Git User's Manual (for version 1.5.3 or newer)

COLLABORATORS

	TITLE : Git User's Manual (for version 1.5.3 or newer)	*REFERENCE* :

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		October 19, 2007	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Repositories and Branches **8**

1.1 How to get a git repository . . . 8
1.2 How to check out a different version of a project . . . 8
1.3 Understanding History: Commits . . . 9
1.3.1 Understanding history: commits, parents, and reachability . . . 10
1.3.2 Understanding history: History diagrams . . . 10
1.3.3 Understanding history: What is a branch? . . . 10
1.4 Manipulating branches . . . 10
1.5 Examining an old version without creating a new branch . . . 11
1.6 Examining branches from a remote repository . . . 11
1.7 Naming branches, tags, and other references . . . 12
1.8 Updating a repository with git fetch . . . 12
1.9 Fetching branches from other repositories . . . 12

2 Exploring git history **14**

2.1 How to use bisect to find a regression . . . 14
2.2 Naming commits . . . 15
2.3 Creating tags . . . 15
2.4 Browsing revisions . . . 16
2.5 Generating diffs . . . 16
2.6 Viewing old file versions . . . 16
2.7 Examples . . . 17
2.7.1 Counting the number of commits on a branch . . . 17
2.7.2 Check whether two branches point at the same history . . . 17
2.7.3 Find first tagged version including a given fix . . . 17
2.7.4 Showing commits unique to a given branch . . . 18
2.7.5 Creating a changelog and tarball for a software release . . . 19
2.7.6 Finding commits referencing a file with given content . . . 19

3 Developing with git **20**
3.1 Telling git your name . . . 20
3.2 Creating a new repository . . . 20
3.3 How to make a commit . . . 20
3.4 Creating good commit messages . . . 22
3.5 Ignoring files . . . 22
3.6 How to merge . . . 22
3.7 Resolving a merge . . . 23
3.7.1 Getting conflict-resolution help during a merge . . . 23
3.8 Undoing a merge . . . 24
3.9 Fast-forward merges . . . 25
3.10 Fixing mistakes . . . 25
3.10.1 Fixing a mistake with a new commit . . . 25
3.10.2 Fixing a mistake by editing history . . . 25
3.10.3 Checking out an old version of a file . . . 26
3.10.4 Temporarily setting aside work in progress . . . 26
3.11 Ensuring good performance . . . 26
3.12 Ensuring reliability . . . 26
3.12.1 Checking the repository for corruption . . . 26
3.12.2 Recovering lost changes . . . 27
3.12.2.1 Reflogs . . . 27
3.12.2.2 Examining dangling objects . . . 27

4 Sharing development with others **29**
4.1 Getting updates with git pull . . . 29
4.2 Submitting patches to a project . . . 29
4.3 Importing patches to a project . . . 30
4.4 Public git repositories . . . 30
4.4.1 Setting up a public repository . . . 31
4.4.2 Exporting a git repository via the git protocol . . . 31
4.4.3 Exporting a git repository via http . . . 31
4.4.4 Pushing changes to a public repository . . . 31
4.4.5 Setting up a shared repository . . . 32
4.4.6 Allowing web browsing of a repository . . . 32
4.5 Examples . . . 32
4.5.1 Maintaining topic branches for a Linux subsystem maintainer . . . 32

5 Rewriting history and maintaining patch series **37**
5.1 Creating the perfect patch series . . . 37
5.2 Keeping a patch series up to date using git-rebase . . . 37
5.3 Modifying a single commit . . . 38
5.4 Reordering or selecting from a patch series . . . 39
5.5 Other tools . . . 39
5.6 Problems with rewriting history . . . 39

6 Advanced branch management **41**
6.1 Fetching individual branches . . . 41
6.2 git fetch and fast-forwards . . . 41
6.3 Forcing git fetch to do non-fast-forward updates . . . 42
6.4 Configuring remote branches . . . 42

7 Git internals **43**
7.1 The Object Database . . . 43
7.2 Blob Object . . . 43
7.3 Tree Object . . . 44
7.4 Commit Object . . . 44
7.5 Trust . . . 44
7.6 Tag Object . . . 44
7.7 The "index" aka "Current Directory Cache" . . . 45
7.8 The Workflow . . . 45
7.8.1 working directory -> index . . . 45
7.8.2 index -> object database . . . 46
7.8.3 object database -> index . . . 46
7.8.4 index -> working directory . . . 46
7.8.5 Tying it all together . . . 46
7.9 Examining the data . . . 47
7.10 Merging multiple trees . . . 48
7.11 Merging multiple trees, continued . . . 48
7.12 How git stores objects efficiently: pack files . . . 49
7.13 Dangling objects . . . 50
7.14 A birds-eye view of Git's source code . . . 50

8 GIT Glossary **53**

A Git Quick Reference **57**

A.1 Creating a new repository . . . 57

A.2 Managing branches . . . 57

A.3 Exploring history . . . 58

A.4 Making changes . . . 59

A.5 Merging . . . 59

A.6 Sharing your changes . . . 59

A.7 Repository maintenance . . . 60

B Notes and todo list for this manual **61**

Preface

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. For a command such as "git clone", just use

```
$ man git-clone
```

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. 10MB download):
$ git clone git://git.kernel.org/pub/scm/git/git.git
        # the linux kernel (approx. 150MB download):
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.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-2.6" in the examples above). After you cd into this directory, you will see that it contains a copy of the project files, 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.A single git repository may contain multiple branches. It keeps track of them 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 "SHA1 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 the 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 you are deleting points to a commit which is not reachable from the current branch, this command will fail with a warning.

git branch -D <branch> even if the branch points to a commit not reachable from the current branch, you may know that that commit is still reachable from some other branch or tag. In that case it is safe to use this command to force git to delete the branch.

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: moving to "v2.6.17" which isn't a local branch
If you want to create a new branch from this checkout, 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 SHA1 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
* (no branch)
  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, 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
```

You cannot check out these remote-tracking branches, but you can examine them on a branch of your own, just as you would a tag:

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

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.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 git-rev-parse(1).

1.8 Updating a repository with git fetch

Eventually the developer cloned from will do additional work in her repository, creating new commits and advancing the branches to point at the new commits.The command "git fetch", with no arguments, will update all of the remote-tracking branches to the latest version found in her 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 linux-nfs git://linux-nfs.org/pub/nfs-2.6.git
$ git fetch linux-nfs
* refs/remotes/linux-nfs/master: storing branch 'master' ...
  commit: bf81b46
```

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

```
$ git branch -r
linux-nfs/master
origin/master
```

If you run "git fetch <remote>" later, the 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 "linux-nfs"]
        url = git://linux-nfs.org/pub/nfs-2.6.git
        fetch = +refs/heads/*:refs/remotes/linux-nfs/*
...
```

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 t
```

If you run "git branch" at this point, you'll see that git has temporarily moved you to a new branch named "bisect". This branch points to a commit (with commit id 65934...) that is reachable from v2.6.19 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 and delete the temporary "bisect" branch.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". Chose 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.

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 git-rev-parse(1) 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
```

Sometimes what you want instead is a set of patches:

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

will generate a file with a patch for each commit reachable from test but not from master. Note that if master also has commits which are not reachable from test, then the combined result of these patches will not be the same as the diff produced by the git-diff example.

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 SHA1'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 contained 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, you can 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 search for a line that looks like

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

Which shows that e05db0fd is reachable from itself, from v1.5.0-rc1, and from v1.5.0-rc2, but 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 git-rev-parse(1) 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 --format=tar --prefix=project/ HEAD | gzip >latest.tar.gz
```

will use HEAD to produce a tar archive in which each filename is preceded by "project/".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 -- filename |
        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 make sure the following lines appear in 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.)

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 new contents of a modified file, use

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

To add the contents of a new file to the index, use

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

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

```
$ 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. Tools that turn commits into email, for example, use the first line 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 .`" and "`git commit -a`" 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. 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, but including any
                        # nonconflicting changes from MERGE_HEAD
$ git show :3:file.txt  # the version from MERGE_HEAD, but including any
                        # nonconflicting changes from HEAD.
```

Since the stage 2 and stage 3 versions have already been updated with nonconflicting changes, the only remaining differences between them are the important ones; thus git-diff(1) can use the information in the index to show only those conflicts.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 a descendant of the other—so every commit present in the one is already contained in the other—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 previous 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 editing 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 edit 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 "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 apply`:

```
$ git stash apply
```

3.11 Ensuring good performance

On large repositories, git depends on compression to keep the history information from taking up to much space on disk or in memory.This compression is not performed automatically. Therefore you should occasionally run git-gc(1):

```
$ git gc
```

to recompress the archive. This can be very time-consuming, so you may prefer to run git-gc when you are not doing other work.

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. The most common warning by far is about "dangling" objects:

```
$ 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
...
```

Dangling objects are not a problem. At worst they may take up a little extra disk space. They can sometimes provide a last-resort method for recovering lost work—see Section 7.13 for details. However, if you wish, you can remove them with git-prune(1) or the —prune option to git-gc(1):

```
$ git gc --prune
```

This may be time-consuming. Unlike most other git operations (including git-gc when run without any options), it is not safe to prune while other git operations are in progress in the same repository.

3.12.2 Recovering lost changes

3.12.2.1 Reflogs

Say you modify a branch with git-reset(1) —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 head. This syntax can be used to 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 git-rev-parse(1) 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 7.13 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 make 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 by default "git pull" merges from the HEAD branch of the origin repository. So often you can accomplish the above with just a simple

```
$ git pull
```

More generally, a branch that is created from a remote 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. The former is actually very commonly used.

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.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 how they prefer such patches be handled.

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 --resolved
```

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
$ chmod a+x hooks/post-update
```

(For an explanation of the last two lines, see git-update-server-info(1), and the documentation Hooks used by git.)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. Normally this is a sign of something wrong. However, if you are sure you know what you're doing, you may force git-push to perform the update anyway by proceeding the branch name by a plus sign:

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

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 the working tree will not be updated by the push. This may lead to unexpected results if the branch you push to is the currently checked-out branch!As with git-fetch, you may also set up configuration options to save typing; so, for example, after

```
$ cat >>.git/config <<EOF
[remote "public-repo"]
        url = ssh://yourserver.com/~you/proj.git
EOF
```

you should be able to perform the above 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 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 git for CVS users 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".

4.4.6 Allowing web browsing of a repository

The gitweb cgi script provides users an easy way to browse your project's files and history without having to install git; see the file gitweb/INSTALL in the git source tree for instructions on setting it up.

4.5 Examples

4.5.1 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-2.6.git work
$ cd work
```

Linus's tree will be stored in the remote 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-2.6.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 the current tip of Linus's branch:

```
$ git checkout -b speed-up-spinlocks origin
```

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 pull it into the "test" branch in preparation to make it public:

```
$ git checkout test && git pull . 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.Some time 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 pull . 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.To create diffstat and shortlog summaries of changes to include in a "please pull" request to Linus you can use:

```
$ git diff --stat origin..release
```

and

```
$ git log -p origin..release | git shortlog
```

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 $1 && git pull . origin
        ;;
origin)
        before=$(cat .git/refs/remotes/origin/master)
```

```
        git fetch origin
        after=$(cat .git/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
}

if [ ! -f .git/refs/heads/"$1" ]
then
        echo "Can't see branch <$1>" 1>&2
        usage
fi

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 `ls .git/refs/heads`
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
       \
        o--o--o <-- 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 ".dotest"), 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
```

5.3 Modifying 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.You can also use a combination of this and git-rebase(1) to edit commits further back in your history. First, tag the problematic commit with

```
$ git tag bad mywork~5
```

(Either gitk or git-log may be useful for finding the commit.)Then check out that commit, edit it, and rebase the rest of the series on top of it (note that we could check out the commit on a temporary branch, but instead we're using a detached head):

```
$ git checkout bad
$ # make changes here and update the index
$ git commit --amend
$ git rebase --onto HEAD bad mywork
```

When you're done, you'll be left with mywork checked out, with the top patches on mywork reapplied on top of your modified commit. You can then clean up with

```
$ git tag -d bad
```

Note that the immutable nature of git history means that you haven't really "modified" existing commits; instead, you have replaced the old commits with new commits having new object names.

5.4 Reordering or selecting from a patch series

Given one existing commit, the git-cherry-pick(1) command allows you to apply the change introduced by that commit and create a new commit that records it. So, for example, if "mywork" points to a series of patches on top of "origin", you might do something like:

```
$ git checkout -b mywork-new origin
$ gitk origin..mywork &
```

And browse through the list of patches in the mywork branch using gitk, applying them (possibly in a different order) to mywork-new using cherry-pick, and possibly modifying them as you go using commit —amend. The git-gui(1) command may also help as it allows you to individually select diff hunks for inclusion in the index (by right-clicking on the diff hunk and choosing "Stage Hunk for Commit").Another technique is to use git-format-patch to create a series of patches, then reset the state to before the patches:

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

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

5.5 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.

5.6 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.

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 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, after

```
$ git config remote.example.url git://example.com/proj.git
```

then the following two commands will do the same thing:

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

Even better, if you add one more option:

```
$ git config remote.example.fetch master:refs/remotes/example/master
```

then the following commands will all do the same thing:

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

You can also add a "+" to force the update each time:

```
$ git config remote.example.fetch +master:ref/remotes/example/master
```

Don't do this unless you're sure you won't mind "git fetch" possibly throwing away commits on mybranch.Also note that all of the above configuration can be performed by directly editing the file .git/config instead of using git-config(1).See git-config(1) for more details on the configuration options mentioned above.

Chapter 7

Git internals

Git depends on two fundamental abstractions: the "object database", and the "current directory cache" aka "index".

7.1 The Object Database

The object database is literally just a content-addressable collection of objects. All objects are named by their content, which is approximated by the SHA1 hash of the object itself. Objects may refer to other objects (by referencing their SHA1 hash), and so you can build up a hierarchy of objects.All objects have a statically determined "type" which is determined at object creation time, and 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".A "blob" object cannot refer to any other object, and is, as the name implies, a pure storage object containing some user data. It is used to actually store the file data, i.e. a blob object is associated with some particular version of some file.A "tree" object is an object that 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" is associated with exactly one tree (the directory hierarchy at the time of the commit). In addition, a "commit" refers to one or more "parent" commit objects that describe the history of how we arrived at that directory hierarchy.As a special case, a commit object with no parents is called the "root" commit, and is the point of an initial project commit. Each project must have at least one root, and while you can tie several different root objects together into one project by creating a commit object which has two or more separate roots as its ultimate parents, that's probably just going to confuse people. So aim for the notion of "one root object per project", even if git itself does not enforce that.A "tag" object symbolically identifies and can be used to sign other objects. It contains the identifier and type of another object, a symbolic name (of course!) and, optionally, a signature.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 SHA1 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*. (Historical note: in the dawn of the age of git the hash was the sha1 of the *compressed* object.)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).The object types in some more detail:

7.2 Blob Object

A "blob" object is nothing but a binary blob of data, and doesn't refer to anything else. There is no signature or any other verification of the data, so while the object is consistent (it *is* indexed by its sha1 hash, so the data itself is certainly correct), it has absolutely no other attributes. No name associations, no permissions. It is purely a blob of data (i.e. normally "file contents").In particular, 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 in any way.A blob is typically created when git-update-index(1) is run, and its data can be accessed by git-cat-file(1).

7.3 Tree Object

The next hierarchical object type is the "tree" object. A tree object is a list of mode/name/blob data, sorted by name. Alternatively, the mode data may specify a directory mode, in which case instead of naming a blob, that name is associated with another TREE object.Like the "blob" object, a tree object is uniquely determined by the set contents, and so two separate but identical trees will always share the exact same object. This is true at all levels, i.e. it's true for a "leaf" tree (which does not refer to any other trees, only blobs) as well as for a whole subdirectory.For that reason a "tree" object is just a pure data abstraction: it has no history, no signatures, no verification of validity, except that since the contents are again protected by the hash itself, we can trust that the tree is immutable and its contents never change.So you can trust the contents of a tree to be valid, the same way you can trust the contents of a blob, but you don't know where those contents *came* from.Side note on trees: since a "tree" object is a sorted list of "filename+content", you can create a diff between two trees without actually having to unpack two trees. Just ignore all common parts, and your diff will look right. In other words, you can effectively (and efficiently) tell the difference between any two random trees by O(n) where "n" is the size of the difference, rather than the size of the tree.Side note 2 on trees: since the name of a "blob" depends entirely and exclusively on its contents (i.e. there are no names or permissions involved), you can see trivial renames or permission changes by noticing that the blob stayed the same. However, renames with data changes need a smarter "diff" implementation.A tree is 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).

7.4 Commit Object

The "commit" object is an object that introduces the notion of history into the picture. In contrast to the other objects, it doesn't just describe the physical state of a tree, it describes how we got there, and why.A "commit" is defined by the tree-object that it results in, the parent commits (zero, one or more) that led up to that point, and a comment on what happened. Again, a commit is not trusted per se: the contents are well-defined and "safe" due to the cryptographically strong signatures at all levels, but there is no reason to believe that the tree is "good" or that the merge information makes sense. The parents do not have to actually have any relationship with the result, for example.Note on commits: unlike some SCM's, commits do not contain rename information or file mode change information. All of that is implicit in the trees involved (the result tree, and the result trees of the parents), and describing that makes no sense in this idiotic file manager.A commit is created with git-commit-tree(1) and its data can be accessed by git-cat-file(1).

7.5 Trust

An aside on the notion of "trust". Trust is really outside the scope of "git", but it's worth noting a few things. First off, since everything is hashed with SHA1, you *can* trust that an object is intact and has not been messed with by external sources. So the name of an object uniquely identifies a known state - just not a state that you may want to trust.Furthermore, since the SHA1 signature of a commit refers to the SHA1 signatures of the tree it is associated with and the signatures of the parent, a single named commit specifies uniquely a whole set of history, with full contents. You can't later fake any step of the way once you have the name of a commit.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 (SHA1 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...

7.6 Tag Object

Git provides the "tag" object to simplify creating, managing and exchanging symbolic and signed tokens. The "tag" object at its simplest simply symbolically identifies another object by containing the sha1, type and symbolic name.However it can

optionally contain additional signature information (which git doesn't care about as long as there's less than 8k of it). This can then be verified externally to git.Note that despite the tag features, "git" itself only handles content integrity; the trust framework (and signature provision and verification) has to come from outside.A tag is created with git-mktag(1), its data can be accessed by git-cat-file(1), and the signature can be verified by git-verify-tag(1).

7.7 The "index" aka "Current Directory Cache"

The index is a simple binary file, which contains an efficient representation of the contents of a virtual directory. It does so by a simple array that associates a set of names, dates, permissions and content (aka "blob") objects together. The cache is always kept ordered by name, and names are unique (with a few very specific rules) at any point in time, but the cache has no long-term meaning, and can be partially updated at any time.In particular, the index certainly does not need to be consistent with the current directory contents (in fact, most operations will depend on different ways to make the index *not* be consistent with the directory hierarchy), but it has three very important attributes:*(a) it can re-generate the full state it caches (not just the directory structure: it contains pointers to the "blob" objects so that it can regenerate the data too)*As a special case, there is a clear and unambiguous one-way mapping from a current directory cache to a "tree object", which can be efficiently created from just the current directory cache without actually looking at any other data. So a directory cache at any one time uniquely specifies one and only one "tree" object (but has additional data to make it easy to match up that tree object with what has happened in the directory)*(b) it has efficient methods for finding inconsistencies between that cached state ("tree object waiting to be instantiated") and the current state.(c) it can additionally 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.*Those are the ONLY three things that the directory cache does. It's a cache, and the normal operation is to re-generate it completely from a known tree object, or update/compare it with a live tree that is being developed. If you blow the directory cache away entirely, you generally haven't lost any information as long as you have the name of the tree that it described.At the same time, the index is also the staging area for creating new trees, and creating a new tree always involves a controlled modification of the index file. In particular, the index file can have the representation of an intermediate tree that has not yet been instantiated. So the index can be thought of as a write-back cache, which can contain dirty information that has not yet been written back to the backing store.

7.8 The Workflow

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 to and from the index file. Either from the database or from the working directory. Thus there are four main combinations:

7.8.1 working directory -> index

You update the index with information from the working directory with the git-update-index(1) command. 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-cache 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.

7.8.2 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:

7.8.3 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 <sha1 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.

7.8.4 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
```

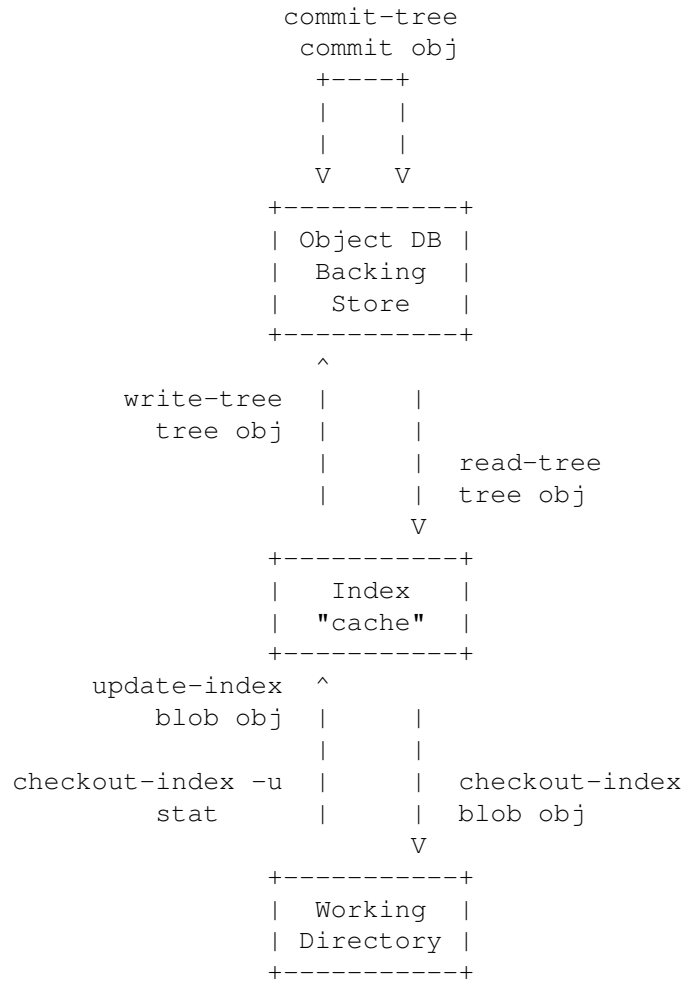
or, if you want to check out all of the index, use `-a`.NOTE! git-checkout-index normally refuses to overwrite old files, so if you have an old version of the tree already checked out, you will need to use the "-f" flag (*before* the "-a" flag or the filename) to *force* the checkout.Finally, there are a few odds and ends which are not purely moving from one representation to the other:

7.8.5 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 an ASCII art by Jon Loeliger that illustrates how various pieces fit together.



7.9 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.

7.10 Merging multiple trees

Git helps you do a three-way merge, which you can expand to n-way by repeating the merge procedure arbitrary times until you finally "commit" the state. The normal situation is that you'd only do one three-way merge (two parents), and commit it, but if you like to, you can do multiple parents in one go.To do a three-way merge, you need the two sets of "commit" objects that you want to merge, use those to find the closest common parent (a third "commit" object), and then use those commit objects to find the state of the directory ("tree" object) at these points.To get the "base" for the merge, you first look up the common parent of two commits with

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

which will return you the commit they are both based on. You should now look up the "tree" objects of those commits, which you can easily do with (for example)

```
$ 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`.

7.11 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 SHA1, *stage number*, and the filename. The *stage number* is git's way to say which tree it came from: stage 1 corresponds to `$orig` tree, stage 2 `HEAD` tree, and stage3 `$target` tree.Earlier we said that trivial merges are done inside `git-read-tree -m`. For example, if the file did not change from `$orig` to `HEAD` nor `$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 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, uses three `git-cat-file` for this. There is `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.

7.12 How git stores objects efficiently: pack files

We've seen how git stores each object in a file named after the object's SHA1 hash.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 technical/pack-format.txt.To put the loose objects into a pack, just run git repack:

```
$ git repack
Generating pack...
Done counting 6020 objects.
Deltifying 6020 objects.
 100% (6020/6020) done
Writing 6020 objects.
 100% (6020/6020) done
Total 6020, written 6020 (delta 4070), reused 0 (delta 0)
Pack pack-3e54ad29d5b2e05838c75df582c65257b8d08e1c created.
```

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.

7.13 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. But 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.(The same is true of "git-fsck" itself, btw - but since git-fsck never actually **changes** the repository, it just reports on what it found, git-fsck itself is never "dangerous" to run. Running it while somebody is actually changing the repository can cause confusing and scary messages, but it won't actually do anything bad. In contrast, running "git prune" while somebody is actively changing the repository is a **BAD** idea).

7.14 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 "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 programs 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`, 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_reference()` 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 8

GIT Glossary

alternate object database Via the alternates mechanism, a repository can inherit part of its object database from another object database, which is called "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 the working tree to a revision which was stored in the object database.

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 different codebase. In GIT, this is performed by "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.

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. However, git also allows you to check out an arbitrary commit that isn't necessarily the tip of any particular branch. In this case HEAD is said to be "detached".

dircache You are **waaaaay** behind. See index.

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.

ent Favorite synonym to "tree-ish" by some total geeks. See `http://en.wikipedia.org/wiki/Ent_(Middle-earth)` for an in-depth explanation. Avoid this term, not to confuse people.

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 these cases, you do not make a new merge commit but instead just update to his revision. This will happen frequently on a 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).

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.

hash In git's context, synonym to object name.

head A named reference to the commit at the tip of a branch. Heads are stored in `$GIT_DIR/refs/heads/`, 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 may reference 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 making 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 SHA1 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 hash of the object's contents using the Secure Hash Algorithm 1 and usually represented by the 40 character hexadecimal encoding of the hash of the object (possibly followed by a white space).

object type One of the identifiers "commit","tree","tag" or "blob" describing the type of an object.

octopus To merge more than two branches. Also denotes an intelligent predator.

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.

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.

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 an ancestor to the branch's local head ref is a direct, 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 40-byte hex representation of a SHA1 or a name that denotes a particular object. These may be stored in `$GIT_DIR/refs/`.

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. They are combined with a colon in the format <src>:<dst>, preceded by an optional plus sign, +. For example: `git fetch $URL refs/heads/master:refs/heads/origin` means "grab the master branch head from the $URL and store it as my origin branch head". And `git push $URL refs/heads/master:refs/heads/to-upstream` means "publish my master branch head as to-upstream branch at $URL". See also git-push(1)

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 A particular state of files and directories which was stored in the object database. It is referenced by a commit object.

rewind To throw away part of the development, i.e. to assign the head to an earlier revision.

SCM Source code management (tool).

SHA1 Synonym for object name.

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).

symref Symbolic reference: instead of containing the SHA1 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 pointing to a tag or commit object. In contrast to a head, a tag is not changed by a commit. Tags (not tag objects) are stored in `$GIT_DIR/refs/tags/`. 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.

tracking branch A regular git branch that is used to follow changes from another repository. A tracking branch should not contain direct modifications or have local commits made to it. A tracking branch can usually be identified as the right-hand-side ref in a Pull: refspec.

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 A ref pointing to either a commit object, a tree object, or a tag object pointing to a tag or commit or tree object.

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.

working tree The tree of actual checked out files. The working tree is normally equal to the HEAD 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 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

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 outScan man pages to see if any assume more background than this manual provides.Simplify beginning by suggesting disconnected head instead of temporary branch creation?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.Document shallow clones? See draft 1.5.0 release notes for some documentation.Add a section on working with other version control systems, including CVS, Subversion, and just imports of series of release tarballs.More details on gitweb?Write a chapter on using plumbing and writing scripts.Alternates, clone -reference, etc.git unpack-objects -r for recovery