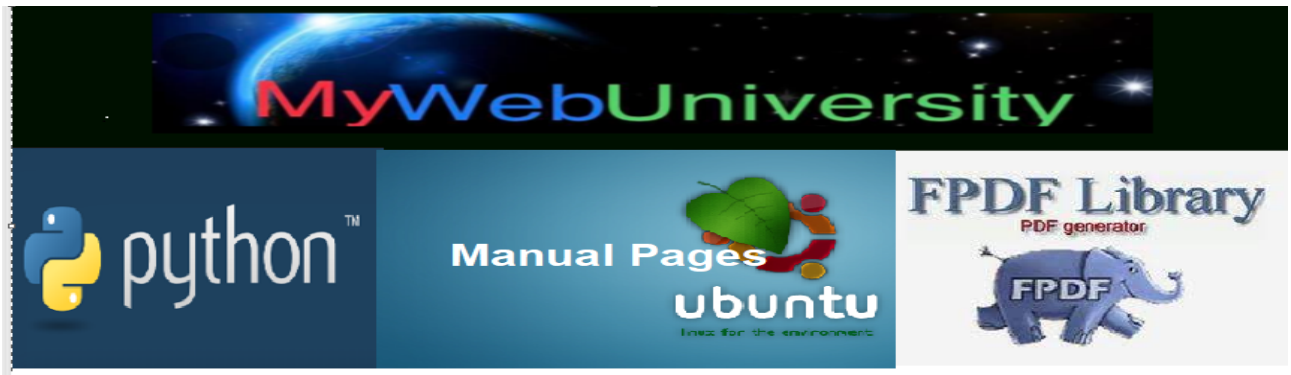

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 Manual Pages on command 'git-read-tree.1'

$ man git-read-tree.1

GIT-READ-TREE(1) Git Manual GIT-READ-TREE(1)

NAME

git-read-tree - Reads tree information into the index

SYNOPSIS

git read-tree [(-m [--trivial] [--aggressive] | --reset | --prefix=<prefix>)

[-u | -i]] [--index-output=<file>] [--no-sparse-checkout]

(--empty | <tree-ish1> [<tree-ish2> [<tree-ish3>]])

DESCRIPTION

Reads the tree information given by <tree-ish> into the index, but does not actually update any of the files it "caches". (see: git-checkout-index(1))

Optionally, it can merge a tree into the index, perform a fast-forward (i.e. 2-way) merge, or a 3-way merge, with the -m flag. When used with -m, the -u flag causes it to also update the files in the work tree with the result of the merge.

Only trivial merges are done by git read-tree itself. Only conflicting paths will be in an unmerged state when git read-tree returns.

OPTIONS

-m

Perform a merge, not just a read. The command will refuse to run if your index file has unmerged entries, indicating that you have not finished a previous merge you started.

--reset

Same as -m, except that unmerged entries are discarded instead of failing. When used with -u, updates leading to loss of working tree changes or untracked files or directories will not abort the operation.

-u

After a successful merge, update the files in the work tree with the result of the merge.

-i

Usually a merge requires the index file as well as the files in the working tree to be up to date with the current head commit, in order not to lose local changes. This flag disables the check with the working tree and is meant to be used when creating a merge of trees that are not directly related to the current working tree status into a temporary index file.

-n, --dry-run

Check if the command would error out, without updating the index or the files in the working tree for real.

-v

Show the progress of checking files out.

--trivial

Restrict three-way merge by git read-tree to happen only if there is no file-level merging required, instead of resolving merge for trivial cases and leaving conflicting files unresolved in the index.

--aggressive

Usually a three-way merge by git read-tree resolves the merge for really trivial cases and leaves other cases unresolved in the index, so that porcelains can implement different merge policies. This flag makes the command resolve a few more cases internally:

- when one side removes a path and the other side leaves the path unmodified. The resolution is to remove that path.
- when both sides remove a path. The resolution is to remove that path.
- when both sides add a path identically. The resolution is to add that path.

--prefix=<prefix>

Keep the current index contents, and read the contents of the named tree-ish under the directory at <prefix>. The command will refuse to overwrite entries that already existed in the original index file.

--index-output=<file>

Instead of writing the results out to $GIT_INDEX_FILE, write the resulting index in the

named file. While the command is operating, the original index file is locked with the same mechanism as usual. The file must allow to be rename(2)ed into from a temporary file that is created next to the usual index file; typically this means it needs to be on the same filesystem as the index file itself, and you need write permission to the directories the index file and index output file are located in.

--recurse-submodules, --no-recurse-submodules

Using --recurse-submodules will update the content of all active submodules according to the commit recorded in the superproject by calling read-tree recursively, also setting the submodules' HEAD to be detached at that commit.

--no-sparse-checkout

Disable sparse checkout support even if core.sparseCheckout is true.

--empty

Instead of reading tree object(s) into the index, just empty it.

-q, --quiet

Quiet, suppress feedback messages.

<tree-ish#>

The id of the tree object(s) to be read/merged.

MERGING

If -m is specified, git read-tree can perform 3 kinds of merge, a single tree merge if only 1 tree is given, a fast-forward merge with 2 trees, or a 3-way merge if 3 or more trees are provided.

Single Tree Merge

If only 1 tree is specified, git read-tree operates as if the user did not specify -m, except that if the original index has an entry for a given pathname, and the contents of the path match with the tree being read, the stat info from the index is used. (In other words, the index?s stat()s take precedence over the merged tree?s).

That means that if you do a git read-tree -m <newtree> followed by a git checkout-index -f -u -a, the git checkout-index only checks out the stuff that really changed.

This is used to avoid unnecessary false hits when git diff-files is run after git read-tree.

Two Tree Merge

Typically, this is invoked as git read-tree -m $H $M, where $H is the head commit of the current repository, and $M is the head of a foreign tree, which is simply ahead of $H (i.e. we are in a fast-forward situation).

When two trees are specified, the user is telling git read-tree the following:

1. The current index and work tree is derived from $H, but the user may have local changes in them since $H.
2. The user wants to fast-forward to $M.

In this case, the git read-tree -m $H $M command makes sure that no local change is lost as the result of this "merge". Here are the "carry forward" rules, where "I" denotes the index, "clean" means that index and work tree coincide, and "exists"/"nothing" refer to the presence of a path in the specified commit:

	I	H	M	Result
0	nothing	nothing	nothing	(does not happen)
1	nothing	nothing	exists	use M
2	nothing	exists	nothing	remove path from index
3	nothing	exists	exists,	use M if "initial checkout", H == M keep index otherwise
				exists, fail
				H != M
	clean I==H	I==M		
4	yes	N/A	N/A	nothing nothing keep index
5	no	N/A	N/A	nothing nothing keep index
6	yes	N/A	yes	nothing exists keep index
7	no	N/A	yes	nothing exists keep index
8	yes	N/A	no	nothing exists fail
9	no	N/A	no	nothing exists fail
10	yes	yes	N/A	exists nothing remove path from index
11	no	yes	N/A	exists nothing fail
12	yes	no	N/A	exists nothing fail
13	no	no	N/A	exists nothing fail
	clean (H==M)			
14	yes		exists	exists keep index
15	no		exists	exists keep index

clean I==H I==M (H!=M)

---

16 yes no no exists exists fail

17 no no no exists exists fail

18 yes no yes exists exists keep index

19 no no yes exists exists keep index

20 yes yes no exists exists use M

21 no yes no exists exists fail

In all "keep index" cases, the index entry stays as in the original index file. If the entry is not up to date, git read-tree keeps the copy in the work tree intact when operating under the -u flag.

When this form of git read-tree returns successfully, you can see which of the "local changes" that you made were carried forward by running git diff-index --cached $M. Note that this does not necessarily match what git diff-index --cached $H would have produced before such a two tree merge. This is because of cases 18 and 19 ? if you already had the changes in $M (e.g. maybe you picked it up via e-mail in a patch form), git diff-index --cached $H would have told you about the change before this merge, but it would not show in git diff-index --cached $M output after the two-tree merge.

Case 3 is slightly tricky and needs explanation. The result from this rule logically should be to remove the path if the user staged the removal of the path and then switching to a new branch. That however will prevent the initial checkout from happening, so the rule is modified to use M (new tree) only when the content of the index is empty. Otherwise the removal of the path is kept as long as $H and $M are the same.

3-Way Merge

Each "index" entry has two bits worth of "stage" state. stage 0 is the normal one, and is the only one you?d see in any kind of normal use.

However, when you do git read-tree with three trees, the "stage" starts out at 1.

This means that you can do

$ git read-tree -m <tree1> <tree2> <tree3>

and you will end up with an index with all of the <tree1> entries in "stage1", all of the <tree2> entries in "stage2" and all of the <tree3> entries in "stage3". When performing a merge of another branch into the current branch, we use the common ancestor tree as <tree1>, the current branch head as <tree2>, and the other branch head as <tree3>.

Furthermore, git read-tree has special-case logic that says: if you see a file that matches in all respects in the following states, it "collapses" back to "stage0":

? stage 2 and 3 are the same; take one or the other (it makes no difference - the same work has been done on our branch in stage 2 and their branch in stage 3)

? stage 1 and stage 2 are the same and stage 3 is different; take stage 3 (our branch in stage 2 did not do anything since the ancestor in stage 1 while their branch in stage 3 worked on it)

? stage 1 and stage 3 are the same and stage 2 is different take stage 2 (we did something while they did nothing)

The git write-tree command refuses to write a nonsensical tree, and it will complain about unmerged entries if it sees a single entry that is not stage 0.

OK, this all sounds like a collection of totally nonsensical rules, but it?s actually exactly what you want in order to do a fast merge. The different stages represent the "result tree" (stage 0, aka "merged"), the original tree (stage 1, aka "orig"), and the two trees you are trying to merge (stage 2 and 3 respectively).

The order of stages 1, 2 and 3 (hence the order of three <tree-ish> command-line arguments) are significant when you start a 3-way merge with an index file that is already populated.

Here is an outline of how the algorithm works:

? if a file exists in identical format in all three trees, it will automatically collapse to "merged" state by git read-tree.

? a file that has any difference what-so-ever in the three trees will stay as separate entries in the index. It?s up to "porcelain policy" to determine how to remove the non-0 stages, and insert a merged version.

? the index file saves and restores with all this information, so you can merge things incrementally, but as long as it has entries in stages 1/2/3 (i.e., "unmerged entries") you can?t write the result. So now the merge algorithm ends up being really simple:

  ? you walk the index in order, and ignore all entries of stage 0, since they?ve already been done.

  ? if you find a "stage1", but no matching "stage2" or "stage3", you know it?s been removed from both trees (it only existed in the original tree), and you remove that entry.

  ? if you find a matching "stage2" and "stage3" tree, you remove one of them, and turn the other into a "stage0" entry. Remove any matching "stage1" entry if it exists

too. .. all the normal trivial rules ..

You would normally use git merge-index with supplied git merge-one-file to do this last step. The script updates the files in the working tree as it merges each path and at the end of a successful merge.

When you start a 3-way merge with an index file that is already populated, it is assumed that it represents the state of the files in your work tree, and you can even have files with changes unrecorded in the index file. It is further assumed that this state is "derived" from the stage 2 tree. The 3-way merge refuses to run if it finds an entry in the original index file that does not match stage 2.

This is done to prevent you from losing your work-in-progress changes, and mixing your random changes in an unrelated merge commit. To illustrate, suppose you start from what has been committed last to your repository:

```
$ JC=`git rev-parse --verify "HEAD^0"`
$ git checkout-index -f -u -a $JC
```

You do random edits, without running git update-index. And then you notice that the tip of your "upstream" tree has advanced since you pulled from him:

```
$ git fetch git://.... linus
$ LT=`git rev-parse FETCH_HEAD`
```

Your work tree is still based on your HEAD ($JC), but you have some edits since. Three-way merge makes sure that you have not added or modified index entries since $JC, and if you haven?t, then does the right thing. So with the following sequence:

```
$ git read-tree -m -u `git merge-base $JC $LT` $JC $LT
$ git merge-index git-merge-one-file -a
$ echo "Merge with Linus" | \
  git commit-tree `git write-tree` -p $JC -p $LT
```

what you would commit is a pure merge between $JC and $LT without your work-in-progress changes, and your work tree would be updated to the result of the merge.

However, if you have local changes in the working tree that would be overwritten by this merge, git read-tree will refuse to run to prevent your changes from being lost.

In other words, there is no need to worry about what exists only in the working tree. When you have local changes in a part of the project that is not involved in the merge, your changes do not interfere with the merge, and are kept intact. When they do interfere, the merge does not even start (git read-tree complains loudly and fails without modifying

anything). In such a case, you can simply continue doing what you were in the middle of doing, and when your working tree is ready (i.e. you have finished your work-in-progress), attempt the merge again.

SPARSE CHECKOUT

Note: The skip-worktree capabilities in git-update-index(1) and read-tree predated the introduction of git-sparse-checkout(1). Users are encouraged to use the sparse-checkout command in preference to these plumbing commands for sparse-checkout/skip-worktree related needs. However, the information below might be useful to users trying to understand the pattern style used in non-cone mode of the sparse-checkout command.

"Sparse checkout" allows populating the working directory sparsely. It uses the skip-worktree bit (see git-update-index(1)) to tell Git whether a file in the working directory is worth looking at.

git read-tree and other merge-based commands (git merge, git checkout...) can help maintaining the skip-worktree bitmap and working directory update.

$GIT_DIR/info/sparse-checkout is used to define the skip-worktree reference bitmap. When git read-tree needs to update the working directory, it resets the skip-worktree bit in the index based on this file, which uses the same syntax as .gitignore files. If an entry matches a pattern in this file, or the entry corresponds to a file present in the working tree, then skip-worktree will not be set on that entry. Otherwise, skip-worktree will be set.

Then it compares the new skip-worktree value with the previous one. If skip-worktree turns from set to unset, it will add the corresponding file back. If it turns from unset to set, that file will be removed.

While $GIT_DIR/info/sparse-checkout is usually used to specify what files are in, you can also specify what files are not in, using negate patterns. For example, to remove the file unwanted:

```
/*
!unwanted
```

Another tricky thing is fully repopulating the working directory when you no longer want sparse checkout. You cannot just disable "sparse checkout" because skip-worktree bits are still in the index and your working directory is still sparsely populated. You should re-populate the working directory with the $GIT_DIR/info/sparse-checkout file content as follows:

/*

Then you can disable sparse checkout. Sparse checkout support in git read-tree and similar commands is disabled by default. You need to turn core.sparseCheckout on in order to have sparse checkout support.

SEE ALSO

git-write-tree(1), git-ls-files(1), gitignore(5), git-sparse-checkout(1)

GIT

Part of the git(1) suite