NAME

git-worktree − Manage multiple working trees

SYNOPSIS

git worktree add [−f] [−−detach] [−−checkout] [−−lock] [−b <new−branch>] <path> [<commit−ish>]
git worktree list [−−porcelain]
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 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. This new working tree is called a "linked working tree" as opposed to the "main working tree" prepared by "git init" or "git clone". A repository has one main working tree (if it's not a bare repository) and zero or more linked working trees. When you are done with a linked working tree, remove it with **git worktree remove**.

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 **git-config**(1)), or you can run **git worktree prune** in the main or any linked working tree to clean up any stale administrative files.

If a linked working tree 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 working tree is locked.

COMMANDS

add <path> [<commit−ish>]

Create **<path>** and checkout **<commit−ish>** into it. The new working directory is linked to the current repository, sharing everything except working directory specific files such as HEAD, index, etc. **−** may also be specified as **<commit−ish>**; it 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 **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).

list

List details of each worktree. The main worktree is listed first, followed by each of the linked

worktrees. The output details include if the worktree is bare, the revision currently checked out, and the branch currently checked out (or *detached HEAD* if none).

lock
: If a working tree 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 working tree to a new location. Note that the main working tree or linked working trees containing submodules cannot be moved.

prune
: Prune working tree information in $GIT_DIR/worktrees.

remove
: Remove a working tree. Only clean working trees (no untracked files and no modification in tracked files) can be removed. Unclean working trees or ones with submodules can be removed with **−−force**. The main working tree cannot be removed.

unlock
: Unlock a working tree, allowing it to be pruned, moved or deleted.

OPTIONS

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

  move refuses to move a locked working tree unless **−−force** is specified twice.

  remove refuses to remove an unclean working tree unless **−−force** is used. To remove a locked working tree, 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 working tree. 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>**.

−−detach
: With **add**, detach HEAD in the new working tree. See "DETACHED HEAD" in **git-checkout**(1).

−−[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 **git-read-tree**(1).

−−[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.

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

−−[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 **git-branch**(1) for details.

−−lock
: Keep the working tree locked after creation. This is the equivalent of **git worktree lock** after **git worktree add**, but without race condition.

−n, −−dry−run
: With **prune**, do not remove anything; just report what it would remove.

−−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. See below for details.

−q, −−quiet
: With *add*, suppress feedback messages.

−v, −−verbose
: With **prune**, report all removals.

−−expire <time>
: With **prune**, only expire unused working trees older than <time>.

−−reason <string>
: With **lock**, an explanation why the working tree is locked.

<worktree>
: Working trees can be identified by path, either relative or absolute.

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

REFS

In multiple working trees, some refs may be shared between all working trees, some refs are local. One example is HEAD is different for all working trees. This section is about the sharing rules and how to access refs of one working tree from another.

In general, all pseudo refs are per working tree 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 one exception to this: refs inside refs/bisect and refs/worktree is not shared.

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

For example, main−worktree/HEAD or main−worktree/refs/bisect/good resolve to the same value as the main working tree'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 **git-rev-parse**(1) or **git-update-ref**(1) which will handle refs correctly.

CONFIGURATION FILE

By default, the repository "config" file is shared across all working trees. If the config variables **core.bare** or **core.worktree** are already present in the config file, they will be applied to the main working trees only.

In order to have configuration specific to working trees, you can turn on "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 you have them in $GIT_DIR/config before, you must move them to the **config.worktree** of the main working tree. You may also take this opportunity to review and move other configuration that you do not want to share to all working trees:

- **core.worktree** and **core.bare** should never be shared
- **core.sparseCheckout** is recommended per working tree, unless you are sure you always use sparse checkout for all working trees.

DETAILS

Each linked working tree has a private sub−directory in the repository's $GIT_DIR/worktrees directory. The private sub−directory's name is usually the base name of the linked working tree's path, possibly appended with a number to make it unique. For example, when **$GIT_DIR=/path/main/.git** the command **git worktree add /path/other/test−next next** creates the linked working tree 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 working tree, $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 working tree's $GIT_DIR (e.g. **/path/main/.git**). These settings are made in a **.git** file located at the top directory of the linked working tree.

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 working tree **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 working trees, except refs/bisect and refs/worktree.

See **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 working tree, you need to update the *gitdir* file in the entry's directory. For example, if a linked working tree 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.

To prevent a $GIT_DIR/worktrees entry from being pruned (which can be useful in some situations, such as when the entry's working tree 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 working tree'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 **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)
```

Porcelain Format

The porcelain format has a line per attribute. 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 only present if and only if the value is true. 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
```

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 **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 working tree 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
```

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 **git**(1) suite