AMD

The Hitchhiker’s Guide to the Git

Russell Liu

21. Mai 2023

SRDC Khronos3D Group

Table of contents

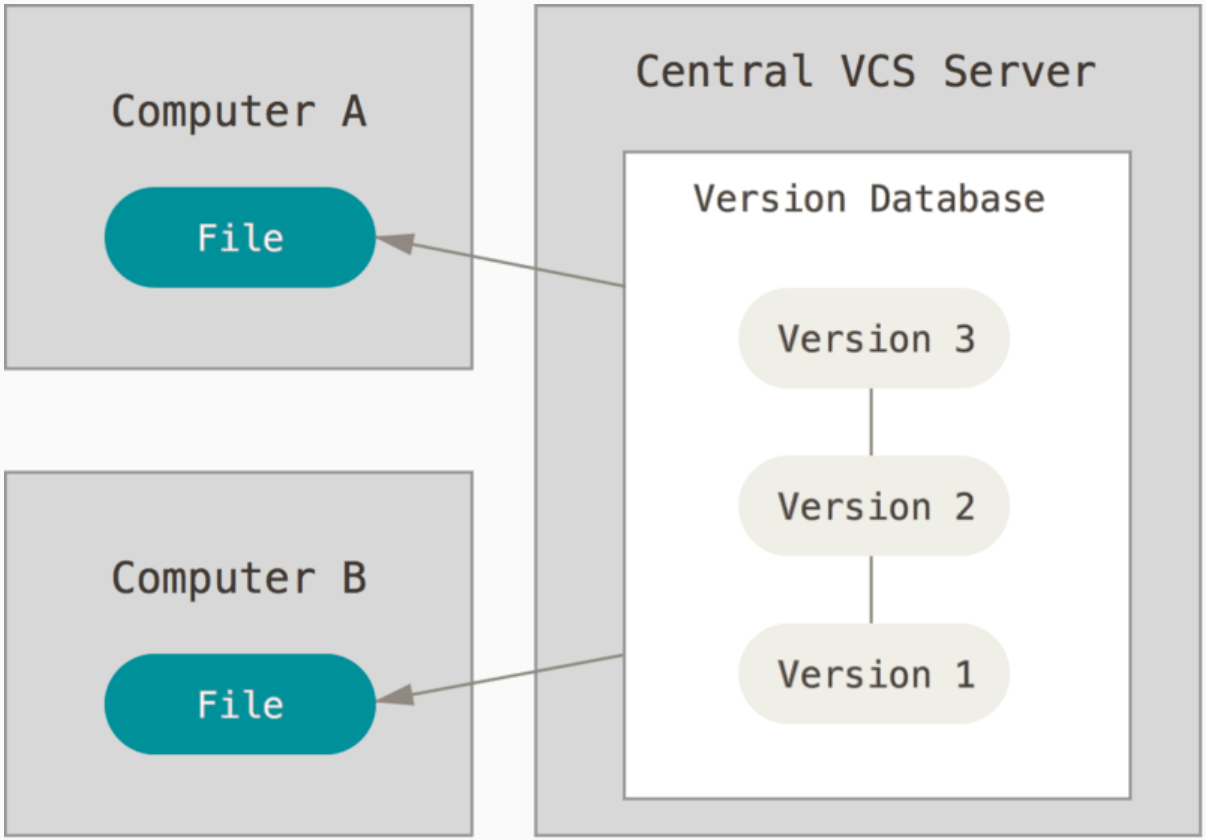
You can do a lot of things with git, and many of the rules of what you should do are not so much technical limitations but are about what works well when working together with other people. So git is a very powerful set of tools.

(Linus Torvalds)

Introduction

Centralized Version Control System

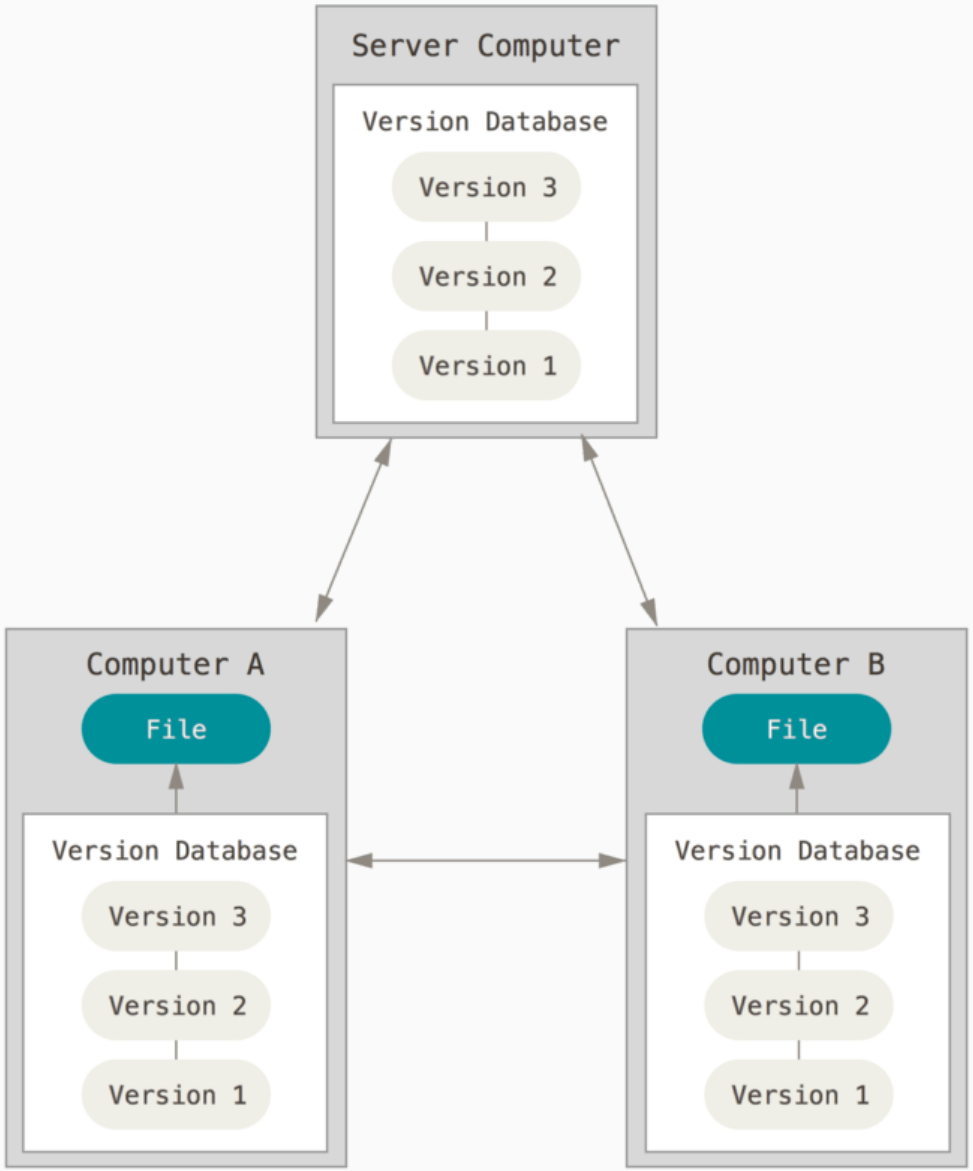
In a CVCS (such as CVS, Subversion, and Perforce) have a single server that contains all the versioned files, and a number of clients that check out files from that central place.



Distributed Version Control System

In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history. Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

Distributed Version Control System



Git

In 2002, the Linux kernel project began using a proprietary DVCS called BitKeeper.

In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down, and the tool's free-of-charge status was revoked. This prompted the Linux development community (and in particular Linus Torvalds, the creator of Linux) to develop their own tool based on some of the lessons they learned while using BitKeeper.

Git

Some of the goals of the new system were as follows:

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)

Comics

(a) Git commit

(b) Git

Abbildung 1: xkcd

Recording Changes

Getting a git repo

- Initializing a Repository

```
mkdir repo && cd repo
git init
git commit --allow-empty -m 'Initial commit'
```

- Cloning an Existing Repository

```
git clone git@github.com:github/gitignore.git
git clone git@github.com:gcc-mirror/gcc.git gnucc
git clone git@github.com:git/git.git -bv2.40.0
```

Checking the Status

The main tool you use to determine which files are in which state is the **git status** command.

You should see something like this:

```
$ git status
On branch master
nothing to commit, working tree clean
```

Checking the Status

Add a new file to your project, a simple **README** file. And run git status, you see your untracked file like so:

```
$ echo 'My Project' > README
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be
↪  committed)
        README
```

Tracking New Files

In order to begin tracking a new file, you use the command git add.

```
git add README
```

If you run your status command again, you can see that your file is now tracked and staged to be committed:

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   README
```

Staging Modified Files

At this point, suppose you remember one little change that you want to make in README before you commit it.

```
$ echo 'Second line' >> README
$ git status
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
```

Unmodifying a Modified File

What if you realize that you don't want to keep your changes to the README file? Luckily, git status tells you how to do that, too. In the last example output, the unstaged area looks like this:

```
$ git status
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
```

Committing Your Changes

Remember that anything that is still unstaged – any files you have created or modified that you haven't run git add on since you edited them – won't go into this commit. They will stay as modified files on your disk.

```
$ git commit -m 'Add README file'
[detached HEAD 21f1664] Add README file
 1 file changed, 1 insertion(+)
 create mode 100644 README
```

Lifecycle

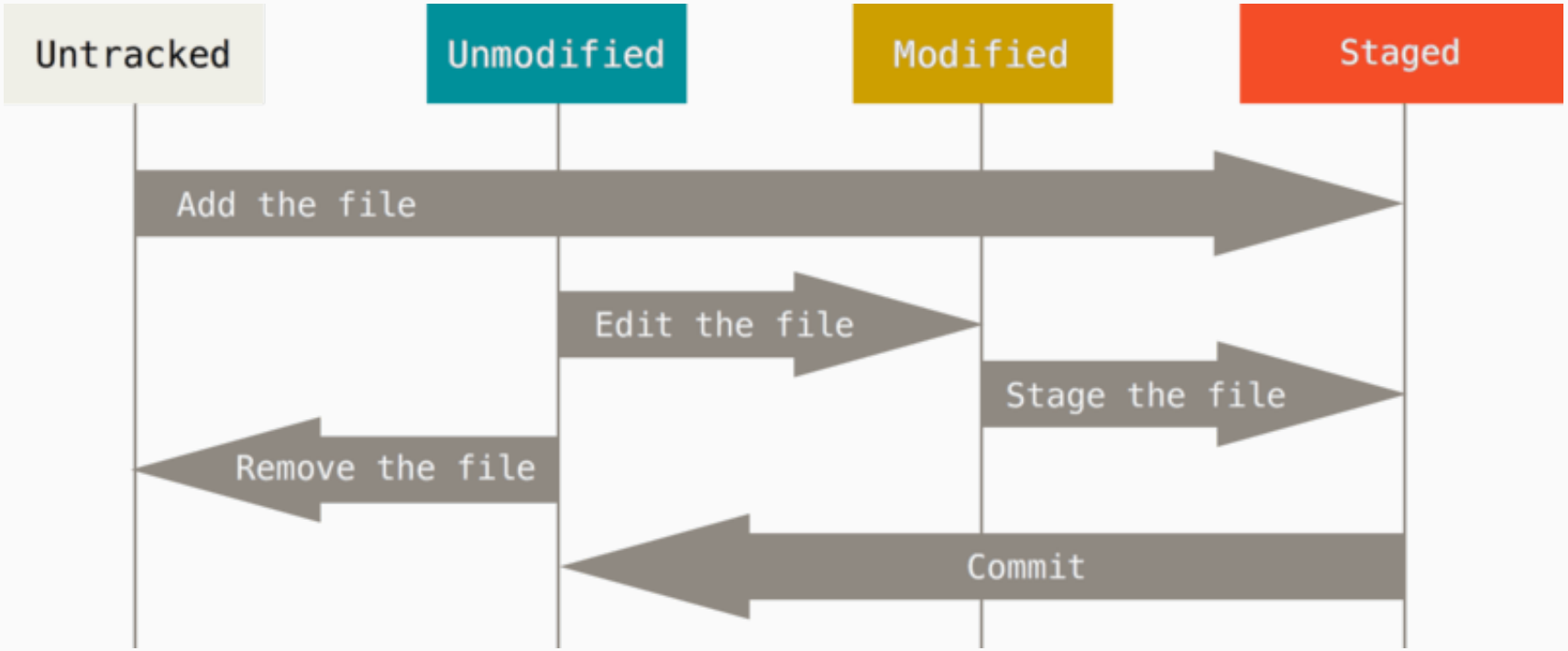


Abbildung 2: File lifecycle

What does the short state mean

If you run the command `git status` with option `--short`, you may see some weird characters. Or files are flaged some characters when use IDE, like M, A, etc.

	unmodified
M	modified
T	file type changed
A	added
D	deleted
R	renamed
C	copied
U	updated but unmerged

What does the short state mean

X	Y	Meaning
	[AMD]	not updated
M	[MTD]	updated in index
T	[MTD]	type changed in index
A	[MTD]	added to index
D		deleted from index
R	[MTD]	renamed in index
C	[MTD]	copied in index
[MTARC]		index and work tree matches
[MTARC]	M	work tree changed since index
[MTARC]	T	type changed in work tree since index
[MTARC]	D	deleted in work tree
?	?	untracked
!	!	ignored

Tabelle 1: Short format of file state

Viewing

Viewing the Commit History

After you have created several commits, or if you have cloned a repository with an existing commit history, you'll probably want to look back to see what has happened. The most basic and powerful tool to do this is the `git log` command.

Viewing the Commit History

When you run git log in this project, you should get output that looks something like this:

```
$ git log
commit 21f16642c07b6ec8d42ce64800d33410c54557ac
Author: Arthur Dent <Arthur.Dent@example.com>
Date:   Thu Apr 13 22:05:12 2023 +0800

    Add README file

commit 2ca005807c96def431c4a4e4dcdbe1185f00d3b0
Author: Arthur Dent <Arthur.Dent@example.com>
Date:   Thu Apr 13 22:04:29 2023 +0800

    Initial commit
```

Graphicalize the Log Output

The `oneline` option prints each commit on a single line, which is useful if you're looking at a lot of commits.

```
git log --oneline
```

The `branches` option prints all branches, which is useful if you're looking at relationships between branches.

```
git log --branches
```

Now output doesn't look good, I don't know relationships between commits. I want to get a **graph**, like gitlab.

```
git log --graph
```

Log Graph

```
git log --graph --oneline release/13.0.0
```

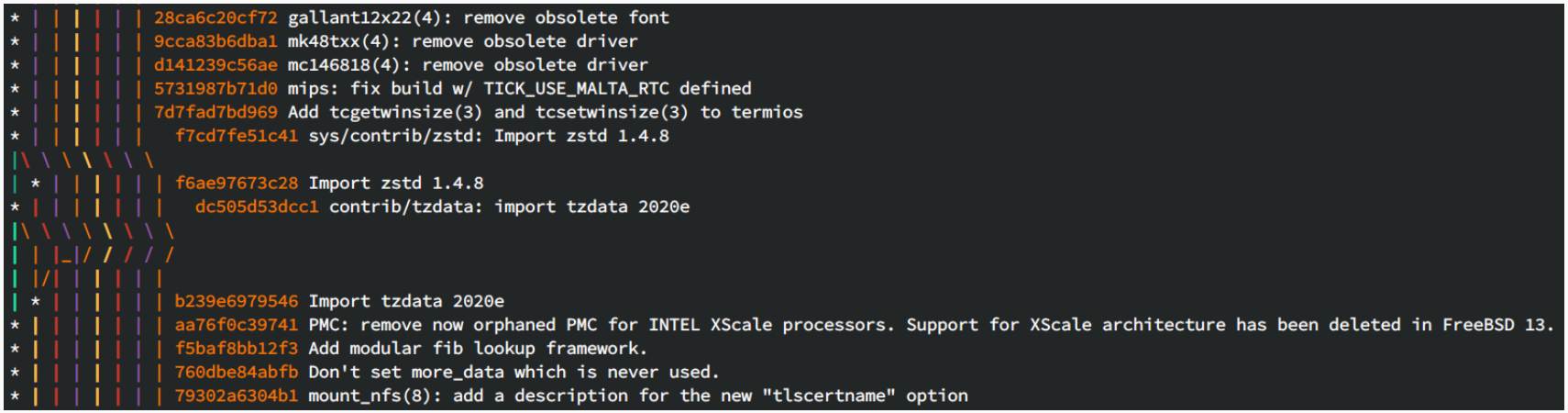


Abbildung 3: FreeBSD release13.0.0 excerpt

Compare changes

Show changes between commits, commit and workspace, etc.

```
git diff
```

No options means compare between workspace and staging.

```
$ git diff
diff --git a/README b/README
index 56266d3..ba4645b 100644
--- a/README
+++ b/README
@@ -1 +1,2 @@
 My Project
+Second line
```

Compare changes

If you want to compare a specified commit to workspace, use

```
git diff <BaseCommit>
```

If you want to compare any two commits, you can use

```
git diff <BaseCommit> <TargetCommit>
```

Who committed this line of code

Option **`blame`** shows what revision and author last modified each line of a file.

```
$ git commit -am 'New line for README'
$ git blame README
21f16642 (Arthur Dent 2023-04-13 22:05:12 +0800 1) My
↪  Project
bb65ea1b (Arthur Dent 2023-04-14 18:09:42 +0800 2) Second
↪  line
```

Show commited in large file

Luckly, Annotate only the line range is allowed.

You can add flag `-L <start>,<end>` to tell blame to print the line range. `<start>` and `<end>` can take one of these forms:

number If <start> or <end> is a number, it specifies an absolute line number.

/regex/ This form will use the first line matching the given POSIX regex. If `<start>` is ^/regex/, it will search from the start of file.

±offset This is only valid for `<end>` and will specify a number of lines before or after the line given by `<start>`.

```
$ git blame README -L 1,1
21f16642 (Arthur Dent 2023-04-13 22:05:12 +0800 1) My
↪  Project
```

Show commited in large file

If `:<funcname>` is given, it is a regular expression that denotes the range from the first funcname line that matches funcname, up to the next funcname line.

```
$ cat <<- EOF |tee main.c; git add .; git commit -m'Hello'
> int printf(char const *restrict format, ...);
> int main(void)
>   { printf("Hello World!\n"); }
> EOF
```

```
$ git blame main.c -L :main
75a7db03 (Arthur Dent 2023-04-14 18:19:51 +0800 2) int
↪  main(void)
75a7db03 (Arthur Dent 2023-04-14 18:19:51 +0800 3)   {
↪  printf("Hello World!\n"); }
```

Show details of commit

Options **show** shows one or more objects (blobs, trees, and commits).

```
$ git show 21f1664
commit 21f16642c07b6ec8d42ce64800d33410c54557ac
Author: Arthur Dent <Arthur.Dent@example.com>
Date:   Thu Apr 13 22:05:12 2023 +0800

    Add README file

diff --git a/README b/README
new file mode 100644
index 0000000..56266d3
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

Show details of file

```
$ git log -p --oneline -- README
bb65ea1 New line for README
diff --git a/README b/README
index 56266d3..ba4645b 100644
--- a/README
+++ b/README
@@ -1 +1,2 @@
 My Project
+Second line
21f1664 Add README file
diff --git a/README b/README
new file mode 100644
index 0000000..56266d3
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

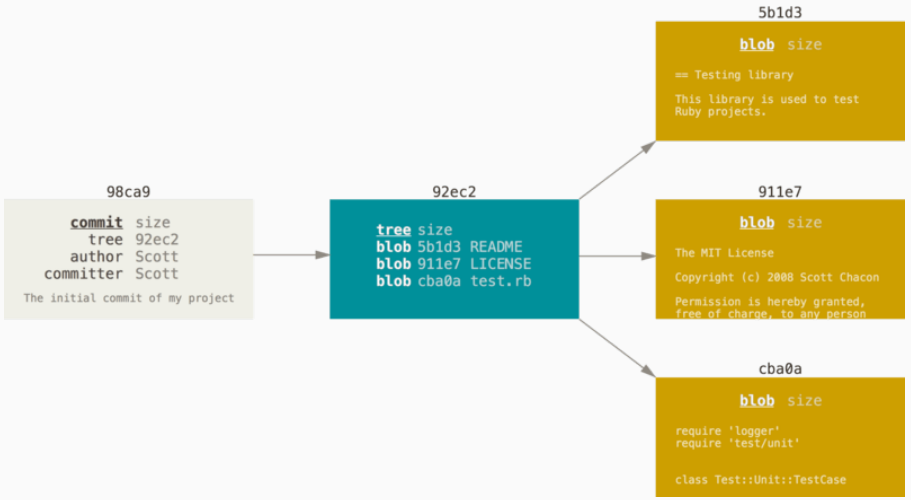
Committing

What is commit

commit example

```
git add README test.rb LICENSE
git commit -m 'Initial commit'
```

Your Git repository now contains five objects: three **blobs**[1], one **tree**[2], and one **commit**[3].



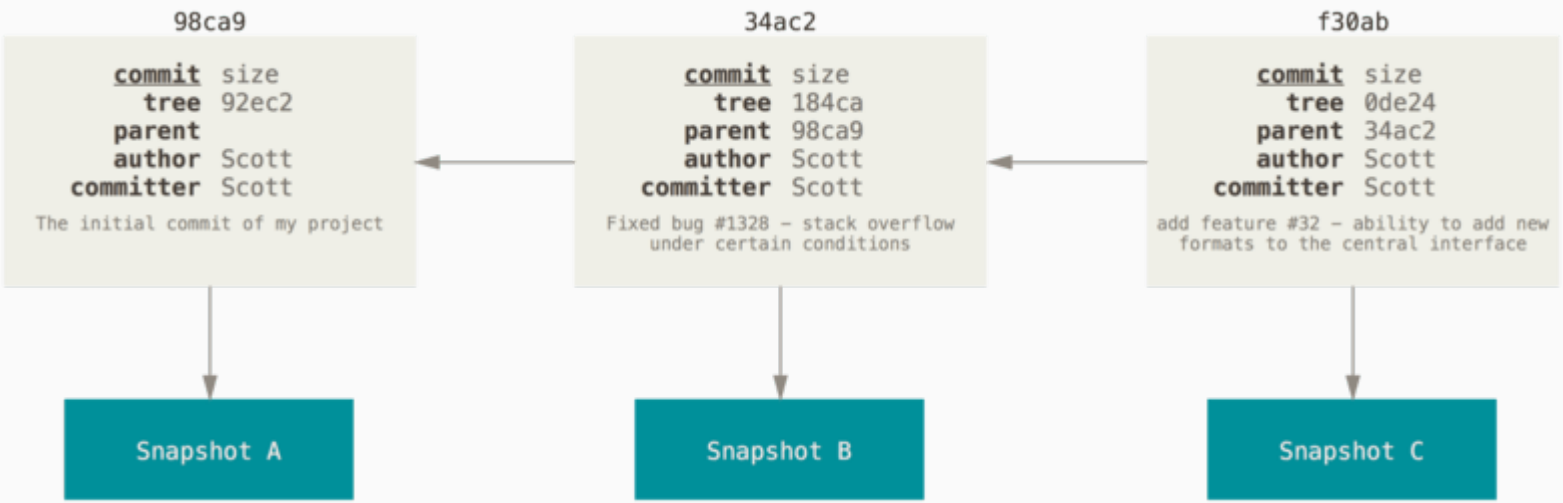
[1] each representing the contents of one of the one file

[2] lists the contents of the directory and specifies which file names are stored as which blobs

[3] root tree and all the commit metadata

Committing String

If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.



What is HEAD

How does Git know what `commit node` you're currently on? It keeps a special pointer called **HEAD**. Note that this is a lot different than the concept of HEAD in other VCSs you may be used to, such as Subversion or CVS. In Git, this is a pointer to the local branch you're currently on.

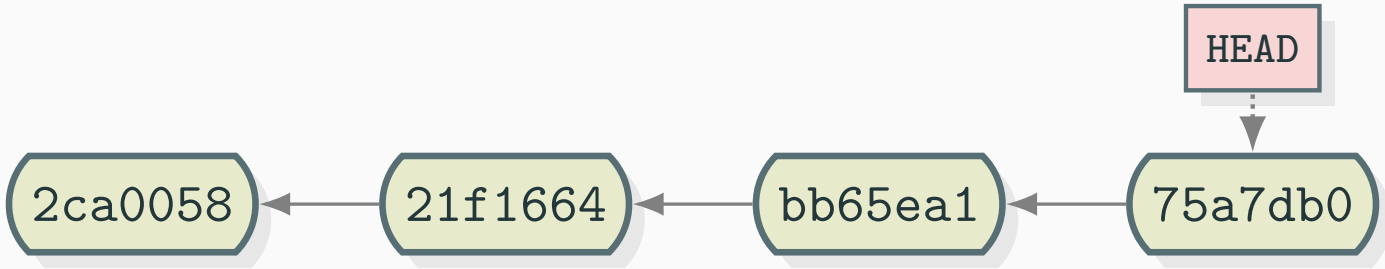


Abbildung 4: HEAD pointer

Relative position of commit

The tilde(~) sign refers to the predecessor in the commit history.

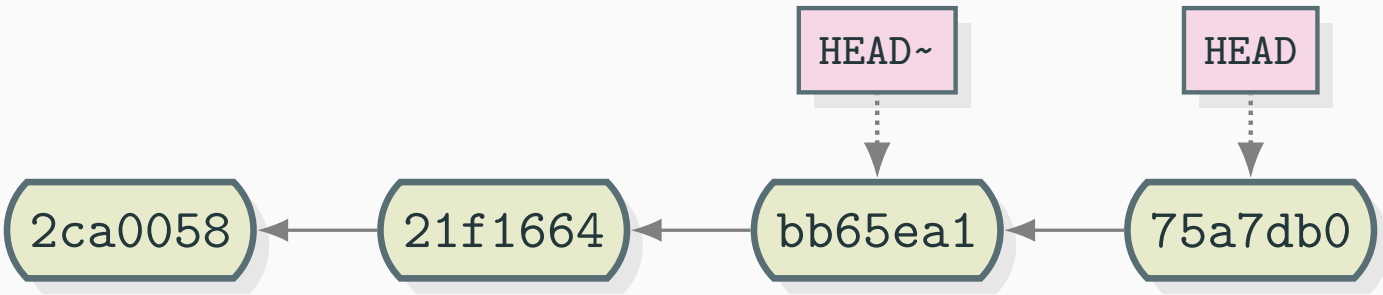


Abbildung 5: Relative position

Relative position of commit

The tilde(~) sign refers to the predecessor in the commit history. And ~<n> means the nth predecessor. As a special rule, <rev>~0 means the commit itself and is used when <rev> is the object name of a tag object that refers to a commit object.

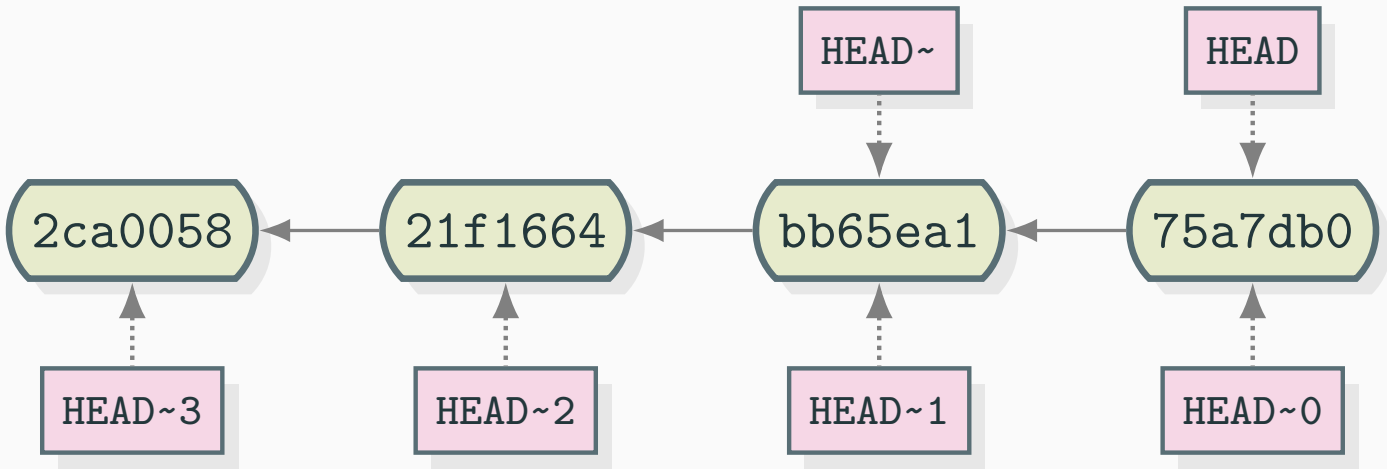


Abbildung 5: Relative position

Move HEAD

Option `siwtch` looks good! Let use try to move the pointer to the predecessor of **HEAD**.

```
$ git switch HEAD~2
fatal: a branch is expected, got commit 'HEAD~2'
hint: If you want to detach HEAD at the commit, try again
↪  with the --detach option.
```

Oops! Something wrong. Try to add `--detach`.

Move HEAD

```
$ git switch --detach HEAD~2
HEAD is now at 21f1664 Add README file
```

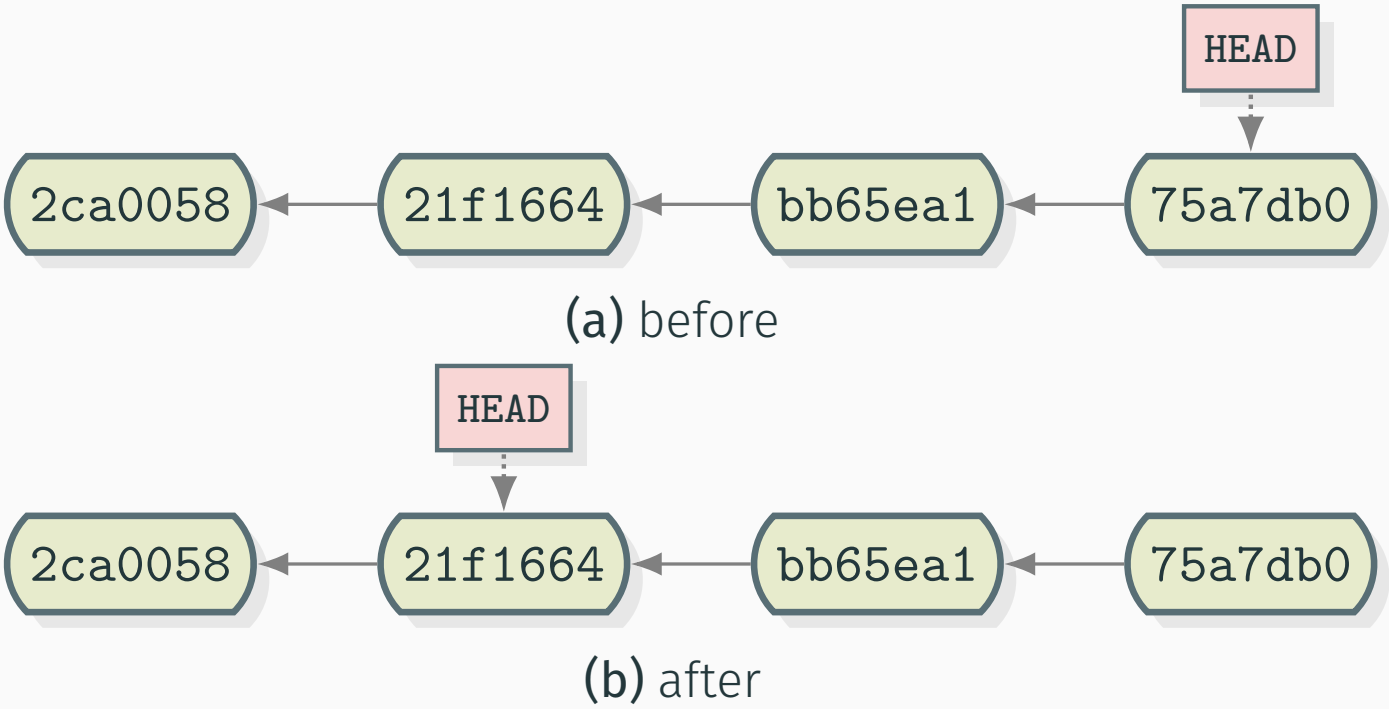


Abbildung 6: Move pointer to predecessor

Another way to move HEAD

`checkout` command also move the HEAD pointer.

For example:

```
git checkout <commit>
```

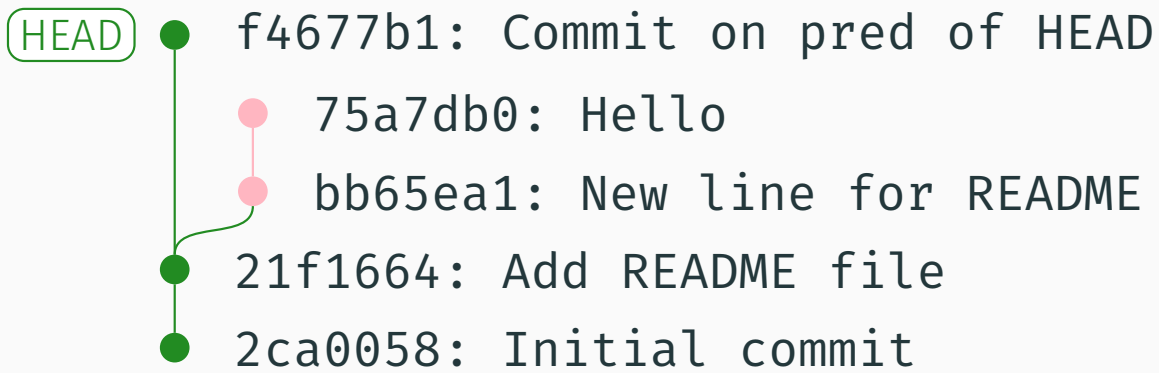
What happens if we commit now

```
echo 'Third line' > README
git commit -am 'Commit on pred of HEAD'
```

What happens if we commit now

```
$ echo 'Third line' > README
$ git commit -am 'Commit on pred of HEAD'
[detached HEAD f4677b1] Commit on pred of HEAD
 1 files changed, 1 insertions(+)
 create mode 100644 README
```

Commit successfully! Let's look the commits.



Copy the commit node

`cherry-pick` applies the changes introduced by some existing commits.

Given one or more existing commits, apply the change each one introduces, recording a new commit for each. This requires your working tree to be clean (no modifications from the HEAD commit).

```
git cherry-pick <commit>...
```

Copy the commit node

`cherry-pick` applies the changes introduced by some existing commits.

Okay! Apply the bb65ea1 NOW!

```
$ git cherry-pick bb65ea1
Auto-merging README
CONFLICT (content): Merge conflict in README
error: could not apply bb65ea1... New line for README
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".
hint: To abort and get back to the state before "git
↪  cherry-pick",
hint: run "git cherry-pick --abort".
```

What happens when cherry-pick running

When it is not obvious how to apply a change, the following happens:

1. The current branch and HEAD pointer stay at the last commit successfully made.
2. The CHERRY_PICK_HEAD ref is set to point at the commit that introduced the change that is difficult to apply.
3. Paths in which the change applied cleanly are updated both in the index file and in your working tree.
4. For conflicting paths, the index file records up to three versions, as described in the "TRUE MERGE" section of git-merge(1). The working tree files will include a description of the conflict bracketed by the usual conflict markers <<<<<<< and >>>>>>>.
5. No other modifications are made.

Conflict

1. Which file is in conflict?
2. What contents is in conflict?
3. How to resolve conflicts?

Conflict

Which file is in conflict?

```
$ git status
HEAD detached from 21f1664
You are currently cherry-picking commit bb65ea1.
  (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 <file>..." to mark resolution)
        both modified:   README
```

Conflict

What contents is in conflict?

```
$ cat README
<<<<<<< HEAD
Third line
=======
My Project
Second line
>>>>>>> bb65ea1 (New line for README)
```

Conflict

How to resolve conflicts?

- Put together the changes you need. (I select above.)

```
$ cat README
<<<<<<< HEAD
My Project
Second line
Third line
=======
My Project
Second line
>>>>>>> bb65ea1 (New line for README)
```

Conflict

How to resolve conflicts?

- Put together the changes you need.
- Delete code of another side.

```
$ cat README
<<<<<<< HEAD
My Project
Second line
Third line
=======
>>>>>>> bb65ea1 (New line for README)
```

Conflict

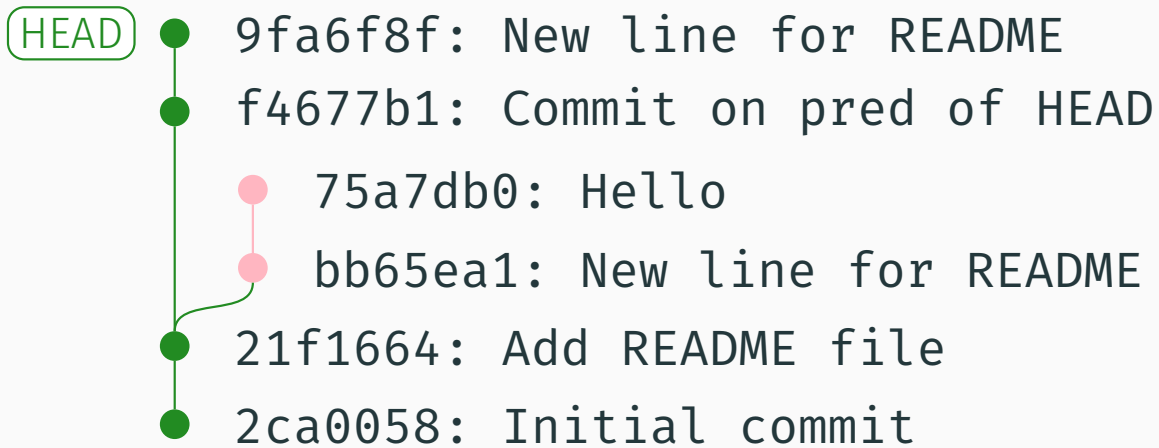
How to resolve conflicts?

- Put together the changes you need.
- Delete code of another side.
- Remove the prompt of conflict.

```
$ cat README
My Project
Second line
Third line
```

Continue to cherry-pick

```
$ git add README
$ git cherry-pick --continue
[detached HEAD 9fa6f8f] New line for README
 Date: Fri Apr 14 18:09:42 2023 +0800
 1 file changed, 2 insertions(+)
```



Picking a commits sequence

You can cherry-pick a range of commits by using the dot notation.

```
git cherry-pick A..B
```

To emphasize, range is a **half-open** interval with a maximum but no minimum. That is,

$$(A, B].$$

What is rebase

This operation works by going to the common ancestor of the two commit string (the one you're on and the one you're rebasing onto), getting the diff introduced by each commit you're on, saving those diffs to temporary files, resetting the current string to the same commit as the branch you are rebasing onto, and finally applying each change in turn.

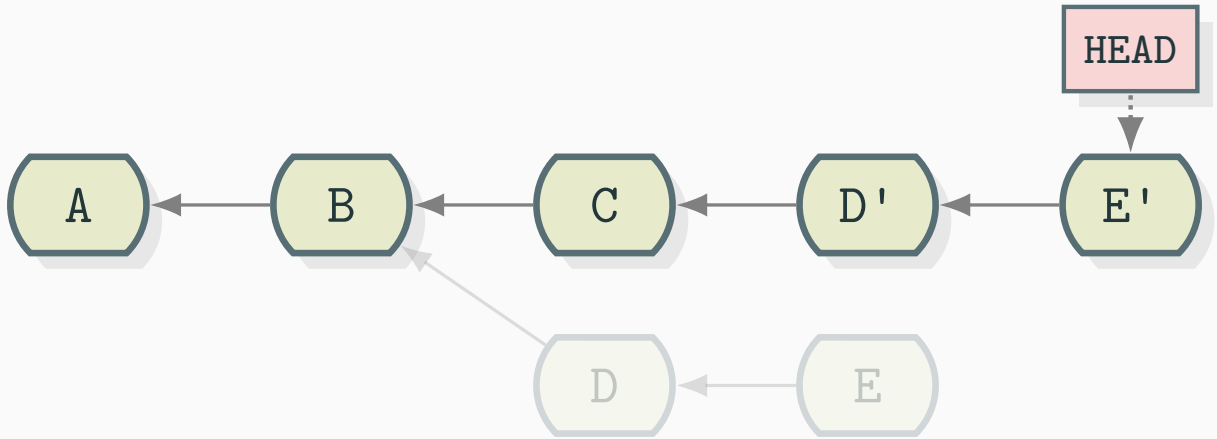


Abbildung 7: A typical `rebase`

Already contains a change

If the another string already contains a change you have made, then that commit will be skipped and warnings will be issued. For example (in which `C'` and `C` introduce the same set of changes, but have different committer information)

```
git rebase D
```

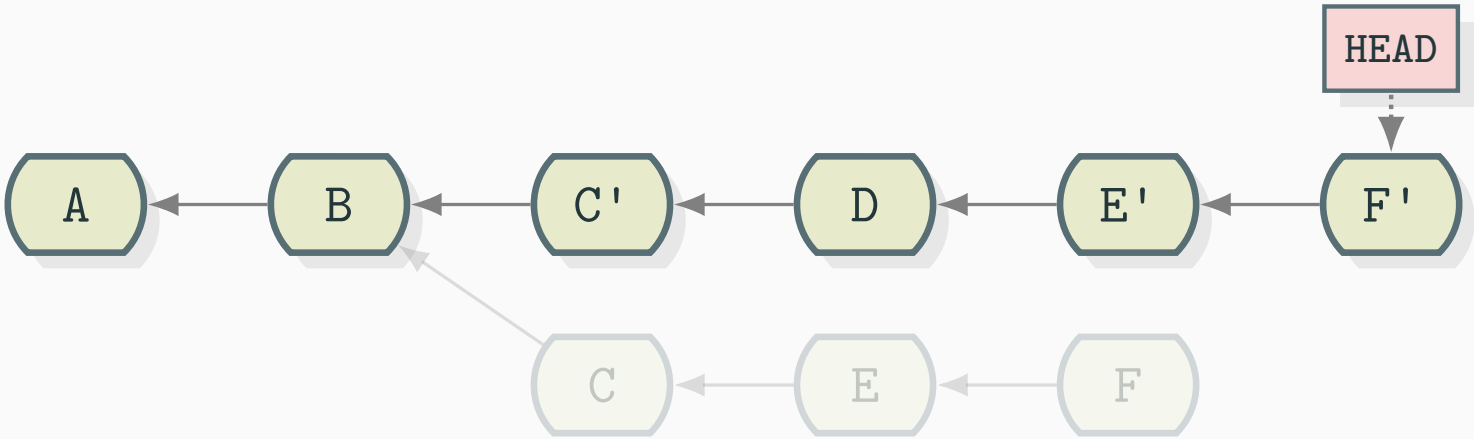


Abbildung 8: Already contains a change

Rebasing string off another string

You can also have your rebase replay on something other than the rebase target string. For example:

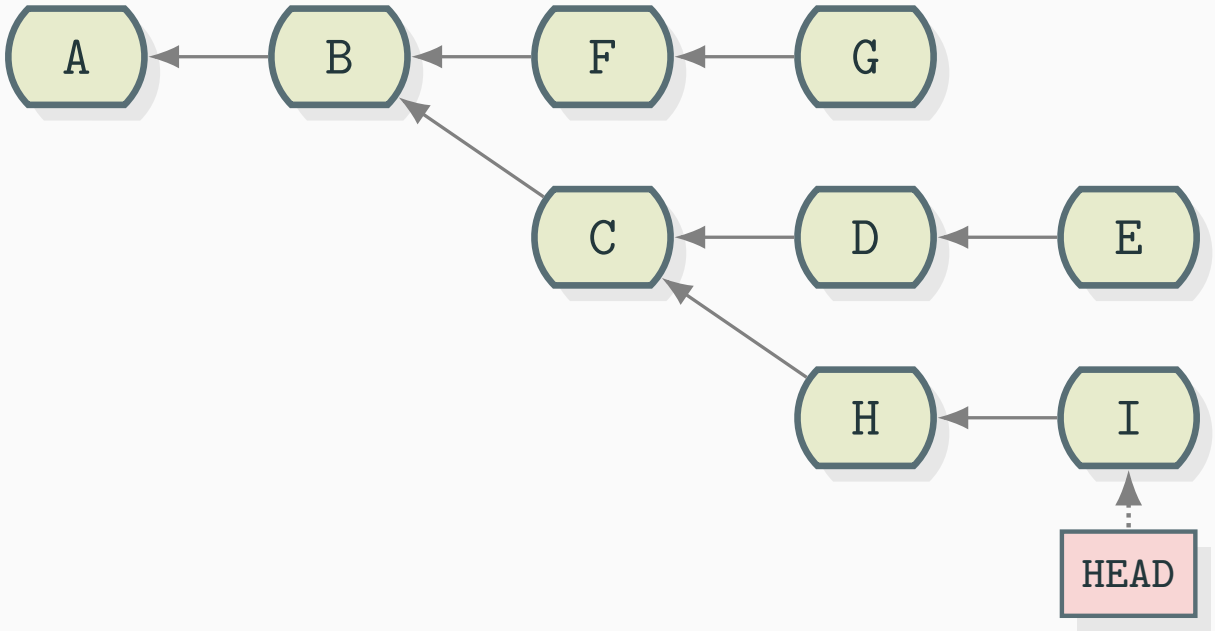


Abbildung 9: A history with a string off another string

Rebasing string off another string

```
git rebase --onto G E I
```

This basically says, “Take the `I` string, figure out the patches since it diverged from the `E` string, and replay these patches in the `I` string as if it was based directly off the `G` string instead.”

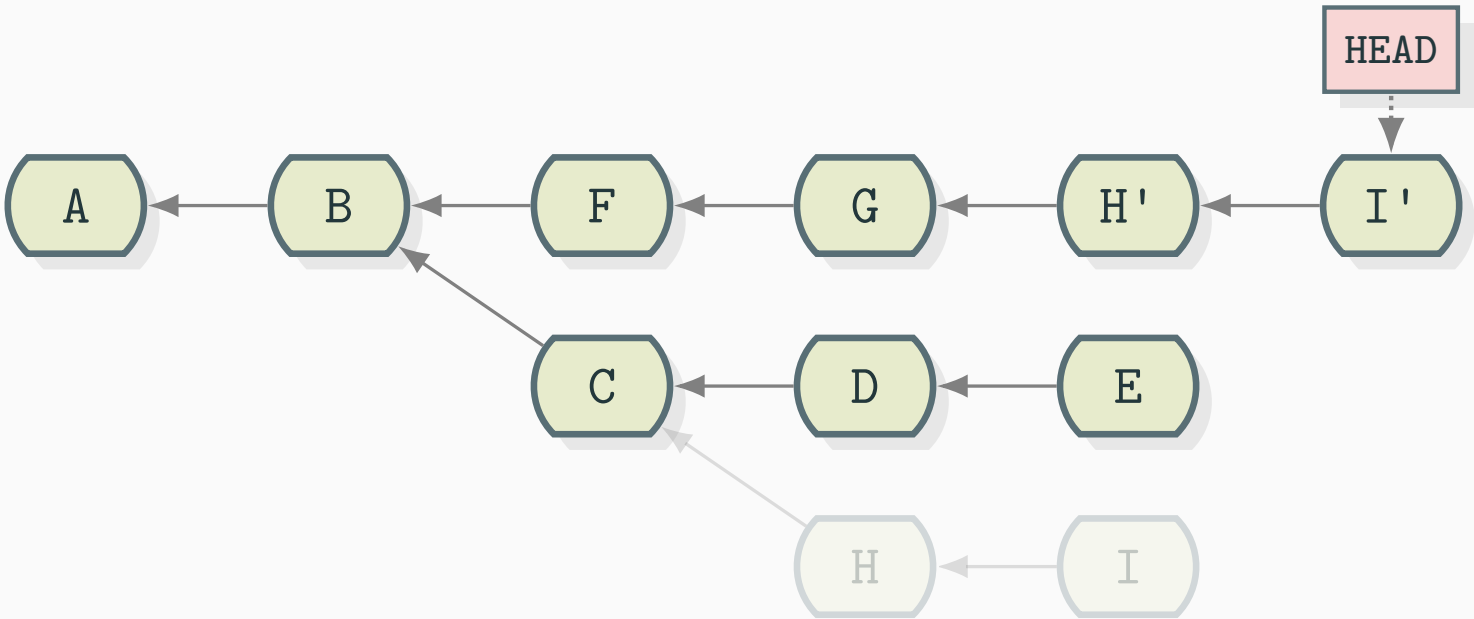


Abbildung 9: Rebasing a string off another string

Rebasing part of a string

Another example of `--onto` option is to rebase part of a string. If we have the following situation:

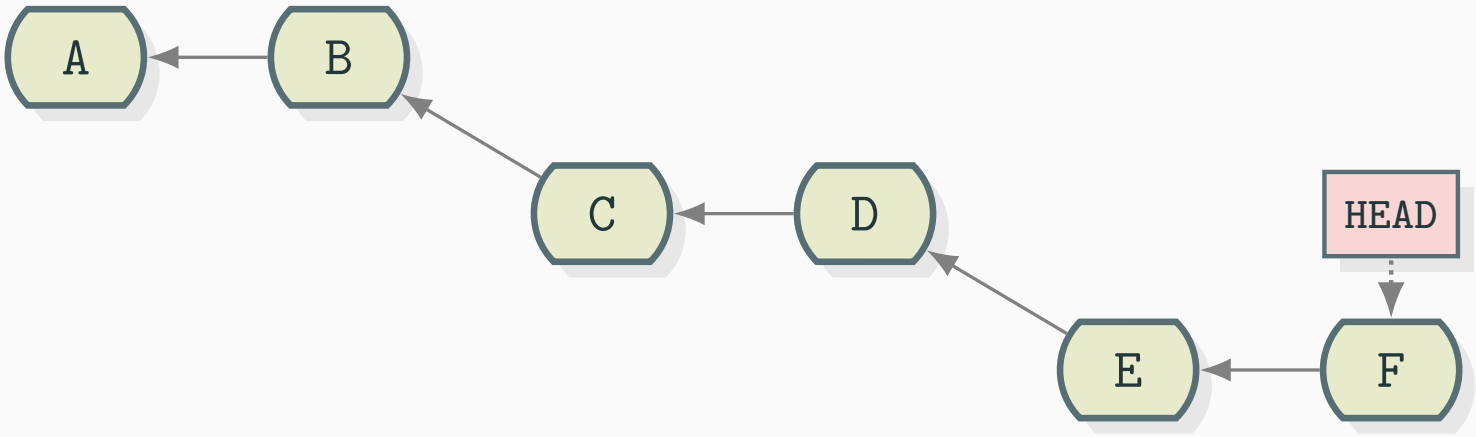


Abbildung 10: A history with a commit string

Rebasing part of a string

```
git rebase --onto B D F
```

would result in:

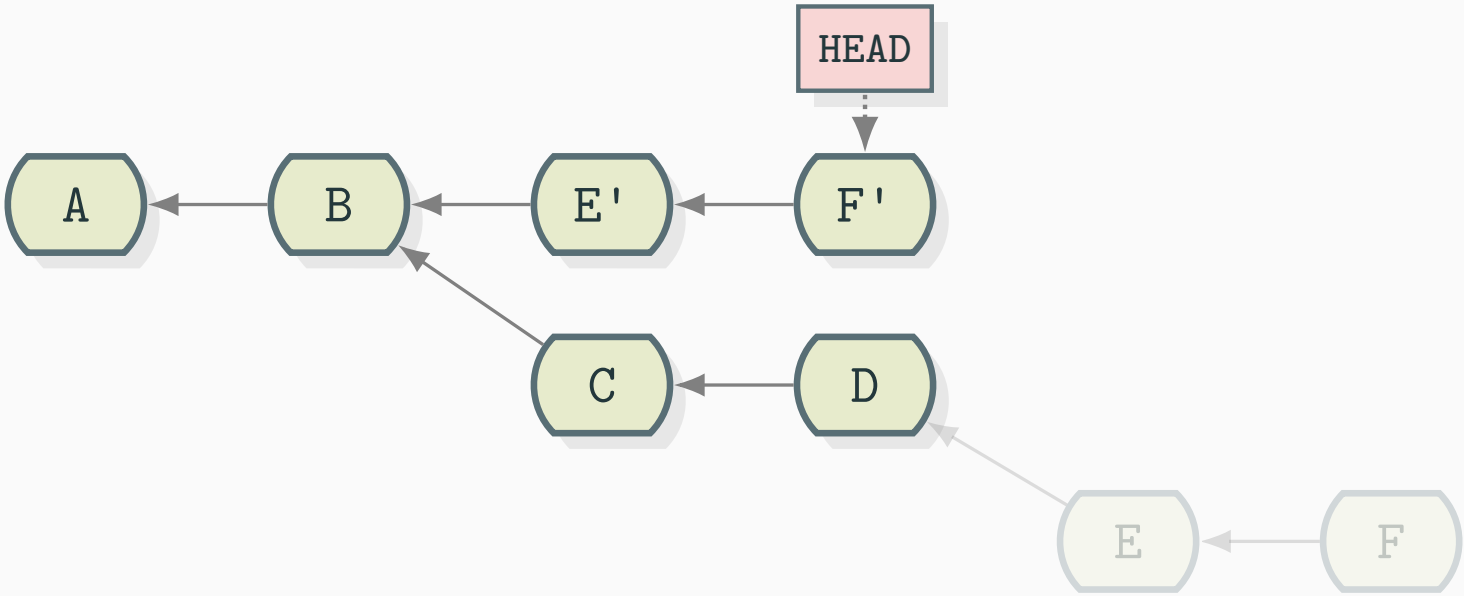


Abbildung 10: Rebasing with a part of commit string

The perils of rebasing

Ahh, but the bliss of rebasing isn't without its drawbacks, which can be summed up in a single line:

Do NOT rebase commits that exist outside your repository and that people may have based work on.

In other words,

Do NOT amend commits that exist outside your repository and that people may have based work on, including `cherry-pick`, `rebase`, and etc.

Changing multiple commit messages

To modify a commit that is farther back in your history, you must move to more complex tools. Git doesn't have a modify-history tool, but you can use the rebase tool to rebase a series of commits onto the HEAD that they were originally based on instead of moving them to another one. You can run rebase interactively by adding the `-i` option to git rebase.

Changing multiple commit messages

For example, if you want to change the last three commit messages:

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

Remember again that this is a rebasing command — every commit in the range `HEAD~3..HEAD` with a changed message and all of its descendants will be rewritten.

git rebase -i HEAD~3

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
```

Interactive commands

Short	Long	Meaning
p	pick	use commit
r	reword	use commit, but edit the commit message
e	edit	use commit, but stop for amending
s	squash	use commit, but meld into previous commit
f	fixup	like “squash”, but discard this commit's log message
x	exec	run command (the rest of the line) using shell
b	break	stop here (continue with ‘git rebase --continue’)
d	drop	remove commit

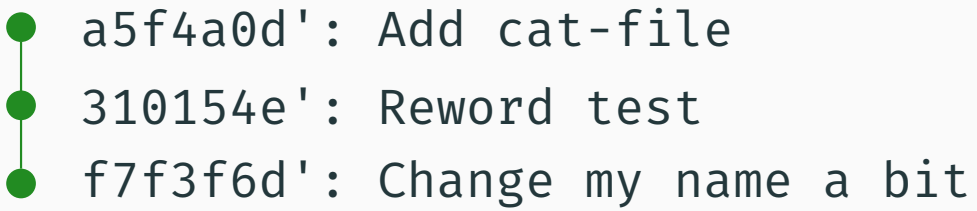
Tabelle 2: Common commands for interactive rebase

Example reword

git rebase -i HEAD~3

```
pick f7f3f6d Change my name a bit
reword 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

Modify message to “Reword test”. After rebase like this:

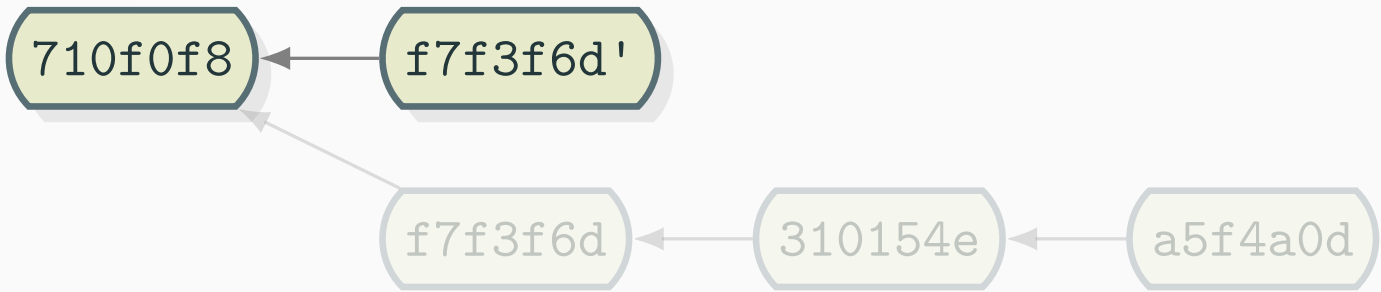


Example squash

git rebase -i HEAD~3

```
pick f7f3f6d Change my name a bit
fixup 310154e Update README formatting and add blame
fixup a5f4a0d Add cat-file
```

Remember, must have an existing commit as a basis for squash. After rebase like this:



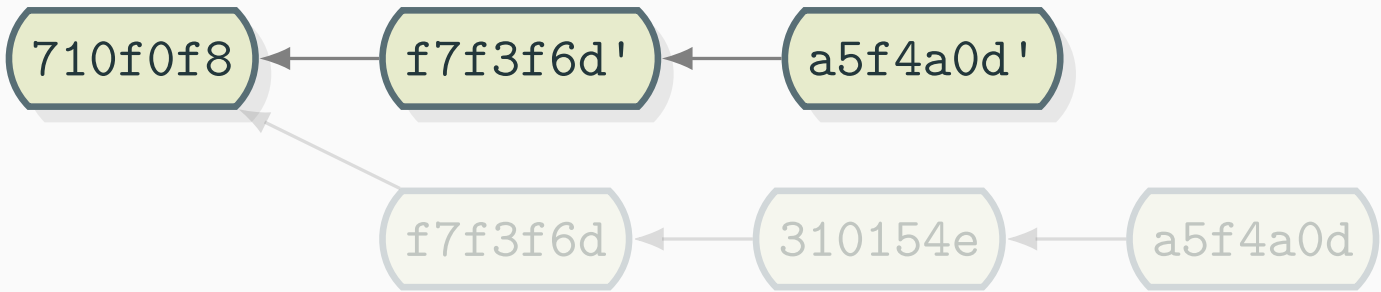
Now all changes of f7f3f6d, 310154e and a5f4a0d in f7f3f6d'.

Example drop

git rebase -i HEAD~3

```
pick f7f3f6d Change my name a bit
drop 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

By the way, remove the line has same effect. After rebase like this:



Now no changes in 310154e were committed.

Formatting patch

`format-patch` is used to generate a series of patches in mbox format that you can use to send to a mailing list properly formatted. Each patch consists of three parts:

- A brief metadata header that begins with from commit with a fixed datestamp to help programs like “file(1)” to recognize that the file is an output from this command, fields that record the author identity, the author date, and the title of the change.
- The second and subsequent paragraphs of the commit log message.
- The “patch”, which is the “diff -p `--`stat” output between the commit and its parent.

Generate email patches

Try to run the command.

```
$ git format-patch HEAD~2
0001-Commit-on-pred-of-HEAD.patch
0002-New-line-for-README.patch
```

Every commit in the range `HEAD~2..HEAD` will be generated patch file.

Generate email patches

We got two files to see what it looks like.

```
$ cat 0001-Commit-on-pred-of-HEAD.patch
From f4677b167ac3cbdf141c6c8967a4b6261f47559c Mon Sep 17
↪  00:00:00 2001
From: Arthur Dent <Arthur.Dent@example.com>
Date: Fri, 14 Apr 2023 18:38:15 +0800
Subject: [PATCH 1/2] Commit on pred of HEAD

---
 README | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/README b/README
index 56266d3..17354a8 100644
--- a/README
+++ b/README
@@ -1 +1 @@
-My Project
+Third line
```

Apply email patches

`am` is used to apply patches from an email inbox, specifically one that is mbox formatted. This is useful for receiving patches over email and applying them to your project easily.

You can apply one commit use command

```
git am 0001-Commit-on-pred-of-HEAD.patch
```

Or apply all the patch files, git will automatically sort the files.

```
git am *.patch
```

Resolve failed apply

Perhaps your main commit string has diverged too far from the patch was built from, or the patch depends on another patch you haven't applied yet. The `git am`process will fail and ask you what you want to do: `git am --resolved`.

This command puts conflict markers in any files it has issues with, much like a conflicted merge or rebase operation. You solve this issue much the same way — edit the file to resolve the conflict, stage the new file, and then run the command to continue to the next patch.

Intelligently resolve conflicts in am

If you want Git to try a bit more intelligently to resolve the conflict, you can pass a `-3` option to it, git tries a bit more intelligently to resolve the conflict.

```
git am -3 0001-Commit-on-pred-of-HEAD.patch
```

This option isn't on by default because it doesn't work if the commit the patch says it was based on isn't in your repository.

Apply changes quickly

Maybe `format-patch` and `am` are too formal.

The `apply` command applies a patch created with the git diff or even GNU diff command. It is similar to what the patch command might do with a few small differences.

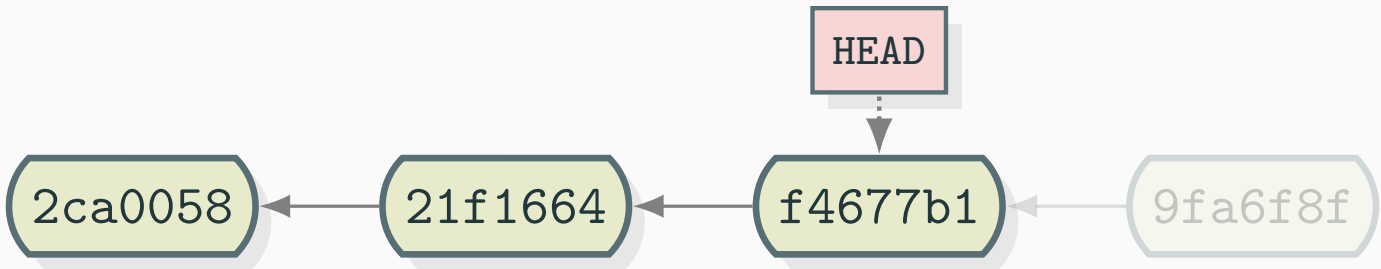
For example,

```
git diff HEAD~1 > fix.patch
git apply fix.patch
```

Reset HEAD

`reset` command will reset current HEAD to the specified state.

```
$ git reset HEAD~
Unstaged changes after reset:
M       README
```



Behavior of resetting

This form resets the current head to specified commit and possibly updates the index and the working tree depending on **mode**. If mode is omitted, defaults to *--mixed*. The mode must be one of the following:

--soft Does not touch the index file or the working tree at all. This leaves all your changed files “Changes to be committed”, as git status would put it.

--mixed Resets the index but not the working tree and reports what has not been updated.

--hard Resets the index and working tree. Any changes to tracked files in the working tree since commit are discarded. Any untracked files or directories in the way of writing any tracked files are simply deleted.

Details for soft resetting

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

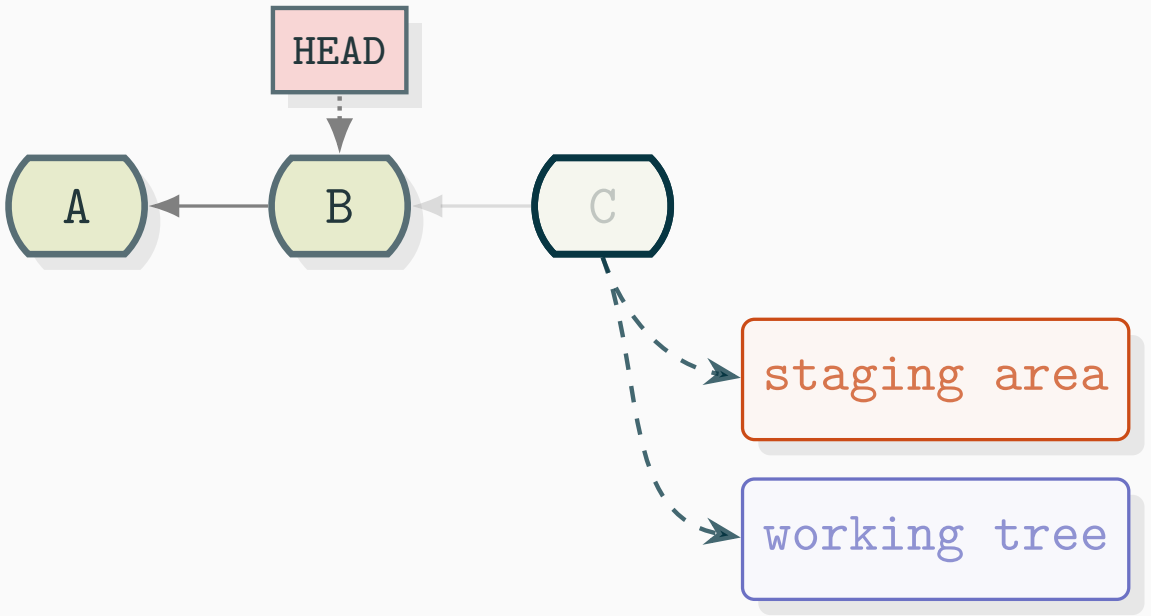


Abbildung 11: Resetting with soft mode

That is, soft mode only moves HEAD pointer.

Details for mixed resetting

```
git reset --mixed HEAD~
```

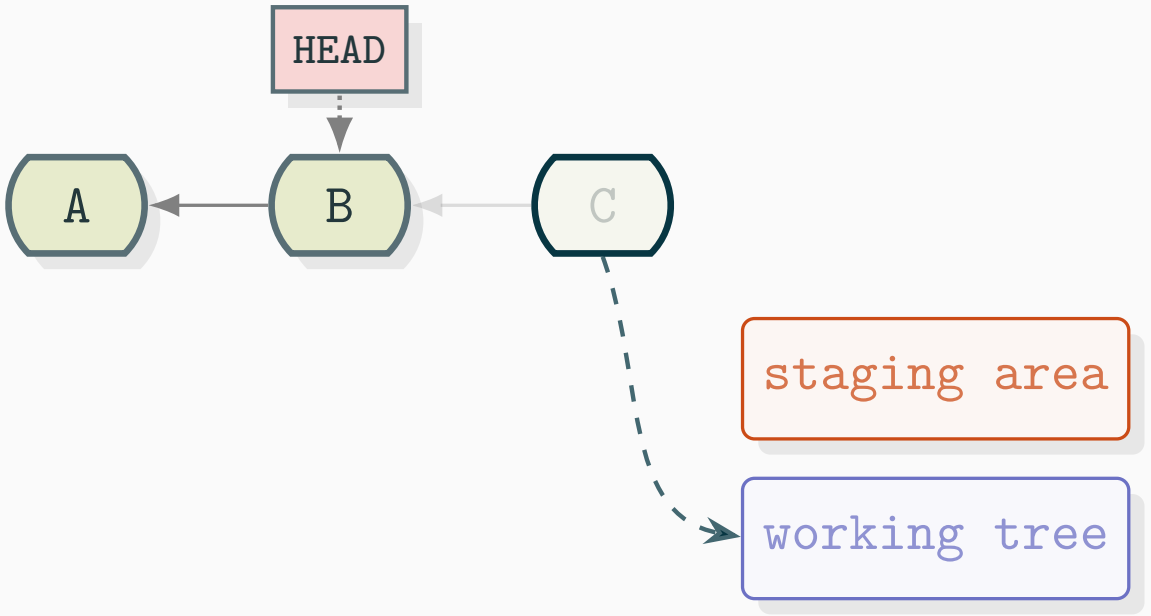


Abbildung 12: Resetting with mixed mode

That is, mixed mode undid your last commit, but also unstaged everything. You rolled back to before you ran all your git add and git commit commands.

Details for hard resetting

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

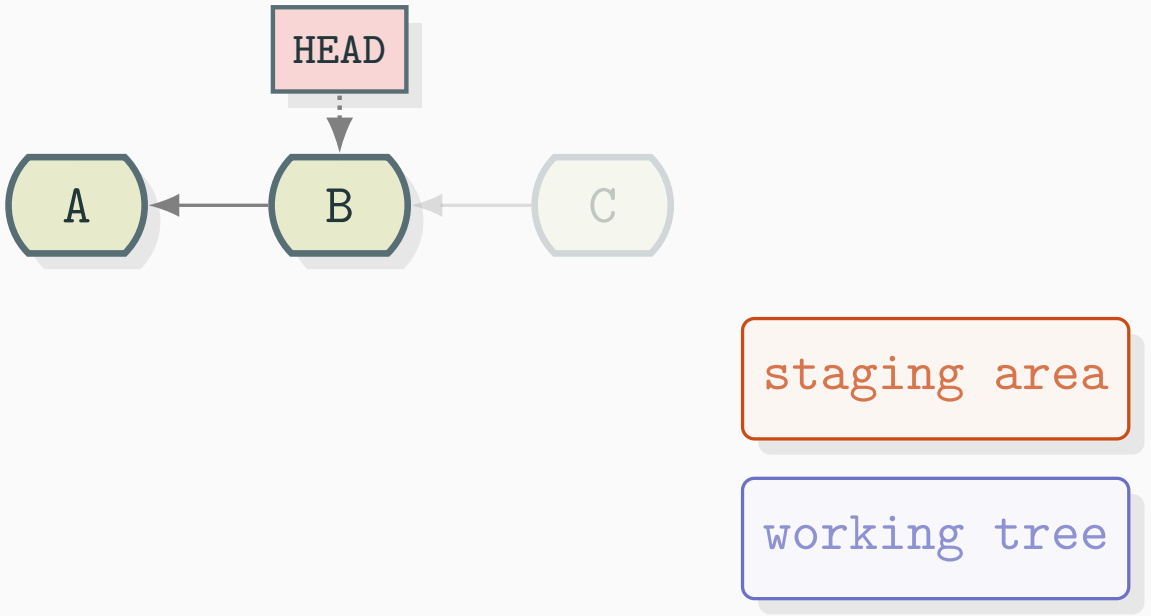


Abbildung 13: Resetting with hard mode

That is, hard mode undid your last commit, the git add and git commit commands, and all the work you did in your working directory.

Revert commits

If you want to revert some commits, reset may be a good command when you work on yourself working tree. But on public working tree we can NOT amend commits, that is, resetting is not allowed.

Given one or more existing commits, revert the changes that the related patches introduce, and record some new commits that record them.

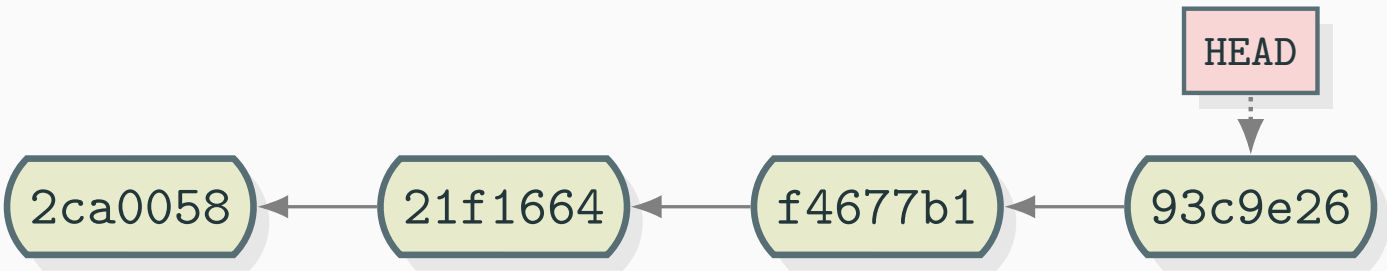
This requires your working tree to be clean (no modifications from the HEAD commit).

Try revert command

Revert the changes specified by the last commit in HEAD and create a new commit with the reverted changes.

```
$ git revert HEAD
[detached HEAD 93c9e26] Revert "Commit on pred of HEAD"
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Hm, a new commit!



Try revert command

Look what has changed.

```
$ git diff HEAD~ HEAD
diff --git a/README b/README
index 17354a8..56266d3 100644
--- a/README
+++ b/README
@@ -1 +1 @@
-Third line
+My Project
```

Look what different bewteen HEAD~2 and HEAD.

```
$ git diff HEAD~2 HEAD
```

Of cause, exactly the same!

Default revertting message

Default revertting message is `Revert "<Commit Message>"`.

If you think the message is good, and want not to edit it, you can add `--no-edit` option.

```
git revert --no-edit <commit>
```

Changing files when revertting

If given the command `--no-commit` option, you can change some files before commit. When you're done making changes, just run `--continue` option.

```
git revert --no-commit <commit>
# changing files
git revert --continue --no-edit
```

Revertting a commits sequence

Like `cherry-pick`, `revert` command alao revertting a range of commits by using the dot notation.

```
git revert A..B
```

`revert` command will create a new commit for each commit in order from newest to oldest. Looks like,

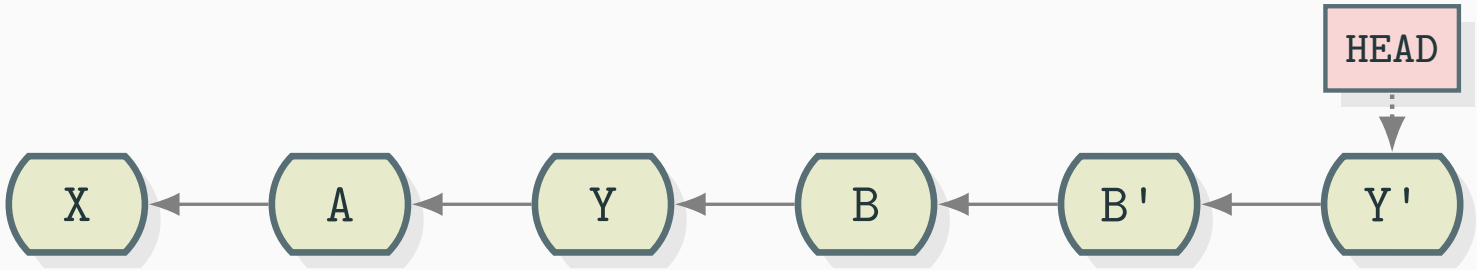


Abbildung 14: Revertting the commits sequence

Revertting vs. Reseting

	Revert	Reset
Behevior	back to previous and commit	back to the previous commits
History	more commits	less commits
Worktree	Nothing	depends on mode
Target	shared branch	private branch

Tabelle 3: Comparison between `revert` and `reset`

Revertting vs. Reseting

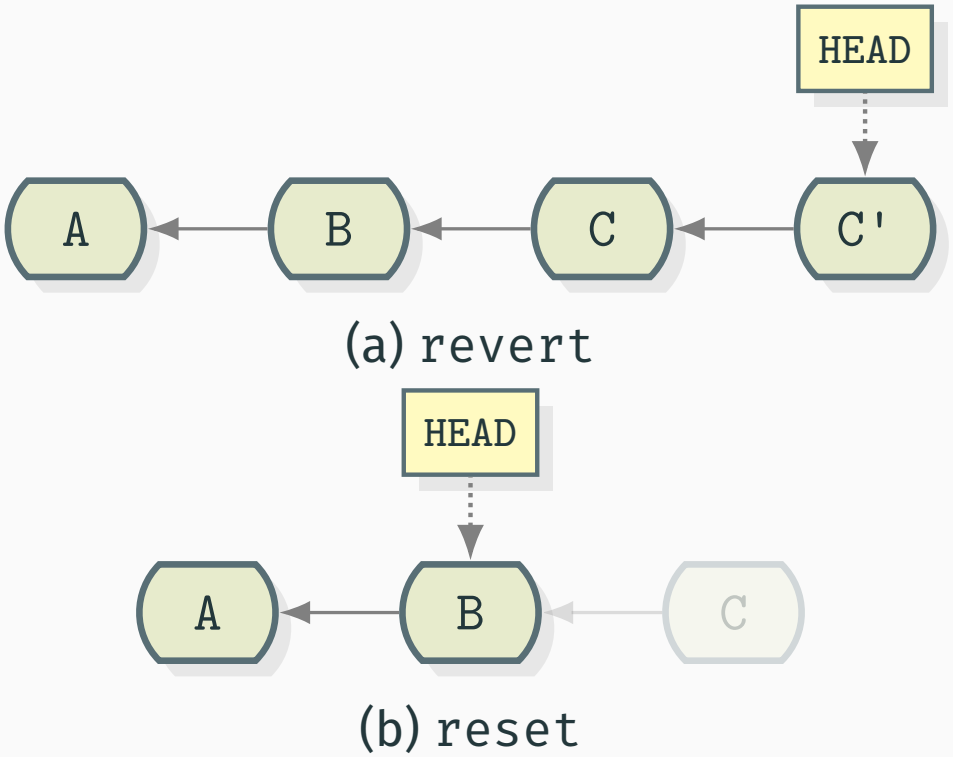


Abbildung 15: Comparison between `revert` and `reset`

Branching

Branches in a nutshell

Branching means you diverge from the main line of development and continue to do work without messing with that main line.

The way Git branches is incredibly lightweight, making branching operations nearly instantaneous, and switching back and forth between branches generally just as fast.

What is branch

A branch is a special HEAD, bound to a commit string, cannot be moved.

Which branch is it on now

HEAD point to the local branch you're currently on.

For example, now branch currently on master.

```
$ git switch master
Switched to branch 'master'
$ git branch --show-current
master
```

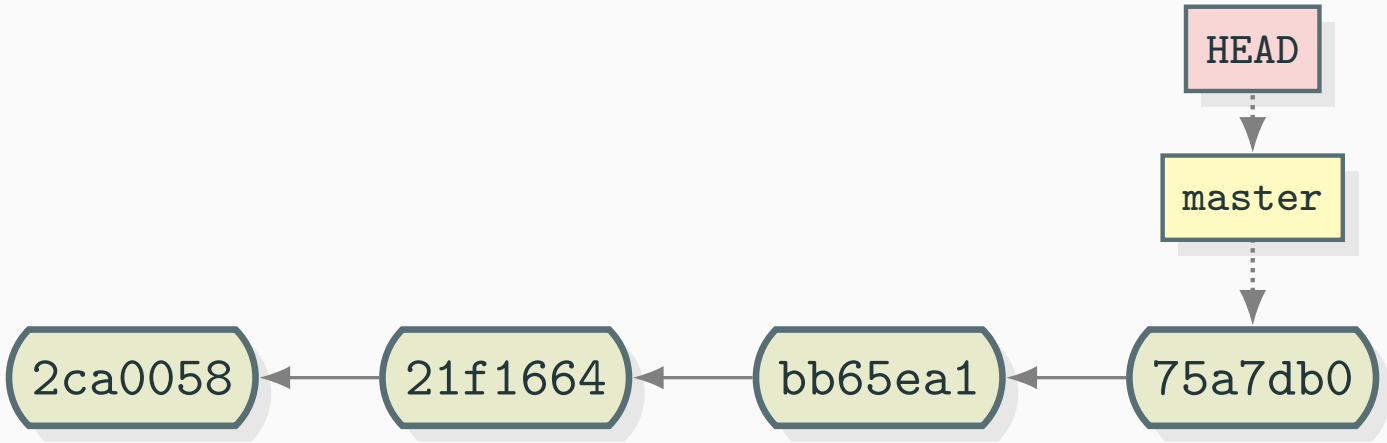


Abbildung 16: Branch currently on master

Which branch is it on now

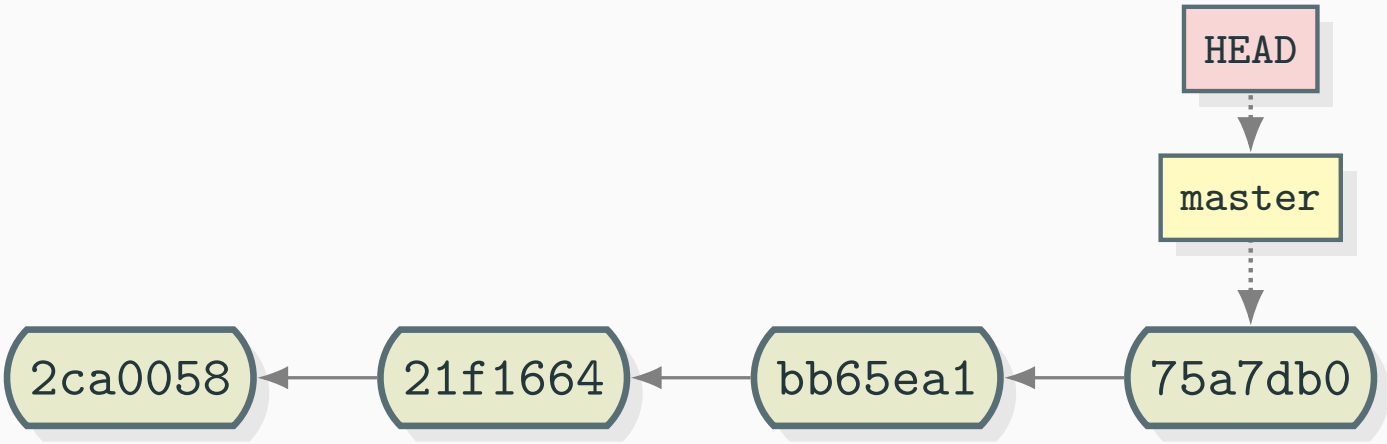


Abbildung 16: Branch currently on master

Look log.

```
$ git log --oneline --graph master
* 75a7db0 (HEAD -> master) Hello
* bb65ea1 New line for README
* 21f1664 Add README file
* 2ca0058 Initial commit
```

Which branch is it on now

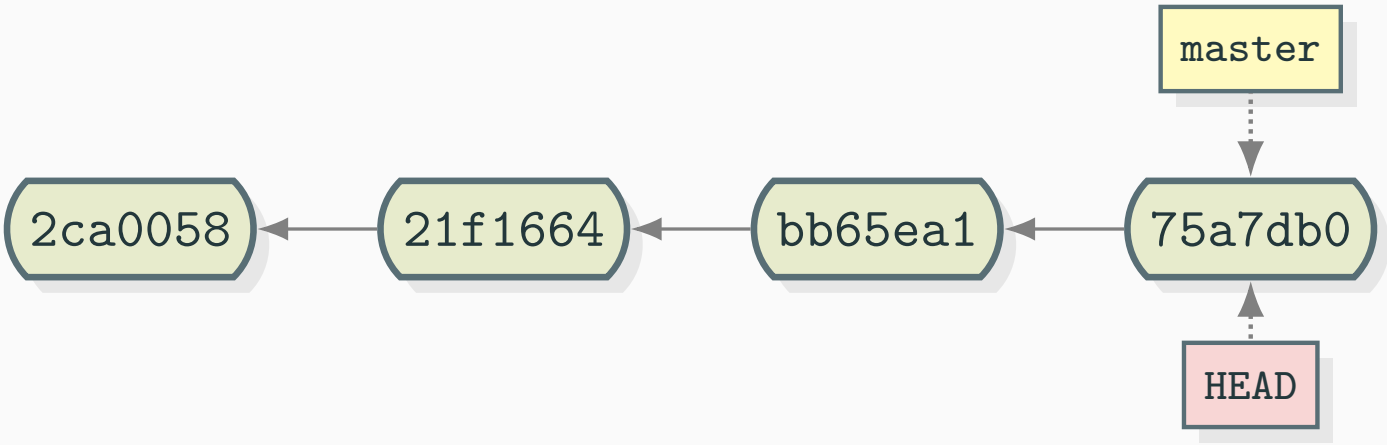


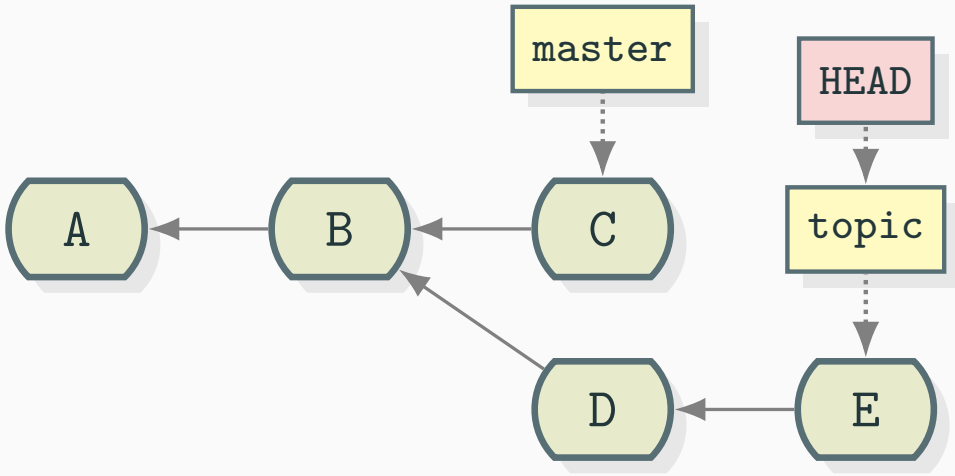
Abbildung 16: Branch is not currently on any

If output looks like this, branch is not currently on any.

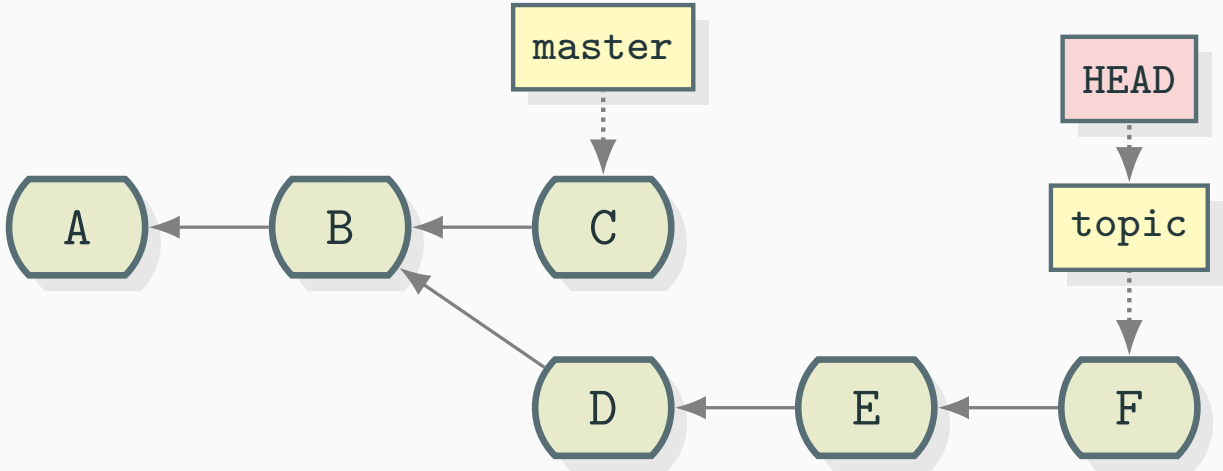
git log

```
* 75a7db0 (HEAD, master) Hello
* bb65ea1 New line for README
* 21f1664 Add README file
* 2ca0058 Initial commit
```

Commit on branch



(a) Before commit



(b) After commit

Abbildung 17: Demonstrating a typical `commit` on branch

Create a branch

Let's say you want to create a new branch called testing. You do this with the git branch command:

```
$ git branch topic 93c9e26
```

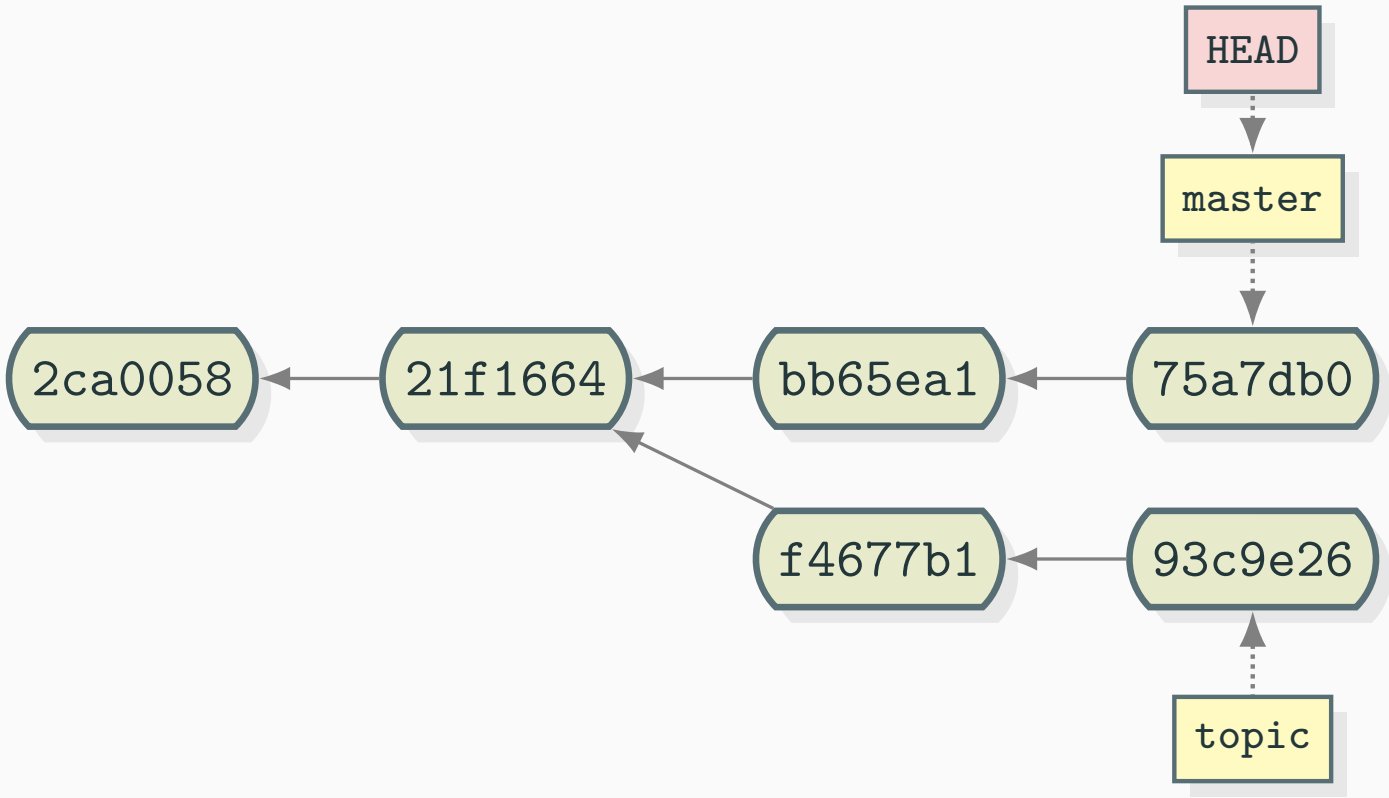
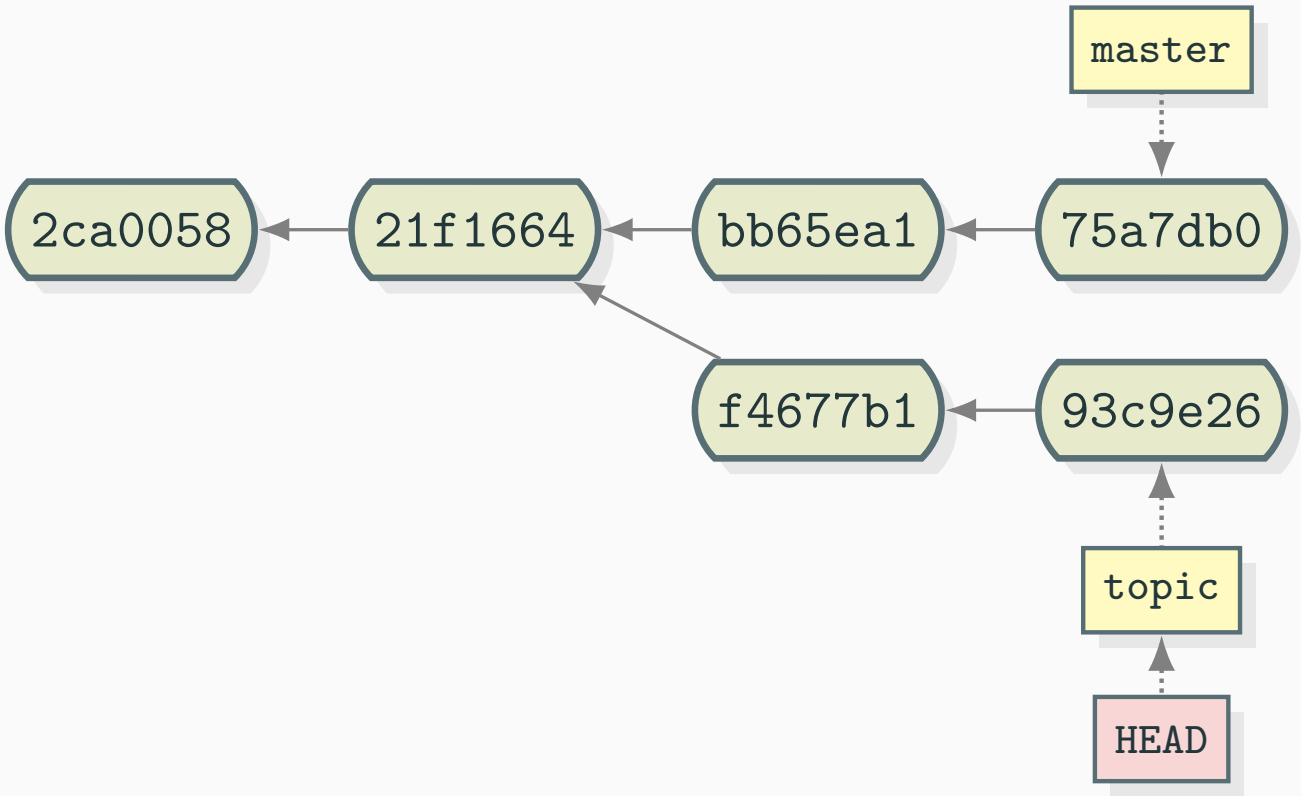


Abbildung 18: Two branches pointing into the same series of commits

Switch to branch

To switch to an existing branch, you run the `git switch` command. Let's switch to the new testing branch:

```
$ git switch topic
Switched to branch 'topic'
```

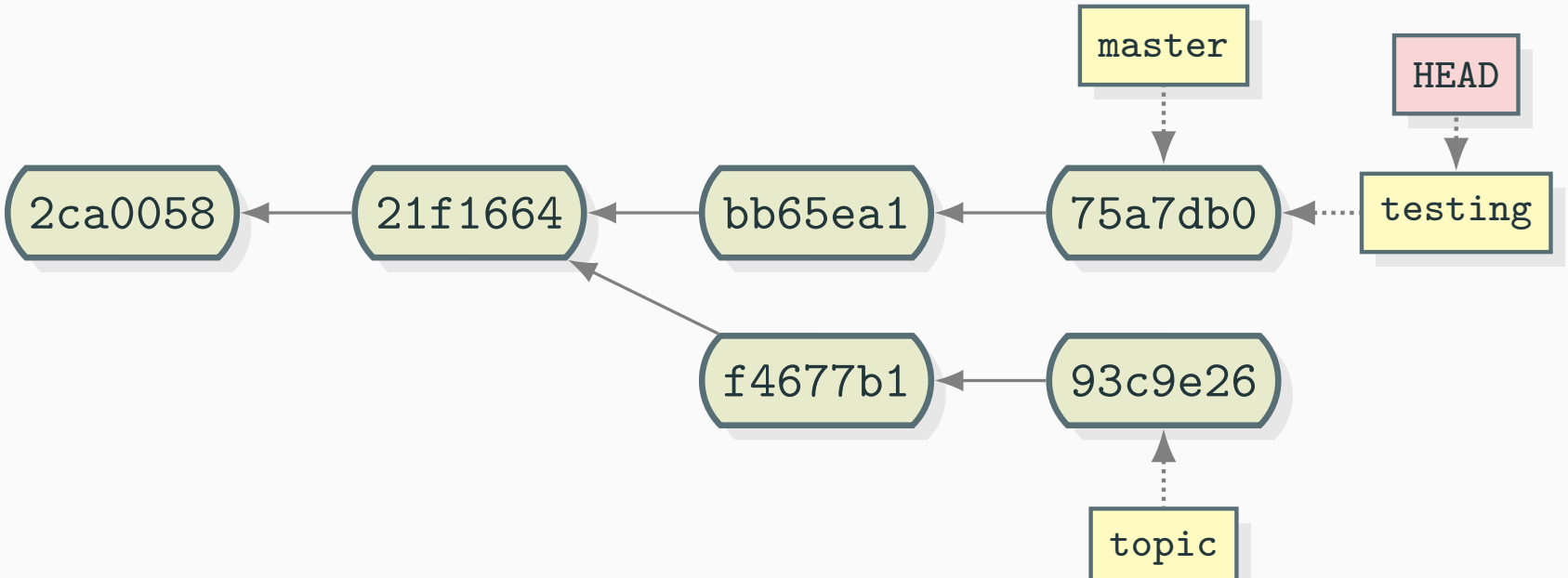


Create and switch branch

If you don’t want to type so many commands, there is a one-step way to do it

```
$ git switch -c testing master
Switched to a new branch 'testing'
```

It will be create a new branch and switch to it, and commit node doesn’t change.



Switch to branch using checkout

Like switching commit, `checkout` command can also switch branch.

```
git checkout topic
```

Create and switch in one-step way.

```
git checkout -b testing master
```

List branches

If `--list` is given, or if there are no non-option arguments, existing branches are listed; the current branch will be highlighted in green and marked with an asterisk. Any branches checked out in linked worktrees will be highlighted in cyan and marked with a plus sign.

```
$ git branch --list
  master
* testing
  topic
```

List branches

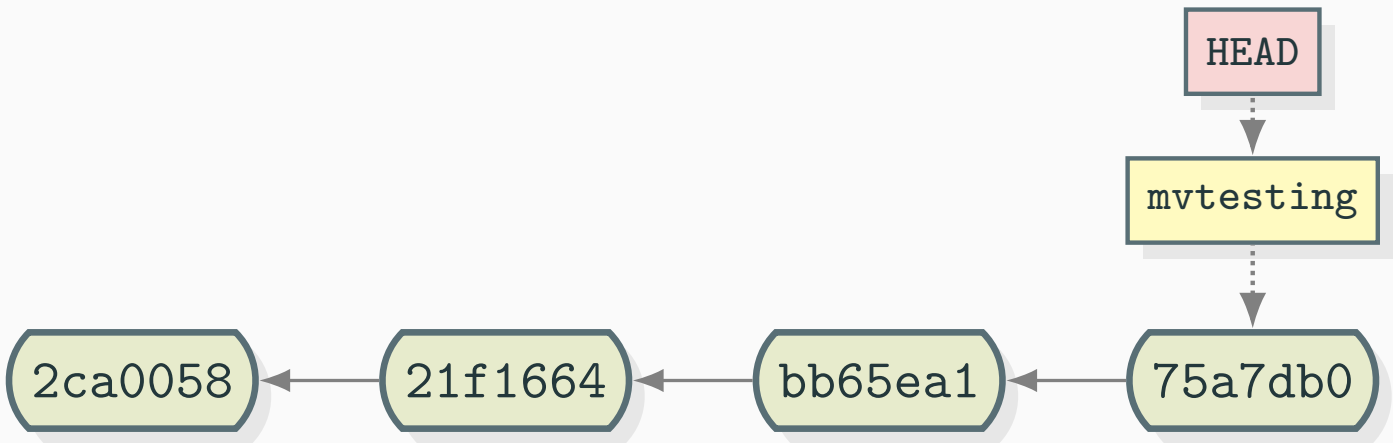
If `-v` / `--verbose` is given in list mode, show sha1 and commit subject line for each head, along with relationship to upstream branch.

```
$ git branch --verbose
  master  75a7db0 Hello
* testing 75a7db0 Hello
  topic   93c9e26 Revert "Commit on pred of HEAD"
```

Rename a branch

With a `-m` or `-M` option, *oldbranch* will be renamed to newbranch. If *oldbranch* had a corresponding reflog, it is renamed to match newbranch, and a reflog entry is created to remember the branch renaming. If newbranch exists, `-M` must be used to force the rename to happen.

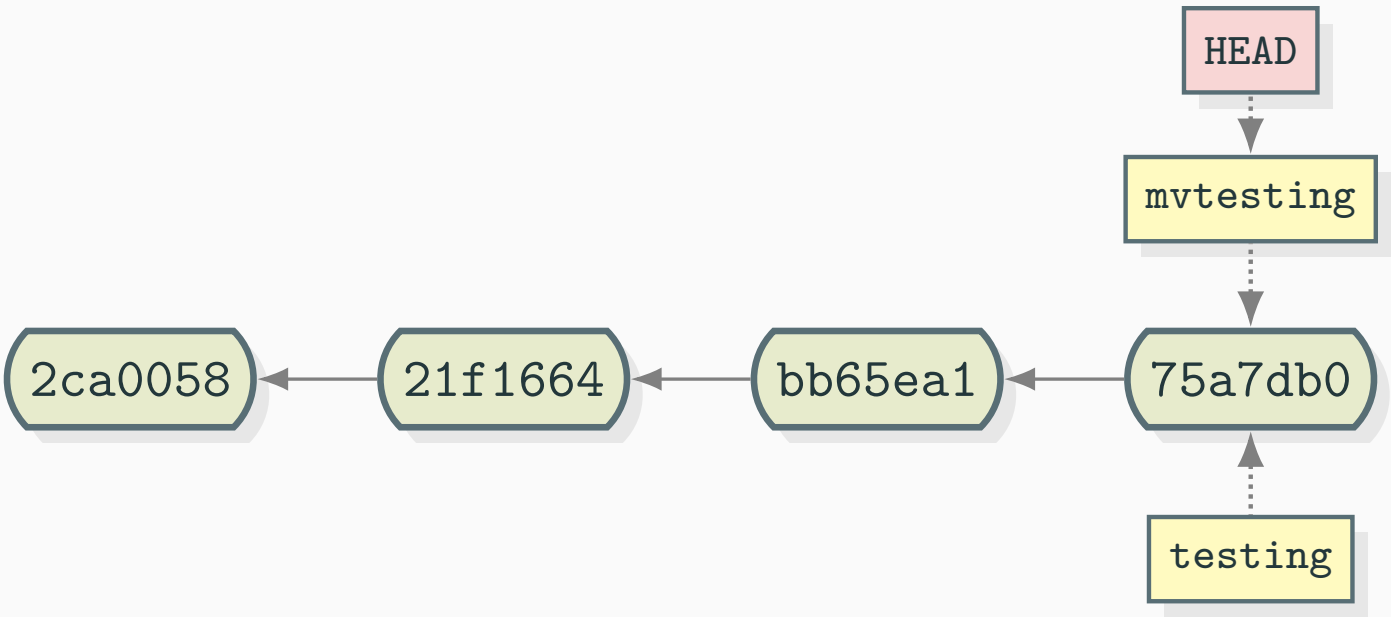
```
$ git branch -m testing mvtesting
```



Copy a branch

The `-c` and `-C` options have the exact same semantics as `-m` and `-M`, except instead of the branch being renamed, it will be copied to a new name, along with its config and reflog.

```
$ git branch -c mvtesting testing
```



Remove branches

With a `-d` or `-D` option, *branchname* will be deleted. You may specify more than one branch for deletion. If the branch currently has a reflog then the reflog will also be deleted.

```
$ git switch master
Switched to branch 'master'
$ git branch -d mvtesting testing
Deleted branch mvtesting (was 75a7db0).
Deleted branch testing (was 75a7db0).
```

Basic Merging

When your work is complete, and ready to be merged into main line. All you have to do is check out the branch you wish to merge into and then run the `merge` command:

```
$ git merge topic
Merge made by the 'ort' strategy.
```

Abbildung 19: A merge commit

Merged branches

The useful `--merged` and `--no-merged` options can filter this list to branches that you have or have not yet merged into the branch you're currently on.

```
$ git branch --merged
* master
  topic
```

Three-way merging

Because the commit on the branch you're on isn't a direct ancestor of the branch you're merging in, Git has to do some work. In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.

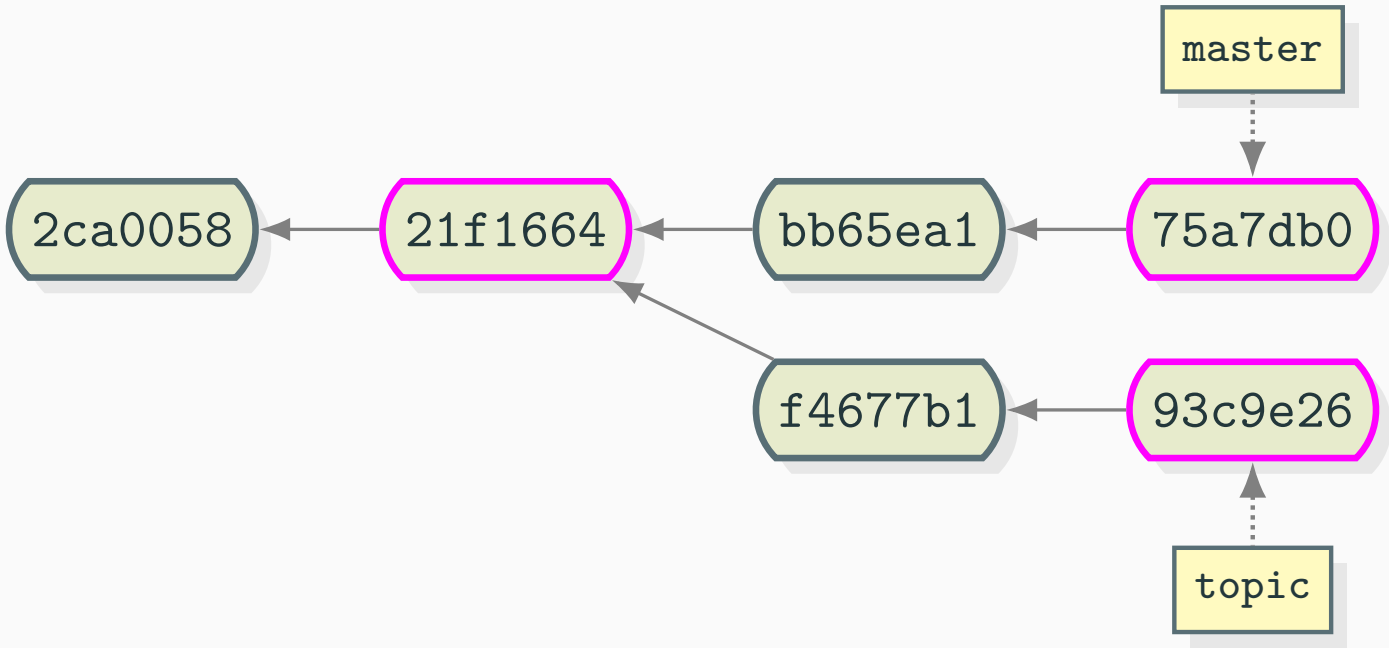
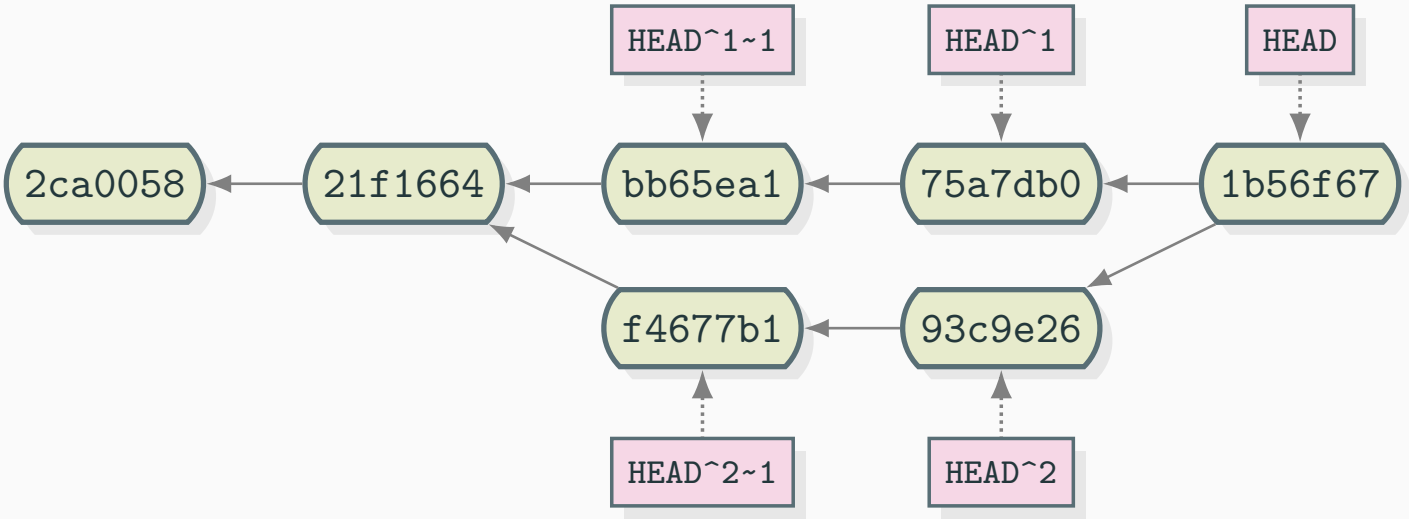


Abbildung 20: Three commits used in a typical merge

Parents of commit

For a commit with only one parent, rev~ and rev^ mean the same thing. ^ becomes useful with merge commits because each one is the child of two or more parents.

HEAD^ means the first immediate parent of the tip of the current branch. HEAD^ is short for HEAD^1, and you can also address HEAD^2 and so on as appropriate.



Merging vs. Rebasing

	Merge	Rebase
Behevior	merge branches	from one branch to another
History	complete	linear
MainLine	combined as a single commit	same number of commits
Target	shared branch	private branch

Tabelle 3: Comparison between `merge` and `rebase`

Merging vs. Rebasing

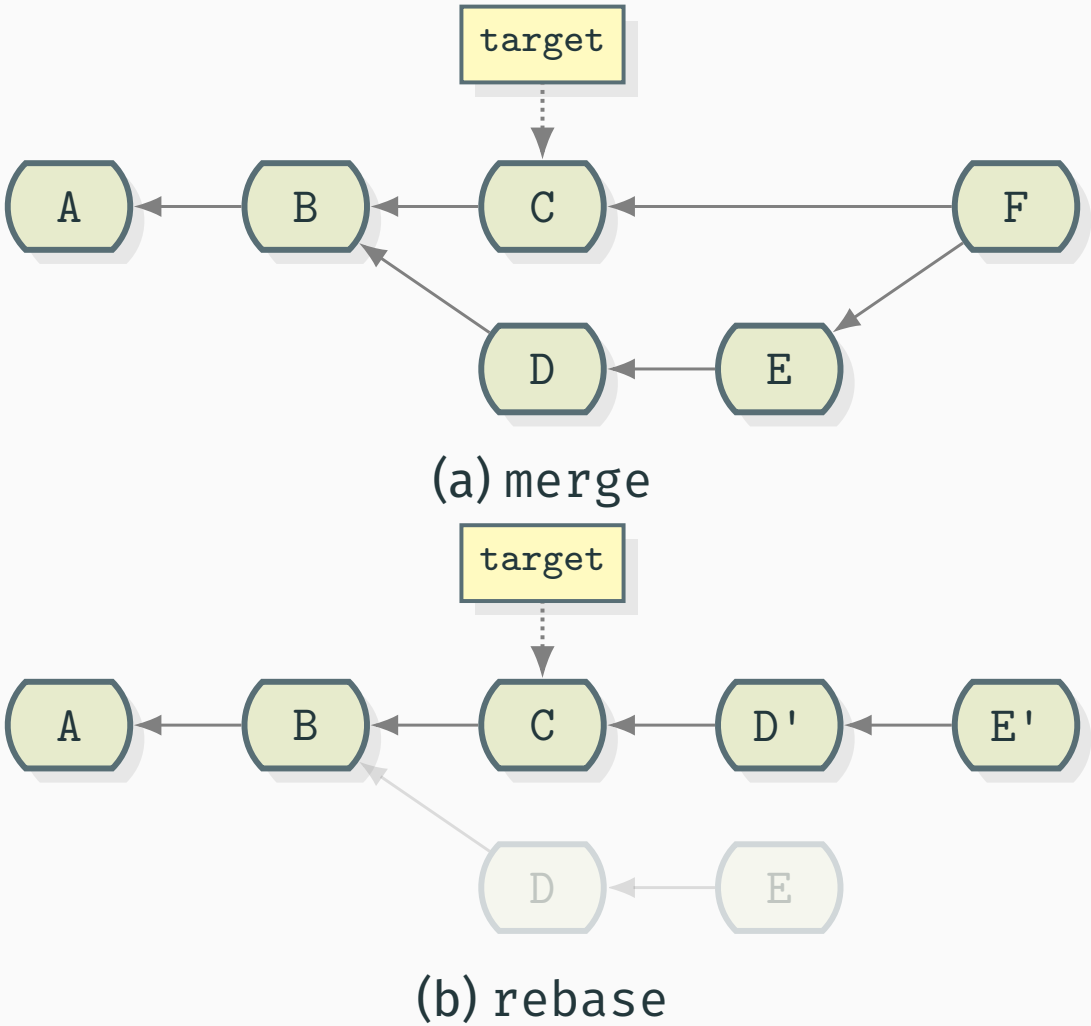


Abbildung 21: Comparison between `merge` and `rebase`

Working

How to presentation absolute path

The root path of current repository is called `:/`, or show the absolute path when use command `status`.

```
git status --porcelain
```

Move or rename a file

```
git mv <source> <destination>
```

Remove a file

Remove files matching pathspec from the index, or from the working tree and the index.

```
git rm <source>
```

If you except removing the file only from the Git repository, but not from the filesystem, add option `--cached`.

```
git rm --cached <source>
```

Stashing

when you've been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don't want to do a commit of half-done work just so you can get back to this point later.

The answer to this issue is the `stash` command.

Stashing Your Work

To demonstrate stashing, you'll go into your project and start working on a couple of files and possibly stage one of the changes. If you run git status, you can see your dirty state:

```
$ echo 'the Answer to the Ultimate Question of Life, the
↪  Universe, and Everything' > 42.txt
$ git add 42.txt
$ git status
On branch master

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   42.txt
```

Stashing Your Work

Now you want to switch branches, but you don't want to commit what you've been working on yet, so you'll stash the changes. To push a new stash onto your stack, run *git stash* or *git stash push*:

```
$ git stash push
Saved working directory and index state WIP on master:
↪  1b56f67 Merge branch 'topic'
```

You can now see that your working directory is clean:

```
$ git status
On branch master

nothing to commit, working tree clean
```

Release the stashed worktree

At this point, you can switch branches and do work elsewhere; your changes are stored on your stack. To see which stashes you've stored, you can use *git stash list*:

```
$ git stash list
stash@{0}: WIP on master: 1b56f67 Merge branch 'topic'
```

In this case, you have only access to one stashed worktree.

Release the stashed worktree

You can reapply the one you just stashed by using the command shown in the help output of the original stash command: *git stash pop*. You can also specify it by naming it, like this: *git stash pop stash@{0}*.

```
$ git stash pop
On branch master

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   42.txt

Dropped refs/stash@{0} (c84561e43f)
```

Fine-grained management of stashed worktree

You can run *git stash apply* and *git stash drop* to achieve the same effect as *git stash pop*.

```
git stash apply stash@{0}
git stash drop stash@{0}
```

Cleaning

You may not want to stash some work or files in your working directory, but simply get rid of them; that's what the `clean` command is for.

```
git clean -d
```

You'll want to be pretty careful with this command, since it's designed to remove files from your working directory that are not tracked.

See what clean would do

If you ever want to see what it would do, you can run the command with the `--dry-run` (or *-n*) option, which means **do a dry run and tell me what you would have removed**.

```
$ echo 'Mostly Harmless' > earth.txt
$ git clean -d --dry-run
Would remove earth.txt
```

A safer option

You'll want to be pretty careful with this command, since it's designed to remove files from your working directory that are not tracked. If you change your mind, there is often no retrieving the content of those files. A safer option is to run `git stash --all` to remove everything but save it in a stash.

Checking out

`checkout` also takes `--` to mean that subsequent arguments are not its optional “treeish” parameter specifying which commit you want.

```
# switch the working copy to the branch
git checkout <branchname>
# discard uncommitted changes to the file
git checkout -- <filename>
```

Project

Submodules

It often happens that while working on one project, you need to use another project from within it. Perhaps it's a library that a third party developed or that you're developing separately and using in multiple parent projects.

Git addresses this issue using `submodule`. Submodules allow you to keep a Git repository as a subdirectory of another Git repository.

Starting with Submodules

Let's start by adding an existing Git repository as a submodule of the repository that we're working on. To add a new submodule you use the option *add* with the absolute or relative URL of the project you would like to start tracking.

```
$ git submodule add
↪  https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

Starting with Submodules

Although DbConnector is a subdirectory in your working directory, Git sees it as a submodule and doesn't track its contents when you're not in that directory. Instead, Git sees it as a particular commit from that repository.

```
$ git commit -am 'Add DbConnector module'
[master fb9093c] Add DbConnector module
 2 files changed, 4 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 DbConnector
$ git push origin master
```

Cloning a Project with Submodules

Here we'll clone a project with a submodule in it. When you clone such a project, by default you get the directories that contain submodules, but none of the files within them yet:

```
$ git clone https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
$ ls -a MainProject
.git         DbConnector  includes  src
.gitmodules  Makefile     scripts
$ ls -a MainProject/DbConnector
$
```

Cloning a Project with Submodules

The DbConnector directory is there, but empty. You must run two commands: `git submodule init` to initialize your local configuration file, and `git submodule update` to fetch all the data from that project and check out the appropriate commit listed in your superproject:

```
$ git submodule init
Submodule 'DbConnector'
↪  (https://github.com/chaconinc/DbConnector) registered
↪  for path 'DbConnector'
$ git submodule update
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out
↪  'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Cloning a Project with Submodules

If you already cloned the project and forgot `--recurse-submodules`, you can combine the `git submodule init` and `git submodule update` steps by running `git submodule update --init`. To also initialize, fetch and checkout any nested submodules, you can use the foolproof `git submodule update --init --recursive`.

Subtrees

`subtree` command is the main alternative to `submodule` command. However, subtrees should not be confused with submodules. `subtree` command is a copy of a Git repository pulled into a parent one, `submodule` command is a pointer to a specific commit in another repository. Unlike submodules, subtrees do not need *.gitmodules* files or *gitlinks* in the repository.

Adding a subtree

Let's say you already have a git repository with at least one commit. You can add another repository into this respository like this:

1. Specify you want to add a subtree
2. Specify the prefix local directory into which you want to pull the subtree
3. Specify the remote repository URL of the subtree being pulled in
4. Specify the remote branch of the subtree being pulled in
5. Specify you want to squash all the remote repository's logs

```
git subtree add --prefix {path} {URL} {branch} --squash
```

Updating in new subtree commits

If you want to pull / push in any new commits to the subtree from the remote, issue the same command as above:

```
git subtree pull --prefix {path} {URL} {branch} --squash
git subtree push --prefix {path} {URL} {branch}
```

Subtrees vs. Submodules

	Subtree	Submodule
Behevior	merge into master repo	multiple repos in master
Cost	equivalent to subrepo	only .gitsubmodule
Clone	no more commands	`submodule update --init`
Pull	no more commands	more submodule commands
Push	complexly push into subrepo	easily in subrepo
User	no need to care subrepo	manual synchronization

Tabelle 3: Comparison between `subtree` and `submodule`

Collaborate

Showing Your Remotes

To see which remote servers you have configured, you can run the `remote` command. It lists the shortnames of each remote handle you've specified. If you've cloned your repository, you should at least see *origin* — that is the default name Git gives to the server you cloned from:

```
$ git clone --quiet https://github.com/schacon/ticgit
$ cd ticgit
$ git remote
origin
$ git remote -v
origin    https://github.com/schacon/ticgit (fetch)
origin    https://github.com/schacon/ticgit (push)
```

Adding Remote Repositories

We've mentioned and given some demonstrations of how the git clone command implicitly adds the origin remote for you. Here's how to add a new remote explicitly. To add a new remote Git repository as a shortname you can reference easily, run `git remote add <shortname> <url>`:

```
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin    https://github.com/schacon/ticgit (fetch)
origin    https://github.com/schacon/ticgit (push)
pb    https://github.com/paulboone/ticgit (fetch)
pb    https://github.com/paulboone/ticgit (push)
```

Inspecting a Remote

If you want to see more information about a particular remote, you can use the `git remote show <remote>` command. If you run this command with a particular shortname, such as origin, you get something like this:

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master                               tracked
    dev-branch                           tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Renaming Remotes

You can run `git remote rename` to change a remote's shortname. For instance, if you want to rename pb to paul, you can do so with git remote rename:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

Resetting Remotes

Run the command `git remote set-url` to change the url of remote if you need:

```
$ git remote set-url paul https://github.com/paul/ticgit
$ git remote -v
origin    https://github.com/schacon/ticgit (fetch)
origin    https://github.com/schacon/ticgit (push)
paul    https://github.com/paul/ticgit (fetch)
paul    https://github.com/paul/ticgit (push)
```

Removing Remotes

If you want to remove a remote for some reason — you've moved the server or are no longer using a particular mirror, or perhaps a contributor isn't contributing anymore — you can either use `git remote remove`:

```
$ git remote remove paul
$ git remote
origin
```

Introduction to remote branches

Remote-tracking branches are references to the state of remote branches. They're local references that you can't move; Git moves them for you whenever you do any network communication, to make sure they accurately represent the state of the remote repository. Think of them as bookmarks, to remind you where the branches in your remote repositories were the last time you connected to them.

Remote-tracking branch names take the form *<remote>/<branch>*.

List the remote branches

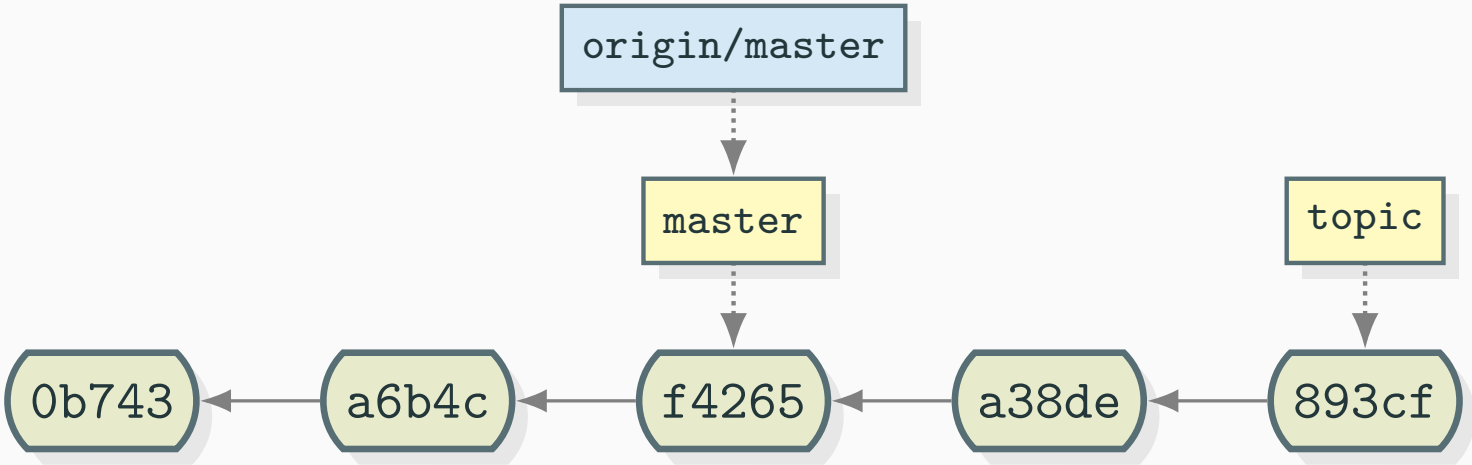


Abbildung 22: Server and local repositories

If the option `-r` / `--remote` is given to the *git branch* command, the remote-tracking branches will be listed.

```
$ git branch -r
  origin/master
```

Updates your remote-tracking branches

To synchronize your work with a given remote, you run a `git fetch <remote>` command. This command looks up which server “origin” is, fetches any data from it that you don’t yet have, and updates your local database, moving your origin/master pointer to its new, more up-to-date position.

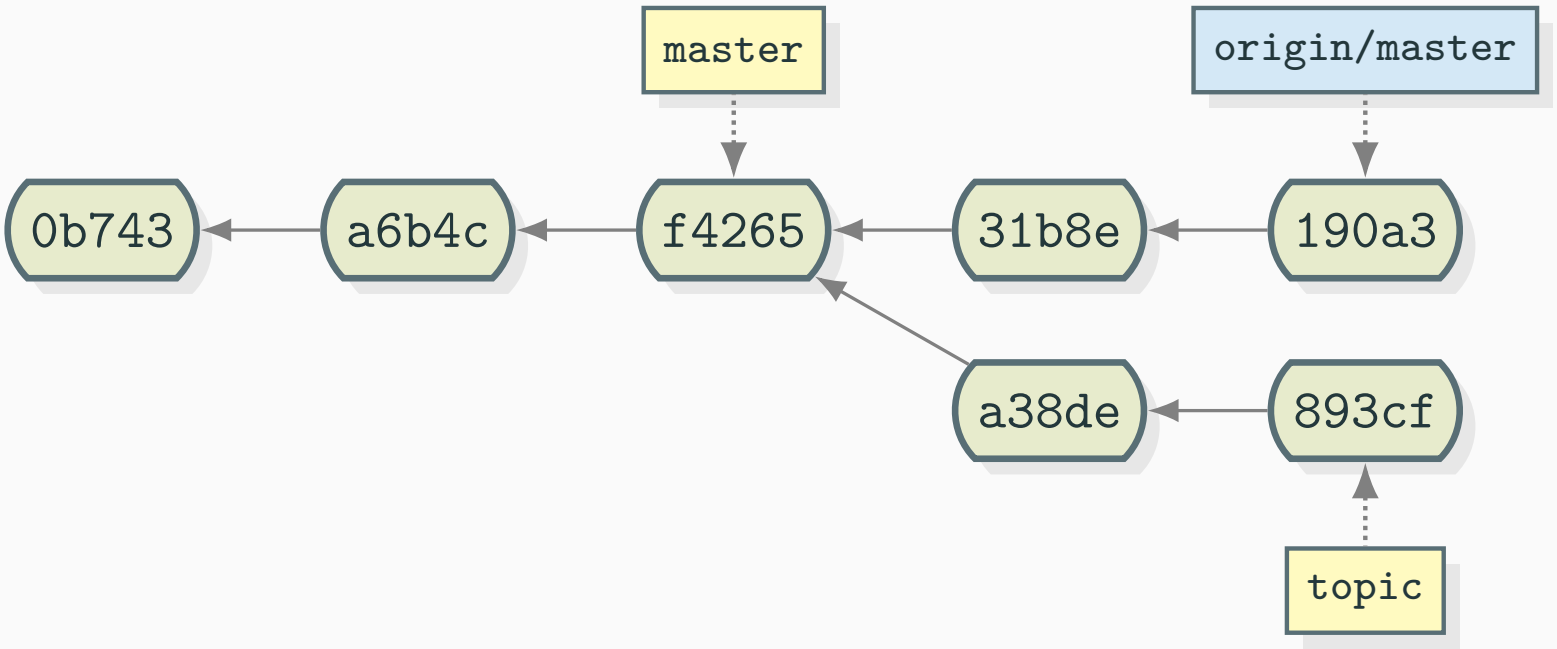


Abbildung 23: Updates your remote-tracking branches

Pushing to remote repository

If you have a branch named topic that you want to work on with others, you can push it up the same way you pushed your first branch.

Run:

```
git push <remote> <branch>
```

```
$ git push origin topic
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      topic -> topic
```

Pushing to remote repository

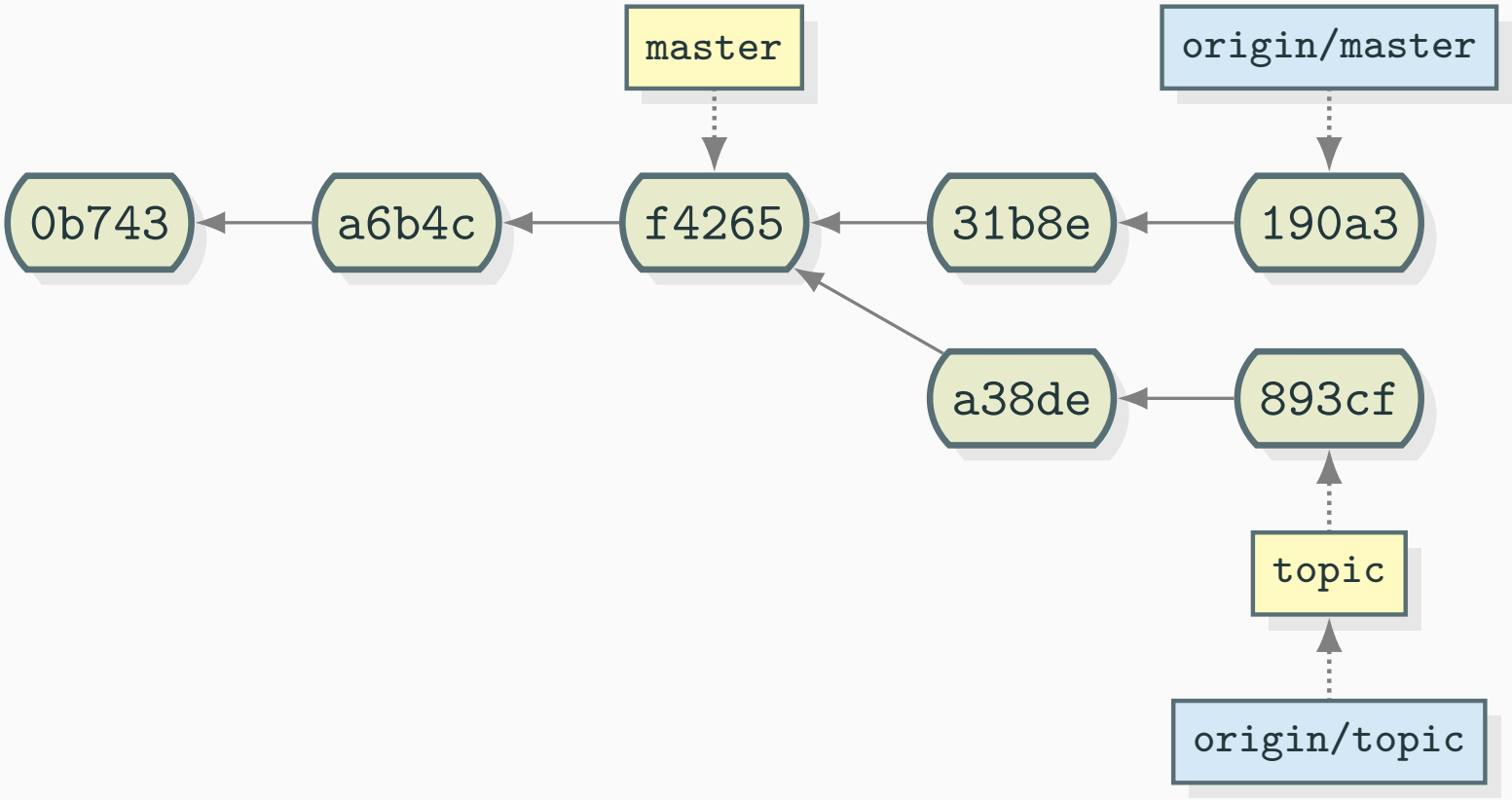
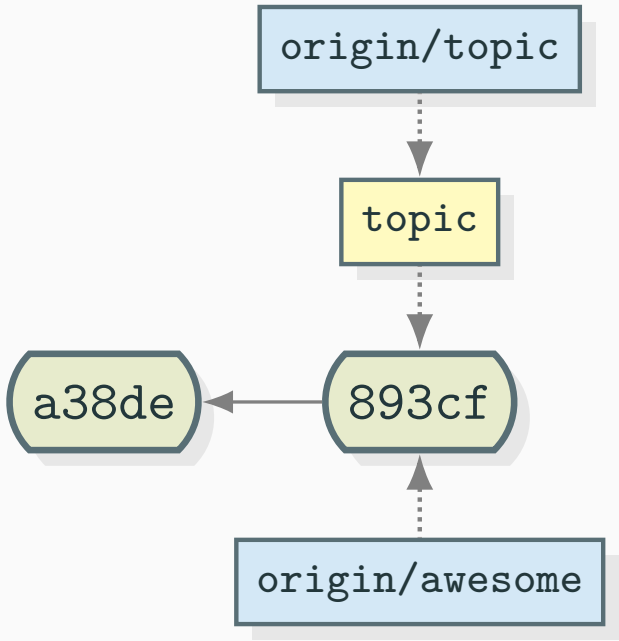


Abbildung 24: Pushing the topic branch

Pushing and renaming the local branch

Git automatically expands push command to topic:topic, which does the same thing — it says, “Take my topic and make it the remote’s topic.” You can use this format to push a local branch into a remote branch that is named differently. If you didn’t want it to be called topic on the remote, you could instead run `git push origin topic:awesome` to push your local *topic* branch to the *awesome* branch on the remote project.



Delete the remote branch

To completely remove a remote branch, you need to use the git push origin command with a `-d` / `--delete` option, then specify the name of the remote branch.

```
$ git push origin --delete awesome
To https://github.com/schacon/simplegit
 - [deleted]         awesome
```

Tracking Branches

Checking out a local branch from a remote-tracking branch automatically creates what is called a “tracking branch” (and the branch it tracks is called an “upstream branch”). If you’re on a tracking branch and type git pull, Git automatically knows which server to fetch from and which branch to merge in.

```
$ git switch -c tracker --track origin/master
Branch 'tracker' set up to track remote branch 'master'
↪  from 'origin'.
Switched to a new branch 'tracker'
```

Tracking Branches

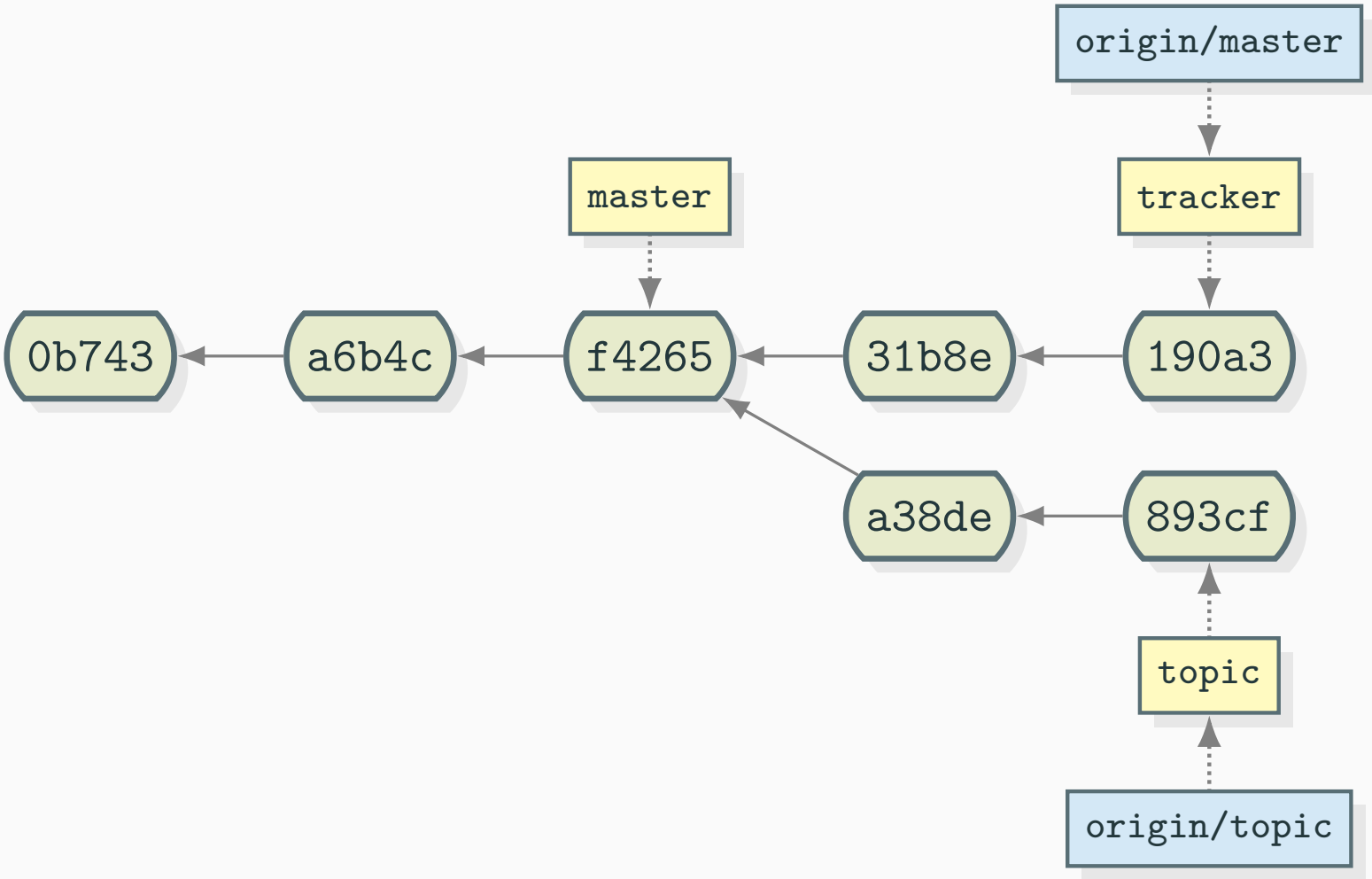


Abbildung 25: Tracking the remote branch

List local branches with upstream

If you want to see what tracking branches you have set up, you can use the *-vv* option to git branch. This will list out your local branches with more information including what each branch is tracking and if your local branch is ahead, behind or both.

```
$ git branch -vv
  master  f4265 [origin/master: behind 2] Deploy index fix
* tracker 190a3 [origin/master] This should do it
  topic   893cf [origin/topic] Add forgotten brackets
```

Fetching the target remote branches

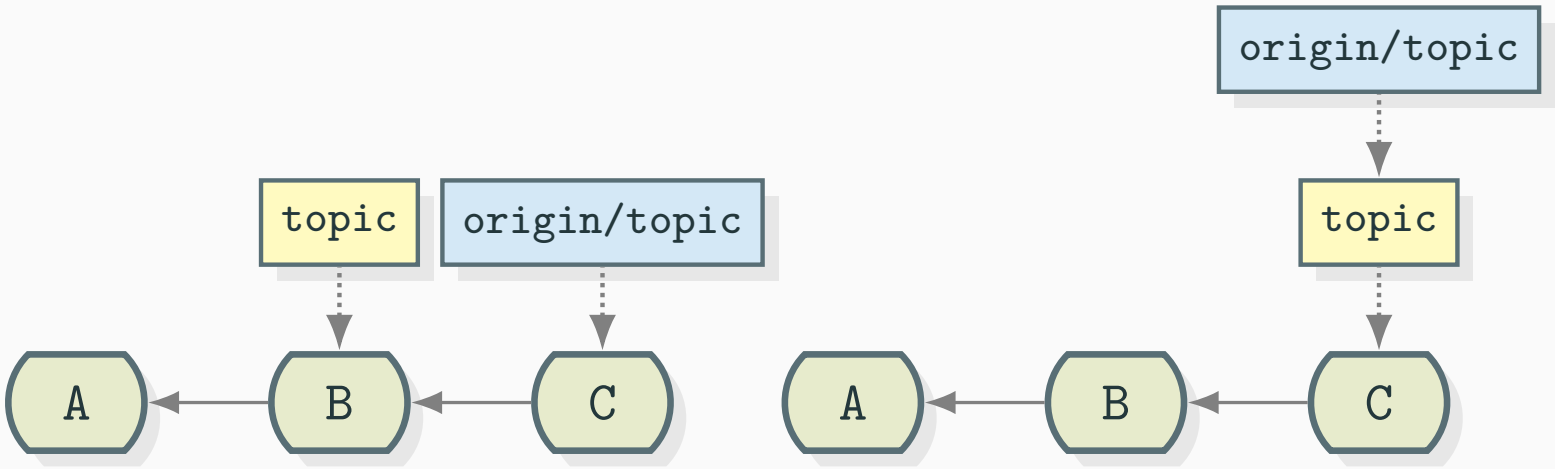
While the `git fetch` command will fetch all the changes on the server that you don't have yet, it will not modify your working directory at all. It will simply get the data for you and let you merge it yourself.

Using `git fetch <remote> <branch>`, the names of refs that are fetched, together with the object names they point at, are written to **FETCH_HEAD**.

Synchronizing the behind branch

If the current branch is behind the remote branch, you'll notice the phrase "fast-forward" in that merge, or pulling with *ff* option:

```
git merge --ff-only FETCH_HEAD
git pull --ff-only <remote> <branch>
```



(a) before: behind 1 **(b)** after

Abbildung 26: Synchronizing the behind branch

Synchronizing the ahind branch

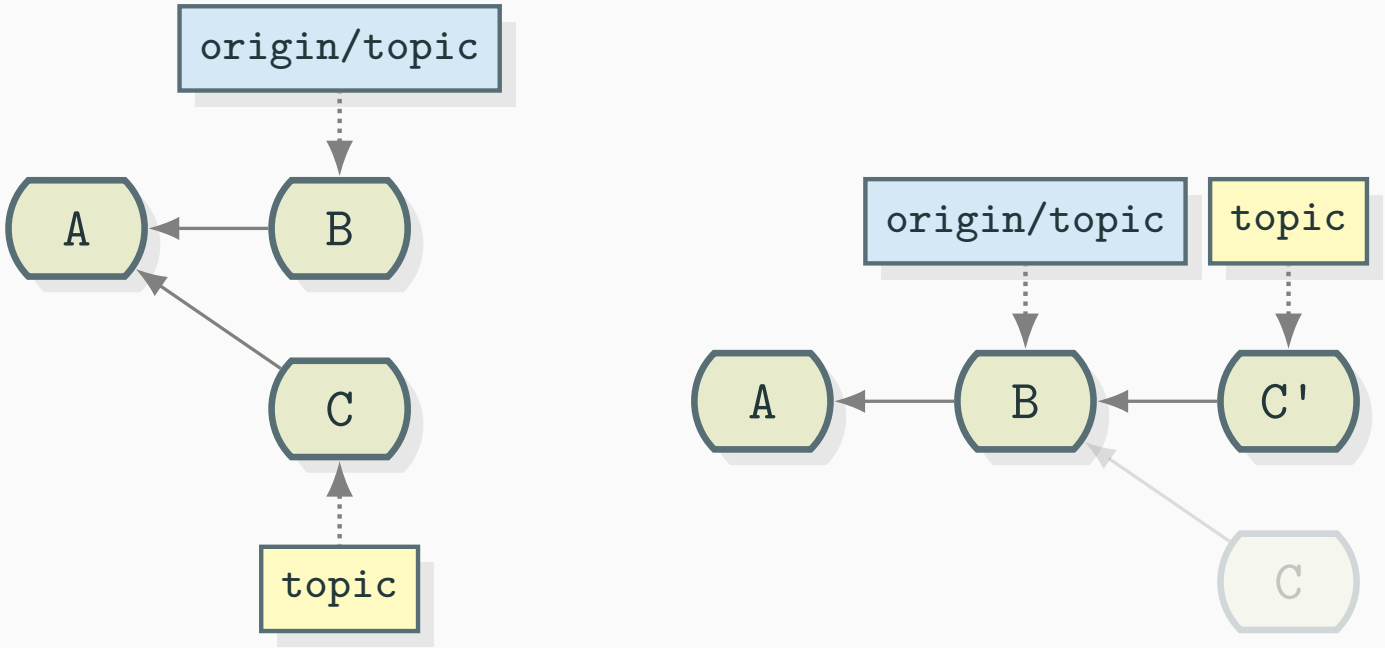
You commit locally, PUSHING! You don’t need to sync, others need to sync to you.

Synchronizing the behind and ahind branch

If others push to the branch, or use some amend commands, you will encounter this case. Try to *rebase* or *merge* command. I like the former.

Synchronizing the behind and ahind branch

```
git rebase FETCH_HEAD
git pull --rebase <remote> <branch>
```



(a) before: behind 1 and ahind 1

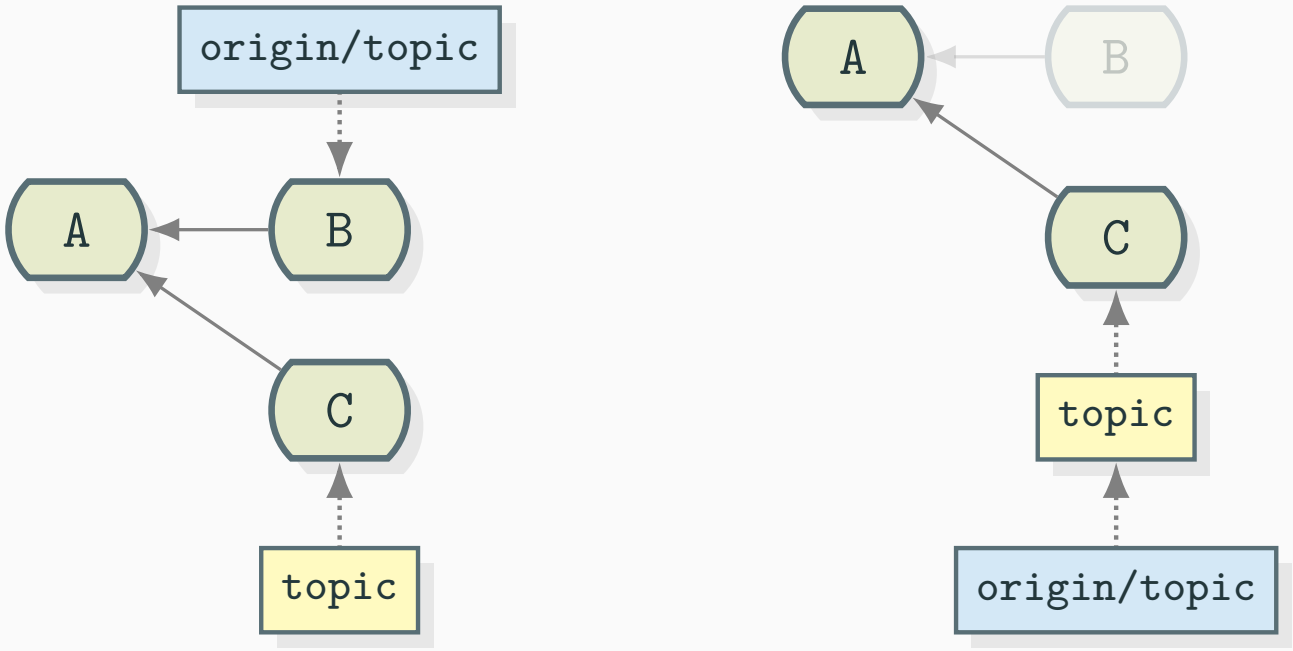
(b) after: ahind 1

Abbildung 27: Synchronizing the behind and branch with rebase

Synchronizing the behind and ahind branch

If you only need your local branch, pushing your branch with `-f` / `--force` option.

```
git push -f <remote> <branch>
```



(a) before: behind 1 and ahind 1

(b) after

Abbildung 27: Force pushing

Conclusion

LICENSE

Thanks

reviewer Rex Xu
Ruiling Song
Shaochi Zhou

engine LaTeX
theme metropolis

Enjoy Git.

Any Questions?