Git Branching

Nearly every VCS has some form of branching support. Branching means you diverge from the main line of development and continue to do work without messing with that main line. In many VCS tools, this is a somewhat expensive process, often requiring you to create a new copy of your source code directory, which can take a long time for large projects.

Some people refer to Git's branching model as its "killer feature," and it certainly sets Git apart in the VCS community. Why is it so special? The way Git branches is incredibly lightweight, making branching operations nearly instantaneous, and switching back and forth between branches generally just as fast. Unlike many other VCSs, Git encourages workflows that branch and merge often, even multiple times in a day. Understanding and mastering this feature gives you a powerful and unique tool and can entirely change the way that you develop.

Branches in a Nutshell

To really understand the way Git does branching, we need to take a step back and examine how Git stores its data.

As you may remember from Getting Started, Git doesn't store data as a series of changesets or differences, but instead as a series of snapshots.

When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged. This object also contains the author's name and email, the message that you typed, and pointers to the commit or commits that directly came before this commit (its parent or parents): zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.

To visualize this, let's assume that you have a directory containing three files, and you stage them all and commit. Staging the files computes a checksum for each one (the SHA-1 hash we mentioned in Getting Started), stores that version of the file in the Git repository (Git refers to them as blobs), and adds that checksum to the staging area:

```
$ git add README test.rb LICENSE
$ git commit -m 'The initial commit of my project'
```

When you create the commit by running git commit, Git checksums each subdirectory (in this case, just the root project directory) and stores those tree objects in the Git repository. Git then creates a commit object that has the metadata and a pointer to the root project tree so it can re-create that snapshot when needed.

Your Git repository now contains five objects: one blob for the contents of each of your three files, one tree that lists the contents of the directory and specifies which file names are stored as which blobs, and one commit with the pointer to that root tree and all the commit metadata.

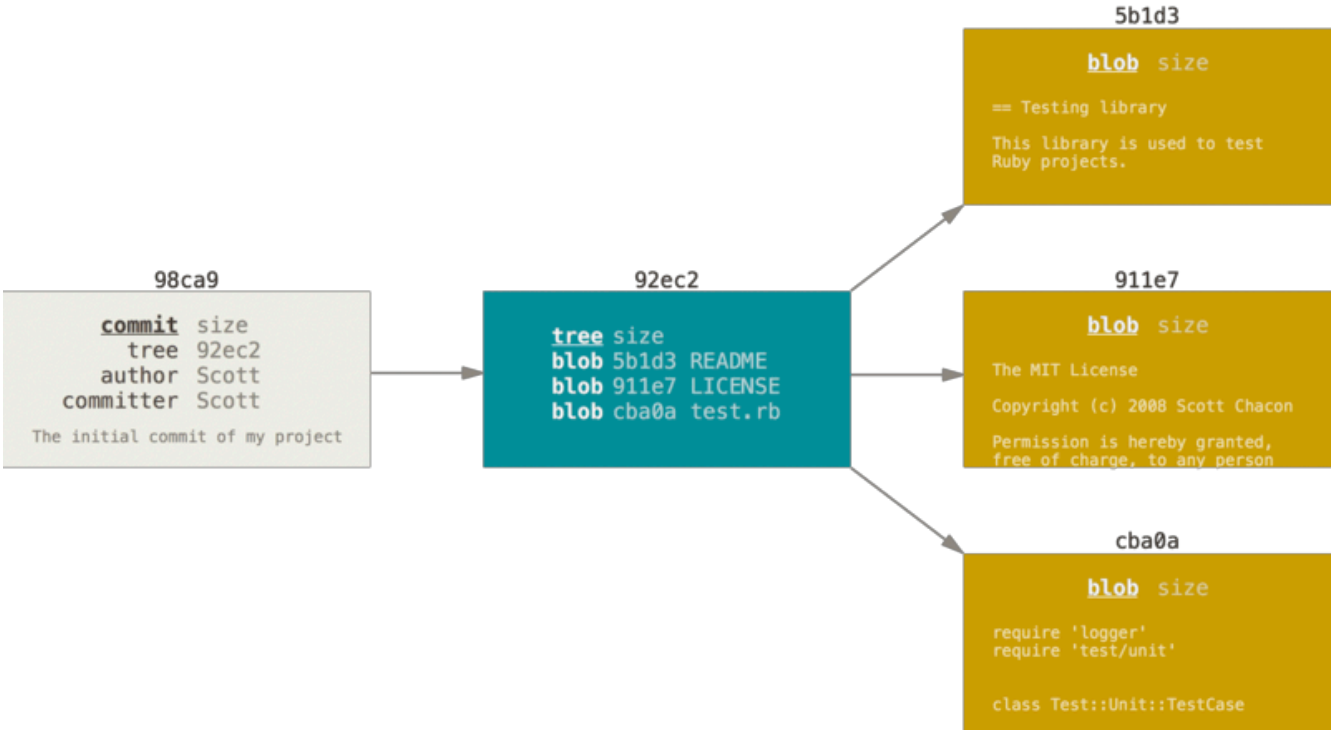


Figure 9. A commit and its tree

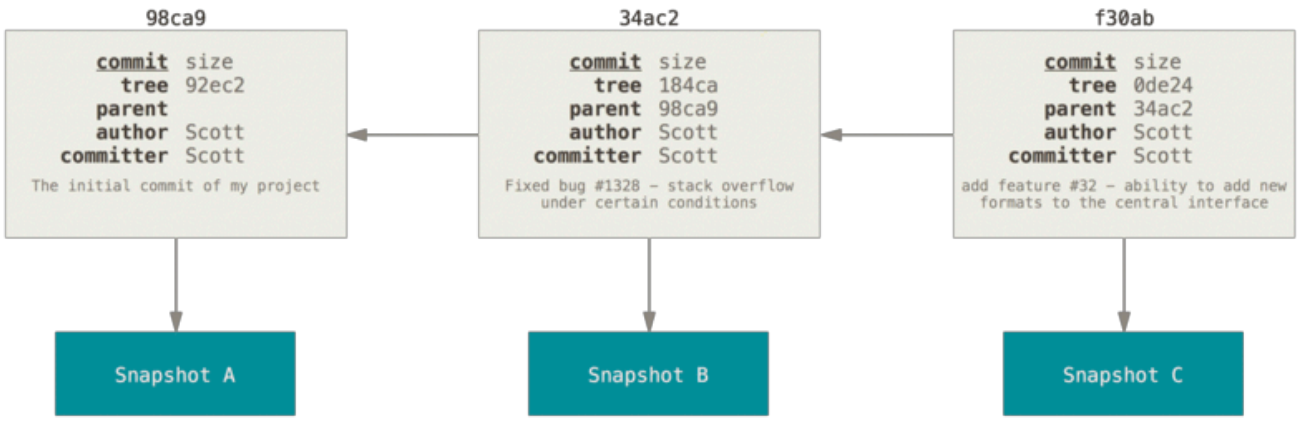
If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.

Figure 10. Commits and their parents

A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is master. As you start making commits, you're given a master branch that points to the last commit you made. Every time you commit, it moves forward automatically.

The "master" branch in Git is not a special branch. It is exactly like any other **NOTE** branch. The only reason nearly every repository has one is that the git init

command creates it by default and most people don't bother to change it.



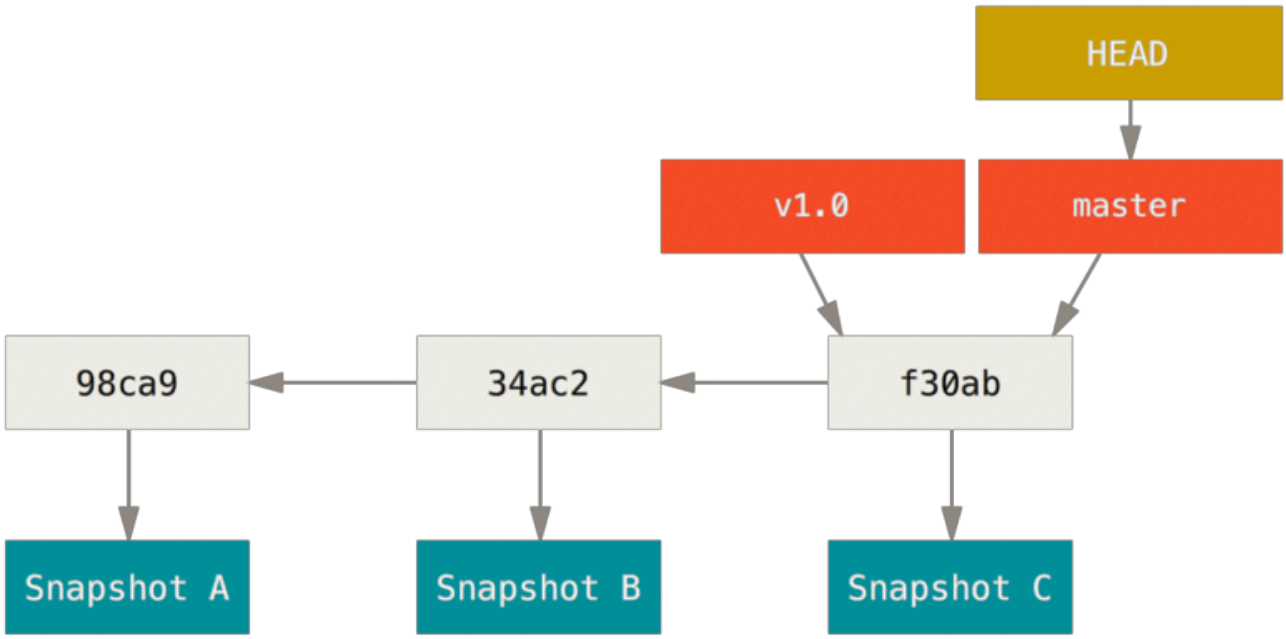


Figure 11. A branch and its commit history

Creating a New Branch

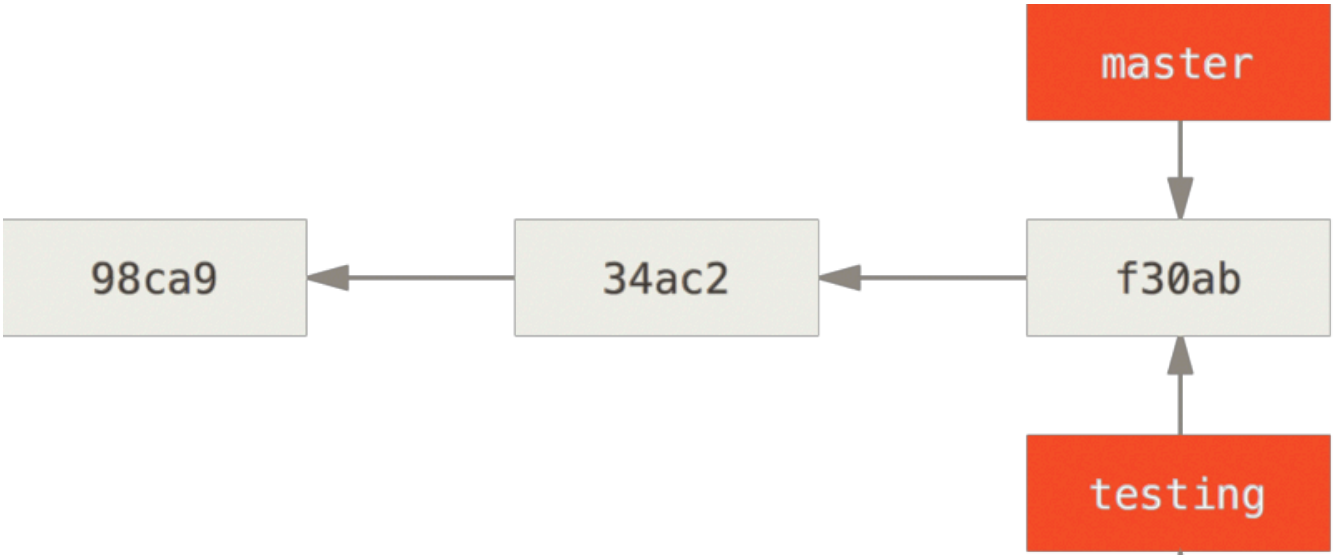
What happens if you create a new branch? Well, doing so creates a new pointer for you to move around. Let's say you create a new branch called testing. You do this with the git branch command:

```
$ git branch testing
```

This creates a new pointer to the same commit you're currently on.

Figure 12. Two branches pointing into the same series of commits

How does Git know what branch you're currently on? It keeps a special pointer called HEAD. Note that this is a lot different than the concept of HEAD in other VCSs you may be used to, such as Subversion or CVS. In Git, this is a pointer to the local branch you're currently on. In this case, you're still on master. The git branch command only *created* a new branch — it didn't switch to that



branch.

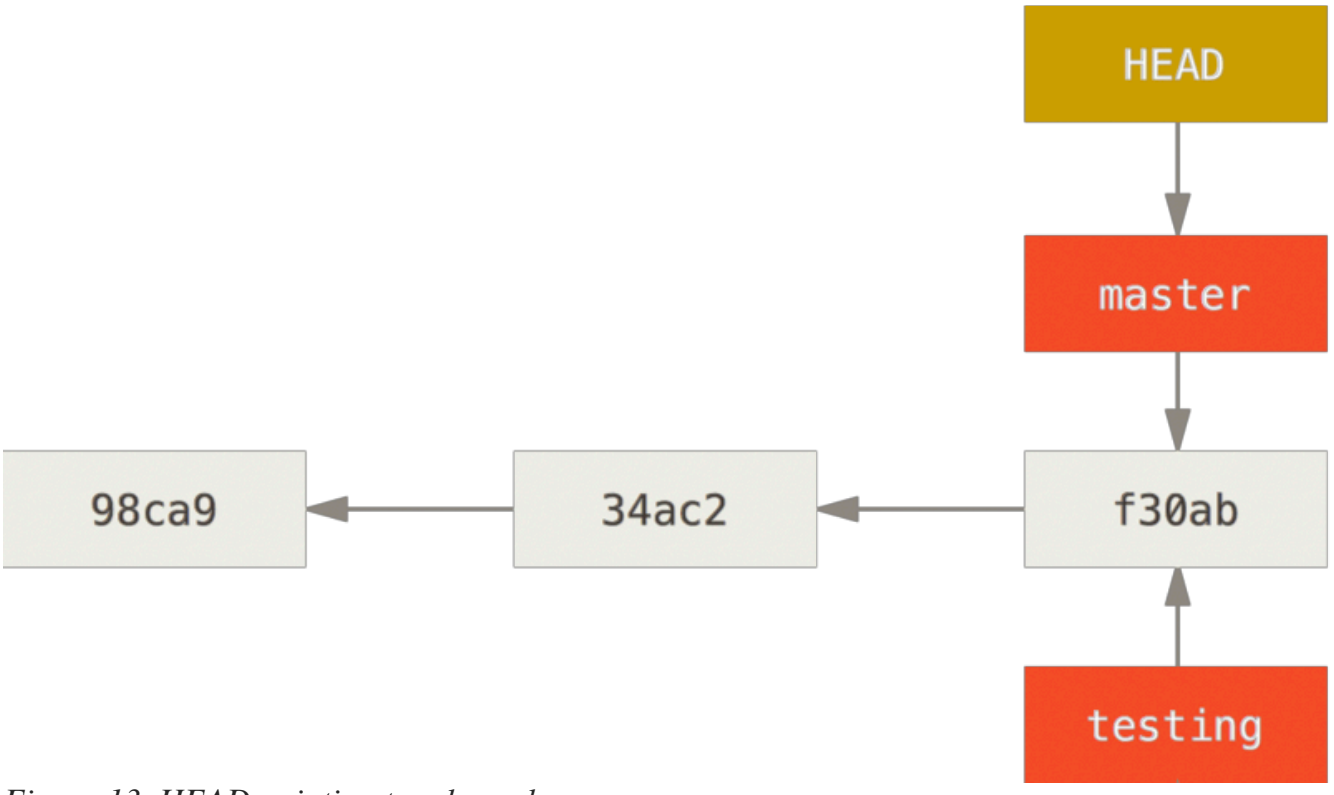


Figure 13. HEAD pointing to a branch

You can easily see this by running a simple git log command that shows you where the branch pointers are pointing. This option is called --decorate.

```
$ git log --oneline --decorate
```

f30ab (HEAD -> master, testing) add feature #32 - ability to add new formats to the
central interface
34ac2 Fixed bug #1328 - stack overflow under certain conditions
98ca9 The initial commit of my project

You can see the “master” and “testing” branches that are right there next to the f30ab commit.