# Navigating in git

The data model of Git is different from other common **version control systems** (**VCSs**) in the way Git handles its data. Traditionally, a VCS will store its data as an initial file, followed by a list of patches for each new version of the file:

Git is different: Instead of the regular file and patches list, Git records a snapshot of all the files tracked by Git and their paths relative to the repository root—that is, the files tracked by Git in the filesystem tree. Each commit in Git records the full tree state. If a file does not change between commits, Git will not store the file again.

In short, the Git data model can be summarized as shown in the following diagram:



* The commit object points to the root tree. The root tree points to subtrees and files.
* Branches and tags point to a commit object and the HEAD object points to the **branch** that is currently checked out. So, for every commit, the full tree state and snapshot are identified by the root tree.

Git's object storage is a key-value storage, the key being the ID of the object and the value being the object itself. The key is an SHA-1 hash of the object

## Git's objects

There are four types of objects in Git

- Files, or blobs as they are also called in the Git context

- Directories, or trees in the Git context

- Commits

- Tags

the special HEAD pointer that refers to the branch/commit currently being checked out

### The commit object

The Git command git cat-file -p will print the object given as an input. Normally, it is not used in everyday Git commands, but it is quite useful to investigate how it ties the objects together

**$ git cat-file -p HEAD**

We can now see the commit object, consisting of the root tree (tree), the parent commit object's ID (parent), the author and timestamp information (author), the committer and timestamp information (committer), and the commit message.

### The tree object

There are many ways to see the objects in the Git database. The git ls-tree command can easily show the content of trees and subtrees, and git show can show the Git objects, but in a different way.

We can also specify that we want the tree object from the commit pointed to by HEAD by specifying:

$ git cat-file -p HEAD^{tree}

The special notation HEAD^{tree} means that from the reference given, HEAD recursively dereferences the object at the reference until a tree object is found.

The first tree object is the root tree object found from the commit pointed to by the master branch, which is pointed to by HEAD.

A generic form of the notation is <rev>^<type>, and will return the first object of <type>, searching recursively from <rev>.

### The blob object

So, the objects are tied together, blobs to trees, trees to other trees, and the root tree to the commit object, all connected by the SHA-1 identifier of the object.

### The branch object

we can take a look at the branch inside the .git folder where the whole Git repository is stored. If we open the text file .git/refs/heads/master, we can actually see the commit ID that the master branch points to. We can do this using cat, as follows:

**$ cat .git/refs/heads/master**

**13dcada077e446d3a05ea9cdbc8ecc261a94e42d**

We can also see that HEAD is pointing to the active branch by using cat with the .git/HEAD file:

**$ cat .git/HEAD**

The branch object is simply a pointer to a commit, identified by its SHA-1 hash.

### The tag object

There are three different kinds of tag: a lightweight (just a label) tag, an annotated tag, and a signed tag. In the example repository, there are two annotated tags:

**$ git cat-file -p v1.0**

As you can see, the tag consists of an object—which, in this case, is the latest commit on the master branch—the object's type (commits, blobs, and trees can be tagged), the tag name, the tagger and timestamp, and finally the tag message.

There are many ways to see the objects in the Git database. The git ls-tree command can easily show the content of trees and subtrees, and git show can show the Git objects, but in a different way.

## The three stages

When a file is moved to the staging area, the SHA-1 hash of the file is created and the blob object is written to Git's database. This happens every time a file is added, but if nothing changes for a file, it means that it is already stored in the database.

We can use the git fsck command to check for dangling objects—that is, objects that are not referred to by other objects or references:

$ git fsck --dangling

=====

The following diagram describes the tree stages and the commands used to move between the stages:



## Viewing the DAG

The history in Git is formed from the commit objects; as development advances, branches are created and merged, and the history will create a directed acyclic graph, the DAG, because of the way that Git ties a commit to its parent commit. The DAG makes it easy to see the development of a project based on the commits.

You can view the history (the DAG) in Git by using its git log command.

The simplest way to see the history is to use the git log command; this will display the history in reverse chronological order.

$ git log --decorate --graph --oneline --all

The --decorate option shows the branch names after the abbreviated commit ID, and the --all option shows all the branches,instead of just the current one(s):

Fortunately, the log command gives us the ability to create our own output format. So, we can make a history view similar to the previous one. The colors are made with the

%C<color-name>text-be-colored%Creset syntax, along with the author and timestamp information

and some colors to display it nicely:

**$ git log --all --graph \**

**--pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%ci) %C(bold blue)**

**<%an>%Creset'**

This is a bit cumbersome to write, but luckily, it can be made as an alias so you only have to write it once:

**git config --global alias.graph "log --all --graph --pretty=format:'%Cred%h%Creset -**

**%C(yellow)%d%Creset %s %Cgreen(%ci) %C(bold blue)<%an>%Creset'"**

let's limit the log to only look through the history since the last tag (release). To find the last tag, we can use git describe:

$ git describe

v3.1.0.201310021548-r-96-gb14a939

The preceding output tells us three things:

The last tag was v3.1.0.201310021548-r

The number of commits since the tag was 96

The current commit in abbreviated form is b14a939

asaki@WW7FRDEV01ASI MINGW64 /d/work/marsng/source (BHAOrdinaryGroup)

$ git describe

MarsNG\_ContainerShip\_2.0-201-g70f486565

$ git log MarsNG\_ContainerShip\_2.0..HEAD --oneline --grep "BHA"

This gives us all the BHA commits, but we can format this to a

format that is easily parsed with the --pretty option.

First, we will print the abbreviated commit ID (%h), followed by a

separator of our choice (|), and then the commit subject (%s, the first

line of the commit message), followed by a new line (%n), and the

body (%b):

--pretty="%h|%s%n%b"

git log MarsNG\_ContainerShip\_2.0..HEAD --grep "BHA" -20 --pretty="%h|%s%n%b"

If we just wanted to extract the bug IDs from the commit messages and didn't care about the commit IDs, we could have just used grep after the git log command, still limiting the log to the last tag:

**$ git log v3.1.0.201310021548-r..HEAD | grep "Bug: "**

If we just want the commit IDs and their subjects, but not the actual bug IDs, we can use the --oneline feature of git log combined with the --grep option:

**$ git log --grep "Bug: " --oneline v3.1.0.201310021548-r..HEAD**

## Getting a list of the changed files

The following command lists all the files that have changed since the last release

(v3.1.0.201310021548-r):

**$ git diff --name-only v3.1.0.201310021548-r..HEAD**

The following command lists all the files that have changed since the last release

$ git diff MarsNG\_ContainerShip\_2.0..HEAD --name-only -5 --oneline

By specifying --name-only, Git will only give the paths of the files that were changed by the commits in the range specified as output.

If we only want to show which files have been deleted in the repository, we can use the --diff-filter switch with git diff:

**$ git diff --name-only --diff-filter=D v3.1.0.201310021548-r..HEAD**

There are also switches for the files that have been added (A), copied (C), deleted (D), modified (M), renamed (R), and so on.

asaki@WW7FRDEV01ASI MINGW64 /d/work/marsng/source (BHAOrdinaryGroup)

$ git diff HEAD~..HEAD --name-only --diff-filter=M

BV.Mars.Kernel/Domain/TransverseBulkhead/OrdinaryStiffener.cs

BVUtils

By specifying --name-only, Git will only give the paths of the files that were changed by the

commits in the range specified as output.

In the repository, run gitk --all & to bring up the gitk interface.

## Finding commits in the history

we can use the --grep option to find specific strings in commit messages. In this recipe, we look at the entire history and search every commit that has "Performance" in its commit message:

$ git log --grep "Performance" --oneline –all

Note that the search is case sensitive—had we searched for "performance" (all in lower case), the list of commits would have been very different:

## Searching through the history code

Sometimes, it is not enough to list the commit messages. You may want to know which commits touched a specific method or variable. This is also possible using git log. You can perform a search for a string, for example, or a variable or method, and git log will give you the commits, adding or deleting the string from the history

We would like to find all the commits that have had changes made to the lines that contain the "FindNextStructure" method.

$ git log -G"FindNextStructure" –oneline

This method is quite convenient to use in finding out when a given string was introduced or deleted, and to get the full context and commit at that point in time.

The -G option used with git log will look for differences in the patches that contain added or deleted lines that match the given string. However, these lines could also have been added or removed because of some other refactoring/renaming of a variable or method. There is another option that can be used with git log, namely -S, which will look through the difference in the patch text in a similar way to the -G option, but will only match commits where there is a change in the number of occurrences of the specified string—that is, a line added or removed, but not added and removed.

Let's see the output of the -S option:

**$ git log -S"isOutdated" --oneline**

We can see the content of the commit with the git show command, and use grep -C4 to limit the output to just the four lines before and after the search string:

$ git show c9c57d3 | grep -C4 " FindNextStructure "

# Configuration

This chapter will give an overview of the most important options available and will provide the right tools for learning and navigating the multitude of

configuration flags and fields available, in order to tailor your Git experience to your own needs.

We could use the command git config --list to list configuration entries. This list is actually made from three different levels of configuration that Git offers: system-wide configuration, SYSTEM; global configuration for the user, GLOBAL; and local repository configuration, LOCAL.

$ git config --list --system

# list the global configuration

$ git config --list –global

# list the configuration for this repository

$ git config --list –local

We can also query a single key and limit the scope to one of the three layers, by using the following command:

$ git config --global user.email

We can set the email address of the user to a different one for the current repository

$ git config --local user.email john@example.com

When a configuration value is needed, Git will first look in the LOCAL configuration. If not found here, the GLOBAL configuration is queried. If it is not found in the GLOBAL configuration, the SYSTEM configuration is used.

You can set the editor to the editor of your choice either by changing the $EDITOR environment variable or with the core.editor configuration target, for example:

$ git config --global core.editor vim

## Querying the existing configuration

If we are just interested in a single configuration item, we can just query it by its

section.key or section.subsection.key:

**$ git config user.name**

Git's configuration is stored in plain text files and works like a key-value storage. You can set/query by key and get the value back.

**$ cat .git/config**

It is also easy to set configuration values. You can use the same syntax as you did when querying the configuration, except you need to add an argument to the value. To set a new email address on the LOCAL layer, we can execute the following command line:

**git config user.email john.doe@example.com**

The LOCAL layer is the default, if nothing else is specified. If you require whitespaces in the value, you can enclose the string in quotation marks, as you would do when configuring your name:

**git config user.name "John Doe"**

It is also very easy to delete/unset configuration entries:

**$ git config --unset my.own.config**

## Templates

To let Git know about our new commit message template, we can set the configuration variable commit.template to point at the file we just created with that template; we'll do it globally so it is applicable to all our repositories:

$ git config --global commit.template $HOME/.gitcommitmsg.txt

When commit.template is set, Git simply uses the content of the template file as a starting point for all commit messages. This is quite convenient if you have a commit-message policy, as it greatly increases the chances of the policy being followed. You can even have different templates tied to different repositories since you can just set the configuration at the local level

## Git aliases

An alias is a nice way to configure long and/or complicated Git commands to represent short useful ones. An alias is simply a configuration entry under the alias section. It is usually configured to --global to apply it everywhere.

The alias method is also good for creating the Git commands you think are missing in Git. One of the common Git aliases is unstage, which is used to move a file out of the staging area, as shown in the following command:

$ git config --global alias.unstage 'reset HEAD --'

Git simply runs the command the alias is short for. It is very convenient for long Git commands, or Git commands that are hard to remember exactly how to write. Now, all you have to remember is the alias and you can always look in the configuration file for it.

A common use case for aliases is to format the history of Git in specific ways. Let's say you want the number of lines added and deleted for each file in the commit displayed along with some common commit data. For this, we can create the following alias so we don't have to type everything each time:

$ git config --global alias.ll "log --pretty=format:"%C(yellow)%h%Cred%d %Creset%s %Cgreen(%cr)%C(bold blue)<%an>%Creset" --numstat"

## The refspec exemplified

In a lot of the Gitcommands, the refspec is used, but often implicitly, that is, the refspec is taken from the configuration file.

The format of the refspec is in the form of <source>:<destination>. For a fetch refspec, this means that <source> is the source on the remote side and <destination> is local. For a push refspec, <source> is local and <destination> is remote. The refspec can be prefixed by a + to indicate that the ref pattern can be updated even though it isn't a fast-forward update. It is not possible to use partial globs in the refspec pattern, as shown in the following line:

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

However, it is possible to use namespacing. That's why we had to rewrite the stable-xxx branches to stable/xxx to fit into a namespace pattern:

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

# Branching, Merging, and Options

### Managing your local branches

If you want to add a description to the branch, you can do this with the --edit-description option for the git branch command:

**$ git branch --edit-description newBugFix**

Git stores the information in the local git config file; this also means that you cannot push this information to a remote repository. To retrieve the description for the branch, you can use the --get flag for the git config command:

**$ git config --get branch.newBugFix.description**

The branch information is stored as a file in .git/refs/heads/newBugFix:

If you need a branch from a specific commit hash, you can create it with the git branch command as follows:

$ git branch anotherBugFix 979e346

$ git log -1 anotherBugFix --format=format:%h

979e346

$ git log -1 anotherBugFix --format=format:%H

979e3467112618cc787e161097986212eaaa4533

2. As you can see, the abbreviated commit hash is shown when you use %h, and the full commit hash is shown when you use %H. You can see that the abbreviated commit hash is the same as the one used to create the branch. Most of the time, you want to create and start working on the branch immediately:

$ git checkout -b lastBugFix 979e346

Switched to a new branch 'lastBugFix'

The remoteBugFix2 branch is just a local branch at the moment with no tracking information; to set the tracking branch, we need to use --setupstream-to or -u as a flag to the git branch command:

$ git branch --set-upstream-to origin/stable-3.2

Branch remoteBugFix2 set up to track remote branch stable-3.2 from origin.

Instead of using Gitk, you can also add -v to the git branch command or even another -v:

**$ git branch -v**

With -v, you can see the abbreviated commit hash for each branch, and with -vv, you can also see that the master branch tracks the origin/master branch:

**$ git branch -vv**

## Forcing a merge commit

To force a merge commit, you need to use the --no-ff

$ git merge origin/stable-3.2 --no-ff --edit --quiet

Then, to create the merge and allow you to decide what will be part of the commit, you can use --no-commit:

$ git merge origin/master --no-ff --no-commit

## Compute the difference between branches

Then, to list files that have changed between these branches, you can use the --name-only flag.

$ git diff --name-only origin/stable-3.1 origin/stable-3.2 org.eclipse.source

We are building the command in this pattern, that is, git diff [options]

<commit> <commit> <path>

To show the deleted or added files between the branches. This is done by using the --diff-filter=DA and --name-status options.

$ git diff --name-status --diff-filter=DA origin/stable-3.1 origin/stable-3.2

This shows the files that have been added and deleted while moving from origin/stable-3.1 to origin/stable-3.2.

## Branches with remotes

We will start by checking out a local branch that tracks a remoteBranch

**$ git checkout -b remoteBugFix --track origin/stable-3.2**

You can also add a remote to an existing branch, which is very handy when you realize that you actually wanted a remote tracking branch but forgot to add the tracking information while creating the branch

**$ git checkout -b remoteBugFix2 2e0d17**

The remoteBugFix2 branch is just a local branch at the moment with no tracking information; to set the tracking branch, we need to use --setupstream-to or -u as a flag to the git branch command:

**$ git branch --set-upstream-to origin/stable-3.2**

## Forcing a merge commit

To force a merge commit, you need to use the --no-ff flag; *no-ff* means no fast forward. We will also use the --quiet flag to minimize the output and --edit to allow us to edit the commit message

Using the --no-commit option, Git will make the merge and stop before committing, allowing you to modify and add files to the merge commit before committing.

**$ git merge origin/master --no-ff --no-commit**

To verify whether the result is as expected, we can compute the difference for this using git diff to show that the files are as they are on the origin/master branch, excluding the LICENSE file:

**$ git diff origin/master !(LICENSE)**

We are telling the diff command to diff our current HEAD commit and branch origin/master, and we do not care about the diffs in LICENSE

## Using git reuse recorded resolution (rerere) to merge Git conflicts

While working on a feature branch, you probably like to merge daily or perhaps more often, but when you work on long-living feature branches, you end up in a situation where you have the same conflicts occurring repeatedly.

Here, you can use git rerere, which stands for *reuse recorded resolution*. Git rerere is not enabled by default, but can be enabled with the following command:

**$ git config rerere.enabled true**

You need to find a commit for which you are interested in getting this information. Then, use the --contains flag for the git branch command

**$ git branch --contains 699900c308**

The previous command lists all the branches that have the specific commit.

*The -a option indicates that you wish to check all the remote branches as well.*

*If you leave this out, it will check only local branches.*

**$ git branch –a --contains 699900c308**

Let's try to see the branches where the v2.3.0.201302130906 tag is present:

**$ git branch -a --contains v2.3.0.201302130906**

## Compute the difference between branches

Checking the difference between branches can show valuable information before merging. A regular git diff between two branches will show you all the information

Then, to list files that have changed between these branches, you can use the –nameonly flag.

the command in this pattern, that is, git diff [options] <commit> <commit> <path>.

$ git diff --name-only origin/stable-3.1 origin/stable-3.2 src/org/transport

Let's try the same diff between branches, but this time we will diff the entire branches, not

just a sub-directory; however, we only want to show the deleted or added files between

the branches. This is done by using the --diff-filter=DA and --name-status options. The --namestatus

option will only show the filenames and the type of change. The --diff-filter=DA

option will only show the deleted and added files:

$ git diff --name-status --diff-filter=DA origin/stable-3.1 origin/stable-3.2

## Orphan branches

However, in some situations, it is useful to have a branch with no parent.

One example would be an instance where you have your code base in two separate Repositories

It is to use an orphan branch that can help you to fetch one repository in another

It is actually easy to create an orphan branch. The flag --orphan to checkout will do it.

$ git checkout --orphan fresh-start

the orphan branch does not share history with the master branch, and git will not allow you to merge the branch. It shouldn't come as a surprise, since it is basically what an orphan branch is all about. However, you can still merge an orphan branch by allowing unrelated histories to be merge

$ git checkout master

$ git merge fresh-start

fatal: refusing to merge unrelated histories

$ git merge fresh-start --allow-unrelated-histories

it is a strong feature to know when you need to reorganize your code base.

# Rebasing Regularly and Interactively, and Other Use Cases

Rebase is an incredibly strong Git feature.

Check out a new branch, rebaseExample, which tracks origin/stable-3.1:

$ git checkout -b rebaseExample --track origin/stable-3.1

When you execute git rebase, Git starts by finding the common ancestor of the current HEAD branch and the branch you want to rebase to. When Git finds merge-base, it will find the commits that are not available in the branch you are rebasing onto. Git will simply try to apply those commits one by one.

$ git checkout -b resetAuthorRebase -t origin/master

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

Switched to a new branch 'resetAuthorRebase'

When you are working on a new feature and have branched from an old release into a feature branch, you might want to rebase this branch onto the latest release. When looking into the list of commits on the feature branch, you may realize that some of the commits are not suitable for the new release. In that case, when you want to rebase the branch onto a new release, you will need to remove some commits. This can be achieved with interactive rebasing, where Git gives you the option to pick the commits you wish to rebase.

Now, we have these six commits on top of the origin/stable-3.1 branch, and we want to

squash these commits into two different commits. This can be done by simply running git rebase --interactive. Note that we are not specifying which branch we want to rebase to, since we have already set up a tracking branch when we created the branch using --track

The rebase, in general, refuses to take merged commits as part of the rebase scenario. Although you can use the --preserve-merges flag, as per the Git Help section, this is not recommended.

# Changing the author of commits using a rebase

You can change the author of the HEAD commit by using git commit --amend --reset-author; however, this will only change the author of HEAD and leave the rest of the commits as they were.

**$ git commit --amend --reset-author**

**$ git log --format='format:%h %an <%ae>' origin/stable-3.2..HEAD**

We will list all the commits from origin/stable-3.2 to HEAD and we will define a format with

%h as the abbreviated commit hash, %an for the author's name, and %ae for the author's email address. From the output, you can see that I am now the author of the HEAD commit, but what we really wanted was to change the author of all the commits. To do this, we will rebase onto the origin/stable-3.2 branch; then, for each commit, we will stop to amend and reset the author. Git can do most of that work with --exec option for the git rebase, as follows:

$ git log --format='format:%h %an <%ae>' origin/stable-3.2..HEAD

We will list all the commits from origin/stable-3.2 to HEAD and we will define a format with

%h as the abbreviated commit hash, %an for the author's name, and %ae for the author's email address.

**$ git rebase --interactive --exec "git commit --amend --reset-author" origin/stable-3.2**

As you will see, this process is not very good, as the commit message editor opens every time and you have to close the editor to allow Git to continue with the rebase. To stop the rebase, clear the commit message editor and Git will return to the command line; then, you can use git rebase --abort as follows:

To achieve what we really want, you can add the --reuse-message option for git commit; this

will reuse the commit message for the commit you will specify. We want to use the

message of HEAD, as we are going to amend it to the HEAD commit. So, try again, as shown in the following command:

**$ git rebase --interactive --exec "git commit --amend --reset-author --reuse-message=HEAD"**

## Autosquashing commits

# Storing Additional Information in Your Repository

A Git note is essentially an extra refs/notes/commits reference in Git. Here, you add additional information to the commits that can be displayed when running a git log command. You can also release the notes into a remote repository so that people can fetch the notes.

## Tagging commits in the repository

There are two types of tags, a lightweight tag and an annotated tag. The lightweight tag is very similar to a branch, since it is just a named reference, such as refs/tags/version123 . This points to the commit hash of the commit you are tagging; whereas if it were a branch, it would be refs/heads/version123. The difference is that the branch moves forward when you work and commit to it. A tag will always point to the same commit hash. We will discuss the annotated tag shortly.

For the git log command, we are using the --no-merges option, which will show commits that only have one parent. The --oneline option we have used before tells Git to limit the output to one line per commit. Moreover, -11 shows us the last 11 commits (10 commits before the latest).

**$ git log -11 --no-merges –oneline**

$ git tag -l "v2.3.0.2\*"

We are using the git tag command with -l as a flag, since we want to list the tags and not tag the current HEAD. Some repositories have a lot of tags; so to prevent the list from becoming too long, you can specify which tags you want to list and use a \* wildcard as we did previously.

So, in order to add more information, we should use an annotated tag. An annotated tag is a tag where you have to add some information to the tag. To create an annotated tag, we use the --annotate tag for the git tag command:

$ git tag --annotate -m "Release Maturity rate 97%" **'v2.3.0.201409022257rc2' 1c4ee41**

To delete a tag, you need to apply the -d flag:

$ git tag -d v1.3.0.201202121842-rc4

$ git tag -a -m "Local created tag" v1.3.0.201202121842-rc4

We have recreated the tag, and it points to HEAD because we did not specify a commit hash at the end of the command

Once you have pushed a tag to a remote repository, you should never change it, since the developers who are fetching from the repository may never know about the changes unless they clone again or delete the tags locally and fetch them again.

# Extracting Data from the Repository

In this chapter, we will cover the following recipes:

Extracting the top contributor

Finding bottlenecks in the source tree

Grepping the commit messages

The contents of the releases

Finding what has been achieved in the repository in the last period

## Extracting the top contributor

The git log command has different options, such as --numstat, that will show the number of files added and lines deleted for each file since each commit.

However, for finding the top committer in the repository, we can just use the git shortlog command. The shortlog Git command is very simple and does not have a lot of options or flags to use with it. To sort and find the top committer, we can use the -n or --numbered option to sort the output; the top committer is on top:

$ git shortlog -5 --numbered

$ git shortlog master..BHAOrdinaryGroup -sn

75 yu peng

44 adel saki

24 Baptiste Célérier

To list it for the repository, we need to add --all at the end of the command so as to execute the command for all branches as follows:

we can use -s or --summary to only show the commit count for each developer as follows

$ git shortlog --numbered --summary --all

So, to list the top committers for the last six months, we can add --since="6 months ago" to the git shortlog command as follows.You can use "n weeks ago", "n days ago", "n months ago", " n hours ago", and so onfor specifying time periods. You can also use specific dates, such as "1 october 2013".

$ git shortlog --numbered –summary --all --since="6 months ago"

You can also list the top committer for a specific month using the --until option, where you can specify the date you wish to list the commit until. This can be done as follows:

$ git shortlog --numbered --summary --email --all --since="30 september

2013" --until="1 november 2013"

You can use the command on the files as well

**$ git shortlog --numbered --summary --email ./pom.xml**

it is fairly simple to get some indication of who to go to for the different files or directories in Git.

$ git shortlog --all -sn \*.sln

## Finding bottlenecks in the source tree

The --dirstat option shows which directories have changed in the commit and how much they have changed compared to each other. The default setting is to count the number of lines added to or removed from the commit.

$ git log -1 –dirstat

You can compensate for this slightly by using --dirstat=lines. This option will look at each file line by line and see whether they have changed compared to the previous version as follows:

$ git log -1 --dirstat=lines

If you would like to limit the output to only show directories with a certain percentage or higher, we can limit the output as follows:

$ git log -1 --dirstat=lines,10

By default, Git does not count the changes in the subdirectories, but only the files in the directory, we can add cumulative to the --dirstat=lines,10 command, and this will cumulate the changes and calculate a percentage.

$ git log -1 --dirstat=files,10,cumulative

Obviously, you can also do this for all the commits between two releases or two commit hashes, we will be using git diff, as Git will show the accumulated diff between the two releases, and git log will show dirstat for each commit between the releases

git diff master..BHAOrdinaryGroup --dirstat=lines,cumulative

We can then dig a little deeper using --stat or --numstat for the directory, and again use git diff. We will also use -- relative="org.eclipse.jgit.test/tst/org/eclipse/", which will show the relative path of the files from BV.Mars.UI.

git diff master..BHAOrdinaryGroup --numstat --relative=BV.Mars.UI

asaki@WW7FRDEV01ASI MINGW64 /d/work/marsng/source (BHAOrdinaryGroup)

$ git diff master..BHAOrdinaryGroup --pretty --numstat --relative=BV.Mars.UI/

31 0 BV.Mars.UI.csproj

7 2 UserInterface/ApplicationShell.cs

5 1 UserInterface/Context/DesignContext/BhDesignContext/BhOrdinaryStiffenerDesignContext.cs

1 1 UserInterface/Controls/CalcTorsionModelControlPresenter.cs

45 0 UserInterface/Forms/InputForms/BhaForms/BhaOrdinaryStiffener/BhaOrdinaryGroupInputCustomController.cs

6 1 UserInterface/Forms/InputForms/BhaForms/BhaOrdinaryStiffener/Data/AddStiffener.cs

30 0 UserInterface/Forms/InputForms/BhaForms/BhaOrdinaryStiffener/Data/AddStiffenerGroupe.cs

34 39 UserInterface/Forms/InputForms/BhaForms/BhaOrdinaryStiffener/Data/DataReferencePointType.cs

The first number is the number of lines added, and the second number is the lines removed from the files between the two branches

We have used git log, git diff, and git shortlog to find information about the repository, but there are so many options for those commands on how to find bottlenecks in the source code.

If we want to find the files with the most commits, and these are not necessarily the files with the most line additions or deletions, we can use git log. We can use git log between the origin/stable-3.1 and origin/stable-3.2 branches and list all the files changed in each commit.

$ git log origin/stable-3.1..origin/stable-3.2 --format=format: --nameonly

You can see from the extensive output that you only see file names and nothing else. This is due to the options used. The --format=format: option tells Git to not display any commit-message related information, and --name-only tells Git to list the files for each commit

asaki@WW7FRDEV01ASI MINGW64 /d/work/marsng/source (BHAOrdinaryGroup)

$ git log master..BHAOrdinaryGroup --format=format: --name-only | sed '/^$/d' | sort | uniq -c | sort -r | head -10

21 BV.Mars.PresentationLogic/PresentationLogic/Presenters/FormsPresenters/InputFormsPresenters/CrossSectionPresenters/CrossSectionInputPresenters/EndConnections/BhStiffEndConnectionInputPresenter.cs

21 BV.Mars.Kernel/Domain/TransverseBulkhead/Bulkhead.cs

18 BV.Mars.PresentationLogic/Applicative/Controllers/CtrlBulkhead.cs

14 BV.Mars.Kernel/Domain/TransverseBulkhead/OrdinaryStiffenerGroup.cs

12 BV.Mars.UI/UserInterface/Forms/InputForms/BhaForms/BhaOrdinaryStiffener/Forms/BhaOrdinaryStiffenerInputForm.resx

12 BV.Mars.UI/UserInterface/Forms/InputForms/BhaForms/BhaOrdinaryStiffener/Forms/BhaOrdinaryStiffenerInputForm.Designer.cs

11 BV.Mars.UI/UserInterface/Forms/InputForms/BhaForms/BhaOrdinaryStiffener/Forms/BhaOrdinaryStiffenerInputForm.cs

10 BV.Mars.Kernel/Domain/TransverseBulkhead/SmallOrdinaryStiffener.cs

10 BV.Mars.Kernel/Domain/TransverseBulkhead/OrdinaryStiffener.cs

9 BV.Mars.UI/UserInterface/Forms/InputForms/BhaForms/BhaOrdinaryStiffener/Forms/EndConnectionInputUc.cs

We got the list of files, and we used sed '/^$/d' to remove empty lines from the output. After this, we used sort to sort the list of files. Then, we used uniq -c, which counts the occurrences of each item in the files and adds the number from the output. Finally, we sorted in reverse

order using sort -r and displayed only the top ten results using head 10.

To proceed from here, we should list all the commits between the branches that are changing the top file as follows

By adding the file to the end of the git log command, we will see the commits between the two branches

$ git log origin/stable-3.1..origin/stable-3.2 org.eclipse.jgit/src/org/eclipse/jgit/api/RebaseCommand.java

## Grepping the commit messages

Now we know how to list and sort files that we make frequent changes to and vice versa, but we are also interested in finding out the bugs that we are fixing, the features that we are implementing, and perhaps who is signing the code. All this information is usually available in the commit message.

Let's see how many commits in the repository are referring to a bug:

$ git log --all --grep="^[bB][uU][gG]: [0-9]+"

So what was the grep doing? The ^[Bb][Uu][gG]: part matches any combination of lowercase and uppercase bugs. The ^ character means from the beginning of the line. The : character is matching :. Then, we have [0-9]+, which will match any number between zero and nine, and the + part means one or more occurrences. but for now, we just want to count the commits. We can do this by piping it to wc -l (word count –l is to count the lines):

$ git log --all --oneline --grep="^[bB][uU][gG]: [0-9]+" | wc -l

Before piping it to wc, remember to use --oneline to limit the output to one line for each commit.

If you are used to using regular expressions in another scripting or programming language, you will see that using --grep does not support everything. You can enable a more extensive regular expression support using the --extended-regexp option for git log; however, the pattern still has to be used with --grep as follows:

$ git log --all --oneline --extended-regexp --grep="^[bB][uU][gG]: [0-9]{6}"

I have used a slightly different expression, and have now added {6} instead of +; the {6} searches for six occurrences of the associated pattern. In our case, it is six digits as it is next to the [0-9] pattern.

To shrink the regular expression even more, we can use --regexpignore-case, which will ignore the case for the pattern:

$ git log --all --oneline --regexp-ignore-case --extended-regexp --grep="^bug: [0-9]{6}"

## The contents of the releases

While extracting information from Git, one of the natural things to do is to generate release notes. To generate a release note, you need all the valid information from the repository between this release and the previous release

We start by listing the commits between two tags, v2.3.1.201302201838-r and v3.0.0.201305080800-m7, and then we build on that information:

By using git log with v3.0.0.201305080800-m7.. v3.0.0.201305080800-m7, we will get the commits between the tags. As we have a lot of commits between these two tags, let's count them using wc -l:

$ git log --oneline v2.3.1.201302201838-r..v3.0.0.201305080800-m7 | wc –l

Now, we will show the most modified files between the releases:

$ git log v2.3.1.201302201838-r..v3.0.0.201305080800-m7 --format=format: --name-only | sed '/^$/d' | sort | uniq -c | sort -r | head -10

This information is useful as we now have an overview of where the majority of the changes are. Then, we can find the commit that refers to bugs so we can list the bug IDs:

**$ git log --format=format:%h --regexp-ignore-case --extended-regexp --grep="bug: [0-9]{6}" v2.3.1.201302201838-r..v3.0.0.201305080800-m7 | xargs -n1 git log -1 | grep --ignore-case -E "commit [0-9a-f]{40}|bug:"**

We are using some Bash tools to get this list of fixed bugs. I will briefly explain what they are doing in this section

* The xargs -n1 git log -1 part will execute git log -1 on each commit coming from the first git log command, git log --format=format:%h --regexp-ignore-case --extended-regexp --grep="bug: [0-9]{6}" v2.3.1.201302201838-r..v3.0.0.201305080800-m7.
* The grep --ignore-case -E "commit [0-9a-f]{40}|bug:" part will ignore the case in the regular expression and -E will enable an extended regular expression. You might see that a lot of these options for the tool grep are the same options we have for git log. The regular expression is matching commit and 40 characters with the [0-9a-f] range or bug. The | character means or. Remember we are in the output from git log -1

## Finding what has been achieved in the repository in the last period

Sometimes it's useful to be able to extract what has been achieved in a specific range of time. Let's see how git log numerous arguments can help with this task.

we want to know everything that has been done in the last 30 days in the jgit repository that we have been analyzing so far:

$ git log --all --since="30 days ago"

Here, we use --all in order to see the commits in all the branches and not only the current one.

let's only show the commits by David Pursehouse:

$ git log --all --since="30 days ago" --oneline --author="David Pursehouse"

It looks like some merge commits are present. These are not really useful to describe the activity of the last month, so let's get rid of those with --no-merges:

$ git log --all --since="30 days ago" --oneline --author="David Pursehouse" --no-merges

# Enhancing Your Daily Work with Git Hooks, Aliases, and Scripts

We have our local descriptioInCommit branch, for which we need to set a description. We will use the --edit-description Git branch to add a description to our local branch. This opens the description editor, and you can type in a message by performing the following steps:

1. When you execute the command, the description editor will open and you can type in a message:

**$ git branch --edit-description descriptioInCommit**

**$ git status --porcelain**

**M fishtank.txt**

What you should note is the space before M the first time we use the --porcelain option for Git status. The porcelain option provides a machine-friendly output that shows the state of the files for Git status. The first character is the status in the staging area, whereas the second character is the status in the work area. So, MM fishtank.txt would mean the file is modified in the work area and in the staging area. So, if you modify fishtank.txt again, the following is the result you can expect:

**$ echo "sharks and oysters" >> fishtank.txt**

**$ git status --porcelain**

**MM fishtank.tx**

## Configuring and using Git aliases

Git aliases, like Unix aliases, are short commands that can be configured on a global level or for each repository. It is a simple way of renaming some Git commands to use short abbreviations.

It's very simple and straightforward to create an alias. You simply need to configure it with git config.

The next alias is a little different, as it will count the number of commits in the repository, and this can be done with the wc (wordcount) tool

However, since this is not a built-in Git tool, we have to use the exclamation mark and also specify Git:

$ git config alias.count '!git log --all --oneline | wc -l'

This also means you can execute external tools as if they were Git tools just by creating a Git alias; for instance, if you are using Windows, Mac, or Linux, you can create an alias as follows:

$ git config alias.wa '!explorer .' # Windows

This alias will open up an Window Explorer at the path you are currently at. The next one shows what changed in the HEAD commit. It executes this with the --name-status option for git log:

$ git config alias.gl1 'log -1 --name-status'

As you can see, it simply lists the commit and the files, including what happened to the files in the commit. As the aliases take arguments, we can actually reuse this functionality to list the information for another branch.

So, you can try and open the .git/config configuration file, or you can list the configuration with git config -list:

$ git config --list | grep alias

The alias feature is very strong, and the idea behind it is that you should use it to shorten those long one-liners that you often use. You can also use this feature to cut down those one-liners to shorter aliases so that you can use the command frequently and with more precision. If you have a long

and complex Git comment as an alias, you will run it the same way every time, where keying a long command is bound to fail once in a while.

## Setting up and using a commit template

Git also has the option of a static commit template. A static template is essentially just a text file configured as a template. Using the template is very easy and straightforward.

To configure the template, we need to use git config commit.template <pathtofile> to set it, and, as soon as it is set, we can try to create a commit and see how it works:

1. Start by configuring the template as follows:

$ git config commit.template ~/committemplate

2. Now list the config file to see that it has been set:

$ git config --list | grep template

As we predicted, the configuration was a success. The template, just like any other configuration, can be set at a global level using git config --global, or it can be set at a local repository level by leaving out the --global option. We configured our commit template for this repository only. Let's try and make a commit:

$ git commit --allow-empty

# Recovering from Mistakes

we'll explore the possibilities for undoing a commit in several ways, depending on what we want to achieve. We'll explore four ways to undo a commit:

* Undo everything, just remove the last commit as if it never happened
* Undo the commit and unstage the files; this takes us back to where we were before we started to add the files
* Undo the commit, but keep the files in the index or staging area so that we can just perform some minor modifications and then completethe commit
* Undo the commit with the dirty work area

## Undo – Remove a commit completely

In this example, we'll learn how we can undo a commit as if it had never happened. We'll learn how we can use the reset command to effectively discard the commit and thereby reset our branch to the desired state.

We will now undo the commit as though it never happened

**$ git reset --hard HEAD^**

## Undo – Remove a commit and retain changes to files

Instead of performing the hard reset and thereby losing all the changes the commit introduced, the reset can be performed so that the changes are retained in the working directory

Now, we'll undo the commit and retain the changes introduced to the working tree:

**$ git reset --mixed HEAD^**

We can see that our commit has been undone, but the changes to the file are preserved in the working tree, so more work can be done in order to create a proper commit.

The staging area is reset, but the working tree is kept as it was before the reset, so the files affected by the undone commit will be in a modified state.

*The --mixed option is the default behavior of git reset, so it can be omitted: git reset HEAD^*

## Undo – Remove a commit and retain changes in the staging area

Of course, it is also possible to undo the commit, but keep the changes to the files in the index or the staging area so that you are ready to recreate the commit with, for example, some minor modifications.

**$ git reset --soft HEAD^**

You can now make minor (or major) changes to the files you need, add them to the staging area, and create a new commit

Git will reset the branch pointer and **HEAD** to point to the previouscommit. However, with the --soft option, the index and working directories are not reset, that is, they have the same state as they had before we created the now undone commit

## Undo – Working with a dirty area

Git provides a smart way to quickly put stuff away so that it can be retrieved later using the git stash command

If you have changes and you want to keep them, you can

stash them away before undoing the commit and retrieve them afterward. Git provides a stash command that can put unfinished changes away, so it is easy to make quick context switches without losing work.

**$ git stash**

**$ git stash pop**

So, the file is back to the state it was in before the reset, and we got rid of the unwanted commit.

The stash command works by saving the current state of your

working directory and the staging area. Then, it reverts your working

directory to a clean state.

## Redo – Recreate the latest commit with new changes

This is quite useful if you've just created a commit, but have perhaps forgotten to add a necessary file to the staging area before you committed, or if you need to reword the commit message.

**$ git commit --amend**

The --amend option to git commit is roughly equivalent to performing git reset --soft HEAD^, followed by fixing the files needed and adding those to the staging area. Then, we will run git commit reusing the commit message from the previous commit (git commit -c ORIG\_HEAD).

We can also use the --amend method to add missing files to our latest commit.

Now, you can amend the latest commit with git commit --amend. The command will include files in the index in the new commit and you can, as with the last example, reword the commit message if needed. It is not needed in this example, so we'll pass the --no-edit option to the command:

$ git commit --amend --no-edit

*You can also reset the author information (name, email, and timestamp) with the*

*commit --amend command. Just pass along the --reset-author option and Git will create a*

*new timestamp and read author information from the configuration or environment,*

*instead of the using information from the old commit object*

## Revert – Undo the changes introduced by a commit

Revert can be used to undo a commit in history that has already been published (pushed), whereas this can't be done with the amend or reset options without rewriting history. Revert works by applying the anti-patch introduced by the commit in question

Revert works by applying the anti-patch introduced by the commit in question. A revert will, by default, create a new commit in history with a commit message that describes which commit has been reverted.

The git revert command applies the anti-patch of the commit in question to the current HEAD pointer. It will generate a new commit with the anti-patch and a commit message that describes the reverted commit(s).

It's possible to revert more than one commit in a single revert, for example, git revert

master~6..master~2 will revert the commits from the sixth commit from the bottom in the master to the third commit from the bottom in the master (both included).

It is also possible not to create a commit while reverting; passing the -n option to git revert will apply the needed patched, but only to the working tree and the staging area.

## Reverting a merge

Merge commits are a special case when it comes to revert. In order to be able to revert a merge commit, you'll have to specify which parent side of the merge you want to keep.

However, when you revert a merge commit, you should keep in mind that though reverting will undo changes to files, it doesn't undo history. This means that when you revert a merge commit, you declare that you will not have any of the changes introduced by the

merge in the target branch.

The effect of this is that the subsequent merges from the other branch will only bring in changes of commits that are not ancestors of the reverted merge commit.



Revert the merge, keeping the history of the first parent:

**$ git revert -m 1 5ae3beb**

The revert command will take the patches introduced by the commit you want to revert and apply the reverse/anti-patch to the working tree. If all goes well, that is, there are no conflicts, a new commit will be made.

While reverting a merge commit, only the changes introduced in the mainline (the -m option) will be kept, and all the changes introduced in the other side of the merge will be reverted.

Though it is easy to revert a merge commit, you might run into issues if you later want to the branch again because the issues on the merge have not been fixed. While reverting the merge commit, you actually tell Git that you do not want any of the changes that the other branch introduced in this branch. So, when you try to merge in the branch again, you will only get the changes from the commits that are not ancestors of the reverted merge commit.

To perform a proper re-merge, we first have to revert the reverting merge commit; this might seem a bit weird, but it is the way to get the changes from before the revert back into our tree. Then, we can perform another merge of the branch, and we'll end up with all the changes introduced by the branch we're merging in.

## Viewing past Git actions with git reflog

The reflog command stores information on updates to the tip of the branches in Git, where the normal git log command shows the ancestry chain from HEAD, and the reflog command shows what HEAD has pointed to in the repository.

This is your history in the repository, which tells you how you have moved between branches, created your commits and resets, and so on.

This means that, by going through the reflog command, you can find lost commits that none of your branches or other commits point to. This makes the reflog command a good starting point for trying to find a lost commit.

From here, there are various ways to resurrect the changes. You can either check out the commit and create a branch; then, you'll have a pointer so that you can easily find it again. You can also

check out specific files from the commit with git checkout –path/to/file SHA-1, or you can use the git show or git cat-file commands to view the files.

For every movement of the HEAD pointer in the repository, Git stores the commit pointed to and the action for getting there. This can be commit, checkout, reset, revert, merge, rebase, and so on. The information is local to the repository and is not shared on pushes, fetches, and clones.

## Finding lost changes with git fsck

Another tool exists in Git that can help you find and recover lost commits and even blobs (files), which is git fsck. The fsck command tests the object database and verifies the SHA-1 ID of the objects and the connections they make. This command can also be used to find objects that are not reachable from any named reference, as it tests all the objects found in the database, which are in the .git/objects folder.

**$ git fsck –unreachable**

The git fsck command will test all the objects found in the .git/objects folder. When the --unreachable option is given, it will report the objects found that can't be reached from another reference; a reference can be a branch, a tag, a commit, a tree, the reflog, or changes that have been stashed away.

# Repository Maintenance

## Pruning remote branches

Usually, these feature branches are deleted in the main repository (the origin). However, branches are not automatically deleted from all clones while fetching and pulling request. Git must explicitly be told to delete branches from the local repository that have been deleted from the origin.

The branches are still there, even if they have been deleted in the

remote repository. We need to tell Git explicitly to delete the

branches that have also been deleted from the remote repository,

using the following command:

**$ git fetch –prune**

There are several ways to remove the branches from Git that have been deleted from the master. It can be done while updating the local repository, as we saw with git fetch --prune, and also with git pull --prune. It can even be performed with the git remote prune origin command. This will also remove the branches from Git that are no longer available on the remote, but it will not update remote-tracking branches in the repository.

## Running garbage collection manually

The garbage collection and packing of loose objects can also be triggered manually by executing the git gc command. Triggering git gc is useful if you have a lot of loose objects

we'll check the repository for unpacked objects; we can do this with the count-objects command:

**$ git count-objects**

We'll also check for unreachable objects, which are objects that can't be reached from any reference (tag, branch, or other object)

**$ git fsck –unreachable**

Let's try to trigger garbage collection manually

**$ git gc**

If we investigate the repository now, we will see the following:

**$ git count-objects**

The object count is smaller. Git has packed the objects to the pack file stored in the .git/objects/pack folder. The size of the repository is also

smaller, as Git compresses and optimizes the objects in the pack file. However, there are still some unreachable objects left. This is because objects will only be deleted if they are older than what is specified in the gc.pruneexpire configuration option, which defaults to two weeks (config value: 2.weeks.ago). We can override the default or configured

option by running the --prune=now option:

**$ git gc --prune=now**

Investigating the repository gives the following output:

**$ git count-objects**

**0 objects, 0 kilobytes**

The git gc command optimizes the repository by compressing file revisions and deleting objects that there are no references to.

## A quick "how-to" submodule

You want to keep projects separate, even though you need to use one project for the other. Git has a mechanism for this kind of project dependency, called submodules. The basic idea is that you can clone another Git repository into your project as a subdirectory, but keep the commits from the two repositories separate, as

shown in the following diagram:



We'll add a subproject, lib\_a, to the super project as a Git submodule:

**$ git submodule add**

.gitmodules is a regular file

The .gitmodules file, as above, contains information about all the

submodules registered in the repository. The lib\_a file stores which

commit the submodule's HEAD is pointing to when added to the super

project. Whenever the submodule is updated with new commits

(created locally or fetched), the super project will show the

submodule as having changed while running git status. If the

changes to the submodule can be accepted, the submodule revision in the super project is updated by adding the submodule file and

committing this to the super project.

**$ git submodule update**

Now we actually reset our submodule to the state described in

the file for that submodule

Notice that, by default, the submodule is in a detached head state,

which means that HEAD is pointing directly to a commit instead of a

branch. You can still edit the submodule and record commits; however, if you perform a submodule update in the super repository without first committing a new submodule state, your changes can be hard to find. Always remember to check out or create a branch while switching to a submodule to work on. If so, you can just check out the branch again and get your changes back. Since Git Version 1.8.2, it has been possible to make submodules track a branch rather than a single commit

To make Git track the branch of a submodule rather than a specific commit, we need to record the name of the branch we want to track. This is done in the .gitmodules file for the submodule; here, we'll use the stable branch:

**$ git config -f .gitmodules submodule.lib\_a.branch stable**

**$ cat .gitmodules**

**[submodule "lib\_a"]**

**path = lib\_a**

**url = https://github.com/PacktPublishing/Git-Version-Control-Cookbook-Second-Edition\_lib\_a.git**

**branch = stable**

**$ git submodule update –remote**

The submodule is still in the detached HEAD state. However, when updating the submodule with git submodule update --remote, changes from the submodule's remote repository will be fetched and the submodule will be updated to the latest commit on the branch it is tracking. We still need to record a commit to the super repository, specifying the state of the submodule.

When you are cloning a repository that contains one or more submodules, you need to explicitly fetch them after the clone

**$ git clone super super\_clone**

Now, initialize and update the submodules:

**$ cd super\_clone**

**$ git submodule init**

**$ git submodule update –remote**

The repository is ready for development!

*When cloning the repository, the submodules can be initialized and updated directly after the clone if the --recursive or --recurse-submodules option is given.*

## Subtree merging

An alternative to submodules is subtree merging. Subtree merging is a strategy that can be used when performing merges with Git. The strategy is useful when merging a branch (or, as we'll see in this recipe, another project) into a subdirectory of a Git repository instead of the root directory. When using the subtree merge strategy, the history of the subproject is joined with the history of the super project, while the subproject's history can be kept clean, except for commits intended to go upstream.

# Patching and Offline Sharing

## Creating archives from a tree

specified by a particular commit, but without the corresponding history. This can, of course, be done by checking the particular commit followed by deleting/omitting the .git folder when creating an archive. But with Git, there is a better way to do this, which is built in so it is possible to create an archive from a particular commit or reference. When using Git to create the archive, you also make sure that the archive only contains the files tracked by Git and not any untracked files or folders you might have in your working directory.

**$ git archive --prefix=offline/ -o offline.zip origin/master**

The --prefix option prepends the specified prefix to each file in the archive, effectively adding an offline directory as a root directory for the files in the repository, and the -o option tells Git to create the archive in the offline.zip file, which of course, is compressed in the ZIP format. We can investigate the ZIP archive to check whether the files contain the following:

The archive command can also be used to create an archive for a

subdirectory of the repository. We can use this on the doc branch of the repository to ZIP the content of the Documentation folder:

**$ git archive --prefix=docs/ -o docs.zip origin/doc:Documentation**

There are other format options besides the ZIP format for the archive, for example, tar, tar.gz, and so on. The format can be specified with the --format=<format> option or as a suffix to the output filename with the –o option. The following two commands will produce the same output file:

**$ git archive --format=tar.gz HEAD > offline.tar.gz**

**$ git archive -o offline.tar.gz HEAD**

The Git archive command behaves a bit differently if a commit/tag ID or a tree ID is passed as an identifier. If a commit or tag ID is given, the ID will be stored in a global extended pax header for the TAR format, and as a file comment for the ZIP format. If only the tree ID is given, no extra information will be stored. You can actually see this in the previous examples, where the first ID was given a branch as a reference. As the branch points to a commit, the ID of this commit was written as a comment on the file and we can actually see it in the output of the archive listing:

asaki@WW7FRDEV01ASI MINGW64 /d/Work/MARSNG/source (master)

$ unzip -l develop.zip

Archive: develop.zip

1bcb72474559f20dac51dc4fed23ac59d429050c

In the second example, we also passed a branch as a reference, but furthermore, we specified the Documentation folder as the subfolder we wanted to create an archive from. This corresponds to passing the ID of the tree to the archive command; hence, no extra information will be stored in the archive.

# Tips and Tricks

## Using git stash

With the git stash command, you can save the state of your current working directory with/without a staging area and restore the working tree to a clean state

We can see that we have one file added to the staging area, foo, one

modified file, bar, and an untracked file in the work area as well, new\_file.

The basic command will put away changes from the staging area and changes made to tracked files. It leaves untracked files in the working directory:

**$ git stash**

**$ git stash pop**

Now, the stashed changes are available again in the working repository and the stash entry has been deleted. Note that the changes are applied only to the working directory, although one of the files was staged when we created the stash.

We have created two commits: one for the index and one for the work area. In gitk, we can see the commits that stash creates to put the changes away (gitk stash)

We can also see the state of the branches after we created the commit (gitk --reflog), as shown in the following screenshot:

Git actually creates two commits under the refs/stash namespace. One

commit contains the contents of the staging area. This commit is called index on master. The other commit is the work in progress in the working directory, WIP on master.

When Git puts away changes by creating commits, it can use its normal resolution methods to apply the stashed changes back to the working directory. This means that if a conflict arises when applying the stash, you need it to be solved in the usual way.

In the preceding example, we saw only the very basic usage of the stash command, putting away changes to untracked files and changes added to the staging area. It is also possible to include untracked files in the stash command. This can be done with the --include-untracked option. We can add foo to the staging area; firstly, to have the same state as when we created the stash earlier and then to create a stash that includes untracked files:

**$ git stash --include-untracked**

Now, we can see that new\_file has disappeared from the working directory. It's included in the stash, and we can check this with Gitk. It will show up as another commit of untracked files:

**$ gitk master stash**

Gitk shows the stash with its untracked files

We can also make sure that the changes we added to the staging area are added back to the staging area after we apply the stash, so we end up with the exact same state as before we stashed our changes away:

**$ git stash pop –index**

It's also possible to put away only the changes in the working directory, while keeping the changes in the staging area. We can do this either for only the tracked files or by stashing away untracked files ( --includeuntracked), as follows:

**$ git stash --keep-index --include-untracked**

## Saving and applying stashes

However, the default names for stashed away changes aren't always helpful. In this example, we'll see how we can save stashes and name them so that it is easy to identify them again when listing the content of the stash. We'll also learn how to apply a stash without deleting it from the stash list.

To save the current state to a stash with a description we can remember at a later point in time, use the following command:

**$ git stash save 'Updates to foo'**

We can apply the stashes back to the working tree (and staging area with the --index option) without deleting them from the stash list:

**$ git stash apply 'stash@{1}'**

For stashes applied with git stash apply, the stash needs to be deleted with git stash drop:

**$ git stash drop 'stash@{1}'**

Keeping the stashes in the stash list by using stash apply and explicitly

deleting them with git stash drop has some advantages over just using stash pop. When using the pop option, the stashes in the list are automatically deleted if they can be successfully applied. But if it fails and triggers the conflict resolution mode, the stash applied is not dropped from the list and continues to exist on the stash stack. This might later lead to accidentally using the wrong stash because it was thought to have been removed. By consistently using git stash apply and git stash drop, you can avoid this scenario.

*The git stash command can also be used to apply debug information to an application. Let's pretend you have been bug hunting and have added a lot of debug statements to your code in order to track down a bug. Instead of deleting all those debug statements, you can save them as a Git stash:*

*$ git stash save "Debug info stash"*

*Then, if you need debug statements later, you can just apply the stash and you'll be ready to debug.*

## Using the blame command

The bisect command is good when you don't know where in your code

there is a bug, but you can test for it and thereby find the commit that introduced it. If you already know where in the code the bug is but want to find the commit that introduced it, you can use git blame. The blame command will annotate every line in the file with the commit that most recently touched that line, making it easy to find the commit ID and then the full context of the commit.

To annotate each line in the file with the commit ID and author, we'll run git blame on the file. We can further limit the search to specific lines with the -L <from>,<to> option.

