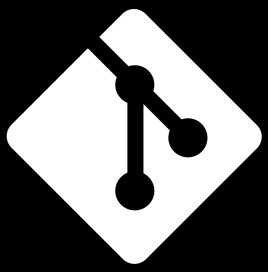


git by example

Anton Zhiyanov

Table of contents

4. Undo

5. Advanced

Preface

Git is *the* distributed version control system used in software development today. It's very powerful, but also known for its not-so-obvious syntax.

I got tired of googling the same Git commands over and over again. So I created this step-by-step guide to Git operations, from basic to advanced. You can read it from start to finish to (hopefully) learn more about Git, or jump to a specific use case that interests you.

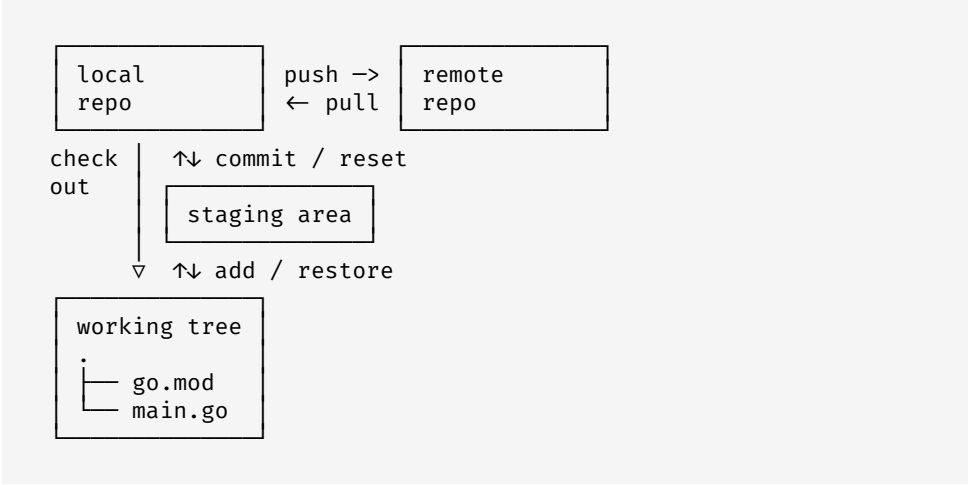
The guide is available in the following formats:

- Interactive online cookbook
- Bookmarkable playground
- PDF minibook

Concepts

This is the only piece of theory in the guide. I'll keep it short and simplified to the π == 3 level. Please don't judge me if you're a Git master.

Working tree, staging area, repository



A *working tree* is the slice of the project at any given moment (usually it's the current moment). When you add or edit code, you change the working tree.

A *staging area* is where you stage the changes from the working tree before making them permanent.

A *repo* (repository) is the collection of permanent changes (*commits*) made throughout the history of the project. Typically, there is a single *remote* repo (managed by GitHub/GitLab/etc) and many *local* repos — one for each developer involved in a project.

When you make a change in the staging area permanent, it is removed from the staging area and *committed* to the local repo. A commit is the permanent record of that change. The repo contains all the commits that have been made.

When you *checkout* a specific commit, the working tree is updated to reflect the project state at the time of that commit.

Local and remote repos are frequently synchronized so that all repos contain all commits from all developers.

Branch, tag, HEAD

A *branch* is an alternate version of the project reality. Typically, there is a main branch, and separate branches for features under development. When work on a feature branch is complete, it is merged into the main branch (or discarded).

A *tag* is a named state of the project. Typically, tags are created on the main branch for important milestones such as releases.

The currently checked-out commit (usually the latest commit in a branch) is referenced as *HEAD*.

Now that the boring stuff is out of the way, let's get to the recipes!

1. Basics

Let's start with basic Git operations on a local repo.

Init repo

Create an empty repo:

```
git init
```

```
Initialized empty Git repository in /tmp/repo/.git/
```

Set user name and email for the repo (they are required):

```
git config user.email alice@example.com
git config user.name "Alice Zakas"
```

Use the `--global` flag to set the name and email at the OS user level instead of the repo level.

Show user and repo configs:

```
git config --list --show-origin
```

```
file:/sandbox/.gitconfig    user.email=sandbox@example.com
file:/sandbox/.gitconfig    user.name=sandbox
file:/sandbox/.gitconfig    init.defaultbranch=main
file:.git/config    core.repositoryformatversion=0
file:.git/config    core.filemode=true
file:.git/config    core.bare=false
file:.git/config    core.logallrefupdates=true
file:.git/config    user.email=alice@example.com
file:.git/config    user.name=Alice Zakas
```

Add file

Create a file and add it to the staging area:

```
echo "git is awesom" > message.txt
git add message.txt
```

View changes in the staging area:

```
git diff --cached
```

```
diff --git a/message.txt b/message.txt
new file mode 100644
index 0000000..0165e86
--- /dev/null
+++ b/message.txt
@@ -0,0 +1 @@
+git is awesom
```

Commit to the local repo:

```
git commit -m "add message"
```

```
[main (root-commit) 3a2bd8f] add message
 1 file changed, 1 insertion(+)
 create mode 100644 message.txt
```

Edit file

Edit the previously committed file:

```
echo "git is awesome" > message.txt
```

View local changes:

```
git diff
```

```
diff --git a/message.txt b/message.txt
index 0165e86..118f108 100644
--- a/message.txt
+++ b/message.txt
@@ -1 +1 @@
-git is awesom
+git is awesome
```

Add modified files and commit in one command:

```
git commit -am "edit message"
```

```
[main ecdeb79] edit message
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Note that `-a` does not add new files, only changes to the already committed files.

Rename file

Rename the previously committed file:

```
git mv message.txt praise.txt
```

The change is already in the staging area, so `git diff` won't show it. Use `--cached`:

```
git diff --cached
```

```
diff --git a/message.txt b/praise.txt
similarity index 100%
rename from message.txt
rename to praise.txt
```

Commit the change:

```
git commit -m "rename message.txt"
```

```
[main d768287] rename message.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename message.txt ⇒ praise.txt (100%)
```

Delete file

Delete the previously committed file:

```
git rm message.txt
```

```
rm 'message.txt'
```

The change is already in the staging area, so `git diff` won't show it. Use `--cached`:

```
git diff --cached
```

```
diff --git a/message.txt b/message.txt
deleted file mode 100644
index 0165e86..0000000
--- a/message.txt
+++ /dev/null
@@ -1 +0,0 @@
-git is awesom
```

Commit the change:

```
git commit -m "delete message.txt"
```

```
[main 6a2d19b] delete message.txt
 1 file changed, 1 deletion(-)
 delete mode 100644 message.txt
```

Show current status

Edit the previously committed file and add the changes to the staging area:

```
echo "git is awesome" > message.txt
git add message.txt
```

Create a new file:

```
echo "git is great" > praise.txt
```

Show the working tree status:

```
git status
```

```
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   message.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    praise.txt
```

Note that `message.txt` is in the staging area, while `praise.txt` is not tracked.

Show commit log

Show commits:

```
git log
```

```
commit ecdeb79aad4565d8d7d725678ffadc48b3cdec52
Author: sandbox <sandbox@example.com>
Date:   Thu Mar 14 15:00:00 2024 +0000

    edit message

commit 3a2bd8f0929c605193120bd1ad12732f49457e99
Author: sandbox <sandbox@example.com>
Date:   Thu Mar 14 15:00:00 2024 +0000

    add message
```

Show only the commit message and the short hash:

```
git log --oneline
```

```
ecdeb79 edit message
3a2bd8f add message
```

Show commits as an ASCII graph:

```
git log --graph
```

```
* commit ecdeb79aad4565d8d7d725678ffadc48b3cdec52
| Author: sandbox <sandbox@example.com>
| Date:   Thu Mar 14 15:00:00 2024 +0000
|
|     edit message
|
* commit 3a2bd8f0929c605193120bd1ad12732f49457e99
  Author: sandbox <sandbox@example.com>
  Date:   Thu Mar 14 15:00:00 2024 +0000

      add message
```

Show compact ASCII graph:

```
git log --oneline --graph
```

```
* ecdeb79 edit message
* 3a2bd8f add message
```

Show specific commit

Show the last commit contents:

```
git show HEAD
```

```
commit ecdeb79aad4565d8d7d725678ffadc48b3cdec52
Author: sandbox <sandbox@example.com>
Date:   Thu Mar 14 15:00:00 2024 +0000

    edit message

diff --git a/message.txt b/message.txt
index 0165e86..118f108 100644
--- a/message.txt
+++ b/message.txt
@@ -1 +1 @@
-git is awesom
+git is awesome
```

Show the second-to-last commit:

```
git show HEAD~1
```

```
commit 3a2bd8f0929c605193120bd1ad12732f49457e99
Author: sandbox <sandbox@example.com>
Date:   Thu Mar 14 15:00:00 2024 +0000

    add message

diff --git a/message.txt b/message.txt
new file mode 100644
index 0000000..0165e86
--- /dev/null
+++ b/message.txt
...
```

Use `HEAD~n` to show the nth-before-last commit or use the specific commit hash instead of `HEAD~n`.

Search repo

There are 3 commits, each adding a new line to `message.txt`:

```
git log --oneline
```

```
cc5b883 no debates
2774a8b is great
31abe57 is awesome
```

The current `message.txt` state:

```
cat message.txt
```

```
git is awesome
git is great
there is nothing to debate
```

Search in working tree (current state):

```
git grep "debate"
```

```
message.txt:there is nothing to debate
```

Search the project as of the second-to-last commit:

```
git grep "great" HEAD~1
```

```
HEAD~1:message.txt:git is great
```

You can use the specific commit hash instead of `HEAD~n`.

2. Branch and merge

Let’s dive into the wondrous world of merging.

Branch

Show branches (there is only `main` now):

```
git branch
```

```
* main
```

Create and switch to a new branch:

```
git branch ohmypy
git switch ohmypy
```

```
Switched to branch 'ohmypy'
```

Show branches (the current one is `ohmypy`):

```
git branch
```

```
  main
* ohmypy
```

Add and commit a file:

```
echo "print('git is awesome')" > ohmy.py
git add ohmy.py
git commit -m "ohmy.py"
```

```
[ohmypy a715138] ohmy.py
 1 file changed, 1 insertion(+)
 create mode 100644 ohmy.py
```

Show only commits from the `ohmypy` branch:

```
git log --oneline main..ohmypy
```

```
a715138 ohmy.py
```

Merge

Show commits from all branches (two commits in `main`, one in `ohmypy`):

```
git log --all --oneline --graph
```

```
* ecdeb79 edit message
| * a715138 ohmy.py
|/
* 3a2bd8f add message
```

We are now on the `main` branch, let's merge the `ohmypy` branch back into main:

```
git merge ohmypy
```

```
Merge made by the 'ort' strategy.
 ohmy.py | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 ohmy.py
```

There are no conflicts, so git commits automatically. Show the new commit history:

```
git log --all --oneline --graph
```

```
*   7d5ac4f Merge branch 'ohmypy'
|\
| * a715138 ohmy.py
* | ecdeb79 edit message
|/
* 3a2bd8f add message
```

Rebase

Show commits from all branches (two commits in `main`, one in `ohmypy`):

```
git log --all --oneline --graph
```

```
* ecdeb79 edit message
| * a715138 ohmy.py
|/
* 3a2bd8f add message
```

We are now on the `main` branch, let's rebase the `ohmypy` branch back into main:

```
git rebase ohmypy
```

```
Rebasing (1/1)
Successfully rebased and updated refs/heads/main.
```

Note that the new commit history is linear, unlike when we do a `git merge ohmypy`:

```
git log --all --oneline --graph
```

```
* c2b0c60 edit message
* a715138 ohmy.py
* 3a2bd8f add message
```

Rebasing rewrites history. So it's better not to rebase branches that have already been pushed to remote.

Squash

Show commits from all branches (two commits in `main`, three in `ohmypy`):

```
git log --all --oneline --graph
```

```
* ecdeb79 edit message
| * b9a7d0f ohmy.lua
| * 5ca4d55 ohmy.sh
| * a715138 ohmy.py
|/
* 3a2bd8f add message
```

If we do `git merge ohmypy` to merge the `ohmypy` branch into `main`, the main branch will receive all three commits from ohmypy.

Sometimes we prefer to “squash” all the branch commits into a single commit, and then merge it into main. Let’s do it.

Switch to the `ohmypy` branch:

```
git switch ohmypy
```

```
Switched to branch 'ohmypy'
```

Combine all `ohmypy` changes into a single commit in the working directory:

```
git merge --squash main
```

```
Squash commit -- not updating HEAD
```

Commit the combined changes:

```
git commit -m "ohmy[py,sh,lua]"
```

```
[ohmypy 4f2a17f] ohmy[py,sh,lua]
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Switch back to the `main` branch:

```
git switch main
```

Merge the `ohmypy` branch into `main`:

```
git merge --no-ff ohmypy -m "ohmy[py,sh,lua]"
```

```
Merge made by the 'ort' strategy.
 ohmy.lua | 1 +
 ohmy.py  | 1 +
 ohmy.sh  | 1 +
 3 files changed, 3 insertions(+)
  ...
```

Note the single commit in `main` made of three commits in `ohmypy`:

```
git log --all --oneline --graph
```

```
*   008dce6 ohmy[py,sh,lua]
|\
| * 4f2a17f ohmy[py,sh,lua]
| * b9a7d0f ohmy.lua
| * 5ca4d55 ohmy.sh
| * a715138 ohmy.py
* | ecdeb79 edit message
|/
* 3a2bd8f add message
```

Cherry-pick

I have a typo in `message.txt`:

```
cat message.txt
```

```
git is awesom
```

And I accidentally fixed it in the `ohmypy` branch instead of `main`:

```
git log --all --oneline --graph --decorate
```

```
* 568193c (HEAD → main) add praise
| * bbce161 (ohmypy) ohmy.sh
| * cbb09c6 fix typo
| * a715138 ohmy.py
|/
* 3a2bd8f add message
```

I'm not ready to merge the entire `ohmypy` branch, so I will cherry-pick the commit:

```
git cherry-pick cbb09c6
```

```
[main b23d3ee] fix typo
 Date: Thu Mar 14 15:00:00 2024 +0000
 1 file changed, 1 insertion(+), 1 deletion(-)
```

`cherry-pick` applied the comment to the `main` branch:

```
git log --all --oneline --graph --decorate
```

```
* b23d3ee (HEAD → main) fix typo
* 568193c add praise
| * bbce161 (ohmypy) ohmy.sh
| * cbb09c6 fix typo
| * a715138 ohmy.py
|/
* 3a2bd8f add message
```

The typo is fixed:

```
cat message.txt
```

```
git is awesome
```

3. Local and remote

Working with a local repo is fun, but adding a remote repo is even funnier.

Push

Alice wants to clone our repo and make some changes.

Clone the remote repo:

```
git clone https://example.org/remote.git /tmp/alice
```

```
Cloning into '/tmp/alice'...
done.
```

Set user name and email:

```
cd /tmp/alice
git config user.email alice@example.com
git config user.name "Alice Zakas"
```

Make some changes and commit:

```
echo "Git is awesome!" > message.txt
git commit -am "edit from alice"
```

```
[main b9714f2] edit from alice
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Push locally committed changes to the remote repo:

```
git push
```

Pull

I want to pull Alice's changes to the local repo.

No commits from Alice yet:

```
git log --oneline
```

```
3a2bd8f add message
```

Pull the latest changes from the remote repo:

```
git pull
```

```
Updating 3a2bd8f..b9714f2
Fast-forward
 message.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

The local repo now contains commits from Alice:

```
git log --oneline
```

```
b9714f2 edit from alice
3a2bd8f add message
```

Resolve conflict

I have a local commit (not yet pushed to the remote) that conflicts with Alice's changes (already pushed to the remote), so I need to resolve it.

Pull the changes from the remote repo:

```
git pull
```

```
Auto-merging message.txt
CONFLICT (content): Merge conflict in message.txt
Automatic merge failed; fix conflicts and then commit the result.
   3a2bd8f..b9714f2  main       → origin/main (exit status 1)
```

There is a conflict in `message.txt`! Let's show it:

```
cat message.txt
```

```
<<<<<<< HEAD
git is awesome
=======
Git is awesome!
>>>>>>> b9714f2c59c7dbd1205cf20e0a99939b7a686d97
```

I like Alice's version better, so let's choose it:

```
git checkout --theirs -- message.txt
# to choose our version, use --ours
```

Add the resolved file to the staging area and complete the merge:

```
git add message.txt
git commit -m "merge alice"
```

```
[main cbb6112] merge alice
```

Push branch

Create the local `ohmypy` branch:

```
git branch ohmypy
git switch ohmypy
```

```
Switched to branch 'ohmypy'
```

Add and commit a file:

```
echo "print('git is awesome')" > ohmy.py
git add ohmy.py
git commit -m "ohmy.py"
```

```
[ohmypy c64073e] ohmy.py
 1 file changed, 1 insertion(+)
 create mode 100644 ohmy.py
```

Push the local branch to remote:

```
git push -u origin ohmypy
```

```
branch 'ohmypy' set up to track 'origin/ohmypy'.
```

Show both local and remote branches:

```
git branch --all
```

```
  main
* ohmypy
  remotes/origin/main
  remotes/origin/ohmypy
```

Fetch branch

Fetch remote branches:

```
git fetch
```

Remote has the `ohmypy` branch, but it's not checked out locally:

```
git branch
```

```
* main
```

Checkout the `ohmypy` branch:

```
git switch ohmypy
# or: git checkout ohmypy
```

```
branch 'ohmypy' set up to track 'origin/ohmypy'.
```

Show branches:

```
git branch
```

```
  main
* ohmypy
```

Tags

Create a tag for the latest commit:

```
git tag 0.1.0 HEAD
```

Create a tag for the nth-before-last commit:

```
git tag 0.1.0-alpha HEAD~1
```

You can use the commit hash instead of `HEAD~n`.

Show tags:

```
git tag -l
```

```
0.1.0
0.1.0-alpha
```

Show compact log with tags:

```
git log --decorate --oneline
```

```
ecdeb79 (HEAD → main, tag: 0.1.0, origin/main) edit message
3a2bd8f (tag: 0.1.0-alpha) add message
```

Delete tag:

```
git tag -d 0.1.0-alpha
```

```
Deleted tag '0.1.0-alpha' (was 3a2bd8f)
```

Push tags to the remote:

```
git push --tags
```

4. Undo

“Damn, how do I undo what I just did?” — is the eternal Git question. Let’s answer it once and for all.

Amend commit

Edit a file and commit:

```
echo "git is awesome" > message.txt
git commit -am "edit nessage"
```

```
[main c0206a0] edit nessage
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Show commits:

```
git log --oneline
```

```
c0206a0 edit nessage
3a2bd8f add message
```

I made a typo, so I want to change the commit message:

```
git commit --amend -m "edit message"
```

```
[main ecdeb79] edit message
 Date: Thu Mar 14 15:00:00 2024 +0000
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Git has replaced the last commit:

```
git log --oneline
```

```
ecdeb79 edit message
3a2bd8f add message
```

To change the commit message for one of the last `n` commits, use `git rebase -i HEAD~n` (interactive) and follow the instructions on the screen.

Amend only works if the commit has not yet been pushed to the remote repo!

Undo uncommitted changes

Edit the previously committed file and add the changes to the staging area:

```
echo "git is awesome" > message.txt
git add message.txt
```

Show the working tree status:

```
git status
```

```
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   message.txt
```

Remove the changes from the staging area:

```
git restore --staged message.txt
```

The local file is still modified, but it's not staged for commit:

```
git status
```

```
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in work dir)
    modified:   message.txt

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

Now let’s discard the changes altogether:

```
git restore message.txt
# or: git checkout message.txt
```

Show the file contents:

```
cat message.txt
```

```
git is awesom
```

The changes are gone.

Undo local commit

I changed my mind about the last commit and I want to undo it.

Show commits:

```
git log --oneline
```

```
ecdeb79 edit message
3a2bd8f add message
```

Undo the last one:

```
git reset --soft HEAD~
```

The commit is gone:

```
git log --oneline
```

```
3a2bd8f add message
```

But the changes are still in the staged area:

```
git status
```

```
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   message.txt
```

To remove both the commit and the local changes, use `--hard` instead of `--soft`:

```
git reset --hard HEAD~
git status
```

```
HEAD is now at 3a2bd8f add message
On branch main
nothing to commit, working tree clean
```

Reset only works if the commit has not yet been pushed to the remote repo!

Undo remote commit

I changed my mind about the last commit and I want to undo it, but the commit is already pushed to the remote repo.

Show commits:

```
git log --oneline
```

```
ecdeb79 edit message
3a2bd8f add message
```

Undo the last one:

```
git revert HEAD --no-edit
```

```
[main 9ffa044] Revert "edit message"
 Date: Thu Mar 14 15:00:00 2024 +0000
 1 file changed, 1 insertion(+), 1 deletion(-)
```

You can revert to nth-before-last commit by using `HEAD~n` or use the specific commit hash instead of `HEAD~n`.

Since the commit has already been pushed, git can't delete it. Instead it creates an "undo" commit:

```
git log --oneline
```

```
9ffa044 Revert "edit message"
ecdeb79 edit message
3a2bd8f add message
```

Push the “undo” commit to the remote:

```
git push
```

Rewind history

Show commits:

```
git log --oneline --graph
```

```
*   7d5ac4f Merge branch 'ohmypy'
|\
| * a715138 ohmy.py
* | ecdeb79 edit message
|/
* 3a2bd8f add message
```

Show all repo states in reverse chronological order:

```
git reflog
```

```
7d5ac4f HEAD@{0}: merge ohmypy: Merge made by the 'ort' strategy.
ecdeb79 HEAD@{1}: commit: edit message
3a2bd8f HEAD@{2}: checkout: moving from ohmypy to main
a715138 HEAD@{3}: commit: ohmy.py
3a2bd8f HEAD@{4}: checkout: moving from main to ohmypy
3a2bd8f HEAD@{5}: commit (initial): add message
```

Suppose I want to go back to `HEAD@{3}`:

```
git reset --hard HEAD@{3}
```

```
HEAD is now at a715138 ohmy.py
```

This resets the entire repo and the working tree to the moment of `HEAD@{3}`:

```
git log --oneline --graph
```

```
* a715138 ohmy.py
* 3a2bd8f add message
```

Stash changes

Edit the previously committed file:

```
echo "git is awesome" > message.txt
git add message.txt
```

Let’s say we need to switch to another branch, but we don’t want to commit the changes yet.

Stash the local changes (i.e. save them in “drafts”):

```
git stash
```

```
Saved working directory and index state WIP on main:
3a2bd8f add message
```

Stash is a stack, so you can push multiple changes onto it:

```
echo "Git is awesome!" > message.txt
git stash
```

```
Saved working directory and index state WIP on main:
3a2bd8f add message
```

Show stash contents:

```
git stash list
```

```
stash@{0}: WIP on main: 3a2bd8f add message
stash@{1}: WIP on main: 3a2bd8f add message
```

Now we can switch to another branch and do something:

```
...(omitted for brevity)...
```

Switch back to the main branch and re-apply the latest changes from the stash:

```
git switch main
git stash pop
```

```
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in work dir)
    modified:   message.txt

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (96af1a51462f29d7b947a7563938847efd5d5aeb)
```

`pop` returns changes from the stack in “last in, first out” order.

Clear the stash:

```
git stash clear
```

5. Advanced stuff

While git gurus probably know all about these features, most developers have never heard of them. Let's fix that.

Log summary

Since the 1.0 release (tag `v1.0`), we have 6 commits from 3 contributors:

```
git log --pretty=format:'%h %an %s %d'
```

```
7611979 bob ohmy.lua  (HEAD → main, origin/main)
ef4f23e bob ohmy.sh
3d8f700 bob ohmy.py
c61962c alice no debates
2ab82f6 alice go is great
ecdeb79 sandbox edit message
3a2bd8f sandbox add message  (tag: v1.0)
```

Note the `--pretty` option which customizes the log fields:

```
%h   commit hash
%an  author
%s   message
%d   decoration (e.g. branch name or tag)
```

List the commits grouped by contributors:

```
git shortlog v1.0..
```

```
alice (2):
      go is great
      no debates

bob (3):
      ohmy.py
      ohmy.sh
      ohmy.lua

sandbox (1):
      edit message
```

A couple of useful options:

- `-n` (`--numbered`) sorts the output by descending number of commits per contributor.
- `-s` (`--summary`) omits commit descriptions and prints only counts.

List contributors along with the number of commits they have authored:

```
git shortlog -ns v1.0..
```

```
3   bob
2   alice
1   sandbox
```

Worktree

I'm in the middle of something important in the `ohmypy` branch:

```
echo "-- pwd --"
pwd
echo "-- branches --"
git branch
echo "-- status --"
git status
```

```
-- pwd --
/tmp/repo
-- branches --
  main
* ohmypy
-- status --
On branch ohmypy
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   ohmy.py
```

Suddenly I need to fix an annoying typo in the `main` branch. I can stash the local changes with `git stash`, or I can checkout multiple branches at the same time with `git worktree`.

Checkout the main branch into `/tmp/hotfix`:

```
git worktree add -b hotfix /tmp/hotfix main
```

```
HEAD is now at 3a2bd8f add message
```

Fix the typo and commit:

```
cd /tmp/hotfix
echo "git is awesome" > message.txt
git commit -am "fix typo"
```

```
[hotfix c3485cd] fix typo
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Push to remote main:

```
git push --set-upstream origin main
```

```
branch 'main' set up to track 'origin/main'.
```

Now I can return to `/tmp/repo` and continue working on the `ohmypy` branch.

Bisect

I have 5 poorly named commits:

```
git log --oneline
```

```
2f568eb main.sh
31ed915 main.sh
f8b2baf main.sh
5e0cf35 main.sh
8f0f1e4 main.sh
51c34ff test.sh
```

And a failing test:

```
sh test.sh
```

```
FAIL (exit status 1)
```

I will use the bisection algorithm to find the commit that introduced the bug:

```
git bisect start
```

```
status: waiting for both good and bad commits
```

The current state is obviously buggy, but I'm pretty sure the first "main.sh" commit was good:

```
git bisect bad HEAD
git bisect good HEAD~4
```

```
status: waiting for good commit(s), bad commit known
Bisecting: 1 revision left to test after this (roughly 1 step)
[f8b2baf93964ec9e0daa87c9ed262bbf5cf66b67] main.sh
```

Git has automatically checked out the middle commit. Let's test it:

```
sh test.sh
```

```
PASS
```

The test passes. Mark the commit as good:

```
git bisect good
```

```
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[31ed915660c42a00aa30b51520a16e3f48201299] main.sh
```

Git has automatically checked out the middle commit. Let's test it:

```
sh test.sh
```

```
FAIL (exit status 1)
```

The test fails. Show the commit details:

```
git show
```

```
commit 31ed915660c42a00aa30b51520a16e3f48201299
Author: sandbox <sandbox@example.com>
Date:   Thu Mar 14 15:00:00 2024 +0000

    main.sh

diff --git a/main.sh b/main.sh
index 7f8f78c..ce533e0 100644
--- a/main.sh
+++ b/main.sh
@@ -1,2 +1,2 @@
 # sum two numbers
-echo $(expr $1 + $2)
+echo $(expr $1 - $2)
```

This is the commit that introduced the bug (subtraction instead of addition)!

Partial checkout

The remote repo looks like this:

```
.
├── go.mod
├── main.go
├── products
│   └── products.go
└── users
    └── users.go
```

We will selectively checkout only some of the directories.

Clone the repo, but do not checkout the working tree:

```
git clone --no-checkout /tmp/remote.git /tmp/repo
cd /tmp/repo
```

```
Cloning into '/tmp/repo'...
done.
```

Tell git to checkout only the root and `users` directories:

```
git sparse-checkout init --cone
git sparse-checkout set users
```

Checkout the directories:

```
git checkout main
```

```
Your branch is up to date with 'origin/main'.
```

Only the root and users directories are checked out:

```
tree
```

```
.
├── go.mod
├── main.go
└── users
    └── users.go

1 directories, 3 files
```

The `products` directory was not checked out.

Partial clone

The partial checkout approach we tried earlier still clones the entire repo. So if the repo itself is huge (which is often the case if it has a long history or large binary files), the clone step can be slow and traffic-intensive.

To reduce the amount of data downloaded during cloning, use *partial clone* with one of the following commands:

```
# Download commits and trees (directories),
# but not blobs (file contents):
git clone --filter=blob:none file:///tmp/remote.git

# Download commits only, without trees (directories)
# or blobs (file contents):
git clone --filter=tree:0 file:///tmp/remote.git
```

In both cases, git will lazily fetch the missing data later when needed.

Note that for this to work, the remote server should support partial cloning (GitHub does).

Final thoughts

We've covered important Git operations, from basic editing to branching and merging, remote syncing, undoing changes, and performing some moderate magic. I hope Git incantations make a little more sense to you now than they did before.

If you find this book useful — please spread the word and subscribe to my other projects at antonz.org/subscribe.

Now go and put your knowledge to good use. And may Git be with you!