git-worktree

git-worktree(1) ==========

NAME ---- git-worktree - Manage multiple working trees

SYNOPSIS --------

git worktree add [-f] [--detach] [--checkout] [--lock [--reason <string>]] [--orphan] [(-b | -B) <new-branch>] <path> [<commit-ish>] git worktree list [-v | --porcelain [-z]] git worktree lock [--reason <string>] <worktree> git worktree move <worktree> <new-path> git worktree prune [-n] [-v] [--expire <expire>] git worktree remove [-f] <worktree> git worktree repair [<path>…] git worktree unlock <worktree>

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

Manage multiple working trees attached to the same repository.

A git repository can support multiple working trees, allowing you to check out more than one branch at a time. With `git` `worktree` `add` a new working tree is associated with the repository, along with additional metadata that differentiates that working tree from others in the same repository. The working tree, along with this metadata, is called a "worktree".

This new worktree is called a "linked worktree" as opposed to the "main worktree" prepared by linkgit:git-init[1] or linkgit:git-clone[1]. A repository has one main worktree (if it’s not a bare repository) and zero or more linked worktrees. When you are done with a linked worktree, remove it with `git` `worktree` `remove`.

In its simplest form, `git` `worktree` `add` `<path>` automatically creates a new branch whose name is the final component of *<path>*, which is convenient if you plan to work on a new topic. For instance, `git` `worktree` `add` `../hotfix` creates new branch `hotfix` and checks it out at path `../hotfix`. To instead work on an existing branch in a new worktree, use `git` `worktree` `add` `<path>` `<branch>`. On the other hand, if you just plan to make some experimental changes or do testing without disturbing existing development, it is often convenient to create a ’throwaway’ worktree not associated with any branch. For instance, `git` `worktree` `add` `-d` `<path>` creates a new worktree with a detached `HEAD` at the same commit as the current branch.

If a working tree is deleted without using `git` `worktree` `remove`, then its associated administrative files, which reside in the repository (see "DETAILS" below), will eventually be removed automatically (see `gc.worktreePruneExpire` in linkgit:git-config[1]), or you can run `git` `worktree` `prune` in the main or any linked worktree to clean up any stale administrative files.

If the working tree for a linked worktree is stored on a portable device or network share which is not always mounted, you can prevent its administrative files from being pruned by issuing the `git worktree lock` command, optionally specifying `--reason` to explain why the worktree is locked.

COMMANDS -------- `add` `<path>` `[<commit-ish>]`::

Create a worktree at *<path>* and checkout *<commit-ish>* into it. The new worktree is linked to the current repository, sharing everything except per-worktree files such as `HEAD`, `index`, etc. As a convenience, *<commit-ish>* may be a bare “-”, which is synonymous with `@{-1}`.
If *<commit-ish>* is a branch name (call it *<branch>*) and is not found, and neither `-b` nor `-B` nor `--detach` are used, but there does exist a tracking branch in exactly one remote (call it *<remote>*) with a matching name, treat as equivalent to:
---------- $ git worktree add --track -b <branch> <path> <remote>/<branch> ----------
If the branch exists in multiple remotes and one of them is named by the `checkout.defaultRemote` configuration variable, we’ll use that one for the purposes of disambiguation, even if the *<branch>* isn’t unique across all remotes. Set it to e.g. `checkout.defaultRemote=origin` to always checkout remote branches from there if *<branch>* is ambiguous but exists on the `origin` remote. See also `checkout.defaultRemote` in linkgit:git-config[1].
If *<commit-ish>* is omitted and neither `-b` nor `-B` nor `--detach` used, then, as a convenience, the new worktree is associated with a branch (call it *<branch>*) named after `$(basename` `<path>)`. If *<branch>* doesn’t exist, a new branch based on `HEAD` is automatically created as if `-b` `<branch>` was given. If *<branch>* does exist, it will be checked out in the new worktree, if it’s not checked out anywhere else, otherwise the command will refuse to create the worktree (unless `--force` is used).
If *<commit-ish>* is omitted, neither `--detach`, or `--orphan` is used, and there are no valid local branches (or remote branches if `--guess-remote` is specified) then, as a convenience, the new worktree is associated with a new unborn branch named *<branch>* (after `$(basename` `<path>)` if neither `-b` or `-B` is used) as if `--orphan` was passed to the command. In the event the repository has a remote and `--guess-remote` is used, but no remote or local branches exist, then the command fails with a warning reminding the user to fetch from their remote first (or override by using `-f`/`--force`).

`list`

List details of each worktree. The main worktree is listed first, followed by each of the linked worktrees. The output details include whether the worktree is bare, the revision currently checked out, the branch currently checked out (or "detached HEAD" if none), "locked" if the worktree is locked, "prunable" if the worktree can be pruned by the `prune` command.

`lock`

If a worktree is on a portable device or network share which is not always mounted, lock it to prevent its administrative files from being pruned automatically. This also prevents it from being moved or deleted. Optionally, specify a reason for the lock with `--reason`.

`move`

Move a worktree to a new location. Note that the main worktree or linked worktrees containing submodules cannot be moved with this command. (The `git worktree repair` command, however, can reestablish the connection with linked worktrees if you move the main worktree manually.)

`prune`

Remove worktree information in `$GIT_DIR/worktrees` for worktrees whose working trees are missing. Useful after manually removing a working tree that is no longer needed (but use "git worktree remove" next time you want to do so). Also, if you *moved* a working tree elsewhere causing the worktree information to become dangling, see "git worktree repair" to reconnect the worktree to the new working tree location.

`remove`

Remove a worktree. Only clean worktrees (no untracked files and no modification in tracked files) can be removed. Unclean worktrees or ones with submodules can be removed with `--force`. The main worktree cannot be removed.

`repair` `[<path>…]`

Repair worktree administrative files, if possible, if they have become corrupted or outdated due to external factors.
For instance, if the main worktree (or bare repository) is moved, linked worktrees will be unable to locate it. Running `repair` in the main worktree will reestablish the connection from linked worktrees back to the main worktree.
Similarly, if the working tree for a linked worktree is moved without using `git worktree move`, the main worktree (or bare repository) will be unable to locate it. Running `repair` within the recently-moved worktree will reestablish the connection. If multiple linked worktrees are moved, running `repair` from any worktree with each tree's new *<path>* as an argument, will reestablish the connection to all the specified paths.
If both the main worktree and linked worktrees have been moved or copied manually, then running `repair` in the main worktree and specifying the new *<path>* of each linked worktree will reestablish all connections in both directions.

`unlock`

Unlock a worktree, allowing it to be pruned, moved or deleted.

OPTIONS -------

`-f`

`--force`
By default, `add` refuses to create a new worktree when *<commit-ish>* is a branch name and is already checked out by another worktree, or if *<path>* is already assigned to some worktree but is missing (for instance, if *<path>* was deleted manually). This option overrides these safeguards. To add a missing but locked worktree path, specify `--force` twice.

`move` refuses to move a locked worktree unless `--force` is specified twice. If the destination is already assigned to some other worktree but is missing (for instance, if *<new-path>* was deleted manually), then `--force` allows the move to proceed; use `--force` twice if the destination is locked.

`remove` refuses to remove an unclean worktree unless `--force` is used. To remove a locked worktree, specify `--force` twice.

`-b` `<new-branch>`

`-B` `<new-branch>`
With `add`, create a new branch named *<new-branch>* starting at *<commit-ish>*, and check out *<new-branch>* into the new worktree. If *<commit-ish>* is omitted, it defaults to `HEAD`. By default, `-b` refuses to create a new branch if it already exists. `-B` overrides this safeguard, resetting *<new-branch>* to *<commit-ish>*.

`-d`

`--detach`
With `add`, detach `HEAD` in the new worktree. See "DETACHED HEAD" in linkgit:git-checkout[1].

`--checkout`

`--no-checkout`
By default, `add` checks out *<commit-ish>*, however, `--no-checkout` can be used to suppress checkout in order to make customizations, such as configuring sparse-checkout. See "Sparse checkout" in linkgit:git-read-tree[1].

`--guess-remote`

`--no-guess-remote`
With `worktree` `add` `<path>`, without *<commit-ish>*, instead of creating a new branch from `HEAD`, if there exists a tracking branch in exactly one remote matching the basename of *<path>*, base the new branch on the remote-tracking branch, and mark the remote-tracking branch as "upstream" from the new branch. If there are multiple matches, the command fails. If there is no match, the command falls back to creating a new branch from `HEAD`.

This can also be set up as the default behaviour by using the `worktree.guessRemote` config option.

`--relative-paths`

`--no-relative-paths`
Link worktrees using relative paths or absolute paths (default). Overrides the `worktree.useRelativePaths` config option, see linkgit:git-config[1].

With `repair`, the linking files will be updated if there’s an absolute/relative mismatch, even if the links are correct.

`--track`

`--no-track`
When creating a new branch, if <*commit-ish*> is a branch, mark it as "upstream" from the new branch. This is the default if <*commit-ish*> is a remote-tracking branch. See `--track` in linkgit:git-branch[1] for details.

`--lock`
Keep the worktree locked after creation. This is the equivalent of `git` `worktree` `lock` after `git` `worktree` `add`, but without a race condition.

`-n`

`--dry-run`
With `prune`, do not remove anything; just report what it would remove.

`--orphan`
With `add`, make the new worktree and index empty, associating the worktree with a new unborn branch named <*new-branch*>.

`--porcelain`
With `list`, output in an easy-to-parse format for scripts. This format will remain stable across Git versions and regardless of user configuration. It is recommended to combine this with `-z`. See below for details.

`-z`
Terminate each line with a *NUL* rather than a newline when `--porcelain` is specified with `list`. This makes it possible to parse the output when a worktree path contains a newline character.

`-q`

`--quiet`
With `add`, suppress feedback messages.

`-v`

`--verbose`
With `prune`, report all removals.

With `list`, output additional information about worktrees (see below).

`--expire` `<time>`
With `prune`, only prune missing worktrees if older than <*time*>.

With `list`, annotate missing worktrees as prunable if they are older than <*time*>.

`--reason` `<string>`
With `lock` or with `add` `--lock`, an explanation why the worktree is locked.

<worktree>
Worktrees can be identified by path, either relative or absolute.

If the last path components in the worktree's path is unique among worktrees, it can be used to identify a worktree. For example if you only have two worktrees, at `/abc/def/ghi` and `/abc/def/ggg`, then `ghi` or `def/ghi` is enough to point to the former worktree.

REFS ---- When using multiple worktrees, some refs are shared between all worktrees, but others are specific to an individual worktree. One example is `HEAD`, which is different for each worktree. This section is about the sharing rules and how to access refs of one worktree from another.

In general, all pseudo refs are per-worktree and all refs starting with `refs/` are shared. Pseudo refs are ones like `HEAD` which are directly under `$GIT_DIR` instead of inside `$GIT_DIR/refs`. There are exceptions, however: refs inside `refs/bisect`, `refs/worktree` and `refs/rewritten` are not shared.

Refs that are per-worktree can still be accessed from another worktree via two special paths, `main-worktree` and `worktrees`. The former gives access to per-worktree refs of the main worktree, while the latter to all linked worktrees.

For example, `main-worktree/HEAD` or `main-worktree/refs/bisect/good` resolve to the same value as the main worktree's `HEAD` and `refs/bisect/good` respectively. Similarly, `worktrees/foo/HEAD` or `worktrees/bar/refs/bisect/bad` are the same as `$GIT_COMMON_DIR/worktrees/foo/HEAD` and `$GIT_COMMON_DIR/worktrees/bar/refs/bisect/bad`.

To access refs, it's best not to look inside `$GIT_DIR` directly. Instead use commands such as linkgit:git-rev-parse[1] or linkgit:git-update-ref[1] which will handle refs correctly.

CONFIGURATION FILE ---------- By default, the repository `config` file is shared across all worktrees. If the config variables `core.bare` or `core.worktree` are present in the common config file and `extensions.worktreeConfig` is disabled, then they will be applied to the main worktree only.

In order to have worktree-specific configuration, you can turn on the `worktreeConfig` extension, e.g.:

```
$ git config extensions.worktreeConfig true
```

In this mode, specific configuration stays in the path pointed by `git rev-parse --git-path config.worktree`. You can add or update configuration in this file with `git config --worktree`. Older Git versions will refuse to access repositories with this extension.

Note that in this file, the exception for `core.bare` and `core.worktree` is gone. If they exist in `$GIT_DIR/config`, you must move them to the `config.worktree` of the main worktree. You may also take this opportunity to review and move other configuration that you do not want to share to all worktrees:

- `core.worktree` should never be shared.

- `core.bare` should not be shared if the value is `core.bare=true`.
- `core.sparseCheckout` should not be shared, unless you are sure you always use sparse checkout for all worktrees.

See the documentation of `extensions.worktreeConfig` in linkgit:git-config[1] for more details.

DETAILS ------- Each linked worktree has a private sub-directory in the repository's `$GIT_DIR/worktrees` directory. The private sub-directory's name is usually the base name of the linked worktree's path, possibly appended with a number to make it unique. For example, when `$GIT_DIR=/path/main/.git` the command `git` `worktree` `add` `/path/other/test-next` `next` creates the linked worktree in `/path/other/test-next` and also creates a `$GIT_DIR/worktrees/test-next` directory (or `$GIT_DIR/worktrees/test-next1` if `test-next` is already taken).

Within a linked worktree, `$GIT_DIR` is set to point to this private directory (e.g. `/path/main/.git/worktrees/test-next` in the example) and `$GIT_COMMON_DIR` is set to point back to the main worktree's `$GIT_DIR` (e.g. `/path/main/.git`). These settings are made in a `.git` file located at the top directory of the linked worktree.

Path resolution via `git` `rev-parse` `--git-path` uses either `$GIT_DIR` or `$GIT_COMMON_DIR` depending on the path. For example, in the linked worktree `git` `rev-parse` `--git-path` `HEAD` returns `/path/main/.git/worktrees/test-next/HEAD` (not `/path/other/test-next/.git/HEAD` or `/path/main/.git/HEAD`) while `git` `rev-parse` `--git-path` `refs/heads/master` uses `$GIT_COMMON_DIR` and returns `/path/main/.git/refs/heads/master`, since refs are shared across all worktrees, except `refs/bisect`, `refs/worktree` and `refs/rewritten`.

See linkgit:gitrepository-layout[5] for more information. The rule of thumb is do not make any assumption about whether a path belongs to `$GIT_DIR` or `$GIT_COMMON_DIR` when you need to directly access something inside `$GIT_DIR`. Use `git` `rev-parse` `--git-path` to get the final path.

If you manually move a linked worktree, you need to update the `gitdir` file in the entry's directory. For example, if a linked worktree is moved to `/newpath/test-next` and its `.git` file points to `/path/main/.git/worktrees/test-next`, then update `/path/main/.git/worktrees/test-next/gitdir` to reference `/newpath/test-next` instead. Better yet, run `git` `worktree` `repair` to reestablish the connection automatically.

To prevent a `$GIT_DIR/worktrees` entry from being pruned (which can be useful in some situations, such as when the entry's worktree is stored on a portable device), use the `git` `worktree` `lock` command, which adds a file named `locked` to the entry's directory. The file contains the reason in plain text. For example, if a linked worktree's `.git` file points to `/path/main/.git/worktrees/test-next` then a file named `/path/main/.git/worktrees/test-next/locked` will prevent the `test-next` entry from being pruned. See linkgit:gitrepository-layout[5] for details.

When `extensions.worktreeConfig` is enabled, the config file `.git/worktrees/<id>/config.worktree` is read after `.git/config` is.

LIST OUTPUT FORMAT ---------- The `worktree` `list` command has two output formats. The default format shows the details on a single line with columns. For example:

```
$ git worktree list
/path/to/bare-source            (bare)
/path/to/linked-worktree        abcd1234 [master]
/path/to/other-linked-worktree  1234abc  (detached HEAD)
```

The command also shows annotations for each worktree, according to its state. These annotations are:

- `locked`, if the worktree is locked.
- `prunable`, if the worktree can be pruned via `git` `worktree` `prune`.

```
$ git worktree list
/path/to/linked-worktree    abcd1234 [master]
/path/to/locked-worktree    acbd5678 (brancha) locked
/path/to/prunable-worktree  5678abc  (detached HEAD) prunable
```

For these annotations, a reason might also be available and this can be seen using the verbose mode. The annotation is then moved to the next line indented followed by the additional information.

```
$ git worktree list --verbose
/path/to/linked-worktree              abcd1234 [master]
/path/to/locked-worktree-no-reason    abcd5678 (detached HEAD) locked
/path/to/locked-worktree-with-reason  1234abcd (brancha)
    locked: worktree path is mounted on a portable device
/path/to/prunable-worktree            5678abc1 (detached HEAD)
    prunable: gitdir file points to non-existent location
```

Note that the annotation is moved to the next line if the additional information is available, otherwise it stays on the same line as the worktree itself.

Porcelain Format ~~~~~~~~~~~~~~~~~ The porcelain format has a line per attribute. If `-z` is given then the lines are terminated with NUL rather than a newline. Attributes are listed with a label and value separated by a single space. Boolean attributes (like `bare` and `detached`) are listed as a label only, and are present only if the value is true. Some attributes (like `locked`) can be listed as a label only or with a value depending upon whether a reason is available. The first attribute of a worktree is always `worktree`, an empty line indicates the end of the record. For example:

```
$ git worktree list --porcelain
worktree /path/to/bare-source
```

```
bare

worktree /path/to/linked-worktree
HEAD abcd1234abcd1234abcd1234abcd1234abcd1234
branch refs/heads/master

worktree /path/to/other-linked-worktree
HEAD 1234abc1234abc1234abc1234abc1234abc1234a
detached

worktree /path/to/linked-worktree-locked-no-reason
HEAD 5678abc5678abc5678abc5678abc5678abc5678c
branch refs/heads/locked-no-reason
locked

worktree /path/to/linked-worktree-locked-with-reason
HEAD 3456def3456def3456def3456def3456def3456b
branch refs/heads/locked-with-reason
locked reason why is locked

worktree /path/to/linked-worktree-prunable
HEAD 1233def1234def1234def1234def1234def1234b
detached
prunable gitdir file points to non-existent location
```

Unless `-z` is used any "unusual" characters in the lock reason such as newlines are escaped and the entire reason is quoted as explained for the configuration variable `core.quotePath` (see linkgit:git-config[1]). For Example:

```
$ git worktree list --porcelain
...
locked "reason\nwhy is locked"
...
```

EXAMPLES -------- You are in the middle of a refactoring session and your boss comes in and demands that you fix something immediately. You might typically use linkgit:git-stash[1] to store your changes away temporarily, however, your working tree is in such a state of disarray (with new, moved, and removed files, and other bits and pieces strewn around) that you don’t want to risk disturbing any of it. Instead, you create a temporary linked worktree to make the emergency fix, remove it when done, and then resume your earlier refactoring session.

```
$ git worktree add -b emergency-fix ../temp master
$ pushd ../temp
# ... hack hack hack ...
$ git commit -a -m 'emergency fix for boss'
$ popd
$ git worktree remove ../temp
```

CONFIGURATION ----------

Everything below this line in this section is selectively included from the linkgit:git-config[1] documentation. The content is the same as what’s found there:

`worktree.guessRemote`
If no branch is specified and neither `-b` nor `-B` nor `--detach` is used, then `git worktree add` defaults to creating a new branch from HEAD. If `worktree.guessRemote` is set to true, `worktree add` tries to find a remote-tracking branch whose name uniquely matches the new branch name. If such a branch exists, it is checked out and set as "upstream" for the new branch. If multiple matches are found, the command fails. If no such match can be found, it falls back to creating a new branch from the current `HEAD`.

`worktree.useRelativePaths`
Link worktrees using relative paths (when “true”) or absolute paths (when “false”). This is particularly useful for setups where the repository and worktrees may be moved between different locations or environments. Defaults to “false”.

Note that setting `worktree.useRelativePaths` to “true” implies enabling the `extensions.relativeWorktrees` config (see linkgit:git-config[1]), thus making it incompatible with older versions of Git.

BUGS ---- Multiple checkout in general is still experimental, and the support for submodules is incomplete. It is NOT recommended to make multiple checkouts of a superproject.

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

□□□ #1
□ □□□ □□□ 2026-10-31 20:13:40 UTC
□ □□□ □□□ 2026-10-31 20:13:40 UTC