Rocky Enterprise Linux 9.2 Manual Pages on command 'gittutorial-2.7'

C:\>man gittutorial-2.7

GITTUTORIAL-2(7) Git Manual GITTUTORIAL-2(7)

NAME

gittutorial-2 - A tutorial introduction to Git: part two

SYNOPSIS

git *

DESCRIPTION

You should work through gittutorial(7) before reading this tutorial.

The goal of this tutorial is to introduce two fundamental pieces of Git?s architecture?the object database and the index file?and to provide the reader with everything necessary to understand the rest of the Git documentation.

THE GIT OBJECT DATABASE

Let?s start a new project and create a small amount of history:

```
$ mkdir test-project
$ cd test-project
$ git init
Initialized empty Git repository in .git/
$ echo 'hello world' > file.txt
$ git add .
$ git commit -a -m "initial commit"
[master (root-commit) 54196cc] initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 file.txt
```

```
$ echo 'hello world!' >file.txt
$ git commit -a -m "add emphasis"
[master c4d59f3] add emphasis
 1 file changed, 1 insertion(+), 1 deletion(-)
```

What are the 7 digits of hex that Git responded to the commit with?

We saw in part one of the tutorial that commits have names like this. It turns out that every object in the Git history is stored under a 40-digit hex name. That name is the SHA-1 hash of the object?s contents; among other things, this ensures that Git will never store the same data twice (since identical data is given an identical SHA-1 name), and that the contents of a Git object will never change (since that would change the object?s name as well). The 7 char hex strings here are simply the abbreviation of such 40 character long strings. Abbreviations can be used everywhere where the 40 character strings can be used, so long as they are unambiguous.

It is expected that the content of the commit object you created while following the example above generates a different SHA-1 hash than the one shown above because the commit object records the time when it was created and the name of the person performing the commit.

We can ask Git about this particular object with the cat-file command. Don?t copy the 40 hex digits from this example but use those from your own version. Note that you can shorten it to only a few characters to save yourself typing all 40 hex digits:

```
$ git cat-file -t 54196cc2
commit
$ git cat-file commit 54196cc2
tree 92b8b694ffb1675e5975148e1121810081dbdffe
author J. Bruce Fields <bfields@puzzle.fieldses.org> 1143414668 -0500
committer J. Bruce Fields <bfields@puzzle.fieldses.org> 1143414668 -0500
initial commit
```

A tree can refer to one or more "blob" objects, each corresponding to a file. In addition, a tree can also refer to other tree objects, thus creating a directory hierarchy. You can examine the contents of any tree using ls-tree (remember that a long enough initial portion of the SHA-1 will also work):

```
$ git ls-tree 92b8b694
100644 blob 3b18e512dba79e4c8300dd08aeb37f8e728b8dad    file.txt
```

Thus we see that this tree has one file in it. The SHA-1 hash is a reference to that file?s data:

```
$ git cat-file -t 3b18e512
blob
```

A "blob" is just file data, which we can also examine with cat-file:

```
$ git cat-file blob 3b18e512
hello world
```

Note that this is the old file data; so the object that Git named in its response to the initial tree was a tree with a snapshot of the directory state that was recorded by the first commit.

All of these objects are stored under their SHA-1 names inside the Git directory:

```
$ find .git/objects/
.git/objects/
.git/objects/pack
.git/objects/info
.git/objects/3b
.git/objects/3b/18e512dba79e4c8300dd08aeb37f8e728b8dad
.git/objects/92
.git/objects/92/b8b694ffb1675e5975148e1121810081dbdffe
.git/objects/54
.git/objects/54/196cc2703dc165cbd373a65a4dcf22d50ae7f7
.git/objects/a0
.git/objects/a0/423896973644771497bdc03eb99d5281615b51
.git/objects/d0
.git/objects/d0/492b368b66bdabf2ac1fd8c92b39d3db916e59
.git/objects/c4
.git/objects/c4/d59f390b9cfd4318117afde11d601c1085f241
```

and the contents of these files is just the compressed data plus a header identifying their length and their type. The type is either a blob, a tree, a commit, or a tag.

The simplest commit to find is the HEAD commit, which we can find from .git/HEAD:

```
$ cat .git/HEAD
ref: refs/heads/master
```

As you can see, this tells us which branch we?re currently on, and it tells us this by naming a file under the .git directory, which itself contains a SHA-1 name referring to a commit object, which we can examine with cat-file:

```
$ cat .git/refs/heads/master
c4d59f390b9cfd4318117afde11d601c1085f241
$ git cat-file -t c4d59f39
commit
$ git cat-file commit c4d59f39
tree d0492b368b66bdabf2ac1fd8c92b39d3db916e59
parent 54196cc2703dc165cbd373a65a4dcf22d50ae7f7
author J. Bruce Fields <bfields@puzzle.fieldses.org> 1143418702 -0500
committer J. Bruce Fields <bfields@puzzle.fieldses.org> 1143418702 -0500
add emphasis
```

The "tree" object here refers to the new state of the tree:

```
$ git ls-tree d0492b36
100644 blob a042389697364477 1497bdc03eb99d5281615b51    file.txt
$ git cat-file blob a0423896
hello world!
```

and the "parent" object refers to the previous commit:

```
$ git cat-file commit 54196cc2
tree 92b8b694ffb1675e5975148e1121810081dbdffe
author J. Bruce Fields <bfields@puzzle.fieldses.org> 1143414668 -0500
committer J. Bruce Fields <bfields@puzzle.fieldses.org> 1143414668 -0500
initial commit
```

The tree object is the tree we examined first, and this commit is unusual in that it lacks any parent.

Most commits have only one parent, but it is also common for a commit to have multiple parents. In that case the commit represents a merge, with the parent references pointing to the heads of the merged branches.

Besides blobs, trees, and commits, the only remaining type of object is a "tag", which we won?t discuss here; refer to git-tag(1) for details.

So now we know how Git uses the object database to represent a project?s history:

? "commit" objects refer to "tree" objects representing the snapshot of a directory tree at a particular point in the history, and refer to "parent" commits to show how they?re connected into the project history.

? "tree" objects represent the state of a single directory, associating directory names to "blob" objects containing file data and "tree" objects containing subdirectory information.

? "blob" objects contain file data without any other structure.

? References to commit objects at the head of each branch are stored in files under .git/refs/heads/.

? The name of the current branch is stored in .git/HEAD.

Note, by the way, that lots of commands take a tree as an argument. But as we can see above, a tree can be referred to in many different ways?by the SHA-1 name for that tree, by the name of a commit that refers to the tree, by the name of a branch whose head refers to that tree, etc.--and most such commands can accept any of these names.

In command synopses, the word "tree-ish" is sometimes used to designate such an argument.

THE INDEX FILE

The primary tool we?ve been using to create commits is git-commit -a, which creates a commit including every change you?ve made to your working tree. But what if you want to commit changes only to certain files? Or only certain changes to certain files?

If we look at the way commits are created under the cover, we?ll see that there are more flexible ways creating commits.

Continuing with our test-project, let?s modify file.txt again:

```
$ echo "hello world, again" >>file.txt
```

but this time instead of immediately making the commit, let?s take an intermediate step, and ask for diffs along the way to keep track of what?s happening:

```
$ git diff
--- a/file.txt
+++ b/file.txt
@@ -1 +1,2 @@
```

```
 hello world!
+hello world, again
$ git add file.txt
$ git diff
```

The last diff is empty, but no new commits have been made, and the head still doesn?t contain the new line:

```
$ git diff HEAD
diff --git a/file.txt b/file.txt
index a042389..513feba 100644
--- a/file.txt
+++ b/file.txt
@@ -1 +1,2 @@
 hello world!
+hello world, again
```

So git diff is comparing against something other than the head. The thing that it?s comparing against is actually the index file, which is stored in .git/index in a binary format, but whose contents we can examine with ls-files:

```
$ git ls-files --stage
100644 513feba2e53ebbd2532419ded848ba19de88ba00 0       file.txt
$ git cat-file -t 513feba2
blob
$ git cat-file blob 513feba2
hello world!
hello world, again
```

So what our git add did was store a new blob and then put a reference to it in the index file. If we modify the file again, we?ll see that the new modifications are reflected in the git diff output:

```
$ echo 'again?' >>file.txt
$ git diff
index 513feba..ba3da7b 100644
--- a/file.txt
+++ b/file.txt
@@ -1,2 +1,3 @@
```

```
 hello world!
 hello world, again
+again?
```

With the right arguments, git diff can also show us the difference between the working directory and the last commit, or between the index and the last commit:

```
$ git diff HEAD
diff --git a/file.txt b/file.txt
index a042389..ba3da7b 100644
--- a/file.txt
+++ b/file.txt
@@ -1 +1,3 @@
 hello world!
+hello world, again
+again?
$ git diff --cached
diff --git a/file.txt b/file.txt
index a042389..513feba 100644
--- a/file.txt
+++ b/file.txt
@@ -1 +1,2 @@
 hello world!
+hello world, again
```

At any time, we can create a new commit using git commit (without the "-a" option), and verify that the state committed only includes the changes stored in the index file, not the additional change that is still only in our working tree:

```
$ git commit -m "repeat"
$ git diff HEAD
diff --git a/file.txt b/file.txt
index 513feba..ba3da7b 100644
--- a/file.txt
+++ b/file.txt
@@ -1,2 +1,3 @@
 hello world!
```

```
 hello world, again
+again?
```

So by default git commit uses the index to create the commit, not the working tree; the "-a" option to commit tells it to first update the index with all changes in the working tree.

Finally, it?s worth looking at the effect of git add on the index file:

```
$ echo "goodbye, world" >closing.txt
$ git add closing.txt
```

The effect of the git add was to add one entry to the index file:

```
$ git ls-files --stage
100644 8b9743b20d4b15be3955fc8d5cd2b09cd2336138 0       closing.txt
100644 513feba2e53ebbd2532419ded848ba19de88ba00 0       file.txt
```

And, as you can see with cat-file, this new entry refers to the current contents of the file:

```
$ git cat-file blob 8b9743b2
goodbye, world
```

The "status" command is a useful way to get a quick summary of the situation:

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   closing.txt
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file.txt
```

Since the current state of closing.txt is cached in the index file, it is listed as "Changes to be committed". Since file.txt has changes in the working directory that aren?t reflected in the index, it is marked "changed but not updated". At this point, running "git commit" would create a commit that added closing.txt (with its new contents), but that didn?t modify file.txt.

Also, note that a bare git diff shows the changes to file.txt, but not the addition of closing.txt, because the version of closing.txt in the index file is identical

to the one in the working directory.

In addition to being the staging area for new commits, the index file is also populated from the object database when checking out a branch, and is used to hold the trees involved in a merge operation. See gitcore-tutorial(7) and the relevant man pages for details.

WHAT NEXT?

At this point you should know everything necessary to read the man pages for any of the git commands; one good place to start would be with the commands mentioned in giteveryday(7). You should be able to find any unknown jargon in gitglossary(7).

The Git User?s Manual[1] provides a more comprehensive introduction to Git.

gitcvs-migration(7) explains how to import a CVS repository into Git, and shows how to use Git in a CVS-like way.

For some interesting examples of Git use, see the howtos[2].

For Git developers, gitcore-tutorial(7) goes into detail on the lower-level Git mechanisms involved in, for example, creating a new commit.

SEE ALSO

gittutorial(7), gitcvs-migration(7), gitcore-tutorial(7), gitglossary(7), git-help(1), giteveryday(7), The Git User?s Manual[1]

GIT

Part of the git(1) suite

NOTES

1. Git User?s Manual

   file:///usr/share/doc/git/html/user-manual.html

2. howtos

   file:///usr/share/doc/git/html/howto-index.html