## Chapter W24 Version Control for Software Code Using git and GitHub

Objectives:

\* To introduce forms of source code version control in Linux, particularly Git

\* To detail the basic concepts and structure of a Git repository

\* To illustrate the Git staging model

\* To define a Directed Acyclic Graph (DAG)

\* To provide numerous practical examples of using the git command

\* To show strategies of managing a Git repository with branches via a long example

\* To give a tutorial on GitHub use

\* To allow you to use GitHub to obtain the supplementary materials for this book

Commands and primitives covered:

**git**

Note: The numbering of the Sections, and Examples in this chapter, proceeds in a mostly-sequential order as they have been selectively extracted from, and cross referenced to, the Sections and Examples in Chapter 14 of the printed book. This selective extraction is done according to the pedagogic needs of the presentation of this advanced topic chapter. They are therefore not numbered here in this chapter in strictly cardinal order. This non-cardinal numbering applies as well to In-Chapter Exercises, and End-of-Chapter Problems.

Anything that you are required to type on the command line is show in **bold** type.

## W24.5.7 Introduction to Version Control

Studies have shown that about two-thirds of the cost of a software product is spent on maintenance. As we mentioned before, the maintenance of a software product comprises corrective maintenance and enhancement. In corrective maintenance, the errors and bugs found after the deployment of a software product are fixed. In enhancement, the product is enhanced to include more features, such as an improved user interface. Regardless of its type, maintenance means changing and/or revising the source code for the product and generating new executables. This means using a *version control system* (VCS). As you revise source code, you may need to undo changes made to it and go back to an earlier version of the software. Moreover, if an individual or team of programmers is working on a piece of software, they should be able to locally and autonomously maintain editable (modifiable) versions that can be joined together at a convenient time.

Git and GitHub are atomic-level, distributed, content-oriented VCSs. *Atomic-level* means that when you take a snapshot of the software package, everything in it is captured in the snapshot at a single instance in time. *Distributed* means that the entire software package you are working with is available to all collaborators locally at all times. *Content-oriented* means that when you join different branches of work on the software, only the content of lines in the files along the branches you are merging are considered. Here, *content* means “with form,” and *context* means “with meaning.” Git only considers content when it operates on your software files and directories; the individual(s), collaborator(s), or integrator(s) of the software project are responsible for the merged context of the files in the software package. The combining of different lines of development of a project that causes content conflicts are indicated by Git with several useful strategies and mechanisms, and their resolution is aided by many tools that are available as add-ons to Git. But only the people writing and testing how the software works, and those people managing that process, are responsible for resolving merged-context conflicts.

In-Chapter Exercise W24.9

Give a detailed example of what you think is *content* tracking in a revision control system. Then contrast that example with what you think is *context* tracking.

Git, as a source code maintenance tool, is a software database for tracking changes made to a set of source code files over time. Although programmers most often use it to coordinate changes to software source code, you can use Git to track any kind of content. Git can

* Examine the state of your source code project at earlier points in time
* Show the differences among various states of the project, and the files present at those states
* Split the project development into multiple independent lines, called *branches*, which can evolve separately
* Regularly recombine branches by *merging*, or reconciling, the changes made in two or more branches
* Allow many people to work on a project simultaneously, sharing and combining their work as needed

Git is a member of the newer generation of distributed VCSs. Older systems such as SCCS, RCS, CVS, and Subversion are centralized, meaning that there is a single, central copy of the project content and history to which all users must refer. If the central copy is unavailable, all users must wait until the central copy is online again. Distributed systems such as Git have no central copy. Each user has a complete, independent copy of the entire project history, called a *repository*, and full access to all version control facilities. Network access is only needed to share changes among members of the same development team or group.

Git’s distributed nature accommodates many different styles of interaction, or *workflows*. Individuals can share work directly between their personal repositories. Git is the technology behind the *social coding* website GitHub, which includes many well-known open-source projects, most notably for the Linux kernel.

We discuss Git and GitHub in an examples-based tutorial presentation. For Git and GitHub, the following subsections will include:

1. How Git is used, and how it works on a Linux system

2. A high-level overview of the Git terminology, data structures, objects, and actions

3. Illustrations, both graphical and verbal, of the Git staging model, what a directed acyclic graph (DAG) is, and a finer-grained view of the object store contents

4. A short and a long example of how to create, edit, branch, and merge branches of a Git repository

5. An exploration of GitHub as a remote repository

6. Three basic examples of how to use git clone, git push, and git pull to transfer repository contents between a local repository and GitHub

To get the most useful information out of this section, you are encouraged to first read through the high-level background materials in first three subsections. Then do the examples in the subsequent three sections, as many times as necessary to become comfortable using Git and GitHub. Finally, be sure to do all in-chapter exercises and the problem set on Git and GitHub at the end of this chapter.

*W24.5.7.1 What Is Git Used for and How Does It Work?*

Git is used to manage one or more source code project repositories, each packaged into its own directory that you create for the source code files in that project. A repository is a database containing all the information needed to archive and manage the revisions and history of a project. In Git, as with most VCSs, a repository archives a complete copy of the entire project, with all of the revisions to it.

Git maintains a set of configuration settings and files within each repository. Unlike file data and other repository metadata, configuration settings are not propagated from one repository to another during a *cloning*, or *duplicating*, operation. Instead, Git manages and inspects the configuration and setup information on a per-site, per-user, and per-repository basis. Within a repository, Git maintains two primary data structures, the object store and the index. 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. All of this repository data is stored at the root of your working directory in a hidden subdirectory named **.git**.

Simply put, with Git and GitHub, you can manage your source code repositories and work in an independent, collaborative, or integrative management way. A majority of the introductory material we present here is aimed at the independent developer. From a learning point of view, you need to first know how to use Git in an independent way. Then we also show some collaborative workflows, particularly with GitHub. We do not touch on the integrative management techniques and commands of Git or GitHub.

In-Chapter Exercise W24.10

What do you think the role of an integrator of a project would be, in terms of what a revision control system accomplishes for a software development and maintenance program?

*W24.5.7.2 Basic Git Terminology*

Following is a glossary of basic Git terminology used throughout our examples. This glossary is partitioned into categories that reflect the basic structure of the Git repository itself.

W24.5.7.2.1 Top-Level Terminology

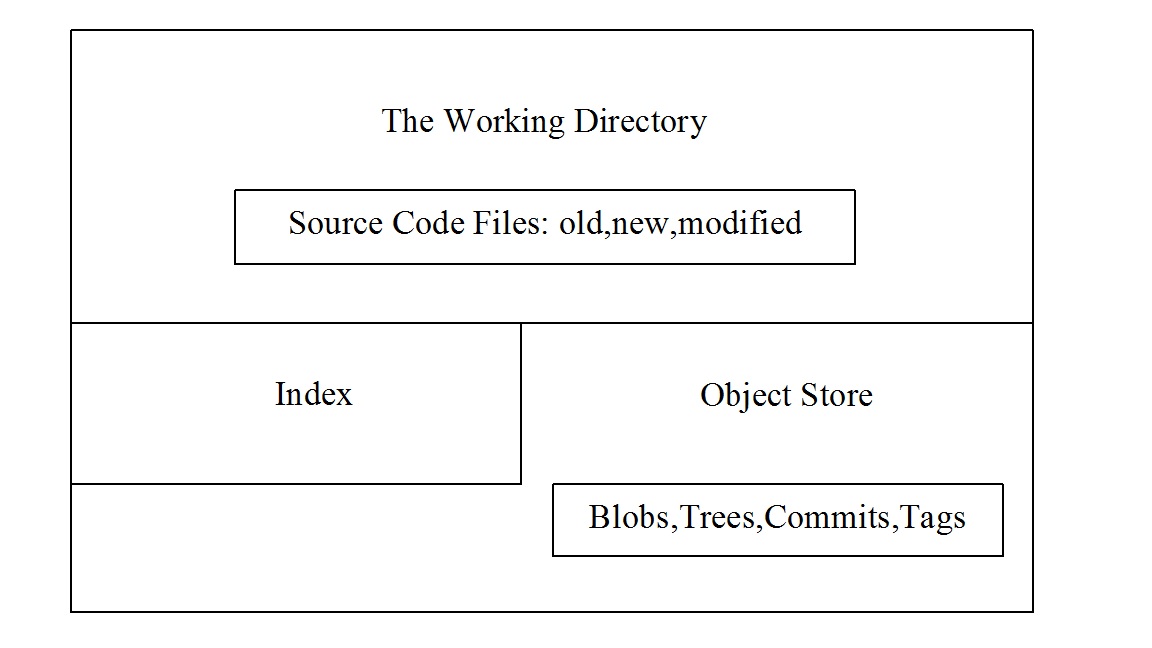


Figure W24.7 The structure of the repository

*Repository*:The repository is the *working directory*, and contains inside of itself the source code files you want to maintain and control, the *object store*, and the *index*, as shown in Figure W24.7. The advantage of having the repository self-sufficient inside of its own container is that the container can then be shared locally and globally.

A repository can also be thought of as a collection of *commits*, each of which is an archive of what the project’s *working tree* looked like at a past date, whether on your machine or someone else’s. It also defines *HEAD*, which identifies the branch or commit the current working tree stemmed from. It contains a set of branches and tags, to identify certain commits by name.

*Working directory*:The working directory is any directory on your file system that has a repository associated with it, typically indicated by the presence of a subdirectory within it named **.git**. It includes all the files and subdirectories in that directory.

W24.5.7.2.2 Primary Data Structures

*Object store*:Holds the changes in your source code over time, as you perform more commit operations. It is found in the **.git** subdirectory of your working directory. Its primary components or data structures are *blobs*, *trees*, *commits*, and *tags*.

*The index (staging area)*: This is a *cache*, or intermediate area, between your working tree and your repository. You can add changes to the index and build your next commit step by step. When your index content is complete, you then commit from the index. It is also used to keep information during failed merges (your side, their side, and current state). Unlike other, similar tools you may have used, Git does not commit changes directly from the working tree into the repository. Instead, changes are first made in the index. Think of it as a way of double-checking your additions or modifications, one by one, before doing a commit. You can also call it the *staging area*.

W24.5.7.2.3 Basic Object Types in the Object Store

*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 reference other subtree objects recursively 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. However, as we explain later, a commit, called a merge commit, can reference more than one parent. A commit is the state of your project, or of your working tree at some point in time. The state of HEAD at the time your commit is made becomes that commit’s parent. This is what creates the *revision history*.

*Tags*:A tag is also a name for a commit, similar to a branch, except that it always names the same commit, and can have its own shorthand descriptive text name. A tag object assigns an arbitrary, human-readable name to a specific object, usually a commit. Although the 40-digit-long hexadecimal number is an exact reference to a commit, a more tractable, understandable, and familiar tag name like **Ver-1.0-Beta** is more useful for humans.

W24.5.7.2.4 Components and Actions

*Working tree*:A working tree is a data structure component that represents the state of your source code files and directories at any given point in the history of the repository. Sometimes referenced as the contents of the index, it can be best thought of for beginners in Git as the data structure tree that loads or fills the working directory with its files and directories when you either create or checkout a commit.

*Adding*:Putting files from the working directory into the index for staging.

*Branch*:A branch is just a name for a line of commits, also called a reference. It is the parentage of a commit that defines its history—hence, the typical notion of a “branch of project development.” It can be simply thought of as a different line of development in the project. A branch in Git is just a “label” that points to a commit. You can get the full history through the parent pointers. A branch by default is only local to your repository.

*Checking out*:Bringing a branch of the repository into the working directory is called *checking out*.

*Directed acyclic graph (DAG)*:A DAG is a graph of the state of a repository, showing all commits and the parent–child relationships of the commits. It is also a good graphic representation of the branches, tags, and location of HEAD, if those are included in the graph. See Section W24.5.7.4 for more complete and descriptive information.

*Master*:The main line of development in most repositories is done on a branch called the *master*. It is the default name for the main branch of development.

*HEAD*:Your repository uses HEAD to define what is currently checked out. If you checkout a branch, HEAD symbolically refers to that branch, indicating that the branch name should be updated after the next commit operation. If you checkout a specific commit, HEAD refers to that commit only. This is referred to as a *detached HEAD*, and occurs, for example, if you check out a tag name.

*Clone*:A *clone* is a replicated copy of the entire repository, with all of its data structures, files, configurations, and so on.

*Merge*:The opposite of branch—that is, the fusion of branches and their commits.

In-Chapter Exercise W24.11

What do you think would be the quickest and easiest way to delete a local repository on your Linux system?

*W24.5.7.3 The Git Staging Model*

Git has three main states that your files can be in: *modified*, *staged*, and *committed*. Modified means that you have changed the file but have not committed it to your database yet. Staged means that you have designated a modified file in its current version to go into your next commit snapshot. Committed means that the data is safely stored in your Git repository database. It is held as a data structure consisting of the four types of objects in your object store.

The three main sections of a Git project are seen in Figure W24.8. They are the working directory (where you initially add, create, or modify files), the index or staging area (where you prepare files to be put into the repository), and the repository (that is, in a database held in the **.git** subdirectory of your working directory).

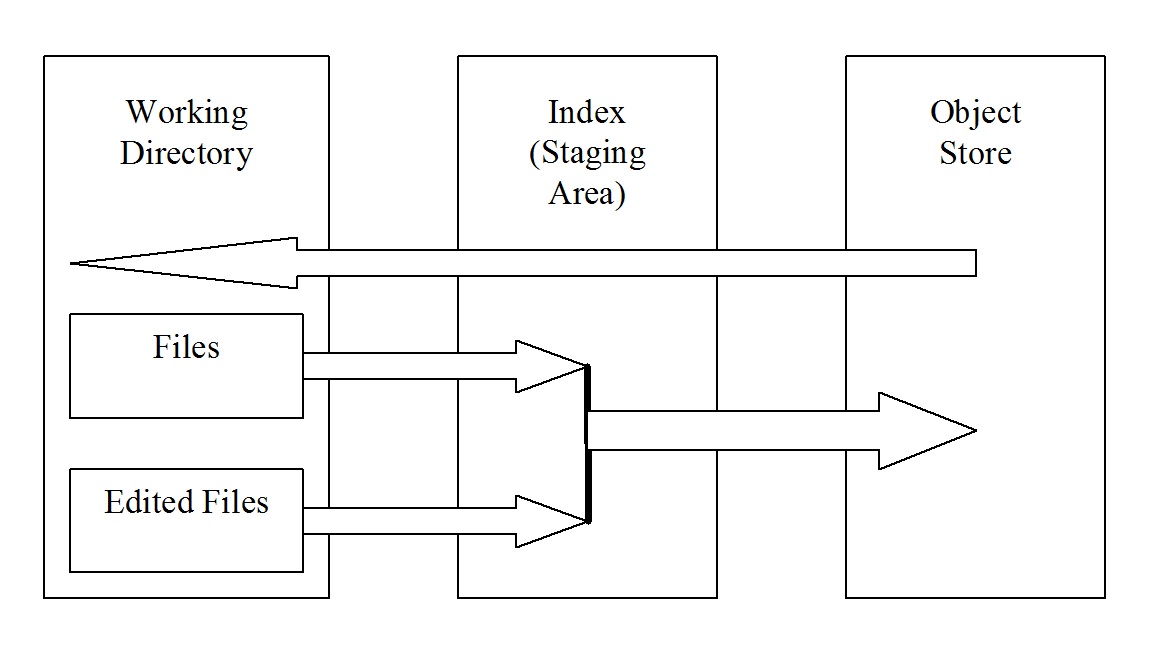


Figure W24.8 Working directory, index (staging area), and repository

The object store, in your **.git** subdirectory of your working directory, is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository to collaborate with other members of a software development team or group. The working directory contains a single checked-out copy of one version of the project. These files are pulled out of the compressed database in the repository directory and placed on disk for you to use or modify using the git checkout command. The index is a file contained in the **.git** subdirectory of your working directory that stores information about what will go into your next commit.

The basic Git workflow is as follows:

1. You place new files into, delete files from, or modify files in your working directory.

2. You stage the files, adding snapshots of them to your index.

3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your object store.

If a particular version of a file is in the object store, it is considered committed. If it is modified but has been added to the index, it is staged. And if it was changed since it was checked out but has not been staged, it is modified.

*W24.5.7.4 Directed Acyclic Graphs*

In order to plan or visualize the history of a repository structure, a directed acyclic graph (DAG), or *commit graph*, can be used. The name of the graph is derived from the fact that the flow of commits happens along the arrows of the graph (directed), and there is no way you can form a closed circle of commits by following the arrows (it is acyclic). We show an example in Figure W24.9, and will employ this graphic aid to help you visualize the state and the history of the kinds of commits we show.

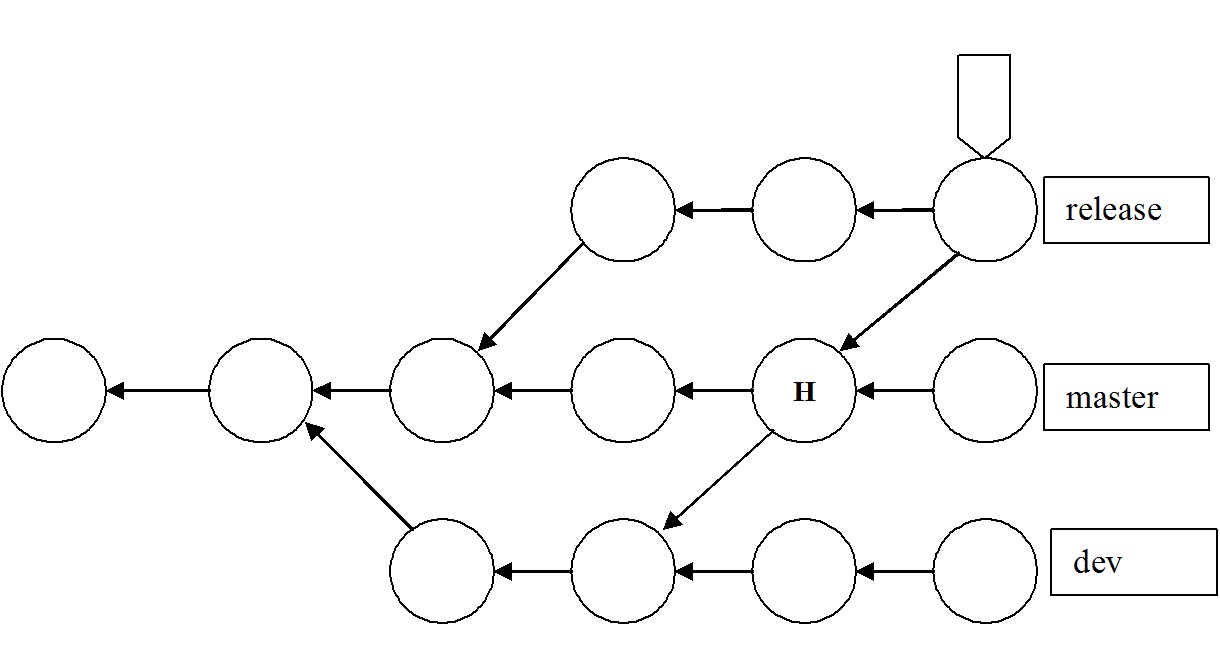


Figure W24.9 Example DAG

In Figure W24.9, the circles represent commits, and arrows point from a commit to its parent(s). Time flows from left to right in a DAG, although there is no precise correlation in terms of a time or date stamp on any of the commits. There is just the implication that commits to the left happened earlier than commits to the right in the graph. For most people, this is counterintuitive; usually we see the arrow pointing from something to its successor. In the DAG, arrows point backward toward a parent from a child. The first commit has no parents and is called a *root commit*; it was the initial commit in this repository’s history. Most commits have a single parent, indicating that they evolved in a linear way from a single previous state of the project, usually incorporating a set of related changes or edits. A commit that has multiple parents is called a *merge commit*. This indicates that the commit incorporates the changes made on one branch of the commit graph into a commit on another branch of the graph.

There are two other important features of a DAG shown in Figure W24.9. The last commit on the “release” branch has a tag at the top of it, which could contain a descriptive abbreviation of the name of that commit—perhaps “Version 1.0,” denoting that this is the first release of the software project. Also, the letter “H” represents the position of HEAD, or the currently checked-out commit on the master branch. We will omit the arrowheads in such diagrams from now on.

The labels on the right side of this picture—**release**, **master**, and **dev**—are the named branches. The branch name refers to the latest commit on that branch. Such a commit is called the *tip of the branch*. The branch itself is defined as the collection of all commits in the graph that are reachable from the tip by following the parent arrows backward along the history to the initial commit.

*W24.5.7.5 Contents of the Object Store*

Now that we have illustrated how the working directory is structured, how commits are staged, and the general layout of a repository as shown in a DAG, we can now take a finer-grained look at the objects contained in the object store. Figure W24.10 shows the four object types that are found in the object store and the relationships between those objects.

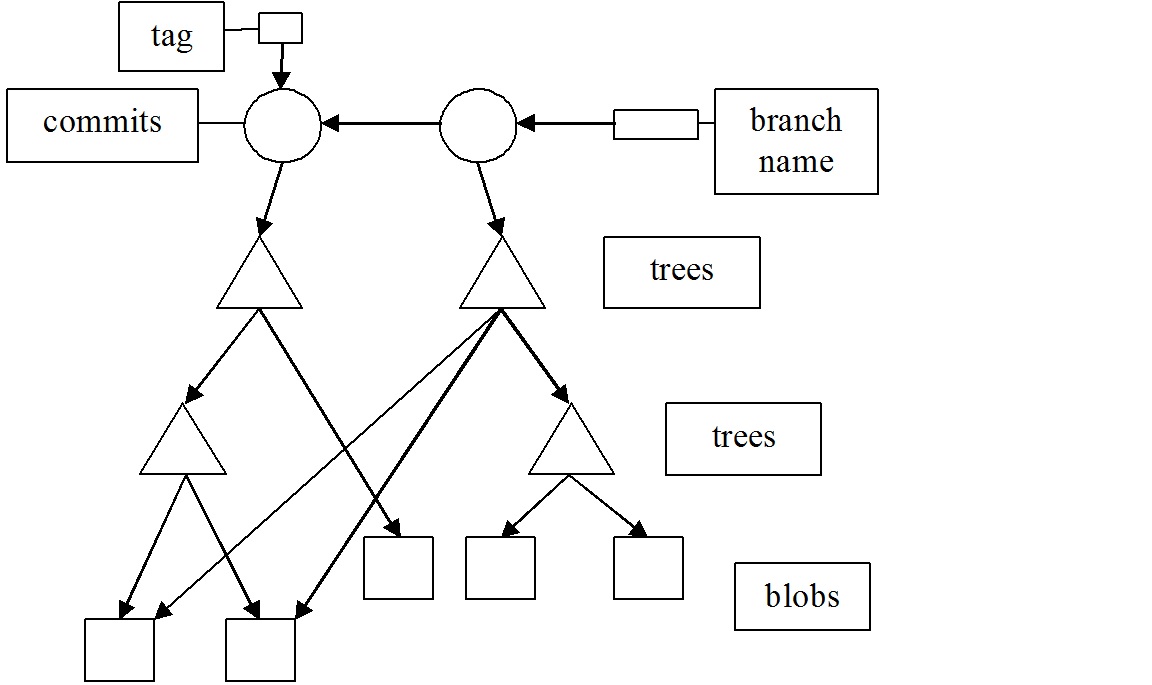


Figure W24.10 Contents of the object store

Remember from the DAG shown in Figure W24.9 that time flows from left to right and an arrow points from a child to a parent. Starting at the top, we see a rectangle to the right representing the branch name, and another smaller square representing a tag object. The name by default is **master**, but can be assigned text that is more meaningful. The tag is a shorthand label that might represent the initial release number or version of the software code. The circles represent commits. So this diagram shows two commits. The triangles represent tree objects, which can be thought of as directory or linking information between commits and blobs. Finally, at the bottom are a number of blobs, shown linked to the trees that point to them.

In the next section, we discuss a few examples involving Git and GitHub. Before discussing these examples, we give a brief description of the git command.

|  |
| --- |
| SYNTAX |
| git [options] [option arguments] <command> [<command arguments>] |
| **Purpose:** A distributed version control system with a rich command set that tracks the content of changes in source code files  **Commonly used commands and command arguments:**  **add <file>** Stage a file to the index  **commit** Take a snapshot of the working tree  **config --global username "<name>"** Configure a repository global username  **log --oneline** Display a short log of commits on the current branch  **init** Initialize a repository in the current directory  **status** Display the status of a repository |

*W24.5.7.6 Examples of Using Git and GitHub*

We show a few examples describing how to make use of Git and GitHub. Each example consists of the following:

1. Topic covered in the example

2. Objectives of the example

3. Introductory material for understanding the example

4. Git commands used in the example

5. Prerequisites for carrying out the example

6. Detailed procedure

7. Conclusions

**Example W24.1: A Short Introduction to Git**

*Objectives*: To briefly illustrate the Git staging model, and how to see differences between the various states of the parts of the repository.

*Introduction*: As shown previously, the workflow in Git basically follows a pattern of *add*, *edit*, *modify*, *stage*, and *commit*. This is the *staging model*. In this example, we show the essential Git commands that allow you to do that, repetitively if necessary. As you do more commits you add, in a linear fashion, more nodes on the branch named **master** *downstream*, after you create the first node in this example. Earlier commits are called *upstream commits*. The history of the repository flows downstream from the initial commit to the latest commit. Similar to the Linux diffcommand, the four basic forms of the git diff command allow you to examine and compare the files and directories present during different states of the repository.

|  |  |
| --- | --- |
| **Command** | **Description** |
| git config --global user.name "<name>" | Sets the author of commits in this repository |
| git config --global user. Email <email\_address> | Sets the email address of the author of commits in this repository |
| git init | Creates the **.git** directory in the working directory, initializing the data structures and objects necessary for a repository to exist |
| git status | Reports the on the differences between files in the working directory and the index, and what files are untracked. |
| git add <file> | Stages a file to the index |
| git commit | Takes a snapshot of the index, both files and directories |
| git diff | Shows the difference between two project states, in this form meaning your working directory and the index |
| git diff <commit\_identifier> | Shows the differences between your working tree and a specifically identified commit |
| git diff –cached <commit\_identfier> | Shows the differences between staged files in the index and a specifically identified commit |
| git diff <commit\_id\_1> <commit\_id\_2> | Shows the difference between two project states, in this form between commits **commit\_id\_1** and **commit\_id\_2** |
| git log –oneline | Shows history of commits in an abbreviated format |

Table W24.1 Git Commands Referenced

*Git commands referenced*: Table W24.1 shows the Git commands, and a brief description of each, that are used in this example. It is arranged in the order presented. Any argument enclosed in < > is a string of text. In order to get a more complete description of all the commands in the table, you can look at the man page for a particular command. For example, man git-status gives you a complete man page for the git status command.

*Prerequisites:* The following are the prerequisites for carrying out this example:

1. Having Git installed on your Linux system

2. Reading through and doing the in-chapter exercises shown in the previous subsections

*Procedure*: Do the following steps, in order, to meet the objectives of this example.

1. Create a working directory within which your Git repository will exist, and make that the current working directory.

% **mkdir short-git**

% **cd short-git**

%

2. Do an initial configuration of Git, in our case for the user “bob”.

% **git config --global user.name bob**

% **git config --global bob.email “bob’s\_email\_address”**

%

The last two lines above assume that you want to have a user name of “bob” (as shown in the first line), and will use the actual email address of bob in the second line.

If you give the **git config –global --edit** command to check this, you will see the following as output-

% **git config --global --edit**

[user]

name = bob

[bob]

email = bob’s\_email\_address

Make sure you quit the default editor without changing anything!

3. Initialize a repository in the current working directory.

% **git init**

%

This command initializes an empty Git repository in **/usr/home/bob/short-git/.git/**.

4. Create and save a short C program, named **hello.c**, in the current working directory, as shown:

% **cat hello.c**

#include <stdio.h>

int main()

{

printf("%s\n , "Hello Linus");

return 0;

}

%

5. Compile **hello.c**, and list the files in the current working directory.

% **gcc47 hello.c**

% **ls**

a.out hello.c

%

6. Use the git status command to examine the status of the repository at this point. It will show you that you are on the branch named **master**, you can do your initial commit, the untracked files are **a.out** and **hello.c**, and you can stage those files by using git add.

% **git status**

On branch master

Initial commit

Untracked files:

(use "git add <file>..." to include in what will be committed)

a.out

hello.c

nothing added to commit but untracked files present (use "git add" to track)

%

7. Stage only the source code file **hello.c** with the git add command.

% **git add hello.c**

%

8. Now do your initial commit.

% **git commit**

%

Use the default text editor to put the message Added the Hello Linus program on the first line of the file that appears, and then save and quit that file.

/usr/home/bob/short-git/.git/COMMIT\_EDITMSG: 13 lines, 285 characters.

[master (root-commit) 0527389] Added the Hello Linus program

1 file changed, 6 insertions(+)

create mode 100644 hello.c

9. Look at the status of the repository.

% **git status**

On branch master

Untracked files:

(use "git add <file>..." to include in what will be committed)

a.out

nothing added to commit but untracked files present (use "git add" to track)

%

10. Now lets make a change in the file **hello.c**, and track the changes with some of the forms of the git diff command. First, edit the file with your favorite text editor, and on the fourth line, add Torvalds after the word Linus, as shown in the following command output:

% **nl hello.c**

1 #include <stdio.h>

2 int main ()

3 {

4 printf("%s\n", "Hello Linus **Torvalds**");

5 return 0;

6 }

%

11. Examine the status of the repository. The output shows that the file has been changed since the last commit, but has not been staged for a new commit yet.

% **git status**

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: hello.c

Untracked files:

(use "git add <file>..." to include in what will be committed)

a.out

no changes added to commit (use "git add" and/or "git commit -a")

%

12. Now we use git diff to see the difference between what is in the working directory and the index.

% **git diff**

diff --git a/hello.c b/hello.c

index f907257..621aad0 100644

--- a/hello.c

+++ b/hello.c

@@ -1,6 +1,6 @@

#include <stdio.h>

int main ()

{

- printf("%s\n", "Hello Linus");

+ printf("%s\n", "Hello Linus Torvalds");

return 0;

}

%

13. Now stage the file **hello.c** in preparation for a new commit.

% **git add hello.c**

%

14. If you execute git diff, since there are no differences between what is in the working directory and the index, you get no output.

% **git diff**

%

15. Now use another form of the command: git diff –cached commit, to see the differences between the files staged in the index and any given commit. If you omit the commit argument, HEAD is used as the default commit. Remember from the glossary that HEAD is a reference to the current commit. The output should be exactly the same as from step 12. That’s because what is in the working directory is the current commit as seen in the object store: HEAD, the current last commit.

% **git diff --cached**

diff --git a/hello.c b/hello.c

index f907257..621aad0 100644

--- a/hello.c

+++ b/hello.c

@@ -1,6 +1,6 @@

#include <stdio.h>

int main ()

{

- printf("%s\n", "Hello Linus");

+ printf("%s\n", "Hello Linus Torvalds");

return 0;

}

%

16. Now commit the changes using the -m option of git commit, which allows you to add a message line right on the command line instead of in an editor.

% **git commit -m "Added Torvalds to Linus"**

[master 37664b9] Added Torvalds to Linus

1 file changed, 1 insertion(+), 1 deletion(-)

%

W24. Examine the history of commits with the git log command in an abbreviated format with the–oneline option.

% **git log --oneline**

37664b9 Added Torvalds to Linus

0527389 Added the Hello Linus program

%

18. See the differences between the two commits we have done so far by using git diff commit1 commit2, where **commit1** can be referred to as **0527389**, and **commit2** can be referred to as **37664b9**. These references are seen in the git log –oneline command output in step W24.

% **git diff 0527389 37664b9**

diff --git a/hello.c b/hello.c

index f907257..621aad0 100644

--- a/hello.c

+++ b/hello.c

@@ -1,6 +1,6 @@

#include <stdio.h>

int main ()

{

- printf("%s\n", "Hello Linus");

+ printf("%s\n", "Hello Linus Torvalds");

return 0;

}

%

19. Examine with git diff the differences in only commit **0527389**.

% **git diff 0527389**

diff --git a/hello.c b/hello.c

index f907257..621aad0 100644

--- a/hello.c

+++ b/hello.c

@@ -1,6 +1,6 @@

#include <stdio.h>

int main ()

{

- printf("%s\n", "Hello Linus");

+ printf("%s\n", "Hello Linus Torvalds");

return 0;

}

%

20. Repeat steps 4–19 as many times as you want to, each time creating or modifying new or existing files in the working directory. Also, it would be helpful to repeat this entire example several times, in several new directories with newly created repositories to gain practice. Each time, you stage the files with git add and then commit the additions or modifications with git commit. Then examine the differences as shown with git diff and its four variations. As you do more commits you are adding, in a linear fashion, more and more nodes downstream on the branch named **master**.

*Conclusions*: This short example illustrated the staging model in Git. It introduced a small set of Git commands that allowed you to implement the model, once or repetitively, and see the differences between commits.

In-Chapter Exercise W24.12

The git diff command is similar to what Linux command?

**Example W24.2: Creating, Editing, and Branching a Git Repository**

*Objectives*: To introduce the Git commands that create, edit, and allow you to develop a C source code project along different branches in a Git repository. To show how different branches may be merged.

*Introduction*: In order to appreciate and utilize the Git concepts shown in the previous section, we present a complete Git example of repository creation, editing, branching, and merging. Maintaining source code files and their history is the primary objective of Git. In the following example we create C source code files as needed with a text editor in the directory that has the Git repository in it. This method of introducing the files into the working directory does not preclude placing those files in that directory by any other viable means, for example by copying them from another directory or file system. We then edit those files to change their content and commit those changes. Finally, we show how to create branches along which different lines of development of the source code can proceed, and how to merge different branches. We try to emphasize the staging model, or the *edit-stage-commit* workflow model, as detailed in the previous section, throughout this example. We have purposefully not done commits and merges of branches that would produce merge conflicts. The mechanisms and strategies for resolving content conflicts are more usefully covered in other Git reference sources beyond the scope of this section, just as the mechanisms and strategies for resolving context conflicts are.

|  |  |
| --- | --- |
| git init | Create a Git repository in the current directory |
| git status | View the status of each file in a repository |
| git add <file> | Stage a file for the next commit |
| git commit | Commit the staged files with a descriptive message |
| git log | View a repository’s commit history |
| git config --global user.name "<name>" | Define the author name to be used in all repositories |
| git config --global user. Email <email> | Define the author email to be used in all repositories |
| git checkout <commit-id> | Move a previous commit into the working directory |
| git tag -a <tag-name> -m "<description>" | Create an annotated tag pointing to the most recent commit |
| git revert <commit-id> | Undo the specified commit by applying a new commit |
| git reset –hard | Reset tracked files to match the most recent commit |
| git clean –f | Remove untracked files |
| git reset --hard / git clean –f | Permanently undo uncommitted changes |
| git branch | List all branches |
| git branch <branch-name> | Create a new branch using the working directory as its base |
| git checkout <branch-name> | Make the working directory and HEAD match the specified branch |
| git merge <branch-name> | Merge a branch into the checked-out branch |
| git branch -d <branch-name> | Delete a branch |
| git rm <file> | Remove a file from the working directory (if applicable) and stop tracking that file |

Table W24.2 Git Commands Referenced

*Git Commands Referenced*: Table W24.2 shows the Git commands, and a brief description of each, that are used in this example. It is arranged in the order presented. Any argument enclosed in < > is a string of text. In order to get a more complete description of all the commands in the table, you can look at the man page for a particular command. For example, man git-status gives you a complete man page for the git status command.

*Prerequisites*: The following are the prerequisites for carrying out this example:

1. A recent version of Git available on your system, executable by an ordinary user from the command line. In Linux, Git is preinstalled. For , see the addendum to this example showing how to install Git.

2. Being able to use a text editor, such as vi, vim, or emacs, to create C program source code files.

3. Having reviewed and done the in-chapter exercises in Section W24.1 on Git concepts. This not only gives you a conceptual, top-down view of Git, but also shows you how to obtain Git help and use the man pages on the system for Git commands.

4. Completion of Example W24.1.

*Procedure*: Do the steps shown, in the order presented, to meet the objectives of this example. This is a long and detailed example. If you make mistakes, which for a beginner not familiar with the commands are irrevocable, simply start over again in a new directory that has another name than the one shown in step 1.

1. The first step in creating a repository to retain a history of your source code project files is to create a directory within which the repository can exist. We name this directory **first-git**. Then you can do a very elementary configuration of Git to identify yourself to the system. In our case, we do this for the user “bob”.

% **mkdir first-git**

% **cd first-git**

% **git config --global user.name bob**

% **git config --global bob.email “bob’s\_email\_address”**

%

The last two lines above assume that you want to have a user name of “bob” (as shown in the first line), and will use the actual email address of bob in the second line.

If you give the git config –global --edit command to check this, you will see the following as output-

% **git config --global --edit**

[user]

name = bob

[bob]

email = bob’s\_email\_address

Make sure you quit the default editor without changing anything!

2. Create a C source code file named **first.c** with the text editor of your choice. Save it in the current working directory, which should be **first-git**.

3. The next command initializes the repository, which enables the Git program in the current working directory. There is now a **.git** subdirectory in **first-git** that stores all the tracking data for our repository. The **.git** folder is the only difference between a Git repository and an ordinary folder, so deleting it will turn your project back into an unversioned collection of files.

% **git init**

Initialized empty Git repository in /usr/home/bob/first-git/.git/

%

4. Before we try to start creating revisions, view the status of our new repository. Execute the following command:

% **git status**

On branch master

Initial commit

Untracked files:

(use "git add <file>..." to include in what will be committed)

first.c

nothing added to commit but untracked files present (use "git add" to track)

%

This status message tells you that we are about to make our initial commit and that we have nothing to commit but *untracked files*. An untracked file is one that is not under version control. Git doesn’t automatically track files because there are often project files that we don’t want to keep under revision control. These might be binaries created by a C program, compiled Python modules (**.pyc** files), or any other unnecessary files. To keep a project small and efficient, you should only track source files and omit anything that can be generated from those files. This latter content is part of the build process, not revision control.

5. The next step stages the file **first.c** in preparation for doing the first commit.

% **git add first.c**

%

We added **first.c** to the snapshot of the index for the next commit. Git’s term for creating a snapshot is called *staging* because we can add or remove multiple files before actually committing it to the project history. The index holds a snapshot of the content of the working tree, and it is this snapshot that is taken as the contents of the next commit. Thus, after making any changes to the working directory and before running the commit command, you must use the add command to add any new or modified files to the index.

6. Now we examine the repository status with the git status command.

% **git status**

On branch master

Initial commit

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: first.c

%

Now, instead of **first.c** being an untracked file, it is shown as being staged to be committed.

7. We are ready to commit.

% **git commit**

%

The first part of committing is to use the default text editor you are put into by Git to add The initial commit as the first line in that file. Then save and quit that file.

/usr/home/bob/first-git/.git/COMMIT\_EDITMSG: 10 lines, 245 characters.

[master (root-commit) 74088f6] The initial commit

1 file changed, 1 insertion(+)

create mode 100644 first.c

8. We need a new command, git log, to view the project revision history. When you execute this command, Git will output information about our first commit:

% **git log**

commit 74088f645993f3df16f27565628ea38c271357e0

Author: bob <your\_email\_address>

Date: Mon Nov 10 18:59:44 2014 -0800

The initial commit

%

9. We continue to add new C source code files to our working directory. Create two C source code files named **second.c** and **third.c** with the text editor of your choice. Save them in the current working directory, which should be **first-git**.

10. We now need to stage those two new files, in preparation for committing them to our repository.

% **git add second.c third.c**

%

11. Take a look at the status of the repository.

% **git status**

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: second.c

new file: third.c

%

12. Take a look at the history of the repository.

% **git log**

commit 74088f645993f3df16f27565628ea38c271357e0

Author: bob <your\_email\_address>

Date: Mon Nov 10 18:59:44 2014 -0800

The initial commit

%

13. Commit the two new files.

% **git commit**

%

Use the text editor that automatically launches to add second.c and third.c added as the first line in the file. Then save the file and quit the text editor.

/usr/home/bob/first-git/.git/COMMIT\_EDITMSG: 8 lines, 255 characters.

[master 4ea0534] second.c and third.c added

2 files changed, 2 insertions(+)

create mode 100644 second.c

create mode 100644 third.c

14. The git add command is used to stage new files. It can also be used to stage modified files. In order to demonstrate this, use a text editor to modify the previously created C source code files **first.c**, **second.c**, and **third.c**.

15. Then take a look at the status of the repository. Git lists the tracked files as being modified.

% **git status**

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: first.c

modified: second.c

modified: third.c

no changes added to commit (use "git add" and/or "git commit -a")

%

16. Stage those modified files.

% **git add first.c second.c third.c**

%

17. Now commit the modified, staged files. The -m option of commit lets you specify a commit message on the command line instead of opening a text editor. This is a shortcut method; it has the exact same effect as our previous commits.

% **git commit -m "Revised all three files"**

[master b0a6b40] Revised all three files

3 files changed, 3 insertions(+)

%

18. Our history can now be shown as follows:

% **git log --oneline**

b0a6b40 Revised all three files

4ea0534 second.c and third.c added

74088f6 The initial commit

%

The git log command comes with formatting options. For now we show the --oneline flag, as in git log –oneline.

19. Condensing output to a single line is one way to get an overview of a repository. Another useful configuration is to pass a filename to git log:

% **git log --oneline second.c**

b0a6b40 Revised all three files

4ea0534 second.c and third.c added

%

20. So far, we have recorded versions of a project into a Git repository. Maintaining these copies provides us with backups. More importantly, we can have independent versions of the state of the project that can be used for the purposes of creating multiple lines or tracks of development. Our next objective is to be able to view the previous states of a project, revert back to them, and reset uncommitted changes if necessary. First, let’s return to the state of the repository at the commit **4ea0534 second.c and third.c added**. The HEAD is now at **4ea0534**. The git checkout command positions HEAD at any commit we desire, going all the way back to the initial commit.

% **git checkout 4ea0534**

Note: checking out '4ea0534'.

%

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

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

% **git checkout -b new\_branch\_name**

HEAD is now at 4ea0534... second.c and third.c added

%

21. Let’s go back to the initial commit.

% **git checkout 74088f6**

Previous HEAD position was 4ea0534... second.c and third.c added

HEAD is now at 74088f6... The initial commit

%

22. We can check the status of the repository at this point.

% **git status**

HEAD detached at 74088f6

nothing to commit, working directory clean

%

23. All previous steps worked on the master branch, where our second and third commits reside. To retrieve our complete history, we just have to check out this entire branch. This is a very brief introduction to branches, but it’s all we need to know to navigate between commits. The following command makes Git update our working directory to reflect the state of the master branch’s snapshot. It recreates the **second.c** and **third.c** files for us, and the content of **first.c** is updated as well. We’re now back to the current state of the entire commits history of the project.

% **git checkout master**

Previous HEAD position was 74088f6... The initial commit

Switched to branch 'master'

%

24. Tags are references to milestones, or releases in a software project. They let developers easily browse and check out important revisions. For example, we can now use the **v1.0** tag to refer to the third commit instead of its random ID. To view a list of existing tags, execute git tag without any arguments. So, we can label this a stable version of the C program modules. The -a option tells Git to create an annotated tag, which lets us record our name, the date, and a descriptive message, specified via the -m option. We can finalize it by tagging the most recent commit with a version number as follows:

% **git tag -a v1.0 -m "Stable version of the software"**

%

25. Now we can add C modules to the working directory that allow us to experiment, without committing those modules. Use the text editor of your choice to create an experimental C source code file named **experiment.c**, and save it in the current working directory.

26. Then stage that file.

% **git add experiment.c**

%

27. Check on the status of the repository.

% **git status**

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: experiment.c

%

28. Commit that file.

% **git commit -m "Add an experimental C program"**

[master 1707e3e] Add an experimental C program

1 file changed, 1 insertion(+)

create mode 100644 experiment.c

%

29. See a history of commits to the repository.

% **git log**

commit 1707e3e6f3e12ef1eaa20417dd601bb11180d731

Author: bob <your\_email\_address>

Date: Mon Nov 10 19:32:31 2014 -0800

Add an experimental C program

commit b0a6b40c4ab515fb10c036b0d7d2ebcacd281865

Author: bob <your\_email\_address>

Date: Mon Nov 10 19:16:05 2014 -0800

Revised all three files

commit 4ea0534d1e88d297d38a0c63e31dab8b6da7c5ca

Author: bob <your\_email\_address>

Date: Mon Nov 10 19:08:00 2014 -0800

second.c and third.c added

commit 74088f645993f3df16f27565628ea38c271357e0

Author: bob <your\_email\_address>

Date: Mon Nov 10 18:59:44 2014 -0800

The initial commit

%

30. Let’s go back to our stable revision. Remember that the **v1.0** tag is now a shortcut to the third commit’s ID.

% **git checkout v1.0**

Note: checking out 'v1.0'.

%

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

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

% **git checkout -b new\_branch\_name**

HEAD is now at b0a6b40... Revised all three files

%

31. After seeing the stable version of the site, we decide to scrap the C code experiment we started in step 25. But, before we undo the changes to the repository, we need to return to the **master** branch. If we didn’t, all of our updates would be on some nonexistent branch. You should never make changes directly to a previous revision.

% **git checkout master**

Previous HEAD position was b0a6b40... Revised all three files

Switched to branch 'master'

%

32. Now examine the history of our repository with the git log command. This gives us the shorthand name of the last commit we executed entitled Add an experimental C program.

% **git log --oneline**

1707e3e Add an experimental C program

b0a6b40 Revised all three files

4ea0534 second.c and third.c added

74088f6 The initial commit

%

33. Now we want to restore our stable release by removing the most recent commit. Make sure to change **1707e3e** to the ID supplied by your system’s Git for the experimental C code commit before running the next command. Also, the command we use, git revert, undoes the commit we specify as its argument.

% **git revert 1707e3e**

%

You are put into the default text editor, which allows you to change the title of the reverted commit. Leave the commit title the same, save the file and quit the editor.

34. Look at what files are in the working directory, and see a history of your commits.

% **ls**

first.c second.c third.c

% **git log --oneline**

03ece49 Revert "Add an experimental C program"

1707e3e Add an experimental C program

b0a6b40 Revised all three files

4ea0534 second.c and third.c added

74088f6 The initial commit

%

Notice that instead of deleting the Add an experimental C program commit, Git undoes the changes it contains, then adds on another commit showing the reversion. So, our fifth commit and our third commit represent the exact same snapshot, as follows. Again, Git is designed to never lose history: the fourth snapshot is still accessible, just in case we want to continue developing it.

35. Now we can try to add a file that we definitely will want to get rid of completely. Use your text editor to create a file named **dumbc.c**, and then edit **first.c** to make a small change in it. Now look at the status of the repository.

% **git status**

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: first.c

Untracked files:

(use "git add <file>..." to include in what will be committed)

dumbc.c

no changes added to commit (use "git add" and/or "git commit -a")

%

36. We have a tracked file and an untracked file that need to be changed. First, we’ll take care of the tracked **first.c**.

% **git reset --hard**

HEAD is now at 03ece49 Revert "Add an experimental C program"

%

This changes all tracked files to match the most recent commit. You can also pass a filename to this command to reset only that file—for example, git reset --hard first.c. The --hard flag is what actually updates the file. Running git reset first.c without any flags will simply unstage the file, leaving its contents as is. In either case, git reset only operates on the working directory and the staging area, so our git log history remains unchanged.

37. Now remove the **dumb.c** file. Of course, we could manually delete it, but using Git to reset changes eliminates human errors when working with several files in large teams. Run the following command:

% **git clean -f**

Removing dumbc.c

%

This will remove all untracked files. With **dumb.c** gone, git status should now tell us that we have a “clean” working directory, meaning our project repository matches the most recent commit. Be careful with git reset and git clean. Both operate on the working directory, not on the committed snapshots. Unlike git revert, they permanently undo changes, so make sure you really want to delete what you’re working on before you use them.

38. To begin creating and using branches, list what branches exist at this point.

% **git branch**

\* master

%

This command displays the only current branch, named **\* master**. The **master** branch is Git’s default branch, and the asterisk next to it means that it is currently checked out. This means that the most recent snapshot in the master branch resides in the working directory. There is only one working directory for each project, only one branch can be checked out at a time.

39. Look at some previous commits before we begin creating a new branch. First get a shorthand list of the repository commit history.

% **git log --oneline**

03ece49 Revert "Add an experimental C program"

1707e3e Add an experimental C program

b0a6b40 Revised all three files

4ea0534 second.c and third.c added

74088f6 The initial commit

%

40. Next, checkout the **Add an experimental C program** commit.

% **git checkout 1707e3e**

Note: checking out '1707e3e'.

%

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

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

% **git checkout -b new\_branch\_name**

HEAD is now at 1707e3e... Add an experimental C program

%

The HEAD normally is on the tip of a development branch, meaning you are on that branch. But when we checked out the previous commit, the HEAD moved to the middle of the branch. We are no longer on the master branch since it contains more recent snapshots than the HEAD. This is reflected in the Git branch output from the previous command, which tells us that we’re currently not on a branch.

41. We can now create a branch from this commit. Name it **test**.

% **git branch test**

%

42. To be able to add commits to the new branch, move onto that branch by checking it out.

% **git checkout test**

Switched to branch 'test'

%

43. Use your favorite text editor to make minor changes to the file **experiment.c**, so that we can begin development along this branch. Be sure to save the modified **experiment.c**.

44. Now stage the modified **experiment.c**.

% **git add experiment.c**

% **git status**

On branch test

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: experiment.c

%

45. The following commit will create a fork in our project repository, as shown in Figure W24.11.

% **git commit -m "Modified experiment.c"**

[test 1ff0291] Modified experiment.c

1 file changed, 1 insertion(+)

%

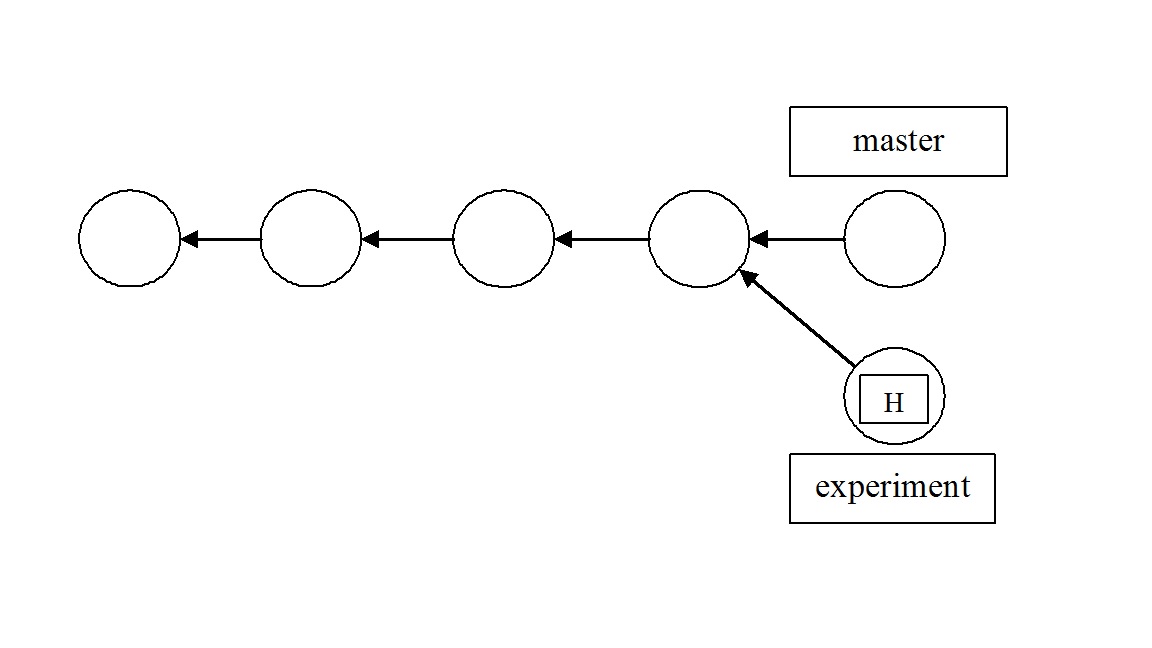


Figure W24.11 Forked project repository

46. Take a look at the history of commits in abbreviated form.

% **git log --oneline**

1ff0291 Modified experiment.c

1707e3e Add an experimental C program

b0a6b40 Revised all three files

4ea0534 second.c and third.c added

74088f6 The initial commit

%

The history before the fork is considered part of the new branch. So, since we are on the branch test, the test history spans all the way back to the first commit. The project repository has a complex history, but each individual branch still has a linear history. Snapshots and commits occur one after another in a linear fashion. This means that we can work within branches in the same way we did in steps 1–37.

47. Let’s add one more snapshot to the test branch. Rename **experiment.c** to **experiment2.c**, then use the following Git commands to update the repository.

% **mv experiment.c experiment2.c**

% **git status**

On branch test

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

deleted: experiment.c

Untracked files:

(use "git add <file>..." to include in what will be committed)

experiment2.c

no changes added to commit (use "git add" and/or "git commit -a")

% **git rm experiment.c**

rm 'experiment.c'

% **git status**

On branch test

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

deleted: experiment.c

Untracked files:

(use "git add <file>..." to include in what will be committed)

experiment2.c

% **git add experiment2.c**

% **git status**

On branch test

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

renamed: experiment.c -> experiment2.c

%

The git rm command tells Git to stop tracking **experiment.c** (and delete it if necessary), and git add starts tracking **experiment2.c**. The renamed: experiment.c => experiment2.c message in the final status output shows us that Git knows when we are just renaming a file. You could have just made editing changes to **experiment.c** to justify moving the branch forward with another commit. Our snapshot is staged and ready to be committed.

48. We now do a new commit along the new branch.

% **git commit -m "Renamed experiment.c to experiment2.c"**

[test 27d1e0d] Renamed experiment.c to experiment2.c

1 file changed, 0 insertions(+), 0 deletions(-)

renamed experiment.c => experiment2.c (100%)

%

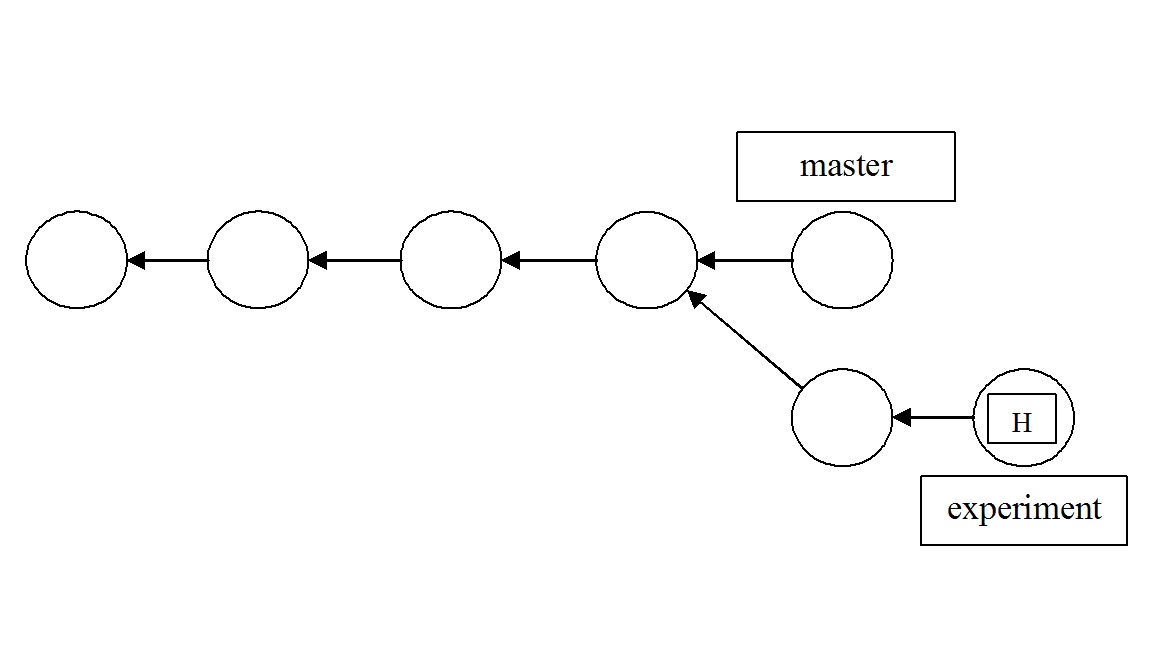


Figure W24.12 DAG of project repository

49. Look at the history of commits along this branch. Our project repository now looks as shown in Figure W24.12.

% **git log --oneline**

27d1e0d Renamed experiment.c to experiment2.c

1ff0291 Modified experiment.c

1707e3e Add an experimental C program

b0a6b40 Revised all three files

4ea0534 second.c and third.c added

74088f6 The initial commit

%

50. Now we will fork another branch off the master branch. In preparation for doing this, return the HEAD to the **master** branch by using the git checkout command.

% **git checkout master**

Switched to branch 'master'

%

51. Now that you are back on the **master** branch, list the branches in this repository.

% **git branch**

\* master

test

% **git log --oneline**

03ece49 Revert "Add an experimental C program"

1707e3e Add an experimental C program

b0a6b40 Revised all three files

4ea0534 second.c and third.c added

74088f6 The initial commit

%

52. We will now create a new branch, forked off the **master** branch and named **modules**.

% **git branch modules**

%

53. Make the modules branch the current branch.

% **git checkout modules**

Switched to branch 'modules'

%

54. In your favorite text editor, create a C program file called **module1.c**. Then stage **module1.c**.

% **git add module1.c**

%

55. Check the status of the repository.

% **git status**

On branch modules

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: module1.c

%

56. Now commit **module1.c**.

% **git commit -m "Added module1.c"**

[modules 9b37c23] Added module1.c

1 file changed, 1 insertion(+)

create mode 100644 module1.c

%

57. In your favorite text editor, add a reference to the C code in **module1.c** in the files **first.c**, **second.c**, and **third.c**.

58. Stage the changes you made in **first.c**, **second.c**, and **third.c**.

% **git add first.c second.c third.c**

%

59. Check the status of the project repository.

% **git status**

On branch modules

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: first.c

modified: second.c

modified: third.c

%

60. Commit the changes to the files **first.c**, **second.c**, and **third.c**.

% **git commit -m "Add references to module1.c"**

[modules 436a966] Add references to module1.c

3 files changed, 3 insertions(+)

%

61. Now examine the history of commits along this branch.

% **git log --oneline**

436a966 Add references to module1.c

9b37c23 Added module1.c

03ece49 Revert "Add an experimental C program"

1707e3e Add an experimental C program

b0a6b40 Revised all three files

4ea0534 second.c and third.c added

74088f6 The initial commit

%

A DAG of your project repository at this point is shown in Figure W24.13.

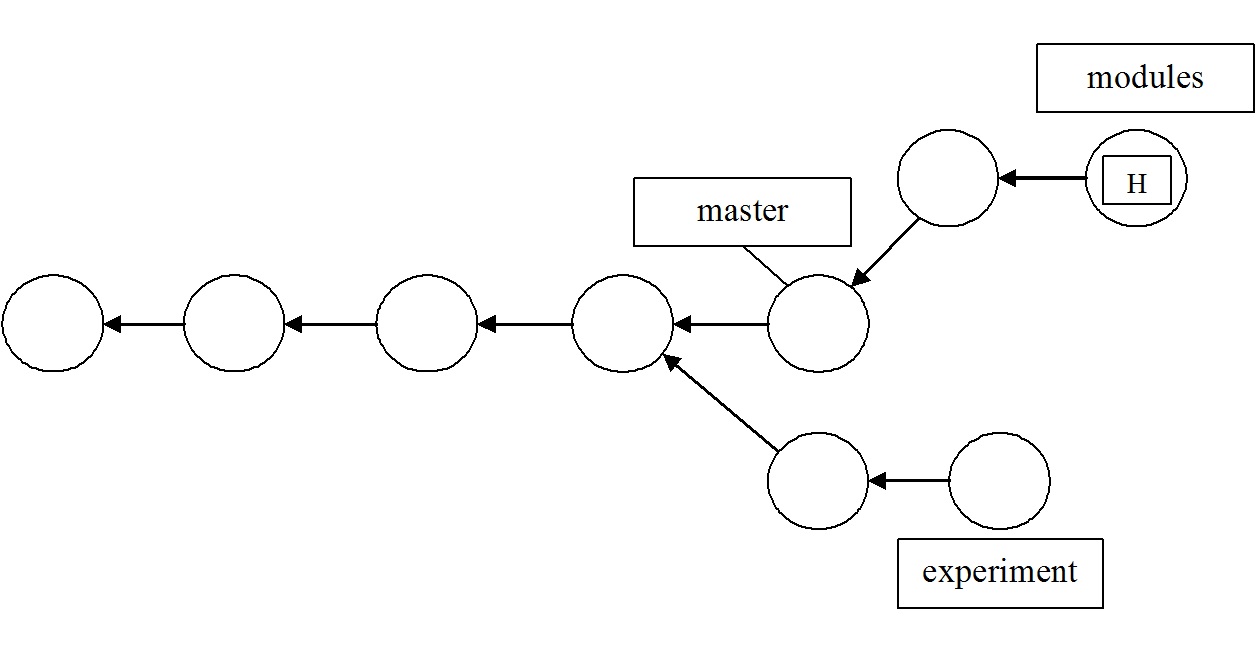


Figure W24.13 DAG of repository with three branches

62. In preparation for merging the **modules** branch with the **master** branch, do the following. First, switch to the **master** branch, then check the files in the working directory. Finally look at the commit history of the repository as seen along the **master** branch.

% **git checkout master**

Switched to branch 'master'

% **ls**

first.c second.c third.c

% **git log --oneline**

03ece49 Revert "Add an experimental C program"

1707e3e Add an experimental C program

b0a6b40 Revised all three files

4ea0534 second.c and third.c added

74088f6 The initial commit

%

63. We will now merge the **modules** branch with the **master** branch. This command always merges into the current branch. The **modules** branch is unchanged. Check the history of commits withgit log --oneline to make sure that **modules**’s history of commits has been added to **master**’s history of commits. The DAG representing the final state of the repository is shown in Figure W24.14.

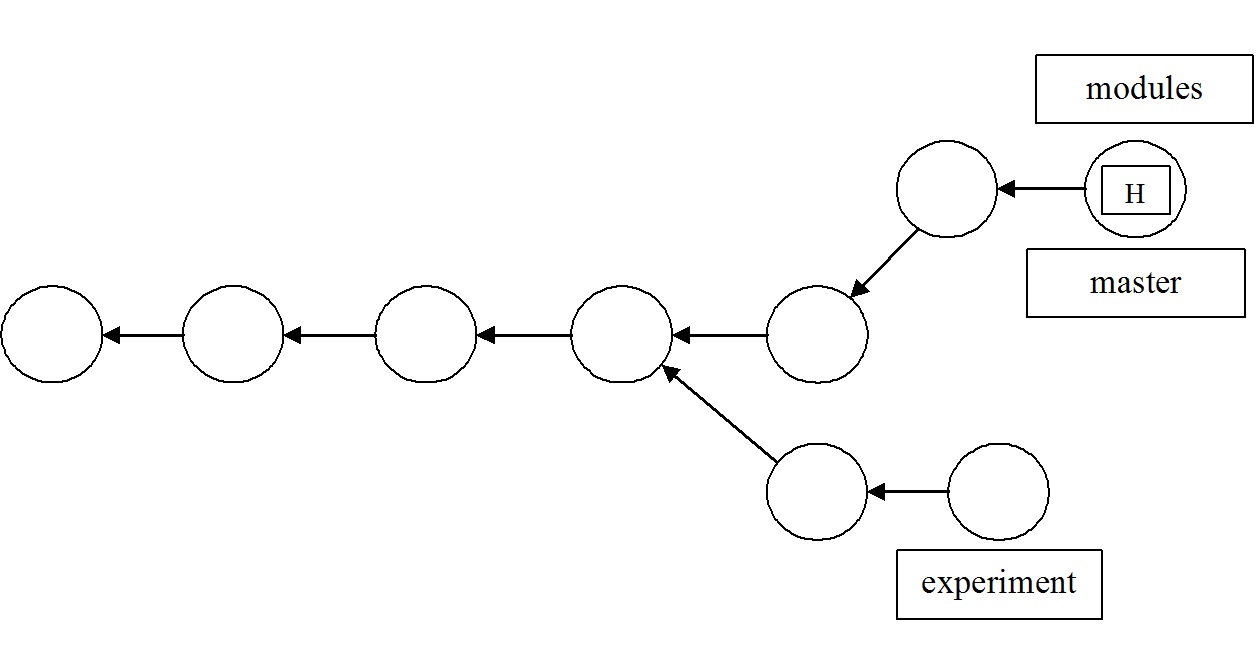


Figure W24.14 DAG of final state of project repository

% **git merge modules**

Updating 03ece49..436a966

Fast-forward

first.c | 1 +

module1.c | 1 +

second.c | 1 +

third.c | 1 +

4 files changed, 4 insertions(+)

create mode 100644 module1.c

% **git log --oneline**

436a966 Add references to module1.c

9b37c23 Added module1.c

03ece49 Revert "Add an experimental C program"

1707e3e Add an experimental C program

b0a6b40 Revised all three files

4ea0534 second.c and third.c added

74088f6 The initial commit

%

*Conclusions*: In this example, we created C source code files as needed with a text editor in the directory that has the Git repository in it. We then edited those files to change their content and committed those changes. Finally, we showed how to create branches along which different lines of development of the source code can proceed, and how to merge different branches. We emphasized the staging model, or edit-stage-commit workflow model, as detailed in the previous section, throughout this example.

In-Chapter Exercise W24.13

Under what circumstances would you want to track, or stage, other kinds of files in a Git repository related to the C program development and build process?

In-Chapter Exercise W24.14

If, after step 63, you were to create a new text file in the working directory, but not stage and commit, would that file still be in the working directory after you do a checkout of the initial commit? Why/why not?

*W24.5.7.7 GitHub as a Remote Repository*

GitHub is a popular remote repository where you can easily and securely work together with a team to do the development and maintenance of a software project. We first provide some background information on Git URLs and *refspecs*. Then, in this section’s examples, we show the basics of how to take files from a local repository and put them on a GitHub repository using the Linux command line. We also show how to take files from GitHub and retrieve them back on to a local repository. The basis and groundwork for these operations are expedited on the Linux command line with the Git commands we have illustrated thus far. Therefore, having completed the previous subsections on Git is necessary to your understanding of the GitHub interactions shown here. We also introduce new Git commands to allow you to work with a GitHub remote repository.

We do not show how to get an account on GitHub, or how to create a new repository using the web-based GUI interface of GitHub. It is assumed in all of the examples that you can do those two basic steps via a Web browser at www.github.com, and can navigate to the URL we show. The repository on your system is called the *local* or *current repository*, and a repository like GitHub is called the *remote repository*.

Use the git remote command and its options and arguments to create, remove, manipulate, and view a remote. For example, to add a remote reference specification, or refspec, to the current local repository, use the git remote addcommand with options and arguments. You can also look at what has been defined as a remote in the **.git/config** file. All the remotes you added are recorded in the **.git/config** file and can be manipulated using git config and its options and arguments.

The basic Git commands that refer to remote repositories are:

git clone Transfers a remote repository into the local repository

git fetchRetrieves objects and their related data structures from a remote repository

git pullMerges changes from a remote repository into a corresponding local branch

git pushCopies objects and their related data structures to a remote repository

git ls-remoteLists references in a given remote repository

W24.5.7.7.1 Git URLs

For the git remote command, Git names the argument forms of reference to the remote repository as *Uniform Resource Locators* (URLs). A Git URL that refers to a repository on a local file system can be:

/pathname/repo.git

file:///pathname/repo.git

The first reference form uses hard links within the Linux file system to directly share exactly the same objects between the current and remote repository. The second and preferred form copies the data instead of sharing it via links.

A Git URL that refers to a repository on a remote system can take several forms. These forms include http, https, ssh, scp, rsync, and ftp. The primary and preferred ways of designating a remote repository using http or https, and which we use in the examples, are as follows:

http://github.com/pathname/repository\_name

https://github.com/pathname/repository\_name

where **pathname** is a username on GitHub, and **repository\_name** is a specific named repository for that user. The named repository does not have to end in with a **.git** suffix. These two URL forms are most favored by GitHub.

Server firewalls usually allow the http port 80 and https port 443 to remain open, and by default on Linux, port 22 for ssh.

For a remote repository whose data must be retrieved across a wide area network, such as the Internet, you can also use the Git native protocol, which refers to the custom protocol used internally by Git to transfer data. Examples of a native protocol URL include:

git://example.com/pathname/repo.git

git://example.com/~user/pathname/repo.git

These forms are used by Git to publish repositories for anonymous read. You can both clone and fetch using these URL forms.

The Git native protocol can be tunneled over an ssh connection using the following URL specification:

ssh://[user@]github.com[:port]/pathname/repo.git

where **user@** is the client-side ssh username on the local system, **port** is the optional designation of a port on the client other than the default port 22, **pathname** is the username on GitHub, and **repo.git** is the name of the specific repository on GitHub.

Git also supports a URL form with scp-like syntax. It is identical to the ssh forms, but there is no way to specify a port parameter:

[user@]example.com:/pathname/repo.git

For a more complete list and explanation of the remote URL specifications, use the command man git-clone. Then, you should page down to the URL specifications section of the man page.

W24.5.7.7.2 Understanding Remote Pull and Push Operations

Git and GitHub workflow models, branching strategies, and particularly the resolution of merge conflicts when working with those models and strategies, can be very complex. Those models are also as varied as the different kinds of software development and maintenance teams, the size of those teams, and their respective goals. For the basic Git commands that allow you to work with remote repositories, it is helpful for a beginner to know some background material for those commands discussed at the beginning of Section W24.5.7. Since most of your workflow as a beginner involves using the git push and git pull commands, it is very helpful to know what the underlying assumptions and basis for those commands are.

After you have cloned a remote repository to a local one, the commands git pull and git push keep the two repositories synchronized as far as their content is concerned. The most important thing to remember about keeping repositories synchronized is that, with regard to content, a repository consists of two things: an object store and a set of references, or *refs*—in other words, a commit graph and a set of branch names and tags that designate commits. When you clone a repository, such as with the command git cloneURL/repository, the Git does the following things in the order shown:

 Creates a new local repository that is essentially a replica of the remote repository

 Adds a remote named **origin** to refer to the repository being cloned in **.git/config**:

[remote "origin"]

fetch = +refs/heads/\*:refs/remotes/origin/\*

url = URL/repository

The line in the **config** file with fetch in it is the refspec, an assignment statement that specifies a correspondence between sets of refs in the two repositories: the pattern on the left side of the colon names refs in the remote, associated with the pattern on the right side of the colon, which are the corresponding refs in the local repository.

 Runs the command git fetch origin, which updates our local refs for the remote’s branches (creating them in this case), and asks the remote to send any objects we need to complete the history for those refs (in the case of this new repository, all of them).

 Checks out the remote repository’s current branch (its HEAD ref), giving you a working directory, and **.git** directory in it—that is, a replicated local repository cloned from the remote repository.

So now you can execute the git show-ref command as follows and view the local repository refs:

% **git show-ref --abbrev master**

b5216a81 refs/heads/master

b5416a91 refs/remotes/origin/master

%

When you give the git pull command, Git first executes a fetch on the remote for the current branch, updating the remote’s local tracking refs and obtaining any new objects needed to complete the history of those refs—that is, all commits, tags, trees, and blobs reachable from the new branch tips. Then it tries to update the current local branch to match the corresponding branch in the remote. If only one side has added content to the branch, then this will succeed, and is called a *fast-forward update* since one ref is simply moved forward along the branch to catch up with the other.

If both sides have committed to the branch, however, then Git has to do something to incorporate both versions of the branch history into one shared version. By default, this is a merge: Git merges the remote branch into the local one, producing a new commit that refers to both sides of the history via its parent pointers. And this would most likely lead to merge conflicts.

When you give the git push command, Git updates the corresponding branch in the remote with your local repository branch contents, sending any objects the remote needs to complete the new state of the remote repository. This will fail if the update is non-fast-forward, and Git will suggest that you first git pull in order to resolve the differences between repositories.

Nothing in remote-tracking branches ties the things you do to your repository to the remote; the relationship is one way. Each remote-tracking branch is just a branch in your repository like any other branch, a ref pointing to a particular commit. They are only “remote” in the sense that they point to a remote repository. They track the state of corresponding branches in the remote, and you can update them using the command git pull.

A repository can have many remotes, set up at any time; see the git remote add command in Example W24.3. If the original repository you cloned from is no longer available, you can fix its URL by editing the **.git/config** file for a particular local repository. You can remove a remote reference entirely with git remote rm. This command will remove the remote-tracking branches for that remote repository too.

*W24.5.7.8 GitHub Examples*

The following section illustrates your basic interaction with GitHub as a remote repository, using Git commands from the Linux command line.

**Example W24.3: Basic GitHub Operations**

*Objective*: To create a new repository in your existing account at GitHub, and transmit files to the new GitHub repository from a local repository.

*Introduction*: In this example, we first create a working directory with a new Git repository in it. Then we add a file to this new local repository and use the git push command to move that file up to a repository at GitHub.

|  |  |
| --- | --- |
| **Command** | **Description** |
| git init | Add a Git repository in the current directory |
| git add <file> | Stage <file> for the next commit |
| git commit -m "Message" | Execute a commit with Messageautomatically added using the –m option. |
| git remote add origin <path> | Identify the valid <path>as a Git remote repository reference |
| git push -u origin master | Transmit the branch named **master** to the current remote repository and add upstream tracking information |
| git remote –v | List the remotes defined for this repository |

Table W24.3 Git Commands Referenced

*Git Commands Referenced*: Table W24.3 shows the Git commands, and a brief description of each, that are used in this example. It is arranged in the order presented. Any argument enclosed in < > is a string of text. In order to get a more complete description of all the commands in the table, you can look at the man page for a particular command. For example, man git-push gives you a complete man page for the git push command.

*Prerequisites*: The following are the prerequisites for carrying out this example:

1. Having an account on GitHub and knowing how to add a new repository to that account

2. Having completed the previous sub-sections that familiarize you with Git commands executed on the Linux command line

3. Having an Internet connection and a suitable Web browser installed and operating on your Linux system

4. Local and remote GitHub repositories with only one branch each

5. Having completed Example W24.2

*Procedure*: Do the following steps, in the order presented, to meet the objectives of this example.

1. Create the working directory and make it the current directory.

% **mkdir githubtest**

% **cd githubtest**

%

2. Add a new file to the directory.

% **touch README.md**

%

3. Initialize Git in this new directory.

% **git init**

Initialized empty Git repository in /usr/home/bob/githubtest/.git/

%

4. Stage the file **README.md**, and do an initial commit to the repository.

% **git add README.md**

% **git commit -m "First Commit"**

[master (root-commit) f971b1d] First Commit

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 README.md

%

5. In your Web browser, navigate to GitHub at www.github.com, log in, and create a new repository in your GitHub account. Name that repository **test**.

6. Use the git remote command to designate your GitHub repository **test** as a remote repository for the local repository we created in steps 1-4. To find the URL to designate as the GitHub repository, look in the URL bar of your browser when you are in the GitHub repository named **test** that you created in step 5.

On our system, in our browser, the URL to this new repository on GitHub is **https://github.com/bobk48/test.git**.

This will be different for you on your system, into your account at Github!

% **git remote add origin https://github.com/bobk48/test.git**

%

7. Use the git push command to take the local repository and move it up to GitHub. Supply the username and password for the GitHub repository as needed.

% **git push -u origin master**

Username for 'https://github.com': bobk48

Password for 'https://bobk48@github.com': xxx

Counting objects: 3, done.

Writing objects: 100% (3/3), 207 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/bobk48/test.git

\* [new branch] master -> master

Branch master set up to track remote branch master from origin.

%

8. Check the names of the files in the local repository.

% **ls**

README.md

%

In your browser, check the content of the repository **test**.

9. Use the git remote command to list the remote repositories for this local repository.

% **git remote -v**

origin https://github.com/bobk48/test.git (fetch)

origin https://github.com/bobk48/test.git (push)

%

What this output shows you is that you can transfer (using the git push command) new content to the remote repository, and get content from it (using the git fetch command).

10. To add a new file to the GitHub repository from the local repository, first create the file with your favorite text editor.

% **vi newfile.txt**

... Create and save a new textfile named newfile.txt ...

newfile.txt: new file: 1 lines, 47 characters.

%

11. List the files in your working directory.

% **ls**

README.md newfile.txt

%

12. Stage **newfile.txt** and commit it.

% **git add newfile.txt**

% **git commit -m "second new file added"**

[master 4dc2de7] second new file added

1 file changed, 1 insertion(+)

create mode 100644 newfile.txt

%

13. Now use git push again to push the contents of the repository to your GitHub repository.

% **git push origin master**

Username for 'https://github.com': bobk48

Password for 'https://bobk48@gmail.com@github.com': xxx

Counting objects: 4, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 325 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/bobk48/test.git

f971b1d..4dc2de7 master -> master

%

In your browser, check the content of your GitHub repository named **test**. You should see the same files there that are in your local repository along the branch **master**.

*Conclusions*: By designating a GitHub repository as a remote repository reference using an https URL, we accomplished moving files up from a local repository to a GitHub repository.

In-Chapter Exercise W24.15

What refspec URL and fetch assignments are listed for the repository **test**? What branch refspecs are listed? How did you find this out?

In-Chapter Exercise W24.16

What Git command do you use locally to put an earlier, or *upstream*, commit into the working directory?

**Example W24.4: Cloning a GitHub Repository**

*Objectives*: To clone an existing remote GitHub repository into a new local repository.

*Introduction*: In order to share the contents of an existing GitHub repository between members of a software development and maintenance team, it is a usual practice to clone, or copy, a complete repository from GitHub into a new local repository. In the previous Example W24.3, we first created a working directory and a new Git repository in it. Then we added a file to this new local repository and used the git push command to move that file up to an existing repository at GitHub. In this example, we will use the git clone command to create an entirely new local Git repository from an existing remote GitHub repository. Then we will add a new file to the local repository and use git push to transfer that file to the GitHub repository. To simplify things for the beginner, there is only one branch on the remote GitHub repository.

|  |  |
| --- | --- |
| **Command** | **Description** |
| git clone <remote\_designation> | Transfers a complete repository from the remote designated into a local repository, maintaining the branch and file structure |
| git status | Shows the current state of the repository |
| git add <object(s)> | Stages the named object(s) to the index |
| git commit –m "Message" | Commits the contents of the staged files in the index |
| git push <remote\_designation> master | Transfers the working directory to the remote designated on the branch **master** |

Table W24.4 Git Commands Referenced

*Git Commands Referenced*: Table W24.4 shows the Git commands, and a brief description of each, that are used in this example. It is arranged in the order presented. Any argument enclosed in < > is a string of text. In order to get a more complete description of all the commands in the table, you can look at the man page for a particular command. For example, man git-clone gives you a complete man page for the git clone command.

*Prerequisites*: The following are the prerequisites for carrying out this example:

1. Having completed the previous subsections that familiarize you with Git commands executed on the Linux command line

2. Having an Internet connection and a suitable Web browser installed and operating on your Linux system

3. Having completed Example W24.3 and having your Web browser pointed at the test repository created in that example so you can check on its contents

4. Having access to an account on GitHub that has in it the existing repository **test** created in Example W24.3.

*Procedure*: Perform the following steps, in the order presented, to meet the objectives of this example:

1. Create a new empty directory beneath your home directory on your Linux system named **github\_clone** and make that directory the current working directory.

% **mkdir github\_clone**

% **cd github\_clone**

%

This new directory will serve as the file system *landing zone*, within which the git clone command shown in the next step will replicate the entire remote GitHub repository.

2. Use the git clone command to transfer the contents of the remote GitHub repository into the current working directory. Remember that the URL we show in the command is different from the one that you will be seeing on your system, so make the appropriate changes. Remember that the URL for this repository is valid for our system and account at Github. Your URL and account will be different from the one shown here!

% **git clone https://github.com/bobk48/test**

Cloning into 'test'...

remote: Counting objects: 6, done.

remote: Compressing objects: 100% (3/3), done.

remote: Total 6 (delta 0), reused 6 (delta 0)

Receiving objects: 100% (6/6), done.

Checking connectivity... done.

%

3. List the contents of the working directory. It should contain the complete repository from your GitHub repository in Example W24.3. The directory listed is the working directory for the cloned repository. If you descend into that directory, everything that is in your GitHub repository is in **test**.

% **ls**

test

%

4. Make the directory **test** the current directory, in preparation for putting a new file in it that you will then transfer up to the GitHub repository.

% **cd test**

%

The test directory is now your working directory in Git terminology.

5. With your favorite text editor, create a new file, with any contents you want in it, in the directory **test**. Save the file and exit the text editor.

% **vi newfile2.txt**

newfile2.txt: new file: 1 lines, 58 characters.

%

6. List the contents of directory **test**.

% **ls**

README.md newfile.txt newfile2.txt

%

7. Check the status of the local repository with the git status command.

% **git status**

On branch master

Your branch is up-to-date with 'origin/master'.

Untracked files:

(use "git add <file>..." to include in what will be committed)

newfile2.txt

nothing added to commit but untracked files present (use "git add" to track)

%

8. Stage and commit the new file to the local repository, in preparation for transferring it up to GitHub.

% **git add newfile2.txt**

% **git commit –m “Added newfile2.txt to test”**

%

9. Use git push to transfer the new file up to the GitHub repository named **test** on the branch **master**.

Remember that the URL for this repository is valid for our system and account at Github. Your URL and account will be different from the one shown here!

% **git push https://github.com/bobk48/test master**

Username for 'https://github.com': your\_username

Password for 'https://bobk48@gmail.com@github.com': xxx

Counting objects: 4, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 371 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/bobk48/test

4dc2de7..b651617 master -> master

%

10. From your Web browser, examine the GitHub repository named **test**. It should now contain the file you pushed to it in step 9.

*Conclusions*: The easiest way to create a local repository that is an exact copy of a GitHub repository is to use the git clone command.

In-Chapter Exercise W24.17

What refspec URL and fetch assignments are listed for the repository **test**? What branch refspecs are listed? How did you find this out?

In-Chapter Exercise W24.18

After having completed both Examples W24.3 and W24.4, what Linux command(s) would enable you to update the repository **test** from Example W24.3 with what is in your online GitHub repository named **test**?

In-Chapter Exercise W24.19

What Git command can you use to see the abbreviated list of commits in the current branch of a repository and their commit comments?

**Example W24.5: Pulling from a GitHub Repository**

*Objectives*: To show the mechanics of taking content from a GitHub repository branch and adding it to a local repository by merging it with a local repository branch.

*Introduction*: The easiest way to share content from a GitHub repository is to use the git pull command. This command combines git fetch and git merge so that the content of a GitHub repository branch can be duplicated on a branch of one of your local repositories. We create a new local working directory and repository in it to receive the content from a remote source on GitHub. We then use the GitHub repository **https://github.com/bobk48/linuxthetextbook**, which contains all of the source code examples for the book you are now reading, as the remote source. This is the URL specified in the Preface, and in Section 14.5.7, in the printed book! We repeat the instructions given in Section 14.5.7 in the printed book for your convenience here.

|  |  |
| --- | --- |
| **Command** | **Description** |
| git init | Creates the **.git** directory in the working directory, initializing the data structures and objects necessary for a repository to exist |
| git status | Reports on the differences between files in the working directory and the index, and what files are untracked |
| git add <file> | Stages a file to the index |
| git commit -m "<Message>" | Takes a snapshot of the index, both files and directories, with <Message> automatically added |
| git pull <ref> master | Retrieves the branch named **master** from the remote <ref> designated into the current branch |

Table W24.5 Git Commands Referenced

*Git Commands Referenced*: Table W24.5 shows the Git commands, and a brief description of each, that are used in this example. It is arranged in the order presented. Any argument enclosed in < > is a string of text. In order to get a more complete description of all the commands in the table, you can look at the man page for a particular command. For example, man git-pull gives you a complete man page for the git pull command.

*Prerequisites*: The following are the prerequisites for carrying out this example:

1. Knowing how to navigate to www.github.com using a Web browser GUI interface

2. Having completed the previous sub-sections that familiarize you with Git commands executed on the Linux command line

3. Having an Internet connection and a suitable Web browser installed and operating on your Linux system

4. Having completed Examples W24.3 and W24.4

*Procedure*: Carry out the following steps, in the order presented, to meet the objectives of this example:

1. Begin by setting up a new local repository working directory and initializing it as a Git repository.

% **mkdir linuxthetextbook**

% **cd linuxthetextbook**

% **git init**

Initialized empty Git repository in /usr/home/bob/linuxthetextbook/.git/

%

2. Put a file in the new repository.

% **touch Readme.txt**

%

3. Examine the status of the new repository.

% **git status**

On branch master

Initial commit

Untracked files:

(use "git add <file>..." to include in what will be committed)

Readme.txt

nothing added to commit but untracked files present (use "git add" to track)

%

4. Stage the **Readme.txt** file, and make your initial commit into the new repository.

% **git add Readme.txt**

% **git commit -m "first commit"**

[master (root-commit) 57e0400] first commit

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 Readme.txt

%

5. Use the git pull command to fetch and merge the entire l**inuxthetextbook** repository from the branch named **master**.

% **git pull https://github.com/bobk48/linuxthetextbook master**

From https://github.com/bobk48/linuxthetestbook

%

You are placed in the default editor. Leave the first line, numbered 1, as is, and save and quit the file.

\* branch master -> FETCH\_HEAD

1 Merge branch 'master' of https://github.com/bobk48/linuxthetextbook

2

3 # Please enter a commit message to explain why this merge is necessary,

4 # especially if it merges an updated upstream into a topic branch.

5 #

6 # Lines starting with '#' will be ignored, and an empty message aborts

7 # the commit.

/usr/home/bob/linuxthetextbook/.git/MERGE\_MSG: 7 lines, 295 characters.

Merge made by the 'recursive' strategy.

.gitattributes | 22 ++++++

.gitignore | 43 ++++++++++++

README.md | 4 ++

...<Output Truncated>

6. Examine the contents of the working directory.

% **ls**

The current contents of the linuxthetextbook repository

%

*Conclusion*: Using the git pull command, you can take content from a GitHub repository branch and put it on a local repository branch.

In-Chapter Exercise W24.20

What refspec URL and fetch assignments are listed for the repository l**inuxthetextbook**? How did you find this out?

**Questions and Problems**

**16.** Create a three-branch repository of commits exactly like Figure W24.15. Use any number of text files that you modify between commits on the three branches. Keep the default name for the branch **master**, but name the other two branches **test** and **dev**, as seen in Figure W24.15.

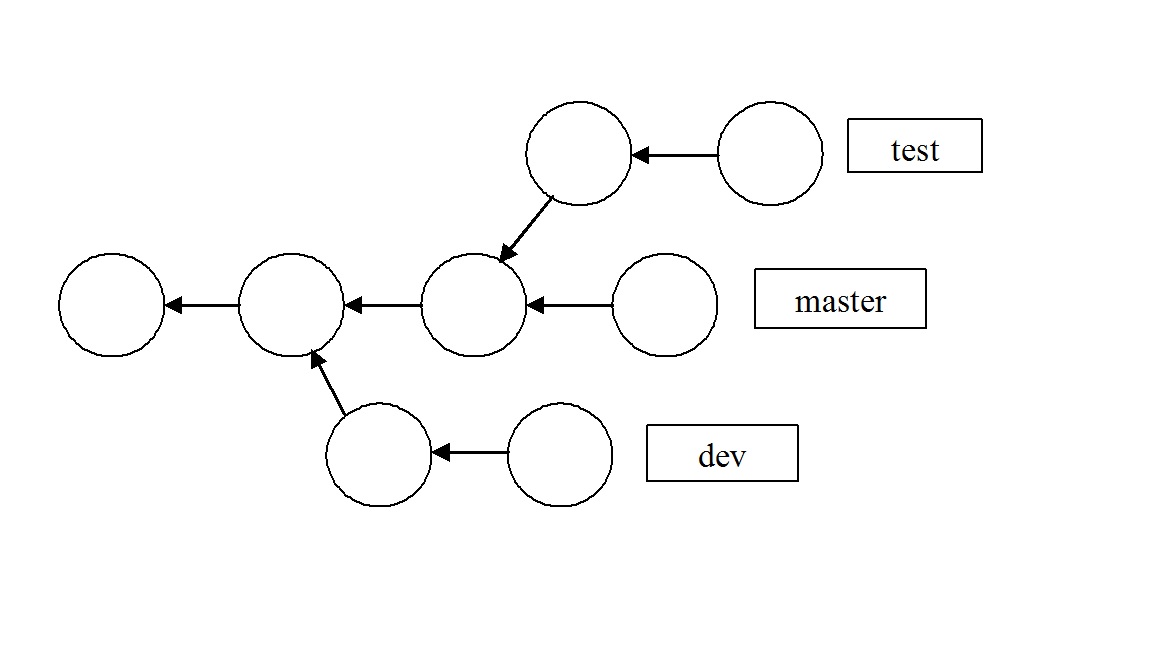


Figure W24.15 Three Branch Repository #1

**17.** Create a three-branch repository of commits exactly like Figure W24.16. Use any number of text files that you modify between commits on the three branches. Keep the default name for the branch master, but name the other two branches **test** and **dev**, as seen in Figure W24.16.

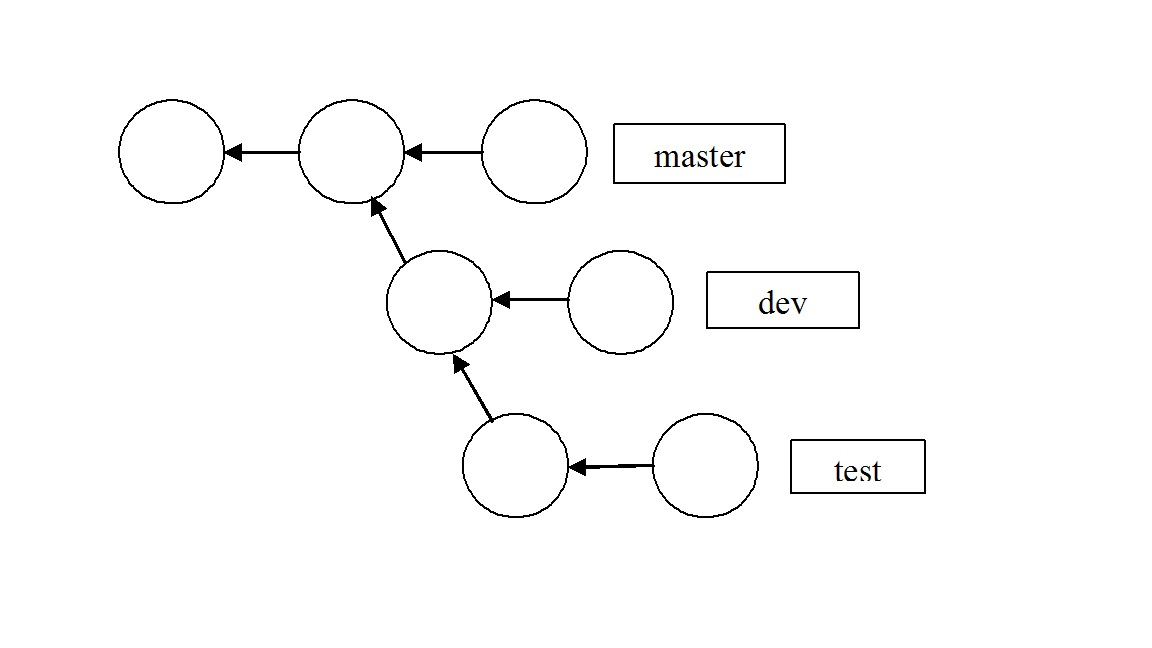


Figure W24.16 Three Branch Repository #2

**18.** As an alternative to using git pull in Example W24.5 to obtain the source code for this book from the listed GitHub repository (**https://github.com/bobk48/linuxthetextbook**), use the git clone command, as shown in Example W24.4, from your home directory on your Linux system. What will be the name of the repository directory created by Git on your local machine that contains the source code files?

**19.** Before doing this problem, go through all of the material in Chapter W22 at the book website on the Zettabyte File System, including the in-chapter exercises and problems. Then, in your own words, describe the differences and similarities between Git, GitHub, and ZFS. For example, the commands zfs snapshot and git commit are very similar. In what way do they differ? In your opinion, is it possible to use the commands zpool and zfsto achieve the same or similar results as using Git and GitHub? What other Linux commands would be needed to augment the ZFS file system commands to attain results that are similar to Git and GitHub functions, commands, and capabilities?