gitsubmodules

gitsubmodules(7) ================

NAME ---- gitsubmodules - Mounting one repository inside another

SYNOPSIS -------- .gitmodules, $GIT_DIR/config

```
git submodule
git <command> --recurse-submodules
```

DESCRIPTION -----------

A submodule is a repository embedded inside another repository. The submodule has its own history; the repository it is embedded in is called a superproject.

On the filesystem, a submodule usually (but not always - see FORMS below) consists of (i) a Git directory located under the `$GIT_DIR/modules/` directory of its superproject, (ii) a working directory inside the superproject's working directory, and a `.git` file at the root of the submodule's working directory pointing to (i).

Assuming the submodule has a Git directory at `$GIT_DIR/modules/foo/` and a working directory at `path/to/bar/`, the superproject tracks the submodule via a `gitlink` entry in the tree at `path/to/bar` and an entry in its `.gitmodules` file (see linkgit:gitmodules[5]) of the form `submodule.foo.path` `=` `path/to/bar`.

The `gitlink` entry contains the object name of the commit that the superproject expects the submodule's working directory to be at.

The section `submodule.foo.*` in the `.gitmodules` file gives additional hints to Git's porcelain layer. For example, the `submodule.foo.url` setting specifies where to obtain the submodule.

Submodules can be used for at least two different use cases:

1. Using another project while maintaining independent history. Submodules allow you to contain the working tree of another project within your own working tree while keeping the history of both projects separate. Also, since submodules are fixed to an arbitrary version, the other project can be independently developed without affecting the superproject, allowing the superproject project to fix itself to new versions only when desired.

2. Splitting a (logically single) project into multiple repositories and tying them back together. This can be used to overcome current limitations of Git's implementation to have finer grained access:

- Size of the Git repository: In its current form Git scales up poorly for large repositories containing content that is not compressed by delta computation between trees. For example, you can use submodules to hold large binary assets and these repositories can be shallowly cloned such that you do not have a large history locally.
- Transfer size: In its current form Git requires the whole working tree present. It does not allow partial trees to be transferred in fetch or clone. If the project you work on consists of multiple repositories tied together as submodules in a superproject, you can avoid fetching the working trees of the repositories you are not interested in.
- Access control: By restricting user access to submodules, this can be used to implement read/write policies for different users.

The configuration of submodules ----------

Submodule operations can be configured using the following mechanisms (from highest to lowest precedence):

- The command line for those commands that support taking submodules as part of their pathspecs. Most commands have a boolean flag `--recurse-submodules` which specifies whether to recurse into submodules. Examples are `grep` and `checkout`. Some commands take enums, such as `fetch` and `push`, where you can specify how submodules are affected.

- The configuration inside the submodule. This includes `$GIT_DIR/config` in the submodule, but also settings in the tree such as a `.gitattributes` or `.gitignore` files that specify behavior of commands inside the submodule.
  For example an effect from the submodule's `.gitignore` file would be observed when you run `git status --ignore-submodules=none` in the superproject. This collects information from the submodule's working directory by running `status` in the submodule while paying attention to the `.gitignore` file of the submodule.
  The submodule's `$GIT_DIR/config` file would come into play when running `git push --recurse-submodules=check` in the superproject, as this would check if the submodule has any changes not published to any remote. The remotes are configured in the submodule as usual in the `$GIT_DIR/config` file.

- The configuration file `$GIT_DIR/config` in the superproject. Git only recurses into active submodules (see "ACTIVE SUBMODULES" section below).
  If the submodule is not yet initialized, then the configuration inside the submodule does not exist yet, so where to obtain the submodule from is configured here for example.

- The `.gitmodules` file inside the superproject. A project usually uses this file to suggest defaults for the upstream collection of repositories for the mapping that is required between a submodule's name and its path.

This file mainly serves as the mapping between the name and path of submodules in the superproject, such that the submodule's Git directory can be located.
If the submodule has never been initialized, this is the only place where submodule configuration is found. It serves as the last fallback to specify where to obtain the submodule from.

FORMS -----

Submodules can take the following forms:

- The basic form described in DESCRIPTION with a Git directory, a working directory, a `gitlink`, and a `.gitmodules` entry.

- "Old-form" submodule: A working directory with an embedded `.git` directory, and the tracking `gitlink` and `.gitmodules` entry in the superproject. This is typically found in repositories generated using older versions of Git.
  It is possible to construct these old form repositories manually.
  When deinitialized or deleted (see below), the submodule's Git directory is automatically moved to `$GIT_DIR/modules/<name>/` of the superproject.

- Deinitialized submodule: A `gitlink`, and a `.gitmodules` entry, but no submodule working directory. The submodule's Git directory may be there as after deinitializing the Git directory is kept around. The directory which is supposed to be the working directory is empty instead.
  A submodule can be deinitialized by running `git` `submodule` `deinit`. Besides emptying the working directory, this command only modifies the superproject's `$GIT_DIR/config` file, so the superproject's history is not affected. This can be undone using `git` `submodule` `init`.

- Deleted submodule: A submodule can be deleted by running `git` `rm` `<submodule-path>` `&&` `git` `commit`. This can be undone using `git` `revert`.
  The deletion removes the superproject's tracking data, which are both the `gitlink` entry and the section in the `.gitmodules` file. The submodule's working directory is removed from the file system, but the Git directory is kept around as it to make it possible to checkout past commits without requiring fetching from another repository.
  To completely remove a submodule, manually delete `$GIT_DIR/modules/<name>/`.

ACTIVE SUBMODULES ----------

A submodule is considered active,

```
1. if `submodule.<name>.active` is set to `true`
```

or

```
2. if the submodule's path matches the pathspec in `submodule.active`
```

or

```
3. if `submodule.<name>.url` is set.
```

and these are evaluated in this order.

For example:

```
[submodule "foo"]
  active = false
  url = https://example.org/foo
[submodule "bar"]
  active = true
  url = https://example.org/bar
[submodule "baz"]
  url = https://example.org/baz
```

In the above config only the submodules ’bar’ and ’baz’ are active, ’bar’ due to (1) and ’baz’ due to (3). ’foo’ is inactive because (1) takes precedence over (3)

Note that (3) is a historical artefact and will be ignored if the (1) and (2) specify that the submodule is not active. In other words, if we have a `submodule.<name>.active` set to `false` or if the submodule’s path is excluded in the pathspec in `submodule.active`, the url doesn’t matter whether it is present or not. This is illustrated in the example that follows.

```
[submodule "foo"]
  active = true
  url = https://example.org/foo
[submodule "bar"]
  url = https://example.org/bar
[submodule "baz"]
  url = https://example.org/baz
[submodule "bob"]
  ignore = true
[submodule]
  active = b*
  active = :(exclude) baz
```

In here all submodules except ’baz’ (foo, bar, bob) are active. ’foo’ due to its own active flag and all the others due to the submodule active pathspec, which specifies that any submodule starting with ’b’ except ’baz’ are also active, regardless of the presence of the .url field.

Workflow for a third party library ----------

```
# Add a submodule
git submodule add <URL> <path>

# Occasionally update the submodule to a new version:
git -C <path> checkout <new-version>
git add <path>
git commit -m "update submodule to new version"

# See the list of submodules in a superproject
git submodule status

# See FORMS on removing submodules
```

Workflow for an artificially split repo ----------

```
# Enable recursion for relevant commands, such that
# regular commands recurse into submodules by default
git config --global submodule.recurse true

# Unlike most other commands below, clone still needs
# its own recurse flag:
git clone --recurse <URL> <directory>
cd <directory>

# Get to know the code:
git grep foo
git ls-files --recurse-submodules
```

`git` `ls-files` also requires its own `--recurse-submodules` flag.

```
# Get new code
git fetch
git pull --rebase

# Change worktree
```

```
git checkout
git reset
```

Implementation details ----------------------

When cloning or pulling a repository containing submodules the submodules will not be checked out by default; you can instruct `clone` to recurse into submodules. The `init` and `update` subcommands of `git` `submodule` will maintain submodules checked out and at an appropriate revision in your working tree. Alternatively you can set `submodule.recurse` to have `checkout` recurse into submodules (note that `submodule.recurse` also affects other Git commands, see linkgit:git-config[1] for a complete list).

SEE ALSO -------- linkgit:git-submodule[1], linkgit:gitmodules[5].

GIT --- Part of the linkgit:git[1] suite

□□□ #1
□ □□□ □□□ 2026-10-31 20:14:29 UTC
□ □□□ □□□ 2026-10-31 20:14:29 UTC