Git User Manual

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Git is a fast distributed revision control system.

This manual is designed to be readable by someone with basic UNIX command-line skills, but no previous knowledge of Git.

Chapter 1 and Chapter 2 explain how to fetch and study a project using git—read these chapters to learn how to build and test a particular version of a software project, search for regressions, and so on.

People needing to do actual development will also want to read Chapter 3 and Chapter 4.

Further chapters cover more specialized topics.

Comprehensive reference documentation is available through the man pages, or git-help(1) command. For example, for the command `git clone <repo>`, you can either use:

```
$ man git-clone
```

or:

```
$ git help clone
```

With the latter, you can use the manual viewer of your choice; see git-help(1) for more information.

See also Appendix A for a brief overview of Git commands, without any explanation.

Finally, see Appendix B for ways that you can help make this manual more complete.

Chapter 1

Repositories and Branches

1.1 How to get a Git repository

It will be useful to have a Git repository to experiment with as you read this manual.

The best way to get one is by using the git-clone(1) command to download a copy of an existing repository. If you don't already have a project in mind, here are some interesting examples:

```
        # Git itself (approx. 40MB download):
$ git clone git://git.kernel.org/pub/scm/git/git.git
        # the Linux kernel (approx. 640MB download):
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

The initial clone may be time-consuming for a large project, but you will only need to clone once.

The clone command creates a new directory named after the project (git or linux in the examples above). After you cd into this directory, you will see that it contains a copy of the project files, called the working tree, together with a special top-level directory named .git, which contains all the information about the history of the project.

1.2 How to check out a different version of a project

Git is best thought of as a tool for storing the history of a collection of files. It stores the history as a compressed collection of interrelated snapshots of the project's contents. In Git each such version is called a commit.

Those snapshots aren't necessarily all arranged in a single line from oldest to newest; instead, work may simultaneously proceed along parallel lines of development, called branches, which may merge and diverge.

A single Git repository can track development on multiple branches. It does this by keeping a list of heads which reference the latest commit on each branch; the git-branch(1) command shows you the list of branch heads:

```
$ git branch
* master
```

A freshly cloned repository contains a single branch head, by default named "master", with the working directory initialized to the state of the project referred to by that branch head.

Most projects also use tags. Tags, like heads, are references into the project's history, and can be listed using the git-tag(1) command:

```
$ git tag -l
v2.6.11
v2.6.11-tree
```

```
v2.6.12
v2.6.12-rc2
v2.6.12-rc3
v2.6.12-rc4
v2.6.12-rc5
v2.6.12-rc6
v2.6.13
...
```

Tags are expected to always point at the same version of a project, while heads are expected to advance as development progresses.

Create a new branch head pointing to one of these versions and check it out using git-checkout(1):

```
$ git checkout -b new v2.6.13
```

The working directory then reflects the contents that the project had when it was tagged v2.6.13, and git-branch(1) shows two branches, with an asterisk marking the currently checked-out branch:

```
$ git branch
  master
* new
```

If you decide that you'd rather see version 2.6.17, you can modify the current branch to point at v2.6.17 instead, with

```
$ git reset --hard v2.6.17
```

Note that if the current branch head was your only reference to a particular point in history, then resetting that branch may leave you with no way to find the history it used to point to; so use this command carefully.

1.3   **Understanding History: Commits**

Every change in the history of a project is represented by a commit. The git-show(1) command shows the most recent commit on the current branch:

```
$ git show
commit 17cf781661e6d38f737f15f53ab552f1e95960d7
Author: Linus Torvalds <torvalds@ppc970.osdl.org.(none)>
Date:   Tue Apr 19 14:11:06 2005 -0700

    Remove duplicate getenv(DB_ENVIRONMENT) call

    Noted by Tony Luck.

diff --git a/init-db.c b/init-db.c
index 65898fa..b002dc6 100644
--- a/init-db.c
+++ b/init-db.c
@@ -7,7 +7,7 @@

 int main(int argc, char **argv)
 {
-       char *sha1_dir = getenv(DB_ENVIRONMENT), *path;
+       char *sha1_dir, *path;
        int len, i;

        if (mkdir(".git", 0755) < 0) {
```

As you can see, a commit shows who made the latest change, what they did, and why.

Every commit has a 40-hexdigit id, sometimes called the "object name" or the "SHA-1 id", shown on the first line of the `git show` output. You can usually refer to a commit by a shorter name, such as a tag or a branch name, but this longer name can also be useful. Most importantly, it is a globally unique name for this commit: so if you tell somebody else the object name (for example in email), then you are guaranteed that name will refer to the same commit in their repository that it does in yours (assuming their repository has that commit at all). Since the object name is computed as a hash over the contents of the commit, you are guaranteed that the commit can never change without its name also changing.

In fact, in Chapter 7 we shall see that everything stored in Git history, including file data and directory contents, is stored in an object with a name that is a hash of its contents.

1.3.1   Understanding history: commits, parents, and reachability

Every commit (except the very first commit in a project) also has a parent commit which shows what happened before this commit. Following the chain of parents will eventually take you back to the beginning of the project.

However, the commits do not form a simple list; Git allows lines of development to diverge and then reconverge, and the point where two lines of development reconverge is called a "merge". The commit representing a merge can therefore have more than one parent, with each parent representing the most recent commit on one of the lines of development leading to that point.

The best way to see how this works is using the gitk(1) command; running gitk now on a Git repository and looking for merge commits will help understand how Git organizes history.

In the following, we say that commit X is "reachable" from commit Y if commit X is an ancestor of commit Y. Equivalently, you could say that Y is a descendant of X, or that there is a chain of parents leading from commit Y to commit X.

1.3.2   Understanding history: History diagrams

We will sometimes represent Git history using diagrams like the one below. Commits are shown as "o", and the links between them with lines drawn with - / and \. Time goes left to right:

```
        o--o--o <-- Branch A
       /
 o--o--o <-- master
       \
        o--o--o <-- Branch B
```

If we need to talk about a particular commit, the character "o" may be replaced with another letter or number.

1.3.3   Understanding history: What is a branch?

When we need to be precise, we will use the word "branch" to mean a line of development, and "branch head" (or just "head") to mean a reference to the most recent commit on a branch. In the example above, the branch head named "A" is a pointer to one particular commit, but we refer to the line of three commits leading up to that point as all being part of "branch A".

However, when no confusion will result, we often just use the term "branch" both for branches and for branch heads.

1.4   Manipulating branches

Creating, deleting, and modifying branches is quick and easy; here's a summary of the commands:

git branch
     list all branches.

git branch <branch>
     create a new branch named <branch>, referencing the same point in history as the current branch.

git branch <branch> <start-point>
    create a new branch named <branch>, referencing <start-point>, which may be specified any way you like, including using a branch name or a tag name.

git branch -d <branch>
    delete the branch <branch>; if the branch is not fully merged in its upstream branch or contained in the current branch, this command will fail with a warning.

git branch -D <branch>
    delete the branch <branch> irrespective of its merged status.

git checkout <branch>
    make the current branch <branch>, updating the working directory to reflect the version referenced by <branch>.

git checkout -b <new> <start-point>
    create a new branch <new> referencing <start-point>, and check it out.

The special symbol "HEAD" can always be used to refer to the current branch. In fact, Git uses a file named HEAD in the .git directory to remember which branch is current:

```
$ cat .git/HEAD
ref: refs/heads/master
```

1.5 Examining an old version without creating a new branch

The git checkout command normally expects a branch head, but will also accept an arbitrary commit; for example, you can check out the commit referenced by a tag:

```
$ git checkout v2.6.17
Note: checking out 'v2.6.17'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 427abfa Linux v2.6.17
```

The HEAD then refers to the SHA-1 of the commit instead of to a branch, and git branch shows that you are no longer on a branch:

```
$ cat .git/HEAD
427abfa28afedffadfca9dd8b067eb6d36bac53f
$ git branch
* (detached from v2.6.17)
  master
```

In this case we say that the HEAD is "detached".

This is an easy way to check out a particular version without having to make up a name for the new branch. You can still create a new branch (or tag) for this version later if you decide to.

1.6   Examining branches from a remote repository

The "master" branch that was created at the time you cloned is a copy of the HEAD in the repository that you cloned from. That repository may also have had other branches, though, and your local repository keeps branches which track each of those remote branches, called remote-tracking branches, which you can view using the `-r` option to git-branch(1):

```
$ git branch -r
  origin/HEAD
  origin/html
  origin/maint
  origin/man
  origin/master
  origin/next
  origin/pu
  origin/todo
```

In this example, "origin" is called a remote repository, or "remote" for short. The branches of this repository are called "remote branches" from our point of view. The remote-tracking branches listed above were created based on the remote branches at clone time and will be updated by `git fetch` (hence `git pull`) and `git push`. See Section 1.8 for details.

You might want to build on one of these remote-tracking branches on a branch of your own, just as you would for a tag:

```
$ git checkout -b my-todo-copy origin/todo
```

You can also check out `origin/todo` directly to examine it or write a one-off patch. See detached head.

Note that the name "origin" is just the name that Git uses by default to refer to the repository that you cloned from.

1.7   Naming branches, tags, and other references

Branches, remote-tracking branches, and tags are all references to commits. All references are named with a slash-separated path name starting with `refs`; the names we've been using so far are actually shorthand:

- The branch `test` is short for `refs/heads/test`.

- The tag `v2.6.18` is short for `refs/tags/v2.6.18`.

- `origin/master` is short for `refs/remotes/origin/master`.

The full name is occasionally useful if, for example, there ever exists a tag and a branch with the same name.

(Newly created refs are actually stored in the `.git/refs` directory, under the path given by their name. However, for efficiency reasons they may also be packed together in a single file; see git-pack-refs(1)).

As another useful shortcut, the "HEAD" of a repository can be referred to just using the name of that repository. So, for example, "origin" is usually a shortcut for the HEAD branch in the repository "origin".

For the complete list of paths which Git checks for references, and the order it uses to decide which to choose when there are multiple references with the same shorthand name, see the "SPECIFYING REVISIONS" section of gitrevisions(7).

1.8   Updating a repository with git fetch

After you clone a repository and commit a few changes of your own, you may wish to check the original repository for updates.

The `git-fetch` command, with no arguments, will update all of the remote-tracking branches to the latest version found in the original repository. It will not touch any of your own branches—not even the "master" branch that was created for you on clone.

1.9   Fetching branches from other repositories

You can also track branches from repositories other than the one you cloned from, using git-remote(1):

```
$ git remote add staging git://git.kernel.org/.../gregkh/staging.git
$ git fetch staging
...
From git://git.kernel.org/pub/scm/linux/kernel/git/gregkh/staging
 * [new branch]      master      -> staging/master
 * [new branch]      staging-linus -> staging/staging-linus
 * [new branch]      staging-next -> staging/staging-next
```

New remote-tracking branches will be stored under the shorthand name that you gave git remote add, in this case staging:

```
$ git branch -r
  origin/HEAD -> origin/master
  origin/master
  staging/master
  staging/staging-linus
  staging/staging-next
```

If you run git fetch <remote> later, the remote-tracking branches for the named <remote> will be updated.

If you examine the file .git/config, you will see that Git has added a new stanza:

```
$ cat .git/config
...
[remote "staging"]
        url = git://git.kernel.org/pub/scm/linux/kernel/git/gregkh/staging.git
        fetch = +refs/heads/*:refs/remotes/staging/*
...
```

This is what causes Git to track the remote's branches; you may modify or delete these configuration options by editing .git/config with a text editor. (See the "CONFIGURATION FILE" section of git-config(1) for details.)

Chapter 2

Exploring Git history

Git is best thought of as a tool for storing the history of a collection of files. It does this by storing compressed snapshots of the contents of a file hierarchy, together with "commits" which show the relationships between these snapshots.

Git provides extremely flexible and fast tools for exploring the history of a project.

We start with one specialized tool that is useful for finding the commit that introduced a bug into a project.

2.1 How to use bisect to find a regression

Suppose version 2.6.18 of your project worked, but the version at "master" crashes. Sometimes the best way to find the cause of such a regression is to perform a brute-force search through the project's history to find the particular commit that caused the problem. The git-bisect(1) command can help you do this:

```
$ git bisect start
$ git bisect good v2.6.18
$ git bisect bad master
Bisecting: 3537 revisions left to test after this
[65934a9a028b88e83e2b0f8b36618fe503349f8e] BLOCK: Make USB storage depend on SCSI  ←
    rather than selecting it [try #6]
```

If you run `git branch` at this point, you'll see that Git has temporarily moved you in "(no branch)". HEAD is now detached from any branch and points directly to a commit (with commit id 65934) that is reachable from "master" but not from v2.6.18. Compile and test it, and see whether it crashes. Assume it does crash. Then:

```
$ git bisect bad
Bisecting: 1769 revisions left to test after this
[7eff82c8b1511017ae605f0c99ac275a7e21b867] i2c-core: Drop useless bitmaskings
```

checks out an older version. Continue like this, telling Git at each stage whether the version it gives you is good or bad, and notice that the number of revisions left to test is cut approximately in half each time.

After about 13 tests (in this case), it will output the commit id of the guilty commit. You can then examine the commit with git-show(1), find out who wrote it, and mail them your bug report with the commit id. Finally, run

```
$ git bisect reset
```

to return you to the branch you were on before.

Note that the version which `git bisect` checks out for you at each point is just a suggestion, and you're free to try a different version if you think it would be a good idea. For example, occasionally you may land on a commit that broke something unrelated; run

```
$ git bisect visualize
```

which will run gitk and label the commit it chose with a marker that says "bisect". Choose a safe-looking commit nearby, note its commit id, and check it out with:

```
$ git reset --hard fb47ddb2db
```

then test, run `bisect good` or `bisect bad` as appropriate, and continue.

Instead of `git bisect visualize` and then `git reset --hard fb47ddb2db`, you might just want to tell Git that you want to skip the current commit:

```
$ git bisect skip
```

In this case, though, Git may not eventually be able to tell the first bad one between some first skipped commits and a later bad commit.

There are also ways to automate the bisecting process if you have a test script that can tell a good from a bad commit. See git-bisect(1) for more information about this and other `git bisect` features.

2.2 Naming commits

We have seen several ways of naming commits already:

- 40-hexdigit object name

- branch name: refers to the commit at the head of the given branch

- tag name: refers to the commit pointed to by the given tag (we've seen branches and tags are special cases of references).

- HEAD: refers to the head of the current branch

There are many more; see the "SPECIFYING REVISIONS" section of the gitrevisions(7) man page for the complete list of ways to name revisions. Some examples:

```
$ git show fb47ddb2 # the first few characters of the object name
                    # are usually enough to specify it uniquely
$ git show HEAD^    # the parent of the HEAD commit
$ git show HEAD^^   # the grandparent
$ git show HEAD~4   # the great-great-grandparent
```

Recall that merge commits may have more than one parent; by default, ^ and ~ follow the first parent listed in the commit, but you can also choose:

```
$ git show HEAD^1   # show the first parent of HEAD
$ git show HEAD^2   # show the second parent of HEAD
```

In addition to HEAD, there are several other special names for commits:

Merges (to be discussed later), as well as operations such as `git reset`, which change the currently checked-out commit, generally set ORIG_HEAD to the value HEAD had before the current operation.

The `git fetch` operation always stores the head of the last fetched branch in FETCH_HEAD. For example, if you run `git fetch` without specifying a local branch as the target of the operation

```
$ git fetch git://example.com/proj.git theirbranch
```

the fetched commits will still be available from FETCH_HEAD.

When we discuss merges we'll also see the special name MERGE_HEAD, which refers to the other branch that we're merging in to the current branch.

The git-rev-parse(1) command is a low-level command that is occasionally useful for translating some name for a commit to the object name for that commit:

```
$ git rev-parse origin
e05db0fd4f31dde7005f075a84f96b360d05984b
```

2.3 Creating tags

We can also create a tag to refer to a particular commit; after running

```
$ git tag stable-1 1b2e1d63ff
```

You can use `stable-1` to refer to the commit 1b2e1d63ff.

This creates a "lightweight" tag. If you would also like to include a comment with the tag, and possibly sign it cryptographically, then you should create a tag object instead; see the git-tag(1) man page for details.

2.4 Browsing revisions

The git-log(1) command can show lists of commits. On its own, it shows all commits reachable from the parent commit; but you can also make more specific requests:

```
$ git log v2.5..          # commits since (not reachable from) v2.5
$ git log test..master    # commits reachable from master but not test
$ git log master..test    # ...reachable from test but not master
$ git log master...test   # ...reachable from either test or master,
                          #    but not both
$ git log --since="2 weeks ago" # commits from the last 2 weeks
$ git log Makefile        # commits which modify Makefile
$ git log fs/             # ... which modify any file under fs/
$ git log -S'foo()'       # commits which add or remove any file data
                          # matching the string 'foo()'
```

And of course you can combine all of these; the following finds commits since v2.5 which touch the `Makefile` or any file under `fs`:

```
$ git log v2.5.. Makefile fs/
```

You can also ask git log to show patches:

```
$ git log -p
```

See the `--pretty` option in the git-log(1) man page for more display options.

Note that git log starts with the most recent commit and works backwards through the parents; however, since Git history can contain multiple independent lines of development, the particular order that commits are listed in may be somewhat arbitrary.

2.5   Generating diffs

You can generate diffs between any two versions using git-diff(1):

```
$ git diff master..test
```

That will produce the diff between the tips of the two branches. If you'd prefer to find the diff from their common ancestor to test, you can use three dots instead of two:

```
$ git diff master...test
```

Sometimes what you want instead is a set of patches; for this you can use git-format-patch(1):

```
$ git format-patch master..test
```

will generate a file with a patch for each commit reachable from test but not from master.

2.6   Viewing old file versions

You can always view an old version of a file by just checking out the correct revision first. But sometimes it is more convenient to be able to view an old version of a single file without checking anything out; this command does that:

```
$ git show v2.5:fs/locks.c
```

Before the colon may be anything that names a commit, and after it may be any path to a file tracked by Git.

2.7   Examples

2.7.1   Counting the number of commits on a branch

Suppose you want to know how many commits you've made on mybranch since it diverged from origin:

```
$ git log --pretty=oneline origin..mybranch | wc -l
```

Alternatively, you may often see this sort of thing done with the lower-level command git-rev-list(1), which just lists the SHA-1's of all the given commits:

```
$ git rev-list origin..mybranch | wc -l
```

2.7.2   Check whether two branches point at the same history

Suppose you want to check whether two branches point at the same point in history.

```
$ git diff origin..master
```

will tell you whether the contents of the project are the same at the two branches; in theory, however, it's possible that the same project contents could have been arrived at by two different historical routes. You could compare the object names:

```
$ git rev-list origin
e05db0fd4f31dde7005f075a84f96b360d05984b
$ git rev-list master
e05db0fd4f31dde7005f075a84f96b360d05984b
```

Or you could recall that the `...` operator selects all commits reachable from either one reference or the other but not both; so

```
$ git log origin...master
```

will return no commits when the two branches are equal.

2.7.3 Find first tagged version including a given fix

Suppose you know that the commit e05db0fd fixed a certain problem. You'd like to find the earliest tagged release that contains that fix.

Of course, there may be more than one answer—if the history branched after commit e05db0fd, then there could be multiple "earliest" tagged releases.

You could just visually inspect the commits since e05db0fd:

```
$ gitk e05db0fd..
```

or you can use git-name-rev(1), which will give the commit a name based on any tag it finds pointing to one of the commit's descendants:

```
$ git name-rev --tags e05db0fd
e05db0fd tags/v1.5.0-rc1^0~23
```

The git-describe(1) command does the opposite, naming the revision using a tag on which the given commit is based:

```
$ git describe e05db0fd
v1.5.0-rc0-260-ge05db0f
```

but that may sometimes help you guess which tags might come after the given commit.

If you just want to verify whether a given tagged version contains a given commit, you could use git-merge-base(1):

```
$ git merge-base e05db0fd v1.5.0-rc1
e05db0fd4f31dde7005f075a84f96b360d05984b
```

The merge-base command finds a common ancestor of the given commits, and always returns one or the other in the case where one is a descendant of the other; so the above output shows that e05db0fd actually is an ancestor of v1.5.0-rc1.

Alternatively, note that

```
$ git log v1.5.0-rc1..e05db0fd
```

will produce empty output if and only if v1.5.0-rc1 includes e05db0fd, because it outputs only commits that are not reachable from v1.5.0-rc1.

As yet another alternative, the git-show-branch(1) command lists the commits reachable from its arguments with a display on the left-hand side that indicates which arguments that commit is reachable from. So, if you run something like

```
$ git show-branch e05db0fd v1.5.0-rc0 v1.5.0-rc1 v1.5.0-rc2
! [e05db0fd] Fix warnings in sha1_file.c - use C99 printf format if
available
 ! [v1.5.0-rc0] GIT v1.5.0 preview
  ! [v1.5.0-rc1] GIT v1.5.0-rc1
   ! [v1.5.0-rc2] GIT v1.5.0-rc2
...
```

then a line like

```
+ ++ [e05db0fd] Fix warnings in sha1_file.c - use C99 printf format if
available
```

shows that e05db0fd is reachable from itself, from v1.5.0-rc1, and from v1.5.0-rc2, and not from v1.5.0-rc0.

2.7.4 Showing commits unique to a given branch

Suppose you would like to see all the commits reachable from the branch head named `master` but not from any other head in your repository.

We can list all the heads in this repository with git-show-ref(1):

```
$ git show-ref --heads
bf62196b5e363d73353a9dcf094c59595f3153b7 refs/heads/core-tutorial
db768d5504c1bb46f63ee9d6e1772bd047e05bf9 refs/heads/maint
a07157ac624b2524a059a3414e99f6f44bebc1e7 refs/heads/master
24dbc180ea14dc1aebe09f14c8ecf32010690627 refs/heads/tutorial-2
1e87486ae06626c2f31eaa63d26fc0fd646c8af2 refs/heads/tutorial-fixes
```

We can get just the branch-head names, and remove `master`, with the help of the standard utilities cut and grep:

```
$ git show-ref --heads | cut -d' ' -f2 | grep -v '^refs/heads/master'
refs/heads/core-tutorial
refs/heads/maint
refs/heads/tutorial-2
refs/heads/tutorial-fixes
```

And then we can ask to see all the commits reachable from master but not from these other heads:

```
$ gitk master --not $( git show-ref --heads | cut -d' ' -f2 |
                                 grep -v '^refs/heads/master' )
```

Obviously, endless variations are possible; for example, to see all commits reachable from some head but not from any tag in the repository:

```
$ gitk $( git show-ref --heads ) --not  $( git show-ref --tags )
```

(See gitrevisions(7) for explanations of commit-selecting syntax such as `--not`.)

2.7.5 Creating a changelog and tarball for a software release

The git-archive(1) command can create a tar or zip archive from any version of a project; for example:

```
$ git archive -o latest.tar.gz --prefix=project/ HEAD
```

will use HEAD to produce a gzipped tar archive in which each filename is preceded by `project/`. The output file format is inferred from the output file extension if possible, see git-archive(1) for details.

Versions of Git older than 1.7.7 don't know about the `tar.gz` format, you'll need to use gzip explicitly:

```
$ git archive --format=tar --prefix=project/ HEAD | gzip >latest.tar.gz
```

If you're releasing a new version of a software project, you may want to simultaneously make a changelog to include in the release announcement.

Linus Torvalds, for example, makes new kernel releases by tagging them, then running:

```
$ release-script 2.6.12 2.6.13-rc6 2.6.13-rc7
```

where release-script is a shell script that looks like:

```
#!/bin/sh
stable="$1"
last="$2"
new="$3"
echo "# git tag v$new"
echo "git archive --prefix=linux-$new/ v$new | gzip -9 > ../linux-$new.tar.gz"
echo "git diff v$stable v$new | gzip -9 > ../patch-$new.gz"
echo "git log --no-merges v$new ^v$last > ../ChangeLog-$new"
echo "git shortlog --no-merges v$new ^v$last > ../ShortLog"
echo "git diff --stat --summary -M v$last v$new > ../diffstat-$new"
```

and then he just cut-and-pastes the output commands after verifying that they look OK.

2.7.6 Finding commits referencing a file with given content

Somebody hands you a copy of a file, and asks which commits modified a file such that it contained the given content either before or after the commit. You can find out with this:

```
$  git log --raw --abbrev=40 --pretty=oneline |
        grep -B 1 `git hash-object filename`
```

Figuring out why this works is left as an exercise to the (advanced) student. The git-log(1), git-diff-tree(1), and git-hash-object(1) man pages may prove helpful.

Chapter 3

Developing with Git

3.1   Telling Git your name

Before creating any commits, you should introduce yourself to Git. The easiest way to do so is to use git-config(1):

```
$ git config --global user.name 'Your Name Comes Here'
$ git config --global user.email 'you@yourdomain.example.com'
```

Which will add the following to a file named .gitconfig in your home directory:

```
[user]
        name = Your Name Comes Here
        email = you@yourdomain.example.com
```

See the "CONFIGURATION FILE" section of git-config(1) for details on the configuration file. The file is plain text, so you can also edit it with your favorite editor.

3.2   Creating a new repository

Creating a new repository from scratch is very easy:

```
$ mkdir project
$ cd project
$ git init
```

If you have some initial content (say, a tarball):

```
$ tar xzvf project.tar.gz
$ cd project
$ git init
$ git add . # include everything below ./ in the first commit:
$ git commit
```

3.3   How to make a commit

Creating a new commit takes three steps:

1. Making some changes to the working directory using your favorite editor.

2. Telling Git about your changes.

3. Creating the commit using the content you told Git about in step 2.

In practice, you can interleave and repeat steps 1 and 2 as many times as you want: in order to keep track of what you want committed at step 3, Git maintains a snapshot of the tree's contents in a special staging area called "the index."

At the beginning, the content of the index will be identical to that of the HEAD. The command `git diff --cached`, which shows the difference between the HEAD and the index, should therefore produce no output at that point.

Modifying the index is easy:

To update the index with the contents of a new or modified file, use

```
$ git add path/to/file
```

To remove a file from the index and from the working tree, use

```
$ git rm path/to/file
```

After each step you can verify that

```
$ git diff --cached
```

always shows the difference between the HEAD and the index file—this is what you'd commit if you created the commit now—and that

```
$ git diff
```

shows the difference between the working tree and the index file.

Note that `git add` always adds just the current contents of a file to the index; further changes to the same file will be ignored unless you run `git add` on the file again.

When you're ready, just run

```
$ git commit
```

and Git will prompt you for a commit message and then create the new commit. Check to make sure it looks like what you expected with

```
$ git show
```

As a special shortcut,

```
$ git commit -a
```

will update the index with any files that you've modified or removed and create a commit, all in one step.

A number of commands are useful for keeping track of what you're about to commit:

```
$ git diff --cached # difference between HEAD and the index; what
                    # would be committed if you ran "commit" now.
$ git diff          # difference between the index file and your
                    # working directory; changes that would not
                    # be included if you ran "commit" now.
$ git diff HEAD     # difference between HEAD and working tree; what
                    # would be committed if you ran "commit -a" now.
$ git status        # a brief per-file summary of the above.
```

You can also use git-gui(1) to create commits, view changes in the index and the working tree files, and individually select diff hunks for inclusion in the index (by right-clicking on the diff hunk and choosing "Stage Hunk For Commit").

3.4   Creating good commit messages

Though not required, it's a good idea to begin the commit message with a single short (less than 50 character) line summarizing the change, followed by a blank line and then a more thorough description. The text up to the first blank line in a commit message is treated as the commit title, and that title is used throughout Git. For example, git-format-patch(1) turns a commit into email, and it uses the title on the Subject line and the rest of the commit in the body.

3.5   Ignoring files

A project will often generate files that you do *not* want to track with Git. This typically includes files generated by a build process or temporary backup files made by your editor. Of course, *not* tracking files with Git is just a matter of *not* calling `git add` on them. But it quickly becomes annoying to have these untracked files lying around; e.g. they make `git add .` practically useless, and they keep showing up in the output of `git status`.

You can tell Git to ignore certain files by creating a file called `.gitignore` in the top level of your working directory, with contents such as:

```
# Lines starting with '#' are considered comments.
# Ignore any file named foo.txt.
foo.txt
# Ignore (generated) html files,
*.html
# except foo.html which is maintained by hand.
!foo.html
# Ignore objects and archives.
*.[oa]
```

See gitignore(5) for a detailed explanation of the syntax. You can also place .gitignore files in other directories in your working tree, and they will apply to those directories and their subdirectories. The `.gitignore` files can be added to your repository like any other files (just run `git add .gitignore` and `git commit`, as usual), which is convenient when the exclude patterns (such as patterns matching build output files) would also make sense for other users who clone your repository.

If you wish the exclude patterns to affect only certain repositories (instead of every repository for a given project), you may instead put them in a file in your repository named `.git/info/exclude`, or in any file specified by the `core.excludesFile` configuration variable. Some Git commands can also take exclude patterns directly on the command line. See gitignore(5) for the details.

3.6   How to merge

You can rejoin two diverging branches of development using git-merge(1):

```
$ git merge branchname
```

merges the development in the branch `branchname` into the current branch.

A merge is made by combining the changes made in `branchname` and the changes made up to the latest commit in your current branch since their histories forked. The work tree is overwritten by the result of the merge when this combining is done cleanly, or overwritten by a half-merged results when this combining results in conflicts. Therefore, if you have uncommitted changes touching the same files as the ones impacted by the merge, Git will refuse to proceed. Most of the time, you will want to commit your changes before you can merge, and if you don't, then git-stash(1) can take these changes away while you're doing the merge, and reapply them afterwards.

If the changes are independent enough, Git will automatically complete the merge and commit the result (or reuse an existing commit in case of fast-forward, see below). On the other hand, if there are conflicts—for example, if the same file is modified in two different ways in the remote branch and the local branch—then you are warned; the output may look something like this:

```
$ git merge next
 100% (4/4) done
Auto-merged file.txt
CONFLICT (content): Merge conflict in file.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Conflict markers are left in the problematic files, and after you resolve the conflicts manually, you can update the index with the contents and run Git commit, as you normally would when creating a new file.

If you examine the resulting commit using gitk, you will see that it has two parents, one pointing to the top of the current branch, and one to the top of the other branch.

3.7 Resolving a merge

When a merge isn't resolved automatically, Git leaves the index and the working tree in a special state that gives you all the information you need to help resolve the merge.

Files with conflicts are marked specially in the index, so until you resolve the problem and update the index, git-commit(1) will fail:

```
$ git commit
file.txt: needs merge
```

Also, git-status(1) will list those files as "unmerged", and the files with conflicts will have conflict markers added, like this:

```
<<<<<<< HEAD:file.txt
Hello world
=======
Goodbye
>>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt
```

All you need to do is edit the files to resolve the conflicts, and then

```
$ git add file.txt
$ git commit
```

Note that the commit message will already be filled in for you with some information about the merge. Normally you can just use this default message unchanged, but you may add additional commentary of your own if desired.

The above is all you need to know to resolve a simple merge. But Git also provides more information to help resolve conflicts:

3.7.1 Getting conflict-resolution help during a merge

All of the changes that Git was able to merge automatically are already added to the index file, so git-diff(1) shows only the conflicts. It uses an unusual syntax:

```
$ git diff
diff --cc file.txt
index 802992c,2b60207..0000000
--- a/file.txt
+++ b/file.txt
@@@ -1,1 -1,1 +1,5 @@@
++<<<<<<< HEAD:file.txt
 +Hello world
++=======
+ Goodbye
++>>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt
```

Recall that the commit which will be committed after we resolve this conflict will have two parents instead of the usual one: one parent will be HEAD, the tip of the current branch; the other will be the tip of the other branch, which is stored temporarily in MERGE_HEAD.

During the merge, the index holds three versions of each file. Each of these three "file stages" represents a different version of the file:

```
$ git show :1:file.txt  # the file in a common ancestor of both branches
$ git show :2:file.txt  # the version from HEAD.
$ git show :3:file.txt  # the version from MERGE_HEAD.
```

When you ask git-diff(1) to show the conflicts, it runs a three-way diff between the conflicted merge results in the work tree with stages 2 and 3 to show only hunks whose contents come from both sides, mixed (in other words, when a hunk's merge results come only from stage 2, that part is not conflicting and is not shown. Same for stage 3).

The diff above shows the differences between the working-tree version of file.txt and the stage 2 and stage 3 versions. So instead of preceding each line by a single + or −, it now uses two columns: the first column is used for differences between the first parent and the working directory copy, and the second for differences between the second parent and the working directory copy. (See the "COMBINED DIFF FORMAT" section of git-diff-files(1) for a details of the format.)

After resolving the conflict in the obvious way (but before updating the index), the diff will look like:

```
$ git diff
diff --cc file.txt
index 802992c,2b60207..0000000
--- a/file.txt
+++ b/file.txt
@@@ -1,1 -1,1 +1,1 @@@
- Hello world
 -Goodbye
++Goodbye world
```

This shows that our resolved version deleted "Hello world" from the first parent, deleted "Goodbye" from the second parent, and added "Goodbye world", which was previously absent from both.

Some special diff options allow diffing the working directory against any of these stages:

```
$ git diff -1 file.txt          # diff against stage 1
$ git diff --base file.txt      # same as the above
$ git diff -2 file.txt          # diff against stage 2
$ git diff --ours file.txt      # same as the above
$ git diff -3 file.txt          # diff against stage 3
$ git diff --theirs file.txt    # same as the above.
```

The git-log(1) and gitk(1) commands also provide special help for merges:

```
$ git log --merge
$ gitk --merge
```

These will display all commits which exist only on HEAD or on MERGE_HEAD, and which touch an unmerged file.

You may also use git-mergetool(1), which lets you merge the unmerged files using external tools such as Emacs or kdiff3.

Each time you resolve the conflicts in a file and update the index:

```
$ git add file.txt
```

the different stages of that file will be "collapsed", after which `git diff` will (by default) no longer show diffs for that file.

3.8 Undoing a merge

If you get stuck and decide to just give up and throw the whole mess away, you can always return to the pre-merge state with

```
$ git reset --hard HEAD
```

Or, if you've already committed the merge that you want to throw away,

```
$ git reset --hard ORIG_HEAD
```

However, this last command can be dangerous in some cases—never throw away a commit you have already committed if that commit may itself have been merged into another branch, as doing so may confuse further merges.

3.9 Fast-forward merges

There is one special case not mentioned above, which is treated differently. Normally, a merge results in a merge commit, with two parents, one pointing at each of the two lines of development that were merged.

However, if the current branch is an ancestor of the other—so every commit present in the current branch is already contained in the other branch—then Git just performs a "fast-forward"; the head of the current branch is moved forward to point at the head of the merged-in branch, without any new commits being created.

3.10 Fixing mistakes

If you've messed up the working tree, but haven't yet committed your mistake, you can return the entire working tree to the last committed state with

```
$ git reset --hard HEAD
```

If you make a commit that you later wish you hadn't, there are two fundamentally different ways to fix the problem:

1. You can create a new commit that undoes whatever was done by the old commit. This is the correct thing if your mistake has already been made public.

2. You can go back and modify the old commit. You should never do this if you have already made the history public; Git does not normally expect the "history" of a project to change, and cannot correctly perform repeated merges from a branch that has had its history changed.

3.10.1 Fixing a mistake with a new commit

Creating a new commit that reverts an earlier change is very easy; just pass the git-revert(1) command a reference to the bad commit; for example, to revert the most recent commit:

```
$ git revert HEAD
```

This will create a new commit which undoes the change in HEAD. You will be given a chance to edit the commit message for the new commit.

You can also revert an earlier change, for example, the next-to-last:

```
$ git revert HEAD^
```

In this case Git will attempt to undo the old change while leaving intact any changes made since then. If more recent changes overlap with the changes to be reverted, then you will be asked to fix conflicts manually, just as in the case of resolving a merge.

3.10.2   Fixing a mistake by rewriting history

If the problematic commit is the most recent commit, and you have not yet made that commit public, then you may just destroy it using git reset.

Alternatively, you can edit the working directory and update the index to fix your mistake, just as if you were going to create a new commit, then run

```
$ git commit --amend
```

which will replace the old commit by a new commit incorporating your changes, giving you a chance to edit the old commit message first.

Again, you should never do this to a commit that may already have been merged into another branch; use git-revert(1) instead in that case.

It is also possible to replace commits further back in the history, but this is an advanced topic to be left for another chapter.

3.10.3   Checking out an old version of a file

In the process of undoing a previous bad change, you may find it useful to check out an older version of a particular file using git-checkout(1). We've used `git checkout` before to switch branches, but it has quite different behavior if it is given a path name: the command

```
$ git checkout HEAD^ path/to/file
```

replaces path/to/file by the contents it had in the commit HEAD^, and also updates the index to match. It does not change branches.

If you just want to look at an old version of the file, without modifying the working directory, you can do that with git-show(1):

```
$ git show HEAD^:path/to/file
```

which will display the given version of the file.

3.10.4   Temporarily setting aside work in progress

While you are in the middle of working on something complicated, you find an unrelated but obvious and trivial bug. You would like to fix it before continuing. You can use git-stash(1) to save the current state of your work, and after fixing the bug (or, optionally after doing so on a different branch and then coming back), unstash the work-in-progress changes.

```
$ git stash push -m "work in progress for foo feature"
```

This command will save your changes away to the `stash`, and reset your working tree and the index to match the tip of your current branch. Then you can make your fix as usual.

```
... edit and test ...
$ git commit -a -m "blorpl: typofix"
```

After that, you can go back to what you were working on with `git stash pop`:

```
$ git stash pop
```

3.11   Ensuring good performance

On large repositories, Git depends on compression to keep the history information from taking up too much space on disk or in memory. Some Git commands may automatically run git-gc(1), so you don't have to worry about running it manually. However, compressing a large repository may take a while, so you may want to call `gc` explicitly to avoid automatic compression kicking in when it is not convenient.

3.12 Ensuring reliability

3.12.1 Checking the repository for corruption

The git-fsck(1) command runs a number of self-consistency checks on the repository, and reports on any problems. This may take some time.

```
$ git fsck
dangling commit 7281251ddd2a61e38657c827739c57015671a6b3
dangling commit 2706a059f258c6b245f298dc4ff2ccd30ec21a63
dangling commit 13472b7c4b80851a1bc551779171dcb03655e9b5
dangling blob 218761f9d90712d37a9c5e36f406f92202db07eb
dangling commit bf093535a34a4d35731aa2bd90fe6b176302f14f
dangling commit 8e4bec7f2ddaa268bef999853c25755452100f8e
dangling tree d50bb86186bf27b681d25af89d3b5b68382e4085
dangling tree b24c2473f1fd3d91352a624795be026d64c8841f
...
```

You will see informational messages on dangling objects. They are objects that still exist in the repository but are no longer referenced by any of your branches, and can (and will) be removed after a while with `gc`. You can run `git fsck --no-dangling` to suppress these messages, and still view real errors.

3.12.2 Recovering lost changes

3.12.2.1 Reflogs

Say you modify a branch with git reset --hard, and then realize that the branch was the only reference you had to that point in history.

Fortunately, Git also keeps a log, called a "reflog", of all the previous values of each branch. So in this case you can still find the old history using, for example,

```
$ git log master@{1}
```

This lists the commits reachable from the previous version of the `master` branch head. This syntax can be used with any Git command that accepts a commit, not just with `git log`. Some other examples:

```
$ git show master@{2}        # See where the branch pointed 2,
$ git show master@{3}        # 3, ... changes ago.
$ gitk master@{yesterday}    # See where it pointed yesterday,
$ gitk master@{"1 week ago"} # ... or last week
$ git log --walk-reflogs master # show reflog entries for master
```

A separate reflog is kept for the HEAD, so

```
$ git show HEAD@{"1 week ago"}
```

will show what HEAD pointed to one week ago, not what the current branch pointed to one week ago. This allows you to see the history of what you've checked out.

The reflogs are kept by default for 30 days, after which they may be pruned. See git-reflog(1) and git-gc(1) to learn how to control this pruning, and see the "SPECIFYING REVISIONS" section of gitrevisions(7) for details.

Note that the reflog history is very different from normal Git history. While normal history is shared by every repository that works on the same project, the reflog history is not shared: it tells you only about how the branches in your local repository have changed over time.

3.12.2.2 Examining dangling objects

In some situations the reflog may not be able to save you. For example, suppose you delete a branch, then realize you need the history it contained. The reflog is also deleted; however, if you have not yet pruned the repository, then you may still be able to find the lost commits in the dangling objects that `git fsck` reports. See Section 7.1.7 for the details.

```
$ git fsck
dangling commit 7281251ddd2a61e38657c827739c57015671a6b3
dangling commit 2706a059f258c6b245f298dc4ff2ccd30ec21a63
dangling commit 13472b7c4b80851a1bc551779171dcb03655e9b5
...
```

You can examine one of those dangling commits with, for example,

```
$ gitk 7281251ddd --not --all
```

which does what it sounds like: it says that you want to see the commit history that is described by the dangling commit(s), but not the history that is described by all your existing branches and tags. Thus you get exactly the history reachable from that commit that is lost. (And notice that it might not be just one commit: we only report the "tip of the line" as being dangling, but there might be a whole deep and complex commit history that was dropped.)

If you decide you want the history back, you can always create a new reference pointing to it, for example, a new branch:

```
$ git branch recovered-branch 7281251ddd
```

Other types of dangling objects (blobs and trees) are also possible, and dangling objects can arise in other situations.

Chapter 4

Sharing development with others

4.1 Getting updates with git pull

After you clone a repository and commit a few changes of your own, you may wish to check the original repository for updates and merge them into your own work.

We have already seen how to keep remote-tracking branches up to date with git-fetch(1), and how to merge two branches. So you can merge in changes from the original repository's master branch with:

```
$ git fetch
$ git merge origin/master
```

However, the git-pull(1) command provides a way to do this in one step:

```
$ git pull origin master
```

In fact, if you have master checked out, then this branch has been configured by git clone to get changes from the HEAD branch of the origin repository. So often you can accomplish the above with just a simple

```
$ git pull
```

This command will fetch changes from the remote branches to your remote-tracking branches origin/*, and merge the default branch into the current branch.

More generally, a branch that is created from a remote-tracking branch will pull by default from that branch. See the descriptions of the branch.<name>.remote and branch.<name>.merge options in git-config(1), and the discussion of the --track option in git-checkout(1), to learn how to control these defaults.

In addition to saving you keystrokes, git pull also helps you by producing a default commit message documenting the branch and repository that you pulled from.

(But note that no such commit will be created in the case of a fast-forward; instead, your branch will just be updated to point to the latest commit from the upstream branch.)

The git pull command can also be given . as the "remote" repository, in which case it just merges in a branch from the current repository; so the commands

```
$ git pull . branch
$ git merge branch
```

are roughly equivalent.

4.2   Submitting patches to a project

If you just have a few changes, the simplest way to submit them may just be to send them as patches in email:

First, use git-format-patch(1); for example:

```
$ git format-patch origin
```

will produce a numbered series of files in the current directory, one for each patch in the current branch but not in `origin/HEAD`.

`git format-patch` can include an initial "cover letter". You can insert commentary on individual patches after the three dash line which `format-patch` places after the commit message but before the patch itself. If you use `git notes` to track your cover letter material, `git format-patch --notes` will include the commit's notes in a similar manner.

You can then import these into your mail client and send them by hand. However, if you have a lot to send at once, you may prefer to use the git-send-email(1) script to automate the process. Consult the mailing list for your project first to determine their requirements for submitting patches.

4.3   Importing patches to a project

Git also provides a tool called git-am(1) (am stands for "apply mailbox"), for importing such an emailed series of patches. Just save all of the patch-containing messages, in order, into a single mailbox file, say `patches.mbox`, then run

```
$ git am -3 patches.mbox
```

Git will apply each patch in order; if any conflicts are found, it will stop, and you can fix the conflicts as described in "Resolving a merge". (The `-3` option tells Git to perform a merge; if you would prefer it just to abort and leave your tree and index untouched, you may omit that option.)

Once the index is updated with the results of the conflict resolution, instead of creating a new commit, just run

```
$ git am --continue
```

and Git will create the commit for you and continue applying the remaining patches from the mailbox.

The final result will be a series of commits, one for each patch in the original mailbox, with authorship and commit log message each taken from the message containing each patch.

4.4   Public Git repositories

Another way to submit changes to a project is to tell the maintainer of that project to pull the changes from your repository using git-pull(1). In the section "Getting updates with git pull" we described this as a way to get updates from the "main" repository, but it works just as well in the other direction.

If you and the maintainer both have accounts on the same machine, then you can just pull changes from each other's repositories directly; commands that accept repository URLs as arguments will also accept a local directory name:

```
$ git clone /path/to/repository
$ git pull /path/to/other/repository
```

or an ssh URL:

```
$ git clone ssh://yourhost/~you/repository
```

For projects with few developers, or for synchronizing a few private repositories, this may be all you need.

However, the more common way to do this is to maintain a separate public repository (usually on a different host) for others to pull changes from. This is usually more convenient, and allows you to cleanly separate private work in progress from publicly visible work.

You will continue to do your day-to-day work in your personal repository, but periodically "push" changes from your personal repository into your public repository, allowing other developers to pull from that repository. So the flow of changes, in a situation where there is one other developer with a public repository, looks like this:

```
                      you push
your personal repo ------------------> your public repo
      ^                                     |
      |                                     |
      | you pull                            | they pull
      |                                     |
      |                                     |
      |              they push              V
their public repo <------------------ their repo
```

We explain how to do this in the following sections.

4.4.1   Setting up a public repository

Assume your personal repository is in the directory ~/proj. We first create a new clone of the repository and tell git daemon that it is meant to be public:

```
$ git clone --bare ~/proj proj.git
$ touch proj.git/git-daemon-export-ok
```

The resulting directory proj.git contains a "bare" git repository—it is just the contents of the .git directory, without any files checked out around it.

Next, copy proj.git to the server where you plan to host the public repository. You can use scp, rsync, or whatever is most convenient.

4.4.2   Exporting a Git repository via the Git protocol

This is the preferred method.

If someone else administers the server, they should tell you what directory to put the repository in, and what git:// URL it will appear at. You can then skip to the section "Pushing changes to a public repository", below.

Otherwise, all you need to do is start git-daemon(1); it will listen on port 9418. By default, it will allow access to any directory that looks like a Git directory and contains the magic file git-daemon-export-ok. Passing some directory paths as git daemon arguments will further restrict the exports to those paths.

You can also run git daemon as an inetd service; see the git-daemon(1) man page for details. (See especially the examples section.)

4.4.3   Exporting a git repository via HTTP

The Git protocol gives better performance and reliability, but on a host with a web server set up, HTTP exports may be simpler to set up.

All you need to do is place the newly created bare Git repository in a directory that is exported by the web server, and make some adjustments to give web clients some extra information they need:

```
$ mv proj.git /home/you/public_html/proj.git
$ cd proj.git
$ git --bare update-server-info
$ mv hooks/post-update.sample hooks/post-update
```

(For an explanation of the last two lines, see git-update-server-info(1) and githooks(5).)

Advertise the URL of proj.git. Anybody else should then be able to clone or pull from that URL, for example with a command line like:

```
$ git clone http://yourserver.com/~you/proj.git
```

(See also setup-git-server-over-http for a slightly more sophisticated setup using WebDAV which also allows pushing over HTTP.)

4.4.4   Pushing changes to a public repository

Note that the two techniques outlined above (exporting via http or git) allow other maintainers to fetch your latest changes, but they do not allow write access, which you will need to update the public repository with the latest changes created in your private repository.

The simplest way to do this is using git-push(1) and ssh; to update the remote branch named master with the latest state of your branch named master, run

```
$ git push ssh://yourserver.com/~you/proj.git master:master
```

or just

```
$ git push ssh://yourserver.com/~you/proj.git master
```

As with git fetch, git push will complain if this does not result in a fast-forward; see the following section for details on handling this case.

Note that the target of a push is normally a bare repository. You can also push to a repository that has a checked-out working tree, but a push to update the currently checked-out branch is denied by default to prevent confusion. See the description of the receive.denyCurrentBranch option in git-config(1) for details.

As with git fetch, you may also set up configuration options to save typing; so, for example:

```
$ git remote add public-repo ssh://yourserver.com/~you/proj.git
```

adds the following to .git/config:

```
[remote "public-repo"]
        url = yourserver.com:proj.git
        fetch = +refs/heads/*:refs/remotes/example/*
```

which lets you do the same push with just

```
$ git push public-repo master
```

See the explanations of the remote.<name>.url, branch.<name>.remote, and remote.<name>.push options in git-config(1) for details.

4.4.5 What to do when a push fails

If a push would not result in a fast-forward of the remote branch, then it will fail with an error like:

```
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to '...'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

This can happen, for example, if you:

- use `git reset --hard` to remove already-published commits, or

- use `git commit --amend` to replace already-published commits (as in Section 3.10.2), or

- use `git rebase` to rebase any already-published commits (as in Section 5.2).

You may force `git push` to perform the update anyway by preceding the branch name with a plus sign:

```
$ git push ssh://yourserver.com/~you/proj.git +master
```

Note the addition of the + sign. Alternatively, you can use the -f flag to force the remote update, as in:

```
$ git push -f ssh://yourserver.com/~you/proj.git master
```

Normally whenever a branch head in a public repository is modified, it is modified to point to a descendant of the commit that it pointed to before. By forcing a push in this situation, you break that convention. (See Section 5.7.)

Nevertheless, this is a common practice for people that need a simple way to publish a work-in-progress patch series, and it is an acceptable compromise as long as you warn other developers that this is how you intend to manage the branch.

It's also possible for a push to fail in this way when other people have the right to push to the same repository. In that case, the correct solution is to retry the push after first updating your work: either by a pull, or by a fetch followed by a rebase; see the next section and gitcvs-migration(7) for more.

4.4.6 Setting up a shared repository

Another way to collaborate is by using a model similar to that commonly used in CVS, where several developers with special rights all push to and pull from a single shared repository. See gitcvs-migration(7) for instructions on how to set this up.

However, while there is nothing wrong with Git's support for shared repositories, this mode of operation is not generally recommended, simply because the mode of collaboration that Git supports—by exchanging patches and pulling from public repositories—has so many advantages over the central shared repository:

- Git's ability to quickly import and merge patches allows a single maintainer to process incoming changes even at very high rates. And when that becomes too much, `git pull` provides an easy way for that maintainer to delegate this job to other maintainers while still allowing optional review of incoming changes.

- Since every developer's repository has the same complete copy of the project history, no repository is special, and it is trivial for another developer to take over maintenance of a project, either by mutual agreement, or because a maintainer becomes unresponsive or difficult to work with.

- The lack of a central group of "committers" means there is less need for formal decisions about who is "in" and who is "out".

4.4.7   Allowing web browsing of a repository

The gitweb cgi script provides users an easy way to browse your project's revisions, file contents and logs without having to install Git. Features like RSS/Atom feeds and blame/annotation details may optionally be enabled.

The git-instaweb(1) command provides a simple way to start browsing the repository using gitweb. The default server when using instaweb is lighttpd.

See the file gitweb/INSTALL in the Git source tree and gitweb(1) for instructions on details setting up a permanent installation with a CGI or Perl capable server.

4.5   How to get a Git repository with minimal history

A shallow clone, with its truncated history, is useful when one is interested only in recent history of a project and getting full history from the upstream is expensive.

A shallow clone is created by specifying the git-clone(1) `--depth` switch. The depth can later be changed with the git-fetch(1) `--depth` switch, or full history restored with `--unshallow`.

Merging inside a shallow clone will work as long as a merge base is in the recent history. Otherwise, it will be like merging unrelated histories and may have to result in huge conflicts. This limitation may make such a repository unsuitable to be used in merge based workflows.

4.6   Examples

4.6.1   Maintaining topic branches for a Linux subsystem maintainer

This describes how Tony Luck uses Git in his role as maintainer of the IA64 architecture for the Linux kernel.

He uses two public branches:

- A "test" tree into which patches are initially placed so that they can get some exposure when integrated with other ongoing development. This tree is available to Andrew for pulling into -mm whenever he wants.

- A "release" tree into which tested patches are moved for final sanity checking, and as a vehicle to send them upstream to Linus (by sending him a "please pull" request.)

He also uses a set of temporary branches ("topic branches"), each containing a logical grouping of patches.

To set this up, first create your work tree by cloning Linus's public tree:

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git work
$ cd work
```

Linus's tree will be stored in the remote-tracking branch named origin/master, and can be updated using git-fetch(1); you can track other public trees using git-remote(1) to set up a "remote" and git-fetch(1) to keep them up to date; see Chapter 1.

Now create the branches in which you are going to work; these start out at the current tip of origin/master branch, and should be set up (using the `--track` option to git-branch(1)) to merge changes in from Linus by default.

```
$ git branch --track test origin/master
$ git branch --track release origin/master
```

These can be easily kept up to date using git-pull(1).

```
$ git checkout test && git pull
$ git checkout release && git pull
```

Important note! If you have any local changes in these branches, then this merge will create a commit object in the history (with no local changes Git will simply do a "fast-forward" merge). Many people dislike the "noise" that this creates in the Linux history, so you should avoid doing this capriciously in the release branch, as these noisy commits will become part of the permanent history when you ask Linus to pull from the release branch.

A few configuration variables (see git-config(1)) can make it easy to push both branches to your public tree. (See Section 4.4.1.)

```
$ cat >> .git/config <<EOF
[remote "mytree"]
        url =  master.kernel.org:/pub/scm/linux/kernel/git/aegl/linux.git
        push = release
        push = test
EOF
```

Then you can push both the test and release trees using git-push(1):

```
$ git push mytree
```

or push just one of the test and release branches using:

```
$ git push mytree test
```

or

```
$ git push mytree release
```

Now to apply some patches from the community. Think of a short snappy name for a branch to hold this patch (or related group of patches), and create a new branch from a recent stable tag of Linus's branch. Picking a stable base for your branch will: 1) help you: by avoiding inclusion of unrelated and perhaps lightly tested changes 2) help future bug hunters that use git bisect to find problems

```
$ git checkout -b speed-up-spinlocks v2.6.35
```

Now you apply the patch(es), run some tests, and commit the change(s). If the patch is a multi-part series, then you should apply each as a separate commit to this branch.

```
$ ... patch ... test  ... commit [ ... patch ... test ... commit ]*
```

When you are happy with the state of this change, you can merge it into the "test" branch in preparation to make it public:

```
$ git checkout test && git merge speed-up-spinlocks
```

It is unlikely that you would have any conflicts here ... but you might if you spent a while on this step and had also pulled new versions from upstream.

Sometime later when enough time has passed and testing done, you can pull the same branch into the release tree ready to go upstream. This is where you see the value of keeping each patch (or patch series) in its own branch. It means that the patches can be moved into the release tree in any order.

```
$ git checkout release && git merge speed-up-spinlocks
```

After a while, you will have a number of branches, and despite the well chosen names you picked for each of them, you may forget what they are for, or what status they are in. To get a reminder of what changes are in a specific branch, use:

```
$ git log linux..branchname | git shortlog
```

To see whether it has already been merged into the test or release branches, use:

```
$ git log test..branchname
```

or

```
$ git log release..branchname
```

(If this branch has not yet been merged, you will see some log entries. If it has been merged, then there will be no output.)

Once a patch completes the great cycle (moving from test to release, then pulled by Linus, and finally coming back into your local `origin/master` branch), the branch for this change is no longer needed. You detect this when the output from:

```
$ git log origin..branchname
```

is empty. At this point the branch can be deleted:

```
$ git branch -d branchname
```

Some changes are so trivial that it is not necessary to create a separate branch and then merge into each of the test and release branches. For these changes, just apply directly to the `release` branch, and then merge that into the `test` branch.

After pushing your work to `mytree`, you can use git-request-pull(1) to prepare a "please pull" request message to send to Linus:

```
$ git push mytree
$ git request-pull origin mytree release
```

Here are some of the scripts that simplify all this even further.

```
==== update script ====
# Update a branch in my Git tree.  If the branch to be updated
# is origin, then pull from kernel.org.  Otherwise merge
# origin/master branch into test|release branch

case "$1" in
test|release)
        git checkout $1 && git pull . origin
        ;;
origin)
        before=$(git rev-parse refs/remotes/origin/master)
        git fetch origin
        after=$(git rev-parse refs/remotes/origin/master)
        if [ $before != $after ]
        then
                git log $before..$after | git shortlog
        fi
        ;;
*)
        echo "usage: $0 origin|test|release" 1>&2
        exit 1
        ;;
esac


==== merge script ====
# Merge a branch into either the test or release branch

pname=$0

usage()
{
        echo "usage: $pname branch test|release" 1>&2
        exit 1
}
```

```
git show-ref -q --verify -- refs/heads/"$1" || {
        echo "Can't see branch <$1>" 1>&2
        usage
}

case "$2" in
test|release)
        if [ $(git log $2..$1 | wc -c) -eq 0 ]
        then
                echo $1 already merged into $2 1>&2
                exit 1
        fi
        git checkout $2 && git pull . $1
        ;;
*)
        usage
        ;;
esac

==== status script ====
# report on status of my ia64 Git tree

gb=$(tput setab 2)
rb=$(tput setab 1)
restore=$(tput setab 9)

if [ `git rev-list test..release | wc -c` -gt 0 ]
then
        echo $rb Warning: commits in release that are not in test $restore
        git log test..release
fi

for branch in `git show-ref --heads | sed 's|^.*/||'`
do
        if [ $branch = test -o $branch = release ]
        then
                continue
        fi

        echo -n $gb ======= $branch ====== $restore " "
        status=
        for ref in test release origin/master
        do
                if [ `git rev-list $ref..$branch | wc -c` -gt 0 ]
                then
                        status=$status${ref:0:1}
                fi
        done
        case $status in
        trl)
                echo $rb Need to pull into test $restore
                ;;
        rl)
                echo "In test"
                ;;
        l)
```

```
                echo "Waiting for linus"
                ;;
        "")
                echo $rb All done $restore
                ;;
        *)
                echo $rb "<$status>" $restore
                ;;
        esac
        git log origin/master..$branch | git shortlog
done
```

Chapter 5

Rewriting history and maintaining patch series

Normally commits are only added to a project, never taken away or replaced. Git is designed with this assumption, and violating it will cause Git's merge machinery (for example) to do the wrong thing.

However, there is a situation in which it can be useful to violate this assumption.

5.1   Creating the perfect patch series

Suppose you are a contributor to a large project, and you want to add a complicated feature, and to present it to the other developers in a way that makes it easy for them to read your changes, verify that they are correct, and understand why you made each change.

If you present all of your changes as a single patch (or commit), they may find that it is too much to digest all at once.

If you present them with the entire history of your work, complete with mistakes, corrections, and dead ends, they may be overwhelmed.

So the ideal is usually to produce a series of patches such that:

1. Each patch can be applied in order.

2. Each patch includes a single logical change, together with a message explaining the change.

3. No patch introduces a regression: after applying any initial part of the series, the resulting project still compiles and works, and has no bugs that it didn't have before.

4. The complete series produces the same end result as your own (probably much messier!) development process did.

We will introduce some tools that can help you do this, explain how to use them, and then explain some of the problems that can arise because you are rewriting history.

5.2   Keeping a patch series up to date using git rebase

Suppose that you create a branch `mywork` on a remote-tracking branch `origin`, and create some commits on top of it:

```
$ git checkout -b mywork origin
$ vi file.txt
$ git commit
$ vi otherfile.txt
$ git commit
...
```

You have performed no merges into mywork, so it is just a simple linear sequence of patches on top of `origin`:

```
 o--o--O <-- origin
         \
          a--b--c <-- mywork
```

Some more interesting work has been done in the upstream project, and `origin` has advanced:

```
 o--o--O--o--o--o <-- origin
         \
          a--b--c <-- mywork
```

At this point, you could use `pull` to merge your changes back in; the result would create a new merge commit, like this:

```
 o--o--O--o--o--o <-- origin
         \         \
          a--b--c--m <-- mywork
```

However, if you prefer to keep the history in mywork a simple series of commits without any merges, you may instead choose to use git-rebase(1):

```
$ git checkout mywork
$ git rebase origin
```

This will remove each of your commits from mywork, temporarily saving them as patches (in a directory named `.git/rebase-apply` update mywork to point at the latest version of origin, then apply each of the saved patches to the new mywork. The result will look like:

```
 o--o--O--o--o--o <-- origin
                  \
                   a'--b'--c' <-- mywork
```

In the process, it may discover conflicts. In that case it will stop and allow you to fix the conflicts; after fixing conflicts, use `git add` to update the index with those contents, and then, instead of running `git commit`, just run

```
$ git rebase --continue
```

and Git will continue applying the rest of the patches.

At any point you may use the `--abort` option to abort this process and return mywork to the state it had before you started the rebase:

```
$ git rebase --abort
```

If you need to reorder or edit a number of commits in a branch, it may be easier to use `git rebase -i`, which allows you to reorder and squash commits, as well as marking them for individual editing during the rebase. See Section 5.5 for details, and Section 5.4 for alternatives.

5.3   Rewriting a single commit

We saw in Section 3.10.2 that you can replace the most recent commit using

```
$ git commit --amend
```

which will replace the old commit by a new commit incorporating your changes, giving you a chance to edit the old commit message first. This is useful for fixing typos in your last commit, or for adjusting the patch contents of a poorly staged commit.

If you need to amend commits from deeper in your history, you can use interactive rebase's edit instruction.

5.4 Reordering or selecting from a patch series

Sometimes you want to edit a commit deeper in your history. One approach is to use `git format-patch` to create a series of patches and then reset the state to before the patches:

```
$ git format-patch origin
$ git reset --hard origin
```

Then modify, reorder, or eliminate patches as needed before applying them again with git-am(1):

```
$ git am *.patch
```

5.5 Using interactive rebases

You can also edit a patch series with an interactive rebase. This is the same as reordering a patch series using format-patch, so use whichever interface you like best.

Rebase your current HEAD on the last commit you want to retain as-is. For example, if you want to reorder the last 5 commits, use:

```
$ git rebase -i HEAD~5
```

This will open your editor with a list of steps to be taken to perform your rebase.

```
pick deadbee The oneline of this commit
pick fa1afe1 The oneline of the next commit
...

# Rebase c0ffeee..deadbee onto c0ffeee
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

As explained in the comments, you can reorder commits, squash them together, edit commit messages, etc. by editing the list. Once you are satisfied, save the list and close your editor, and the rebase will begin.

The rebase will stop where `pick` has been replaced with `edit` or when a step in the list fails to mechanically resolve conflicts and needs your help. When you are done editing and/or resolving conflicts you can continue with `git rebase --continue`. If you decide that things are getting too hairy, you can always bail out with `git rebase --abort`. Even after the rebase is complete, you can still recover the original branch by using the reflog.

For a more detailed discussion of the procedure and additional tips, see the "INTERACTIVE MODE" section of git-rebase(1).

5.6 Other tools

There are numerous other tools, such as StGit, which exist for the purpose of maintaining a patch series. These are outside of the scope of this manual.

5.7 Problems with rewriting history

The primary problem with rewriting the history of a branch has to do with merging. Suppose somebody fetches your branch and merges it into their branch, with a result something like this:

```
 o--o--O--o--o--o <-- origin
       \         \
        t--t--t--m <-- their branch:
```

Then suppose you modify the last three commits:

```
         o--o--o <-- new head of origin
        /
 o--o--O--o--o--o <-- old head of origin
```

If we examined all this history together in one repository, it will look like:

```
         o--o--o <-- new head of origin
        /
 o--o--O--o--o--o <-- old head of origin
       \         \
        t--t--t--m <-- their branch:
```

Git has no way of knowing that the new head is an updated version of the old head; it treats this situation exactly the same as it would if two developers had independently done the work on the old and new heads in parallel. At this point, if someone attempts to merge the new head in to their branch, Git will attempt to merge together the two (old and new) lines of development, instead of trying to replace the old by the new. The results are likely to be unexpected.

You may still choose to publish branches whose history is rewritten, and it may be useful for others to be able to fetch those branches in order to examine or test them, but they should not attempt to pull such branches into their own work.

For true distributed development that supports proper merging, published branches should never be rewritten.

5.8 Why bisecting merge commits can be harder than bisecting linear history

The git-bisect(1) command correctly handles history that includes merge commits. However, when the commit that it finds is a merge commit, the user may need to work harder than usual to figure out why that commit introduced a problem.

Imagine this history:

```
      ---Z---o---X---...---o---A---C---D
          \                     /
           o---o---Y---...---o---B
```

Suppose that on the upper line of development, the meaning of one of the functions that exists at Z is changed at commit X. The commits from Z leading to A change both the function's implementation and all calling sites that exist at Z, as well as new calling sites they add, to be consistent. There is no bug at A.

Suppose that in the meantime on the lower line of development somebody adds a new calling site for that function at commit Y. The commits from Z leading to B all assume the old semantics of that function and the callers and the callee are consistent with each other. There is no bug at B, either.

Suppose further that the two development lines merge cleanly at C, so no conflict resolution is required.

Nevertheless, the code at C is broken, because the callers added on the lower line of development have not been converted to the new semantics introduced on the upper line of development. So if all you know is that D is bad, that Z is good, and that git-bisect(1) identifies C as the culprit, how will you figure out that the problem is due to this change in semantics?

When the result of a `git bisect` is a non-merge commit, you should normally be able to discover the problem by examining just that commit. Developers can make this easy by breaking their changes into small self-contained commits. That won't help in the case above, however, because the problem isn't obvious from examination of any single commit; instead, a global view of the development is required. To make matters worse, the change in semantics in the problematic function may be just one small part of the changes in the upper line of development.

On the other hand, if instead of merging at C you had rebased the history between Z to B on top of A, you would have gotten this linear history:

```
---Z---o---X--...---o---A---o---o---Y*--...---o---B*--D*
```

Bisecting between Z and D* would hit a single culprit commit Y*, and understanding why Y* was broken would probably be easier.

Partly for this reason, many experienced Git users, even when working on an otherwise merge-heavy project, keep the history linear by rebasing against the latest upstream version before publishing.

Chapter 6

Advanced branch management

6.1 Fetching individual branches

Instead of using git-remote(1), you can also choose just to update one branch at a time, and to store it locally under an arbitrary name:

```
$ git fetch origin todo:my-todo-work
```

The first argument, `origin`, just tells Git to fetch from the repository you originally cloned from. The second argument tells Git to fetch the branch named `todo` from the remote repository, and to store it locally under the name `refs/heads/my-todo-work`.

You can also fetch branches from other repositories; so

```
$ git fetch git://example.com/proj.git master:example-master
```

will create a new branch named `example-master` and store in it the branch named `master` from the repository at the given URL. If you already have a branch named example-master, it will attempt to fast-forward to the commit given by example.com's master branch. In more detail:

6.2 git fetch and fast-forwards

In the previous example, when updating an existing branch, `git fetch` checks to make sure that the most recent commit on the remote branch is a descendant of the most recent commit on your copy of the branch before updating your copy of the branch to point at the new commit. Git calls this process a fast-forward.

A fast-forward looks something like this:

```
 o--o--o--o <-- old head of the branch
          \
           o--o--o <-- new head of the branch
```

In some cases it is possible that the new head will **not** actually be a descendant of the old head. For example, the developer may have realized she made a serious mistake, and decided to backtrack, resulting in a situation like:

```
 o--o--o--o--a--b <-- old head of the branch
          \
           o--o--o <-- new head of the branch
```

In this case, `git fetch` will fail, and print out a warning.

In that case, you can still force Git to update to the new head, as described in the following section. However, note that in the situation above this may mean losing the commits labeled `a` and `b`, unless you've already created a reference of your own pointing to them.

6.3   Forcing git fetch to do non-fast-forward updates

If git fetch fails because the new head of a branch is not a descendant of the old head, you may force the update with:

```
$ git fetch git://example.com/proj.git +master:refs/remotes/example/master
```

Note the addition of the + sign. Alternatively, you can use the -f flag to force updates of all the fetched branches, as in:

```
$ git fetch -f origin
```

Be aware that commits that the old version of example/master pointed at may be lost, as we saw in the previous section.

6.4   Configuring remote-tracking branches

We saw above that origin is just a shortcut to refer to the repository that you originally cloned from. This information is stored in Git configuration variables, which you can see using git-config(1):

```
$ git config -l
core.repositoryformatversion=0
core.filemode=true
core.logallrefupdates=true
remote.origin.url=git://git.kernel.org/pub/scm/git/git.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
```

If there are other repositories that you also use frequently, you can create similar configuration options to save typing; for example,

```
$ git remote add example git://example.com/proj.git
```

adds the following to .git/config:

```
[remote "example"]
        url = git://example.com/proj.git
        fetch = +refs/heads/*:refs/remotes/example/*
```

Also note that the above configuration can be performed by directly editing the file .git/config instead of using git-remote(1).

After configuring the remote, the following three commands will do the same thing:

```
$ git fetch git://example.com/proj.git +refs/heads/*:refs/remotes/example/*
$ git fetch example +refs/heads/*:refs/remotes/example/*
$ git fetch example
```

See git-config(1) for more details on the configuration options mentioned above and git-fetch(1) for more details on the refspec syntax.

Chapter 7

Git concepts

Git is built on a small number of simple but powerful ideas. While it is possible to get things done without understanding them, you will find Git much more intuitive if you do.

We start with the most important, the object database and the index.

7.1   The Object Database

We already saw in Section 1.3 that all commits are stored under a 40-digit "object name". In fact, all the information needed to represent the history of a project is stored in objects with such names. In each case the name is calculated by taking the SHA-1 hash of the contents of the object. The SHA-1 hash is a cryptographic hash function. What that means to us is that it is impossible to find two different objects with the same name. This has a number of advantages; among others:

- Git can quickly determine whether two objects are identical or not, just by comparing names.

- Since object names are computed the same way in every repository, the same content stored in two repositories will always be stored under the same name.

- Git can detect errors when it reads an object, by checking that the object's name is still the SHA-1 hash of its contents.

(See Section 10.1 for the details of the object formatting and SHA-1 calculation.)

There are four different types of objects: "blob", "tree", "commit", and "tag".

- A "blob" object is used to store file data.

- A "tree" object ties one or more "blob" objects into a directory structure. In addition, a tree object can refer to other tree objects, thus creating a directory hierarchy.

- A "commit" object ties such directory hierarchies together into a directed acyclic graph of revisions—each commit contains the object name of exactly one tree designating the directory hierarchy at the time of the commit. In addition, a commit refers to "parent" commit objects that describe the history of how we arrived at that directory hierarchy.

- A "tag" object symbolically identifies and can be used to sign other objects. It contains the object name and type of another object, a symbolic name (of course!) and, optionally, a signature.

The object types in some more detail:

7.1.1 Commit Object

The "commit" object links a physical state of a tree with a description of how we got there and why. Use the `--pretty=raw` option to git-show(1) or git-log(1) to examine your favorite commit:

```
$ git show -s --pretty=raw 2be7fcb476
commit 2be7fcb4764f2dbcee52635b91fedb1b3dcf7ab4
tree fb3a8bdd0ceddd019615af4d57a53f43d8cee2bf
parent 257a84d9d02e90447b149af58b271c19405edb6a
author Dave Watson <dwatson@mimvista.com> 1187576872 -0400
committer Junio C Hamano <gitster@pobox.com> 1187591163 -0700

    Fix misspelling of 'suppress' in docs

    Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

As you can see, a commit is defined by:

- a tree: The SHA-1 name of a tree object (as defined below), representing the contents of a directory at a certain point in time.

- parent(s): The SHA-1 name(s) of some number of commits which represent the immediately previous step(s) in the history of the project. The example above has one parent; merge commits may have more than one. A commit with no parents is called a "root" commit, and represents the initial revision of a project. Each project must have at least one root. A project can also have multiple roots, though that isn't common (or necessarily a good idea).

- an author: The name of the person responsible for this change, together with its date.

- a committer: The name of the person who actually created the commit, with the date it was done. This may be different from the author, for example, if the author was someone who wrote a patch and emailed it to the person who used it to create the commit.

- a comment describing this commit.

Note that a commit does not itself contain any information about what actually changed; all changes are calculated by comparing the contents of the tree referred to by this commit with the trees associated with its parents. In particular, Git does not attempt to record file renames explicitly, though it can identify cases where the existence of the same file data at changing paths suggests a rename. (See, for example, the `-M` option to git-diff(1)).

A commit is usually created by git-commit(1), which creates a commit whose parent is normally the current HEAD, and whose tree is taken from the content currently stored in the index.

7.1.2 Tree Object

The ever-versatile git-show(1) command can also be used to examine tree objects, but git-ls-tree(1) will give you more details:

```
$ git ls-tree fb3a8bdd0ce
100644 blob 63c918c667fa005ff12ad89437f2fdc80926e21c    .gitignore
100644 blob 5529b198e8d14decbe4ad99db3f7fb632de0439d    .mailmap
100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3    COPYING
040000 tree 2fb783e477100ce076f6bf57e4a6f026013dc745    Documentation
100755 blob 3c0032cec592a765692234f1cba47dfdcc3a9200    GIT-VERSION-GEN
100644 blob 289b046a443c0647624607d471289b2c7dcd470b    INSTALL
100644 blob 4eb463797adc693dc168b926b6932ff53f17d0b1    Makefile
100644 blob 548142c327a6790ff8821d67c2ee1eff7a656b52    README
...
```

As you can see, a tree object contains a list of entries, each with a mode, object type, SHA-1 name, and name, sorted by name. It represents the contents of a single directory tree.

The object type may be a blob, representing the contents of a file, or another tree, representing the contents of a subdirectory. Since trees and blobs, like all other objects, are named by the SHA-1 hash of their contents, two trees have the same SHA-1 name if and only if their contents (including, recursively, the contents of all subdirectories) are identical. This allows Git to quickly determine the differences between two related tree objects, since it can ignore any entries with identical object names.

(Note: in the presence of submodules, trees may also have commits as entries. See Chapter 8 for documentation.)

Note that the files all have mode 644 or 755: Git actually only pays attention to the executable bit.

7.1.3 Blob Object

You can use git-show(1) to examine the contents of a blob; take, for example, the blob in the entry for COPYING from the tree above:

```
$ git show 6ff87c4664

 Note that the only valid version of the GPL as far as this project
 is concerned is _this_ particular version of the license (ie v2, not
 v2.2 or v3.x or whatever), unless explicitly otherwise stated.
...
```

A "blob" object is nothing but a binary blob of data. It doesn't refer to anything else or have attributes of any kind.

Since the blob is entirely defined by its data, if two files in a directory tree (or in multiple different versions of the repository) have the same contents, they will share the same blob object. The object is totally independent of its location in the directory tree, and renaming a file does not change the object that file is associated with.

Note that any tree or blob object can be examined using git-show(1) with the <revision>:<path> syntax. This can sometimes be useful for browsing the contents of a tree that is not currently checked out.

7.1.4 Trust

If you receive the SHA-1 name of a blob from one source, and its contents from another (possibly untrusted) source, you can still trust that those contents are correct as long as the SHA-1 name agrees. This is because the SHA-1 is designed so that it is infeasible to find different contents that produce the same hash.

Similarly, you need only trust the SHA-1 name of a top-level tree object to trust the contents of the entire directory that it refers to, and if you receive the SHA-1 name of a commit from a trusted source, then you can easily verify the entire history of commits reachable through parents of that commit, and all of those contents of the trees referred to by those commits.

So to introduce some real trust in the system, the only thing you need to do is to digitally sign just *one* special note, which includes the name of a top-level commit. Your digital signature shows others that you trust that commit, and the immutability of the history of commits tells others that they can trust the whole history.

In other words, you can easily validate a whole archive by just sending out a single email that tells the people the name (SHA-1 hash) of the top commit, and digitally sign that email using something like GPG/PGP.

To assist in this, Git also provides the tag object. . .

7.1.5 Tag Object

A tag object contains an object, object type, tag name, the name of the person ("tagger") who created the tag, and a message, which may contain a signature, as can be seen using git-cat-file(1):

```
$ git cat-file tag v1.5.0
object 437b1b20df4b356c9342dac8d38849f24ef44f27
type commit
tag v1.5.0
tagger Junio C Hamano <junkio@cox.net> 1171411200 +0000

GIT 1.5.0
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.6 (GNU/Linux)

iD8DBQBF0lGqwMbZpPMRm5oRAuRiAJ9ohBLd7s2kqjkKlq1qqC57SbnmzQCdG4ui
nLE/L9aUXdWeTFPron96DLA=
=2E+0
-----END PGP SIGNATURE-----
```

See the git-tag(1) command to learn how to create and verify tag objects. (Note that git-tag(1) can also be used to create "lightweight tags", which are not tag objects at all, but just simple references whose names begin with refs/tags/).

7.1.6 How Git stores objects efficiently: pack files

Newly created objects are initially created in a file named after the object's SHA-1 hash (stored in .git/objects).

Unfortunately this system becomes inefficient once a project has a lot of objects. Try this on an old project:

```
$ git count-objects
6930 objects, 47620 kilobytes
```

The first number is the number of objects which are kept in individual files. The second is the amount of space taken up by those "loose" objects.

You can save space and make Git faster by moving these loose objects in to a "pack file", which stores a group of objects in an efficient compressed format; the details of how pack files are formatted can be found in pack format.

To put the loose objects into a pack, just run git repack:

```
$ git repack
Counting objects: 6020, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6020/6020), done.
Writing objects: 100% (6020/6020), done.
Total 6020 (delta 4070), reused 0 (delta 0)
```

This creates a single "pack file" in .git/objects/pack/ containing all currently unpacked objects. You can then run

```
$ git prune
```

to remove any of the "loose" objects that are now contained in the pack. This will also remove any unreferenced objects (which may be created when, for example, you use git reset to remove a commit). You can verify that the loose objects are gone by looking at the .git/objects directory or by running

```
$ git count-objects
0 objects, 0 kilobytes
```

Although the object files are gone, any commands that refer to those objects will work exactly as they did before.

The git-gc(1) command performs packing, pruning, and more for you, so is normally the only high-level command you need.

7.1.7   Dangling objects

The git-fsck(1) command will sometimes complain about dangling objects. They are not a problem.

The most common cause of dangling objects is that you've rebased a branch, or you have pulled from somebody else who rebased a branch—see Chapter 5. In that case, the old head of the original branch still exists, as does everything it pointed to. The branch pointer itself just doesn't, since you replaced it with another one.

There are also other situations that cause dangling objects. For example, a "dangling blob" may arise because you did a `git add` of a file, but then, before you actually committed it and made it part of the bigger picture, you changed something else in that file and committed that **updated** thing—the old state that you added originally ends up not being pointed to by any commit or tree, so it's now a dangling blob object.

Similarly, when the "recursive" merge strategy runs, and finds that there are criss-cross merges and thus more than one merge base (which is fairly unusual, but it does happen), it will generate one temporary midway tree (or possibly even more, if you had lots of criss-crossing merges and more than two merge bases) as a temporary internal merge base, and again, those are real objects, but the end result will not end up pointing to them, so they end up "dangling" in your repository.

Generally, dangling objects aren't anything to worry about. They can even be very useful: if you screw something up, the dangling objects can be how you recover your old tree (say, you did a rebase, and realized that you really didn't want to—you can look at what dangling objects you have, and decide to reset your head to some old dangling state).

For commits, you can just use:

```
$ gitk <dangling-commit-sha-goes-here> --not --all
```

This asks for all the history reachable from the given commit but not from any branch, tag, or other reference. If you decide it's something you want, you can always create a new reference to it, e.g.,

```
$ git branch recovered-branch <dangling-commit-sha-goes-here>
```

For blobs and trees, you can't do the same, but you can still examine them. You can just do

```
$ git show <dangling-blob/tree-sha-goes-here>
```

to show what the contents of the blob were (or, for a tree, basically what the `ls` for that directory was), and that may give you some idea of what the operation was that left that dangling object.

Usually, dangling blobs and trees aren't very interesting. They're almost always the result of either being a half-way mergebase (the blob will often even have the conflict markers from a merge in it, if you have had conflicting merges that you fixed up by hand), or simply because you interrupted a `git fetch` with ^C or something like that, leaving *some* of the new objects in the object database, but just dangling and useless.

Anyway, once you are sure that you're not interested in any dangling state, you can just prune all unreachable objects:

```
$ git prune
```

and they'll be gone. (You should only run `git prune` on a quiescent repository—it's kind of like doing a filesystem fsck recovery: you don't want to do that while the filesystem is mounted. `git prune` is designed not to cause any harm in such cases of concurrent accesses to a repository but you might receive confusing or scary messages.)

7.1.8   Recovering from repository corruption

By design, Git treats data trusted to it with caution. However, even in the absence of bugs in Git itself, it is still possible that hardware or operating system errors could corrupt data.

The first defense against such problems is backups. You can back up a Git directory using clone, or just using cp, tar, or any other backup mechanism.

As a last resort, you can search for the corrupted objects and attempt to replace them by hand. Back up your repository before attempting this in case you corrupt things even more in the process.

We'll assume that the problem is a single missing or corrupted blob, which is sometimes a solvable problem. (Recovering missing trees and especially commits is **much** harder).

Before starting, verify that there is corruption, and figure out where it is with git-fsck(1); this may be time-consuming.

Assume the output looks like this:

```
$ git fsck --full --no-dangling
broken link from    tree 2d9263c6d23595e7cb2a21e5ebbb53655278dff8
             to     blob 4b9458b3786228369c63936db65827de3cc06200
missing blob 4b9458b3786228369c63936db65827de3cc06200
```

Now you know that blob 4b9458b3 is missing, and that the tree 2d9263c6 points to it. If you could find just one copy of that missing blob object, possibly in some other repository, you could move it into `.git/objects/4b/9458b3...` and be done. Suppose you can't. You can still examine the tree that pointed to it with git-ls-tree(1), which might output something like:

```
$ git ls-tree 2d9263c6d23595e7cb2a21e5ebbb53655278dff8
100644 blob 8d14531846b95bfa3564b58ccfb7913a034323b8    .gitignore
100644 blob ebf9bf84da0aab5ed944264a5db2a65fe3a3e883    .mailmap
100644 blob ca442d313d86dc67e0a2e5d584b465bd382cbf5c    COPYING
...
100644 blob 4b9458b3786228369c63936db65827de3cc06200    myfile
...
```

So now you know that the missing blob was the data for a file named `myfile`. And chances are you can also identify the directory—let's say it's in `somedirectory`. If you're lucky the missing copy might be the same as the copy you have checked out in your working tree at `somedirectory/myfile`; you can test whether that's right with git-hash-object(1):

```
$ git hash-object -w somedirectory/myfile
```

which will create and store a blob object with the contents of somedirectory/myfile, and output the SHA-1 of that object. if you're extremely lucky it might be 4b9458b3786228369c63936db65827de3cc06200, in which case you've guessed right, and the corruption is fixed!

Otherwise, you need more information. How do you tell which version of the file has been lost?

The easiest way to do this is with:

```
$ git log --raw --all --full-history -- somedirectory/myfile
```

Because you're asking for raw output, you'll now get something like

```
commit abc
Author:
Date:
...
:100644 100644 4b9458b newsha M somedirectory/myfile


commit xyz
Author:
Date:

...
:100644 100644 oldsha 4b9458b M somedirectory/myfile
```

This tells you that the immediately following version of the file was "newsha", and that the immediately preceding version was "oldsha". You also know the commit messages that went with the change from oldsha to 4b9458b and with the change from 4b9458b to newsha.

If you've been committing small enough changes, you may now have a good shot at reconstructing the contents of the in-between state 4b9458b.

If you can do that, you can now recreate the missing object with

```
$ git hash-object -w <recreated-file>
```

and your repository is good again!

(Btw, you could have ignored the `fsck`, and started with doing a

```
$ git log --raw --all
```

and just looked for the sha of the missing object (4b9458b) in that whole thing. It's up to you—Git does **have** a lot of information, it is just missing one particular blob version.

7.2   The index

The index is a binary file (generally kept in `.git/index`) containing a sorted list of path names, each with permissions and the SHA-1 of a blob object; git-ls-files(1) can show you the contents of the index:

```
$ git ls-files --stage
100644 63c918c667fa005ff12ad89437f2fdc80926e21c 0       .gitignore
100644 5529b198e8d14decbe4ad99db3f7fb632de0439d 0       .mailmap
100644 6ff87c4664981e4397625791c8ea3bbb5f2279a3 0       COPYING
100644 a37b2152bd26be2c2289e1f57a292534a51a93c7 0       Documentation/.gitignore
100644 fbefe9a45b00a54b58d94d06eca48b03d40a50e0 0       Documentation/Makefile
...
100644 2511aef8d89ab52be5ec6a5e46236b4b6bcd07ea 0       xdiff/xtypes.h
100644 2ade97b2574a9f77e7ae4002a4e07a6a38e46d07 0       xdiff/xutils.c
100644 d5de8292e05e7c36c4b68857c1cf9855e3d2f70a 0       xdiff/xutils.h
```

Note that in older documentation you may see the index called the "current directory cache" or just the "cache". It has three important properties:

1. The index contains all the information necessary to generate a single (uniquely determined) tree object.

   For example, running git-commit(1) generates this tree object from the index, stores it in the object database, and uses it as the tree object associated with the new commit.

2. The index enables fast comparisons between the tree object it defines and the working tree.

   It does this by storing some additional data for each entry (such as the last modified time). This data is not displayed above, and is not stored in the created tree object, but it can be used to determine quickly which files in the working directory differ from what was stored in the index, and thus save Git from having to read all of the data from such files to look for changes.

3. It can efficiently represent information about merge conflicts between different tree objects, allowing each pathname to be associated with sufficient information about the trees involved that you can create a three-way merge between them.

   We saw in Section 3.7.1 that during a merge the index can store multiple versions of a single file (called "stages"). The third column in the git-ls-files(1) output above is the stage number, and will take on values other than 0 for files with merge conflicts.

The index is thus a sort of temporary staging area, which is filled with a tree which you are in the process of working on.

If you blow the index away entirely, you generally haven't lost any information as long as you have the name of the tree that it described.

Chapter 8

Submodules

Large projects are often composed of smaller, self-contained modules. For example, an embedded Linux distribution's source tree would include every piece of software in the distribution with some local modifications; a movie player might need to build against a specific, known-working version of a decompression library; several independent programs might all share the same build scripts.

With centralized revision control systems this is often accomplished by including every module in one single repository. Developers can check out all modules or only the modules they need to work with. They can even modify files across several modules in a single commit while moving things around or updating APIs and translations.

Git does not allow partial checkouts, so duplicating this approach in Git would force developers to keep a local copy of modules they are not interested in touching. Commits in an enormous checkout would be slower than you'd expect as Git would have to scan every directory for changes. If modules have a lot of local history, clones would take forever.

On the plus side, distributed revision control systems can much better integrate with external sources. In a centralized model, a single arbitrary snapshot of the external project is exported from its own revision control and then imported into the local revision control on a vendor branch. All the history is hidden. With distributed revision control you can clone the entire external history and much more easily follow development and re-merge local changes.

Git's submodule support allows a repository to contain, as a subdirectory, a checkout of an external project. Submodules maintain their own identity; the submodule support just stores the submodule repository location and commit ID, so other developers who clone the containing project ("superproject") can easily clone all the submodules at the same revision. Partial checkouts of the superproject are possible: you can tell Git to clone none, some or all of the submodules.

The git-submodule(1) command is available since Git 1.5.3. Users with Git 1.5.2 can look up the submodule commits in the repository and manually check them out; earlier versions won't recognize the submodules at all.

To see how submodule support works, create four example repositories that can be used later as a submodule:

```
$ mkdir ~/git
$ cd ~/git
$ for i in a b c d
do
        mkdir $i
        cd $i
        git init
        echo "module $i" > $i.txt
        git add $i.txt
        git commit -m "Initial commit, submodule $i"
        cd ..
done
```

Now create the superproject and add all the submodules:

```
$ mkdir super
```

```
$ cd super
$ git init
$ for i in a b c d
do
        git submodule add ~/git/$i $i
done
```

> **Note**
> Do not use local URLs here if you plan to publish your superproject!

See what files `git submodule` created:

```
$ ls -a
.  ..  .git  .gitmodules  a  b  c  d
```

The `git submodule add <repo> <path>` command does a couple of things:

- It clones the submodule from `<repo>` to the given `<path>` under the current directory and by default checks out the master branch.
- It adds the submodule's clone path to the gitmodules(5) file and adds this file to the index, ready to be committed.
- It adds the submodule's current commit ID to the index, ready to be committed.

Commit the superproject:

```
$ git commit -m "Add submodules a, b, c and d."
```

Now clone the superproject:

```
$ cd ..
$ git clone super cloned
$ cd cloned
```

The submodule directories are there, but they're empty:

```
$ ls -a a
.  ..
$ git submodule status
-d266b9873ad50488163457f025db7cdd9683d88b a
-e81d457da15309b4fef4249aba9b50187999670d b
-c1536a972b9affea0f16e0680ba87332dc059146 c
-d96249ff5d57de5de093e6baff9e0aafa5276a74 d
```

> **Note**
> The commit object names shown above would be different for you, but they should match the HEAD commit object names of your repositories. You can check it by running `git ls-remote ../a`.

Pulling down the submodules is a two-step process. First run `git submodule init` to add the submodule repository URLs to `.git/config`:

```
$ git submodule init
```

Now use `git submodule update` to clone the repositories and check out the commits specified in the superproject:

```
$ git submodule update
$ cd a
$ ls -a
.  ..  .git  a.txt
```

One major difference between `git submodule update` and `git submodule add` is that `git submodule update` checks out a specific commit, rather than the tip of a branch. It's like checking out a tag: the head is detached, so you're not working on a branch.

```
$ git branch
* (detached from d266b98)
  master
```

If you want to make a change within a submodule and you have a detached head, then you should create or checkout a branch, make your changes, publish the change within the submodule, and then update the superproject to reference the new commit:

```
$ git checkout master
```

or

```
$ git checkout -b fix-up
```

then

```
$ echo "adding a line again" >> a.txt
$ git commit -a -m "Updated the submodule from within the superproject."
$ git push
$ cd ..
$ git diff
diff --git a/a b/a
index d266b98..261dfac 160000
--- a/a
+++ b/a
@@ -1 +1 @@
-Subproject commit d266b9873ad50488163457f025db7cdd9683d88b
+Subproject commit 261dfac35cb99d380eb966e102c1197139f7fa24
$ git add a
$ git commit -m "Updated submodule a."
$ git push
```

You have to run `git submodule update` after `git pull` if you want to update submodules, too.

8.1   Pitfalls with submodules

Always publish the submodule change before publishing the change to the superproject that references it. If you forget to publish the submodule change, others won't be able to clone the repository:

```
$ cd ~/git/super/a
$ echo i added another line to this file >> a.txt
$ git commit -a -m "doing it wrong this time"
$ cd ..
$ git add a
$ git commit -m "Updated submodule a again."
$ git push
$ cd ~/git/cloned
```

```
$ git pull
$ git submodule update
error: pathspec '261dfac35cb99d380eb966e102c1197139f7fa24' did not match any file( ←
    s) known to git.
Did you forget to 'git add'?
Unable to checkout '261dfac35cb99d380eb966e102c1197139f7fa24' in submodule path 'a ←
    '
```

In older Git versions it could be easily forgotten to commit new or modified files in a submodule, which silently leads to similar problems as not pushing the submodule changes. Starting with Git 1.7.0 both `git status` and `git diff` in the superproject show submodules as modified when they contain new or modified files to protect against accidentally committing such a state. `git diff` will also add a `-dirty` to the work tree side when generating patch output or used with the `--submodule` option:

```
$ git diff
diff --git a/sub b/sub
--- a/sub
+++ b/sub
@@ -1 +1 @@
-Subproject commit 3f356705649b5d566d97ff843cf193359229a453
+Subproject commit 3f356705649b5d566d97ff843cf193359229a453-dirty
$ git diff --submodule
Submodule sub 3f35670..3f35670-dirty:
```

You also should not rewind branches in a submodule beyond commits that were ever recorded in any superproject.

It's not safe to run `git submodule update` if you've made and committed changes within a submodule without checking out a branch first. They will be silently overwritten:

```
$ cat a.txt
module a
$ echo line added from private2 >> a.txt
$ git commit -a -m "line added inside private2"
$ cd ..
$ git submodule update
Submodule path 'a': checked out 'd266b9873ad50488163457f025db7cdd9683d88b'
$ cd a
$ cat a.txt
module a
```

Note
The changes are still visible in the submodule's reflog.

If you have uncommitted changes in your submodule working tree, `git submodule update` will not overwrite them. Instead, you get the usual warning about not being able switch from a dirty branch.

Chapter 9

Low-level Git operations

Many of the higher-level commands were originally implemented as shell scripts using a smaller core of low-level Git commands. These can still be useful when doing unusual things with Git, or just as a way to understand its inner workings.

9.1   Object access and manipulation

The git-cat-file(1) command can show the contents of any object, though the higher-level git-show(1) is usually more useful.

The git-commit-tree(1) command allows constructing commits with arbitrary parents and trees.

A tree can be created with git-write-tree(1) and its data can be accessed by git-ls-tree(1). Two trees can be compared with git-diff-tree(1).

A tag is created with git-mktag(1), and the signature can be verified by git-verify-tag(1), though it is normally simpler to use git-tag(1) for both.

9.2   The Workflow

High-level operations such as git-commit(1), git-checkout(1) and git-reset(1) work by moving data between the working tree, the index, and the object database. Git provides low-level operations which perform each of these steps individually.

Generally, all Git operations work on the index file. Some operations work **purely** on the index file (showing the current state of the index), but most operations move data between the index file and either the database or the working directory. Thus there are four main combinations:

9.2.1   working directory → index

The git-update-index(1) command updates the index with information from the working directory. You generally update the index information by just specifying the filename you want to update, like so:

```
$ git update-index filename
```

but to avoid common mistakes with filename globbing etc., the command will not normally add totally new entries or remove old entries, i.e. it will normally just update existing cache entries.

To tell Git that yes, you really do realize that certain files no longer exist, or that new files should be added, you should use the --remove and --add flags respectively.

NOTE! A --remove flag does *not* mean that subsequent filenames will necessarily be removed: if the files still exist in your directory structure, the index will be updated with their new status, not removed. The only thing --remove means is that

update-index will be considering a removed file to be a valid thing, and if the file really does not exist any more, it will update the index accordingly.

As a special case, you can also do `git update-index --refresh`, which will refresh the "stat" information of each index to match the current stat information. It will *not* update the object status itself, and it will only update the fields that are used to quickly test whether an object still matches its old backing store object.

The previously introduced git-add(1) is just a wrapper for git-update-index(1).

9.2.2 index → object database

You write your current index file to a "tree" object with the program

```
$ git write-tree
```

that doesn't come with any options—it will just write out the current index into the set of tree objects that describe that state, and it will return the name of the resulting top-level tree. You can use that tree to re-generate the index at any time by going in the other direction:

9.2.3 object database → index

You read a "tree" file from the object database, and use that to populate (and overwrite—don't do this if your index contains any unsaved state that you might want to restore later!) your current index. Normal operation is just

```
$ git read-tree <SHA-1 of tree>
```

and your index file will now be equivalent to the tree that you saved earlier. However, that is only your *index* file: your working directory contents have not been modified.

9.2.4 index → working directory

You update your working directory from the index by "checking out" files. This is not a very common operation, since normally you'd just keep your files updated, and rather than write to your working directory, you'd tell the index files about the changes in your working directory (i.e. `git update-index`).

However, if you decide to jump to a new version, or check out somebody else's version, or just restore a previous tree, you'd populate your index file with read-tree, and then you need to check out the result with

```
$ git checkout-index filename
```

or, if you want to check out all of the index, use `-a`.

NOTE! `git checkout-index` normally refuses to overwrite old files, so if you have an old version of the tree already checked out, you will need to use the `-f` flag (*before* the `-a` flag or the filename) to *force* the checkout.

Finally, there are a few odds and ends which are not purely moving from one representation to the other:

9.2.5 Tying it all together

To commit a tree you have instantiated with `git write-tree`, you'd create a "commit" object that refers to that tree and the history behind it—most notably the "parent" commits that preceded it in history.

Normally a "commit" has one parent: the previous state of the tree before a certain change was made. However, sometimes it can have two or more parent commits, in which case we call it a "merge", due to the fact that such a commit brings together ("merges") two or more previous states represented by other commits.

In other words, while a "tree" represents a particular directory state of a working directory, a "commit" represents that state in time, and explains how we got there.
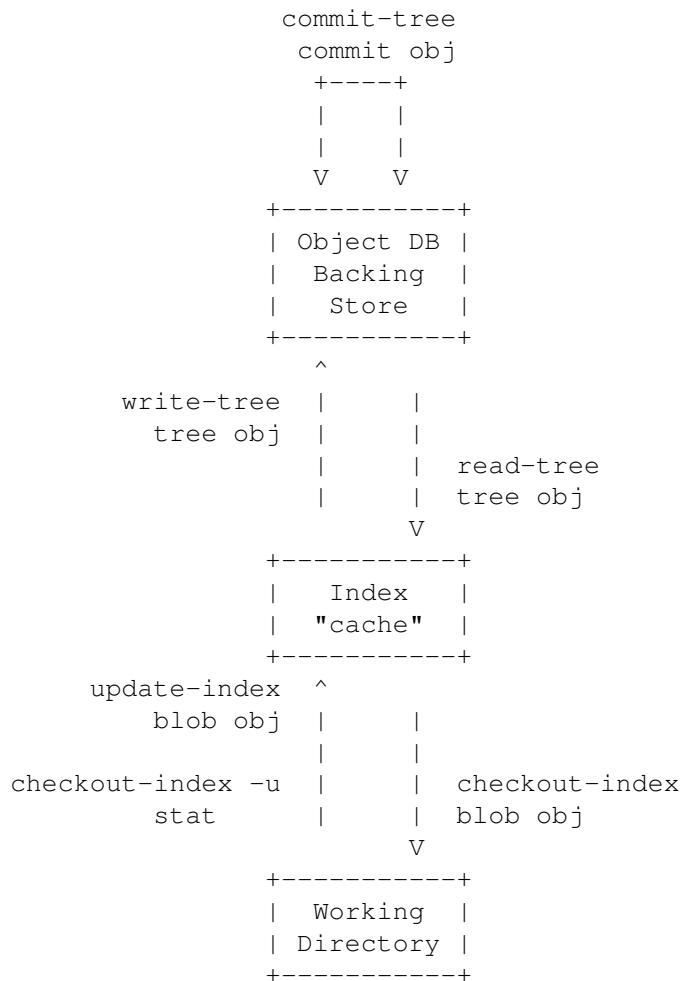
You create a commit object by giving it the tree that describes the state at the time of the commit, and a list of parents:

```
$ git commit-tree <tree> -p <parent> [(-p <parent2>)...]
```

and then giving the reason for the commit on stdin (either through redirection from a pipe or file, or by just typing it at the tty).

`git commit-tree` will return the name of the object that represents that commit, and you should save it away for later use. Normally, you'd commit a new `HEAD` state, and while Git doesn't care where you save the note about that state, in practice we tend to just write the result to the file pointed at by `.git/HEAD`, so that we can always see what the last committed state was.

Here is a picture that illustrates how various pieces fit together:

```
                 commit-tree
                  commit obj
                   +----+
                   |    |
                   |    |
                   V    V
              +-----------+
              | Object DB |
              |  Backing  |
              |   Store   |
              +-----------+
                   ^
       write-tree  |      |
         tree obj  |      |
                   |      |  read-tree
                   |      |  tree obj
                        V
              +-----------+
              |   Index   |
              |  "cache"  |
              +-----------+
     update-index  ^
         blob obj  |      |
                   |      |
   checkout-index -u  |      |  checkout-index
         stat      |      |  blob obj
                        V
              +-----------+
              |  Working  |
              | Directory |
              +-----------+
```

9.3 Examining the data

You can examine the data represented in the object database and the index with various helper tools. For every object, you can use git-cat-file(1) to examine details about the object:

```
$ git cat-file -t <objectname>
```

shows the type of the object, and once you have the type (which is usually implicit in where you find the object), you can use

```
$ git cat-file blob|tree|commit|tag <objectname>
```

to show its contents. NOTE! Trees have binary content, and as a result there is a special helper for showing that content, called `git ls-tree`, which turns the binary content into a more easily readable form.

It's especially instructive to look at "commit" objects, since those tend to be small and fairly self-explanatory. In particular, if you follow the convention of having the top commit name in `.git/HEAD`, you can do

```
$ git cat-file commit HEAD
```

to see what the top commit was.

9.4 Merging multiple trees

Git can help you perform a three-way merge, which can in turn be used for a many-way merge by repeating the merge procedure several times. The usual situation is that you only do one three-way merge (reconciling two lines of history) and commit the result, but if you like to, you can merge several branches in one go.

To perform a three-way merge, you start with the two commits you want to merge, find their closest common parent (a third commit), and compare the trees corresponding to these three commits.

To get the "base" for the merge, look up the common parent of two commits:

```
$ git merge-base <commit1> <commit2>
```

This prints the name of a commit they are both based on. You should now look up the tree objects of those commits, which you can easily do with

```
$ git cat-file commit <commitname> | head -1
```

since the tree object information is always the first line in a commit object.

Once you know the three trees you are going to merge (the one "original" tree, aka the common tree, and the two "result" trees, aka the branches you want to merge), you do a "merge" read into the index. This will complain if it has to throw away your old index contents, so you should make sure that you've committed those—in fact you would normally always do a merge against your last commit (which should thus match what you have in your current index anyway).

To do the merge, do

```
$ git read-tree -m -u <origtree> <yourtree> <targettree>
```

which will do all trivial merge operations for you directly in the index file, and you can just write the result out with `git write-tree`.

9.5 Merging multiple trees, continued

Sadly, many merges aren't trivial. If there are files that have been added, moved or removed, or if both branches have modified the same file, you will be left with an index tree that contains "merge entries" in it. Such an index tree can *NOT* be written out to a tree object, and you will have to resolve any such merge clashes using other tools before you can write out the result.

You can examine such index state with `git ls-files --unmerged` command. An example:

```
$ git read-tree -m $orig HEAD $target
$ git ls-files --unmerged
100644 263414f423d0e4d70dae8fe53fa34614ff3e2860 1       hello.c
100644 06fa6a24256dc7e560efa5687fa84b51f0263c3a 2       hello.c
100644 cc44c73eb783565da5831b4d820c962954019b69 3       hello.c
```

Each line of the `git ls-files --unmerged` output begins with the blob mode bits, blob SHA-1, *stage number*, and the filename. The *stage number* is Git's way to say which tree it came from: stage 1 corresponds to the `$orig` tree, stage 2 to the `HEAD` tree, and stage 3 to the `$target` tree.

Earlier we said that trivial merges are done inside `git read-tree -m`. For example, if the file did not change from `$orig` to `HEAD` or `$target`, or if the file changed from `$orig` to `HEAD` and `$orig` to `$target` the same way, obviously the final outcome is what is in `HEAD`. What the above example shows is that file `hello.c` was changed from `$orig` to `HEAD` and `$orig` to `$target` in a different way. You could resolve this by running your favorite 3-way merge program, e.g. `diff3`, `merge`, or Git's own merge-file, on the blob objects from these three stages yourself, like this:

```
$ git cat-file blob 263414f >hello.c~1
$ git cat-file blob 06fa6a2 >hello.c~2
$ git cat-file blob cc44c73 >hello.c~3
$ git merge-file hello.c~2 hello.c~1 hello.c~3
```

This would leave the merge result in `hello.c~2` file, along with conflict markers if there are conflicts. After verifying the merge result makes sense, you can tell Git what the final merge result for this file is by:

```
$ mv -f hello.c~2 hello.c
$ git update-index hello.c
```

When a path is in the "unmerged" state, running `git update-index` for that path tells Git to mark the path resolved.

The above is the description of a Git merge at the lowest level, to help you understand what conceptually happens under the hood. In practice, nobody, not even Git itself, runs `git cat-file` three times for this. There is a `git merge-index` program that extracts the stages to temporary files and calls a "merge" script on it:

```
$ git merge-index git-merge-one-file hello.c
```

and that is what higher level `git merge -s resolve` is implemented with.

Chapter 10

Hacking Git

This chapter covers internal details of the Git implementation which probably only Git developers need to understand.

10.1   Object storage format

All objects have a statically determined "type" which identifies the format of the object (i.e. how it is used, and how it can refer to other objects). There are currently four different object types: "blob", "tree", "commit", and "tag".

Regardless of object type, all objects share the following characteristics: they are all deflated with zlib, and have a header that not only specifies their type, but also provides size information about the data in the object. It's worth noting that the SHA-1 hash that is used to name the object is the hash of the original data plus this header, so `sha1sum` *file* does not match the object name for *file*.

As a result, the general consistency of an object can always be tested independently of the contents or the type of the object: all objects can be validated by verifying that (a) their hashes match the content of the file and (b) the object successfully inflates to a stream of bytes that forms a sequence of `<ascii type without space> + <space> + <ascii decimal size> + <byte\0> + <binary object data>`.

The structured objects can further have their structure and connectivity to other objects verified. This is generally done with the `git fsck` program, which generates a full dependency graph of all objects, and verifies their internal consistency (in addition to just verifying their superficial consistency through the hash).

10.2   A birds-eye view of Git's source code

It is not always easy for new developers to find their way through Git's source code. This section gives you a little guidance to show where to start.

A good place to start is with the contents of the initial commit, with:

```
$ git checkout e83c5163
```

The initial revision lays the foundation for almost everything Git has today, but is small enough to read in one sitting.

Note that terminology has changed since that revision. For example, the README in that revision uses the word "changeset" to describe what we now call a commit.

Also, we do not call it "cache" any more, but rather "index"; however, the file is still called `cache.h`. Remark: Not much reason to change it now, especially since there is no good single name for it anyway, because it is basically *the* header file which is included by *all* of Git's C sources.

If you grasp the ideas in that initial commit, you should check out a more recent version and skim `cache.h`, `object.h` and `commit.h`.

In the early days, Git (in the tradition of UNIX) was a bunch of programs which were extremely simple, and which you used in scripts, piping the output of one into another. This turned out to be good for initial development, since it was easier to test new things. However, recently many of these parts have become builtins, and some of the core has been "libified", i.e. put into libgit.a for performance, portability reasons, and to avoid code duplication.

By now, you know what the index is (and find the corresponding data structures in `cache.h`), and that there are just a couple of object types (blobs, trees, commits and tags) which inherit their common structure from `struct object`, which is their first member (and thus, you can cast e.g. `(struct object *)commit` to achieve the *same* as `&commit->object`, i.e. get at the object name and flags).

Now is a good point to take a break to let this information sink in.

Next step: get familiar with the object naming. Read Section 2.2. There are quite a few ways to name an object (and not only revisions!). All of these are handled in `sha1_name.c`. Just have a quick look at the function `get_sha1()`. A lot of the special handling is done by functions like `get_sha1_basic()` or the likes.

This is just to get you into the groove for the most libified part of Git: the revision walker.

Basically, the initial version of `git log` was a shell script:

```
$ git-rev-list --pretty $(git-rev-parse --default HEAD "$@") | \
        LESS=-S ${PAGER:-less}
```

What does this mean?

`git rev-list` is the original version of the revision walker, which *always* printed a list of revisions to stdout. It is still functional, and needs to, since most new Git commands start out as scripts using `git rev-list`.

`git rev-parse` is not as important any more; it was only used to filter out options that were relevant for the different plumbing commands that were called by the script.

Most of what `git rev-list` did is contained in `revision.c` and `revision.h`. It wraps the options in a struct named `rev_info`, which controls how and what revisions are walked, and more.

The original job of `git rev-parse` is now taken by the function `setup_revisions()`, which parses the revisions and the common command-line options for the revision walker. This information is stored in the struct `rev_info` for later consumption. You can do your own command-line option parsing after calling `setup_revisions()`. After that, you have to call `prepare_revision_walk()` for initialization, and then you can get the commits one by one with the function `get_revision()`.

If you are interested in more details of the revision walking process, just have a look at the first implementation of `cmd_log()`; call `git show v1.3.0~155^2~4` and scroll down to that function (note that you no longer need to call `setup_pager()` directly).

Nowadays, `git log` is a builtin, which means that it is *contained* in the command `git`. The source side of a builtin is

- a function called `cmd_<bla>`, typically defined in `builtin/<bla.c>` (note that older versions of Git used to have it in `builtin-<bla>.c` instead), and declared in `builtin.h`.

- an entry in the `commands[]` array in `git.c`, and

- an entry in `BUILTIN_OBJECTS` in the `Makefile`.

Sometimes, more than one builtin is contained in one source file. For example, `cmd_whatchanged()` and `cmd_log()` both reside in `builtin/log.c`, since they share quite a bit of code. In that case, the commands which are *not* named like the `.c` file in which they live have to be listed in `BUILT_INS` in the `Makefile`.

`git log` looks more complicated in C than it does in the original script, but that allows for a much greater flexibility and performance.

Here again it is a good point to take a pause.

Lesson three is: study the code. Really, it is the best way to learn about the organization of Git (after you know the basic concepts).

So, think about something which you are interested in, say, "how can I access a blob just knowing the object name of it?". The first step is to find a Git command with which you can do it. In this example, it is either `git show` or `git cat-file`.

For the sake of clarity, let's stay with `git cat-file`, because it

- is plumbing, and

- was around even in the initial commit (it literally went only through some 20 revisions as `cat-file.c`, was renamed to `builtin/cat-file.c` when made a builtin, and then saw less than 10 versions).

So, look into `builtin/cat-file.c`, search for `cmd_cat_file()` and look what it does.

```
git_config(git_default_config);
if (argc != 3)
        usage("git cat-file [-t|-s|-e|-p|<type>] <sha1>");
if (get_sha1(argv[2], sha1))
        die("Not a valid object name %s", argv[2]);
```

Let's skip over the obvious details; the only really interesting part here is the call to `get_sha1()`. It tries to interpret `argv[2]` as an object name, and if it refers to an object which is present in the current repository, it writes the resulting SHA-1 into the variable `sha1`.

Two things are interesting here:

- `get_sha1()` returns 0 on *success*. This might surprise some new Git hackers, but there is a long tradition in UNIX to return different negative numbers in case of different errors—and 0 on success.

- the variable `sha1` in the function signature of `get_sha1()` is `unsigned char *`, but is actually expected to be a pointer to `unsigned char[20]`. This variable will contain the 160-bit SHA-1 of the given commit. Note that whenever a SHA-1 is passed as `unsigned char *`, it is the binary representation, as opposed to the ASCII representation in hex characters, which is passed as `char *`.

You will see both of these things throughout the code.

Now, for the meat:

```
case 0:
        buf = read_object_with_reference(sha1, argv[1], &size, NULL);
```

This is how you read a blob (actually, not only a blob, but any type of object). To know how the function `read_object_with_refer` actually works, find the source code for it (something like `git grep read_object_with | grep ":[a-z]"` in the Git repository), and read the source.

To find out how the result can be used, just read on in `cmd_cat_file()`:

```
write_or_die(1, buf, size);
```

Sometimes, you do not know where to look for a feature. In many such cases, it helps to search through the output of `git log`, and then `git show` the corresponding commit.

Example: If you know that there was some test case for `git bundle`, but do not remember where it was (yes, you *could* `git grep bundle t/`, but that does not illustrate the point!):

```
$ git log --no-merges t/
```

In the pager (`less`), just search for "bundle", go a few lines back, and see that it is in commit 18449ab0. Now just copy this object name, and paste it into the command line

```
$ git show 18449ab0
```

Voila.

Another example: Find out what to do in order to make some script a builtin:

```
$ git log --no-merges --diff-filter=A builtin/*.c
```

You see, Git is actually the best tool to find out about the source of Git itself!

Chapter 11

Git Glossary

11.1 Git explained

alternate object database

Via the alternates mechanism, a repository can inherit part of its object database from another object database, which is called an "alternate".

bare repository

A bare repository is normally an appropriately named directory with a `.git` suffix that does not have a locally checked-out copy of any of the files under revision control. That is, all of the Git administrative and control files that would normally be present in the hidden `.git` sub-directory are directly present in the `repository.git` directory instead, and no other files are present and checked out. Usually publishers of public repositories make bare repositories available.

blob object

Untyped object, e.g. the contents of a file.

branch

A "branch" is an active line of development. The most recent commit on a branch is referred to as the tip of that branch. The tip of the branch is referenced by a branch head, which moves forward as additional development is done on the branch. A single Git repository can track an arbitrary number of branches, but your working tree is associated with just one of them (the "current" or "checked out" branch), and HEAD points to that branch.

cache

Obsolete for: index.

chain

A list of objects, where each object in the list contains a reference to its successor (for example, the successor of a commit could be one of its parents).

changeset

BitKeeper/cvsps speak for "commit". Since Git does not store changes, but states, it really does not make sense to use the term "changesets" with Git.

checkout

The action of updating all or part of the working tree with a tree object or blob from the object database, and updating the index and HEAD if the whole working tree has been pointed at a new branch.

cherry-picking

In SCM jargon, "cherry pick" means to choose a subset of changes out of a series of changes (typically commits) and record them as a new series of changes on top of a different codebase. In Git, this is performed by the "git cherry-pick" command to extract the change introduced by an existing commit and to record it based on the tip of the current branch as a new commit.

clean

A working tree is clean, if it corresponds to the revision referenced by the current head. Also see "dirty".

commit

As a noun: A single point in the Git history; the entire history of a project is represented as a set of interrelated commits. The word "commit" is often used by Git in the same places other revision control systems use the words "revision" or "version". Also used as a short hand for commit object.

As a verb: The action of storing a new snapshot of the project's state in the Git history, by creating a new commit representing the current state of the index and advancing HEAD to point at the new commit.

commit object

An object which contains the information about a particular revision, such as parents, committer, author, date and the tree object which corresponds to the top directory of the stored revision.

commit-ish (also committish)

A commit object or an object that can be recursively dereferenced to a commit object. The following are all commit-ishes: a commit object, a tag object that points to a commit object, a tag object that points to a tag object that points to a commit object, etc.

core Git

Fundamental data structures and utilities of Git. Exposes only limited source code management tools.

DAG

Directed acyclic graph. The commit objects form a directed acyclic graph, because they have parents (directed), and the graph of commit objects is acyclic (there is no chain which begins and ends with the same object).

dangling object

An unreachable object which is not reachable even from other unreachable objects; a dangling object has no references to it from any reference or object in the repository.

detached HEAD

Normally the HEAD stores the name of a branch, and commands that operate on the history HEAD represents operate on the history leading to the tip of the branch the HEAD points at. However, Git also allows you to check out an arbitrary commit that isn't necessarily the tip of any particular branch. The HEAD in such a state is called "detached".

Note that commands that operate on the history of the current branch (e.g. `git commit` to build a new history on top of it) still work while the HEAD is detached. They update the HEAD to point at the tip of the updated history without affecting any branch. Commands that update or inquire information *about* the current branch (e.g. `git branch --set-upstream-to` that sets what remote-tracking branch the current branch integrates with) obviously do not work, as there is no (real) current branch to ask about in this state.

directory

The list you get with "ls" :-)

dirty

A working tree is said to be "dirty" if it contains modifications which have not been committed to the current branch.

evil merge

An evil merge is a merge that introduces changes that do not appear in any parent.

fast-forward

A fast-forward is a special type of merge where you have a revision and you are "merging" another branch's changes that happen to be a descendant of what you have. In such a case, you do not make a new merge commit but instead just update to his revision. This will happen frequently on a remote-tracking branch of a remote repository.

fetch

Fetching a branch means to get the branch's head ref from a remote repository, to find out which objects are missing from the local object database, and to get them, too. See also git-fetch(1).

file system

Linus Torvalds originally designed Git to be a user space file system, i.e. the infrastructure to hold files and directories. That ensured the efficiency and speed of Git.

Git archive

> Synonym for repository (for arch people).

gitfile

> A plain file .git at the root of a working tree that points at the directory that is the real repository.

grafts

> Grafts enables two otherwise different lines of development to be joined together by recording fake ancestry information for commits. This way you can make Git pretend the set of parents a commit has is different from what was recorded when the commit was created. Configured via the .git/info/grafts file.
>
> Note that the grafts mechanism is outdated and can lead to problems transferring objects between repositories; see git-replace(1) for a more flexible and robust system to do the same thing.

hash

> In Git's context, synonym for object name.

head

> A named reference to the commit at the tip of a branch. Heads are stored in a file in $GIT_DIR/refs/heads/ directory, except when using packed refs. (See git-pack-refs(1).)

HEAD

> The current branch. In more detail: Your working tree is normally derived from the state of the tree referred to by HEAD. HEAD is a reference to one of the heads in your repository, except when using a detached HEAD, in which case it directly references an arbitrary commit.

head ref

> A synonym for head.

hook

> During the normal execution of several Git commands, call-outs are made to optional scripts that allow a developer to add functionality or checking. Typically, the hooks allow for a command to be pre-verified and potentially aborted, and allow for a post-notification after the operation is done. The hook scripts are found in the $GIT_DIR/hooks/ directory, and are enabled by simply removing the .sample suffix from the filename. In earlier versions of Git you had to make them executable.

index

> A collection of files with stat information, whose contents are stored as objects. The index is a stored version of your working tree. Truth be told, it can also contain a second, and even a third version of a working tree, which are used when merging.

index entry

> The information regarding a particular file, stored in the index. An index entry can be unmerged, if a merge was started, but not yet finished (i.e. if the index contains multiple versions of that file).

master

> The default development branch. Whenever you create a Git repository, a branch named "master" is created, and becomes the active branch. In most cases, this contains the local development, though that is purely by convention and is not required.

merge

> As a verb: To bring the contents of another branch (possibly from an external repository) into the current branch. In the case where the merged-in branch is from a different repository, this is done by first fetching the remote branch and then merging the result into the current branch. This combination of fetch and merge operations is called a pull. Merging is performed by an automatic process that identifies changes made since the branches diverged, and then applies all those changes together. In cases where changes conflict, manual intervention may be required to complete the merge.
>
> As a noun: unless it is a fast-forward, a successful merge results in the creation of a new commit representing the result of the merge, and having as parents the tips of the merged branches. This commit is referred to as a "merge commit", or sometimes just a "merge".

object

The unit of storage in Git. It is uniquely identified by the SHA-1 of its contents. Consequently, an object can not be changed.

object database

Stores a set of "objects", and an individual object is identified by its object name. The objects usually live in `$GIT_DIR/objects`

object identifier

Synonym for object name.

object name

The unique identifier of an object. The object name is usually represented by a 40 character hexadecimal string. Also colloquially called SHA-1.

object type

One of the identifiers "commit", "tree", "tag" or "blob" describing the type of an object.

octopus

To merge more than two branches.

origin

The default upstream repository. Most projects have at least one upstream project which they track. By default *origin* is used for that purpose. New upstream updates will be fetched into remote-tracking branches named origin/name-of-upstream-branch, which you can see using `git branch -r`.

pack

A set of objects which have been compressed into one file (to save space or to transmit them efficiently).

pack index

The list of identifiers, and other information, of the objects in a pack, to assist in efficiently accessing the contents of a pack.

pathspec

Pattern used to limit paths in Git commands.

Pathspecs are used on the command line of "git ls-files", "git ls-tree", "git add", "git grep", "git diff", "git checkout", and many other commands to limit the scope of operations to some subset of the tree or worktree. See the documentation of each command for whether paths are relative to the current directory or toplevel. The pathspec syntax is as follows:

- any path matches itself

- the pathspec up to the last slash represents a directory prefix. The scope of that pathspec is limited to that subtree.

- the rest of the pathspec is a pattern for the remainder of the pathname. Paths relative to the directory prefix will be matched against that pattern using fnmatch(3); in particular, * and *?* *can* match directory separators.

For example, Documentation/*.jpg will match all .jpg files in the Documentation subtree, including Documentation/chapter_1/figure_1.jpg.

A pathspec that begins with a colon `:` has special meaning. In the short form, the leading colon `:` is followed by zero or more "magic signature" letters (which optionally is terminated by another colon `:`), and the remainder is the pattern to match against the path. The "magic signature" consists of ASCII symbols that are neither alphanumeric, glob, regex special characters nor colon. The optional colon that terminates the "magic signature" can be omitted if the pattern begins with a character that does not belong to "magic signature" symbol set and is not a colon.

In the long form, the leading colon `:` is followed by a open parenthesis `(`, a comma-separated list of zero or more "magic words", and a close parentheses `)`, and the remainder is the pattern to match against the path.

A pathspec with only a colon means "there is no pathspec". This form should not be combined with other pathspec.

top

The magic word `top` (magic signature: `/`) makes the pattern match from the root of the working tree, even when you are running the command from inside a subdirectory.

literal

Wildcards in the pattern such as `*` or `?` are treated as literal characters.

icase

Case insensitive match.

glob

Git treats the pattern as a shell glob suitable for consumption by fnmatch(3) with the FNM_PATHNAME flag: wildcards in the pattern will not match a / in the pathname. For example, "Documentation/*.html" matches "Documentation/git.html" but not "Documentation/ppc/ppc.html" or "tools/perf/Documentation/perf.html".

Two consecutive asterisks ("`**`") in patterns matched against full pathname may have special meaning:

- A leading "`**`" followed by a slash means match in all directories. For example, "`**/foo`" matches file or directory "`foo`" anywhere, the same as pattern "`foo`". "`**/foo/bar`" matches file or directory "`bar`" anywhere that is directly under directory "`foo`".
- A trailing "`/**`" matches everything inside. For example, "`abc/**`" matches all files inside directory "abc", relative to the location of the `.gitignore` file, with infinite depth.
- A slash followed by two consecutive asterisks then a slash matches zero or more directories. For example, "`a/**/b`" matches "`a/b`", "`a/x/b`", "`a/x/y/b`" and so on.
- Other consecutive asterisks are considered invalid.
  Glob magic is incompatible with literal magic.

attr

After `attr:` comes a space separated list of "attribute requirements", all of which must be met in order for the path to be considered a match; this is in addition to the usual non-magic pathspec pattern matching. See gitattributes(5).

Each of the attribute requirements for the path takes one of these forms:

- "`ATTR`" requires that the attribute `ATTR` be set.
- "`-ATTR`" requires that the attribute `ATTR` be unset.
- "`ATTR=VALUE`" requires that the attribute `ATTR` be set to the string `VALUE`.
- "`!ATTR`" requires that the attribute `ATTR` be unspecified.

exclude

After a path matches any non-exclude pathspec, it will be run through all exclude pathspecs (magic signature: `!` or its synonym `^`). If it matches, the path is ignored. When there is no non-exclude pathspec, the exclusion is applied to the result set as if invoked without any pathspec.

parent

A commit object contains a (possibly empty) list of the logical predecessor(s) in the line of development, i.e. its parents.

pickaxe

The term pickaxe refers to an option to the diffcore routines that help select changes that add or delete a given text string. With the `--pickaxe-all` option, it can be used to view the full changeset that introduced or removed, say, a particular line of text. See git-diff(1).

plumbing

Cute name for core Git.

porcelain

Cute name for programs and program suites depending on core Git, presenting a high level access to core Git. Porcelains expose more of a SCM interface than the plumbing.

per-worktree ref

Refs that are per-worktree, rather than global. This is presently only HEAD and any refs that start with `refs/bisect/`, but might later include other unusual refs.

pseudoref

Pseudorefs are a class of files under `$GIT_DIR` which behave like refs for the purposes of rev-parse, but which are treated specially by git. Pseudorefs both have names that are all-caps, and always start with a line consisting of a SHA-1 followed by whitespace. So, HEAD is not a pseudoref, because it is sometimes a symbolic ref. They might optionally contain some additional data. `MERGE_HEAD` and `CHERRY_PICK_HEAD` are examples. Unlike per-worktree refs, these files cannot be symbolic refs, and never have reflogs. They also cannot be updated through the normal ref update machinery. Instead, they are updated by directly writing to the files. However, they can be read as if they were refs, so `git rev-parse MERGE_HEAD` will work.

pull

Pulling a branch means to fetch it and merge it. See also git-pull(1).

push

Pushing a branch means to get the branch's head ref from a remote repository, find out if it is a direct ancestor to the branch's local head ref, and in that case, putting all objects, which are reachable from the local head ref, and which are missing from the remote repository, into the remote object database, and updating the remote head ref. If the remote head is not an ancestor to the local head, the push fails.

reachable

All of the ancestors of a given commit are said to be "reachable" from that commit. More generally, one object is reachable from another if we can reach the one from the other by a chain that follows tags to whatever they tag, commits to their parents or trees, and trees to the trees or blobs that they contain.

rebase

To reapply a series of changes from a branch to a different base, and reset the head of that branch to the result.

ref

A name that begins with `refs/` (e.g. `refs/heads/master`) that points to an object name or another ref (the latter is called a symbolic ref). For convenience, a ref can sometimes be abbreviated when used as an argument to a Git command; see gitrevisions(7) for details. Refs are stored in the repository.

The ref namespace is hierarchical. Different subhierarchies are used for different purposes (e.g. the `refs/heads/` hierarchy is used to represent local branches).

There are a few special-purpose refs that do not begin with `refs/`. The most notable example is `HEAD`.

reflog

A reflog shows the local "history" of a ref. In other words, it can tell you what the 3rd last revision in *this* repository was, and what was the current state in *this* repository, yesterday 9:14pm. See git-reflog(1) for details.

refspec

A "refspec" is used by fetch and push to describe the mapping between remote ref and local ref.

remote repository

A repository which is used to track the same project but resides somewhere else. To communicate with remotes, see fetch or push.

remote-tracking branch

A ref that is used to follow changes from another repository. It typically looks like *refs/remotes/foo/bar* (indicating that it tracks a branch named *bar* in a remote named *foo*), and matches the right-hand-side of a configured fetch refspec. A remote-tracking branch should not contain direct modifications or have local commits made to it.

repository

A collection of refs together with an object database containing all objects which are reachable from the refs, possibly accompanied by meta data from one or more porcelains. A repository can share an object database with other repositories via alternates mechanism.

resolve

The action of fixing up manually what a failed automatic merge left behind.

revision

Synonym for commit (the noun).

rewind

To throw away part of the development, i.e. to assign the head to an earlier revision.

SCM

Source code management (tool).

SHA-1

"Secure Hash Algorithm 1"; a cryptographic hash function. In the context of Git used as a synonym for object name.

shallow clone

Mostly a synonym to shallow repository but the phrase makes it more explicit that it was created by running `git clone --depth=...` command.

shallow repository

A shallow repository has an incomplete history some of whose commits have parents cauterized away (in other words, Git is told to pretend that these commits do not have the parents, even though they are recorded in the commit object). This is sometimes useful when you are interested only in the recent history of a project even though the real history recorded in the upstream is much larger. A shallow repository is created by giving the `--depth` option to git-clone(1), and its history can be later deepened with git-fetch(1).

stash entry

An object used to temporarily store the contents of a dirty working directory and the index for future reuse.

submodule

A repository that holds the history of a separate project inside another repository (the latter of which is called superproject).

superproject

A repository that references repositories of other projects in its working tree as submodules. The superproject knows about the names of (but does not hold copies of) commit objects of the contained submodules.

symref

Symbolic reference: instead of containing the SHA-1 id itself, it is of the format *ref: refs/some/thing* and when referenced, it recursively dereferences to this reference. *HEAD* is a prime example of a symref. Symbolic references are manipulated with the git-symbolic-ref(1) command.

tag

A ref under `refs/tags/` namespace that points to an object of an arbitrary type (typically a tag points to either a tag or a commit object). In contrast to a head, a tag is not updated by the `commit` command. A Git tag has nothing to do with a Lisp tag (which would be called an object type in Git's context). A tag is most typically used to mark a particular point in the commit ancestry chain.

tag object

An object containing a ref pointing to another object, which can contain a message just like a commit object. It can also contain a (PGP) signature, in which case it is called a "signed tag object".

topic branch

A regular Git branch that is used by a developer to identify a conceptual line of development. Since branches are very easy and inexpensive, it is often desirable to have several small branches that each contain very well defined concepts or small incremental yet related changes.

tree

Either a working tree, or a tree object together with the dependent blob and tree objects (i.e. a stored representation of a working tree).

tree object

An object containing a list of file names and modes along with refs to the associated blob and/or tree objects. A tree is equivalent to a directory.

tree-ish (also treeish)

A tree object or an object that can be recursively dereferenced to a tree object. Dereferencing a commit object yields the tree object corresponding to the revision's top directory. The following are all tree-ishes: a commit-ish, a tree object, a tag object that points to a tree object, a tag object that points to a tag object that points to a tree object, etc.

unmerged index

An index which contains unmerged index entries.

unreachable object

An object which is not reachable from a branch, tag, or any other reference.

upstream branch

> The default branch that is merged into the branch in question (or the branch in question is rebased onto). It is configured via branch.<name>.remote and branch.<name>.merge. If the upstream branch of *A* is *origin/B* sometimes we say "*A* is tracking *origin/B*".

working tree

> The tree of actual checked out files. The working tree normally contains the contents of the HEAD commit's tree, plus any local changes that you have made but not yet committed.

Appendix A

Git Quick Reference

This is a quick summary of the major commands; the previous chapters explain how these work in more detail.

A.1 Creating a new repository

From a tarball:

```
$ tar xzf project.tar.gz
$ cd project
$ git init
Initialized empty Git repository in .git/
$ git add .
$ git commit
```

From a remote repository:

```
$ git clone git://example.com/pub/project.git
$ cd project
```

A.2 Managing branches

```
$ git branch           # list all local branches in this repo
$ git checkout test  # switch working directory to branch "test"
$ git branch new     # create branch "new" starting at current HEAD
$ git branch -d new  # delete branch "new"
```

Instead of basing a new branch on current HEAD (the default), use:

```
$ git branch new test    # branch named "test"
$ git branch new v2.6.15 # tag named v2.6.15
$ git branch new HEAD^   # commit before the most recent
$ git branch new HEAD^^  # commit before that
$ git branch new test~10 # ten commits before tip of branch "test"
```

Create and switch to a new branch at the same time:

```
$ git checkout -b new v2.6.15
```

Update and examine branches from the repository you cloned from:

```
$ git fetch              # update
$ git branch -r          # list
  origin/master
  origin/next
  ...
$ git checkout -b masterwork origin/master
```

Fetch a branch from a different repository, and give it a new name in your repository:

```
$ git fetch git://example.com/project.git theirbranch:mybranch
$ git fetch git://example.com/project.git v2.6.15:mybranch
```

Keep a list of repositories you work with regularly:

```
$ git remote add example git://example.com/project.git
$ git remote                     # list remote repositories
example
origin
$ git remote show example        # get details
* remote example
  URL: git://example.com/project.git
  Tracked remote branches
    master
    next
    ...
$ git fetch example              # update branches from example
$ git branch -r                  # list all remote branches
```

A.3 Exploring history

```
$ gitk                    # visualize and browse history
$ git log                 # list all commits
$ git log src/            # ...modifying src/
$ git log v2.6.15..v2.6.16 # ...in v2.6.16, not in v2.6.15
$ git log master..test    # ...in branch test, not in branch master
$ git log test..master    # ...in branch master, but not in test
$ git log test...master   # ...in one branch, not in both
$ git log -S'foo()'       # ...where difference contain "foo()"
$ git log --since="2 weeks ago"
$ git log -p              # show patches as well
$ git show                # most recent commit
$ git diff v2.6.15..v2.6.16 # diff between two tagged versions
$ git diff v2.6.15..HEAD    # diff with current head
$ git grep "foo()"        # search working directory for "foo()"
$ git grep v2.6.15 "foo()"  # search old tree for "foo()"
$ git show v2.6.15:a.txt    # look at old version of a.txt
```

Search for regressions:

```
$ git bisect start
$ git bisect bad                 # current version is bad
$ git bisect good v2.6.13-rc2    # last known good revision
Bisecting: 675 revisions left to test after this
                                 # test here, then:
```

```
$ git bisect good            # if this revision is good, or
$ git bisect bad             # if this revision is bad.
                             # repeat until done.
```

A.4 Making changes

Make sure Git knows who to blame:

```
$ cat >>~/.gitconfig <<\EOF
[user]
        name = Your Name Comes Here
        email = you@yourdomain.example.com
EOF
```

Select file contents to include in the next commit, then make the commit:

```
$ git add a.txt     # updated file
$ git add b.txt     # new file
$ git rm c.txt      # old file
$ git commit
```

Or, prepare and create the commit in one step:

```
$ git commit d.txt # use latest content only of d.txt
$ git commit -a    # use latest content of all tracked files
```

A.5 Merging

```
$ git merge test   # merge branch "test" into the current branch
$ git pull git://example.com/project.git master
                   # fetch and merge in remote branch
$ git pull . test  # equivalent to git merge test
```

A.6 Sharing your changes

Importing or exporting patches:

```
$ git format-patch origin..HEAD # format a patch for each commit
                                # in HEAD but not in origin
$ git am mbox # import patches from the mailbox "mbox"
```

Fetch a branch in a different Git repository, then merge into the current branch:

```
$ git pull git://example.com/project.git theirbranch
```

Store the fetched branch into a local branch before merging into the current branch:

```
$ git pull git://example.com/project.git theirbranch:mybranch
```

After creating commits on a local branch, update the remote branch with your commits:

```
$ git push ssh://example.com/project.git mybranch:theirbranch
```

When remote and local branch are both named "test":

```
$ git push ssh://example.com/project.git test
```

Shortcut version for a frequently used remote repository:

```
$ git remote add example ssh://example.com/project.git
$ git push example test
```

A.7 Repository maintenance

Check for corruption:

```
$ git fsck
```

Recompress, remove unused cruft:

```
$ git gc
```

Appendix B

Notes and todo list for this manual

B.1   Todo list

This is a work in progress.

The basic requirements:

- It must be readable in order, from beginning to end, by someone intelligent with a basic grasp of the UNIX command line, but without any special knowledge of Git. If necessary, any other prerequisites should be specifically mentioned as they arise.

- Whenever possible, section headings should clearly describe the task they explain how to do, in language that requires no more knowledge than necessary: for example, "importing patches into a project" rather than "the `git am` command"

Think about how to create a clear chapter dependency graph that will allow people to get to important topics without necessarily reading everything in between.

Scan `Documentation/` for other stuff left out; in particular:

- howto's

- some of `technical/`?

- hooks

- list of commands in git(1)

Scan email archives for other stuff left out

Scan man pages to see if any assume more background than this manual provides.

Add more good examples. Entire sections of just cookbook examples might be a good idea; maybe make an "advanced examples" section a standard end-of-chapter section?

Include cross-references to the glossary, where appropriate.

Add a section on working with other version control systems, including CVS, Subversion, and just imports of series of release tarballs.

Write a chapter on using plumbing and writing scripts.

Alternates, clone -reference, etc.

More on recovery from repository corruption. See: http://marc.info/?l=git&m=117263864820799&w=2 http://marc.info/?l=git&m=117