Understanding Git Conceptually

A Tutorial on Git

by Charles Duan

Originally published at http://www.sbf5.com/~cduan/technical/git/

Introduction

This is a tutorial on the Git version control system.

Git is quickly becoming one of the most popular version control systems in use. There are plenty of tutorials on Git already. How is this one different?

A Story

When I first started using Git, I read plenty of tutorials, as well as the user manual. Though I picked up the basic usage patterns and commands, I never felt like I grasped what was going on “under the hood,” so to speak. Frequently this resulted in cryptic error messages, caused by my random guessing at the right command to use at a given time. These difficulties worsened as I began to need more advanced (and less well documented) features.

After a few months, I started to understand those under-the-hood concepts. Once I did, suddenly everything made sense. I could understand the manual pages and perform all sorts of source control tasks. Everything that seemed so cryptic and obscure now was perfectly clear.

Understanding Git

The conclusion I draw from this is that **you can only really use Git if you understand how Git works.** Merely memorizing which commands you should run at what times will work in the short run, but it’s only a matter of time before you get stuck or, worse, break something.

Half of the existing resources on Git, unfortunately, take just that approach: they walk you through which commands to run when, and expect that you should do fine if you just mimic those commands. The other half does go through all the concepts, but from what I have seen, they explain Git in a manner that assumes you already understand how Git works.

This tutorial, then, will take a **conceptual approach** to Git. My goal will be, first and foremost, to explain the Git universe and its objectives, and secondarily to illustrate how to use Git commands to manipulate that universe.

- I will begin by describing the Git data model, the repository.
- From there I will describe the various operations Git provides for manipulating the repository, starting from the simplest (adding data to a repository) and moving through the more complex operations of branching and merging.
- I will then discuss how to use Git in a collaborative setting.
- Finally I will look at the Git rebase function, which provides an alternative to merging, and consider its pros and cons.

Go on to the next page: Repositories

Understanding Git: Repositories

In This Section

Repository Contents

The purpose of Git is to manage a project, or a set of files, as they change over time. Git stores this information in a data structure called a repository.

A git **repository** contains, among other things, the following:

- A set of **commit objects**.
- A set of references to commit objects, called **heads**.

The Git repository is stored in the same directory as the project itself, in a subdirectory called `.git`. Note differences from central-repository systems like CVS or Subversion:

- There is only one `.git` directory, in the root directory of the project.
- The repository is stored in files alongside the project. There is no central server repository.

Commit Objects

A **commit object** contains three things:

- A set of **files**, reflecting the state of a project at a given point in time.
- References to **parent commit objects**.
- An **SHA1 name**, a 40-character string that uniquely identifies the commit object. The name is composed of a hash of relevant aspects of the commit, so identical commits will always have the same name.

The parent commit objects are those commits that were edited to produce the subsequent state of the project. Generally a commit object will have one parent commit, because one generally takes a project in a given state, makes a few changes, and saves the new state of the project. The section below on merges explains how a commit object could have two or more parents.

A project always has one commit object with no parents. This is the first commit made to the project repository.

Based on the above, you can visualize a repository as a directed acyclic graph of commit objects, with pointers to parent commits always pointing backwards in time, ultimately to the first commit. Starting from any commit, you can walk along the tree by parent commits to see the history of changes that led to that commit.

The idea behind Git is that version control is all about manipulating this graph of commits. Whenever you want to perform some operation to query or manipulate the repository, you should be thinking, “how do I want to query or manipulate the graph of commits?”

Heads

A **head** is simply a reference to a commit object. Each head has a name. By default, there is a head in every repository called *master*. A repository can contain any number of heads. At any given time, one head is selected as the “current head.” This head is aliased to *HEAD*, always in capitals.

Note this difference: a “head” (lowercase) refers to any one of the named heads in the repository; “*HEAD*” (uppercase) refers exclusively to the currently active head. This distinction is used frequently in Git documentation. I also use the convention that names of heads, including *HEAD*, are set in italics.

A Simple Repository

To create a repository, create a directory for the project if it doesn’t exist, enter it, and run the command `git init`. The directory does not need to be empty.

```
mkdir [project]
cd [project]
git init
```

This will create a `.git` directory in the `[project]` directory.

To create a commit, you need to do two things:

1. Tell Git which files to include in the commit, with `git add`. If a file has not changed since the previous commit (the “parent” commit), Git will automatically include it in the commit you are about to perform. Thus, you only need to add files that you have added or modified. Note that it adds directories recursively, so `git add .` will add everything that has changed.
2. Call `git commit` to create the commit object. The new commit object will have the current *HEAD* as its parent (and then, after the commit is complete, *HEAD* will point to the new commit object).

As a shortcut, `git commit -a` will automatically add all modified files (but not new ones).

Note that if you modify a file but do not add it, then Git will include the previous version (before modifications) to the commit. The modified file will remain in place.

Say you create three commits this way. Your repository will look like this:

```
    ---->  time  ----->

    (A) <-- (B) <-- (C)
                     ^
                     |
                   master
                     ^
                     |
                    HEAD
```

where (A), (B), and (C) are the first, second, and third commits, respectively.

Other commands that are useful at this point:

- `git log` shows a log of all commits starting from *HEAD* back to the initial commit. (It can do more than that, of course.)
- `git status` shows which files have changed between the current project state and *HEAD*. Files are put in one of three categories: new files that haven't been added (with `git add`), modified files that haven't been added, and files that have been added.
- `git diff` shows the diff between *HEAD* and the current project state. With the `--cached` option it compares added files against *HEAD*; otherwise it compares files not yet added.
- `git mv` and `git rm` mark files to be moved (rename) and removed, respectively, much like `git add`.

My personal workflow usually looks like:

1. Do some programming.
2. `git status` to see what files I changed.
3. `git diff [file]` to see exactly what I modified.`
4. `git commit -a -m [message]` to commit.`

Referring to a Commit

Now that you've created commits, how do you refer to a specific commit? Git provides many ways to do so. Here are a few:

- By its SHA1 name, which you can get from `git log`.
- By the first few characters of its SHA1 name.

- By a head. For example, `HEAD` refers to the commit object referenced by *HEAD*. You can also use the name, such as `master`.
- Relative to a commit. Putting a caret (^) after a commit name retrieves the parent of that commit. For example, *HEAD^* is the parent of the current head commit.

Go on to the next page: Branching

Understanding Git: Branching

In This Section

The Purpose of Branching

Say you are working on a paper. You've gotten a first draft out, submitted for review. You then get a new batch of data, and you're in the process of integrating it into the paper. Halfway in, however, the review committee calls you up and tells you that you need to change some of your section headings to conform to format specifications. What do you do?

Obviously you don't want to send them your half-baked revisions with corrected headings. What you want to do is jump back to the version you sent out, change the headings on that version, and send off that copy, all the while keeping your recent work safely stored somewhere else.

This is the idea behind **branching,** and Git makes it easy to do.

Note on terminology: The terms "branch" and "head" are nearly synonymous in Git. Every branch is represented by one head, and every head represents one branch. Sometimes, "branch" will be used to refer to a head and the entire history of ancestor commits preceding that head, whereas "head" will be used to refer exclusively to a single commit object, the most recent commit in the branch.

Creating a Branch

To create a branch, say your repository looks like this:

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

where (B) was the version you sent to the conference and (C) is your new revision state. (I'm dropping the arrowheads because they always point to the left.)

To jump back to commit (B) and start new work from there, you first need to know how to reference the commit. You could either use `git log` to get the SHA1 name of (B), or you could use *HEAD^* to retrieve it (as you will remember from the previous page, *HEAD^* means the parent of the *HEAD* commit).

Now, we use the `git branch` command:

```
git branch [new-head-name] [reference-to-(B)]
```

or, for example:

```
git branch fix-headers HEAD^
```

This command will create a new head with the given name, and point that head at the requested commit object. If the commit object is left out, it will point to *HEAD*.

Now our commit tree looks like this:

```
(A) -- (B) ------- (C)
        |           |
   fix-headers    master
                    |
                   HEAD
```

Switching Between Branches

In order to start working on the headers, you need to set the fix-headers head to be the current head. This is done with `git checkout`:

```
git checkout [head-name]
```

This command does the following:

- Points *HEAD* to the commit object specified by *[head-name]*
- Rewrites all the files in the directory to match the files stored in the new *HEAD* commit.

Important note: if there are any uncommitted changes when you run `git checkout`, Git will behave very strangely. The strangeness is predictable and sometimes useful, but it is best to avoid it. All you need to do, of course, is commit all the new changes before checking out the new head.

After checking out the *fix-headers* head, you fix the headers. Now you add and commit the changes as above. The resulting repository looks like this:

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

(You can see now why it's called "branching": the commit tree has grown a new branch. Note that the angle of the line connecting (B) and (D) is irrelevant; pointers do not store whether they are horizontal or slanted.)

The ancestry of *master* is (C), (B), (A). The ancestry of *fix-headers* is (D), (B), (A). You can see this with `git log`.

Related Commands

Other useful commands at this point:

- `git branch` with no arguments lists the existing heads, with a star next to the current head.
- `git diff [head1]..[head2]` shows the diff between the commits referenced by *head2* and *head1*.
- `git diff [head1]...[head2]` (three dots) shows the diff between *head2* and the common ancestor of *head1* and *head2*. For example, `diff master...fix-headers` above would show the diff between (D) and (B).
- `git log [head1]..[head2]` shows the change log between *head2* and the common ancestor of *head1* and *head2*. With three dots, it also shows the changes between *head1* and the common ancestor; this is not so useful. (Switching *head1* and *head2*, on the other hand, is very useful.)

Common Branching Use Patterns

A common way to use Git branching is to maintain one "main" or "trunk" branch and create new branches to implement new features. Often the default Git branch, *master*, is used as the main branch.

So, in the example above, it may have been better to leave *master* at (B), where the paper was submitted to the reviewers. You could then start a new branch to store changes regarding new data.

Ideally, in this pattern, **the *master* branch is always in a releaseable state.** Other branches will contain half-finished work, new features, and so on.

This pattern is particularly important when there are multiple developers working on a single project. If all developers are adding commits in sequence to a single branch, then new features need to be added in a single commit, in order not to cause the branch to become unusable. However, if each developer creates a new branch to make a new feature, then commits can be made at any time, whether or not they are unfinished.

This is what Git users mean when they say that **commits are cheap.** If you are working on your own branch, there is no reason you need to be particularly careful about what you commit to the repository. It won't affect anything else.

Go on to the next section: Merging

Understanding Git: Merging

In This Section

Merging

After you have finished implementing a new feature on a branch, you want to bring that new feature into the main branch, so that everyone can use it. You can do so with the `git merge` or `git pull` command.

The syntax for the commands is as follows:

```
git merge [head]
git pull . [head]
```

They are identical in result. (Though the `merge` form seems simpler for now, the reason for the `pull` form will become apparent when discussing multiple developers.)

These commands perform the following operations. Let the current head be called *current*, and the head to be merged called *merge*.

1. Identify the common ancestor of *current* and *merge*. Call it *ancestor-commit*.
2. Deal with the easy cases. If the *ancestor-commit* equals *merge*, then do nothing. If *ancestor-commit* equals *current*, then do a **fast forward merge**.
3. Otherwise, determine the changes between the *ancestor-commit* and *merge*.
4. Attempt to merge those changes into the files in *current*.
5. If there were no conflicts, create a new commit, with two parents, *current* and *merge*. Set *current* (and *HEAD*) to point to this new commit, and update the working files for the project accordingly.
6. If there was a conflict, insert appropriate conflict markers and inform the user. No commit is created.

Important note: Git can get very confused if there are uncommitted changes in the files when you ask it to perform a merge. So make sure to commit whatever changes you have made so far before you merge.

So, to complete the above example, say you check out the *master* head again and finish writing up the new data for your paper. Now you want to bring in those changes you made to the headers.

The repository looks like this:

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

where (E) is the commit reflecting the completed version with the new data.

You would run:

```
git merge fix-headers
```

If there are no conflicts, the resulting respository looks like this:

```
         +---------- (D) ---------------+
        /             |                  \
(A) -- (B) -- (C) -------------- (E) -- (F)
                      |                  |
                 fix-headers           master
                                         |
                                        HEAD
```

The merge commit is (F), having parents (D) and (E). Because (B) is the common ancestor between (D) and (E), the files in (F) should contain the changes between (B) and (D), namely the heading fixes, incorporated into the files from (E).

Note on terminology: When I say “merge head A *into* head B,” I mean that head B is the current head, and you are drawing changes from head A into it. Head B gets updated; nothing is done to head A. (If you replace the word “merge” with the word “pull,” it may make more sense.)

Resolving Conflicts

A conflict arises if the commit to be merged in has a change in one place, and the current commit has a change in the same place. Git has no way of telling which change should take precedence.

To resolve the commit, edit the files to fix the conflicting changes. Then run `git add` to add the resolved files, and run `git commit` to commit the repaired merge. Git remembers that you were in the middle of a merge, so it sets the parents of the commit correctly.

Fast Forward Merges

A fast forward merge is a simple optimization for merging. Say your repository looks like this:

```
                    +-- (D) -- (E)
                   /            |
    (A) -- (B) -- (C)           |
                   |            |
                current     to-merge
                   |
                  HEAD
```

and you run `git merge to-merge`. In this case, all Git needs to do is set *current* to point to (E). Since (C) is the common ancestor, there are no changes to actually “merge.”

Hence, the resulting merged repository looks like:

```
                    +-- (D) -- (E)
                   /            |
    (A) -- (B) -- (C)           |
                                |
                        to-merge, current
                                     |
                                    HEAD
```

That is, *to-merge* and *current* both point to commit (E), and *HEAD* still points to *current*.

Note an important difference: no new commit object is created for the merge. Git only shifts the head pointers around.

Common Merge Use Patterns

There are two common reasons to merge two branches. The first, as explained above, is to draw the changes from a new feature branch into the main branch.

The second use pattern is to draw the main branch into a feature branch you are developing. This keeps the feature branch up to date with the latest bug fixes and new features added to the main branch. Doing this regularly reduces the risk of creating a conflict when you merge your feature into the main branch.

One disadvantage of doing the above is that your feature branch will end up with a lot of merge commits. An alternative that solves this problem is rebasing, although that comes with problems of its own.

Deleting a Branch

After you have merged a development branch into the main branch, you probably don't need the development branch anymore. Hence, you may want to delete it so it doesn't clutter your `git branch` listing.

To delete a branch, use `git branch -d [head]`. This simply removes the specified head from the repository's list of heads.

For example, in this repository from above:

```
            +---------- (D) ---------------+
           /             |                  \
    (A) -- (B) -- (C) -------------- (E) -- (F)
                         |                  |
                    fix-headers           master
                                            |
                                           HEAD
```

we probably don't need the *fix-headers* head any more. So we can use:

```
    git branch -d fix-headers
```

and the resulting repository looks like:

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

Important note: `git branch -d` will cause an error if the branch to be deleted is not reachable from another head. Why? Consider the following repository:

```
            +----------- (E)
           /              |
    (A) -- (B) -- (C)     |
                   |      |
                 head1   head2
```

Say you delete *head2*. Now how can you use commit (E)? You can’t check it out, because it isn’t a head. And it doesn’t appear in any logs or anywhere else, because it isn’t an ancestor of *head1*. So commit (E) is practically useless. In Git terminology, it is a “dangling commit,” and its information is lost.

Git does allow you to use the `-D` option to force deletion of a branch that would create a dangling commit. However, it should be a rare situation that you want to do that. **Think very carefully before using `git branch -D`.**

Go on to the next page: Collaborating

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 and head references 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

Understanding Git: Rebasing

In This Section

Rebasing

Git offers a unique feature called `rebasing` as an alternative to merging. Its syntax is:

```
    git rebase [new-commit]
```

When run, Git performs the following steps:

- Identifies each commit that is an ancestor of the current commit but not of [new-commit]. This can be thought of as a two-step process: first, find the common ancestor of [new-commit] and the current commit; call this the *ancestor commit*. Second, collect all the commits between the *ancestor commit* and the current commit.
- Determines what changed for each of those commits, and puts those changes aside.
- Sets the current head to point to [new-commit].
- For each of the changes set aside, replays that change onto the current head and creates a new commit.

The result is that HEAD, the current commit, is a descendant of [new-commit], but it contains all the changes as if it had been merged with [new-commit].

For example, imagine that a repository looks like:

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

Performing `git rebase master` would produce the following:

```
                  +------ (E)
                 /        |
(A) -- (B) -- (C)         |
              |           |
           master  new-branch
                          |
                         HEAD
```

where (E) is a new commit that incorporates the changes between (B) and (D), but reorganized to place those changes on (C).

Rebase has the advantage that there is no merge commit created. However, because HEAD is not a descendant of the pre-rebase HEAD commit, rebasing can be problematic. For one thing, it means that a rebased head cannot be pushed to a remote server, because it does not result in a fast forward merge. Moreover, it results in a loss of information. It is no longer known that (D) was once on the *new-branch* head. This results in a "changing of history" that could confuse someone who already knows about commit (D).

Common Rebasing Use Practices

Because of this danger of rebasing, it is best reserved for two situations.

First, if you are developing a branch on your own and not sharing it with anyone, you could rebase it to keep the branch up to date with respect to the main branch. Then, when you finally merge your developed branch into the main branch, it will be free of merge commits, because it will appear that your development branch was a descendant of the main head. Moreover, the main branch can move forward with a fast forward merge rather than a regular merge commit.

Second, if you commit to a branch but that branch changes at the same time on a remote machine, you can use rebase to shift your own commits, allowing you to push your commits to the remote repository. From the example above:

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

Your repository            Remote repository
```

If you perform a `fetch`, your repository will look like this:

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

Now, assuming *master* is your current head, you run:

```
git rebase origin/master
```

And your repository will look like:

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

Commit (E) contains the changes you made between commits (B) and (C), but now commit (E) is a descendant of (D). This allows you to push your head *master*, as a fast forward merge, to update *origin/master*.

Moving On

There are many more things possible with Git. The manual pages generally document, in fairly good detail, the possible operations.

> The entire Pro Git book, written by Scott Chacon and published by Apress, is available on the Git website.

If you understand how the Git repository is a tree of commits and see how operations like branching, merging, pushing, and pulling manipulate that tree, then understanding other Git commands should not be difficult. You should be able to visualize how any command will search or modify the tree and specify the right commits on the tree for Git to operate on.

So, go forth and code!

My e-mail address is “website.comments” (without the quotes) followed by an @ symbol, my first initial and last name concatenated, a dot, and “com.”

This version converted to markdown, epub, mobi and html by Kim Plowright