Version Control with Git

Table des matières

[1 Introduction 1](#_Toc529350946)

[3 Getting Started 1](#_Toc529350947)

[Viewing your commits 2](#_Toc529350948)

[Viewing Commit differences 2](#_Toc529350949)

[Removing and Renaming Files in your repository 2](#_Toc529350950)

[Configuration Files 2](#_Toc529350951)

[Configuring an Alias 3](#_Toc529350952)

[4 Basic Git Concepts 3](#_Toc529350953)

[Git Object Types 3](#_Toc529350954)

[Inside the .git directory 4](#_Toc529350955)

[Files and Trees 4](#_Toc529350956)

[Tags 5](#_Toc529350957)

[5 File management and the index 5](#_Toc529350958)

[21 Git and Github 5](#_Toc529350959)

# 1 Introduction

A tool that manages and tracks different versions of software or other content is referred to generically as a version control system (VCS), a source code manager (SCM), a revision control system (RCS).

# 3 Getting Started

Git commands understand both “short” and “long” options. For example, the git commit command treats the following examples as equivalents

$ **git commit -m "Fixed a typo."**

$ **git commit --message="Fixed a typo."**

The short form, -m, uses a single hyphen, whereas the long form, --message, uses two.(This is consistent with the GNU long options extension.) Some options exist only in one form.

Finally, you can separate options from a list of arguments via the “bare double dash” convention. For instance, use the double dash to contrast the control portion of the command line from a list of operands, such as filenames.

$ **git diff -w master origin -- tools/Makefile**

You may need to use the double dash to separate and explicitly identify filenames if they might otherwise be mistaken for another part of the command. For example, if you happened to have both a file and a tag named *main.c*, then you will get different behavior:

# Checkout the tag named "main.c"

$ **git checkout main.c**

# Checkout the file named "main.c"

$ **git checkout -- main.c**

Git places all its revision information in this one, top-level *.git* directory.

## Viewing your commits

Once you have one or more commits in the repository, you can inspect them in a variety of ways.

$ **git log**

To see more detail about a particular commit, use git show with a commit number

$ **git show 9da581d910c9c4ac93557ca4859e767f5caf5169**

If you run git show without an explicit commit number, it simply shows the details of the most recent commit.

## Viewing Commit differences

To see the differences between the two revisions of *index.html*, recall both full commit ID names and run git diff:

$ **git diff 9da581d910c9c4ac93557ca4859e767f5caf5169 \**

**ec232cddfb94e0dfd5b5855af8ded7f5eb5c90d6**

# Removing and Renaming Files in your repository

Removing a file from a repository is analogous to adding a file but uses git rm.

## Configuration Files

Git’s configuration files are all simple text files in the style of *.ini* files. They record various choices and settings used by many Git commands. Some settings represent purely personal preferences (should a color.pager be used?); others are vital to a repository functioning correctly (core.repositoryformatversion); and still others tweak command behavior a bit (gc.auto).

Like many tools, Git supports a hierarchy of configuration files. In decreasing precedence they are:

*git/config*

* Repository-specific configuration settings manipulated with the --file option or by default. These settings have the highest precedence.

*~/.gitconfig*

* User-specific configuration settings manipulated with the --global option.

*/etc/gitconfig*

* System-wide configuration settings manipulated with the --system option if you have proper Unix file write permissions on it. These settings have the lowest precedence. Depending on your actual installation, the system settings file might be somewhere else (perhaps in */usr/local/etc/gitconfig*), or may be entirely absent.

Or, to set a repository-specific name and email address that would override a

--global setting, simply omit the --global flag:

$ **git config user.name "Jon Loeliger"**

$ **git config user.email "jdl@special-project.example.org"**

Use git config -l to list the settings of all the variables collectively found in the complete set of configuration files:

$ **git config –l**

Because the configuration files are simple text files, you can view their contents with cat and edit them with your favorite text editor, too.

$ **cat .git/config**

Use the --unset option to remove a setting:

$ **git config --unset --global user.email**

## Configuring an Alias

For starters, here is a tip for setting up command aliases. If there is a common but

complex Git command that you type frequently, consider setting up a simple Git alias

for it.

$ **git config --global alias.show-graph 'log --graph --abbrev-commit --pretty=oneline'**

# 4 Basic Git Concepts

A Git *repository* is simply a database containing all the information needed to retain and manage the revisions and history of a project

Within a repository, Git maintains two primary data structures, the *object store* and the *index*. All of this repository data is stored at the root of your working directory in a hidden subdirectory named *.git*.

The object store is designed to be efficiently copied during a clone operation as part of the mechanism that supports a fully distributed VCS. The index is transitory information, is private to a repository, and can be created or modified on demand as needed.

### Git Object Types

At the heart of Git’s repository implementation is the object store. It contains your original data files and all the log messages, author information, dates, and other information required to rebuild any version or branch of the project. Git places only four types of objects in the object store: the *blobs*, *trees*, *commits*, and *tags*. These four atomic objects form the foundation of Git’s higher level data structures.

*Blobs*

* Each version of a file is represented as a *blob*. Blob, a contraction of “binary large object,” is a term that’s commonly used in computing to refer to some variable or file that can contain any data and whose internal structure is ignored by the program. A blob is treated as being opaque. A blob holds a file’s data but does not contain any metadata about the file or even its name.

*Trees*

* A *tree* object represents one level of directory information. It records blob identifiers, path names, and a bit of metadata for all the files in one directory. It can also recursively reference other (sub)tree objects and thus build a complete hierarchy of files and subdirectories.

*Commits*

* A *commit* object holds metadata for each change introduced into the repository, including the author, committer, commit date, and log message. Each commit points to a tree object that captures, in one complete snapshot, the state of the repository at the time the commit was performed. The initial commit, or *root commit*, has no parent. Most commits have one commit parent, although later in the book (Chapter 9) we explain how a commit can reference more than one parent.

*Tags*

* A *tag* object assigns an arbitrary yet presumably human readable name to a specific object, usually a commit. Although 9da581d910c9c4ac93557ca4859e767f5caf5169 refers to an exact and well-defined commit, a more familiar tag name like Ver-1.0-Alpha might make more sense!

Over time, all the information in the object store changes and grows, tracking and modeling your project edits, additions, and deletions. To use disk space and network bandwidth efficiently, Git compresses and stores the objects in *pack files*, which are also placed in the object store.

The Git object store is organized and implemented as a content-addressable storage system. Specifically, each object in the object store has a unique name produced by applying SHA1 to the contents of the object, yielding an SHA1 hash value

## Inside the .git directory

To begin, initialize an empty repository using git init and then run find to reveal what’s created.

$ **mkdir /tmp/hello**

$ **cd /tmp/hello**

$ **git init**

Initialized empty Git repository in /tmp/hello/.git/

# List all the files in the current directory

$ **find .**

As you can see, *.git* contains a lot of stuff.

Initially, the *.git/objects* directory (the directory for all of Git’s objects) is empty, except

for a few placeholders.

$ **find .git/objects**

Let’s now carefully create a simple object:

$ **echo "hello world" > hello.txt**

$ **git add hello.txt**

If you typed “hello world” exactly as it appears here (with no changes to spacing or

capitalization), then your objects directory should now look like this:

$ **find .git/objects**

The hash in this case is 3b18e512dba79e4c8300dd08aeb37f8e728b8dad. The 160 bits of an SHA1 hash correspond to 20 bytes, which takes 40 bytes of hexadecimal to display, so the content is stored as *.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad*. Git inserts a */* after the first two digits to improve filesystem efficiency.

To show that Git really hasn’t done very much with the content in the file (it’s still the

same comforting “hello world”), you can use the hash to pull it back out of the object

store any time you want:

$ **git cat-file -p 3b18e512dba79e4c8300dd08aeb37f8e728b8dad**

hello world

## Files and Trees

The index is found in *.git/index* and keeps track of file pathnames and corresponding blobs. Each time you run commands such as git add, git rm, or git mv, Git updates the index with the new pathname and blob information

Whenever you want, you can create a tree object from your current index by capturing

a snapshot of its current information with the low-level git write-tree command.

At the moment, the index contains exactly one file, *hello.txt*.

$ **git ls-files -s**

100644 3b18e512dba79e4c8300dd08aeb37f8e728b8dad 0 hello.txt

Here you can see the association of the file, *hello.txt*, and the 3b18e5... blob.

Next, let’s capture the index state and save it to a tree object:

$ **git write-tree**

68aba62e560c0ebc3396e8ae9335232cd93a3f60

$ **find .git/objects**

.git/objects

.git/objects/68

But what does a tree look like? Because it’s an object, just like the blob, you can use

the same low-level command to view it.

$ **git cat-file -p 68aba6**

It is now easy to see that the tree object has captured the information that was in the

index when you ran git ls-files -s.

## Tags

Finally, the last object Git manages is the tag. Although Git implements only one kind

of tag object, there are two basic tag types, usually called *lightweight* and *annotated*.

You create an annotated, unsigned tag with a message on a commit using the git tag

command:

$ **git tag -m "Tag version 1.0" V1.0 3ede462**

You can see the tag object via the git cat-file -p command

Git usually tags a commit object, which points to a tree object, which encompasses the

total state of the entire hierarchy of files and directories within your repository

# 5 File management and the index

When you manage your code with Git, you edit in your working directory, accumulate changes in your index, and commit whatever has amassed in the index as a single changeset.

You can query the state of the index at any time with the command git status. It explicitly calls out what files Git considers staged. You can also peer into the internal state of Git with “plumbing” commands such as git ls-files.

You’ll also likely find the git diff command useful during staging. (Diffs are discussed extensively in Chapter 8.) This command can display two different sets of changes: git diff displays the changes that remain in your working directory and are not staged;

git diff --cached shows changes that are staged and will therefore contribute to your next commit

You can use both variations of git diff to guide you through the process of staging changes. Initially, git diff is a large set of all modifications, and --cached is empty. As you stage, the former set will shrink and the latter set will grow. If all your working changes are staged and ready for a commit, the --cached will be full and git diff will show nothing.

Editors and build environments often leave temporary or transient files among your source code. Such files usually shouldn’t be tracked as source files in a repository. To have Git ignore a file within a directory, simply add that file’s name to the special file *.gitignore*

You can use git ls-files to peer under the object model hood and find the SHA1 values for those staged files:

$ **git ls-files –stage**

Using git commit --all

The -a or --all option to git commit causes it to automatically stage all unstaged, tracked file changes—including removals of tracked files from the working copy— before it performs the commit.

Writing Commit Log Messages

If you are in the editor writing a commit log message and for some reason decide to abort the operation, simply exit the editor without saving; this results in an empty log message. If it’s too late for that because you’ve already saved, just delete the entire log message and save again. Git will not process an empty (no text) commit.

Using git rm

Git will remove a file only from the index or from the index and working directory simultaneously. Git will not remove a file just from the working directory; the regular rm command may be used for that purpose

Because git rm is also an operation on the index, the command won’t work on a file

that hasn’t been previously added to the repository or index; Git must first be aware of

a file

To convert a file from staged to unstaged, use git rm --cached:

Whereas git rm --cached removes the file from the index and leaves it in the working directory, git rm removes the file from both the index and the working directory

Before Git removes a file, it checks to make sure the version of the file in the working

directory matches the latest version in the current branch (the version that Git commands

call the HEAD). This verification precludes the accidental loss of any changes (due

to your editing) that may have been made to the file.

Use git rm -f to *force* the removal of your file. Force is an explicit mandate and removes the file even if you have altered it since your last commit.

Darn! Git removed the working copy, too! But don’t worry. VCSs are good at recovering

old versions of files:

$ **git checkout HEAD – data**

Using git mv

the following sequences of commands are equivalent Git operations:

$ **mv stuff newstuff**

$ **git rm stuff**

$ **git add newstuff**

and

$ **git mv stuff newstuff**

If you happen to check the history of the file, you may be a bit disturbed to see that Git has apparently lost the history of the original *data* file and remembers only that it renamed *data* to the current name:

$ **git log mydata**

Git does still remember the whole history, but the display is limited to the particular filename you specified in the command. The --follow option asks Git to trace back through the log and find the whole history associated with the content:

$ **git log --follow mydata**

One of the classic problems with VCSs is that renaming a file can cause them to lose track of a file’s history. Git preserves this information even after a rename.

Git, on the other hand, doesn’t keep track of a rename. You can move or copy *hello.txt* anywhere you want, but doing so affects only tree objects. (Remember that tree objects store the relationships between content, whereas the content itself is stored in blobs.) A look at the differences between two trees makes it obvious that the blob named 3b18e5... has moved to a new place

The .gitignore File

But Git also supports a much richer mechanism. A *.gitignore* file can contain a list of

filename *patterns* that specify what files to ignore. The format of *.gitignore* is as follows

• Blank lines are ignored, and lines starting with a pound sign (#) can be used for comments. However, the # does not represent a comment if it follows other text on the line.

• A simple, literal filename matches a file in any directory with that name.

• A directory name is marked by a trailing slash character (/). This matches the named directory and any subdirectory but does not match a file or a symbolic link.

• A pattern containing shell globbing characters, such as an asterisk (\*), is expanded as a shell glob pattern. Just as in standard shell globbing, the match cannot extend across directories and so an asterisk can match only a single file or directory name. But an asterisk can still be part of a pattern that includes slashes to specify directory names (e.g., *debug/32bit/\*.o*).

• An initial exclamation point (!) inverts the sense of the pattern on the rest of the line. Additionally, any file excluded by an earlier pattern but matching an inversion rule is included. An inverted pattern overrides lower precedence rules.

Furthermore, Git allows you to have a *.gitignore* file in any directory within your repository. Each file affects its directory and all subdirectories. The *.gitignore* rules also cascade: you can override the rules in a higher directory by including an inverted pattern (using the initial !) in one of the subdirectories.

To resolve a hierarchy with multiple *.gitignore* directories, and to allow command-line addenda to the list of ignored files, Git honors the following precedence, from highest to lowest:

• Patterns specified on the command line.

• Patterns read from *.gitignore* in the same directory.

• Patterns in parent directories, proceeding upward. Hence, the current directory’s patterns overrule the parents’ patterns, and the parents close to the current directory take precedence over higher parents.

• Patterns from the *.git/info/exclude* file.

• Patterns from the file specified by the configuration variable core.excludefile.

Because a *.gitignore* is treated as a regular file within your repository, it is copied during clone operations and applies to all copies of your repository. In general, you should place entries into your version controlled *.gitignore* files only if the patterns apply to *all* derived repositories universally.

If the exclusion pattern is somehow specific to your one repository and should not (or might not) be applicable to anyone else’s clone of your repository, then the patterns should instead go into the *.git/info/exclude* file, because it is not propagated during clone operations. Its pattern format and treatment is the same as *.gitignore* files.

Here’s another scenario. It’s typical to exclude *.o* files, which are generated from source by the compiler. To ignore *.o* files, place *\*.o* in your top level *.gitignore*. But what if you also had a particular *\*.o* file that was, say, supplied by someone else and for which you couldn’t generate a replacement yourself? You’d likely want to explicitly track that particular file. You might then have a configuration like this:

CHAPTER 6

Commits

refs and symrefs

A *ref* is an SHA1 hash ID that refers to an object within the Git object store. Although a ref may refer to any Git object, it usually refers to a commit object. A *symbolic reference*, or *symref*, is a name that indirectly points to a Git object. It is still just a ref.

Local topic branch names, remote tracking branch names, and tag names are all refs.

Each symbolic ref has an explicit, full name that begins with refs/ and each is stored hierarchically within the repository in the *.git/refs/* directory. There are basically three different namespaces represented in refs/: refs/heads/*ref* for your local branches, refs/remotes/*ref* for your remote tracking branches, and refs/tags/*ref* for your tags.

You can use either a full ref name or its abbreviation, but if you have a branch and a tag with the same name, Git applies a disambiguation heuristic and uses the first match according to this list from the git rev-parse manpage:

.git/*ref*

.git/refs/*ref*

.git/refs/tags/*ref*

.git/refs/heads/*ref*

.git/refs/remotes/*ref*

.git/refs/remotes/*ref*/HEAD

For example, a local topic branch named dev is really a short form of refs/heads/dev. Remote tracking branches are in the refs/remotes/ namespace, so origin/master really names refs/remotes/origin/master. And finally, a tag such as v2.6.23 is short for refs/tags/v2.6.23.

The first rule is usually just for a few refs described later: HEAD, ORIG\_HEAD, FETCH\_HEAD,

CHERRY\_PICK\_HEAD, and MERGE\_HEAD.

Git maintains several special symrefs automatically for particular purposes. They can be used anywhere a commit is used.

HEAD:

HEAD always refers to the most recent commit on the current branch. When you change branches, HEAD is updated to refer to the new branch’s latest commit.

ORIG\_HEAD

Certain operations, such as merge and reset, record the previous version of HEAD in

ORIG\_HEAD just prior to adjusting it to a new value. You can use ORIG\_HEAD to recover

or revert to the previous state or to make a comparison.

FETCH\_HEAD

When remote repositories are used, git fetch records the heads of all branches

fetched in the file *.git/FETCH\_HEAD*. FETCH\_HEAD is a shorthand for the head of

the last branch fetched and is valid only immediately after a fetch operation. Using

this symref, you can find the HEAD of commits from git fetch even if an anonymous

fetch that doesn’t specifically name a branch is used. The fetch operation is covered

in Chapter 12.

MERGE\_HEAD

When a merge is in progress, the tip of the *other* branch is temporarily recorded in

the symref MERGE\_HEAD. In other words, MERGE\_HEAD is the commit that is being

merged into HEAD.

All of these symbolic references are managed by the plumbing command git symbolic-ref.

Relative Commit Names

Git also provides mechanisms for identifying a commit relative to another reference,

commonly the tip of a branch.

Within a single generation, the caret is used to select a different parent. Given a commit

C, C^1 is the first parent, C^2 is the second parent, C^3 is the third parent, and so on, as

shown in Figure 6-1.

The tilde is used to go back before an ancestral parent and select a preceding generation. Again, given the commit C, C~1 is the first parent, C~2 is the first grandparent, and C~3

is the first great-grandparent. When there are multiple parents in a generation, the first

parent of the first parent is followed. You might also notice that both C^1 and C~1 refer

to the first parent; either name is correct, and is shown in Figure 6-2.



Commit History

Viewing Old Commits

One technique to constrain history is to specify a commit *range* using the form *since*..*until*. Given a range, git log shows all commits following *since* running through *until*. Here’s an example.

$ **git log --pretty=short --abbrev-commit master~12..master~10**

Use the -p option to print the patch, or changes, introduced by the commit.

$ **git log -1 -p 4fe86488**

The --stat option enumerates the files changed in a commit and tallies how many lines were modified in each file.

$ **git log --pretty=short --stat master~12..master~10**

Compare the output of git log --stat with the output of git diff --stat. There is a fundamental difference in their displays. The former produces a summary for each individual commit named in the range, whereas the latter prints a single summary of the total difference between two repository states named on the command line.

Another command to display objects from the object store is git show. You can use it to see a commit:

$ **git show HEAD~2**

or to see a specific blob object:

$ **git show origin/master:Makefile**

In the latter display, the blob shown is the *Makefile* from the branch named origin/master.

Commit Ranges

Many Git commands allow you to specify a *commit range*. In its simplest instantiation,

a commit range is a shorthand for a series of commits. More complex forms allow you

to include and exclude commits.

A range is denoted with a double-period (..), as in *start*..*end*, where *start* and *end*

The example used the range master~12..master~10 to specify the 11th and 10th prior commits on the master branch

The range M~12..M~10 represents two commits, the 11th and 10th oldest commits, which are labeled A and B. The range does not include M~12. Why? It’s a matter of definition. A *commit range*, *start*..*end*, is defined as the set of commits reachable from *end* that are not reachable from *start*. In other words, “the commit *end* is *included*” whereas “the commit *start* is *excluded*.” Usually this is simplified to just the phrase “in *end* but not *start*.”

When you specify a commit Y, to git log, you are actually requesting Git to show the log for all commits that are reachable from Y. You can exclude a specific commit X and all commits reachable from X with the expression ^X.

Combining the two forms, git log ^X Y is the same as git log X..Y and might be paraphrased as “give me all commits that are reachable from Y and don’t give me any commit leading up to and including X.”

The commit range X..Y is mathematically equivalent to ^X Y. You can also think of it as a set subtraction: Use everything leading up to Y minus everything leading up to and including X.



The range topic..master represents W, X, Y, and Z.



the range topic..master represents V, W, X, Y, and Z.



In this case, topic..master, contains only the commits W, X, Y, and Z

There are two other range permutations. If you leave either the *start* or *end* commits out of range, HEAD is assumed. Thus, ..*end* is equivalent to HEAD..*end* and *start*.. is equivalent to *start*..HEAD.

Finally, just as *start*..*end* can be thought of as representing a set subtraction operation, the notation *A*...*B* (using three periods) represents the *symmetric difference* between *A* and *B*, or the set of commits that are reachable from either *A* or *B* but not from both. Because of the function’s symmetry, neither commit can really be considered a start or end. In this sense *A* and *B* are equal.

More formally, the set of revisions in the symmetric difference between *A* and *B*,

*A*...*B*, is given by

$ **git rev-list *A B* --not $(git merge-base --all *A B*)**

Using git blame

Another tool you can use to help identify a particular commit is git blame. This command

tells you who last modified each line of a file and which commit made the change.

$ **git blame -L 35, init/version.c**

Using Pickaxe

Wheareas git blame tells you about the current state of a file, git log -S*string* searches back through the history of a file’s diffs for the given *string*. By searching the actual diffs between revisions, this command can find commits that perform a *change* in both additions and deletions.

$ **git log -Sinclude --pretty=oneline --abbrev-commit init/version.c**

Each of the commits listed on the left (cd354f1, etc.) will either add or delete lines that contain the word include. Be careful, though. If a commit both adds and subtracts exactly the same number of instances of lines with your key phrase, that won’t be shown. The commit must have a *change* in the number of additions and deletions in order to count. The -S option to git log is called *pickaxe*. That’s brute force archeology for you.

# CHAPTER 7 Branches

A *branch* is the fundamental means of launching a separate line of development within a software project.

Because the original commit from which a branch was started is not explicitly identified,

that commit (or its equivalent) can be found algorithmically using the name of the

original branch from which the new branch forked:

$ **git merge-base *original-branch new-branch***

For instance, to make a bug fix on the Version 2.3 release of your software, you might specify a branch named rel-2.3 as the starting commit:

$ **git branch prs/pr-1138 rel-2.3**

The git branch command lists branch names found in the repository.

$ **git branch**

Checking out Branches

As mentioned earlier in this chapter, your working directory can reflect only one branch at a time. To start working on a different branch, issue the git checkout command. Given a branch name, git checkout makes the branch the new, current working branch. It changes your working tree file and directory structure to match the state of the given branch.

$ **git checkout bug/pr-1**

The files and directory structure of your working tree have been updated to reflect the state and contents of the new branch, bug/pr-1.

Checking out When You Have Uncommitted Changes

Git precludes the accidental removal or modification of data in your local working tree

without your explicit request. Files and directories in your working directory that are

not being tracked are always left alone; Git won’t remove or modify them. However,

if you have local modifications to a file that are different from changes that are present

on the new branch, Git issues an error message such as the following and refuses to

check out the target branch:

If you really don’t care about losing changes in your working directory

and are willing to throw them away, you can force Git to perform the

checkout by using the -f option.

Merging Changes into a Different Branch

In the previous section, the current state of your working directory conflicted with that

of the branch you wanted to switch to. What’s needed is a merge: The changes in your

working directory must be merged with the files being checked out.

If possible or if specifically requested with the -m option, Git attempts to carry your

local change into the new working directory by performing a merge operation between

your local modifications and the target branch.

$ **git checkout -m dev**

Unless some problem prevents a checkout command from completing, the command:

$ **git checkout -b *new-branch start-point***

is exactly the same as the two-command sequence:

$ **git branch *new-branch start-point***

$ **git checkout *new-branch***

Detached HEAD Branches

However, you can check out any commit. In such an instance, Git creates a sort of

anonymous branch for you called a *detached HEAD*.

Deleting Branches

The command git branch -d *branch* removes the named branch from a repository. Git prevents you from removing the current branch:

$ **git branch -d bug/pr-3**

But there is another subtle issue. Git won’t allow you to delete a branch that contains commits that are not also present on the current branch. That is, Git prevents you from accidentally removing development in commits that will be lost if the branch were to be deleted.

Finally, as the error message suggests, you can override Git’s safety check by using –D instead of -d. Do this if you are certain you don’t want the extra content in that branch

Otherwise, without a reference to them, commits and blobs are unreachable and will eventually be collected as garbage by the git gc tool.

After accidentally removing a branch or other ref, you can recover it by using the git reflog command. Other commands such as git fsck and configuration options such as gc.reflogExpire and gc.pruneExpire can

also help recover lost commits, files, and branch heads.

# CHAPTER 8 Diffs

A *diff* is a compact summary of the differences (hence the name “diff”) between two items

Git has its own diff facility and can likewise produce a digest of differences. The command

git diff can compare files much akin to Unix’s diff command. Moreover, like diff -r, Git can traverse two *tree* objects and generate a representation of the variances. But git diff also has its own nuances and powerful features tailored to the particular needs of Git users

Forms of the git diff Command

There are three basic sources for tree or treelike objects to use with git diff:

• Any tree object anywhere within the entire commit graph

• Your working directory

• The index

Typically, the trees compared in a git diff command are named via commits, branch names, or tags

The git diff command can perform four fundamental comparisons using various combinations of those three sources.

* git diff

git diff shows the difference between your working directory and the index. It exposes what is *dirty* in your working directory and is thus a candidate to *stage* for your next commit. This command does not reveal differences between what’s in your index and what’s permanently stored in the repository (not to mention remote repositories you might be working with).

* git diff *commit*

This form summarizes the differences between your working directory and thegiven *commit*. Common variants of this command name HEAD or a particular branch name as the *commit*.

* git diff - -cached *commit*

This command shows the differences between the staged changes in the index andthe given *commit*. A common commit for the comparison—and the default if no commit is specified—is HEAD. With HEAD, this command shows you how your next commit will alter the current branch. If the option --cached doesn’t make sense to you, perhaps the synonym –staged will. It is available in Git version 1.6.1 and later.

* git diff *commit1 commit2*

If you specify two arbitrary commits, the command displays the differencesbetween the two. This command ignores the index and working directory, and it is the workhorse for arbitrary comparisons between two trees that are already in your object store.

The number of parameters on the command line determines what fundamental form is used and what is compared.

If you don’t supply a tree object or two, then git diff compares implied sources, such as your index or working directory.



In addition to the four basic forms of git diff, there are myriad options as well. Here are a few of the more useful ones.

--M

The --M option detects renames and generates a simplified output that simply records the file rename rather than the complete removal and subsequent addition of the source file. If the rename is not a pure rename but also has some additional content changes, Git calls those out.

-w *or* --ignore-all-space

Both -w and --ignore-all-space compare lines without considering changes in

whitespace as significant.

--stat

The --stat option adds statistics about the difference between any two tree states.

It reports in a compact syntax how many lines changed, how many were added,

and how many were elided.

--color

The --color option colorizes the output; a unique color represents each of the different types of changes present in the diff.

Finally, the git diff may be limited to show diffs for a specific set of files or directories

To get both staged and unstaged changes, use git diff HEAD. The lack of symmetry is unfortunate and counterintuitive.

git diff and Commit Ranges

The git diff command supports a double-dot syntax to represent the difference between two commits. Thus, the following two commands are equivalent:

$ **git diff master bug/pr-1**

$ **git diff master..bug/pr-1**

Unfortunately, the double-dot syntax in git diff means something quite different from

the same syntax in git log, which you learned about in Chapter 6.

Some points to keep in mind for the following example:

• git diff doesn’t care about the history of the files it compares or anything about

Branches

• git log is extremely conscious of how one file changed to become another—for example, when two branches diverged and what happened on each branch The log and diff commands perform two fundamentally different operations. Whereas log operates on a set of commits, diff operates on two different end points.

If you type git log -p master..bug/pr-1, you will see one commit, because the syntax master..bug/pr-1 represents all those commits in bug/pr-1 that are not also in master. The command traces back to the point where bug/pr-1 diverged from master, but it does not look at anything that happened to master since that point.

$ **git log -p master..bug/pr-1**

In contrast, git diff master..bug/pr-1 shows the total set of differences between the two trees represented by the heads of the master and bug/pr-1 branches. History doesn’t matter; only the current state of the files does.

$ **git diff master..bug/pr-1**



In this case, git log master..maint represents the five individual commits V, W, ..., Z. On the other hand, git diff master..maint represents the differences in the trees at H and Z, an accumulated 11 commits: C, D, ..., H and V, ..., Z

Similarly, both git log and git diff accept the form *commit1*...*commit2* to produce a *symmetrical difference*. As before, however, git log *commit1*...*commit2* and git diff *commit1*...*commit2* yield different results.

As discussed in “Commit Ranges” on page 78 of Chapter 6, the command git log *commit1*...*commit2* displays the commits reachable from either commit but not both. Thus, git log master...maint in the previous example would yield C, D, ...,H and V, ..., Z.

The symmetric difference in git diff shows the differences between a commit that is a common ancestor (or *merge base*) of *commit1* and *commit2*. Given the same genealogy in Figure 8-2, git diff master...maint combines the changes in the commits V, W, ..., Z.

git diff with Path Limiting

By default, the command git diff operates on the entire directory structure rooted at a given tree object. However, you can leverage the same *path limiting* technique employed by git log to limit the output of git diff to a subset of the repository.

For example, at one point1 in the development of the Git’s own repository, git diff --stat displayed this:

$ **git diff --stat master~5 master**

Of course, you can view the diffs for a single file, too.

$ **git diff master~5 master Documentation/git-add.txt**

# 21 Git and Github

Disk backup tools like SparkleShare.

*Pushing the Local Contents to GitHub*

Once one of the two options has been followed to connect the local repository to the remote repository, the contents of the local repo can be pushed to GitHub. This is done with the git push *remote branch* command. If the branch has never been published before, the more specific invocation git push -u origin master is appropriate, in which -u tells Git to track the pushed branch, push it to the origin remote, and to push just the master branch

Forks

The next idea that GitHub popularized, so much so that the phrase has spread to other

domains, is personal forks of projects (Figure 21-11). The term forking has commonly

carried a negative connotation. In the coding landscape of yesteryear, forking often

meant an aggressive parting of ways with the primary copy of the project with the intent

of taking the program in a different direction.

GitHub’s idea of forking is a positive one that enables a greater number of contributors

to make a greater number of contributions in a controlled and highly visible way. Forking

is the democratic ability of *any* potential contributor to get a personal copy of a

project’s code.