NAME

gitcore-tutorial − A git core tutorial for developers

SYNOPSIS

git *

DESCRIPTION

This tutorial explains how to use the "core" git commands to set up and work with a git repository.

If you just need to use git as a revision control system you may prefer to start with "A Tutorial Introduction to GIT" (**gittutorial**(7)) or **the GIT User Manual**[1].

However, an understanding of these low−level tools can be helpful if you want to understand git's internals.

The core git is often called "plumbing", with the prettier user interfaces on top of it called "porcelain". You may not want to use the plumbing directly very often, but it can be good to know what the plumbing does for when the porcelain isn't flushing.

Back when this document was originally written, many porcelain commands were shell scripts. For simplicity, it still uses them as examples to illustrate how plumbing is fit together to form the porcelain commands. The source tree includes some of these scripts in contrib/examples/ for reference. Although these are not implemented as shell scripts anymore, the description of what the plumbing layer commands do is still valid.

Note

Deeper technical details are often marked as Notes, which you can skip on your first reading.

CREATING A GIT REPOSITORY

Creating a new git repository couldn't be easier: all git repositories start out empty, and the only thing you need to do is find yourself a subdirectory that you want to use as a working tree − either an empty one for a totally new project, or an existing working tree that you want to import into git.

For our first example, we're going to start a totally new repository from scratch, with no pre−existing files, and we'll call it *git−tutorial*. To start up, create a subdirectory for it, change into that subdirectory, and initialize the git infrastructure with *git init*:

```
$ mkdir git−tutorial
$ cd git−tutorial
$ git init
```

to which git will reply

```
Initialized empty Git repository in .git/
```

which is just git's way of saying that you haven't been doing anything strange, and that it will have created a local .git directory setup for your new project. You will now have a .git directory, and you can inspect that with *ls*. For your new empty project, it should show you three entries, among other things:

- a file called HEAD, that has ref: refs/heads/master in it. This is similar to a symbolic link and points at refs/heads/master relative to the HEAD file.

  Don't worry about the fact that the file that the HEAD link points to doesn't even exist yet — you haven't created the commit that will start your HEAD development branch yet.

- a subdirectory called objects, which will contain all the objects of your project. You should never have any real reason to look at the objects directly, but you might want to know that these objects

are what contains all the real *data* in your repository.

- a subdirectory called refs, which contains references to objects.

In particular, the refs subdirectory will contain two other subdirectories, named heads and tags respectively. They do exactly what their names imply: they contain references to any number of different *heads* of development (aka *branches*), and to any *tags* that you have created to name specific versions in your repository.

One note: the special master head is the default branch, which is why the .git/HEAD file was created points to it even if it doesn't yet exist. Basically, the HEAD link is supposed to always point to the branch you are working on right now, and you always start out expecting to work on the master branch.

However, this is only a convention, and you can name your branches anything you want, and don't have to ever even *have* a master branch. A number of the git tools will assume that .git/HEAD is valid, though.

Note

An *object* is identified by its 160−bit SHA1 hash, aka *object name*, and a reference to an object is always the 40−byte hex representation of that SHA1 name. The files in the 'refs' subdirectory are expected to contain these hex references (usually with a final '\n\' at the end), and you should thus expect to see a number of 41−byte files containing these references in these refs subdirectories when you actually start populating your tree.

Note

An advanced user may want to take a look at **gitrepository-layout**(5) after finishing this tutorial.

You have now created your first git repository. Of course, since it's empty, that's not very useful, so let's start populating it with data.

POPULATING A GIT REPOSITORY

We'll keep this simple and stupid, so we'll start off with populating a few trivial files just to get a feel for it.

Start off with just creating any random files that you want to maintain in your git repository. We'll start off with a few bad examples, just to get a feel for how this works:

```
$ echo "Hello World" >hello
$ echo "Silly example" >example
```

you have now created two files in your working tree (aka *working directory*), but to actually check in your hard work, you will have to go through two steps:

- fill in the *index* file (aka *cache*) with the information about your working tree state.
- commit that index file as an object.

The first step is trivial: when you want to tell git about any changes to your working tree, you use the *git update−index* program. That program normally just takes a list of filenames you want to update, but to avoid trivial mistakes, it refuses to add new entries to the index (or remove existing ones) unless you explicitly tell it that you're adding a new entry with the −−add flag (or removing an entry with the −−remove) flag.

So to populate the index with the two files you just created, you can do

```
$ git update−index −−add hello example
```

and you have now told git to track those two files.

In fact, as you did that, if you now look into your object directory, you'll notice that git will have added two new objects to the object database. If you did exactly the steps above, you should now be able to do

$ ls .git/objects/??/*

and see two files:

.git/objects/55/7db03de997c86a4a028e1ebd3a1ceb225be238
.git/objects/f2/4c74a2e500f5ee1332c86b94199f52b1d1d962

which correspond with the objects with names of 557db... and f24c7... respectively.

If you want to, you can use *git cat−file* to look at those objects, but you'll have to use the object name, not the filename of the object:

$ git cat−file −t 557db03de997c86a4a028e1ebd3a1ceb225be238

where the −t tells *git cat−file* to tell you what the "type" of the object is. git will tell you that you have a "blob" object (i.e., just a regular file), and you can see the contents with

$ git cat−file blob 557db03

which will print out "Hello World". The object 557db03 is nothing more than the contents of your file hello.

Note

Don't confuse that object with the file hello itself. The object is literally just those specific **contents** of the file, and however much you later change the contents in file hello, the object we just looked at will never change. Objects are immutable.

Note

The second example demonstrates that you can abbreviate the object name to only the first several hexadecimal digits in most places.

Anyway, as we mentioned previously, you normally never actually take a look at the objects themselves, and typing long 40−character hex names is not something you'd normally want to do. The above digression was just to show that *git update−index* did something magical, and actually saved away the contents of your files into the git object database.

Updating the index did something else too: it created a .git/index file. This is the index that describes your current working tree, and something you should be very aware of. Again, you normally never worry about the index file itself, but you should be aware of the fact that you have not actually really "checked in" your files into git so far, you've only **told** git about them.

However, since git knows about them, you can now start using some of the most basic git commands to manipulate the files or look at their status.

In particular, let's not even check in the two files into git yet, we'll start off by adding another line to hello first:

$ echo "It´s a new day for git" >>hello

and you can now, since you told git about the previous state of hello, ask git what has changed in the tree compared to

your old index, using the *git diff−files* command:

$ git diff−files

Oops. That wasn’t very readable. It just spit out its own internal version of a *diff*, but that internal version really just tells you that it has noticed that "hello" has been modified, and that the old object contents it had have been replaced with something else.

To make it readable, we can tell *git diff−files* to output the differences as a patch, using the −p flag:

```
$ git diff−files −p
diff −−git a/hello b/hello
index 557db03..263414f 100644
−−− a/hello
+++ b/hello
@@ −1 +1,2 @@
 Hello World
+It´s a new day for git
```

i.e. the diff of the change we caused by adding another line to hello.

In other words, *git diff−files* always shows us the difference between what is recorded in the index, and what is currently in the working tree. That’s very useful.

A common shorthand for git diff−files −p is to just write git diff, which will do the same thing.

```
$ git diff
diff −−git a/hello b/hello
index 557db03..263414f 100644
−−− a/hello
+++ b/hello
@@ −1 +1,2 @@
 Hello World
+It´s a new day for git
```

COMMITTING GIT STATE

Now, we want to go to the next stage in git, which is to take the files that git knows about in the index, and commit them as a real tree. We do that in two phases: creating a *tree* object, and committing that *tree* object as a *commit* object together with an explanation of what the tree was all about, along with information of how we came to that state.

Creating a tree object is trivial, and is done with *git write−tree*. There are no options or other input: git write−tree will take the current index state, and write an object that describes that whole index. In other words, we’re now tying together all the different filenames with their contents (and their permissions), and we’re creating the equivalent of a git "directory" object:

$ git write−tree

and this will just output the name of the resulting tree, in this case (if you have done exactly as I’ve described) it should be

8988da15d077d4829fc51d8544c097def6644dbb

which is another incomprehensible object name. Again, if you want to, you can use git cat−file −t 8988d... to see that this time the object is not a "blob" object, but a "tree" object (you can also use git cat−file to actually output the raw object contents, but you'll see mainly a binary mess, so that's less interesting).

However — normally you'd never use *git write−tree* on its own, because normally you always commit a tree into a commit object using the *git commit−tree* command. In fact, it's easier to not actually use *git write−tree* on its own at all, but to just pass its result in as an argument to *git commit−tree*.

git commit−tree normally takes several arguments — it wants to know what the *parent* of a commit was, but since this is the first commit ever in this new repository, and it has no parents, we only need to pass in the object name of the tree. However, *git commit−tree* also wants to get a commit message on its standard input, and it will write out the resulting object name for the commit to its standard output.

And this is where we create the .git/refs/heads/master file which is pointed at by HEAD. This file is supposed to contain the reference to the top−of−tree of the master branch, and since that's exactly what *git commit−tree* spits out, we can do this all with a sequence of simple shell commands:

```
$ tree=$(git write−tree)
$ commit=$(echo ´Initial commit´ | git commit−tree $tree)
$ git update−ref HEAD $commit
```

In this case this creates a totally new commit that is not related to anything else. Normally you do this only **once** for a project ever, and all later commits will be parented on top of an earlier commit.

Again, normally you'd never actually do this by hand. There is a helpful script called git commit that will do all of this for you. So you could have just written git commit instead, and it would have done the above magic scripting for you.

MAKING A CHANGE

Remember how we did the *git update−index* on file hello and then we changed hello afterward, and could compare the new state of hello with the state we saved in the index file?

Further, remember how I said that *git write−tree* writes the contents of the **index** file to the tree, and thus what we just committed was in fact the **original** contents of the file hello, not the new ones. We did that on purpose, to show the difference between the index state, and the state in the working tree, and how they don't have to match, even when we commit things.

As before, if we do git diff−files −p in our git−tutorial project, we'll still see the same difference we saw last time: the index file hasn't changed by the act of committing anything. However, now that we have committed something, we can also learn to use a new command: *git diff−index*.

Unlike *git diff−files*, which showed the difference between the index file and the working tree, *git diff−index* shows the differences between a committed **tree** and either the index file or the working tree. In other words, *git diff−index* wants a tree to be diffed against, and before we did the commit, we couldn't do that, because we didn't have anything to diff against.

But now we can do

```
$ git diff−index −p HEAD
```

(where −p has the same meaning as it did in *git diff−files*), and it will show us the same difference, but for a totally different reason. Now we're comparing the working tree not against the index file, but against the tree we just wrote. It just so happens that those two are obviously the same, so we get the same result.

Again, because this is a common operation, you can also just shorthand it with

$ git diff HEAD

which ends up doing the above for you.

In other words, *git diff−index* normally compares a tree against the working tree, but when given the −−cached flag, it is told to instead compare against just the index cache contents, and ignore the current working tree state entirely. Since we just wrote the index file to HEAD, doing git diff−index −−cached −p HEAD should thus return an empty set of differences, and that's exactly what it does.

Note

git diff−index really always uses the index for its comparisons, and saying that it compares a tree against the working tree is thus not strictly accurate. In particular, the list of files to compare (the "meta−data") **always** comes from the index file, regardless of whether the −−cached flag is used or not. The −−cached flag really only determines whether the file **contents** to be compared come from the working tree or not.

This is not hard to understand, as soon as you realize that git simply never knows (or cares) about files that it is not told about explicitly. git will never go **looking** for files to compare, it expects you to tell it what the files are, and that's what the index is there for.

However, our next step is to commit the **change** we did, and again, to understand what's going on, keep in mind the difference between "working tree contents", "index file" and "committed tree". We have changes in the working tree that we want to commit, and we always have to work through the index file, so the first thing we need to do is to update the index cache:

$ git update−index hello

(note how we didn't need the −−add flag this time, since git knew about the file already).

Note what happens to the different *git diff−** versions here. After we've updated hello in the index, git diff−files −p now shows no differences, but git diff−index −p HEAD still **does** show that the current state is different from the state we committed. In fact, now *git diff−index* shows the same difference whether we use the −−cached flag or not, since now the index is coherent with the working tree.

Now, since we've updated hello in the index, we can commit the new version. We could do it by writing the tree by hand again, and committing the tree (this time we'd have to use the −p HEAD flag to tell commit that the HEAD was the **parent** of the new commit, and that this wasn't an initial commit any more), but you've done that once already, so let's just use the helpful script this time:

$ git commit

which starts an editor for you to write the commit message and tells you a bit about what you have done.

Write whatever message you want, and all the lines that start with # will be pruned out, and the rest will be used as the commit message for the change. If you decide you don't want to commit anything after all at this point (you can continue to edit things and update the index), you can just leave an empty message. Otherwise git commit will commit the change for you.

You’ve now made your first real git commit. And if you’re interested in looking at what git commit really does, feel free to investigate: it’s a few very simple shell scripts to generate the helpful (?) commit message headers, and a few one−liners that actually do the commit itself (*git commit*).

INSPECTING CHANGES

While creating changes is useful, it’s even more useful if you can tell later what changed. The most useful command for this is another of the *diff* family, namely *git diff−tree*.

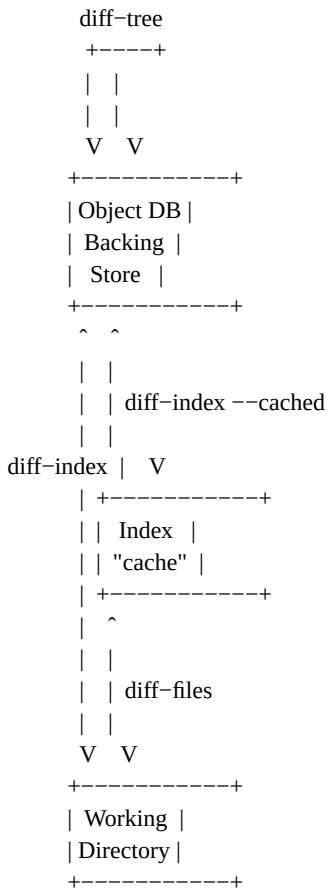
git diff−tree can be given two arbitrary trees, and it will tell you the differences between them. Perhaps even more commonly, though, you can give it just a single commit object, and it will figure out the parent of that commit itself, and show the difference directly. Thus, to get the same diff that we’ve already seen several times, we can now do

$ git diff−tree −p HEAD

(again, −p means to show the difference as a human−readable patch), and it will show what the last commit (in HEAD) actually changed.

Note

Here is an ASCII art by Jon Loeliger that illustrates how various diff−* commands compare things.



More interestingly, you can also give *git diff−tree* the −−pretty flag, which tells it to also show the commit message and author and date of the commit, and you can tell it to show a whole series of diffs. Alternatively, you can tell it to be "silent", and not show the diffs at all, but just show the actual commit message.

In fact, together with the *git rev−list* program (which generates a list of revisions), *git diff−tree* ends up being a

veritable fount of changes. A trivial (but very useful) script called *git whatchanged* is included with git which does exactly this, and shows a log of recent activities.

To see the whole history of our pitiful little git−tutorial project, you can do

```
$ git log
```

which shows just the log messages, or if we want to see the log together with the associated patches use the more complex (and much more powerful)

```
$ git whatchanged −p
```

and you will see exactly what has changed in the repository over its short history.

Note
When using the above two commands, the initial commit will be shown. If this is a problem because it is huge, you can hide it by setting the log.showroot configuration variable to false. Having this, you can still show it for each command just adding the −−root option, which is a flag for *git diff−tree* accepted by both commands.

With that, you should now be having some inkling of what git does, and can explore on your own.

Note
Most likely, you are not directly using the core git Plumbing commands, but using Porcelain such as *git add*, ‘git−rm’ and ‘git−commit’.

TAGGING A VERSION

In git, there are two kinds of tags, a "light" one, and an "annotated tag".

A "light" tag is technically nothing more than a branch, except we put it in the .git/refs/tags/ subdirectory instead of calling it a head. So the simplest form of tag involves nothing more than

```
$ git tag my−first−tag
```

which just writes the current HEAD into the .git/refs/tags/my−first−tag file, after which point you can then use this symbolic name for that particular state. You can, for example, do

```
$ git diff my−first−tag
```

to diff your current state against that tag which at this point will obviously be an empty diff, but if you continue to develop and commit stuff, you can use your tag as an "anchor−point" to see what has changed since you tagged it.

An "annotated tag" is actually a real git object, and contains not only a pointer to the state you want to tag, but also a small tag name and message, along with optionally a PGP signature that says that yes, you really did that tag. You create these annotated tags with either the −a or −s flag to *git tag*:

```
$ git tag −s <tagname>
```

which will sign the current HEAD (but you can also give it another argument that specifies the thing to tag, e.g., you could have tagged the current mybranch point by using git tag <tagname> mybranch).

You normally only do signed tags for major releases or things like that, while the light−weight tags are useful for any marking you want to do — any time you decide that you want to remember a certain point, just create a private tag for it, and you have a nice symbolic name for the state at that point.

COPYING REPOSITORIES

git repositories are normally totally self−sufficient and relocatable. Unlike CVS, for example, there is no separate notion of "repository" and "working tree". A git repository normally **is** the working tree, with the local git information hidden in the .git subdirectory. There is nothing else. What you see is what you got.

Note

You can tell git to split the git internal information from the directory that it tracks, but we'll ignore that for now: it's not how normal projects work, and it's really only meant for special uses. So the mental model of "the git information is always tied directly to the working tree that it describes" may not be technically 100% accurate, but it's a good model for all normal use.

This has two implications:

- if you grow bored with the tutorial repository you created (or you've made a mistake and want to start all over), you can just do simple

  $ rm −rf git−tutorial

  and it will be gone. There's no external repository, and there's no history outside the project you created.

- if you want to move or duplicate a git repository, you can do so. There is *git clone* command, but if all you want to do is just to create a copy of your repository (with all the full history that went along with it), you can do so with a regular cp −a git−tutorial new−git−tutorial.

  Note that when you've moved or copied a git repository, your git index file (which caches various information, notably some of the "stat" information for the files involved) will likely need to be refreshed. So after you do a cp −a to create a new copy, you'll want to do

  $ git update−index −−refresh

  in the new repository to make sure that the index file is up−to−date.

Note that the second point is true even across machines. You can duplicate a remote git repository with **any** regular copy mechanism, be it *scp*, *rsync* or *wget*.

When copying a remote repository, you'll want to at a minimum update the index cache when you do this, and especially with other peoples´ repositories you often want to make sure that the index cache is in some known state (you don't know **what** they've done and not yet checked in), so usually you'll precede the *git update−index* with a

```
$ git read−tree −−reset HEAD
$ git update−index −−refresh
```

which will force a total index re−build from the tree pointed to by HEAD. It resets the index contents to HEAD, and then the *git update−index* makes sure to match up all index entries with the checked−out files. If the original repository had uncommitted changes in its working tree, git update−index −−refresh notices them and tells you they need to be updated.

The above can also be written as simply

$ git reset

and in fact a lot of the common git command combinations can be scripted with the git xyz interfaces. You can learn things by just looking at what the various git scripts do. For example, git reset used to be the above two lines implemented in *git reset*, but some things like *git status* and *git commit* are slightly more complex scripts around the basic git commands.

Many (most?) public remote repositories will not contain any of the checked out files or even an index file, and will **only** contain the actual core git files. Such a repository usually doesn't even have the .git subdirectory, but has all the git files directly in the repository.

To create your own local live copy of such a "raw" git repository, you'd first create your own subdirectory for the project, and then copy the raw repository contents into the .git directory. For example, to create your own copy of the git repository, you'd do the following

```
$ mkdir my−git
$ cd my−git
$ rsync −rL rsync://rsync.kernel.org/pub/scm/git/git.git/ .git
```

followed by

```
$ git read−tree HEAD
```

to populate the index. However, now you have populated the index, and you have all the git internal files, but you will notice that you don't actually have any of the working tree files to work on. To get those, you'd check them out with

```
$ git checkout−index −u −a
```

where the −u flag means that you want the checkout to keep the index up−to−date (so that you don't have to refresh it afterward), and the −a flag means "check out all files" (if you have a stale copy or an older version of a checked out tree you may also need to add the −f flag first, to tell *git checkout−index* to **force** overwriting of any old files).

Again, this can all be simplified with

```
$ git clone rsync://rsync.kernel.org/pub/scm/git/git.git/ my−git
$ cd my−git
$ git checkout
```

which will end up doing all of the above for you.

You have now successfully copied somebody else's (mine) remote repository, and checked it out.

CREATING A NEW BRANCH

Branches in git are really nothing more than pointers into the git object database from within the .git/refs/ subdirectory, and as we already discussed, the HEAD branch is nothing but a symlink to one of these object pointers.

You can at any time create a new branch by just picking an arbitrary point in the project history, and just writing the SHA1 name of that object into a file under .git/refs/heads/. You can use any filename you want (and indeed, subdirectories), but the convention is that the "normal" branch is called master. That's just a convention, though, and nothing enforces it.

To show that as an example, let’s go back to the git−tutorial repository we used earlier, and create a branch in it. You do that by simply just saying that you want to check out a new branch:

$ git checkout −b mybranch

will create a new branch based at the current HEAD position, and switch to it.

Note

If you make the decision to start your new branch at some other point in the history than the current HEAD, you can do so by just telling *git checkout* what the base of the checkout would be. In other words, if you have an earlier tag or branch, you’d just do

$ git checkout −b mybranch earlier−commit

and it would create the new branch mybranch at the earlier commit, and check out the state at that time.

You can always just jump back to your original master branch by doing

$ git checkout master

(or any other branch−name, for that matter) and if you forget which branch you happen to be on, a simple

$ cat .git/HEAD

will tell you where it’s pointing. To get the list of branches you have, you can say

$ git branch

which used to be nothing more than a simple script around ls .git/refs/heads. There will be an asterisk in front of the branch you are currently on.

Sometimes you may wish to create a new branch *without* actually checking it out and switching to it. If so, just use the command

$ git branch <branchname> [startingpoint]

which will simply *create* the branch, but will not do anything further. You can then later — once you decide that you want to actually develop on that branch — switch to that branch with a regular *git checkout* with the branchname as the argument.

MERGING TWO BRANCHES

One of the ideas of having a branch is that you do some (possibly experimental) work in it, and eventually merge it back to the main branch. So assuming you created the above mybranch that started out being the same as the original master branch, let’s make sure we’re in that branch, and do some work there.

$ git checkout mybranch
$ echo "Work, work, work" >>hello
$ git commit −m "Some work." −i hello

Here, we just added another line to hello, and we used a shorthand for doing both git update−index hello and git commit by just giving the filename directly to git commit, with an −i flag (it tells git to *include* that file in addition to what you have done to the index file so far when making the commit). The −m flag is to give the commit log message from the command line.

Now, to make it a bit more interesting, let's assume that somebody else does some work in the original branch, and simulate that by going back to the master branch, and editing the same file differently there:

$ git checkout master

Here, take a moment to look at the contents of hello, and notice how they don't contain the work we just did in mybranch — because that work hasn't happened in the master branch at all. Then do

$ echo "Play, play, play" >>hello
$ echo "Lots of fun" >>example
$ git commit −m "Some fun." −i hello example

since the master branch is obviously in a much better mood.

Now, you've got two branches, and you decide that you want to merge the work done. Before we do that, let's introduce a cool graphical tool that helps you view what's going on:

$ gitk −−all

will show you graphically both of your branches (that's what the −−all means: normally it will just show you your current HEAD) and their histories. You can also see exactly how they came to be from a common source.

Anyway, let's exit *gitk* (^Q or the File menu), and decide that we want to merge the work we did on the mybranch branch into the master branch (which is currently our HEAD too). To do that, there's a nice script called *git merge*, which wants to know which branches you want to resolve and what the merge is all about:

$ git merge −m "Merge work in mybranch" mybranch

where the first argument is going to be used as the commit message if the merge can be resolved automatically.

Now, in this case we've intentionally created a situation where the merge will need to be fixed up by hand, though, so git will do as much of it as it can automatically (which in this case is just merge the example file, which had no differences in the mybranch branch), and say:

```
Auto−merging hello
CONFLICT (content): Merge conflict in hello
Automatic merge failed; fix conflicts and then commit the result.
```

It tells you that it did an "Automatic merge", which failed due to conflicts in hello.

Not to worry. It left the (trivial) conflict in hello in the same form you should already be well used to if you've ever used CVS, so let's just open hello in our editor (whatever that may be), and fix it up somehow.

I’d suggest just making it so that hello contains all four lines:

Hello World
It´s a new day for git
Play, play, play
Work, work, work

and once you’re happy with your manual merge, just do a

$ git commit −i hello

which will very loudly warn you that you’re now committing a merge (which is correct, so never mind), and you can write a small merge message about your adventures in *git merge*−land.

After you’re done, start up gitk −−all to see graphically what the history looks like. Notice that mybranch still exists, and you can switch to it, and continue to work with it if you want to. The mybranch branch will not contain the merge, but next time you merge it from the master branch, git will know how you merged it, so you’ll not have to do *that* merge again.

Another useful tool, especially if you do not always work in X−Window environment, is git show−branch.

$ git show−branch −−topo−order −−more=1 master mybranch
* [master] Merge work in mybranch
 ! [mybranch] Some work.
−−
− [master] Merge work in mybranch
*+ [mybranch] Some work.
* [masterˆ] Some fun.

The first two lines indicate that it is showing the two branches and the first line of the commit log message from their top−of−the−tree commits, you are currently on master branch (notice the asterisk * character), and the first column for the later output lines is used to show commits contained in the master branch, and the second column for the mybranch branch. Three commits are shown along with their log messages. All of them have non blank characters in the first column (* shows an ordinary commit on the current branch, − is a merge commit), which means they are now part of the master branch. Only the "Some work" commit has the plus + character in the second column, because mybranch has not been merged to incorporate these commits from the master branch. The string inside brackets before the commit log message is a short name you can use to name the commit. In the above example, *master* and *mybranch* are branch heads. *masterˆ* is the first parent of *master* branch head. Please see **git-rev-parse**(1) if you want to see more complex cases.

Note

Without the *−−more=1* option, *git show−branch* would not output the *[masterˆ]* commit, as *[mybranch]* commit is a common ancestor of both *master* and *mybranch* tips. Please see **git-show-branch**(1) for details.

Note

If there were more commits on the *master* branch after the merge, the merge commit itself would not be shown by *git show−branch* by default. You would need to provide *−−sparse* option to make the merge commit visible in this case.

Now, let’s pretend you are the one who did all the work in mybranch, and the fruit of your hard work has finally been merged to the master branch. Let’s go back to mybranch, and run *git merge* to get the "upstream changes" back to your branch.

```
$ git checkout mybranch
$ git merge −m "Merge upstream changes." master
```

This outputs something like this (the actual commit object names would be different)

```
Updating from ae3a2da... to a80b4aa....
Fast−forward (no commit created; −m option ignored)
 example |    1 +
 hello   |    1 +
 2 files changed, 2 insertions(+), 0 deletions(−)
```

Because your branch did not contain anything more than what had already been merged into the master branch, the merge operation did not actually do a merge. Instead, it just updated the top of the tree of your branch to that of the master branch. This is often called *fast−forward* merge.

You can run gitk −−all again to see how the commit ancestry looks like, or run *show−branch*, which tells you this.

```
$ git show−branch master mybranch
! [master] Merge work in mybranch
 * [mybranch] Merge work in mybranch
−−
−− [master] Merge work in mybranch
```

MERGING EXTERNAL WORK

It’s usually much more common that you merge with somebody else than merging with your own branches, so it’s worth pointing out that git makes that very easy too, and in fact, it’s not that different from doing a *git merge*. In fact, a remote merge ends up being nothing more than "fetch the work from a remote repository into a temporary tag" followed by a *git merge*.

Fetching from a remote repository is done by, unsurprisingly, *git fetch*:

```
$ git fetch <remote−repository>
```

One of the following transports can be used to name the repository to download from:

Rsync

rsync://remote.machine/path/to/repo.git/

Rsync transport is usable for both uploading and downloading, but is completely unaware of what git does, and can produce unexpected results when you download from the public repository while the repository owner is uploading into it via rsync transport. Most notably, it could update the files under refs/ which holds the object name of the topmost commits before uploading the files in objects/ — the downloader would obtain head commit object name while that object itself is still not available in the repository. For this reason, it is considered deprecated.

SSH

remote.machine:/path/to/repo.git/ or

ssh://remote.machine/path/to/repo.git/

This transport can be used for both uploading and downloading, and requires you to have a log−in privilege over ssh to the remote machine. It finds out the set of objects the other side lacks by exchanging the head commits both ends have and transfers (close to) minimum set of objects. It is by far the most efficient way to exchange git objects between repositories.

Local directory

/path/to/repo.git/

This transport is the same as SSH transport but uses *sh* to run both ends on the local machine instead of running other end on the remote machine via *ssh*.

git Native

git://remote.machine/path/to/repo.git/

This transport was designed for anonymous downloading. Like SSH transport, it finds out the set of objects the downstream side lacks and transfers (close to) minimum set of objects.

HTTP(S)

http://remote.machine/path/to/repo.git/

Downloader from http and https URL first obtains the topmost commit object name from the remote site by looking at the specified refname under repo.git/refs/ directory, and then tries to obtain the commit object by downloading from repo.git/objects/xx/xxx... using the object name of that commit object. Then it reads the commit object to find out its parent commits and the associate tree object; it repeats this process until it gets all the necessary objects. Because of this behavior, they are sometimes also called *commit walkers*.

The *commit walkers* are sometimes also called *dumb transports*, because they do not require any git aware smart server like git Native transport does. Any stock HTTP server that does not even support directory index would suffice. But you must prepare your repository with *git update−server−info* to help dumb transport downloaders.

Once you fetch from the remote repository, you merge that with your current branch.

However — it’s such a common thing to fetch and then immediately merge, that it’s called git pull, and you can simply do

$ git pull <remote−repository>

and optionally give a branch−name for the remote end as a second argument.

Note

You could do without using any branches at all, by keeping as many local repositories as you would like to have branches, and merging between them with *git pull*, just like you merge between branches. The advantage of this approach is that it lets you keep a set of files for each branch checked out and you may find it easier to switch back and forth if you juggle multiple lines of development simultaneously. Of course, you will pay the price of more disk usage to hold multiple working trees, but disk space is cheap these days.

It is likely that you will be pulling from the same remote repository from time to time. As a short hand, you can store the remote repository URL in the local repository’s config file like this:

$ git config remote.linus.url http://www.kernel.org/pub/scm/git/git.git/

and use the "linus" keyword with *git pull* instead of the full URL.

Examples.

1. git pull linus
2. git pull linus tag v0.99.1

the above are equivalent to:

1. git pull **http://www.kernel.org/pub/scm/git/git.git/** HEAD
2. git pull **http://www.kernel.org/pub/scm/git/git.git/** tag v0.99.1

HOW DOES THE MERGE WORK?

We said this tutorial shows what plumbing does to help you cope with the porcelain that isn't flushing, but we so far did not talk about how the merge really works. If you are following this tutorial the first time, I'd suggest to skip to "Publishing your work" section and come back here later.

OK, still with me? To give us an example to look at, let's go back to the earlier repository with "hello" and "example" file, and bring ourselves back to the pre−merge state:

```
$ git show−branch −−more=2 master mybranch
! [master] Merge work in mybranch
 * [mybranch] Merge work in mybranch
−−
−− [master] Merge work in mybranch
+* [master^2] Some work.
+* [master^] Some fun.
```

Remember, before running *git merge*, our master head was at "Some fun." commit, while our mybranch head was at "Some work." commit.

```
$ git checkout mybranch
$ git reset −−hard master^2
$ git checkout master
$ git reset −−hard master^
```

After rewinding, the commit structure should look like this:

```
$ git show−branch
* [master] Some fun.
 ! [mybranch] Some work.
−−
*  [master] Some fun.
 + [mybranch] Some work.
*+ [master^] Initial commit
```

Now we are ready to experiment with the merge by hand.

git merge command, when merging two branches, uses 3−way merge algorithm. First, it finds the common ancestor between them. The command it uses is *git merge−base*:

```
$ mb=$(git merge−base HEAD mybranch)
```

The command writes the commit object name of the common ancestor to the standard output, so we captured its output to a variable, because we will be using it in the next step. By the way, the common ancestor commit is the "Initial commit" commit in this case. You can tell it by:

```
$ git name−rev −−name−only −−tags $mb
my−first−tag
```

After finding out a common ancestor commit, the second step is this:

```
$ git read−tree −m −u $mb HEAD mybranch
```

This is the same *git read−tree* command we have already seen, but it takes three trees, unlike previous examples. This reads the contents of each tree into different *stage* in the index file (the first tree goes to stage 1, the second to stage 2, etc.). After reading three trees into three stages, the paths that are the same in all three stages are *collapsed* into stage 0. Also paths that are the same in two of three stages are collapsed into stage 0, taking the SHA1 from either stage 2 or stage 3, whichever is different from stage 1 (i.e. only one side changed from the common ancestor).

After *collapsing* operation, paths that are different in three trees are left in non−zero stages. At this point, you can inspect the index file with this command:

```
$ git ls−files −−stage
100644 7f8b141b65fdcee47321e399a2598a235a032422 0       example
100644 557db03de997c86a4a028e1ebd3a1ceb225be238 1       hello
100644 ba42a2a96e3027f3333e13ede4ccf4498c3ae942 2       hello
100644 cc44c73eb783565da5831b4d820c962954019b69 3       hello
```

In our example of only two files, we did not have unchanged files so only *example* resulted in collapsing. But in real−life large projects, when only a small number of files change in one commit, this *collapsing* tends to trivially merge most of the paths fairly quickly, leaving only a handful of real changes in non−zero stages.

To look at only non−zero stages, use −−unmerged flag:

```
$ git ls−files −−unmerged
100644 557db03de997c86a4a028e1ebd3a1ceb225be238 1       hello
100644 ba42a2a96e3027f3333e13ede4ccf4498c3ae942 2       hello
100644 cc44c73eb783565da5831b4d820c962954019b69 3       hello
```

The next step of merging is to merge these three versions of the file, using 3−way merge. This is done by giving *git merge−one−file* command as one of the arguments to *git merge−index* command:

```
$ git merge−index git−merge−one−file hello
Auto−merging hello
ERROR: Merge conflict in hello
fatal: merge program failed
```

git merge−one−file script is called with parameters to describe those three versions, and is responsible to leave the merge results in the working tree. It is a fairly straightforward shell script, and eventually calls

merge program from RCS suite to perform a file−level 3−way merge. In this case, *merge* detects conflicts, and the merge result with conflict marks is left in the working tree.. This can be seen if you run ls−files −−stage again at this point:

```
$ git ls−files −−stage
100644 7f8b141b65fdcee47321e399a2598a235a032422 0       example
100644 557db03de997c86a4a028e1ebd3a1ceb225be238 1       hello
100644 ba42a2a96e3027f3333e13ede4ccf4498c3ae942 2       hello
100644 cc44c73eb783565da5831b4d820c962954019b69 3       hello
```

This is the state of the index file and the working file after *git merge* returns control back to you, leaving the conflicting merge for you to resolve. Notice that the path hello is still unmerged, and what you see with *git diff* at this point is differences since stage 2 (i.e. your version).

PUBLISHING YOUR WORK

So, we can use somebody else's work from a remote repository, but how can **you** prepare a repository to let other people pull from it?

You do your real work in your working tree that has your primary repository hanging under it as its .git subdirectory. You **could** make that repository accessible remotely and ask people to pull from it, but in practice that is not the way things are usually done. A recommended way is to have a public repository, make it reachable by other people, and when the changes you made in your primary working tree are in good shape, update the public repository from it. This is often called *pushing*.

Note

This public repository could further be mirrored, and that is how git repositories at kernel.org are managed.

Publishing the changes from your local (private) repository to your remote (public) repository requires a write privilege on the remote machine. You need to have an SSH account there to run a single command, *git−receive−pack*.

First, you need to create an empty repository on the remote machine that will house your public repository. This empty repository will be populated and be kept up−to−date by pushing into it later. Obviously, this repository creation needs to be done only once.

Note

git push uses a pair of commands, *git send−pack* on your local machine, and *git−receive−pack* on the remote machine. The communication between the two over the network internally uses an SSH connection.

Your private repository's git directory is usually .git, but your public repository is often named after the project name, i.e. <project>.git. Let's create such a public repository for project my−git. After logging into the remote machine, create an empty directory:

```
$ mkdir my−git.git
```

Then, make that directory into a git repository by running *git init*, but this time, since its name is not the usual .git, we do things slightly differently:

```
$ GIT_DIR=my−git.git git init
```

Make sure this directory is available for others you want your changes to be pulled via the transport of your choice. Also you need to make sure that you have the *git−receive−pack* program on the $PATH.

Note

Many installations of sshd do not invoke your shell as the login shell when you directly run programs; what this

means is that if your login shell is *bash*, only .bashrc is read and not .bash_profile. As a workaround, make sure .bashrc sets up $PATH so that you can run *git−receive−pack* program.

Note
If you plan to publish this repository to be accessed over http, you should do mv my−git.git/hooks/post−update.sample my−git.git/hooks/post−update at this point. This makes sure that every time you push into this repository, git update−server−info is run.

Your "public repository" is now ready to accept your changes. Come back to the machine you have your private repository. From there, run this command:

$ git push <public−host>:/path/to/my−git.git master

This synchronizes your public repository to match the named branch head (i.e. master in this case) and objects reachable from them in your current repository.

As a real example, this is how I update my public git repository. Kernel.org mirror network takes care of the propagation to other publicly visible machines:

$ git push master.kernel.org:/pub/scm/git/git.git/

PACKING YOUR REPOSITORY

Earlier, we saw that one file under .git/objects/??/ directory is stored for each git object you create. This representation is efficient to create atomically and safely, but not so convenient to transport over the network. Since git objects are immutable once they are created, there is a way to optimize the storage by "packing them together". The command

$ git repack

will do it for you. If you followed the tutorial examples, you would have accumulated about 17 objects in .git/objects/??/ directories by now. *git repack* tells you how many objects it packed, and stores the packed file in .git/objects/pack directory.

Note
You will see two files, pack−*.pack and pack−*.idx, in .git/objects/pack directory. They are closely related to each other, and if you ever copy them by hand to a different repository for whatever reason, you should make sure you copy them together. The former holds all the data from the objects in the pack, and the latter holds the index for random access.

If you are paranoid, running *git verify−pack* command would detect if you have a corrupt pack, but do not worry too much. Our programs are always perfect ;−).

Once you have packed objects, you do not need to leave the unpacked objects that are contained in the pack file anymore.

$ git prune−packed

would remove them for you.

You can try running find .git/objects −type f before and after you run git prune−packed if you are curious. Also git count−objects would tell you how many unpacked objects are in your repository and how much space they are consuming.

Note
git pull is slightly cumbersome for HTTP transport, as a packed repository may contain relatively few objects in a relatively large pack. If you expect many HTTP pulls from your public repository you might want to repack & prune often, or never.

If you run git repack again at this point, it will say "Nothing new to pack.". Once you continue your development and accumulate the changes, running git repack again will create a new pack, that contains objects created since you packed your repository the last time. We recommend that you pack your project soon after the initial import (unless you are starting your project from scratch), and then run git repack every once in a while, depending on how active your project is.

When a repository is synchronized via git push and git pull objects packed in the source repository are usually stored unpacked in the destination, unless rsync transport is used. While this allows you to use different packing strategies on both ends, it also means you may need to repack both repositories every once in a while.

WORKING WITH OTHERS

Although git is a truly distributed system, it is often convenient to organize your project with an informal hierarchy of developers. Linux kernel development is run this way. There is a nice illustration (page 17, "Merges to Mainline") in **Randy Dunlap's presentation**[2].

It should be stressed that this hierarchy is purely **informal**. There is nothing fundamental in git that enforces the "chain of patch flow" this hierarchy implies. You do not have to pull from only one remote repository.

A recommended workflow for a "project lead" goes like this:

1. Prepare your primary repository on your local machine. Your work is done there.
2. Prepare a public repository accessible to others.

   If other people are pulling from your repository over dumb transport protocols (HTTP), you need to keep this repository *dumb transport friendly*. After git init, $GIT_DIR/hooks/post−update.sample copied from the standard templates would contain a call to *git update−server−info* but you need to manually enable the hook with mv post−update.sample post−update. This makes sure *git update−server−info* keeps the necessary files up−to−date.
3. Push into the public repository from your primary repository.
4. *git repack* the public repository. This establishes a big pack that contains the initial set of objects as the baseline, and possibly *git prune* if the transport used for pulling from your repository supports packed repositories.
5. Keep working in your primary repository. Your changes include modifications of your own, patches you receive via e−mails, and merges resulting from pulling the "public" repositories of your "subsystem maintainers".

   You can repack this private repository whenever you feel like.
6. Push your changes to the public repository, and announce it to the public.
7. Every once in a while, *git repack* the public repository. Go back to step 5. and continue working.

A recommended work cycle for a "subsystem maintainer" who works on that project and has an own "public repository" goes like this:

1. Prepare your work repository, by *git clone* the public repository of the "project lead". The URL used for the initial cloning is stored in the remote.origin.url configuration variable.
2. Prepare a public repository accessible to others, just like the "project lead" person does.

3. Copy over the packed files from "project lead" public repository to your public repository, unless the "project lead" repository lives on the same machine as yours. In the latter case, you can use objects/info/alternates file to point at the repository you are borrowing from.
4. Push into the public repository from your primary repository. Run *git repack*, and possibly *git prune* if the transport used for pulling from your repository supports packed repositories.
5. Keep working in your primary repository. Your changes include modifications of your own, patches you receive via e−mails, and merges resulting from pulling the "public" repositories of your "project lead" and possibly your "sub−subsystem maintainers".

   You can repack this private repository whenever you feel like.
6. Push your changes to your public repository, and ask your "project lead" and possibly your "sub−subsystem maintainers" to pull from it.
7. Every once in a while, *git repack* the public repository. Go back to step 5. and continue working.

A recommended work cycle for an "individual developer" who does not have a "public" repository is somewhat different. It goes like this:

1. Prepare your work repository, by *git clone* the public repository of the "project lead" (or a "subsystem maintainer", if you work on a subsystem). The URL used for the initial cloning is stored in the remote.origin.url configuration variable.
2. Do your work in your repository on *master* branch.
3. Run git fetch origin from the public repository of your upstream every once in a while. This does only the first half of git pull but does not merge. The head of the public repository is stored in .git/refs/remotes/origin/master.
4. Use git cherry origin to see which ones of your patches were accepted, and/or use git rebase origin to port your unmerged changes forward to the updated upstream.
5. Use git format−patch origin to prepare patches for e−mail submission to your upstream and send it out. Go back to step 2. and continue.

WORKING WITH OTHERS, SHARED REPOSITORY STYLE

If you are coming from CVS background, the style of cooperation suggested in the previous section may be new to you. You do not have to worry. git supports "shared public repository" style of cooperation you are probably more familiar with as well.

See **gitcvs-migration**(7) for the details.

BUNDLING YOUR WORK TOGETHER

It is likely that you will be working on more than one thing at a time. It is easy to manage those more−or−less independent tasks using branches with git.

We have already seen how branches work previously, with "fun and work" example using two branches. The idea is the same if there are more than two branches. Let’s say you started out from "master" head, and have some new code in the "master" branch, and two independent fixes in the "commit−fix" and "diff−fix" branches:

```
$ git show−branch
! [commit−fix] Fix commit message normalization.
 ! [diff−fix] Fix rename detection.
  * [master] Release candidate #1
−−−
 + [diff−fix] Fix rename detection.
 + [diff−fix^1] Better common substring algorithm.
+  [commit−fix] Fix commit message normalization.
```

```
  * [master] Release candidate #1
++* [diff−fix^2] Pretty−print messages.
```

Both fixes are tested well, and at this point, you want to merge in both of them. You could merge in *diff−fix* first and then *commit−fix* next, like this:

```
$ git merge −m "Merge fix in diff−fix" diff−fix
$ git merge −m "Merge fix in commit−fix" commit−fix
```

Which would result in:

```
$ git show−branch
! [commit−fix] Fix commit message normalization.
 ! [diff−fix] Fix rename detection.
  * [master] Merge fix in commit−fix
−−−
  − [master] Merge fix in commit−fix
+ * [commit−fix] Fix commit message normalization.
  − [master^1] Merge fix in diff−fix
 +* [diff−fix] Fix rename detection.
 +* [diff−fix^1] Better common substring algorithm.
  * [master^2] Release candidate #1
++* [master^3] Pretty−print messages.
```

However, there is no particular reason to merge in one branch first and the other next, when what you have are a set of truly independent changes (if the order mattered, then they are not independent by definition). You could instead merge those two branches into the current branch at once. First let’s undo what we just did and start over. We would want to get the master branch before these two merges by resetting it to *master^2*:

```
$ git reset −−hard master^2
```

You can make sure git show−branch matches the state before those two *git merge* you just did. Then, instead of running two *git merge* commands in a row, you would merge these two branch heads (this is known as *making an Octopus*):

```
$ git merge commit−fix diff−fix
$ git show−branch
! [commit−fix] Fix commit message normalization.
 ! [diff−fix] Fix rename detection.
  * [master] Octopus merge of branches ´diff−fix´ and ´commit−fix´
−−−
  − [master] Octopus merge of branches ´diff−fix´ and ´commit−fix´
+ * [commit−fix] Fix commit message normalization.
 +* [diff−fix] Fix rename detection.
 +* [diff−fix^1] Better common substring algorithm.
  * [master^1] Release candidate #1
++* [master^2] Pretty−print messages.
```

Note that you should not do Octopus because you can. An octopus is a valid thing to do and often makes it easier to view the commit history if you are merging more than two independent changes at the same time. However, if you have merge conflicts with any of the branches you are merging in and need to hand resolve, that is an indication that the development happened in those branches were not independent after all, and you should merge two at a time, documenting how you resolved the conflicts, and the reason why you preferred changes made in one side over the other. Otherwise it would make the project history harder to follow, not easier.

SEE ALSO

gittutorial(7), **gittutorial-2**(7), **gitcvs-migration**(7), **git-help**(1), **Everyday git**[3], **The Git User's Manual**[1]

GIT

Part of the **git**(1) suite.

NOTES

1. the GIT User Manual
   file:///usr/share/doc/git-1.7.1/user-manual.html
2. Randy Dunlap's presentation
   http://www.xenotime.net/linux/mentor/linux-mentoring-2006.pdf
3. Everyday git
   file:///usr/share/doc/git-1.7.1/everyday.html