# Understanding Git

by @tednaleid

# you can't modify commits

# only add new ones

# commits are completely immutable and are *impossible* to accidentally destroy with git commands

though `rm -rf .git` will lose anything not yet pushed out

# uncommitted work is easily destroyed, so

# commit early & often

# garbage collection is the only truly

# destructive git action

garbage collection only destroys commits with *nothing* pointing at them

what points at commits?

other commits

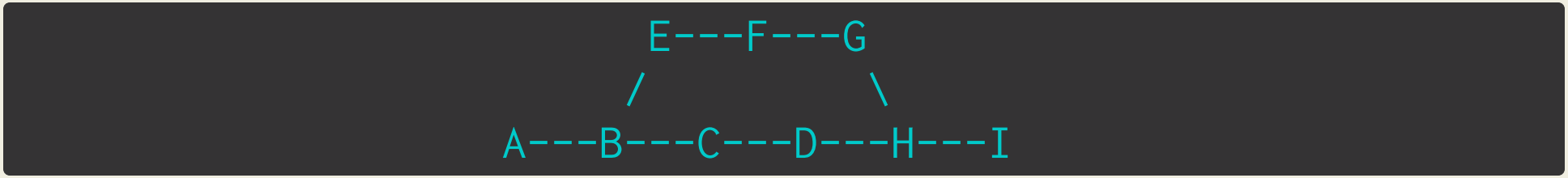
tags

branches

the reflog

# commit

point at 0..N parent commits




most commonly 1 or 2 parent commits

# tag

fixed commit pointers

```
A---B---C
        ↑
   release_1.0
```

```
% git commit -m "adding stuff to C"
```

```
A---B---C---D
        ↑
   release_1.0
```

# branch

floating commit pointer

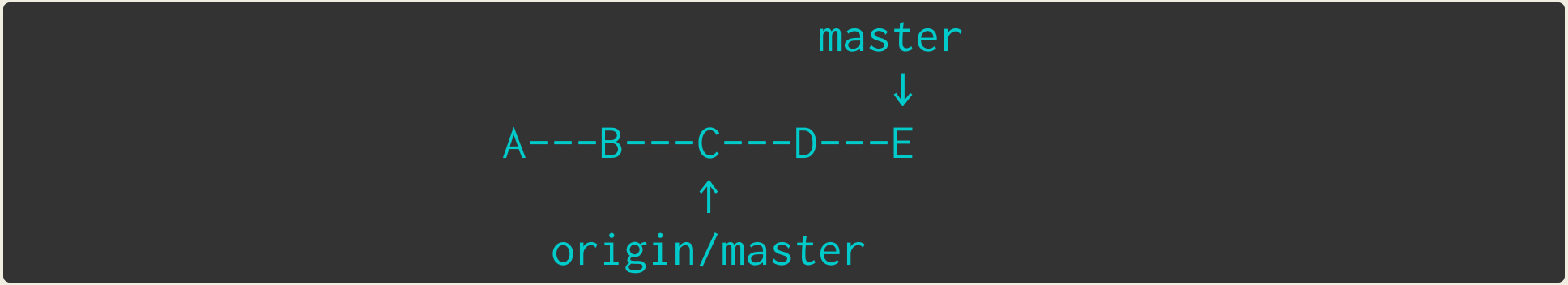
```
A---B---C
        ↑
      master
```

```
% git commit -m "adding stuff to B"
```

```
A---B---C---D
            ↑
          master
```

# remote branch

a “remote” branch is just a commit pointer in your local repo




it's updated whenever you do a `fetch` or `pull`, otherwise nothing

remote about them

# branch

text files in the `.git` directory

```
% ls -1 .git/refs/heads/**/*
.git/refs/heads/master
.git/refs/heads/my_feature_branch
```

```
% ls -1 .git/refs/remotes/**/*
.git/refs/remotes/origin/HEAD
.git/refs/remotes/origin/master
.git/refs/remotes/origin/my_feature_branch
```

# branch

contains is the SHA of the commit it's pointing at

```
% cat .git/refs/heads/master
0981e8c8ffbd3a1277dda1173fb6f5cbf4750d51

# .git/objects/09/81e8c8ffbd3a1277dda1173fb6f5cbf4750d51
```

# branches point at commits

Contain `tree` (filesystem), `parent` commits and commit metadata

```
% git cat-file -p 0981e8c8ffbd3a1277dda1173fb6f5cbf4750d51
tree 4fd7894316b4659ef3f53426166697858d51a291
parent e324971ecf1e0f626d4ba8b0adfc22465091c100
parent d33700dde6d38b051ba240ee97d685afdaf07515
author Ted Naleid <contact@naleid.com> 1328567163 -0800
committer Ted Naleid <contact@naleid.com> 1328567163 -0800

merge commit of two branches
```

The ID is the SHA of the commit's contents

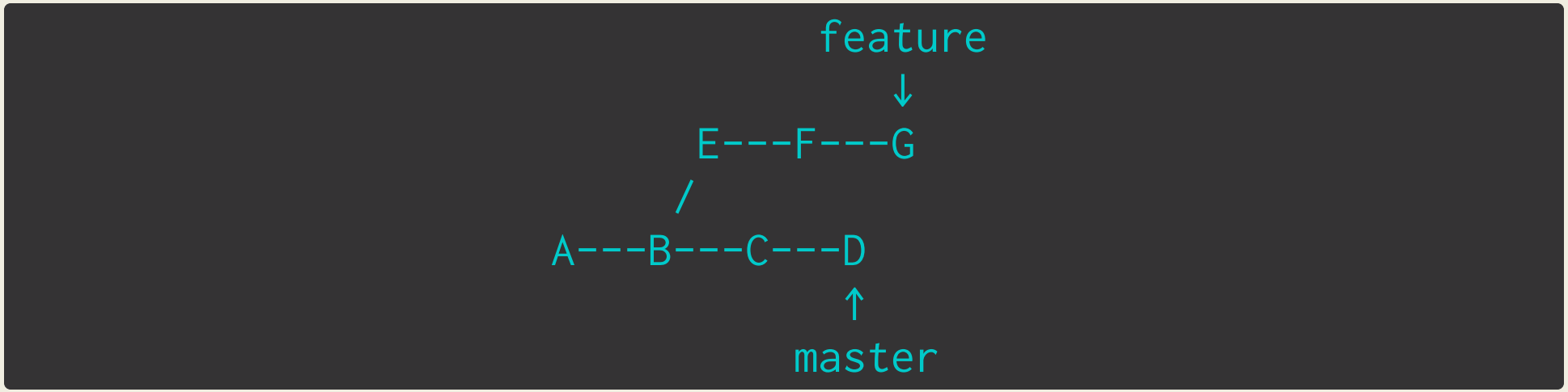
# branches

commits don't “belong to” branches, there's nothing in the commit

metadata about branches

# branches

a branch's commits are implied by the ancestry of the commit the branch points at




`master` is `A-B-C-D` and `feature` is `A-B-E-F-G`

# HEAD

`HEAD` is the current branch/commit

This will be the parent of the next commit

```
% cat .git/HEAD
ref: refs/heads/master
```

most of the time it points to a branch, but can point directly to a SHA when

“detached”

# the reflog

a log of recent `HEAD` movement

```
% git reflog
d72efc4 HEAD@{0}: commit: adding bar.txt
6435f38 HEAD@{1}: commit (initial): adding foo.txt
```

```
% git commit -m "adding baz.txt"
```

```
% git reflog
b5416cb HEAD@{0}: commit: adding baz.txt
d72efc4 HEAD@{1}: commit: adding bar.txt
6435f38 HEAD@{2}: commit (initial): adding foo.txt
```

by default it keeps at least 30 days of history

# the reflog

unique to a repository instance

# the reflog

can be scoped to a particular branch

```
% git reflog my_branch
347f5fe my_branch@{0}: merge master: Merge made by the recurs…
4e6007e my_branch@{1}: merge origin/my_branch: Fast-forward
32834d8 my_branch@{2}: commit (amend): upgrade redis version
2720e40 my_branch@{3}: commit: upgrade redis version
```

# dangling commit

if the only thing pointing to a commit is the reflog, it's “dangling”

# dangling commit

```
A---B---C---D---E---F
                    ↑
                  master
```

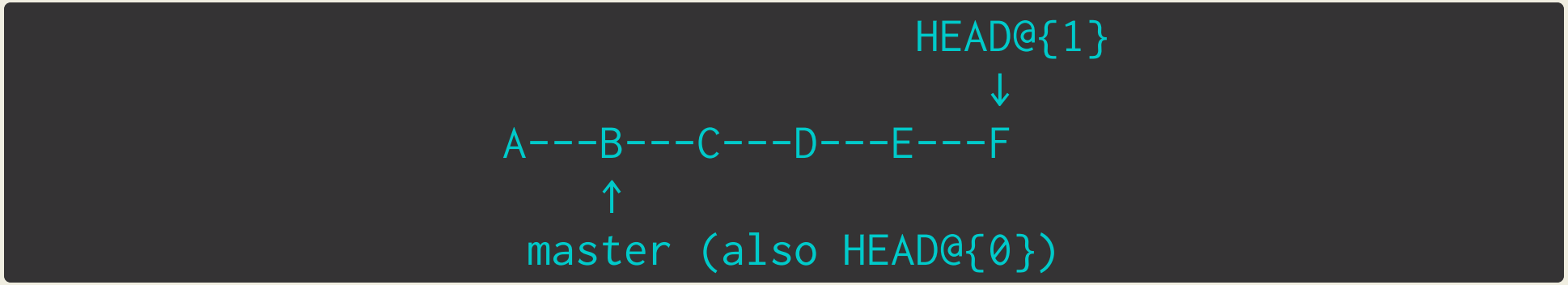
```
% git reset --hard SHA_OF_B
```

```
A---B---C---D---E---F
    ↑
  master
```

`C..F` are now dangling

# dangling commit

but they will be safe for ~30 days because of the reflog




`HEAD@{1}` will become `HEAD@{2}`..`HEAD@{N}` as refs are added to the reflog

# garbage collection

once a dangling commit leaves the reflog, it is “loose” and is at risk of garbage collection

# garbage collection

git does a `gc` when the number of “loose” objects hits a threshold

something like every 1000 commits

# garbage collection

to prevent garbage collecting a commit, just point something at it

```
% git tag mytag SHA_OF_DANGLING_COMMIT
```

# the index

a pre-commit staging area

`add -A :/` puts all changes in the index ready for commit

some bypass the index with `git commit -a -m "msg"`

# you should have courage to experiment

you have *weeks* to retrieve prior commits if something doesn't work

# understand where you are

before you try to go somewhere else

# You need (at least) one repo visualization tool

# that you grok

# Here's Mine:

```
~/.gitconfig:
[alias]
l = log --graph --pretty='%Cred%h%Creset -%C(yellow)%d%Creset %s %Cblue[%an]%Creset %Cgreen(%cr)%Creset' --abbrev-commit --date=relative
la = !git l --all
```

```
git la
```

# There are others - Git Tower

# There are others - SourceTree

# Learn “the good parts”

# and make them your own

# checkout -

just like `cd -`, takes you to your previous branch

```
                  E---F  ← feature & HEAD
                 /
A---B---C---D
            ↑
          master
```

```
% git checkout -
```

```
                  E---F  ← feature
                 /
A---B---C---D
            ↑
       master & HEAD
```

# commit --amend

redo the last commit

```
A---B---C
        ↑
  master & HEAD
```

```
<... change some files ... >
% git commit -a --amend --no-edit
```

```
          C' ← master & HEAD
         /
A---B---C
        ↑
(dangling but still in reflog)
```

# rebasing

reapplies a series of commits to a new parent commit

then moves the current branch pointer

# rebasing

```
          E---F  ← feature & HEAD
         /
A---B---C---D
            ↑
          master
```

```
% git rebase master
```

```
          (dangling but still in reflog)
                    ↓
          E---F
         /
A---B---C---D---E'--F'
            ↑       ↑
          master  feature & HEAD
```

# rebasing

```
% git rebase --abort
```

If you get in trouble `--abort` and try again.

If you *really* get in trouble, you can `reset --hard` back to your last commit.

# rebasing - a private activity

should never be done with commits that have been pushed

# rebasing - a private activity

public rebasing is bad as others could have the same commits with different SHAs

# cherry picking

apply a subset of changes from another branch

```
          E---F---G
         /
A---B---C---D
            ↑
          master & HEAD
```

```
% git cherry-pick SHA_OF_F
```

```
          E---F---G
         /
A---B---C---D---F'
                ↑
              master & HEAD
```

`reset` is for moving branch pointers

# reset --soft

```
A---B---C---D---E
                ↑
              master
```

```
% git reset --soft SHA_OF_C
```

```
working dir & index still look like
                ↓
A---B---C---D---E
        ↑
      master
```

1. moves `HEAD` & the current branch to the specified `<SHA>`
2. index - unchanged
3. working directory - unchanged

# reset --soft

useful for squashing the last few messy commits into one pristine commit

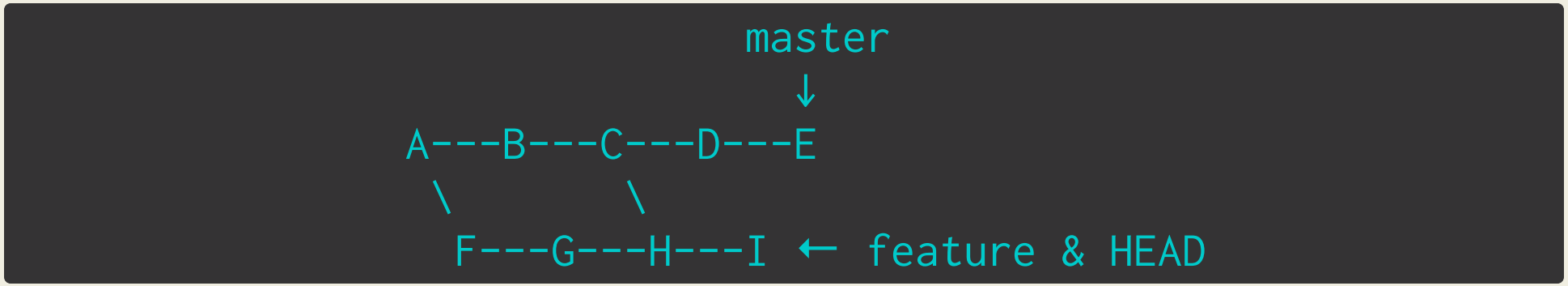
```
                working dir & index still look like
                                  ↓
               A---B---C---D---E
                       ↑
                    master
```

```
% git commit -m "perfect code on the 'first' try"
```

```
               A---B---C---E'
                           ↑
                        master
```

# reset --soft

What if you've got a more complicated situation:

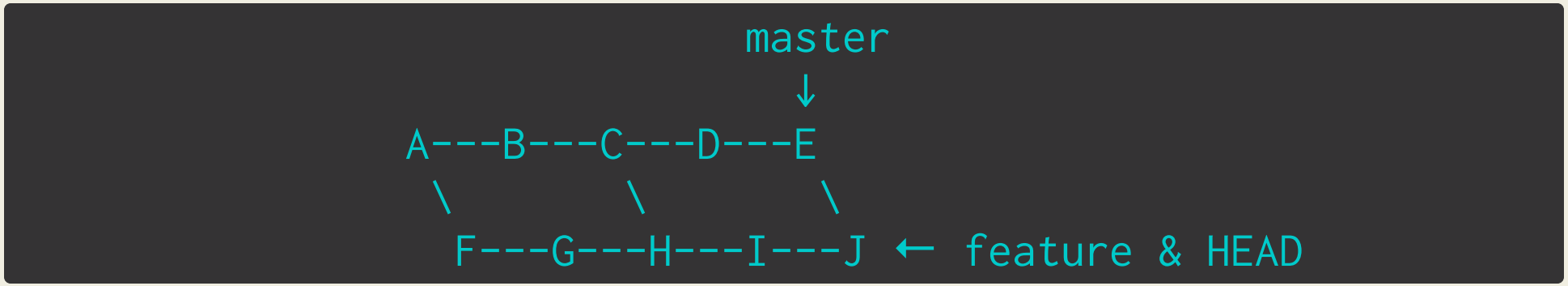



Can't `reset` our way out of this, right?

# reset --soft

Just do one last merge

```
% git merge master
```



# reset --soft

and then we can `reset` into a single commit

```
% git reset --soft master
```

```
                A---B---C---D---E ← feature & HEAD & master
                                \
                                 J ← working dir & index
```

```
% git commit -m "pristine J"
```

```
                              master
                                ↓
                A---B---C---D---E---J' ← feature & HEAD
```

# reset --hard

```
% git reset --hard <SHA>
```

1. moves `HEAD` & the current branch to the specified `<SHA>`
2. clean the index, make it look like `<SHA>`
3. clean the working copy, make it look like `<SHA>`

dangerous if you have uncommitted work, useful for undoing bad commits

# reset --hard HEAD

```
% git reset --hard HEAD
```

just means clean out the working directory and any staged information,

don't move the branch pointer

for more info on `reset`, see: http://progit.org/2011/07/11/reset.html

# fetch

download new commits and update the remote branch pointer

does not move any local branches

# fetch

```
                    origin/master
(local)                   ↓
                  A---B---C---D ← master & HEAD
```

```
                  A---B---E---F
(origin)                      ↑
                            master (in remote repo)
```

```
% git fetch
```

```
                            origin/master
                                  ↓
                              E---F
(local)                      /
                  A---B---C---D ← master & HEAD
```

# pull

`pull` is `fetch` plus `merge`

# pull

```
                    origin/master
                          ↓
(local)             A---B---C---D ← master & HEAD
```

```
                    A---B---E---F
(origin)                        ↑
                              master (local ref in remote repo)
```

```
% git pull
```

```
                          origin/master
                                ↓
                          E---F----
                         /         \
(local)             A---B---C---D---G ← master & HEAD
```

# the “right” way to pull down changes from the server

1. `stash` any uncommitted changes (if any)
2. `fetch` the latest refs and commits from origin
3. `rebase -p` your changes (if any) onto origin's head

   else, just fast-forward your head to match origin's
4. un-`stash` any previously stashed changes

`fetch` + `rebase` avoids unnecessary commits

# rebasing pull

As of git 1.8.5, git has finally added a rebase switch to `pull`:

```
% git pull --rebase
```

This will do the `fetch` + `rebase` for you (you still stash on your own).

# git is dangerous

myth #1

# git is the *safest* version control

reality

# git lets you rewrite history

myth #2

# rewriting history is a *lie*

reality

# git syntax is terrible

myth #3

# git syntax is *really terrible*

reality

# git mislabels things

ex: git branches aren't what you think they are

# throw away your preconceptions

# from other version control systems

# Questions?

# Bonus Section!

# reset (default)

```
% git reset [--mixed] <SHA>
```

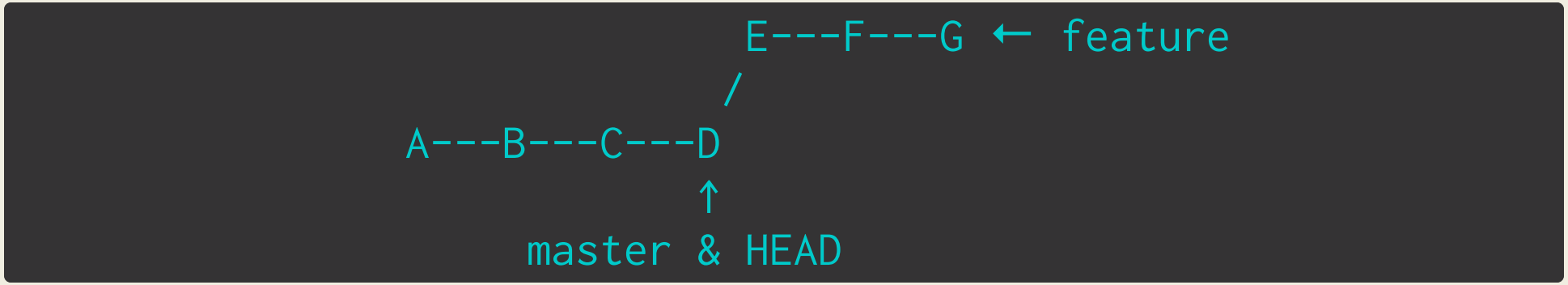
1. moves `HEAD` & the current branch to the specified `<SHA>`
2. clean the index, make it look like `<SHA>`
3. working directory - unchanged

`git reset HEAD` will unstage everything in the index

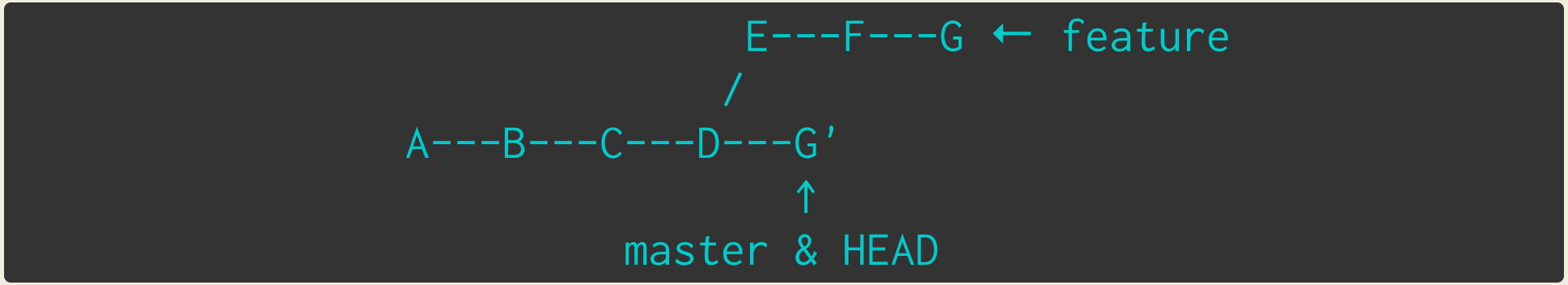
# squashing

compresses N commits into one commit that's appended to a destination branch

# squashing




```
% git merge --squash feature
```




cleans up history, when the thinking behind `E..F` is unimportant

# recovering commits

Oops, I really wanted `C`!

```
                          C' ← master & HEAD
                         /
                A---B---C ← (dangling)
```

```
% git reflog master  # find SHA_OF_C
% git reset --hard SHA_OF_C
```

```
                          C' ← (dangling)
                         /
                A---B---C
                        ↑
                    master & HEAD
```