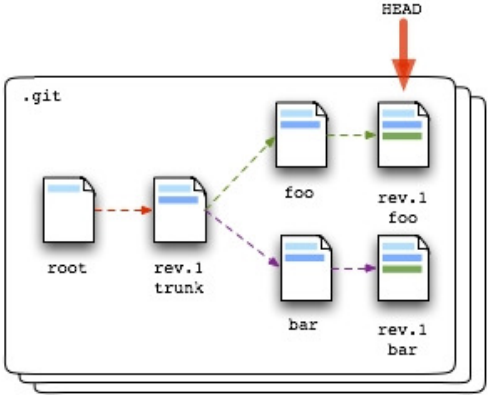

Figure 11. Revised branch foo.

Now switch to the trunk by typing git-checkout master at the prompt. Here, master is a reserved tag for the trunk. Again, Git updates your working copy with files from the trunk, and places the HEAD pointer at that trunk. Open Foobar.text and insert the following as the second line.

```
Dickery Dickery Dare
```

Save your changes and type git-checkout foo at the prompt. This time, Git aborts the switch telling you that it is unable to merge the revised Foobar.text with the one in branch foo. This is because the revised file belongs to the trunk, not to branch foo. To force the switch, type git-checkout -f foo Foobar.text. The -f option tells Git to dispose all changes to Foobar.text before switching to branch foo. The repository remains unchanged after the switch.

Viewing branches

Next, suppose you want to examine your branches. To get a list of branches, type git-branch without any arguments. Git responds by returning a list like the one shown in Listing 2. Notice the asterisk next to one of the branch names. This asterisk marks the active branch, which is branch foo in this case. So, if you type git-checkout bar and then git-branch, you will find the same asterisk next to branch bar.

Listing 2. A sample list of branches.

```
git-branch
  bar
* foo
  master
```

To get details about a specific branch, for example foo, type git-show foo at the prompt. Git returns the details similar to that shown in Listing 3. Here, the first line shows the SHA1 key of the last valid committal. The next two lines are the user who performs the committal and when he did the committal. The third line is the latest committal message. The rest shows the differences between the specified branch and the trunk.

Listing 3. Sample branch details.

```
git-show foo
commit cdbdc1de9be1c69815ef2d18d5a685ec2336adeb
Author: JSmith <j.smith@mail.box.com>
Date:   Thu Feb 1 12:05:14 2009 -0800
    Revision 1 at branch foo
diff --git a/Foobar.text b/Foobar.text
index 1882332..c7fba90 100644
--- a/Foobar.text
+++ b/Foobar.text
@@ -1,3 +1,4 @@
 <EF><BB><BF>Pease porridge hot, pease porridge cold,
 Pease porridge in the pot, nine days old;
-Some like it hot, some like it cold,
\ No newline at end of file
+Some like it hot, some like it cold,
+Hickory Dickory Dock.
\ No newline at end of file
```

To view the revision log for a file in each branch, first use git-checkout to switch to that branch. Then use git-log, passing along the file's name as the argument. For example, the following statements gives the log for Foobar.text on branch foo.

```
git-checkout foo; git-log Foobar.text
```

Git returns all the revision entries for Foobar.text, from the root up to branch foo. To view the revision log for the entire branch, type git-log and pass the branch's name as the argument.

```
git-log bar
```

Again, Git returns all the revision entries from the root to the branch. Unlike the previous statement, you need not switch to the branch in question.

Deleting branches

Suppose you want to remove a specific branch from the repository. To do this, use a -d option with your git-branch statement. First, we need a branch that we can remove. Make the trunk active by typing git-checkout master at the Terminal prompt. Create a temp branch with git-branch temp. The repository now appears as shown in Figure 12.

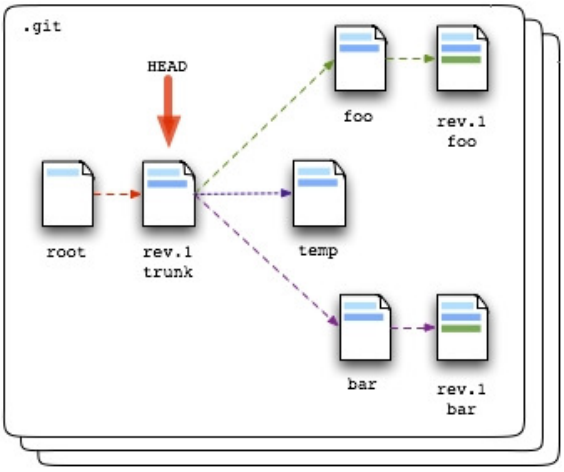

Figure 12. Added branch temp.

While still at the trunk, type git-branch -d temp. Git responds by quietly deleting branch temp, including all the files stored in that branch.

Let us try something different. Recreate branch temp by typing git-checkout -b temp. Open Foobar.text and add the following line to that file.

```
THIS FILE IS DUE FOR DELETION
```

Save your changes, and then commit them with the statement git-commit -m "Revision 1 at branch temp" Foobar.text. The repository now has the form shown in Figure 13. While still in branch temp, type git branch -d temp at the Terminal prompt. Git responds with an error message stating that you cannot delete an active branch.

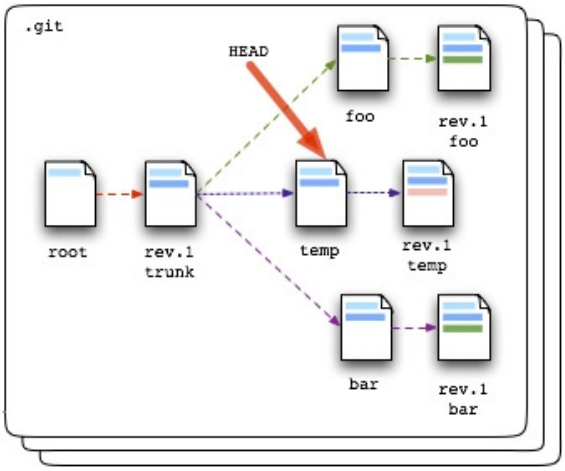

Figure 13. Revising branch temp.

Switch to branch foo by typing git-checkout foo at the Terminal prompt. And then type git branch -d temp. Now Git responds with a new error message. This time, the message states that branch temp is not a strict subset of branch foo. This is not surprising since temp came off the project trunk, not off of branch foo.

Switch back to the trunk by typing git-checkout master at the prompt. Then type git-branch -d temp. Again, Git responds with an error message stating that temp is not a strict subset of the trunk. But this time, it is because temp has at least one revision of Foobar.text. To actually delete branch temp, type git-branch -D temp (note the uppercase letter). Git will then remove branch temp and its files as expected.

These last three attempts show Git protecting your revisions from reckless deletions. It displays messages that describe why it is unable to delete the specified branch. Of course, you can force the deletion as shown above. Keep in mind, though, that deleting a branch is a permanent action; so proceed with caution.

Managing With Merges

Now that you have one or more branches in your project, you may choose to merge one branch with another branch or with the trunk itself. Merging a branch has three basic steps. In the first step, you examine the differences between two branches, or between the branch and the trunk. The second step is the actual merge itself. Third, you handle any conflicts that appear during the merge.

So, for the rest this article, we will learn how to do a basic merge using Git.

Finding the differences

To compare one branch with another, use the git-diff command. This command takes up to two arguments, which are listed below.

```
git-diff source-branch [target-branch]
```

The argument source-branch is the branch you want to compare, while target-branch is the branch you want to compare with. If target-branch is blank, Git will use the active branch as its target.

Start by typing git-diff foo foo at the Terminal prompt. Git responds with a null result. This is not surprising, of course, as you are comparing a branch with itself. Now, type git-diff foo master at the prompt. This time, Git responds with the results shown in Listing 4.

Listing 4. Sample diff output.

```
git-diff foo master
diff --git a/Foobar.text b/Foobar.text
index c7fba90..1882332 100644
--- a/Foobar.text
+++ b/Foobar.text
@@ -1,4 +1,3 @@
 <EF><BB><BF>Pease porridge hot, pease porridge cold,
 Pease porridge in the pot, nine days old;
-Some like it hot, some like it cold,
-Hickory Dickory Dock.
\ No newline at end of file
+Some like it hot, some like it cold,
\ No newline at end of file
```

The first line shows the two files compared by the command. Here, the source file has an a/ prefix, the target a b/ prefix. The second line shows the hash indices of those two files. The next three lines are the standard header for a unified diff. The remaining lines show how the two files will change during the merge. The sample above shows that Git will remove two lines from the source file. Then it will copy one line from the target file into the source file.

Using git-diff will help locate any possible conflicts between the branches and the trunk. Keep in mind, however, that git-diff works only with files stored in the repository. Changes in your working copy of those files will be ignored.

Merging a branch

Merging a branch is done with the git-merge command. This command takes two arguments as shown below.

```
git-merge source-branch [target-branch]
```

Here, the argument source-branch provides the data for the merge, while target-branch serves as the "container" for the merge. Thus, any file revised by the merge goes straight into target-branch (Figure 14). Again, if target-branch is blank, Git assumes the active branch in its place. Git also updates your working copy with files revised by the merge. And it moves the HEAD pointer to the revised target-branch.

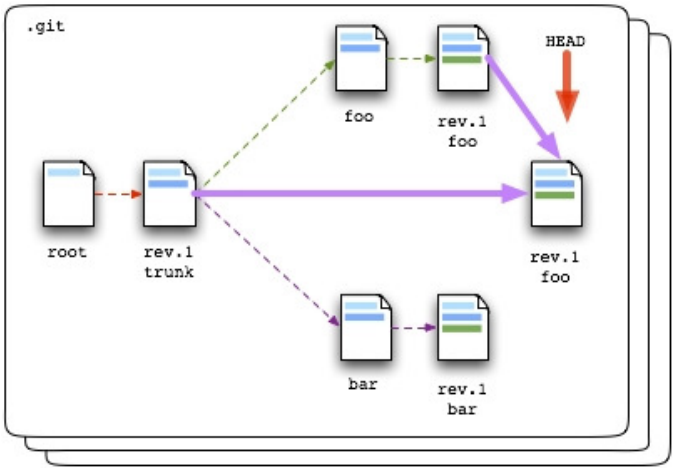

Figure 14. The merge process.

Suppose you want to merge branch foo with the trunk. To do this, first switch to the trunk by typing git-checkout master at the Terminal prompt. Then type git-merge foo at Terminal prompt. Git first compares the file Foobar.text in branch foo with the one in the trunk. When finds no conflicts, Git combines the two files and commits the merge file into the trunk. And it returns its merge results (Listing 5) back to the Terminal window.

Listing 5. Sample results of a merge.

```
git-merge foo
Updating b553a1b..cdbdc1d
Fast forward
 Foobar.text |    3 ++-
 1 files changed, 2 insertions(+), 1 deletions(-)
```

The above results show the hash keys of the two files being merged. Then, it shows the name of revised file, and the number of lines added or removed from that file. Finally, it gives a tally of all the changes made by the merge.

To undo the merge, type git-reset HEAD^ at the Terminal prompt. Git then moves the HEAD pointer to the last state before the merge (Figure 15). The revised branch remains in the repository, its links to the trunk now broken. But Git does not update your working copy, which still has the results of the merge. To force an update, type git-checkout -f target-branch, where target-branch is the branch affected by the merge.

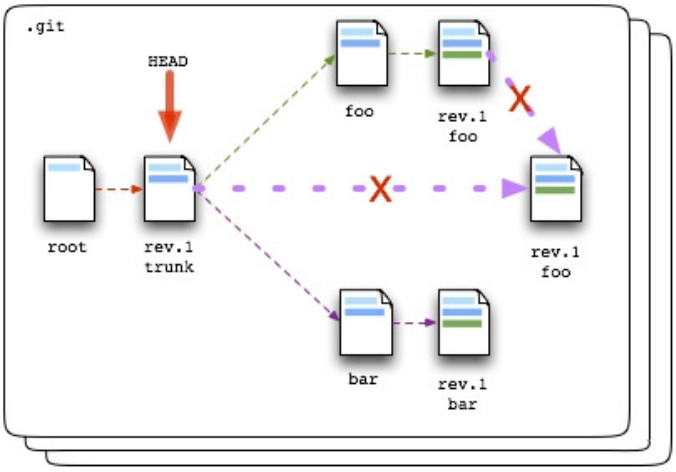

Figure 15. Undoing a merge.

Dealing with conflicts

Now suppose you typed git-merge foo bar at the Terminal prompt. This tells Git to merge the files in branch foo with those in branch bar. This time, Git displays a different set of results to the Terminal window (Listing 6). These results tell you that the files in foo and bar has conflicting entries

Listing 6. Sample output of a merge conflict.

```
git-merge foo bar
Trying simple merge with cdbdc1de9be1c69815ef2d18d5a685ec2336adeb
Trying simple merge with 4de4b56e7b3c16bf4d83e090cefa2418f4338dc1
Simple merge did not work, trying automatic merge.
Auto-merging Foobar.text
ERROR: Merge conflict in Foobar.text
fatal: merge program failed
Automatic merge failed; fix conflicts and then commit the result.
```

Open the file Foobar.text with a text editor. You will find that Git divided the conflicting lines into two sets, and enclosed both sets inside a '<<<<<<<...>>>>>>>' block (Listing 7). The first set consists of lines from branch foo, the second set from branch bar. Lines outside the '<<<<<<<...>>>>>>>' block are shared by both branches.

Listing 7. Conflicting entries in Foobar.text.

```
Pease porridge hot, pease porridge cold,
Pease porridge in the pot, nine days old;
Some like it hot, some like it cold,
<<<<<<< .merge_file_9fiCGO
Hickory Dickory Dock.
=======
Some like it in the pot, nine days old.
>>>>>>> .merge_file_InoRhg
```

Try typing git-checkout foo at the Terminal prompt. You will find that Git prevents you from working with either branches until you resolve the conflicts. Now, type git-status at the prompt. Git then informs you that you have two copies of Foobar.text awaiting your actions. One copy is labeled unmerge, the other modified.

To resolve the conflict, modify Foobar.text as shown in Listing 8. At the Terminal prompt, type git-commit -a Foobar.text. Git responds by displaying its default message for the committal. You can either accept this message or edit it to your taste. Save the commit message to continue the committal. If you type git-status at the Terminal prompt, you will find no pending revisions of your project files.

Listing 8. Foobar.text after resolution.

```
Pease porridge hot, pease porridge cold,
Pease porridge in the pot, nine days old;
Some like it hot, some like it cold,
Some like it in the pot, nine days old.
```

Closing Remarks

Branches are important parts of the project cycle. With branches, we can isolate unplanned changes in a separate branch, protecting the main project from their effects. If those changes pass our tests, we can combine that branch back into the project itself. If they fail, we leave the branch alone for historical reasons. We can also use branches to work on multiple variants of the same project in parallel.

Git gives us the means to create, manage, and merge branches in a given project. It lets us find out which branch is active, and switch from one branch to another. It allows us to compare two branches, as well as merge them together. It even tracks all changes made in each branch. And it lets us resolve any merge conflicts quickly and simply.

In the next Git article, we will learn how to customize Git to suit our project needs. So, stay tuned to this spot and thanks for reading.

Bibliography and References

A. King, et al. "Basic Branching and Merging." . The Git-SCM Project. 2008. pp 26-31. Available: http://book.git-scm.com/

J.C. Cruz. "Getting Started With Git." vol 24, no. 02, 2008 Feb. Xplain Corporation, 2008.

J.C. Cruz. "Sharing With Git." vol 24, no. 03, 2008 Mar. Xplain Corporation, 2008.

B. Lynn. "Branch Wizardry." . Standford University, 1984-2007. pp 13-18. Available: http://www-cs-students.stanford.edu/~blynn/gitmagic/index.html

---

JC is a freelance engineering writer from North Vancouver, British Columbia. He spends

his time writing technical articles; tinkering with Cocoa, REALbasic, and Python; and visiting his foster nephew. He can be reached at anarakisware@gmail.com.