OReilly Version Control with GIT Book Survey

Chapter 1 (Min Zheng)

Chapter 2

Chapter 3 (Suzy)

Chapter 4

Chapter 5

Chapter 6

**Chapter 7. Branches(Chen Zhang)**

In chapter 7, we learnt about branches in Git. Branching in Git is kinda like parallel universe in science fiction movies. While we were learning Git in front of the laptop hard in one universe, maybe the us in another universe were learning SVN :) If both universes do not disturb/affect each other, then at some point of time, the two universes merge, we would end up knowing both Git and SVN!

So how do we use Git branches in non-science fiction real world? Suppose we are about to develop a new functionality, but it takes two weeks to finish. On the first week we finished 50% of the coding, if we commit/check in the code immediately, since the coding is not finished, the incomplete code will cause other developer to not be able to work after our commit, but if we check in the code after finish, then we will risk losing the progress we made everyday.

But great news, now we have Git branch! If we create a personal branch, it is not visible to others, they will be working on the normal branch, meanwhile we can work on our branch, we can commit whenever we want, and after the development finishes, we can do an one-time merge to the original branch, in this manner, it is safe, we have the version control of all the changes, as well as safe, we do not risk losing our work, and above all, we are not affecting other’s work.

Some other version control systems all have branch managements, but after actually using them, it turns out branching in them are very slow, it goes to an extend that people stop using them, it becomes a redundant feature.

Branching in Git is unique, no matter if you are branching, switching between branches or deleting a branch, Git will handle it in one second. It doesn’t matter if your repository has one file or ten thousand!

We already knew that after every commit, Git will chain them all into one timeline, this timeline is a branch itself. Up till now, there is only one single timeline, in Git, this branch is called master branch. Theoretically speaking, HEAD does not point to the committed node, it points to master, and master points to commit, so HEAD is pointing to the current branch.

At the beginning of time, master is just one single timeline, Git use master to point to the latest commit, then uses HEAD to point to master, this way we know which is the current branch and the latest commit of the current branch.



With every commit, master branch moves forward one step, in this manner, the more commits we make, the longer the master branch.

When we create new branches, like this branch dev, Git creates a new pointer called dev, it points to the same commit of the master, then Git points the HEAD to dev, it means we are current on the branch dev.



So here we can understand why git branching is fast, because the workspace is not changed at all except for changing the HEAD pointer and creating a new pointer called dev.

But starting now, the modification we make will be saved for dev branch, for example, after every new commit, dev pointer moves forward one step, but the master pointer doesn’t change:



Suppose our work is done on the dev branch, we can merge it to master. So how does Git handle the merge? The simplest way is to point master to the current commit of dev, this will complete the merge.



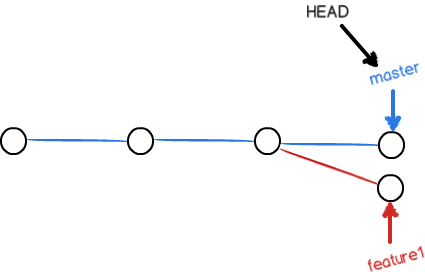
That’s why git merges are also fast, it’s just the change of pointer, no changes made to the workspace whatsoever.

When the merging is done, we can even delete dev branch. Deleting a branch will only remove the dev pointer, after the deletion, we have only one master left.

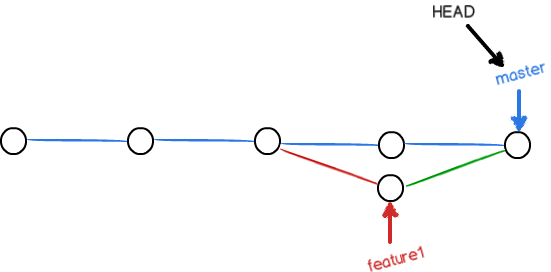


At this point, we cannot even see that some of the commits are from branches.

But merging branches are not alway easy to do, sometimes conflicts happen while merging. Supposingly, we have a new feature1 branch, and development happens on this branch. After committing, we switch to master, Git will inform us feature1 is ahead of master by how many commits. But if we work on the master branch and happen to edit the same line of the same file with the changes we did on feature1 and commit:



In this case, if we try to merge the feature1 branch to master, Git cannot perform a fast-forward merge, it will try to merge the changes we made for the two branches and, unfortunately, conflicts will happen.

Git will use <<<<<<<<<, =========, >>>>>>>>> to markup the different content of the branches, we can manually choose what we want for that particular line in the particular file, remove the Git conflict notations and commit the file, so the conflict will be resolved. Now the branches will look like this:  


We can also use Git log command to see the status of the branches：

$ git log --graph --pretty=oneline --abbrev-commit

Finally we can safely delete feature1 and the workflow is finished.

In actual software development, we can not alway keep on working in one single task until it is done, sometimes there will be something like emergency bug fixes. Thank to Git’s powerful branching system, each and every bugfix development can happen on a temporary new branch, after the fix is done, we can merge it to master, then delete the temporary branch.

When we are ready to work on a bug, say 101, naturally we would want to create a branch issue-101 for the fix, but wait, the current development on dev is not done and we are not ready to commit the current changes.

$ git status  
*# On branch dev*  
*# Changes to be committed:*  
*# (use "git reset HEAD <file>..." to unstage)*  
*#*  
*# new file: hello.py*  
*#*  
*# Changes not staged for commit:*  
*# (use "git add <file>..." to update what will be committed)*  
*# (use "git checkout -- <file>..." to discard changes in working directory)*  
*#*  
*# modified: readme.txt*  
*#*  
Luckily, Git has a stash feature, we can “stash” the current workspace, and recover it later when we are done with the emergency fix.

$ git stash  
Saved working directory **and** index state WIP on dev: 6224937 add merge  
HEAD is now at 6224937 add merge  
Now the workspace is clean, we can safely create a bug fix branch and work on it.

$ git checkout master  
Switched to branch 'master'  
Your branch is ahead of 'origin/master' by 6 commits.  
$ git checkout -b issue-101  
Switched to a new branch 'issue-101'  
After fixing the issue and committing it to issue-101, we can switch to master and merge the changes in issue-101 to master then delete the temporary branch.

$ git checkout master  
Switched to branch 'master'  
Your branch is ahead of 'origin/master' by 2 commits.  
$ git merge --no-ff -m "merged bug fix 101" issue-101  
Merge made by the 'recursive' strategy.  
 readme.txt | 2 +-  
 1 file changed, 1 insertion(+), 1 deletion(-)  
$ git branch -d issue-101  
Deleted branch issue-101 (was cc17032).  
Then we can check out to the dev branch and keep on our work.

$ git checkout dev  
Switched to branch 'dev'  
$ git status  
*# On branch dev*  
nothing to commit (working directory clean)

Workspace is clean, where are the stashed changes we made, we can check them by git stash list.

$ git stash list  
stash@{0}: WIP on dev: 6224937 add merge  
There are two ways of recover the stashed changes, one is git stash apply, apply doesn’t delete the content in stashing stage, we can later delete is using git stash drop, the other way is to use git stash pop, this will recover the workspace and delete the stashing stage content.

$ git stash pop  
*# On branch dev*  
*# Changes to be committed:*  
*# (use "git reset HEAD <file>..." to unstage)*  
*#*  
*# new file: hello.py*  
*#*  
*# Changes not staged for commit:*  
*# (use "git add <file>..." to update what will be committed)*  
*# (use "git checkout -- <file>..." to discard changes in working directory)*  
*#*  
*# modified: readme.txt*  
*#*  
Dropped refs/stash@{0} (f624f8e5f082f2df2bed8a4e09c12fd2943bdd40)

We can also stash multiple times and while recovering, we can point to a particular stashed changeset.

$ git stash list  
stash@{0}: WIP on dev: 6224937 add merge

$ git stash apply stash@{0}