# gitrepository-layout


- Git Repository Format Versions
  - Version 0
  - Version 1


gitrepository-layout(5) ==========

NAME ---- gitrepository-layout - Git Repository Layout

SYNOPSIS -------- $GIT_DIR/*

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

A Git repository comes in two different flavours:

- a `.git` directory at the root of the working tree;
- a `<project>.git` directory that is a 'bare' repository (i.e. without its own working tree), that is typically used for exchanging histories with others by pushing into it and fetching from it.

**Note**: Also you can have a plain text file `.git` at the root of your working tree, containing `gitdir: <path>` to point at the real directory that has the repository. This mechanism is called a 'gitfile' and is usually managed via the `git submodule` and `git worktree` commands. It is often used for a working tree of a submodule checkout, to allow you in the containing superproject to `git checkout` a branch that does not have the submodule. The `checkout` has to remove the entire submodule working tree, without losing the submodule repository.

These things may exist in a Git repository.

objects
Object store associated with this repository. Usually an object store is self sufficient (i.e. all the objects that are referred to by an object found in it are also found in it), but there are a few ways to violate it.

1. You could have an incomplete but locally usable repository by creating a shallow clone.
   See linkgit:git-clone[1].

2. You could be using the `objects/info/alternates` or `$GIT_ALTERNATE_OBJECT_DIRECTORIES` mechanisms to 'borrow' objects from other object stores. A repository with this kind of incomplete object store is not suitable to be published for use with dumb transports but otherwise is OK as long as `objects/info/alternates` points at the object stores it borrows from.
   This directory is ignored if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/objects" will be used instead.

objects/[0-9a-f][0-9a-f]
A newly created object is stored in its own file. The objects are splayed over 256 subdirectories using the first two characters of the sha1 object name to keep the number of directory entries in `objects` itself to a manageable number. Objects found here are often called 'unpacked' (or 'loose') objects.

objects/pack
Packs (files that store many objects in compressed form, along with index files to allow them to be randomly accessed) are found in this directory.

objects/info
Additional information about the object store is recorded in this directory.

objects/info/packs
This file is to help dumb transports discover what packs are available in this object store. Whenever a pack is added or removed, `git` `update-server-info` should be run to keep this file up to date if the repository is published for dumb transports. 'git repack' does this by default.

objects/info/alternates
This file records paths to alternate object stores that this object store borrows objects from, one pathname per line. Note that not only native Git tools use it locally, but the HTTP fetcher also tries to use it remotely; this will usually work if you have relative paths (relative to the object database, not to the repository!) in your alternates file, but it will not work if you use absolute paths unless the absolute path in filesystem and web URL is the same. See also `objects/info/http-alternates`.

objects/info/http-alternates
This file records URLs to alternate object stores that this object store borrows objects from, to be used when the repository is fetched over HTTP.

refs
References are stored in subdirectories of this directory. The 'git prune' command knows to preserve objects reachable from refs found in this directory and its subdirectories. This directory is ignored (except refs/bisect, refs/rewritten and refs/worktree) if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/refs" will be used instead.

refs/heads/`name`
records tip-of-the-tree commit objects of branch `name`

refs/tags/`name`
records any object name (not necessarily a commit object, or a tag object that points at a commit object).

refs/remotes/`name`
records tip-of-the-tree commit objects of branches copied from a remote repository.

refs/replace/`<obj-sha1>`
records the SHA-1 of the object that replaces `<obj-sha1>`. This is similar to info/grafts and is internally used and maintained by linkgit:git-replace[1]. Such refs can be exchanged between repositories while grafts are not.

packed-refs
records the same information as refs/heads/, refs/tags/, and friends record in a more efficient way. See linkgit:git-pack-refs[1]. This file is ignored if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/packed-refs" will be used instead.

HEAD
A symref (see glossary) to the `refs/heads/` namespace describing the currently active branch. It does not mean much if the repository is not associated with any working tree (i.e. a ’bare’ repository), but a valid Git repository **must** have the HEAD file; some porcelains may use it to guess the designated "default" branch of the repository (usually ’master’). It is legal if the named branch ’name’ does not (yet) exist. In some legacy setups, it is a symbolic link instead of a symref that points at the current branch.

HEAD can also record a specific commit directly, instead of being a symref to point at the current branch. Such a state is often called ’detached HEAD.’ See linkgit:git-checkout[1] for details.

config
Repository specific configuration file. This file is ignored if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/config" will be used instead.

config.worktree
Working directory specific configuration file for the main working directory in multiple working directory setup (see linkgit:git-worktree[1]).

ifndef::with-breaking-changes[] branches:: A deprecated way to store shorthands to be used to specify a URL to ’git fetch’, ’git pull’ and ’git push’. A file can be stored as `branches/<name>` and then ’name’ can be given to these commands in place of ’repository’ argument. See the REMOTES section in linkgit:git-fetch[1] for details. This mechanism is legacy and not likely to be found in modern repositories. This directory is ignored if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/branches" will be used instead.
Git will stop reading remotes from this directory in Git 3.0. endif::with-breaking-changes[]

hooks
Hooks are customization scripts used by various Git commands. A handful of sample hooks are installed when ’git init’ is run, but all of them are disabled by default. To enable, the `.sample` suffix

has to be removed from the filename by renaming. Read linkgit:githooks[5] for more details about each hook. This directory is ignored if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/hooks" will be used instead.

common
When multiple working trees are used, most of files in $GIT_DIR are per-worktree with a few known exceptions. All files under ’common’ however will be shared between all working trees.

index
The current index file for the repository. It is usually not found in a bare repository.

sharedindex.<SHA-1>
The shared index part, to be referenced by $GIT_DIR/index and other temporary index files. Only valid in split index mode.

info
Additional information about the repository is recorded in this directory. This directory is ignored if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/info" will be used instead.

info/refs
This file helps dumb transports discover what refs are available in this repository. If the repository is published for dumb transports, this file should be regenerated by ’git update-server-info’ every time a tag or branch is created or modified. This is normally done from the `hooks/update` hook, which is run by the ’git-receive-pack’ command when you ’git push’ into the repository.

info/grafts
This file records fake commit ancestry information, to pretend the set of parents a commit has is different from how the commit was actually created. One record per line describes a commit and its fake parents by listing their 40-byte hexadecimal object names separated by a space and terminated by a newline.

Note that the grafts mechanism is outdated and can lead to problems transferring objects between repositories; see linkgit:git-replace[1] for a more flexible and robust system to do the same thing.

info/exclude
This file, by convention among Porcelains, stores the exclude pattern list. `.gitignore` is the per-directory ignore file. ’git status’, ’git add’, ’git rm’ and ’git clean’ look at it but the core Git commands do not look at it. See also: linkgit:gitignore[5].

info/attributes
Defines which attributes to assign to a path, similar to per-directory `.gitattributes` files. See also: linkgit:gitattributes[5].

info/sparse-checkout
This file stores sparse checkout patterns. See also: linkgit:git-read-tree[1].

ifndef::with-breaking-changes[] remotes:: Stores shorthands for URL and default refnames for use when interacting with remote repositories via ’git fetch’, ’git pull’ and ’git push’ commands. See the REMOTES section in linkgit:git-fetch[1] for details. This mechanism is legacy and not likely to be found in modern repositories. This directory is ignored if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/remotes" will be used instead.
Git will stop reading remotes from this directory in Git 3.0. endif::with-breaking-changes[]

logs
Records of changes made to refs are stored in this directory. See linkgit:git-update-ref[1] for more information. This directory is ignored (except logs/HEAD) if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/logs" will be used instead.

logs/refs/heads/`name`
Records all changes made to the branch tip named `name`.

logs/refs/tags/`name`
Records all changes made to the tag named `name`.

shallow
This is similar to `info/grafts` but is internally used and maintained by shallow clone mechanism. See `--depth` option to linkgit:git-clone[1] and linkgit:git-fetch[1]. This file is ignored if $GIT_COMMON_DIR is set and "$GIT_COMMON_DIR/shallow" will be used instead.

commondir
If this file exists, $GIT_COMMON_DIR (see linkgit:git[1]) will be set to the path specified in this file if it is not explicitly set. If the specified path is relative, it is relative to $GIT_DIR. The repository with commondir is incomplete without the repository pointed by "commondir".

modules
Contains the git-repositories of the submodules.

worktrees
Contains administrative data for linked working trees. Each subdirectory contains the working tree-related part of a linked working tree. This directory is ignored if $GIT_COMMON_DIR is set, in which case "$GIT_COMMON_DIR/worktrees" will be used instead.

worktrees/<id>/gitdir
A text file containing the absolute path back to the .git file that points to here. This is used to check if the linked repository has been manually removed and there is no need to keep this directory any more. The mtime of this file should be updated every time the linked repository is accessed.

worktrees/<id>/locked
If this file exists, the linked working tree may be on a portable device and not available. The presence of this file prevents `worktrees/<id>` from being pruned either automatically or manually by `git` `worktree` `prune`. The file may contain a string explaining why the repository is locked.

worktrees/<id>/config.worktree
Working directory specific configuration file.

# Git Repository Format Versions

Every git repository is marked with a numeric version in the `core.repositoryformatversion` key of its `config` file. This version specifies the rules for operating on the on-disk repository data. An implementation of git which does not understand a particular version advertised by an on-disk repository MUST NOT operate on that repository; doing so risks not only producing wrong results, but actually losing data.

Because of this rule, version bumps should be kept to an absolute minimum. Instead, we generally prefer these strategies:

- bumping format version numbers of individual data files (e.g., index, packfiles, etc). This restricts the incompatibilities only to those files.

- introducing new data that gracefully degrades when used by older clients (e.g., pack bitmap files are ignored by older clients, which simply do not take advantage of the optimization they provide).

A whole-repository format version bump should only be part of a change that cannot be independently versioned. For instance, if one were to change the reachability rules for objects, or the rules for locking refs, that would require a bump of the repository format version.

Note that this applies only to accessing the repository's disk contents directly. An older client which understands only format `0` may still connect via `git://` to a repository using format `1`, as long as the server process understands format `1`.

The preferred strategy for rolling out a version bump (whether whole repository or for a single file) is to teach git to read the new format, and allow writing the new format with a config switch or command line option (for experimentation or for those who do not care about backwards compatibility with older gits). Then after a long period to allow the reading capability to become common, we may switch to writing the new format by default.

The currently defined format versions are:

# Version `0`

This is the format defined by the initial version of git, including but not limited to the format of the repository directory, the repository configuration file, and the object and ref storage. Specifying the complete behavior of git is beyond the scope of this document.

# Version `1`

This format is identical to version `0`, with the following exceptions:

```
1. When reading the `core.repositoryformatversion` variable, a git
   implementation which supports version 1 MUST also read any
   configuration keys found in the `extensions` section of the
   configuration file.


2. If a version-1 repository specifies any `extensions.*` keys that
   the running git has not implemented, the operation MUST NOT
   proceed. Similarly, if the value of any known key is not understood
   by the implementation, the operation MUST NOT proceed.
```

Note that if no extensions are specified in the config file, then `core.repositoryformatversion` SHOULD be set to `0` (setting it to `1` provides no benefit, and makes the repository incompatible with older implementations of git).

The defined extensions are given in the `extensions.*` section of linkgit:git-config[1]. Any implementation wishing to define a new extension should make a note of it there, in order to claim the name.

SEE ALSO -------- linkgit:git-init[1], linkgit:git-clone[1], linkgit:git-config[1], linkgit:git-fetch[1], linkgit:git-pack-refs[1], linkgit:git-gc[1], linkgit:git-checkout[1], linkgit:gitglossary[7], The Git User’s Manual

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

---

□□□ #1
□ □□□ □□□ 2026-10-31 20:14:26 UTC
□ □□□ □□□ 2026-10-31 20:14:26 UTC