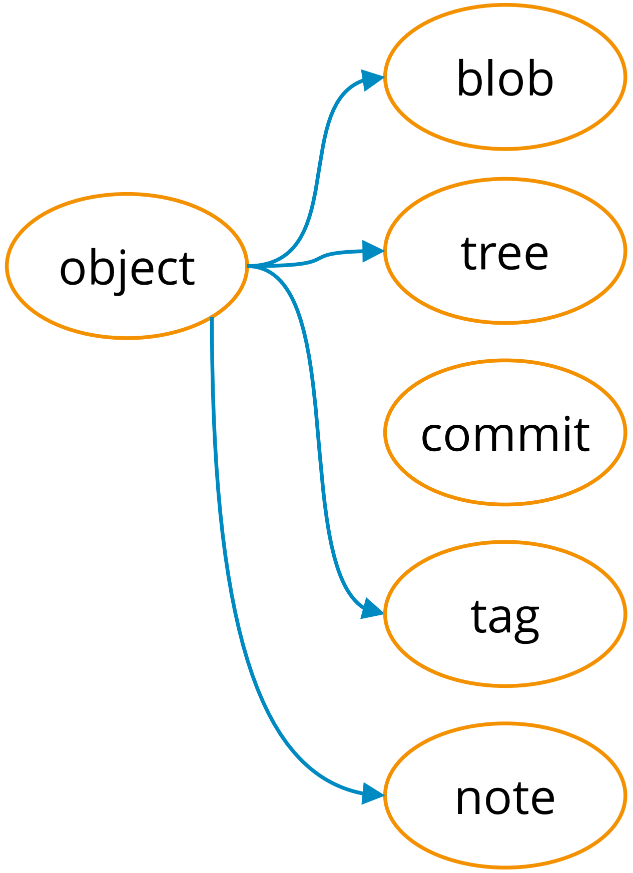
Git

Design and advanced usage

Objects



File contents (with no name/path or metadata)
Name and mode are set in parent tree.

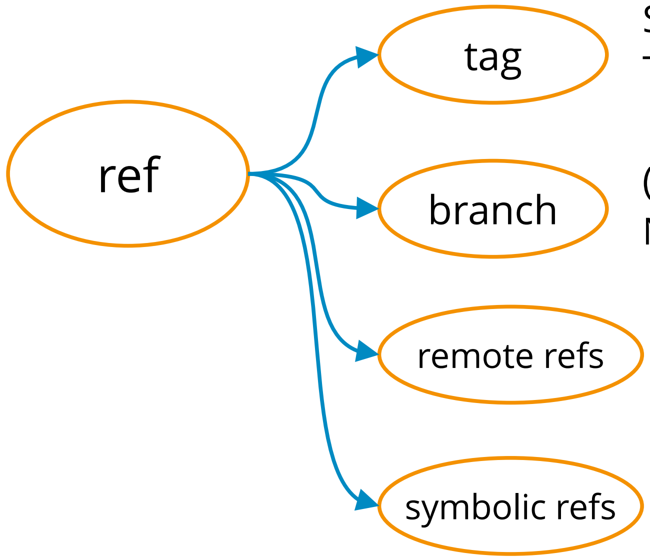
Directory contents (with no name/path for itself)
Recursive (can contain other tree)
Name is set in parent tree.

Unit of work. Conains tree id, author, committer, timestamps

Static ref to a commit (not auto-updated)

https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

References



Static ref to a commit (not auto-updated)
True objects

(aka head) Dynamic ref to a commit (auto-updated)
Not true objects

Fetched remote branches: refs/remotes/*RemoteName*/*Branch*
Not true objects

HEAD
refs/remotes/*RemoteName*/**HEAD**

References are not objects

Tags

Q: What are tags? Objects or references?

A: Yes!

There are tag refs and tag objects.

Lightweight tags

$$\begin{tikzcd} \text{tag ref} \arrow[r, "\text{points}"] & \text{commit} & \text{(or any object)} \end{tikzcd}$$

Annotated tags

$$\begin{tikzcd} \text{tag ref} \arrow[r, "\text{points}"] & \text{tag object} \arrow[r, "\text{points}"] & \text{commit} & \text{(or any object)} \end{tikzcd}$$

Tags

From `man git tag`

If one of **-a**, **-s**, or **-u <key-id>** is passed, the command creates a **tag object**, and requires a tag message. Unless **-m** <msg> or **-F** <file> is given, an editor is started for the user to type in the tag message.

If -m <msg> or -F <file> is given and -a, -s, and -u <key-id> are absent, -a is implied.

Otherwise, a tag reference that **points directly** at the given object (i.e., a lightweight tag) is created.

Tags

Example of an annotated tag:

```
$ cat .git/refs/tags/3.2.3
9cb963df94e0701b31f40af7f7258d538ec42cb7

$ git cat-file -p 9cb963df94e0701b31f40af7f7258d538ec42cb7
object 7101592899ca6674f76489a9ccfe115f5c8a93df
type commit
tag 3.2.3
tagger Saeed Rasooli <saeed.gnu@gmail.com> 1721447216 +0330

version 3.2.3

$ git cat-file -p 3.2.3
  (same output as above command)
```

Object Database

Git includes a key-value data storage for objects

Key: hash of contents (SHA1)

```
git hash-object -w test.txt
```

Value: contents (immutable)

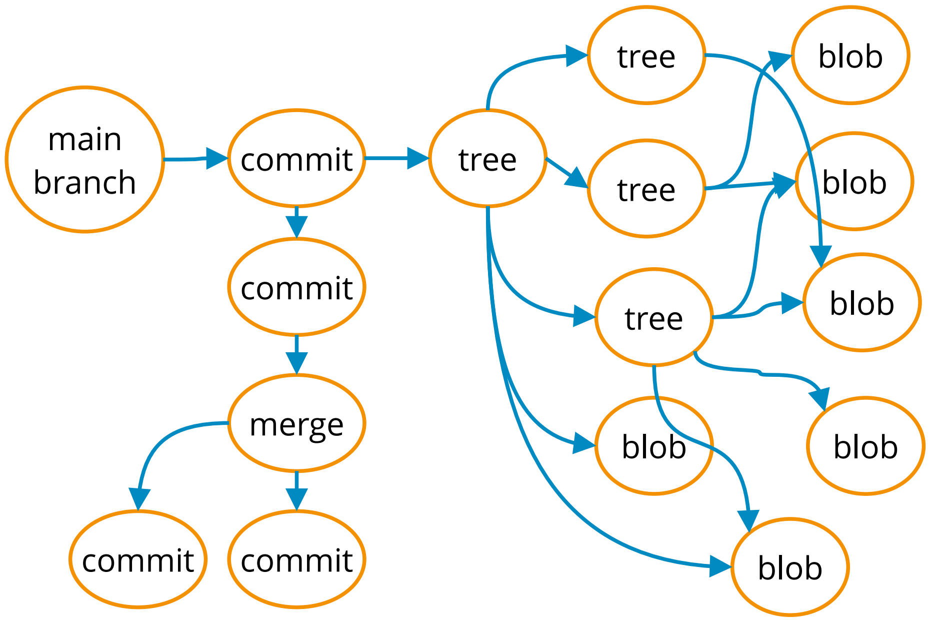
Stored in `.git/objects/`

```
find .git/objects -type f
```

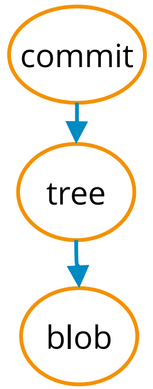
```
    git cat-file -p ObjectID
    git cat-file -p Reference:FilePath
    git cat-file -p Reference:FilePath
    git cat-file -p Reference:DirPath
    git cat-file -p HEAD
```

Object / ref Graph

Many objects / refs point to other objects / refs
But in a non-cyclic manner (no cycles)
In Math terms: Directed Acyclic Graph (DAG)
For example: a tree can not point to itself or its parent



Orphan objects:
Not reachable by any ref (branch, tag, remote ref, symbolic ref)



Commit objects

What's inside a commit object:

- tree (root directory)
- parent(s)
- author (email)
- committer (email)
- author timestamp
- committer timestamp
- signature
- message
- notes

No patch / diff is stored for commits
Unlike svn, cvs, hg, etc

(pack files store diff for large objects for optimization)

Commit objects

An eample of a commit object

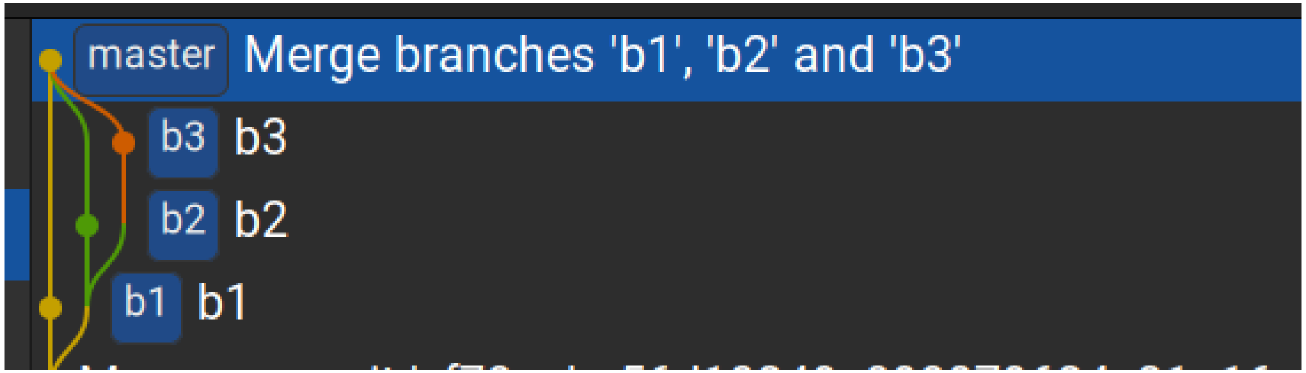
```
starcal $ git cat-file -p main
tree ad656b15b11bdbd3eadcfcbf914c7fc6f7812a55
parent 3c11a8ae61fa3548c10a2209fea24a9afe8d9c50
author Saeed Rasooli <saeed.gnu@gmail.com> 1721600511 +0330
committer Saeed Rasooli <saeed.gnu@gmail.com> 1721600511 +0330
gpgsig -----BEGIN PGP SIGNATURE-----
....
 -----END PGP SIGNATURE-----

support building package for AlmaLinux using docker
```

Commit parents

Non-merge commits have one parent (except for initial commit)

Merge commits typically have two parents

Merge commits with 3 or more parents are called octo-merge or Octopus Merge.
Simply write: `git merge branch1 branch2 ...`



master Merge branches 'b1', 'b2' and 'b3'
b3 b3
b2 b2
b1 b1

Author and committer

Author is the person who supposedly made the change (sent a patch for example).
Committer is the person who committed the change.
Often they are the same.

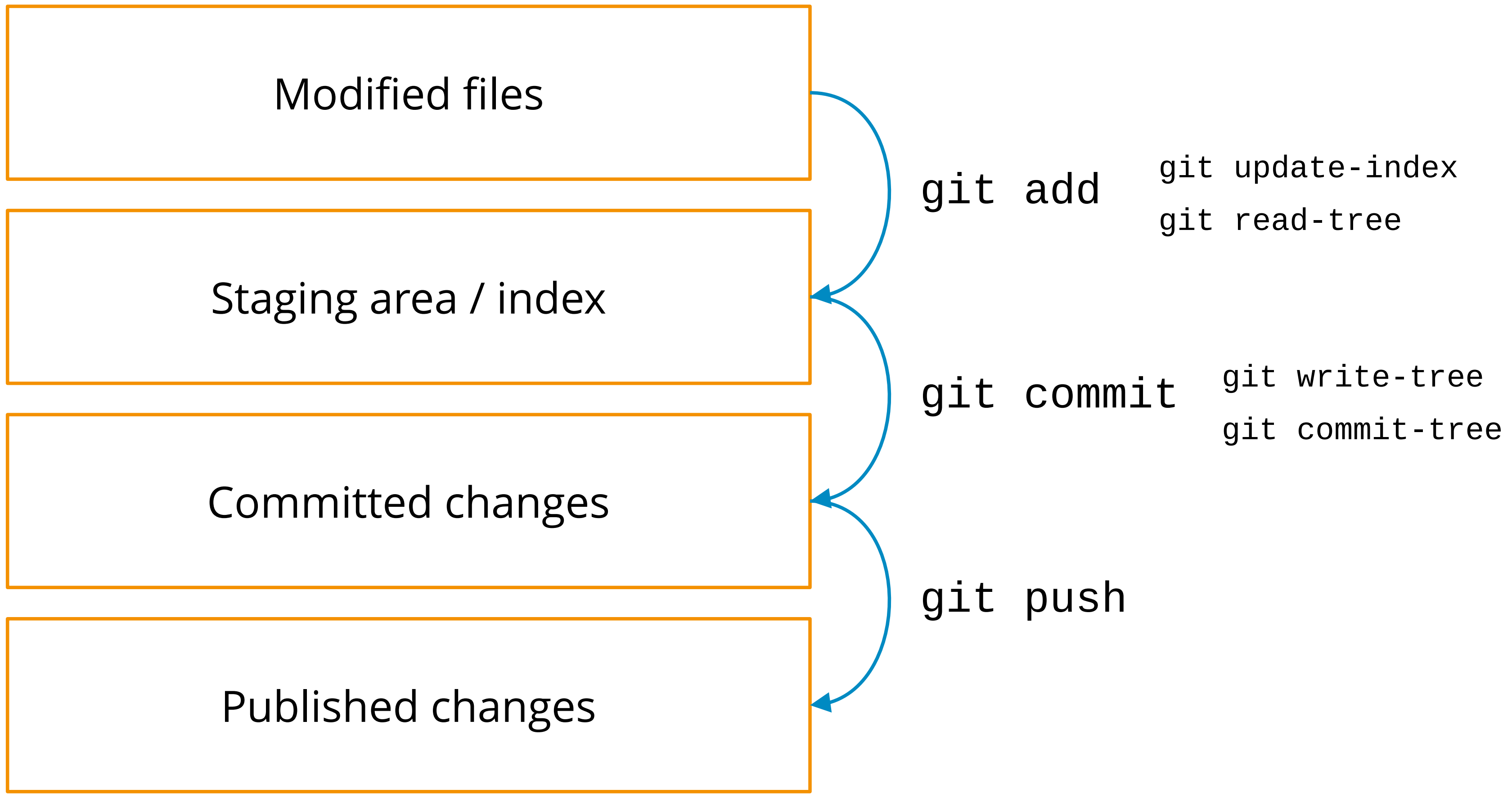
`git log` only shows author time by default.

When ***rebase*** or ***amend*** a commit, only **committer time** changes.

Can fake timestamps with env vars:

```
GIT_AUTHOR_DATE
GIT_COMMITTER_DATE
```

Stages of data



Recover or Cleanup

To access unreachable objects: **`git fsck --unreachable`**

See: **`man git fsck`**

To remove all unreachable commits, trees and blobs

```
git reflog expire --expire-unreachable=now --all
git gc --prune=now
```

Lesson: do WIP commits, and `git commit --amend` frequently!

Fetch and pull

`git fetch` connects to a remote, downloads all changes and updates its refs (representations of remote branches) stored in `.git/refs/remotes/NAME/`

`git pull` is equivalent to **`git fetch`** and then **`git merge`**

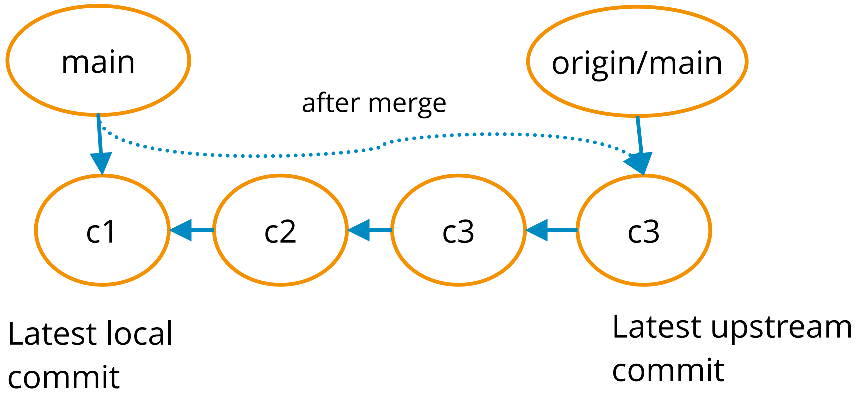
`git pull -r` is equivalent to **`git fetch`** and then **`git rebase`**

Merge: fast-forward

When you run `git pull` or `git merge`, and the latest upstream commit is a linear descendant of the latest local commit, no merge commit is needed.

So the branch ref will be changed to point to the new commit.

Then it’s called fast-forward.



Merge commit

When fast-forward is not possible, a merge commit needs to be created.

Git uses 3-way merge algorithm to combine and create a merge commit

Merge commits generally have two parents, but can have more as mentioned above

Merge commit

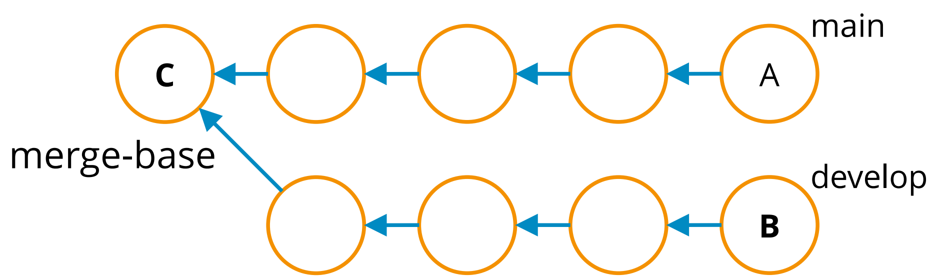
```
$ git cat-file -p e3f229047af4ccd23d66814fa2194e3313fd6c1f
tree 46b2bc03ab556f209a74d5c9073c98a37cc56a6d
parent efe837835946b6f8eb5c9e2e9c7a9b751ce05e8a
parent d15e17eab787103b30fe51d88b02f25cdd5cbece
author Saeed Rasooli <saeedgnu@riseup.net> 1643816137 +0330
committer GitHub <noreply@github.com> 1643816137 +0330
gpgsig -----BEGIN PGP SIGNATURE-----

 wsBcBAABCAAQBQJh+qTJCRBK7hj4Ov3rIwAAMMsIADakXiYX3Y3byrSVHQP5ZDMh
 bzp5/Jq5EdfdZVH7nr33jtawY8vyes4VR62CV+H1+Asf0XVoqYAoc1dATvvHRTZu
 iASDLwNIsWKuvOJrZbaaI8o/4yrvPazJLhQAmczpEMEJls2BubY07aerwKvTkKly
 nXdBeNBKGDfAG5CTNTNmmJiG8ccHtQR3PkGeRE+QewMWmdvLZeJDKFy7SDlXpJiw
 Sy+3qu/L8sJL+cXxl61/YDuWJd/pqZGQyaFRCbLUrWzzd0Mzv9pKbbRd+ceMq2xi
 rEPzluE+XhxYXutmS1gDo7WooD1NsPUWpKqTR1w3Lzt+Qf2wIMu5hIsE44M7jdc=
 =haBm
 -----END PGP SIGNATURE-----


Merge pull request #361 from BoboTiG/fix-deprecation-warnings-escape-sequence

Fix DeprecationWarning: invalid escape sequence
```

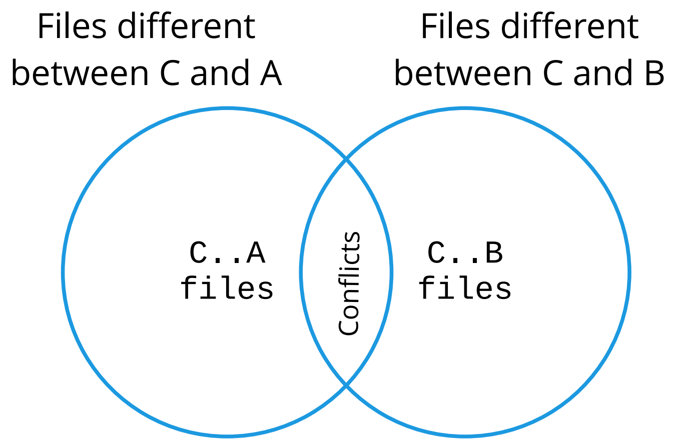
Merge algorithm

But what's 3-way merge algorithm?

Let's say we want to merge commit **A** (branch main) with commit **B** (branch develop)

First: git finds the Last Commit Ancestor (merge-base), let's call it **C**




Files different between C and A
Files different between C and B
C..A files
Conflicts
C..B files

Note: commits are pointing to their parents

Merge algorithm

Conflicts are files that are different between C and A, and also between C and B.

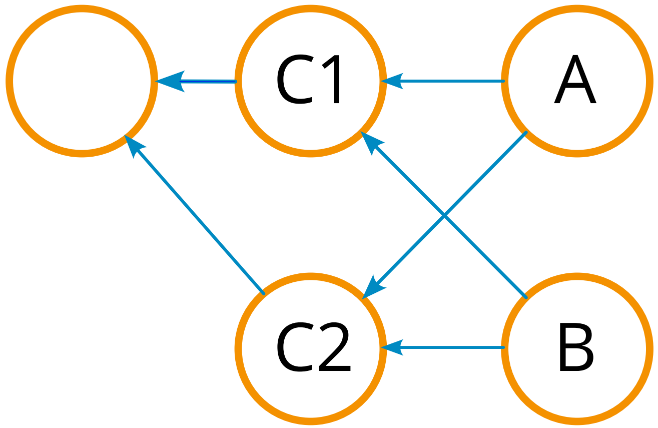
Non-conflict files are easily added to the new tree.
Git can auto-resolve some conflicts (if it can apply a patch successfully)
Other conflicts need to be resolved manually by user.

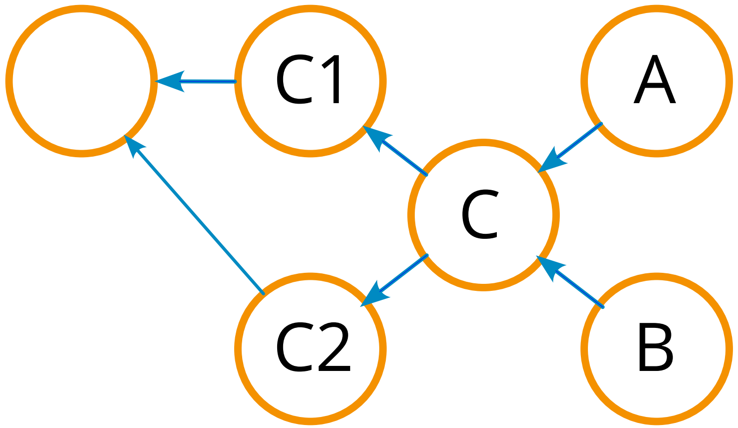
Only 3 commits are involved in this, hence the name: 3-way merge.

See: https://git-scm.com/docs/merge-strategies

Merge algorithm

In rare cases with **two** last commit ancestors (merge bases), here: C1 and C2 a virtual merge-base is created (here: C) and is used for 3-way merge.
This is the default for git, and is called **`ort`** (Ostensibly Recursive's Twin)





Note: commits are pointing to their parents

Commit signing

Commits can be signed with a private key (RSA, EC etc)

Using: `git commit --sign`

(You need to add your public key to Github or other hosting)

To automatically sign all commits, change `~/.gitconfig`, for example:

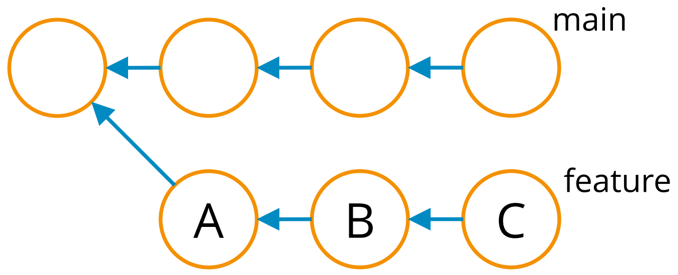
```
[user]
    name = Saeed Rasooli
    email = saeed.gnu@gmail.com
    signingkey = FD046A7C28FA209E
[gpg]
    program = gpg
[commit]
    gpgsign = true
```

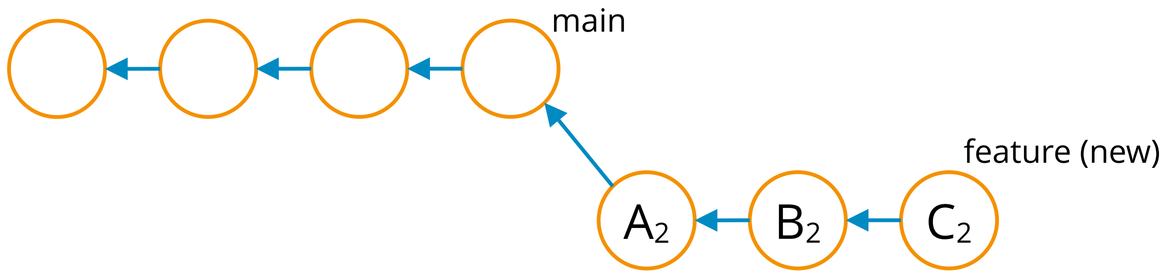
Rebase

`git` **`rebase`**: **re**-creates commits on top of a new **base**, then moves the head
Untimately, moving changes to a new base (generally branch)

```
git checkout feature
git rebase main
```

You can rebase based on a commit id as well.
Like: `git rebase 327b2f5`



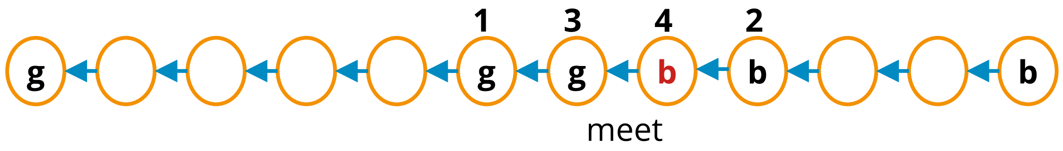


Note: show an example of interactive rebase: `git rebase -i`

Bisect

`git bisect` helps you find the commit causing a certian bug, using a **binary search**.
(or a commit that accidentically fixed a certain bug)

First you mark a **good** commit and a **bad** commit (two ends of our search range)
Then at each step, `bisect` checks out to the commit in the **middle** and asks you to mark it as **good** or **bad**, then **halves the search range**.
Until good and bad **meet each other**! (one is parent of other)



Git notes

Write notes for existing commits (amend creates a new commit)

Displayed on `git log` and `git show`

```
git notes list

git notes add <commit_hash>

git notes show <commit_hash>

git notes edit <commit_hash>

git notes remove <commit_hash>
```

Git blame

To figure out who made a certain change in a certain file. and when.
Shows the author and time of last modification for each line of file.

Simple usage:

```
git blame FILE_PATH
```

There are colorful wrappers and VS Code/Coduim extension for it.

Git ignore

To prevent git from adding or tracking certain files

Can list patterns of file name/path in any of these files:

- `~/.gitignore`
- `.gitignore` in the repository
- `.gitignore` in a sub-directory of the repository
- `.git/info/exclude` to avoid publishing the list

Git hooks

Hooks are scripts/programs you can place in a **`.git/hooks/`** directory to trigger actions at certain points in git's execution.

Can be used for:

- Checking/linting, formatting/normalizing code before commit
- Running tests before/after commit or before push
- Building things (executable, docker image etc) after commit
- Checking commit message conventions
- And more, see **`.git/hooks/`** directory and **`man githooks`**

Git config

Show my `~/.gitconfig` file

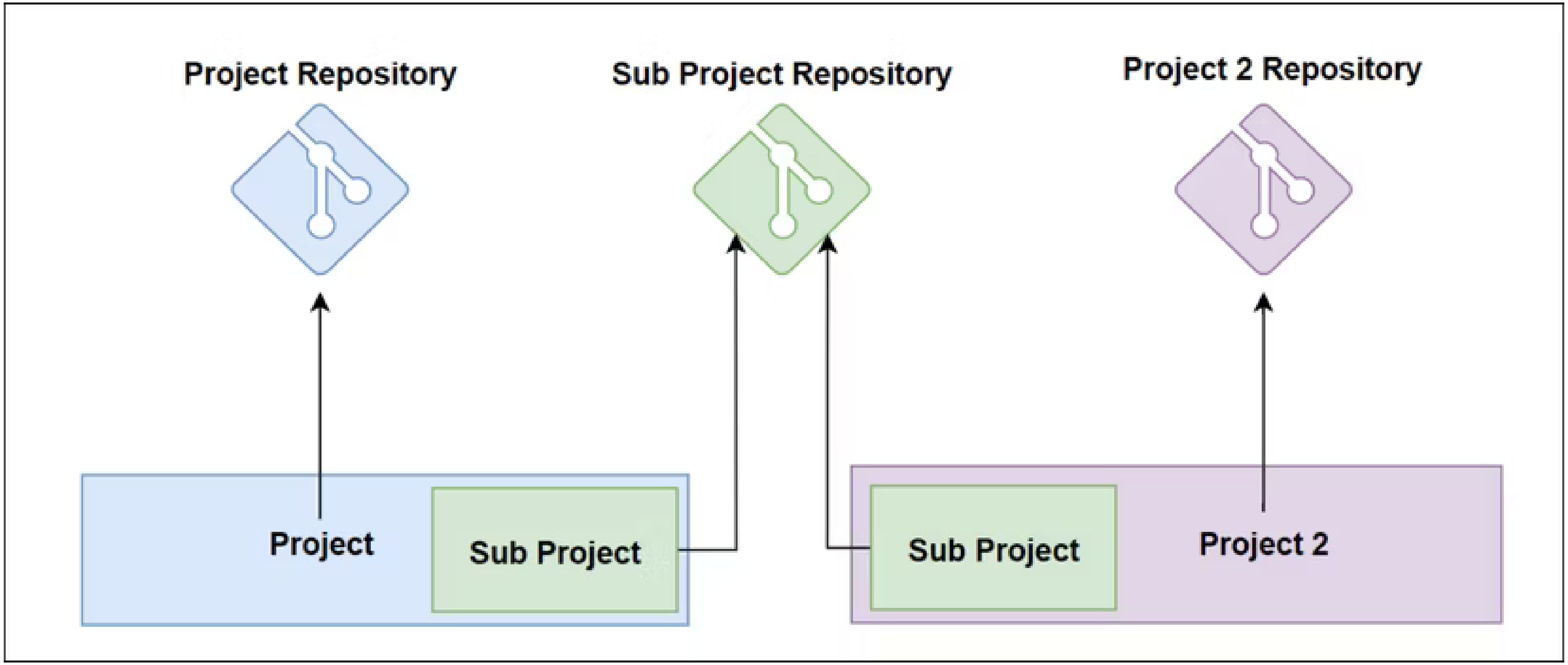
Git subtree

`git subtree` lets you nest one repository inside another as a sub-directory. It is one of several ways Git projects can manage project dependencies.

Unlike **`submodule`**, `subtree` does not require users of your repository to learn anything new. They can ignore the fact that you are using git subtree to manage dependencies.

All new changes to sub-directory are only stored in parent repo by default.

Git subtree



Source: https://**www.howtogeek.com**/devops/how-to-use-git-subtree-to-manage-multiple-project-repositories/

Git subtree

```
git remote add subtreeName git@github.com:user/projectName

git subtree add --prefix=dirName subtreeName subtreeBranch
git subtree pull --prefix=dirName subtreeName subtreeBranch [--squash]
git subtree merge --prefix=dirName subtreeName subtreeBranch [--squash]
```

See: **`man git subtree`**

Specially: **`--squash`**

Git subtree

```
git subtree push --prefix=dirName subtreeName subtreeBranch
```

Until ***`subtree push`***, all new commits on `dirName` are only stored in parent repo and pushed to parent repo on **`git push`**.
`subtree push` creates new commit objects for the subtree and pushes them to the subtree remote URL.

https://www.atlassian.com/git/tutorials/git-subtree
https://medium.com/@v/git-subtrees-a-tutorial-6ff568381844

Git submodule

To vendor / include extrenal dependencies in a repo without adding their code or full history to the repo. Only maintains commit hash of the child/sub-repo.
The command line interface is often confusing and troubling.
Best to automate every step / scenario with Makefile or scripts.

```
git submodule add URL [subDir]
git add .gitmodules subDir ; git commit
```

After clone (non-recursive): `git submodule update --init`

Use the status of the submodule's remote-tracking branch:

```
git submodule update --remote
```

Questions

Check out my Github:
https://github.com/ilius

Any questions?