Understanding Git: Collaborating

In This Section

Working with Other Repositories

A key feature of Git is that the repository is stored alongside the working copies of the files themselves. This has the advantage that the repository stores the entire history of the project, and Git can function without needing to connect to an external server.

However, this means that, in order to manipulate the repository, you need to also have access to the working files. This means that two Git developers cannot, by default, share a repository.

To share work among developers, Git uses a **distributed model** of version control. It assumes no central repository. It is possible, of course, to use one repository as the "central" one, but it is important to understand the distributed model first.

Distributed Version Control

Say you and your friend want to work on the same paper. Your friend already has done some work on it. There are three tasks you need to figure out to do so:

1. How to get from your friend a copy of the work to date.
2. How to get the changes your friend makes into your own repository.
3. How to let your friend know about changes you make.

Git provides a number of transport protocols for sharing repository information, such as SSH and HTTP. The easiest (which you can use for testing) is simply accessing both repositories on the same filesystem.

Each protocol is identified by a “remote-specification.” For repositories on the same filesystem, the remote-specification is simply the path to the other repository.

Copying the Repository

To make a copy of your friend's repository for your own use, run `git clone [remote-specification]`.

The *[remote-specification]* identifies the location of your friend's repository, and it can be as simple as another folder on the filesystem. Where your friend's repository is accessible by a network protocol such as ssh, Git refers to the repository by a URL-like name, such as `ssh://server/repository`.

For example, if your friend's repository is located in `~/jdoe/project`, you would run:

```
   git clone ~/jdoe/project
```

This would do the following:

- Create a directory *project* and initialize a repository in it.
- Copy all the commit objects from the project into the new repository.
- Add a **remote repository reference** named *origin* to the new repository, and associate *origin* with `~/jdoe/project` as described below. (Like *master*, *origin* is a default name used by Git.)
- Add **remote heads** named *origin/[head-name]* that correspond to the heads in the remote repository.
- Set up one head in the repository to **track** the corresponding *origin/[current-head-name]* head, namely the one that was currently active in the repository being cloned.

A **remote repository reference** is a name Git uses to refer to the remote repository. Generally it will be *origin*. Among other things, Git internally associates the *remote-specification* with the remote repository reference, so you never need to refer to the original repository again.

A branch that **tracks** a remote branch retains an internal reference to the remote branch. This is a convenience that allows you to avoid typing the name of the remote branch in many situations, as will be described below.

The important thing to note is that you now have a complete copy of your friend's entire repository. When you branch, commit, merge, or otherwise operate on the repository, you operate only on your own repository. Git only interacts with your friend's repository when you specifically ask it to do so.

Say your friend's repository looked like this:

```
                   +---------------(E)
                  /                 |
   (A) -- (B) -- (C) -- (D)         |
                         |          |
                       master    feature
                         |
                        HEAD
```

After cloning, your repository would look like:

```
                   +-------------- (E)
                  /                 |
   (A) -- (B) -- (C) -- (D)         |
                         |          |
        origin/master, master    origin/feature
                         |
                        HEAD
```

If you would like to work with the *origin/feature* remote branch locally, you should manually set up a tracking branch. This is done with the following command:

```
   git branch --track [new-local-branch] [remote-branch]
```

In our example, that command would be `git branch --track feature origin/feature`. Note that the `--track` option is on by default and thus redundant (but I like to keep it in anyway, for clarity).

Receiving Changes from the Remote Repository

After you have cloned the repository, your friend adds new commits to his repository. How do you get copies of those changes?

The command `git fetch [remote-repository-reference]` retrieves the new commit objects in your friend's repository and creates and/or updates the remote heads accordingly. By default, *[remote-repository-reference]* is *origin*.

Say your friend's repository now looks like this:

```
                   +-------- (E) -- (F) -- (G)
                  /                         |
   (A) -- (B) -- (C) -- (D) -- (H)          |
                                |           |
                              master     feature
                                |
                               HEAD
```

After a `git fetch`, your repository would look like this:

```
                   +------------ (E) ------------ (F) ---- (G)
                  /               |                         |
   (A) -- (B) -- (C) -- (D) --------------- (H)             |
                         |        |          |              |
                       master  feature origin/master   origin/feature
                         |
                        HEAD
```

Note that your own heads are unaffected. The only difference is that the remote heads, those starting with “origin/”, are updated, and the new commit objects have been added to the repository.

Now, you want to update your *master* and *feature* heads to reflect your friend's changes. This is done with a merge, usually done with `git pull`. The general syntax is:

```
git pull [remote-repository-reference] [remote-head-name]
```

This will merge the head named *[remote-repository-reference]/[remote-head-name]* into *HEAD*.

Git provides two features to make this simpler. First, if a head is set up with tracking, a simple `git pull` with no arguments will merge the correct remote head. Second, `git pull` will perform a `fetch` automatically, so you rarely need to call `git fetch` yourself.

Thus, in the above example, if you called `git pull` on your repository, your *master* head would be moved forward to commit (H):

```
                +-------- (E) ------------- (F) ----- (G)
               /           |                           |
(A) -- (B) -- (C) -- (D) ------------ (H)              |
                           |           |               |
                        feature  origin/master,  origin/feature
                                     master
                                       |
                                      HEAD
```

Sending Changes to the Remote Repository

Now, say that you make changes to your own repository. You want to send these changes to your friend's repository.

The command `git push`, logically doing the opposite of `pull`, sends data to the remote server. Its full syntax is:

```
git push [remote-repository-reference] [remote-head-name]
```

When this command is called, Git directs the remote repository to do the following:

- Add new commit objects sent by the pushing repository.
- Set *[remote-head-name]* to point to the same commit that it points to on the pushing repository.

On the sending repository, Git also updates the corresponding remote head reference.

If no arguments are given to `git push`, it will push all the branches in the repository that are set up for tracking.

Pushing and Fast Forward Merges

Git requires that **the push result in a fast-forward merge on the remote repository**. That means that, before the merge, the remote head points to an ancestor of the commit that it will point to after the merge. If this is not the case, Git will complain, and this complaint should be taken seriously.

The reason for this is as follows. Each branch should contain an incrementally improving version of the project. A fast forward merge always indicates simple improvement on a branch, as the head is only pushed forward and not shifted to a different place on the tree. If the head moves to a different place on the tree, then information on how the branch arrived at its most recent state is lost.

For example, imagine that the remote repository looks like this:

```
(A) -- (B) -- (C)
               |
             master
```

And your cloned repository is changed to look like this:

```
         +------------- (D)
        /                |
(A) -- (B) -- (C)        |
               |         |
             other     master
```

Now, if you push *master* to the remote repository, it will look like this:

```
         +------------- (D)
        /                |
(A) -- (B) -- (C)        |
                         |
                       master
```

Now (C) is a dangling commit. Moreover, the *master* branch has lost all record that it used to point to (C). Thus, (C) is no longer part of the history of *master*.

Relatedly, what happens when you attempt to `pull` changes from the remote repository, which do not result in a fast forward merge? This can happen, for example, when you make changes on a branch in your local repository and branch changes on the remote repository as well. Consider this situation:

```
                                     +-- (D)
                                    /     |
(A) -- (B) -- (C)           (A) -- (B)    |
               |                          |
             master                     master

Your repository             Remote repository
```

This may arise if, say, you last pulled from the repository when `master` was at commit (B), you did work and advanced `master` to commit (C), and another developer, working on the remote repository, did different work to advance `master` to (D).

Now, you pull `master` from the remote repository. What should happen? It was once the case that Git would produce an error, but now it does something smart: it merges the two changes into a new commit, and moves `master` to that new merge commit.

```
         +--- (D) ---+
        /             \
(A) -- (B) -- (C) -- (E)
                      |
                    master
```

Alternately, you can use `rebasing`, explained on the next page, to rewrite your commit (C). However, the best approach is to avoid this problem by placing all of your own work on separate branches until you are ready to incorporate them, and then do the incorporation quickly before anyone else has a chance to update the remote repository.

Adding and Deleting Branches Remotely

To create a new branch on the remote repository, use the following commands, while the new branch is your current head. Let the remote repository reference be *origin*, and the current branch name be *new-branch*.

```
git push origin new-branch
git checkout [some-other-branch]
git branch -f new-branch origin/new-branch
git checkout new-branch
```

The first command will first create the appropriate head in the remote repository and push the new branch commits out. The remaining three are used to recreate the new branch to track the remote branch.

As of Git 1.7, you can perform the above procedure using an option to `git push`:

```
git push --set-upstream origin new-branch
```

There are also options `--track` and `--set-upstream` for `git branch` to achieve similar effects.

To delete a branch on the remote repository, use the following command:

```
git push [remote-repository-reference] :[head-name]
```

That is, put a colon before the head name, which means to push “no data” into that head on the remote repository.

To list the available remote branches, use `git branch -r`.

Git with a Central Repository

If you understand how Git transmits information between repositories using `push` and `pull`, then setting up a central repository for a project is simple. Developers simply do their own work and push and pull to the central repository.

When creating the central repository, you can use a “bare” repository. This is a repository that has no working files, only repository information. This is useful because a central repository shouldn't have anyone editing on it; it should only receive pushes and pulls.

Setting up a Git central repository that is accessible by remote protocol is tricky. The easiest way to do it is to use an existing system such as Gitosis or GitHub.

Go on to the next page: Rebasing