Git in Action*

Joshua Gloor

August 12, 2024

*Some parts are inspired by [CS14]. Consequently, to contribute to the community, all content is licensed under the Creative Commons Attribution Non Commercial Share Alike 3.0 license. We used Git version 2.43.0 for all sessions.

Session 01: Basics

1 Status of Files

We will use a local git repository to demonstrate the examples in this session. We can create a local git repository in the current directory as follows:

term 1

```
> git init
```

This creates a `.git` subdirectory that contains all the necessary repository files. We will explore this directory in more detail in a later session.

1.1 Lifecycle of the Status of Files

Each file in our working directory is either *tracked* or *untracked* by Git. As the word suggests, a tracked file is tracked by Git, i.e., Git is aware of the file. Untracked files are files that we do not want to track with Git. Since we do not care about them, neither will Git. That means they are not version controlled.

Further, a tracked file can be either *unmodified*, *modified*, or *staged*.

To understand the meaning of these words better, we will cover them one by one.

1.1.1 Untracked Files

To determine which files are in which state, we use the `git status` command. Running this command in our empty directory results in the following output:

term 2

```
> git status
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Let's create a new file and run the command again.

term 3

```
> echo "This is file1." > file1.txt
> git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file1.txt

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

As we can see, Git is very nice and tells us that the new file is untracked.

1.1.2 Tracked Files

To start tracking our newly created file, we run `git add` to stage it.

term 4

```
> git add file1.txt
> git status
On branch main

No commits yet
```

```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.txt
```

Now, our file is *staged*. Let's commit our staged file, together with a helpful commit message:

term 5

```
> git commit -m "Helpful commit msg."
[main (root-commit) 835d62f] Helpful commit msg.
 1 file changed, 1 insertion(+)
 create mode 100644 file1.txt
> git status
On branch main
nothing to commit, working tree clean
```

The output of `git commit` shows us more details of what we just committed. As we can see, our working tree is clean right after our commit. This means our file is currently *unmodified*.

The working tree represents the latest version of the files in our directory. Since there are no untracked and only unmodified files, our working tree is "clean".

Next, observe what happens when we modify the previously committed file:

term 6

```
> echo "This is line 2." >> file1.txt
> 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 working directory)
        modified:   file1.txt

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

Our file is now *modified*. We can stage the file again if we want to commit the change.

term 7

```
> git add file1.txt
> git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file1.txt
> git commit -m "Snd commit."
[main 8f2e277] Snd commit.
 1 file changed, 1 insertion(+)
> git status
On branch main
nothing to commit, working tree clean
```

Next, we will see how we can untrack a file. We can untrack a file *and* keep our local copy, including uncommitted changes, by executing `git rm --cached <file>`. Using Git terminology, this removes the file from the index[1], but keeps it in the working tree.

term 8

```
> git rm --cached file1.txt
rm 'file1.txt'
> git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:   file1.txt
```

[1] The index is our proposed next commit, the "staging area".

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file1.txt
```

As we can infer from the status, our file is once again *untracked*. Also, the deletion of the file has been staged.

In order for us to continue the next section with a clean working tree, we simply start tracking `file1.txt` again:

term 9

```
> git add file1.txt
> git status
On branch main
nothing to commit, working tree clean
```

This concludes the overview of the lifecycle of the status of files. Figure 1 provides a nice summary of the different statuses and which action is responsible for a change in status.

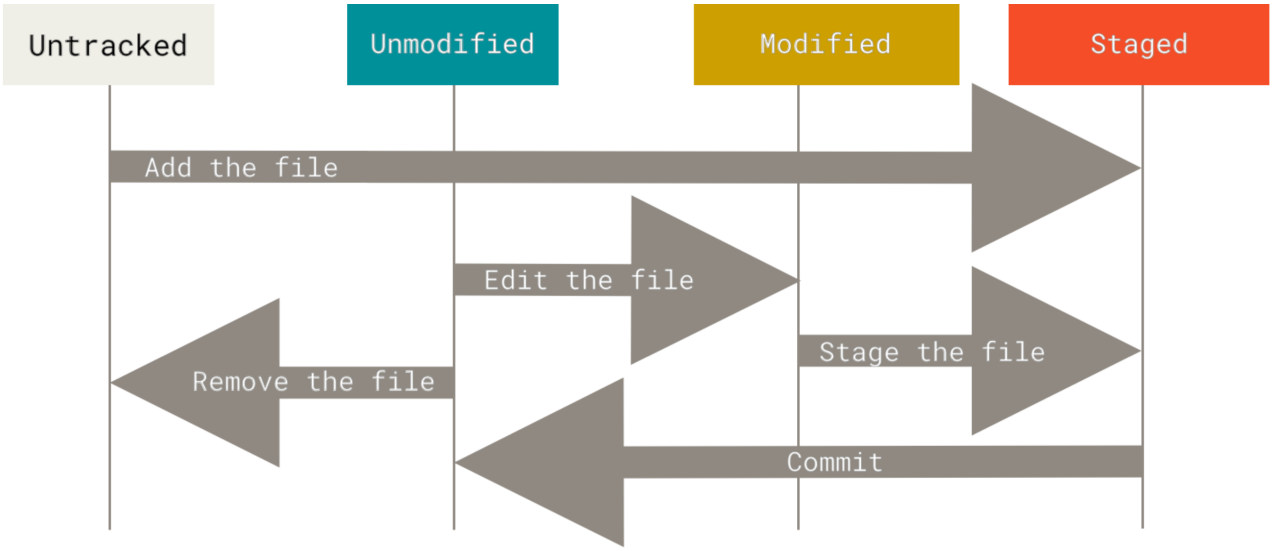

Figure 1: The lifecycle of the status of files [CS14]

1.2 More on File Status

Next, let's examine what happens when we stage a modified file and then edit it again. Starting from a clean working tree, we do the following to find that out:

term 10

```
> echo "Line 3." >> file1.txt
> git add file1.txt
> echo "Line 4." >> file1.txt
> git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file1.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.txt
```

As always, Git is kind and tells us that we staged the file and also that the file has been modified again. The crucial point to realize here is that the version of the file that is staged is different from the version that has been changed but not yet staged.

The version of the file that goes into the next commit is always the version of the file when it was staged, not the current version of the file. That is consistent with what the status is telling us, it differentiates between "Changes to be committed" and "Changes not staged for commit".

1.2.1 Viewing Changes

To examine what we described earlier, it would be helpful if there is a way to find out what changes have been made to our files. Of course, there is a way to do that; the command is `git diff`.

To find out what we have changed but not yet staged, we can run `git diff` with no other arguments:

term 11

```
> git diff
diff --git a/file1.txt b/file1.txt
index 009d01b..d6e18e1 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,3 +1,4 @@
 This is file1.
 This is line 2.
 Line 3.
+Line 4.
```

To find out what we have staged that will go into our next commit, we can run `git diff --staged` or the equivalent `git diff --cached`:

term 12

```
> git diff --staged
diff --git a/file1.txt b/file1.txt
index 11b567a..009d01b 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,2 +1,3 @@
 This is file1.
 This is line 2.
+Line 3.
```

Assume, we want to commit the latest changes to the file as well by staging it. Observe what happens when we run `git diff` again:

term 13

```
> git add file1.txt
> git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file1.txt

> git diff
>
```

We see nothing. As mentioned earlier, `git diff` with no other arguments shows *only* the changes that have not been staged. Since there are no unstaged changes left, we see nothing.

Of course, running `git diff --staged` now reveals the changes that have been staged:

term 14

```
> git diff --staged
diff --git a/file1.txt b/file1.txt
index 11b567a..d6e18e1 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1,2 +1,4 @@
 This is file1.
 This is line 2.
+Line 3.
+Line 4.
```

1.2.2 Removing Files

In term 8 we learned how we can remove a file from our index, but keep it in our working tree. To remove a file from the index *and* the working tree, i.e., untracking the file and deleting the local copy as well, we can run `git rm <file>`.

term 15

```
> git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file1.txt

> git rm file1.txt
error: the following file has changes staged in the index:
    file1.txt
(use --cached to keep the file, or -f to force removal)
```

What happened? This is a good example of Git warning us that we are about to do something that cannot be undone. Anything that is committed in Git can almost always be recovered. However, in this case, we have staged changes that have not been committed yet. Therefore, if we force remove the file, our staged changes cannot be recovered. In our case, this would not matter much because it is just dummy data, but it is good practice to always think twice (and know what you are about to do) when setting an `-f` flag.

To avoid losing our changes, let's commit them first and then demonstrate the removal of the file again:

term 16

```
> git commit -m "Better commit these changes."
[main 0258113] Better commit these changes.
 1 file changed, 2 insertions(+)
> git status
On branch main
nothing to commit, working tree clean
> git rm file1.txt
rm 'file1.txt'
> git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:   file1.txt

> ls
>
```

As we can see, the file has been removed from both the index and the working tree as desired.

In order for us to continue the next section with a clean working tree, we simply undo the removal of `file1.txt` again. We can do this by following the suggestions given in the output of `git status`.

term 17

```
> git restore --staged file1.txt
> git status
On branch main
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:   file1.txt

no changes added to commit (use "git add" and/or "git commit -a")
> git restore file1.txt
> git status
On branch main
nothing to commit, working tree clean
> ls
```

```
file1.txt
```

1.2.3 Moving Files

To rename a file, we use `git mv file_from file_to`.

term 18

```
> git mv file1.txt file.txt
> git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    file1.txt -> file.txt
> git mv file.txt file1.txt # Reverse renaming of file
> git status
On branch main
nothing to commit, working tree clean
```

We can also use the usual `mv` to move the file. However, this will involve making Git aware of the name change.

term 19

```
> mv file1.txt file.txt
> git status
On branch main
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    file1.txt

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

no changes added to commit (use "git add" and/or "git commit -a")
> git rm file1.txt
rm 'file1.txt'
> git add file.txt
> git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:  file1.txt -> file.txt
```

There is no difference between term 18 and term 19, i.e., `git mv file1.txt file.txt` is equivalent to the three commands: `mv file1.txt file.txt`, `git rm file1.txt`, and `git add file.txt`. However, of course, we prefer to avoid unnecessary commands. term 19 is still worth knowing for cases where the renaming of a file is done by some other tool.

2 Undoing Things Related to Status of Files

We create a local Git directory in another directory for the subsequent examples:

term 20

```
> git init
> echo "# Informative README" > README.md
> git add README.md
> git commit -m "Init"
[main (root-commit) cf110a4] Init
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

2.1 Unmodifying a Modified File

Warning: The following command deletes local changes of a file. The changes are not recoverable afterward. If you are unsure whether you will need these changes again at some point in the future, better create another branch and commit your changes or stash them. In general, everything that has been committed is likely recoverable, and everything that has not been committed can potentially be lost.

Let's say we modify a tracked file[1] and then decide we do not need all these local changes anymore.

term 21

```
> echo "Forgot the description." >> README.md
> 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 working directory)
        modified:   README.md

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

As we can see, Git suggests the command we can use to delete our unstaged changes:

term 22

```
> git restore README.md
> git status
On branch main
nothing to commit, working tree clean
```

Now, the file has the same content again as in the previous commit.

2.2 Unstaging a Staged File

Let's say we accidentally staged a file that we did not want to stage. We do this for both a tracked and an untracked file:

term 23

```
> echo "Forgot the description." >> README.md
> echo "Another file." > another_file.txt
> 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 working directory)
        modified:   README.md

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

[1] Note that we cannot unmodify an untracked file. If a file is not tracked by Git, it cannot have the modified status in the first place.

```
no changes added to commit (use "git add" and/or "git commit -a")
> git add README.md another_file.txt
> git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   README.md
        new file:   another_file.txt
```

As we can see, Git helps us here as well and mentions the command[2] to use to unstage the file(s). We can use `git restore --staged <file>...` to unstage files. The command is the same for both files, i.e., it does not matter whether the file was previously tracked or untracked. The files will go back to their respective previous status:

term 24

```
> git restore --staged README.md another_file.txt
> 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 working directory)
        modified:   README.md

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

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

[2]Once we covered what exactly `HEAD`, the index, and the working tree are, the manual (`man git-restore`) will become more understandable, and the following sentence will make more sense. We use the default command, i.e., no option, in term 22 because we want `HEAD` to be like the index again, and we use the `--staged` option in term 24 because we want the index to be like `HEAD` again.

3 Commit History

For this section we will use a small repository on GitHub as an example[1]. We can run the following to clone the repository:

term 25

```
> git clone https://github.com/JoshuaGloor/tiny-git-example.git
```

Running `git log` inside the project will give us the following output:

term 26

```
> git log
commit 304c6faa79f8d7aeb69d7458720396a238db8704 (HEAD -> main, origin/main,
origin/HEAD)
Merge:  b9b8e89 bb8847a
Author:  Bob <bob@example.com>
Date:  Sat Dec 23 20:05:10 2023 +0800

    Merge branch 'imp-fun'

commit b9b8e89e03159c4cc9498e0c3bb985c4fb410bb4
Author:  Alice <alice@example.com>
Date:  Sat Dec 23 20:02:52 2023 +0800

    Make it clearer that this script is of no use

commit bb8847a5aa918636c71643c317716487f528a02e
Author:  Bob <bob@example.com>
Date:  Sat Dec 23 20:01:43 2023 +0800

    Improve the fun

commit aa2387282d64d3f9522f5b3c2dfbe21fae8e8479
Author:  Alice <alice@example.com>
Date:  Sat Dec 23 18:53:06 2023 +0800

    Create a fun script

commit 8a8ad16bcdc0ced67ecaaea7b658d131636d08ea
Author:  Alice <alice@example.com>
Date:  Sat Dec 23 18:44:38 2023 +0800

    Initial commit
```

It is important to remember that a simple `git log` without any arguments will list the commits that are reachable by following the parent links[2] of where `HEAD` is pointing[3]. By default, it will not list commits made on branches that are not reachable by `HEAD`.

There are many options for the `git log` command that can be useful in different situations. We will not showcase them in this guide. However, they can always be found by checking the manual page, `man git-log`.

One options combination for `git log` which can be useful to get an overview of all the commits is the following:

term 27

```
> git log --all --graph --oneline
*   304c6fa (HEAD -> main, origin/main, origin/HEAD) Merge branch 'imp-fun'
|\
| * bb8847a Improve the fun
* | b9b8e89 Make it clearer that this script is of no use
```

[1] The repository is public, so you can clone it too. However, it is just a dummy repository and is used solely to demonstrate `git log`. We will not make use of this repository in later sessions and instead always create new repositories from scratch.

[2] Each commit retains pointer(s) to its previous commit(s), called parents.

[3] If you are on a branch, `HEAD` is a pointer to the current branch reference and the current branch reference is a pointer to the last commit made on that branch. In "detached `HEAD`" state you are not on a branch and `HEAD` directly points to a commit. But at the moment we do not need to worry about that. We will cover references later on. It is enough to think of `HEAD` as a pointer to where we are currently at.

```
|/
* aa23872 Create a fun script
* 8a8ad16 Initial commit
```

We can see that a shortened SHA-1, only the first line of the commit message, and a graph are displayed.

4 Amending Commits

Warning: The following command performs what in Git jargon is often referred to as "rewriting history". Avoid the modification of commits that were already pushed, otherwise your collaborators will get angry with you. We will go into more detail on why they would get angry when we cover rebasing. It is generally a good rule to think before you push and consider all pushed commits as final. However, as long as you only change (rewrite) local commits nobody will complain.

We create a local Git directory to showcase this command:

term 28

```
> git init
```

Let's say we do a commit, and then we realize we want to modify a file again, fix a typo, or any other thing that should be part of the commit to which the tip of the current branch is pointing.

term 29

```
> echo "# Informative README" > README.md
> git add README.md
> git commit -m "Init"
[main (root-commit) 5b89991] Init
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
> git log --oneline
5b89991 (HEAD -> main) Init
> echo "Forgot the description." >> README.md
> git add README.md
> git commit --amend --no-edit
[main 86ff344] Init
 Date:  Sat Jan 13 18:50:13 2024 +0800
 1 file changed, 2 insertions(+)
 create mode 100644 README.md
> git log --oneline
86ff344 (HEAD -> main) Init
```

Where we used the `--no-edit` option to indicate that we do not want to modify the commit message. Otherwise, vim (or your preferred editor tool) will open up and allow you to modify the commit message as well.
When looking at the commit hash, we can also clearly see that we are not editing the previous commit, but rather replacing it completely with a new one.

Session 02: Commits and Branches

5 Looking Under the Hood of Commits

To better understand branching, let's first examine how git stores its data. This will also help us better understand what the *index*, the *working tree*, and *HEAD* really are.

We explore this by creating a local git repository and observing how `.git` changes along the way.

term 30

```
> git init
```

To examine all the content in our directory in a nice format, we can use the `tree -a` command, where the `-a` option is used to display hidden files and directories (such as `.git`):

term 31

```
> tree -a
.
`-- .git
    |-- HEAD
    |-- config
    |-- description
    |-- hooks
    |   |-- applypatch-msg.sample
    |   |-- commit-msg.sample
    |   |-- fsmonitor-watchman.sample
    |   |-- post-update.sample
    |   |-- pre-applypatch.sample
    |   |-- pre-commit.sample
    |   |-- pre-merge-commit.sample
    |   |-- pre-push.sample
    |   |-- pre-rebase.sample
    |   |-- pre-receive.sample
    |   |-- prepare-commit-msg.sample
    |   |-- push-to-checkout.sample
    |   |-- sendemail-validate.sample
    |   `-- update.sample
    |-- info
    |   `-- exclude
    |-- objects
    |   |-- info
    |   `-- pack
    `-- refs
        |-- heads
        `-- tags

9 directories, 18 files
```

The hooks directory contains some sample hooks, which are programs that trigger actions at certain points when Git executes. This offers us a way to customize Git with custom scripts. However, since we will not create any hooks and the sample hooks clutter our output, we will use the command `tree -a -I "hooks"` from now on, such that `tree` does not list the contents in the `hooks` directory every time.

Let's create two files and see what happens:

term 32

```
> echo "# Informative README" > README.md
> echo "A file" > file.txt
> tree -a -I "hooks"
.
|-- .git
|   |-- HEAD
```

```
|   |-- config
|   |-- description
|   |-- info
|   |   `-- exclude
|   |-- objects
|   |   |-- info
|   |   `-- pack
|   `-- refs
|       |-- heads
|       `-- tags
|-- README.md
`-- file.txt

8 directories, 6 files
```

We see that the only thing that has changed is the two files we just added. We can easily verify this by the total number of directories and files the `tree` command displays on the last line of the output[1].

From the last session, we know that if we run `git status` now, Git would tell us that `README.md` and `file.txt` are untracked. So, let's stage the files:

term 33

```
> git add README.md file.txt
> tree -a -I "hooks"
.
|-- .git
|   |-- HEAD
|   |-- config
|   |-- description
|   |-- index
|   |-- info
|   |   `-- exclude
|   |-- objects
|   |   |-- 04
|   |   |   `-- 4fbb280515ba19ddfbb8f40acd24956e021bd2
|   |   |-- 51
|   |   |   `-- f466f2e446ade0b0b2e5778ce3e0fa95e380e8
|   |   |-- info
|   |   `-- pack
|   `-- refs
|       |-- heads
|       `-- tags
|-- README.md
`-- file.txt

10 directories, 9 files
```

Interesting, now we have two more directories, `.git/objects/04` and `.git/objects/51`. We also have three more files, one file each in the newly generated directories and one file called `index` at `.git/index`.

5.1 Blobs

When we stage a file, Git computes a checksum for that file, stores that version of the file in a directory inside `.git/objects`, and adds the generated checksum to the staging area. The checksum is a SHA-1 hash, the file containing the version is called a *blob*, and the staging area is nothing other than what the file `index` contains.

Of course, we would like to verify this, so let's see what is in one of the blobs that were created:

[1] There were 14 hooks in the hook directory, and we created two new files: $18 - 14 + 2 = 6$. There is one fewer directory because we do not list the `hooks` directory anymore.

term 34

```
> cat .git/objects/04/4fbb280515ba19ddfbb8f40acd24956e021bd2
xK��OR02dPV��K�/�M,�,Krut�u���%
```

Whoops, these are some weird hieroglyphs... It turns out that Git compresses its content using zlib.

Luckily for us, there is a way to read object files using the `git cat-file` command. Using the SHA-1 hash of the object to uniquely identify it, together with the `-p` option, we can pretty-print its contents.

If this unique hash-to-value concept sounds familiar, you are correct, at the core of Git is a simple key-value data store.

We can find the hash of the object by prepending the parent directory to the file name. For example, to examine the contents of the file `.git/objects/04/4fbb280515ba19ddfbb8f40acd24956e021bd`, we use the hash consisting of the directory name `04` and the file name `4fbb280515ba19ddfbb8f40acd24956e021bd`:

term 35

```
> git cat-file -p 044fbb280515ba19ddfbb8f40acd24956e021bd2
# Informative README
```

Ok, that is cool! We extracted the contents of the blob. We can also check the type of the object by using the `-t` option instead of `-p`:

term 36

```
> git cat-file -t 044fbb280515ba19ddfbb8f40acd24956e021bd2
blob
```

The contents of the other blob can be displayed analogously.

5.2 The Index

Let's turn our attention to the `index` file now. To read that file, we can use the command `git ls-files --stage`:

term 37

```
> git ls-files --stage
100644 044fbb280515ba19ddfbb8f40acd24956e021bd2 0 README.md
100644 51f466f2e446ade0b0b2e5778ce3e0fa95e380e8 0 file.txt
```

The output is grouped into <mode> <object> <stage> <file>, where <mode> is split into object type and UNIX permission. We will not explain all the possibilities of <mode> and <stage> because it is less important, but we will explain what it means for our example.

- <mode>
  The object type does not refer to objects in the object store. In our example, the object type is 100. It means that this file is a regular file. The UNIX permission 644 stands for read and write for the file owner, read for the group, and read for others.
- <object>
  This is the checksum we encountered previously.
- <stage>
  This group has meaning during merges. It is not important to us at the moment.
- <file>
  The file name that has its contents saved at <object>.

It is no coincidence that this file is called `index`; it is the *index* or *staging area* we have been talking about. It mainly just saves the file names and the hashes of the blobs of the things we stage; that is it.

5.3 Commit Objects

Next, we perform a commit:

term 38

```
> git commit -m "Init"
[main (root-commit) 27fcf0d] Init
```

```
 2 files changed, 2 insertions(+)
 create mode 100644 README.md
 create mode 100644 file.txt
> tree -a -I "hooks"
.
|-- .git
|   |-- COMMIT_EDITMSG
|   |-- HEAD
|   |-- config
|   |-- description
|   |-- index
|   |-- info
|   |   `-- exclude
|   |-- logs
|   |   |-- HEAD
|   |   `-- refs
|   |       `-- heads
|   |           `-- main
|   |-- objects
|   |   |-- 04
|   |   |   `-- 4fbb280515ba19ddfbb8f40acd24956e021bd2
|   |   |-- 27
|   |   |   `-- fcf0d749dccb5170673bfa8cc84e815054e772
|   |   |-- 51
|   |   |   `-- f466f2e446ade0b0b2e5778ce3e0fa95e380e8
|   |   |-- fb
|   |   |   `-- 27651563cf40b4d222b903757a2ac4644220e6
|   |   |-- info
|   |   `-- pack
|   `-- refs
|       |-- heads
|       |   `-- main
|       `-- tags
|-- README.md
`-- file.txt

15 directories, 15 files
```

Comparing this output of the `tree` command to the one in term 33, we notice that we have five new directories, namely `.git/logs`, `.git/logs/refs`, `.git/logs/refs/heads`, `.git/objects/27`, and `.git/objects/fb`. We also have six new files, `.git/COMMIT_EDITMSG`, `.git/logs/HEAD`, `.git/logs/refs/heads/main`, one new file each in the newly created directories in the `.git/objects` directory, and `.git/refs/heads/main`.

We will focus our attention on what happened in the `.git/objects` directory.

term 39

```
> git cat-file -p 27fcf0d749dccb5170673bfa8cc84e815054e772
tree fb27651563cf40b4d222b903757a2ac4644220e6
author Alice <alice@example.com> 1706424772 +0800
committer Alice <alice@example.com> 1706424772 +0800

Init
> git cat-file -t 27fcf0d749dccb5170673bfa8cc84e815054e772
commit
```

We see that this is a commit object. It includes things like the author and the commit message, but also a tree and another hash. We notice that this hash is the same as the one from the new object in `.git/objects` that we have not examined yet.

5.4 Tree Objects

Let's see what is in the last object:

term 40

```
> git cat-file -p fb27651563cf40b4d222b903757a2ac4644220e6
100644 blob 044fbb280515ba19ddfbb8f40acd24956e021bd2    README.md
100644 blob 51f466f2e446ade0b0b2e5778ce3e0fa95e380e8    file.txt
> git cat-file -t fb27651563cf40b4d222b903757a2ac4644220e6
tree
```

We have a tree object that contains the hashes of the two blobs we had previously.

5.5 HEAD

Next, we look at what is in `.git/HEAD`:

term 41

```
> cat .git/HEAD
ref:  refs/heads/main
```

Interesting! It contains a pointer to a file called `main` in the directory `.git/refs/heads`. So, let's see what is in there:

term 42

```
> cat .git/refs/heads/main
27fcf0d749dccb5170673bfa8cc84e815054e772
```

It is a hash; more specifically, it is the hash of the last commit. A reference is just a file that saves a SHA-1 hash of an object in Git. References make our lives easier because we do not have to remember the SHA-1 value, just a human-readable name, namely the reference.

That is all there is to it. The vast majority of times, `HEAD` is just a file that points to a reference, and that reference points to a commit object. There is also the case where the `HEAD` file contains a hash of an object in Git directly. That is the case when we are in a "detached `HEAD`" state.

5.6 The Big Picture

When we create a commit, Git calculates a checksum and creates a tree object for each subdirectory. In our case, we only had a root directory. After that, Git creates a commit object that contains the hash to the tree at the root of the project. That enables Git to recreate the whole snapshot.

We find the last commit using `HEAD`:

term 43

```
> cat .git/HEAD
ref:  refs/heads/main
> cat .git/refs/heads/main
27fcf0d749dccb5170673bfa8cc84e815054e772
```

Given the commit hash, we can find the root tree:

term 44

```
> git cat-file -p 27fcf0d749dccb5170673bfa8cc84e815054e772
tree fb27651563cf40b4d222b903757a2ac4644220e6
author Alice <alice@example.com> 1706424772 +0800
committer Alice <alice@example.com> 1706424772 +0800

Init
```

Given the root tree hash, we can find the blobs and further tree hashes if there are any subdirectories (this is not the case in the example we created).

term 45

```
> git cat-file -p fb27651563cf40b4d222b903757a2ac4644220e6
100644 blob 044fbb280515ba19ddfbb8f40acd24956e021bd2    README.md
100644 blob 51f466f2e446ade0b0b2e5778ce3e0fa95e380e8    file.txt
> git cat-file -p 044fbb280515ba19ddfbb8f40acd24956e021bd2
# Informative README
```

```
> git cat-file -p 51f466f2e446ade0b0b2e5778ce3e0fa95e380e8
A file
```

5.6.1 The Next Commit

It is noteworthy that the commit object usually contains a pointer to the previous commit(s) (the commit's parent(s)[2]). That was not the case for the previous commit because it was the first one, and therefore has no parent. We can see the parent pointer when we create another commit:

term 46

```
> echo "Forgot the description." >> README.md
> git add README.md
> git commit -m "Add description"
[main 9d67752] Add description
 1 file changed, 1 insertion(+)
> git cat-file -p HEAD
tree ab0b9cff0b25579775013e48cad736a34b5cf664
parent 27fcf0d749dccb5170673bfa8cc84e815054e772
author Alice <alice@example.com> 1706437634 +0800
committer Alice <alice@example.com> 1706437634 +0800

Add description
```

[2] As we mentioned in a previous session already, this is, for example, how `git log` can display all ancestors of a given commit; it traverses the commits from one parent to the next. Multiple parents are possible when the commit was created from a merge.

6 What Is a Branch?

After covering section 5, we can say that a branch is simply a reference.

To see that, we can create a branch and observe what changes inside the `.git` directory. The current state looks as follows:

term 47

```
> tree -a -I "hooks"
.
|-- .git
|   |-- COMMIT_EDITMSG
|   |-- HEAD
|   |-- config
|   |-- description
|   |-- index
|   |-- info
|   |   `-- exclude
|   |-- logs
|   |   |-- HEAD
|   |   `-- refs
|   |       `-- heads
|   |           `-- main
|   |-- objects
|   |   |-- 04
|   |   |   `-- 4fbb280515ba19ddfbb8f40acd24956e021bd2
|   |   |-- 0d
|   |   |   `-- 18c30e8340a1da76b46963eb248f7c3b40ad42
|   |   |-- 27
|   |   |   `-- fcf0d749dccb5170673bfa8cc84e815054e772
|   |   |-- 51
|   |   |   `-- f466f2e446ade0b0b2e5778ce3e0fa95e380e8
|   |   |-- 9d
|   |   |   `-- 6775294aeff3979bb1a40a5e67d24be5242c01
|   |   |-- ab
|   |   |   `-- 0b9cff0b25579775013e48cad736a34b5cf664
|   |   |-- fb
|   |   |   `-- 27651563cf40b4d222b903757a2ac4644220e6
|   |   |-- info
|   |   `-- pack
|   `-- refs
|       |-- heads
|       |   `-- main
|       `-- tags
|-- README.md
`-- file.txt

18 directories, 18 files
```

Note that, compared to term 38, we have three more directories and files because we created a new commit since then. In section 5.6.1 we edited a file and staged it (this creates one new directory and file), then we created a commit (one new directory and file for the tree object and one new directory and file for the commit object).
Note that a new directory inside `.git/objects` is only created if the two most significant (left-most) bytes of a new hash do not already exist as a directory. That is the case in our example.

Let's create a new branch. We can do this using the `git branch <branchname>` command. Without any other arguments given, this will create a branch that points to the same commit object as `HEAD` currently does.

term 48

```
> git branch testing
> tree -a -I "hooks"
.
|-- .git
```

```
|   |-- COMMIT_EDITMSG
|   |-- HEAD
|   |-- config
|   |-- description
|   |-- index
|   |-- info
|   |   `-- exclude
|   |-- logs
|   |   |-- HEAD
|   |   `-- refs
|   |       `-- heads
|   |           |-- main
|   |           `-- testing
|   |-- objects
|   |   |-- 04
|   |   |   `-- 4fbb280515ba19ddfbb8f40acd24956e021bd2
|   |   |-- 0d
|   |   |   `-- 18c30e8340a1da76b46963eb248f7c3b40ad42
|   |   |-- 27
|   |   |   `-- fcf0d749dccb5170673bfa8cc84e815054e772
|   |   |-- 51
|   |   |   `-- f466f2e446ade0b0b2e5778ce3e0fa95e380e8
|   |   |-- 9d
|   |   |   `-- 6775294aeff3979bb1a40a5e67d24be5242c01
|   |   |-- ab
|   |   |   `-- 0b9cff0b25579775013e48cad736a34b5cf664
|   |   |-- fb
|   |   |   `-- 27651563cf40b4d222b903757a2ac4644220e6
|   |   |-- info
|   |   `-- pack
|   `-- refs
|       |-- heads
|       |   |-- main
|       |   `-- testing
|       `-- tags
|-- README.md
`-- file.txt

18 directories, 20 files
> cat .git/refs/heads/testing
9d6775294aeff3979bb1a40a5e67d24be5242c01
> cat .git/HEAD
ref:  refs/heads/main
> cat .git/refs/heads/main
9d6775294aeff3979bb1a40a5e67d24be5242c01
```

The second new file is inside the `.git/logs` directory. We will find out more about this directory another time.

7 Working Tree and the Three Trees

One question remains. We found out what the index is in section 5.2 and what `HEAD` is in section 5.5; it is left to find out what the working tree is. With our current understanding, we can finally solve this mystery. The working tree is simply our local copy of files. They are not stored anywhere in the `.git` folder. That is also why we can potentially lose files in the working tree because they are not saved anywhere else.

The working tree, the index, and `HEAD` are also called the *three trees*. Trees, in the sense of a collection of files and not the data structure. Each file that has been part of at least one commit[1] can exist in up to three different versions at the same time before the next commit is performed. One version is the version from the last commit (`HEAD`), another version is the version that is staged (index), and the last version is the version of the local copy (working tree). Note that these versions must not be different; they can all be the same, two of them can be the same and the other one different, or they can all be different.

By understanding the three trees, commands such as `git status` can make a lot more sense. Looking at the manual of the `git status` command (`man git-status`), we find the following in the description:

> Displays paths that have differences between the index file and the current HEAD commit, paths that have differences between the working tree and the index file, and paths in the working tree that are not tracked by Git (and are not ignored by **gitignore**(5)).

So, the command calculates the differences between these three trees. If the working tree is said to be clean, it means that the versions of the non-ignored files are equivalent across the three trees.

[1] When a file has never been committed and is untracked, it is only part of the working tree. Also, if a file has never been committed and is staged, it is only part of the working tree and the index, but not `HEAD`.

Session 03: Remotes - Part 1

8 Remotes - Push-Side

To see what remotes are and what actions Git performs, we will simulate a small project on which two people, Alice and Bob, collaborate. To do this using a single GitHub account on the same computer, we will set up a directory in which Alice writes her contributions and a separate directory for Bob. That allows us to showcase the ideas with a single GitHub account.

First, we will create Alice's workspace:

term 49

```
> mkdir alice
```

From now on, we will prepend the current working directory followed by the current branch in the prompt, i.e., before the > symbol, to show in which collaborators workspace we are running the command and which branch we are currently on.

term 50

```
> cd alice
alice > git init
alice main > git config user.name "Alice"
alice main > git config user.email "alice@example.com"
alice main > echo "# Informative README" > README.md
alice main > git add README.md
alice main > git commit -m "Init"
[main (root-commit) d506edd] Init
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

We also changed the name and email of the author to "Alice" and "alice@example.com", respectively. We did this to be able to distinguish Alice's work from Bob's work[1].

So far, these commands are familiar to us from the previous sessions. Let's also run the `tree` command for the same reason as in the previous sessions, to see what will change in our `.git` directory over time:

term 51

```
alice main > tree -a -I "hooks"
.
|-- .git
|   |-- COMMIT_EDITMSG
|   |-- HEAD
|   |-- config
|   |-- description
|   |-- index
|   |-- info
|   |   `-- exclude
|   |-- logs
|   |   |-- HEAD
|   |   `-- refs
|   |       `-- heads
|   |           `-- main
|   |-- objects
|   |   |-- 04
|   |   |   `-- 4fbb280515ba19ddfbb8f40acd24956e021bd2
|   |   |-- 9a
|   |   |   `-- 9df47cb2dd2cd9f8cdca6c5b943a9f5ebf3856
|   |   |-- d5
|   |   |   `-- 06eddd2fa2ae85f7891e2b3b473f779dfcfd93
|   |   |-- info
|   |   `-- pack
```

[1] This config is only a local change and does not affect your existing global configurations. You can observe this by running `git config --global user.name` and comparing it to `git config user.name`. The same holds for the email.

```
|   `-- refs
|       |-- heads
|       |   `-- main
|       `-- tags
`-- README.md

14 directories, 13 files
```

In GitHub, we create a new and empty repository, i.e., we do not add any README, .gitignore, or license file. Let's call our repository *learning-remotes* and then click create repository.

8.1 Adding a Remote

After we create the repository, GitHub displays the SSH URL. We use this URL[2] to add the remote repository to our already existing local repository. We can add the remote using the `git add remote <name> <URL>` command:

term 52

```
alice main > git remote add origin \
git@github.com:JoshuaGloor/learning-remotes.git
```

When we run the `tree` command again, we will not see any changes; no new file or directory was added. That is because remotes are saved in the `.git/config` file, which already existed before. Let's see what was added to our config file:

term 53

```
alice main > cat .git/config
[core]
  repositoryformatversion = 0
  filemode = true
  bare = false
  logallrefupdates = true
  ignorecase = true
  precomposeunicode = true
[user]
  name = Alice
  email = alice@example.com
[remote "origin"]
  url = git@github.com:JoshuaGloor/learning-remotes.git
  fetch = +refs/heads/*:refs/remotes/origin/*
```

We will not look into the `[core]` section of the git config. Under `[user]`, we see the name and email we configured earlier. We will have a look at the last section, `[remote "origin"]`, where "origin" is the name we have given our remote. Note, it is common to name the first remote "origin"; that is also what `git clone` does by default. However, there is nothing special about this name; one can choose any name for the remote.

We see that the value of the `url` option is the URL we used when running the command. We also notice a particular value for the `fetch` option; this is the default set of *refspec* for the `git fetch` command.

8.2 Refspec

The refspec format is `+<src>:<dst>`.

In the case of the refspec in term 53, it has the following meaning[3]:

- `+`
  is optional and indicates to Git whether to update the references even if it is not a fast-forward[4].
- `<src>`
  is the pattern for references on the remote side.
- `<dst>`
  is the pattern for where these references will be tracked locally.

[2] If you follow along, use the URL displayed in your GitHub account.

[3] Note, this describes the refspec for `fetch`. The format stays the same, but it can have a different meaning when used with other commands as we will later see with `git push`.

Using this information, together with the contents of the config file we saw in term 53, we can deduce that if there is a remote reference `refs/heads/main`, it will be tracked at `refs/remotes/origin/main` locally.

The asterics in the `<src>` and `<dst>` pattern function as a glob pattern, i.e., in our case, all the references under `refs/heads` are fetched. We can edit/add/remove the values under `[remote "origin"]` to restrict which references are fetched exactly. For example, we could add the following fetch value:

term 54

```
[remote "origin"]
  url = git@github.com:JoshuaGloor/learning-remotes.git
  fetch = +refs/heads/*:refs/remotes/origin/*
  fetch = ^refs/heads/dontwant
```

This would avoid fetching the `dontwant` reference from the remote (which does not exist currently; this is just to illustrate the idea). We could have also added this value by executing:

term 55

```
alice main > git config --add remote.origin.fetch "^refs/heads/dontwant"
```

8.3 Pushing to Remote and Upstream References

Before we push for the first time, let's quickly run `git branch -a` to display all the branches (remote-tracking and local ones):

term 56

```
alice main > git branch -a
* main
```

Let's push:

term 57

```
alice main > git push -u origin main
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 226 bytes | 226.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:JoshuaGloor/learning-remotes.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

We will break this command down. That will also explain the received output.

- `git push origin main` has some syntactic sugar to it. When we take a look at the synopsis of the git push manual (`man git-push`), we see that the last arguments are: `[<repository> [<refspec>...]]`. That means we typed `main` instead of a refspec. However, as we learned before, the refspec has the format `+<src>:<dst>`. Git recognizes that and tries to infer the full refspec. The command that was actually executed is `git push -u origin refs/heads/main:refs/heads/main`.
  - This means that we take the reference `ref/heads/main` from our repository (this is the `<src>`) and update the reference `refs/heads/main` on the remote side[5] (the `<dst>`).
  - Note, `<dst>` is not `refs/remotes/origin/main` because that is where the remote reference of `main` (i.e., `refs/heads/main`) is tracked locally.
- We used the `-u` option to set an upstream reference. This means that our local `main` branch will now track the upstream branch `refs/heads/main`. This upstream `refs/heads/main` is the branch on the remote (i.e., on GitHub), not our local `refs/heads/main`!

We can observe this tracking of the upstream branch in the config file; a new section was added:

[4] During a merge or rebase, a *fast-forward* is performed if the second commit is a descendant of the first commit. Equivalently, we can say that the commit we want to merge with is ahead of our existing work, i.e., our commit is an ancestor of the commit we want to merge with. If that is the case, Git can simply move the pointer to the second commit because there is no divergent work.

[5] We can verify the existing references on GitHub using the GitHub REST API. See the REST API endpoints for Git references on how to do this.

term 58

```
alice main > cat .git/config
[core]
  repositoryformatversion = 0
  filemode = true
  bare = false
  logallrefupdates = true
  ignorecase = true
  precomposeunicode = true
[user]
  name = Alice
  email = alice@example.com
[remote "origin"]
  url = git@github.com:JoshuaGloor/learning-remotes.git
  fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
  remote = origin
  merge = refs/heads/main
```

Under `[branch "main"]` we can observe the definition of the upstream branch. `remote` specifies which remote and `merge` defines the upstream branch. When on branch `main`, this tells commands `git fetch`, `git pull`, and `git rebase` which upstream branch to use for merging.

Now, we can also observe what has changed in our `.git` folder:

term 59

```
alice main > tree -a -I "hooks"
.
|-- .git
|   |-- COMMIT_EDITMSG
|   |-- HEAD
|   |-- config
|   |-- description
|   |-- index
|   |-- info
|   |   `-- exclude
|   |-- logs
|   |   |-- HEAD
|   |   `-- refs
|   |       |-- heads
|   |       |   `-- main
|   |       `-- remotes
|   |           `-- origin
|   |               `-- main
|   |-- objects
|   |   |-- 04
|   |   |   `-- 4fbb280515ba19ddfbb8f40acd24956e021bd2
|   |   |-- 9a
|   |   |   `-- 9df47cb2dd2cd9f8cdca6c5b943a9f5ebf3856
|   |   |-- d5
|   |   |   `-- 06eddd2fa2ae85f7891e2b3b473f779dfcfd93
|   |   |-- info
|   |   `-- pack
|   `-- refs
|       |-- heads
|       |   `-- main
|       |-- remotes
|       |   `-- origin
|       |       `-- main
|       `-- tags
`-- README.md
```

```
18 directories, 15 files
```

Compared to term 51, we have four new directories and two new files. The new directories are `remotes/origin` under `.git/logs` and `.git/refs`. The two new files are both called `main`, one in each `remotes/origin` folder that was newly created.

8.4 Remote-Tracking Branches

We can also see the new reference that was created by comparing the output of `git branch -a` from term 56 with the current one:

term 60

```
alice main > git branch -a
* main
  remotes/origin/main
```

The `remotes/origin/main` is a remote-tracking branch. Remote-tracking branches are references to the state of remote branches. They are local references that show the state of the remote branch at the time of the last network communication.

This is an important point to understand, as long as we are not in contact with our `origin` server (GitHub), the branch `remotes/origin/main` will not move. Synchronization will not happen automatically without us running a command that communicates with `origin`.
This means that it is possible that our local `main` branch might point to a different commit than the remote-tracking branch `remotes/origin/main`, and the `main` branch on the remote might also point to a different commit[6] than `remotes/origin/main`. This is the case if we did a local commit and someone else committed work to GitHub since we last fetched from it.

At the end of it all, if we now check GitHub. We will see that the repository now contains the `README.md` file that Alice pushed.

[6] We do not observe this `main` branch in our local repository, this refers to the current state on GitHub.

Session 04: Remotes - Part 2

9 Remotes - Clone-Side

In the previous session, Alice pushed her repository to GitHub. In this session, we will examine what happens on Bob's side.

9.1 Cloning a Repository

`git clone` clones the repository into a newly created directory. Since we want to create Bob's working directory anyway, we can just name this directory `bob`:

term 61

```
> git clone git@github.com:JoshuaGloor/learning-remotes.git bob
Cloning into 'bob'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
> cd bob
bob main > git config user.name "Bob"
bob main > git config user.email "bob@example.com"
```

Like last time, we also changed the local name and email to identify Bob's work.

Next, let's find out what `git clone` created for us, i.e., we try to find out whether there is a difference between what Alice pushed and what Bob cloned.

When observing the `.git/config` file, we see that Bob has the same configurations in the config file that we previously observed for Alice after she pushed:

term 62

```
bob main > cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
    ignorecase = true
    precomposeunicode = true
[remote "origin"]
    url = git@github.com:JoshuaGloor/learning-remotes.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
    remote = origin
    merge = refs/heads/main
[user]
    name = Bob
    email = bob@example.com
```

This means the remote was added and given the default name "origin". Also, the existence of the `[branch "main"]` section tells us that a `main` branch was created, and an upstream reference for the remote reference `refs/heads/main` was set.

9.2 Packfiles

When looking at the files in the `.git` folder, we can observe some differences:

term 63

```
bob main > tree -a -I "hooks"
.
|-- .git
```

```
|   |-- HEAD
|   |-- config
|   |-- description
|   |-- index
|   |-- info
|   |   `-- exclude
|   |-- logs
|   |   |-- HEAD
|   |   `-- refs
|   |       |-- heads
|   |       |   `-- main
|   |       `-- remotes
|   |           `-- origin
|   |               `-- HEAD
|   |-- objects
|   |   |-- info
|   |   `-- pack
|   |       |-- pack-5d37b7a1382b38100d880ecaf4f1abd3c4896342.idx
|   |       |-- pack-5d37b7a1382b38100d880ecaf4f1abd3c4896342.pack
|   |       `-- pack-5d37b7a1382b38100d880ecaf4f1abd3c4896342.rev
|   |-- packed-refs
|   `-- refs
|       |-- heads
|       |   `-- main
|       |-- remotes
|       |   `-- origin
|       |       `-- HEAD
|       `-- tags
`-- README.md

15 directories, 15 files
```

What stands out the most here is that the contents inside the `.git/objects` folder have changed. All our objects are gone, and we have a `.git/objects/pack` folder with three files inside instead.

The format we learned about so far is referred to as a *loose* object format; each version of a file is stored in a separate blob. With a growing number of objects, this becomes increasingly inefficient. That is why Git occasionally packs up several objects into a single binary file called a *packfile*.
Loosely speaking, instead of saving each version in a file separately, Git uses delta compression to only store the deltas from one version of the file to the next. The details of the algorithm are beyond our scope[1]. So, this is what is in our newly discovered `.pack` file. The `.idx` file with the same file name as the `.pack` file is an index containing offsets into the packfile to enable faster seek time of a particular object. Finally, the corresponding `.rev` file is a reverse index[2] for the packfile.

However, analogously to the loose object format, we can still retrieve all the information given a SHA-1 hash:

term 64

```
bob main > git cat-file -p HEAD
tree 9a9df47cb2dd2cd9f8cdca6c5b943a9f5ebf3856
author Alice <alice@example.com> 1707640458 +0800
committer Alice <alice@example.com> 1707640458 +0800

Init
```

Git packed up the objects when Alice pushed. Usually, Git triggers this packing up automatically when it determines it to be necessary (too many loose objects exist), when pushing to a remote server, or if we run `git gc` manually. According to the manual (`man git-gc`), manual invocation of this command is seldom necessary, however.

[1] However, if you are interested in some insights about how packing works underneath the surface, I recommend reading this enjoyable conversation in the documentation: git/Documentation/technical/pack-heuristics.txt. A rendered version can be found here.

[2] Not to be confused with an inverted index. Reverse indices are used to improve the balance of an index. The key is simply reversed before inserting it into the index. Inverted indices however are indices that store a mapping from value to key. For example, instead of storing the location of a word in a document to the word, an inverted index is built using words that point to the location of the word in that document.

9.3 origin/HEAD

Another file we did not see before is `.git/refs/remotes/origin/HEAD`. Let's see what is in it:

term 65

```
bob main > cat .git/refs/remotes/origin/HEAD
ref:  refs/remotes/origin/main
```

`remotes/origin/HEAD` is a symbolic reference[3]; it contains a pointer to another reference. In this case, it points to `refs/remotes/origin/main`. We can find out more by looking at the manual. In `man git-remote`, under the command `set-head`, it states that the target (i.e., the reference the symbolic reference is pointing to) of `remotes/origin/HEAD` is the default branch for the remote `origin`. The section provides the following example:

> [...] if the default branch for **origin** is set to **master**, then **origin** may be specified wherever you would normally specify **origin/master**.

It is also the default branch that is checked out when cloning a repository. The only way it changes is if someone changes or deletes the default branch using the `set-head` command of `git remote`. For example, the target does not move when we checkout another branch in our local repository.

9.4 Packed References

Next, let's see what is in `.git/refs/remotes/origin/main`. The only problem is that this file no longer seems to be in our `.git` folder. Indeed, this file does not exist anymore, but the reference can be located in another file, namely in `.git/packed-refs`. As we know from previous sessions, Git creates a file for each reference. With time, this one-file-per-reference becomes inefficient and wastes storage. That is why Git may pack up the references and store them in a single file instead. Let's see what is in there:

term 66

```
bob main > cat .git/packed-refs
# pack-refs with: peeled fully-peeled sorted
d506eddd2fa2ae85f7891e2b3b473f779dfcfd93 refs/remotes/origin/main
```

There, we find our missing reference. Git first tries to find the reference in the `.git/refs` directory. If unsuccessful, Git tries to find it in the `.git/packed-refs` file. That means if we ever look for a reference and cannot find it in `.git/refs`, it is probably in `.git/packed-refs`.

The "peeled fully-peeled" relates to the peeling of tag objects, which is the action of recursively dereferencing it. Since we did not cover tags, this does not concern us, but the meaning can be found in the source code [Hag17].

9.5 Comparing Branches

Finally, we will have a look at the existing branches:

term 67

```
bob main > git branch -a
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
```

Compared to Alice's branches, we additionally have `remotes/origin/HEAD -> origin/main` in the output, which is the symbolic ref we discussed above. The arrow shows us which branch the symbolic reference points to. Remember, `origin/main` is the same as `remotes/origin/main`, which is the same as `refs/remotes/origin/main`.

[3] We learned about `HEAD` in a previous session. `HEAD` is also a symbolic reference. We used the same definition but did not explicitly call it a symbolic reference back then.

Session 05: Rebase - Part 1

10 Rebasing

We will first discuss what rebasing is and how it compares to merging. Since rebasing rewrites history, we cover when it is safe to use and when not. We then provide some examples of rebasing.

The starting point for this session is where we left off last time. We have a simple repository with one initial commit, Alice pushed the repository to GitHub and Bob cloned it. One can also create a new local repository to follow along, we will not push to GitHub in this session.

10.1 What is Rebasing

Rebasing is one of two ways to integrate changes from one branch into another. Merging is the other way. Instead of creating a merge commit to join the two histories, rebasing reapplies the commits from one history, one by one, in the order they were introduced, on top of another history.

10.2 When Not to Rebase

As we mentioned in a previous session when we looked at `git commit --amend`, we have to be careful when we rewrite history. That leaves us with the following question: When is it safe to rebase, and when not?
The rules are as follows:

- Rebase is **safe** if
  - We rebase local commits that have never been pushed to a server.
  - We rebase commits that have been pushed to a server but no other collaborator has based commits from.
- Rebase is **dangerous** if we rebase commits that have been pushed and some other collaborator has based commits from.

Since it can be hard to know whether other people have created commits based on commits that are on the remote, it is a good rule to avoid doing rebase on pushed commits if we are not entirely sure about that.

10.3 Example 1: Basic Rebase

In the first example, we will create another commit on the `main` branch before deciding to create a `feature` branch from the first commit. Then, we create a commit on the `feature` branch. The final goal is to reapply the commit from the `feature` branch onto the `main` branch to achieve a clean working history.

Let's set up the experiment; we first create a commit on the `main` branch:

term 68

```
alice main > echo "Some info" >> README.md
alice main > git add README.md
alice main > git commit -m "Add info to README"
[main 644dd8b] Add info to README
 1 file changed, 1 insertion(+)
alice main > git log --oneline
644dd8b (HEAD -> main) Add info to README
d506edd (origin/main) Init
```

Then, we create the `feature` branch starting at commit `d506edd` and finally create a commit on that branch:

term 69

```
alice main > git switch -c feature d506edd
Switched to a new branch 'feature'
alice feature > echo "feature A" > featureA.txt
alice feature > git add featureA.txt
alice feature > git commit -m "Implement feature A"
[feature 6062965] Implement feature A
 1 file changed, 1 insertion(+)
 create mode 100644 featureA.txt
alice feature > git log --all --graph --oneline
```

```
* 6062965 (HEAD -> feature) Implement feature A
| * 644dd8b (main) Add info to README
|/
* d506edd (origin/main) Init
```

We can also visualize our history horizontally to showcase our starting point and what we want to achieve:

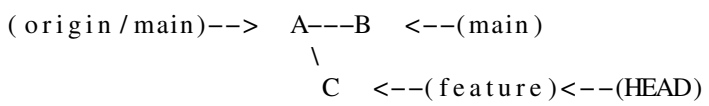

Figure 2: Current history[1]

Where `A` is commit `d506edd`, `B` is `644dd8b` (`main`), and `C` is `6062965` (`feature`). Our goal after rebasing should look as follows:

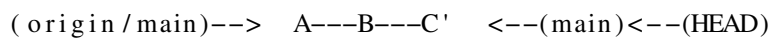

Figure 3: Goal after rebasing

Where `C'` is the new commit with the same contents as commit `C`.

10.3.1 Understanding Steps of Basic Rebase

First, let's think about what rebase will do. As we alluded to in section 10.1, rebasing replays commits from one line of work onto another. Naturally, it is important to know which is the "from" and which is the "onto". Taking a look at the synopsis of `man git-rebase` and not taking into account the optional options, we end up with the simplified syntax `git rebase [<upstream> [<branch>]]`. Next, we read the following in the description:

> If **<branch>** is specified, **git rebase** will perform an automatic **git switch <branch>** before doing anything else. Otherwise it remains on the current branch.

That means that `<branch>` is only needed if we are too lazy to manually switch to the branch before we execute the rebase. That also gives us the first part of our solution, "from" here is `<branch>` or the current branch if we do not specify it.

`<upstream>`, on the other hand, is the tip we want to reapply the changes onto. That answers what is "onto". Since `<upstream>` is also optional, there is a default. The default is the upstream (if configured) of the current branch.

For example, let's say we are on `main` and set `main` to track the upstream branch `refs/heads/main` on remote `origin`. Then, `git rebase` would take the changes on `main` and reapply them onto `origin/main`, which, remember, is our local branch reference for `origin`'s `refs/heads/main`. By the way, this is exactly how we set up `main` previously, as we can observe from the config file:

term 70

```
alice feature > cat .git/config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
    ignorecase = true
    precomposeunicode = true
[user]
    name = Alice
    email = alice@example.com
[remote "origin"]
    url = git@github.com:JoshuaGloor/learning-remotes.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
    remote = origin
    merge = refs/heads/main
```

[1] Yes, I know these graphics are very fancy...

This default makes sense because we likely do some commits locally before fetching work from the remote, which might move the corresponding remote-tracking branch forward (if someone pushed work to the remote since we last communicated with it). Then, we would like to apply our changes onto the tip of the remote-tracking branch, which we could do by only executing the command `git rebase`.

Now we know what the basic parameters of rebase are and which correspond to "from" and "onto". Let's discuss what rebase does before we execute the command.

(i) Git will go to the common ancestor of the branch we are on ("from") and the branch we are rebasing onto. We can find the common ancestor of two commits[2] using the `git merge-base` command, as shown in term 71.

(ii) Git then saves the diffs introduced by each commit of the current branch and saves them to temporary files.

(iii) It resets the current branch to the commit the "onto" branch points at.

(iv) Finally, it reapplies each change in the order they were introduced previously.

term 71

```
alice feature > git merge-base feature main
d506eddd2fa2ae85f7891e2b3b473f779dfcfd93
```

10.3.2 Executing Basic Rebase

We understand what rebase will do and how we have to write the command, so let's execute it:

term 72

```
alice feature > git rebase main
Successfully rebased and updated refs/heads/feature.
alice feature > git log --all --graph --oneline
* a310466 (HEAD -> feature) Implement feature A
* 644dd8b (main) Add info to README
* d506edd (origin/main) Init
```

As we can see from the `git log` output, we have almost achieved our goal, as described in Figure 3. Also, notice that the previous SHA-1 hash, `6062965`, has changed to `a310466`.

The only thing left to do is to fast-forward `main` and delete the `feature` branch since we do not need it anymore:

term 73

```
alice feature > git switch main
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)
alice main > git merge feature
Updating 644dd8b..a310466
Fast-forward
 featureA.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 featureA.txt
alice main > git branch -d feature
Deleted branch feature (was a310466).
alice main > git log --all --graph --oneline
* a310466 (HEAD -> main) Implement feature A
* 644dd8b Add info to README
* d506edd (origin/main) Init
```

We end up with a clean and linear history as desired.

[2]Note, this is only true for two commits; to find a common ancestor for more than two commits, one needs to additionally use the `--octopus` flag, see `man git-merge-base`.

10.4 Example 2: Onto Rebase

In this example[3], we will create an `experiment` branch starting from the parent commit of the commit that "implemented feature A". Assume that the experiments lead us to have the idea to implement "feature A" in another way, taking into consideration the things on the `experiment` branch. We create a branch `feature` from `experiment` and then implement `featureA.txt` (we do this to show how to resolve conflicts). We then perform some more experiments.

Then, we conclude that the new `featureA` is better than the old one. We also notice that the things on `experiment` are not needed for the better `featureA.txt` to work. We thus want to rebase `feature` (without `experiment`) onto `main`.

We set up the experiment according to the above idea as follows:

term 74

```
alice main > git switch -c experiment 644dd8b
Switched to a new branch 'experiment'
alice experiment > echo "Experiment 1" > experiment.txt
alice experiment > git add experiment.txt
alice experiment > git commit -m "Do experiment 1"
[experiment a099164] Do experiment 1
 1 file changed, 1 insertion(+)
 create mode 100644 experiment.txt
alice experiment > git switch -c feature
Switched to a new branch 'feature'
alice feature > echo "feature **A**" > featureA.txt
alice feature > git add featureA.txt
alice feature > git commit -m "Implement feature **A**"
[feature 91f2556] Implement feature **A**
 1 file changed, 1 insertion(+)
 create mode 100644 featureA.txt
alice feature > git switch experiment
Switched to branch 'experiment'
alice experiment > echo "Experiment 2" >> experiment.txt
alice experiment > git add experiment.txt
alice experiment > git commit -m "Do experiment 2"
[experiment 9e3aa3f] Do experiment 2
 1 file changed, 1 insertion(+)
```

After all these commands, we arrive at our desired scenario:

term 75

```
alice experiment > git log --all --graph --oneline
* 9e3aa3f (HEAD -> experiment) Do experiment 2
| * 91f2556 (feature) Implement feature **A**
|/
* a099164 Do experiment 1
| * a310466 (main) Implement feature A
|/
* 644dd8b Add info to README
* d506edd (origin/main) Init
```

We can again visualize our history horizontally to showcase our starting point and what we want to achieve:

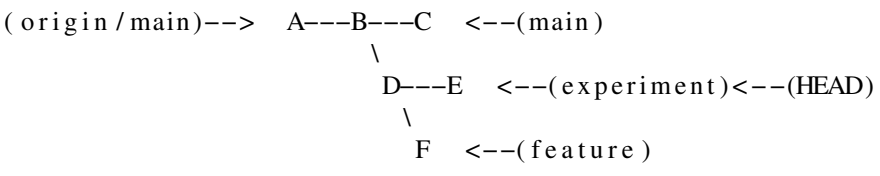

Figure 4: Current history

[3] Yes, I know this scenario may not make much sense, but bear with me; the scenario does not have to make sense. In the end, our priority is to learn about this rebase case.

Where `A`, `B`, and `C` are commits `d506edd`, `644dd8b`, and `a310466` (`main`), respectively. `D` and `E` are commits `a099164` and `9e3aa3f` (`experiment`), respectively. `F` is commit `91f2556` (`feature`). Our goal can be visualized in the following picture:

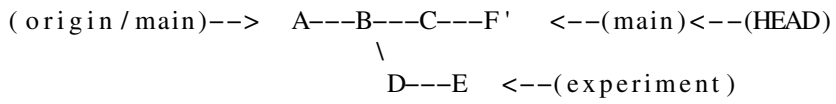

Figure 5: Goal after rebasing

Where `F'` is the new commit with the same contents as commit `F`.

10.4.1 Understanding Steps of Onto Rebase

First, it is important to understand why we have a different rebase here. Let's find out why we cannot use the same command as in term 72.

There is a crucial line in the manual (`man git-rebase`) that provides us with the desired information. The sentence describes which commits are reapplied later onto the desired base tip.

> All changes made by commits in the current branch but that are not in **<upstream>** are saved to a temporary area. This is the same set of commits that would be shown by `git log <upstream>..HEAD;`

The first sentence explains that the commits to be reapplied are determined by the supplied `<upstream>` parameter. The second sentence tells us how to check which commits will be included. The double dot notation is a shorthand for finding the range of commits that are reachable from `HEAD` but not reachable from `<upstream>`.

Assuming we are on the `feature` branch and run `git rebase main`, as we did in section 10.3.1 for the basic rebase example, we would reapply all commits from the range `main..feature` to the `main` branch. Let's see what this range is:

term 76

```
alice feature > git log --oneline main..HEAD
91f2556 (HEAD -> feature) Implement feature **A**
a099164 Do experiment 1
```

Aha! We would also try to reapply `a099164`, but that is not what we want. That is the difference between this rebase and the basic one.

To only apply commits that are part of the `feature` branch but not the `experiment` branch, we must use the `--onto` option. The branch (or, more generally, the commit) we specify after the `--onto` option will be the base tip we reapply the commits onto. So, now "onto" is not determined by `<upstream>` anymore, and we can use `<upstream>` for the purpose of including the commits we want to save to the temporary file (to reapply later). In our case, we want all commits reachable from `feature` that are not reachable from `experiment` (`experiment..feature`). This means we choose `experiment` as our `<upstream>`. We also specify `<branch>` this time to showcase the behavior.

To summarize:

- We only want commits that are reachable from `feature`, but not `experiment`. This means we need `<upstream>` to be `experiment` and `<branch>` to be `feature`.
- We want to reapply these changes onto `main`. Since `main` differs from the branch we chose for `<upstream>`, we must supply `--onto main`.

10.4.2 Executing Onto Rebase

Using the insights from both section 10.3.1 and section 10.4.1, let's execute the command:

term 77

```
alice experiment > git rebase --onto main experiment feature
Auto-merging featureA.txt
CONFLICT (add/add): Merge conflict in featureA.txt
error: could not apply 91f2556... Implement feature **A**
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
```

```
hint:  To abort and get back to the state before "git rebase", run "git
rebase --abort".
Could not apply 91f2556...  Implement feature **A**
```

Whoops, looks like we have a conflict. Let's learn how to resolve it.

To find out more, our trusted helper `git status` can give us more information:

term 78

```
alice (git)-[feature|rebase-i]- > git status
interactive rebase in progress; onto a310466
Last command done (1 command done):
   pick 91f2556 Implement feature **A**
No commands remaining.
You are currently rebasing branch 'feature' on 'a310466'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

Unmerged paths:
  (use "git restore --staged <file>..." to unstage)
  (use "git add <file>..." to mark resolution)
        both added:    featureA.txt

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

`git status` shows us that the conflicts happened in the `featureA.txt` file (we find that information under "unmerged paths"). It also tells us what we have to do after resolving the conflict (this information was already provided by the output when the rebase failed in term 77).

Let's edit the file with Vim to resolve the conflict. Opening the file, we find:

term 79: In Vim

```
<<<<<<< HEAD
feature A
=======
feature **A**
>>>>>>> 91f2556 (Implement feature **A**)
```

We edit the file and remove the conflict markers. In our case, we only want to keep the line "feature **A**". We save and quit Vim.

Then, we continue as previously discussed; we add the file and execute `git rebase --continue`, as Git so nicely told us in term 77.

term 80

```
alice (git)-[feature|rebase-i]- > git add featureA.txt
alice (git)-[feature|rebase-i]- > git rebase --continue
[detached HEAD 60d4853] Implement feature **A**
 1 file changed, 1 insertion(+), 1 deletion(-)
Successfully rebased and updated refs/heads/feature.
```

And we are done. Note that the editor (in our case, Vim) opened up when we run `git rebase --continue` to edit the commit message (because of the merge conflict). In our case, we left the message as is. Now, our history looks as follows:

term 81

```
alice feature > git log --all --graph --oneline
* 60d4853 (HEAD -> feature) Implement feature **A**
* a310466 (main) Implement feature A
| * 9e3aa3f (experiment) Do experiment 2
| * a099164 Do experiment 1
|/
```

```
* 644dd8b Add info to README
* d506edd (origin/main) Init
```

We can now fast-forward the `main` branch and delete the `feature` branch since we do not need it anymore.

term 82

```
alice feature > git switch main
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 2 commits.
  (use "git push" to publish your local commits)
alice main > git merge feature
Updating a310466..60d4853
Fast-forward
 featureA.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
alice main > git branch -d feature
Deleted branch feature (was 60d4853).
alice main > git log --all --graph --oneline
* 60d4853 (HEAD -> main) Implement feature **A**
* a310466 Implement feature A
| * 9e3aa3f (experiment) Do experiment 2
| * a099164 Do experiment 1
|/
* 644dd8b Add info to README
* d506edd (origin/main) Init
```

We end up with the desired history, as shown in Figure 4.

Session 06: Rebase - Part 2

11 Key Considerations When Rebasing

In this session, we will look at why changing history can be dangerous and how to fix it.

The starting point for this session is where we left off last time. However, we do not need all the commits from the last session; it would just clutter our output. We perform a hard reset to `origin/main` and force delete the `experiment` branch, so we are left with the initial commit only.

term 83

```
alice main > git reset --hard origin/main
HEAD is now at d506edd Init
alice main > git branch -D experiment
Deleted branch experiment (was 9e3aa3f).
alice main > git log --all --oneline
d506edd (HEAD -> main, origin/main) Init
alice main > git fetch origin
alice main >
```

As we can see, nothing has changed on the remote, either. The same is true for Bob:

term 84

```
bob main > git log --all --oneline
d506edd (HEAD -> main, origin/main, origin/HEAD) Init
```

11.1 Dangers of Rebasing

We will demonstrate the dangers of rebasing and simultaneously learn how to remove a range of commits using rebasing, what `pull.rebase` does, and why it is generally better to avoid using `git pull` altogether.

Let's say Alice implements several features on the `main` branch and commits after each one. Because this simple example setup repeats commands, we use some basic shell scripting to make our lives easier. However, of course, this can be done manually without a loop, too.

term 85

```
alice main > for c in "A" "B" "C" "D"; do
  f="feature${c}.txt"
  echo "feature $c" > $f
  git add $f
  git commit -m "Implement feature $c"
done
[main 3442e7b] Implement feature A
 1 file changed, 1 insertion(+)
 create mode 100644 featureA.txt
[main 9eb88c8] Implement feature B
 1 file changed, 1 insertion(+)
 create mode 100644 featureB.txt
[main 88e6df2] Implement feature C
 1 file changed, 1 insertion(+)
 create mode 100644 featureC.txt
[main 2ad5243] Implement feature D
 1 file changed, 1 insertion(+)
 create mode 100644 featureD.txt
```

We end up with the below history, which we push to the remote:

term 86

```
alice main > git log --all --graph --oneline
* 2ad5243 (HEAD -> main) Implement feature D
* 88e6df2 Implement feature C
* 9eb88c8 Implement feature B
```

```
* 3442e7b Implement feature A
* d506edd (origin/main) Init
alice main > git push
Enumerating objects: 13, done.
Counting objects: 100% (13/13), done.
Delta compression using up to 8 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (12/12), 932 bytes | 932.00 KiB/s, done.
Total 12 (delta 3), reused 4 (delta 1), pack-reused 0
remote: Resolving deltas: 100% (3/3), done.
To github.com:JoshuaGloor/learning-remotes.git
   d506edd..2ad5243  main -> main
```

Then, Bob pulls from origin and implements another feature.

term 87

```
bob main > git pull
remote: Enumerating objects: 13, done.
remote: Counting objects: 100% (13/13), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 12 (delta 3), reused 12 (delta 3), pack-reused 0
Unpacking objects: 100% (12/12), 912 bytes | 130.00 KiB/s, done.
From github.com:JoshuaGloor/learning-remotes
   d506edd..2ad5243  main       -> origin/main
Updating d506edd..2ad5243
Fast-forward
 featureA.txt | 1 +
 featureB.txt | 1 +
 featureC.txt | 1 +
 featureD.txt | 1 +
 4 files changed, 4 insertions(+)
 create mode 100644 featureA.txt
 create mode 100644 featureB.txt
 create mode 100644 featureC.txt
 create mode 100644 featureD.txt
bob main > echo "feature E" > featureE.txt
bob main > git add featureE.txt
bob main > git commit -m "Implement feature E"
[main 27148a1] Implement feature E
 1 file changed, 1 insertion(+)
 create mode 100644 featureE.txt
bob main > git log --all --graph --oneline
* 27148a1 (HEAD -> main) Implement feature E
* 2ad5243 (origin/main, origin/HEAD) Implement feature D
* 88e6df2 Implement feature C
* 9eb88c8 Implement feature B
* 3442e7b Implement feature A
* d506edd Init
```

As we can see, Bob has committed work on top of Alice's work. According to the rules we discussed in the previous session, it would be dangerous for Alice to do a rebase on the `main` branch, but we will do exactly that to see what happens.

11.1.1 Another Rebase Example

Alice decides that feature B and C are not needed, so she wants to remove them. We can also leverage rebasing to remove commits.

To figure out the correct command, it helps to describe what we want to achieve in more detail.

- We want to delete the commits that introduced features B and C.
- We want to keep the commits that implemented features A and D.

Therefore, we take the commit that implemented feature D and reapply it on top of the commit that introduced feature A. Let's find out what the different parameters are:

(i) `<branch>`: We currently only have remote-tracking branches and the `main` branch. Since we are on the `main` branch, we can omit this parameter.

(ii) `<upstream>`: Which changes do we want to reapply? It is only commit `27148a1`. We must choose `<upstream>` such that all changes made by `main` that are not in `<upstream>` are equal to the commit `27148a1`. That is, we look for a commit such that `<upstream>..main` equals `27148a1`. This commit is the parent of the one we want to keep (`HEAD`). Therefore, `<upstream>` is `88e6df2` or equivalently, `HEAD~`.

(iii) `<newbase>`: This is the parameter we pass if we specify the `--onto` option. First, let's find out if we even need the `--onto` option. We do this by considering whether `<upstream>` is the correct place to reapply the commits. That is not the case because we want to reapply the changes onto the commit that implemented feature A, i.e., commit `3442e7b`. We can also use the "parent suffix" syntax again to refer to this commit as `HEAD~3`.

We end up with the command `git rebase --onto HEAD~3 HEAD~`.

Now we know everything to execute the rebase:

term 88

```
alice main > git rebase --onto HEAD~3 HEAD~
Successfully rebased and updated refs/heads/main.
alice main > git log --all --graph --oneline
* bc1c4f3 (HEAD -> main) Implement feature D
| * 2ad5243 (origin/main) Implement feature D
| * 88e6df2 Implement feature C
| * 9eb88c8 Implement feature B
|/
* 3442e7b Implement feature A
* d506edd Init
```

That is exactly what we wanted; the `main` branch no longer has the commits that implement features B and C.

11.1.2 Pushing Changed History

We try to push the changes made to the remote:

term 89

```
alice main > git push
To github.com:JoshuaGloor/learning-remotes.git
 ! [rejected]          main -> main (non-fast-forward)
error: failed to push some refs to 'github.com:JoshuaGloor/learning-
remotes.git'
hint: Updates were rejected because the tip of your current branch is
hint: behind its remote counterpart. If you want to integrate the remote
hint: changes, use 'git pull' before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

It should not surprise us too much that Git rejects this push attempt. As the output reads, we are trying to do a push that does not result in a fast-forward. However, let's assume Alice does not care about her collaborators (or the warning) and force pushes her changes:

term 90

```
alice main > git push -f
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 320 bytes | 320.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:JoshuaGloor/learning-remotes.git
+ 2ad5243...bc1c4f3 main -> main (forced update)
alice main > git log --all --graph --oneline
* bc1c4f3 (HEAD -> main, origin/main) Implement feature D
```

```
* 3442e7b Implement feature A
* d506edd Init
```

11.1.3 Pulling Changed History

Then, Bob pulls in the latest changes from the remote:

term 91

```
bob main > git pull
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 300 bytes | 100.00 KiB/s, done.
From github.com:JoshuaGloor/learning-remotes
 + 2ad5243...bc1c4f3 main       -> origin/main  (forced update)
hint: You have divergent branches and need to specify how to reconcile
hint: them. You can do so by running one of the following commands sometime
hint: before your next pull:
hint:
hint:   git config pull.rebase false  # merge
hint:   git config pull.rebase true   # rebase
hint:   git config pull.ff only       # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set a
hint: default preference for all repositories. You can also pass --rebase,
hint: --no-rebase, or --ff-only on the command line to override the
hint: configured default per invocation.
fatal:  Need to specify how to reconcile divergent branches.
```

Let's first look at the log, which helps us discuss the hints in term 91.

term 92

```
bob main > git log --all --graph --oneline
* bc1c4f3 (origin/main, origin/HEAD) Implement feature D
| * 27148a1 (HEAD -> main) Implement feature E
| * 2ad5243 Implement feature D
| * 88e6df2 Implement feature C
| * 9eb88c8 Implement feature B
|/
* 3442e7b Implement feature A
* d506edd Init
```

As the hint in term 91 states, we can see that we have a divergent history. By default, `git pull` does a `git fetch` and then a `git merge`, but it performs the merge only if it is a fast-forward. If it is not a fast-forward, which is the case in our example, Git asks us to decide how to handle divergent histories in future pulls.

11.1.4 Do Not Rebase and Merge

We will now discuss why it makes sense not to mix rebasing and non-fast-forward merging.

If we create a merge commit to combine the divergent branches, we not only have a duplicate commit (two commits that implement feature D and are equivalent up to their commit hash), but we also reintroduce feature B and C into the history. However, we do not want them in our `main` branch's history anymore (because Alice deleted them, and we agree with the removal).

We should stick to rebasing because it can solve the duplication problem. If the SHA-1 changes because of rebasing, but the file diffs of the introduced changes are the same[1], Git recognizes that and avoids duplicating the commits during a rebase.

[1] It checks this by using a sum of SHA-1 of the file diffs associated with the changes that have been introduced. This checksum is called a "patch ID".

11.1.5 Do Not Just Blindly Set pull.rebase

Next, we examine what we should know before setting `pull.rebase` to `true`[2]. That will also serve as an example of why using `git fetch` followed by `git rebase` (or `git merge`) is preferable to `git pull`.

In general, `git fetch` followed by `git rebase` (or `git merge`) is preferable because it shows us what has changed on the remote before integrating these changes into our history. However, when it comes to `pull.rebase`, there is a bit more to it. The rebase performed might not be the one we want or expect at first. To showcase this, let's see the difference with Bob's repository when we execute a rebase by specifying `<upstream>` and one where we use the default[3].

To quickly return to the current state of our history, we will copy the commit `main` currently points at (`27148a1`, in our case). We "just remember" the commit hash because we have not covered the reflog yet and do not want a branch (or tag) showing up in our `git log` output. So, if you want to replicate the below commands and are unfamiliar with the reflog, save the SHA-1 your `main` branch currently points to.

Remember, our current view of our repository looks like term 92. If we execute `git rebase` without any parameter, which is what `git pull` would do, we get:

term 93

```
bob main > git rebase
Successfully rebased and updated refs/heads/main.
bob main > git log --all --graph --oneline
* 5d5f4d3 (HEAD -> main) Implement feature E
* bc1c4f3 (origin/main, origin/HEAD) Implement feature D
* 3442e7b Implement feature A
* d506edd Init
```

Notice that we have no duplicate commits, but the commits corresponding to features B and C disappeared. Let's see what happens when we execute the rebase and include the `<upstream>` parameter.

term 94

```
bob main > git reset --hard 27148a1
HEAD is now at 27148a1 Implement feature E
bob main > git log --all --graph --oneline
* bc1c4f3 (origin/main, origin/HEAD) Implement feature D
| * 27148a1 (HEAD -> main) Implement feature E
| * 2ad5243 Implement feature D
| * 88e6df2 Implement feature C
| * 9eb88c8 Implement feature B
|/
* 3442e7b Implement feature A
* d506edd Init
bob main > git rebase origin/main
warning:  skipped previously applied commit 2ad5243
hint:  use --reapply-cherry-picks to include skipped commits
hint:  Disable this message with "git config advice.skippedCherryPicks
hint:  false"
Successfully rebased and updated refs/heads/main.
```

We see that we received a warning this time, which already gives us a hint that the two rebases were not equivalent. Let's check the commit logs.

term 95

```
bob main > git log --all --graph --oneline
* ae35512 (HEAD -> main) Implement feature E
* 3b94041 Implement feature C
* b1220f7 Implement feature B
* bc1c4f3 (origin/main, origin/HEAD) Implement feature D
* 3442e7b Implement feature A
* d506edd Init
```

[2] As discussed in section 11.1.4, setting `pull.rebase` to `false` is also not a good idea (do not mix rebasing and merging). If you still want to use `git pull`, it is probably best to go with the third option, adding the config `pull.ff only` to only execute fast-forward merges automatically.

[3] The default `<upstream>`, as discussed here, only exists if an upstream branch for the given branch (`main`, in our case) is configured. We did this when we set up the repository and configured `main` to track `origin/main`.

Interestingly, `git rebase origin/main` reapplied the commits for features B and C, whereas `git rebase` did not. Also, notice that neither produced a duplicate commit for feature D.

To explain this behavior, we must know the option called `--fork-point` for `git rebase`. In the description of the manual (`man git-rebase`) we find the following:

> If **<upstream>** is not specified, the upstream configured in **branch.<name>.remote** and **branch.<name>.merge** options will be used (see **git-config**(1) for details) and the **--fork-point** option is assumed.

That means if no `<upstream>` is specified, the default `<upstream>` will be used together with the `--fork-point` option. Therefore, when we run `git rebase`, we execute `git rebase --fork-point origin/main`, and when we execute `git rebase origin/main`, we simply execute `git rebase origin/main` (i.e., without the `--fork-point` option).

So, what is fork-point? Quoting the manual (`man git-rebase`) again:

> Use reflog to find a better common ancestor between **<upstream>** and **<branch>** when calculating which commits have been introduced by **<branch>**.
>
> When **--fork-point** is active, fork_point will be used instead of **<upstream>** to calculate the set of commits to rebase, where fork_point is the result of **git merge-base --fork-point <upstream> <branch>** command (see **git-merge-base**(1)).

Aha, so `--fork-point` uses the reflog to find a better commit to calculate the set of commits to rebase.

Let's first reset our repository back to our previous state before we run any rebase:

term 96

```
bob main > git reset --hard 27148a1
HEAD is now at 27148a1 Implement feature E
* bc1c4f3 (origin/main, origin/HEAD) Implement feature D
| * 27148a1 (HEAD -> main) Implement feature E
| * 2ad5243 Implement feature D
| * 88e6df2 Implement feature C
| * 9eb88c8 Implement feature B
|/
* 3442e7b Implement feature A
* d506edd Init
```

Let's observe the difference between `<upstream>` and `fork-point`. First, we check which commits we save to the temporary file when we explicitly specify the `<upstream>`:

term 97

```
bob main > git log --oneline origin/main..HEAD
27148a1 (HEAD -> main) Implement feature E
2ad5243 Implement feature D
88e6df2 Implement feature C
9eb88c8 Implement feature B
```

That shows that when we run `git rebase origin/main`, all commits that implemented features B, C, D, and E are reapplied onto `origin/main`. Notice here that we did not end up with a duplicate commit for feature D because thanks to the patch ID[4], that commit is skipped (as indicated in the warning received in term 94).

Let's now look at the `fork-point`:

term 98

```
bob main > git merge-base --fork-point origin/main main
2ad52435c0f15d495ed22e9c02d8fca0c8309d90
bob main > git log 2ad5243..HEAD --oneline
27148a1 (HEAD -> main) Implement feature E
```

As we can see, when `fork-point` is picked instead of `<upstream>`, we only end up with the commit `27148a1` to reapply onto `origin/main`. That explains why the commits corresponding to features B and C disappear after the rebase

[4] See appendix A for how to check the equivalence of a patch ID.

and why we did not get a warning that the commit implementing feature D has been skipped (because we did not try to reapply that commit in the first place).

We can now answer why we should be aware of the consequences when we set `pull.rebase` to `true`. In our case, we played the roles of both Alice and Bob, but imagine you pull a repository and potentially, without you noticing, commits just disappear. That is why it is preferable to first fetch and then rebase (or merge). We can not only control which kind of rebase we want to run, but we can also observe what has changed before we run a rebase. For example, Alice might have decided that features B and C can be deleted, but we might disagree.

In our case, we want the deleted commits to also be deleted on Bob's side, so we do the rebase that does that:

term 99

```
bob main > git rebase
Successfully rebased and updated refs/heads/main.
```

11.1.6 Final Thoughts

In this example, we were lucky, and resolving the rebased changes was easy. However, this is not always the case. Things can get messy fast, that is one reason why Git repeatedly mentions in its docs that the user should be aware of the dangers of rebasing and know what they are doing.

Session 07: Rebase - Part 3

12 Interactive Rebasing and Cherry-Picking

In this session, we continue our discussion around rebasing. We will look at different things we can do during interactive rebasing, such as splitting a commit into two, squashing, reordering commits, etc. We will also quickly cover cherry-picking.

Since we will not use remotes in this session, we will create a new local Git repository for the subsequent examples. After the setup, our history will look as shown in Figure 6.

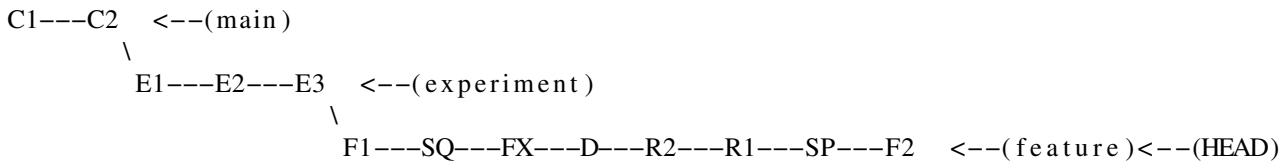

Figure 6: After setting up the experiment

We will add the names we use in Figure 6 to the start of the commit messages to make it easier to recognize the commits during the discussion. Interactive rebasing is often showcased by rebasing the commits from the current line of work without rebasing them onto another tip. However, we can also do an interactive rebase and change the commits before they are reapplied onto another tip, which is analogous to what we discussed in the previous sessions. Therefore, we combine the revision of what we know about rebasing while simultaneously showcasing many features of interactive rebasing.

Our goal is to interactively rebase the commits that are part of the `feature` branch but not in the `experiment` branch onto `main`. To showcase what can be done using the interactive rebase, we will perform the following changes during the rebase:

- Quash commit `SQ` and fixup commit `FX` into `F1`, producing commit `F1'`.
- Delete commit `D`.
- Reorder commit `R2` and `R1` such that after the rebase, `R1'` happened before `R2'`.
- Split commit `SP` into two, resulting in `SP1` and `SP2`.

After the rebase, we desire the history to look like Figure 7.

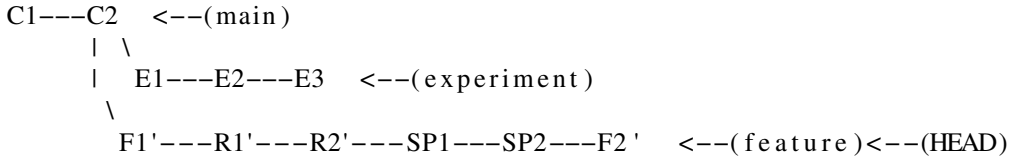

Figure 7: Goal after interactive rebase

Then, we will discuss cherry-picking and cherry-pick commits `E2`, `R1'`, `R2'`, and `SP1` onto `main`. This way, we can showcase that we can cherry-pick both single commits and ranges of commits.

The final history should look like Figure 8.

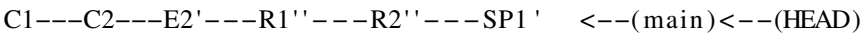

Figure 8: Goal after cherry-picking and deleting obsolete branches

Let's create our example history from Figure 6. We will again use some simple Bash scripting to speed up the setup.

term 100: Setup

```
> create_simple_commit() {
  echo $1 >> $2
  git add $2
  git commit -m $1
}
> git init
```

```
main > create_simple_commit "C1" "c.txt" && create_simple_commit "C2"
"c.txt"
[main (root-commit) c109b12] C1
 1 file changed, 1 insertion(+)
 create mode 100644 c.txt
[main 6e304bb] C2
 1 file changed, 1 insertion(+)
main > git switch -c experiment
Switched to a new branch 'experiment'
experiment > for i in {1..3}; do create_simple_commit "E$i" "e.txt"; done
[experiment d83399e] E1
 1 file changed, 1 insertion(+)
 create mode 100644 e.txt
[experiment cee51e4] E2
 1 file changed, 1 insertion(+)
[experiment 01406a5] E3
 1 file changed, 1 insertion(+)
experiment > git switch -c feature
Switched to a new branch 'feature'
feature > for i in "F1" "SQ" "FX" "D"; do
  create_simple_commit "$i" "f.txt"
done
[feature df98b1b] F1
 1 file changed, 1 insertion(+)
 create mode 100644 f.txt
[feature 3950e36] SQ
 1 file changed, 1 insertion(+)
[feature 8edcd5a] FX
 1 file changed, 1 insertion(+)
[feature c5facae] D
 1 file changed, 1 insertion(+)
feature > create_simple_commit "R2" "r2.txt"
[feature 783974a] R2
 1 file changed, 1 insertion(+)
 create mode 100644 r2.txt
feature > create_simple_commit "R1" "r1.txt"
[feature 887a6fb] R1
 1 file changed, 1 insertion(+)
 create mode 100644 r1.txt
feature > echo "SP1" >> sp1.txt
feature > echo "SP2" >> sp2.txt
feature > git add sp*
feature > git commit -m "SP"
[feature 4ca7108] SP
 2 files changed, 2 insertions(+)
 create mode 100644 sp1.txt
 create mode 100644 sp2.txt
feature > create_simple_commit "F2" "f.txt"
[feature 345157c] F2
 1 file changed, 1 insertion(+)
```

After all this, we arrive at our desired starting point.

term 101

```
feature > git log --all --graph --oneline
* 345157c (HEAD -> feature) F2
* 4ca7108 SP
* 887a6fb R1
* 783974a R2
* c5facae D
* 8edcd5a FX
* 3950e36 SQ
```

```
* df98b1b F1
* 01406a5 (experiment) E3
* cee51e4 E2
* d83399e E1
* 6e304bb (main) C2
* c109b12 C1
```

12.1 Yet Another Rebase Example

In this section, we will again revise how we can derive the correct rebase command. Afterward, in section 12.2, we will execute and discuss the interactive rebase, and in section 12.3, we will cover cherry-picking. Analogous to the other rebase examples, we first think about what we want to do, and then we gradually determine the parameters of the command.

- We want to rebase commits that are on the `feature` branch only, i.e., only rebase commits between `F1` and `F2`, inclusive.
- We want to interactively change some of these commits and then reapply the changes onto `main`.

Let's find out what the different parameters[1] are:

(i) `<branch>`: From where do we want to take the commits from?
As mentioned above, we want to rebase commits between `F1` and `F2`, inclusive. Therefore, we want to take the commits from the `feature` branch. Since `HEAD` already points to the `feature` branch, we do not need to specify `<branch>`. However, for completeness, if we had been on another branch, we would have used `feature` for `<branch>`.

(ii) `<upstream>`: Which commits do we want to save to the temporary file (to reapply them later)?
We want all the commits from `feature` that are not also commits from `<upstream>` (`<upstream>..feature`). In our case, we do not want the commits that are also part of `experiment`. Therefore, `<upstream>` is `experiment`.

(iii) `<newbase>`: Is `<upstream>` also the tip where we want to reapply the commits saved in the temporary file onto?

(a) If yes, we are done and do not need `<newbase>`.

(b) Otherwise, where do we want to apply the commits onto?

Since we want to reapply the commits onto `main`, but `experiment` and `main` do not point to the same commit, we need to use the `--onto` option and choose `<newbase>` as `main`.

Putting the building blocks together and using the `-i` option to make the rebase interactive, we end up with the following command:

```
git rebase -i --onto main experiment
```

12.2 Interactive Rebasing

Let's execute the rebase command we derived in section 12.1.

term 102

```
# Your favorite text editor will open up after executing the below command
feature > git rebase -i --onto main experiment
```

The text editor that opened up (in our case Vim) has the following content:

term 103: In Vim

```
pick df98b1b F1
pick 3950e36 SQ
pick 8edcd5a FX
pick c5facae D
pick 783974a R2
pick 887a6fb R1
pick 4ca7108 SP
pick 345157c F2
```

[1] Remember, these parameters are from the manual (`man git-rebase`); the steps describe a systematic approach for how to derive the correct values of these parameters.

```
# Rebase 01406a5..345157c onto 6e304bb (8 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                    commit's log message, unless -C is used, in which case
#                    keep only this commit's message; -c is same as -C but
#                    opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
#         create a merge commit using the original merge commit's
#         message (or the oneline, if no original merge commit was
#         specified); use -c <commit> to reword the commit message
# u, update-ref <ref> = track a placeholder for the <ref> to be updated
#                       to this position in the new commits. The <ref> is
#                       updated at the end of the rebase
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
```

Let's focus first on the last three lines of the commented-out section (lines starting with #).

(i) First, we have:

> These lines can be re-ordered; they are executed from top to bottom.

The commits are listed in reverse order compared to what we are used to from `git log`. So, the first (top-most) commit is the parent of the second commit, etc. The top lines are divided into command, SHA-1 hash, and part of the commit message. The part of the commit message is merely to help us better recognize the commits. Git only needs the command and commit hash to execute the line[2] correctly.

(ii) Next, we have:

> If you remove a line here THAT COMMIT WILL BE LOST.

Therefore, removing a line is the same as dropping (command `drop`) and commenting out the line using #.

(iii) And finally:

> However, if you remove everything, the rebase will be aborted.

That means, to abort the rebase at this point, we must either delete ALL the lines or comment them ALL out. Simply closing our editor will not abort the commit, as it would still execute the rebase. In this scenario, the commits between `F1` and `F2`, inclusive, would be reapplied onto `main`.

12.2.1 Commands Not Covered

Next, let's look at the commands. We will not cover all commands, but we will provide a bit more color on the commands that are not part of our example:

- `reword` allows us to rename the commit message while keeping the rest.

[2]There are also commands, e.g., `break`, that are standalone commands. They can be on a separate line without a commit hash.

- `exec` will execute a shell command after the commit above it (if one exists) is reapplied. That can be helpful when, e.g., we want to run tests after each reapplied commit. If the test script fails (i.e., returns with non-0 status), the rebase will stop, and we will have the chance to fix the problem.
- `break` will pause the rebase after the commit above it (if there existed one) is reapplied.
- Some commands are primarily used to retain merge commits during rebases. These commands include `label`, `reset`, and `merge`.
- `update-ref` allows us to create a branch which will point to the new commit hash that is generated when reapplying the commit that is listed in the line above this command.

12.2.2 Editing the Interactive Rebase Lines

Now, we will edit the file according to the goal we want to achieve. For that, the commit messages are helpful. After the edit, we have[3]:

term 104: In Vim

```
pick df98b1b F1
s 3950e36 SQ
f 8edcd5a FX
d c5facae D
p 887a6fb R1
p 783974a R2
e 4ca7108 SP
pick 345157c F2
```

We used the one-letter form of the commands to show which lines we changed.

- Commit `F1` stays as is.
- We squash commit `SQ` into `F1`. That means we end up with one commit at the end, and we will have the chance to merge the two commit messages.
- We fixup commit `FX` into commit `F1`. In this case, the commit message of `FX` will be deleted automatically.
- We delete commit `D`. Alternatively, we could have achieved the same result by either commenting out or deleting that line.
- We reorder commits `R2` and `R1`. Notice that they are now in the opposite order. There is no special "reorder" command; we simply reorder the commits together with the `pick` command.
- We want to edit commit `SP` so we can split it.
- Commit `F2` stays as is.

12.2.3 Executing Interactive Rebase

We start the rebase by saving and quitting the editor. Subsequently, we will show what happens while the interactive rebase progresses.

First, our editor opens up again:

term 105: In Vim

```
# This is a combination of 3 commits.
# This is the 1st commit message:

F1

# This is the commit message #2:

SQ

# The commit message #3 will be skipped:
```

[3]The commented out help section is ommitted for brevity.

```
# FX

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Sun Mar 17 16:28:21 2024 +0800
#
# interactive rebase in progress; onto 6e304bb
# Last commands done (3 commands done):
#    squash 3950e36 SQ
#    fixup 8edcd5a FX
# Next commands to do (5 remaining commands):
#    drop c5facae D
#    pick 887a6fb R1
# You are currently rebasing branch 'feature' on '6e304bb'.
#
# Changes to be committed:
#       new file:   f.txt
#
```

That allows us to edit the commit message for `F1'`. Note that we see the uncommented commit message of commit `FX` here, but this is only the case because we also squash a commit. If we only performed a fixup, the editor would not open. We edit the commit message to the below and then save and quit the editor again.

term 106: In Vim

```
# This is a combination of 3 commits.
# This is the 1st commit message:

fixup FX and squash SQ into F1

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Sun Mar 17 16:28:21 2024 +0800
#
# interactive rebase in progress; onto 6e304bb
# Last commands done (3 commands done):
#    squash 3950e36 SQ
#    fixup 8edcd5a FX
# Next commands to do (5 remaining commands):
#    drop c5facae D
#    pick 887a6fb R1
# You are currently rebasing branch 'feature' on '6e304bb'.
#
# Changes to be committed:
#       new file:   f.txt
#
```

Now is the first time we are not in Vim since we started the rebase. We see the following[4]:

term 107

```
feature > git rebase -i --onto main experiment
[detached HEAD 59c082a] fixup FX and squash SQ into F1
 Date: Sun Mar 17 16:28:21 2024 +0800
 1 file changed, 3 insertions(+)
 create mode 100644 f.txt
Stopped at 4ca7108...  SP
You can amend the commit now, with

  git commit --amend
```

[4]Keep in mind that we already ran the rebase command; we just have not seen the output yet because Vim opened up immediately.

```
Once you are satisfied with your changes, run

  git rebase --continue
(git)-[feature|rebase-i]- >
```

As we can see from the output, we stopped at commit `SP`. That means the reordering of `R1` and `R2` already happened. We can see that when checking the log:

term 108

```
(git)-[feature|rebase-i]- > git log --all --graph --oneline
* 0865f34 (HEAD) SP
* ed2c08d R2
* bd440a9 R1
* 59c082a fixup FX and squash SQ into F1
| * 345157c (feature) F2
| * 4ca7108 SP
| * 887a6fb R1
| * 783974a R2
| * c5facae D
| * 8edcd5a FX
| * 3950e36 SQ
| * df98b1b F1
| * 01406a5 (experiment) E3
| * cee51e4 E2
| * d83399e E1
|/
* 6e304bb (main) C2
* c109b12 C1
```

To split our commit into two, we must perform a reset. Previously, we covered the three trees. These are crucial for understanding which reset command we have to run. We do not want to go into too much detail about reset here; [CS14] already has an excellent section, “Reset Demystified”, that explains reset in depth. After reading that section, the argumentation and choice of reset command that follows will make a lot more sense.

In our case, we want to reset (move) the `HEAD` back by one and reset the index, but keep the working tree. That means we want to forget the commit `HEAD` currently points at[5] such that commit `R2` will be the parent of the next commit. At the same time, we also want to keep the changed files that were part of commit `SP` but unstage them. That is exactly what the default `git reset HEAD~` does (which is equivalent to `git reset --mixed HEAD~`).

term 109

```
(git)-[feature|rebase-i]- > git reset HEAD~
(git)-[feature|rebase-i]- > git status
interactive rebase in progress; onto 6e304bb
Last commands done (7 commands done):
   pick 783974a R2
   edit 4ca7108 SP
  (see more in file .git/rebase-merge/done)
Next command to do (1 remaining command):
   pick 345157c F2
  (use "git rebase --edit-todo" to view and edit)
You are currently editing a commit while rebasing branch 'feature' on
 '6e304bb'.
  (use "git commit --amend" to amend the current commit)
  (use "git rebase --continue" once you are satisfied with your changes)

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        sp1.txt
        sp2.txt
```

[5] Side note: here `HEAD` points directly at a commit; therefore, we currently have a detached `HEAD`. However, that is not important for the discussion.

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

As we can see, we can now edit the files and add them to the index again. That allows us to create two commits out of the previous one.

term 110

```
(git)-[feature|rebase-i]- > git add sp1.txt
i-rebase (git)-[feature|rebase-i]- > git commit -m "SP1"
[detached HEAD 65384dc] SP1
 1 file changed, 1 insertion(+)
 create mode 100644 sp1.txt
(git)-[feature|rebase-i]- > git add sp2.txt
(git)-[feature|rebase-i]- > git commit -m "SP2"
[detached HEAD 4bb266e] SP2
 1 file changed, 1 insertion(+)
 create mode 100644 sp2.txt
(git)-[feature|rebase-i]- > git log --all --graph --oneline
* 4bb266e (HEAD) SP2
* 65384dc SP1
* ed2c08d R2
* bd440a9 R1
* 59c082a fixup FX and squash SQ into F1
| * 345157c (feature) F2
| * 4ca7108 SP
| * 887a6fb R1
| * 783974a R2
| * c5facae D
| * 8edcd5a FX
| * 3950e36 SQ
| * df98b1b F1
| * 01406a5 (experiment) E3
| * cee51e4 E2
| * d83399e E1
|/
* 6e304bb (main) C2
* c109b12 C1
```

Since that is everything we wanted to do, we can continue our rebase:

term 111

```
(git)-[feature|rebase-i]- > git rebase --continue
Auto-merging f.txt
CONFLICT (content): Merge conflict in f.txt
error: could not apply 345157c... F2
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase",
hint: run "git rebase --abort".
Could not apply 345157c...  F2
i-rebase (git)-[feature|rebase-i]- > git status
interactive rebase in progress; onto 6e304bb
Last commands done (8 commands done):
   edit 4ca7108 SP
   pick 345157c F2
  (see more in file .git/rebase-merge/done)
No commands remaining.
You are currently rebasing branch 'feature' on '6e304bb'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)
```

```
Unmerged paths:
  (use "git restore --staged <file>..." to unstage)
  (use "git add <file>..." to mark resolution)
        both modified:   f.txt

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

A conflict happened. Luckily, we already know how to resolve conflicts; let's inspect that file and decide what we want to keep:

term 112: In Vim

```
F1
SQ
FX
<<<<<<< HEAD
=======
D
F2
>>>>>>> 345157c (F2)
```

We notice that the deletion of commit `D` caused the conflict[6] since Git did not know how to merge the two versions of `f.txt`. We decide to edit the file to look like this:

term 113: In Vim

```
F1
SQ
FX
F2
```

We mark the conflict as resolved (by executing `git add f.txt`) and then continue the rebase using `git rebase --continue`.

Next, the editor pops up again in case we want to modify the commit message. We will leave it as is ("F2"). After that, we finish our rebase and see the commands we mentioned earlier:

term 114

```
(git)-[feature|rebase-i]- > git add f.txt
(git)-[feature|rebase-i]- > git rebase --continue
[detached HEAD 56c9a3b] F2
 1 file changed, 1 insertion(+)
Successfully rebased and updated refs/heads/feature.
```

Let's look at the log to see what our history looks like now:

term 115

```
feature > git log --all --graph --oneline
* 56c9a3b (HEAD -> feature) F2
* 4bb266e SP2
* 65384dc SP1
* ed2c08d R2
* bd440a9 R1
* 59c082a fixup FX and squash SQ into F1
| * 01406a5 (experiment) E3
| * cee51e4 E2
| * d83399e E1
|/
* 6e304bb (main) C2
* c109b12 C1
```

As we can see, we arrived at the intermediate goal we set out in Figure 7. To verify whether, e.g., commit `59c082a` "fixup FX and squash SQ into F1" really contains all the changes commit `F1`, `SQ`, and `FX` introduced, we can use `git show`:

[6] If you think this conflict was unintentional and I did not want to redo the whole write-up of this session, then you are correct.

term 116

```
feature > git show --oneline 59c082a
59c082a fixup FX and squash SQ into F1
diff --git a/f.txt b/f.txt
new file mode 100644
index 0000000..5f1b366
--- /dev/null
+++ b/f.txt
@@ -0,0 +1,3 @@
+F1
+SQ
+FX
```

As expected, we can see that the previously included changes in `f.txt` from commit `SQ` (+SQ) and commit `FX` (+FX) are now part of commit `F1'` (`59c082a`). That can be done analogously for other commits if we are ever in doubt. But of course, as long as we understand what we are doing, there will not be any big surprises.

In the next section, we will perform the last step, cherry-picking the desired commits.

12.3 Cherry-Picking

Cherry-pick's functionality can be seen as a (proper[7]) subset of (interactive) rebasing. Everything that can be done using cherry-picking can also be achieved using rebasing. However, some scenarios exist where one command might be preferable or more suited to achieve the desired outcome. That is especially true for single commits.

Below, we list some points that we judge as being important to know about cherry-picking:

- First, we can also select ranges and multiple commits to cherry-pick. Git will apply them one at a time from left to right.
- Second, unlike rebase, cherry-pick "draws commits in". That means cherry-pick takes the changes introduced by a commit from somewhere else and reapplies them onto `HEAD`. So, the "drawing in" idea is similar to merges, where we merge histories into our currently checked out branch.
  In contrast, remember, during a rebase[8], we take some commits in our current line of work (or in the history from `<branch>` after we switch to it, in case we specify that parameter) and then reapply these commits onto `<upstream>` (or `<newbase>` if we use the `--onto` option).
  Therefore, our `HEAD` should point to the branch we want to reapply the cherry-picked commits onto.
- Third, cherry-picking does not move references from the commits it cherry-picks. Even if we cherry-pick a whole branch (which, for example, can be done by using the double-dot range specifier), the branch reference of the cherry-picked branch will remain unchanged; only the branch that `HEAD` points to (and `HEAD` itself) will be moved forward.

12.3.1 Cherry-Pick Applies Introduced Changes

To showcase that cherry-pick only applies the *changes* introduced by the specified commit, we will switch to `main` and cherry-pick commit `E2`. Since commit `E2` is the parent of commit `E3` and the branch `experiment` points to `E3`, we have that `experiment~` is commit `E2`.

term 117

```
feature > git switch main
Switched to branch 'main'
main > git cherry-pick experiment~
CONFLICT (modify/delete): e.txt deleted in HEAD and modified in cee51e4
(E2). Version cee51e4 (E2) of e.txt left in tree.
error: could not apply cee51e4... E2
hint: After resolving the conflicts, mark them with
hint: "git add/rm <pathspec>", then run
hint: "git cherry-pick --continue".
hint: You can instead skip this commit with "git cherry-pick --skip".
```

[7] For example, you cannot move branches with `git cherry-pick`. However, this is not only possible with rebasing, but also used more often than not, as we have seen in the previous sessions.

[8] Here, `<branch>`, `<upstream>`, and `<newbase>` refer to the parameters used in the rebase command, see `man git-rebase`.

```
hint:  To abort and get back to the state before "git cherry-pick",
hint:  run "git cherry-pick --abort".
```

Let's try to understand why the conflict message reads:

> e.txt *deleted* in HEAD and modified in cee51e4 (E2).

Git thinks `e.txt` was deleted in `HEAD` because cherry-pick wants to reapply the *changes* introduced by commit `E2`. To understand this more, we look at the changes that were introduced by `E2` next:

term 118

```
(git)-[main|cherry]- > git diff experiment~2 experiment~
diff --git a/e.txt b/e.txt
index b19a14e..88e3521 100644
--- a/e.txt
+++ b/e.txt
@@ -1 +1,2 @@
 E1
+E2
```

As we can see, `E2` appended something to the file `e.txt`. Since this file did not exist in `HEAD`'s history and Git wanted to append changes introduced by `E2` to `e.txt`, Git thought we must have deleted the file somewhere earlier on in the history. That is the reason we ended up with the conflict.

Let's see what `git status` has to say:

term 119

```
(git)-[main|cherry]- > git status
On branch main
You are currently cherry-picking commit cee51e4.
  (fix conflicts and run "git cherry-pick --continue")
  (use "git cherry-pick --skip" to skip this patch)
  (use "git cherry-pick --abort" to cancel the cherry-pick operation)

Unmerged paths:
  (use "git add/rm <file>..." as appropriate to mark resolution)
        deleted by us:   e.txt

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

Here, "us" is our current line of work, i.e., `HEAD`. Therefore, if we want to introduce `e.txt` to our current line of work, we must add it to the index and then continue our cherry-pick with `git cherry-pick --continue`:

term 120

```
(git)-[main|cherry]- > git add e.txt
(git)-[main|cherry]- > git status
On branch main
You are currently cherry-picking commit cee51e4.
  (all conflicts fixed: run "git cherry-pick --continue")
  (use "git cherry-pick --skip" to skip this patch)
  (use "git cherry-pick --abort" to cancel the cherry-pick operation)

Changes to be committed:
        new file:   e.txt

(git)-[main|cherry]- > git cherry-pick --continue
[main 0c39022] E2
 Date: Sun Mar 17 16:22:32 2024 +0800
 1 file changed, 2 insertions(+)
 create mode 100644 e.txt
```

Where the editor opened up after we executed `git cherry-pick --continue` to edit the commit message of the commit; we left it as is ("E2"). After this cherry-pick, our history looks as follows:

term 121

```
main > git log --all --graph --oneline
* 0c39022 (HEAD -> main) E2
| * 56c9a3b (feature) F2
| * 4bb266e SP2
| * 65384dc SP1
| * ed2c08d R2
| * bd440a9 R1
| * 59c082a fixup FX and squash SQ into F1
|/
| * 01406a5 (experiment) E3
| * cee51e4 E2
| * d83399e E1
|/
* 6e304bb C2
* c109b12 C1
```

So far so good.

12.3.2 Cherry-Picking Ranges of Commits

Next, we want to cherry-pick commits `R1`, `R2`, and `SP1` onto `main`. As is often the case, we have different options for specifying the commits. We could list them, one by one, using their commit hashes. Another option is the double-dot syntax, which we will use for this example. Since `feature` points to `F2`, `feature~4` points to commit `R1`, and `feature~2` points to commit `SP1`. To get all the commits that are in the history of `SP1` but exclude everything that comes *after* `R1`, we must use the range `feature~5..feture~2`.

term 122

```
main > git cherry-pick feature~5..feature~2
[main 8d47e66] R1
 Date: Sun Mar 17 16:37:32 2024 +0800
 1 file changed, 1 insertion(+)
 create mode 100644 r1.txt
[main 54171fc] R2
 Date: Sun Mar 17 16:37:27 2024 +0800
 1 file changed, 1 insertion(+)
 create mode 100644 r2.txt
[main f452c42] SP1
 Date: Wed Mar 20 23:53:27 2024 +0800
 1 file changed, 1 insertion(+)
 create mode 100644 sp1.txt
```

That demonstrated how we can cherry-pick a range of commits.

Our current history looks as below. We notice that the only thing left to do to arrive at our final goal, as stated in Figure 8, is to delete the branches.

term 123

```
main > git log --all --graph --oneline
* f452c42 (HEAD -> main) SP1
* 54171fc R2
* 8d47e66 R1
* 0c39022 E2
| * 56c9a3b (feature) F2
| * 4bb266e SP2
| * 65384dc SP1
| * ed2c08d R2
| * bd440a9 R1
| * 59c082a fixup FX and squash SQ into F1
|/
| * 01406a5 (experiment) E3
| * cee51e4 E2
| * d83399e E1
```

```
|/
* 6e304bb C2
* c109b12 C1
```

12.4 Wrapping Up

From term 123, we know that the only thing left to do to arrive at Figure 8 is to clean up our branches, `feature`, and `experiment`.

term 124

```
main > git branch -D feature experiment
Deleted branch feature (was 56c9a3b).
Deleted branch experiment (was 01406a5).
main > git log --all --graph --oneline
* f452c42 (HEAD -> main) SP1
* 54171fc R2
* 8d47e66 R1
* 0c39022 E2
* 6e304bb C2
* c109b12 C1
```

That concludes our discussion about rebasing.

Session 08: Filter-Repo

13 Rewriting Repository History

This session is inspired by a recent problem my friend faced. We will discuss the issue and examine various approaches to resolve it. This example provides a nice example of the limitations of rebasing and where tools like filter-repo are needed instead.

13.1 My Friend's Example

The repository in question has a few commits and no other branches apart from the `main` branch. Let's set up this example:

term 125

```
> git init
main > simple_commit() {
  echo "$1" >> "$2"
  git add "$2"
  git commit -m "$1"
}
main > for c in {1..4}; do
  simple_commit "M$c" m.txt
done
[main (root-commit) bbc2556] M1
 1 file changed, 5 insertions(+)
 create mode 100644 m.txt
[main 8f7662e] M2
 1 file changed, 1 insertion(+)
[main 6ef5e6a] M3
 1 file changed, 1 insertion(+)
[main da511a6] M4
 1 file changed, 1 insertion(+)
```

As we have done in a previous session, we use short commit messages to make it easier to refer to particular commits, i.e., we will, for example, write "commit `M1`" when we mean the commit that has the commit message `M1`. In the case of `M1`, it would be commit `bbc2556`.

Our setup results in the following linear history:

term 126

```
main > git log --all --oneline --graph
* da511a6 (HEAD -> main) M4
* 6ef5e6a M3
* 8f7662e M2
* bbc2556 M1
```

Suddenly, we noticed that we accidentaly used the wrong name and email address for our commits. Instead[1] of *Alice* and *alice@example.com*, we should have used *Bob* and *bob@example.com*. We use `git log` together with the `--pretty` option to check who the author and committer of our commits are:

term 127

```
main > git log --all --pretty=format:"%h%x09%s%x09%an%x09%ae%x09%cn%x09%ce"
da511a6 M4      Alice   alice@example.com       Alice   alice@example.com
6ef5e6a M3      Alice   alice@example.com       Alice   alice@example.com
8f7662e M2      Alice   alice@example.com       Alice   alice@example.com
bbc2556 M1      Alice   alice@example.com       Alice   alice@example.com
```

Where we chose the format in such a way that the output has the following columns:

abbreviated commit hash	subject	author name	author email	committer name	committer email

[1] You will find your default name and email here.

These placeholders (e.g., `%h`) can be found in `man git-log`.

The problem we are now trying to solve is how we fix the author and committer of every commit without starting anew.

13.1.1 First Try: Simple Interactive Rebase

We can solve this problem with the tools we learned in the previous sessions. We need to do the following:

(i) Figure out a command to change the name and email for a single commit.

(ii) Stop at each commit and run the command from the previous step.

That can be achieved by:

(i) Amending the commit using the command:

```
git commit --amend --reset-author --no-edit
```

where `--no-edit` is used to avoid launching the editor to edit the commit message every time.

(ii) Prepare the commands and commits during the interactive rebase.

Note, we use the `--reset-author` option instead of the `--author=<author>` option because the latter changes the commit's author but still uses the currently configured user details for the committer. In our case, we want to change both the author and the committer. If we use `--author=<author>` but do not change the underlying user, we will have the correct author but still the wrong committer. In general[2], the `--amend` option preserves the author of the commit; by using `--amend` with `--author=<author>`, the author is changed, but the configured committer is used, and by using `--amend` with `--reset-author`, the author is declared to be the committer.
Therefore, the first thing we must change before proceeding is the user name and email. Using the command `git config user.name` and `git config user.email`, we can find out the current user details. We can change them locally[3] as follows:

term 128

```
main > git config user.name
Alice
main > git config user.email
alice@example.com
main > git config user.name "Bob"
main > git config user.email "bob@example.com"
```

Where we only executed the first two commands to show the reader the current user before the change. We will continue displaying the current user before changing it each time because we believe that makes it easier to follow along.

The question that remains is what will we choose as `<upstream>` for our rebase command? Based on our current knowledge, we might gravitate towards using `HEAD~4` because `HEAD~4..HEAD` would then select all current commits. The problem is `HEAD~4` is not a valid reference because `HEAD~3` is the root commit, and by definition, the root commit does not have any parents.

If we try to do that anyway, Git will complain about our mistake:

term 129

```
main > git rebase -i HEAD~4
fatal:  invalid upstream 'HEAD~4'
```

We quickly come up with a workaround. We do not have to reapply the first commit; amending it will suffice. Therefore, we can use `HEAD~3` and then execute the amend command for the first commit before we reapply the second commit.

Let's execute this "brute-force" approach:

term 130

```
main > git rebase -i HEAD~3
```

As usual, our editor will pop up. We change the lines from[4]

[2] That assumes `user.name` and `user.email` is used to configure the user. The behavior changes if the more fine-grained `author.name`, `author.email`, `committer.name`, and `committer.email` configurations are set. See `man git-config` for more details.

[3] Tip: If you have faced this issue before because you have multiple accounts, there are ways to have different default users based on your current working directory. See this StackExchange answer on how to do this.

[4] In the subsequent displays, we will omit the commented-out help section provided to us below the commit lines.

term 131: In Vim

```
pick 8f7662e M2
pick 6ef5e6a M3
pick da511a6 M4
```

to

term 132: In Vim

```
exec git commit --amend --reset-author --no-edit
pick 8f7662e M2
exec git commit --amend --reset-author --no-edit
pick 6ef5e6a M3
exec git commit --amend --reset-author --no-edit
pick da511a6 M4
exec git commit --amend --reset-author --no-edit
```

After saving and closing the editor, we see the following in our terminal:

term 133

```
Executing: git commit --amend --reset-author --no-edit
[detached HEAD 39bbb7a] M1
 1 file changed, 5 insertions(+)
 create mode 100644 m.txt
Executing: git commit --amend --reset-author --no-edit
[detached HEAD cca7370] M2
 1 file changed, 1 insertion(+)
Executing: git commit --amend --reset-author --no-edit
[detached HEAD a5b4e59] M3
 1 file changed, 1 insertion(+)
Executing: git commit --amend --reset-author --no-edit
[detached HEAD a6316c8] M4
 1 file changed, 1 insertion(+)
Successfully rebased and updated refs/heads/main.
```

`git log` shows us that we have achieved our desired goal.

term 134

```
main > git log --all --pretty=format:"%h%x09%s%x09%an%x09%ae%x09%cn%x09%ce"
a6316c8 M4      Bob     bob@example.com Bob     bob@example.com
a5b4e59 M3      Bob     bob@example.com Bob     bob@example.com
cca7370 M2      Bob     bob@example.com Bob     bob@example.com
39bbb7a M1      Bob     bob@example.com Bob     bob@example.com
```

This approach works, and it only uses the tools we have learned so far. However, we can make our lives a bit easier with some additional options of the rebase command. We will discuss this in the next section.

13.1.2 Second Try: Leveraging Rebase's Options

The problem we want to solve is still the same as in section 13.1.1. However, since the previous approach worked, we will rename the author's name from *Bob* to *Carol* and the author's email from *bob@example.com* to *carol@example.com*.

This time, we will leverage the options `--root` and `--exec` of `git rebase`. We use these options for the following reasons:

(i) `--root` allows us to rebase all commits reachable from our `<branch>`. Since we will not specify `<branch>` in the upcoming rebase command, `--root` allows us to rebase all commits reachable from our current branch. That solves the problem we had in our first attempt, where we could not include (and reapply) the root commit in the interactive rebase. By using `--root`, we also no longer need to specify `<upstream>`.

(ii) `--exec` allows us to append the specified command after each reapplication of a commit.

Using these two options together, we do not need to run an interactive rebase anymore because we do not have to add or edit any commands. The final command is:

```
git rebase --root --exec "git commit --amend --reset-author --no-edit"
```

Before we run the command, we first have to modify the user details again:

term 135

```
main > git config user.name
Bob
main > git config user.email
bob@example.com
main > git config user.name "Carol"
main > git config user.email "carol@example.com"
```

We can still use the `-i` option to force the editor to pop up and allow us to inspect what will be executed. Let's do that as a sanity check:

term 136

```
main > git rebase -i --root \
--exec "git commit --amend --reset-author --no-edit"
```

In the editor, we see:

term 137: In Vim

```
pick 39bbb7a M1
exec git commit --amend --reset-author --no-edit
pick cca7370 M2
exec git commit --amend --reset-author --no-edit
pick a5b4e59 M3
exec git commit --amend --reset-author --no-edit
pick a6316c8 M4
exec git commit --amend --reset-author --no-edit
```

Notice that this time, the root commit is listed on the first line. Also, the execute command we specified is automatically appended after each commit line as desired. We close the editor and observe the execution of our rebase:

term 138

```
Executing: git commit --amend --reset-author --no-edit
[detached HEAD 8024f39] M1
 1 file changed, 5 insertions(+)
 create mode 100644 m.txt
Executing: git commit --amend --reset-author --no-edit
[detached HEAD 3999b6e] M2
 1 file changed, 1 insertion(+)
Executing: git commit --amend --reset-author --no-edit
[detached HEAD 53d629e] M3
 1 file changed, 1 insertion(+)
Executing: git commit --amend --reset-author --no-edit
[detached HEAD 40fb0b1] M4
 1 file changed, 1 insertion(+)
Successfully rebased and updated refs/heads/main.
```

Checking the log reveals that this more efficient approach also solved our problem:

term 139

```
main > git log --all --pretty=format:"%h%x09%s%x09%an%x09%ae%x09%cn%x09%ce"
40fb0b1 M4      Carol   carol@example.com       Carol   carol@example.com
53d629e M3      Carol   carol@example.com       Carol   carol@example.com
3999b6e M2      Carol   carol@example.com       Carol   carol@example.com
8024f39 M1      Carol   carol@example.com       Carol   carol@example.com
```

13.2 Harder Example

This example is concerned with the same problem as section 13.1. However, this time, we are less lucky, and we have more than a single branch.

For the setup of this example, we reuse the previous example but create a branch `feature`, which branches out from commit `M2`. We create two simple commits on `feature`.

term 140

```
main > git switch -c feature HEAD~2
Switched to a new branch 'feature'
feature > simple_commit "F1" f.txt # See term 125 for def.  of
simple_commit
[feature 853e949] F1
 1 file changed, 1 insertion(+)
 create mode 100644 f.txt
feature > simple_commit "F2" f.txt
[feature 27f92e9] F2
 1 file changed, 1 insertion(+)
```

At the end of the setup, the repository should look as follows, with each commit having Carol as the author and committer together with her corresponding email.

term 141

```
feature > git log --all --oneline --graph
* 27f92e9 (HEAD -> feature) F2
* 853e949 F1
| * 40fb0b1 (main) M4
| * 53d629e M3
|/
* 3999b6e M2
* 8024f39 M1
feature > git log --all \
--pretty=format:"%h%x09%s%x09%an%x09%ae%x09%cn%x09%ce"
27f92e9 F2      Carol   carol@example.com       Carol   carol@example.com
853e949 F1      Carol   carol@example.com       Carol   carol@example.com
40fb0b1 M4      Carol   carol@example.com       Carol   carol@example.com
3999b6e M2      Carol   carol@example.com       Carol   carol@example.com
53d629e M3      Carol   carol@example.com       Carol   carol@example.com
8024f39 M1      Carol   carol@example.com       Carol   carol@example.com
```

13.2.1 First Try: Using What We Learned in the Previous Example

Initially, it looks like we face the same problem as in section 13.1. Naturally, we also try to fix it in the same way. First, let's change the user credentials once more.

term 142

```
feature > git config user.name
Carol
feature > git config user.email
carol@example.com
feature > git config user.name "Erin"
feature > git config user.email "erin@example.com"
```

Then, we will use the same command we used in term 136 and add `main` at the end because we are currently on `feature`.

term 143

```
feature > git rebase --root \
--exec "git commit --amend --reset-author --no-edit" main
Executing: git commit --amend --reset-author --no-edit
[detached HEAD 3b019fa] M1
 1 file changed, 5 insertions(+)
 create mode 100644 m.txt
Executing: git commit --amend --reset-author --no-edit
[detached HEAD e142ccf] M2
 1 file changed, 1 insertion(+)
Executing: git commit --amend --reset-author --no-edit
```

```
[detached HEAD 4b077df] M3
 1 file changed, 1 insertion(+)
Executing: git commit --amend --reset-author --no-edit
[detached HEAD e2f3537] M4
 1 file changed, 1 insertion(+)
Successfully rebased and updated refs/heads/main.
```

At first, this does not look all that bad until we see what our repository looks like now.

term 144

```
main > git log --all --oneline --graph
* e2f3537 (HEAD -> main) M4
* 4b077df M3
* e142ccf M2
* 3b019fa M1
* 27f92e9 (feature) F2
* 853e949 F1
* 3999b6e M2
* 8024f39 M1
main > git log --all --pretty=format:"%h%x09%s%x09%an%x09%ae%x09%cn%x09%ce"
e2f3537 M4      Erin    erin@example.com        Erin    erin@example.com
4b077df M3      Erin    erin@example.com        Erin    erin@example.com
e142ccf M2      Erin    erin@example.com        Erin    erin@example.com
3b019fa M1      Erin    erin@example.com        Erin    erin@example.com
27f92e9 F2      Carol   carol@example.com       Carol   carol@example.com
853e949 F1      Carol   carol@example.com       Carol   carol@example.com
3999b6e M2      Carol   carol@example.com       Carol   carol@example.com
8024f39 M1      Carol   carol@example.com       Carol   carol@example.com
```

It looks as if we reapplied commits `M1` to `M4` on top of `feature`. Also, it appears that we have duplicates for commit `M1` (`3b019fa` and `8024f39`) and `M2` (`e142ccf` and `3999b6e`).

One possibility for how we might spot that something is off here is by remembering what we learned in a previous session about how duplicate commits behave during rebasing. We have learned that if a particular commit is reapplied onto a history during a rebase in which it already exists, it will be skipped. Therefore, either the commits are not duplicates, or we have separate histories!

In a previous session, we also learned that we can detect duplicate commits by checking the equivalence of their patches using their patch ID. Let's quickly verify whether the patches introduced by the two duplicated commits `M1` and `M2` are equal.

term 145

```
main > git show main~3 | git patch-id | awk '{print $1}'
fd76eb1ec8be779f344476c7bf2fa06b68190c51
main > git show feature~3 | git patch-id | awk '{print $1}'
fd76eb1ec8be779f344476c7bf2fa06b68190c51
main > git show main~2 | git patch-id | awk '{print $1}'
03556ad0b4eb3bb7619b669a2f3be6204200a611
main > git show feature~2 | git patch-id | awk '{print $1}'
03556ad0b4eb3bb7619b669a2f3be6204200a611
```

As we can see, the patch IDs of `main~3` and `feature~3` (i.e., the commits `M1`) are equivalent. The same holds for the patch IDs of commits `main~2` and `feature~2` (i.e., the commits `M2`). That means that they are duplicated commits. Therefore, they must be in different histories, i.e., we accidentally disconnected `feature` from our main history. Next, we will discuss ways to verify this assumption.

One way to determine if the history of `main` and `feature` is the same, as the `git log` output makes us believe, is by looking at the commit object[5] of `main~3`. We want to find out whether it has a parent or not. If `F2` is `main~3`'s parent, the history observed[6] using `git log` is a single history. Otherwise, we found a second root commit.

[5] We learned about the `git cat-file` command in a previous session.

[6] Note that the output of `git log` is only confusing because we used the `--all` option. Without it, the `feature` branch would not be displayed, and the history graph would not look like a single history. However, since we often use this command together with this option to get an overview of our repository, it is important to be aware of this behavior.

term 146

```
main > git cat-file -p main~3
tree c514a2cdf5ecb37ac4e2900fd67377fc6e6f6c14
author Erin <erin@example.com> 1714817651 +0800
committer Erin <erin@example.com> 1714817651 +0800

M1
```

As we can see, `main~3` does not have a parent.

Another way to see this is by searching for all commits without parents. To find all root commits, we can leverage the `--max-parent` option of the `git log` command.

term 147

```
main > git log --all --max-parents=0
commit 3b019faf71f66a69d8b7ee0b9c8a3271efe38f70
Author: Erin <erin@example.com>
Date:   Sat May 4 18:14:11 2024 +0800

    M1

commit 8024f39b5ec55b4eeafa1e6f37041150ff29ba7a
Author: Carol <carol@example.com>
Date:   Wed May 1 17:17:54 2024 +0800

    M1
```

We notice two things:

(i) There are two root commits, one from branch `main` and the other from branch `feature`.

(ii) The root commit of `feature` shows that `feature`'s commits' author name and email were not changed.

Both of the above-mentioned findings are not that surprising because we already suspected we had two root commits and that the commits on the `feature` branch were not modified. Therefore, they still have the previous user's name and email.

That leaves us with the goal of also applying the changes to the commits reachable from `feature` and, of course, not having two histories but the same relation structure as before the rebase instead.

We can reuse the tools we learned already to reattach the `feature` branch at the right position (`main~2`) and also make the desired changes to the author's and committer's information for commits `F1` and `F2` (`feature~2..feature`). Using the rebase recipe we have been using many times in the past few sessions, we can derive the following command:

```
git rebase \
--exec "git commit --amend --reset-author --no-edit" \
--onto main~2 feature~2 feature
```

term 148

```
main > git rebase --exec "git commit --amend --reset-author --no-edit" \
--onto main~2 feature~2 feature
Executing: git commit --amend --reset-author --no-edit
[detached HEAD 65a666c] F1
 1 file changed, 1 insertion(+)
 create mode 100644 f.txt
Executing: git commit --amend --reset-author --no-edit
[detached HEAD 0dad508] F2
 1 file changed, 1 insertion(+)
Successfully rebased and updated refs/heads/feature.
```

That finally results in the desired outcome:

term 149

```
feature > git log --all --oneline --graph
* 0dad508 (HEAD -> feature) F2
* 65a666c F1
| * e2f3537 (main) M4
| * 4b077df M3
|/
* e142ccf M2
* 3b019fa M1
feature > git log --all \
--pretty=format:"%h%x09%s%x09%an%x09%ae%x09%cn%x09%ce"
0dad508 F2      Erin    erin@example.com        Erin    erin@example.com
65a666c F1      Erin    erin@example.com        Erin    erin@example.com
e2f3537 M4      Erin    erin@example.com        Erin    erin@example.com
e142ccf M2      Erin    erin@example.com        Erin    erin@example.com
4b077df M3      Erin    erin@example.com        Erin    erin@example.com
3b019fa M1      Erin    erin@example.com        Erin    erin@example.com
```

As we can see, performing such changes with `git rebase` quickly gets unmanageable because it does not scale well to multiple branches. That is exactly where filter-repo comes to our rescue.

13.2.2 Solution: Filter-Repo

git filter-repo is a tool for rewriting repository history[7]. In older literature, `git filter-branch` is often mentioned in this context. However, quoting the manual page of `git filter-branch` reveals that one should avoid using it and favor `git filter-repo` instead:

> Please use an alternative history filtering tool such as **git filter-repo**

It is important to be aware that `git filter-repo` irreversibly rewrites history. There is also a built-in safety check, which we will encounter when executing the command. Since we will not cover filter-repo in depth, it is recommended that one reads the discussion section of the manual and follows the recommended approach.

We will use callbacks[8] to solve our problem of renaming the author's name and email. The command line argument is converted into a Python function, where the parameter forms the function's body.

Now, let's see how we can solve our problem using filter-repo. We can use the option `--name-callback` for changing the name and `--email-callback` for changing the email. Using the examples given in the documentation, we come up with the following command:

```
git-filter-repo --name-callback 'return name.replace(b"Erin", b"Frank")' \
--email-callback \
'return email.replace(b"erin@example.com", b"frank@example.com")'
```

Let's execute that:

term 150

```
feature > git-filter-repo \
--name-callback 'return name.replace(b"Erin", b"Frank")' \
--email-callback \
'return email.replace(b"erin@example.com", b"frank@example.com")'
Aborting: Refusing to destructively overwrite repo history since
this does not look like a fresh clone.
  (expected at most one entry in the reflog for HEAD)
Please operate on a fresh clone instead.  If you want to proceed
anyway, use --force.
```

Here, we see the safety mechanism mentioned previously stopping us from executing the command because we do not follow the recommendation of running filter-repo on a fresh clone. The basic idea is that filter-repo recommends having a backup of the repository in which we are about to execute the command because it irreversibly rewrites history. We will only use the `--force` option here because we are experimenting on a dummy repository. For an actual repository, we should and would not use the `--force` flag and instead follow the recommendation of only operating on a fresh clone.

[7] See the installation guide on how to install it.

[8] Note that we are using version 2.38.0 of `git filter-repo`. We mention this because of the "API BACKWARD COMPATIBILITY CAVEAT" described at the top of the filter-repo Python script.

term 151

```
feature > git-filter-repo --force \
--name-callback 'return name.replace(b"Erin", b"Frank")' \
--email-callback \
'return email.replace(b"erin@example.com", b"frank@example.com")'
Parsed 6 commits
New history written in 0.17 seconds; now repacking/cleaning...
Repacking your repo and cleaning out old unneeded objects
HEAD is now at 4318aca F2
Enumerating objects: 18, done.
Counting objects: 100% (18/18), done.
Delta compression using up to 8 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 0), reused 0 (delta 0), pack-reused 0
Completely finished after 0.56 seconds.
```

With that, we changed the author and committer of our commits without messing up the repository in any way.

term 152

```
feature > git log --all --oneline --graph
* 4318aca (HEAD -> feature) F2
* 02d343b F1
| * d5b77bd (main) M4
| * e341bb9 M3
|/
* 575f8a6 M2
* eac9c86 M1
feature > git log --all \
--pretty=format:"%h%x09%s%x09%an%x09%ae%x09%cn%x09%ce"
4318aca F2      Frank   frank@example.com       Frank   frank@example.com
02d343b F1      Frank   frank@example.com       Frank   frank@example.com
d5b77bd M4      Frank   frank@example.com       Frank   frank@example.com
eac9c86 M1      Frank   frank@example.com       Frank   frank@example.com
e341bb9 M3      Frank   frank@example.com       Frank   frank@example.com
575f8a6 M2      Frank   frank@example.com       Frank   frank@example.com
```

We end this section by noting that if you desire to do something similar to what we have done with multiple names and emails, it is worth looking at the User and email based filtering section in the documentation. In that case, creating a mailmap file and then using the `--mailmap` option will be more efficient. Plus, if you version-control the mailmap file, you will know in the future which names and emails were updated.

13.3 Filter-Repo: Another Example

To better understand filter-repo, we will provide an additional example. We continue with our repo from term 152. Again, we will use the `--force` flag to circumvent the safety mechanism only because we are experimenting with a dummy repository.

Something else we can do with filter-repo is rewriting commit messages in bulk. Let's say we want to rewrite all our commit messages from `Mx` to `main x` and from `Fx` to `feature x`, where `x` is the corresponding integer in the commit message. We can achieve this using the `--replace-message` option together with a file that specifies the desired rules. For our example, the file `expressions.txt` looks as follows:

term 153: expression.txt in Vim

```
regex:M([0-9])==>main \1
regex:F([0-9])==>feature \1
```

We will use this file with the `--replace-message` option:

term 154

```
feature > git filter-repo --force --replace-message expressions.txt
Parsed 6 commits
New history written in 0.14 seconds; now repacking/cleaning...
Repacking your repo and cleaning out old unneeded objects
HEAD is now at 8c8afd2 feature 2
Enumerating objects: 18, done.
Counting objects: 100% (18/18), done.
Delta compression using up to 8 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 0), reused 12 (delta 0), pack-reused 0
Completely finished after 0.47 seconds.
feature > git log --all --oneline --graph
* 8c8afd2 (HEAD -> feature) feature 2
* a05beb9 feature 1
| * c9b1d16 (main) main 4
| * 075a951 main 3
|/
* 77844b1 main 2
* e99b2c6 main 1
```

And there are many more examples where this neat tool comes in handy.

Session 09: Data Recovery

14 Data Recovery

In this session we will discuss how we can recover things when mistakes happen. We will illustrate this using two examples, reverting a wrong rebase and recovering a deleted branch. We will discuss different methods and techniques that can help us tackle these scenarios. Mainly, we will learn about `git reflog`, but we will also discuss `git fsck`.

Let's first set up our example repository:

term 155

```
> git init
main > simple_commit() {
  echo "$1" >> "$2"
  git add "$2"
  git commit -m "$1"
}
main > for c in {1..4}; do
  simple_commit "M$c" m.txt
done
[main (root-commit) 9709844] M1
 1 file changed, 5 insertions(+)
 create mode 100644 m.txt
[main 2a6b2cb] M2
 1 file changed, 1 insertion(+)
[main 3f883d7] M3
 1 file changed, 1 insertion(+)
[main abcf01a] M4
 1 file changed, 1 insertion(+)
main > git switch -c feature main~2
Switched to a new branch 'feature'
feature > simple_commit "F1" f1.txt
[feature 3d58cf8] F1
 1 file changed, 1 insertion(+)
 create mode 100644 f1.txt
feature > simple_commit "F2" f2.txt
[feature 2020cad] F2
 1 file changed, 1 insertion(+)
 create mode 100644 f2.txt
feature > git log --all --oneline --graph
* 2020cad (HEAD -> feature) F2
* 3d58cf8 F1
| * abcf01a (main) M4
| * 3f883d7 M3
|/
* 2a6b2cb M2
* 9709844 M1
```

14.1 Example 1: Revert Rebase

Given this setup, assume we want to rebase the commits on `feature` onto `main`. We will deliberately use `feature~` for the `<upstream>` parameter instead of `feature~2`. That will result in commit `F1` not being saved to the temporary file. Therefore, commit `F1` will be lost, and only commit `F2` will be reapplied onto `main`. Let's intentionally make this mistake and then explore how to restore our repository to its state before the rebase.

term 156

```
feature > git rebase --onto main feature~
Successfully rebased and updated refs/heads/feature.
feature > git log --all --oneline --graph
* 6ca6171 (HEAD -> feature) F2
* abcf01a (main) M4
```

```
* 3f883d7 M3
* 2a6b2cb M2
* 9709844 M1
```

As we can see, commit `F1` is gone. Well, as we already know, it is not really gone in the sense that the commit was deleted, but there does not exist a reference to an ancestor of commit `F1`. Therefore, it is not visible in the log because only commits reachable from references are displayed.

If we still remember the commit hash, `2020cad`, in this case, we could simply execute `git reset --hard 2020cad` while on the `feature` branch. That would reset the `feature` branch to where it was poiting to before the rebase. However, of course, we cannot rely on having written the desired SHA-1 hash down before making a mistake. That is where `git reflog` comes to our rescue.

14.1.1 Reflog to the Rescue

Reflogs are logs of the references; every time a reference is updated, it records its new state in a log, the reflog. It is important to note that the reflog is *local* to your copy of the repository. That means if you push a repository to a remote, the activity recorded in the reflog is not included in the push. Analogously, if you clone a repository, the reflog is initially empty. By default, these logs are stored in `.git/logs`. Let's have a look at what the file structure of the logs looks like in our repository:

term 157

```
feature > tree .git/logs
.git/logs
|-- HEAD
`-- refs
    `-- heads
        |-- feature
        `-- main

2 directories, 3 files
```

As we can see, currently, we have tree reflog files. That makes sense because we have one file for each branch and one for `HEAD`. Let's `cat` one of these files to see what they look like.

term 158

```
feature > cat .git/logs/refs/heads/main # Output reformatted[1]
0000000000000000000000000000000000000000
970984419b1b9efbddbbd22b668a57eb266a9e9a
Alice
<alice@example.com>
1715505114
+0800
commit (initial): M1

970984419b1b9efbddbbd22b668a57eb266a9e9a
2a6b2cb4becb31e1eab57daa641087610fc47264
Alice
<alice@example.com>
1715505115
+0800
commit: M2

2a6b2cb4becb31e1eab57daa641087610fc47264
3f883d741502c66c74dd9cf67df22895d6cb2386
Alice
<alice@example.com>
1715505115
+0800
commit: M3

3f883d741502c66c74dd9cf67df22895d6cb2386
```

```
abcf01a0997f2a29b13fc2ca4c305a893023fb8e
Alice
<alice@example.com>
1715505115
+0800
commit: M4
```

Based on this example and experimenting around a bit (to, e.g., determine whether `author.name` or `committer.name` is used), we conclude that the columns of the reflog files are as follows:

SHA-1 before | SHA-1 after | committer name | committer email | Unix timestamp | UTC offset | description

However, there is generally no reason to examine the file directly; we will use the `git reflog` command instead. The vast majority of times[2], we only need to use the command `git reflog <ref>`, where `<ref>` is the reference we want to see the logs for. If not specified, the default `<ref>` is `HEAD`.

For this example, because we are searching for the previous tip of the `feature` branch, let's examine `feature`'s reflog:

term 159

```
feature > git reflog feature
6ca6171 (HEAD -> feature) feature@{0}:  rebase (finish):
refs/heads/feature onto abcf01a0997f2a29b13fc2ca4c305a893023fb8e
2020cad feature@{1}:  commit:  F2
3d58cf8 feature@{2}:  commit:  F1
2a6b2cb feature@{3}:  branch:  Created from main~2
```

The first thing that stands out is the special *@-syntax*. Very briefly[3], the syntax `<refname>@{<n>}` specifies the n-th prior value of that reference. Knowing this and observing the output of `git reflog feature`, we can deduce that `feature@{0}` is just `feature`. Also, we can see that the description next to it states that that is where the rebase finished.

Next, we look at `feature@{1}`'s description; it reads "commit: F2". That means, currently[4], `feature@{1}` refers to the commit `F2` before the execution of the rebase took place, i.e., the commit hash we were trying to locate. Therefore, we can do a hard reset of the `feature` branch to that commit to reach our goal of reverting `feature` back to its previous state before the rebase.

term 160

```
feature > git reset --hard feature@{1}
HEAD is now at 2020cad F2
feature > git log --all --oneline --graph
* 2020cad (HEAD -> feature) F2
* 3d58cf8 F1
| * abcf01a (main) M4
| * 3f883d7 M3
|/
* 2a6b2cb M2
* 9709844 M1
```

If we check the reflog now, we will see a new entry due to the reset we just performed.

term 161

```
feature > git reflog feature
2020cad (HEAD -> feature) feature@{0}:  reset:  moving to feature@{1}
6ca6171 feature@{1}:  rebase (finish):  refs/heads/feature onto
abcf01a0997f2a29b13fc2ca4c305a893023fb8e
2020cad (HEAD -> feature) feature@{2}:  commit:  F2
3d58cf8 feature@{3}:  commit:  F1
```

[1]Because of the limited width of the paper format, we will reformat the output for convenience. Originally, the columns were separated by spaces or tabs, and the rows by a new line. We separated the columns by a new line and the rows using two new lines.

[2]The reason is that the other subcommands and options of reflog are typically not used by the user directly but rather by `git gc`.

[3]More information on the syntax can be found in `man gitrevisions`. For example, one can also use a date or time to find the value of a ref at a prior point in time using the `<refname>@{<date>}` syntax.

[4]Note that we say *currently* because the syntax is relative to the last update of the reference. As we will see shortly, after the `feature` reference is updated, `feature@{1}` refers to another SHA-1 hash.

```
2a6b2cb feature@{4}: branch: Created from main~2
```

Having a look at the description of `feature@{0}` reveals:

(i) As previously noted, the `<refname>@{<n>}` syntax is relative to the last update of the reference. The description "moving to feature@{1}" was accurate before the update of the reference, but the current `feature@{1}` (commit `6ca6171`) is different from the `feature@{1}` at the time before the update (commit `2020cad`).

(ii) For the reason discussed in the previous point, we prefer using the SHA-1 hash instead of the `<refname>@{<n>}` syntax when executing commands. That makes the reflog easier to understand when we need it again. In this particular case, "reset: moving to 2020cad" is more understandable than "reset: moving to feature@{1}".

We also observe that references (`HEAD` and `feature` in this case) are displayed next to the commit hashes, which makes it easier for us to spot the same commit hashes the references are currently pointing at.

14.2 Example 2: Recovering a Deleted Branch

We reuse the previous setup for this example. We display the repository again for convenience.

term 162

```
feature > git log --all --oneline --graph
* 2020cad (HEAD -> feature) F2
* 3d58cf8 F1
| * abcf01a (main) M4
| * 3f883d7 M3
|/
* 2a6b2cb M2
* 9709844 M1
```

Let's assume we accidentally delete the `feature` branch. How can we recover it?

term 163

```
feature > git switch main
Switched to branch 'main'
main > git branch -D feature
Deleted branch feature (was 2020cad).
main > git log --all --oneline --graph
* abcf01a (HEAD -> main) M4
* 3f883d7 M3
* 2a6b2cb M2
* 9709844 M1
```

The output from the `git branch -D feature` command makes this super easy for us. We simply take the hash printed to the console, which is the hash `feature` was pointing to before deletion, and then we execute `git branch feature 2020cad` to recover it. However, what if we only notice this mistake later, our terminal crashes or anything else keeps us from remembering that SHA-1 hash?

14.2.1 Reflog?

Naturally, we use the tools we have learned already. The problem is that since we deleted the reference, the corresponding reflog entry also got deleted:

term 164

```
main > git reflog feature
fatal: ambiguous argument 'feature': unknown revision or path not in the
working tree.
Use '--' to separate paths from revisions, like this:
'git <command> [<revision>...] -- [<file>...]'
```

However, the reflog can still help us. Since `HEAD` is also a reference, it also has a reflog. Let's examine `HEAD`'s reflog, the default reference for the `git reflog` command.

term 165

```
main > git reflog
abcf01a (HEAD -> main) HEAD@{0}: checkout: moving from feature to main
2020cad HEAD@{1}: reset: moving to feature@{1}
6ca6171 HEAD@{2}: rebase (finish): returning to refs/heads/feature
6ca6171 HEAD@{3}: rebase (pick): F2
abcf01a (HEAD -> main) HEAD@{4}: rebase (start): checkout main
2020cad HEAD@{5}: commit: F2
3d58cf8 HEAD@{6}: commit: F1
2a6b2cb HEAD@{7}: checkout: moving from main to feature
abcf01a (HEAD -> main) HEAD@{8}: commit: M4
3f883d7 HEAD@{9}: commit: M3
2a6b2cb HEAD@{10}: commit: M2
9709844 HEAD@{11}: commit (initial): M1
```

We can then leverage the descriptions provided in the output to find out where `feature` pointed to before being deleted. In this case, we can find it in two ways:

- The description of `HEAD@{0}` reads "checkout: moving from feature to main". That means, before `HEAD` was updated, it pointed to `feature`. Therefore, `HEAD@{1}` must be our desired commit.
- The last commit we created on `feature` had the message "F2". Searching for that message, we find that `HEAD@{5}` also yields our desired commit. We can rule out `HEAD@{3}` because the description mentions this commit was picked during a rebase, which we know was a mistake we discussed in section 14.1.

The problem with `HEAD`'s reflog is that it can easily become non-trivial to track down the desired commit. That does not mean it should not be used[5].

However, in these cases, there is also another Git command that can come in handy, `git fsck`.

14.2.2 File System CheckK

The `git fsck` command verifies the connectivity and validity of the objects in Git's key-value store. We are interested in the connectivity aspect of the command. Of particular importance for our example is understanding the `<object>` option of `git fsck`. Quoting the manual (`man git-fsck`):

> [...] git fsck defaults to using the index file, all SHA-1 references in the **refs** namespace, and all reflogs (unless --no-reflogs is given) as heads.

That means that if our commit object is reachable from the reflog (which it is, as we saw in section 14.2.1), we have to use the `--no-reflog` option. Otherwise, it is considered reachable and will not be part of the output.

Additionaly, we are not looking for any *unreachable* object, we are looking for a *dangling*[6] commit because our commit has no parent. In the manual (`man git-fsck`) we find that fsck prints these dangling commits by default. Let's see what we find:

term 166

```
main > git fsck --no-reflog
Checking object directories: 100% (256/256), done.
dangling commit 2020caddc5b22a9e49dbd3eeb3d4af183d0b0f7e
dangling commit 6ca61719f685e39ae1f3eeb7105d7eedb201464c
```

As we can see, we find two dangling commits. One must be our desired commit[7]. Let's execute a `git log` using the dangling commit found to see if it is the one we have been searching for.

term 167

```
main > git log --oneline 2020caddc5b22a9e49dbd3eeb3d4af183d0b0f7e
2020cad F2
3d58cf8 F1
2a6b2cb M2
```

[5]Things like `<refname>@<date>` syntax (see `man gitrevisions`), `--since=<date>` and `--until=<date>` options of `git log` (`git reflog` is just and alias for a particular `git log` command, see `man git-reflog`) can come in handy if we have to filter through a long reflog and we remember the approximate time when we updated the reference we are trying to recover.

[6]The set of dangling objects is a subset of the set of unreachable objects. See `man gitglossary` for a better definition of a *dangling object* than stated in `man git-fsck`.

[7]The other dangling commit is the one we abandoned after correcting the rebase in section 14.1.

```
9709844 M1
```

Yes, that is our commit, so let's recover the branch:

term 168

```
main > git branch feature 2020caddc5b22a9e49dbd3eeb3d4af183d0b0f7e
main > git log --all --oneline --graph
* 2020cad (feature) F2
* 3d58cf8 F1
| * abcf01a (HEAD -> main) M4
| * 3f883d7 M3
|/
* 2a6b2cb M2
* 9709844 M1
```

As we can see, we could successfully recover our `feature` branch.

A Check Equivalence of Patch ID

We can find the patch ID using the commands `git show` and `git patch-id`. The former shows an object, and the latter computes a unique ID for a patch. To generate the patch ID, we pipe the output of `git show <commit>` into `git patch-id`.

In our case, we had the following situation:

term 169

```
bob main > git log --all --graph --oneline
* bc1c4f3 (origin/main, origin/HEAD) Implement feature D
| * 27148a1 (HEAD -> main) Implement feature E
| * 2ad5243 Implement feature D
| * 88e6df2 Implement feature C
| * 9eb88c8 Implement feature B
|/
* 3442e7b Implement feature A
* d506edd Init
```

We can observe the equivalent patch IDs of both commits that implement feature D as follows:

term 170

```
bob main > git show 2ad5243 | git patch-id
1f6bba223db57210c640053a9d8360604016a963
2ad52435c0f15d495ed22e9c02d8fca0c8309d90
bob main > git show bc1c4f3 | git patch-id
1f6bba223db57210c640053a9d8360604016a963
bc1c4f3e8464108fef4abf5e73e2216678067f62
```

`git patch-id` returns us the patch ID followed by the commit hash (the output would normally be one string separating the hashes by a space; a new line has been used here for better formatting). As we can see, both commits, `2ad5243` and `bc1c4f3`, have the same patch ID as we expected.

Bibliography

[CS14] Scott Chacon and Ben Straub. *Pro git: Everything you need to know about Git*. English. Second. Apress, 2014. URL: https://git-scm.com/book/en/v2.

[Hag17] Michael Haggerty. *Git*. Version 2.43.0. Commit 67be7c5 and 02b920f. Software Freedom Conservancy, 2017. URL: https://github.com/git/git/blob/3e0d3cd5c7def4808247caf168e17f2bbf47892b/refs/packed-backend.c#L620.