How To Git It

**Git manages the Linux kernel source.
Chances are it can manage your code, too**

By Jon Loeliger

G*it* (http://www.kernel.org/git/) is a source control management (*SCM*) system especially adept at sharing large amounts of source code among a vastly distributed group of developers. Initially and specifically created by Linus Torvalds to manage the *Linux* kernel sources (read the gory details in Sam Williams's "Git With It," available online at http://www.linux-mag.com/2005-08/git/), git has quickly been adopted by a large number of projects. Recently, *git 1.0* was released — a perfect opportunity for you to try the software. After all, if it works for Linus (and Andrew and Marco), it'll probably work for you, too.

Like some other source control management tools, *git* supports a fully distributed development model: for a given source code base, there's no necessity to maintain a sole "master" repository. While a project might publish and maintain a master by convention, *git* doesn't require or enforce it. In fact, *git* is fully *peer-to-peer,* allowing an individual developer to base his or her repository on one or more remote repositories, and vice versa, allowing a programmer to publish his or her repository for other developers to use.

The peer-to-peer relationship between repositories essentially dictated a design that allowed any developer to be able to both *push* and *pull* their repositories to remote locations. ("Remote location" is a bit of a misnomer or at least an understatement: *git* supports references to repositories on any network machine or on a local file system through a variety of URL specifications.)

Furthermore, given the peer-to-peer relationship of related repositories, directly supporting very complex merges on very large code bases was an early design goal in *git*. *git's* merge process tries different algorithms, starting with a very fast, yet stupid approach, progressing to alternate algorithms that are more complex but more time consuming. This variety of merge strategies ensures a fast, yet successful merge for a wide variety of use cases. Most developers who track an external repository are able to gain significant savings with *git's* staged merge approach.

Ultimately, *git* supports an arbitrary graph of repository references. *git* is able to recognize duplicates and handle both duplicate changes and identical changes that might show up in one repository through alternate paths of the repository graph.

It's All About the Objects

One of the major differences between *git* and many other SCM systems is that *git doesn't* track files. Instead, it tracks a *collection of file contents*. Thus, *git* supports *change sets,* a related set of changes to multiple files, including permissions, file locations, and an associated change log message, all treated as one atomic unit. A change set that affects multiple files can't be separated to affect a subset of the files, and whether a change set modifies one file or multiple files at once doesn't matter.

At the heart of *git* is the *Object Database,* a collection of content-addressable data and the *Index,* which contains a directory structure for the objects. The core *git* commands manipulate the Object Database and the Index without regard to the actual content of the files, or for that matter, very little regard for the location of the file content. Data in the Object Database can be extracted and presented to the user as a traditional, hierarchical directory of files.

A true content-addressable object store for something with content as loosely defined as "a file" isn't very practical. Using the actual, complete content of a file as the name of the file would become unwieldy and, frankly, redundant. Instead, to obtain an approximation of the data's content, *git* uses the *Secure Hash Algorithm 1* (SHA1, http://www.faqs.org/rfcs/rfc3174.html). An SHA1 is 160-bits of data that *git* represents as a 40-byte hexadecimal value.

git currently manages four "types" of objects in its database: a *blob,* a *tree,* a *commit,* and a *tag*. Each of these object types contains different data, but all are stored in the Object Database, indexed and accessible by their SHA1 names. If

the data in an object changes, it is considered a different object with a different SHA1 name. Objects are immutable.

The blob object is the simplest object, representing a pure collection of data without references to any other object. In other words, a blob is a file — not the file from the file system itself, but rather the contents of one version of that file. In fact, two byte-wise identical files located in different directories of the same repository will have just one blob object and just one SHA1 name to reference them. The blob content is totally independent of its location or its permissions within the repository. Effectively, blobs form the leaf nodes of a larger data structure constructed from the other object types.

A tree object is used to group other blob and tree objects into a hierarchical directory structure. It contains a sorted list of *(mode, object type, referenced object SHA1, path name)* tuples. The modes are file system permissions; object types are either blob or tree and a corresponding referenced "blob" or "tree" object; and finally, the file system directory name or file name for the entry. A tree object represents just one directory and thus one level in the overall hierarchy. Again, all of these entries are interpreted as one big chunk of data whose SHA1 is computed and placed in the object database.

A tree object just provides structure. Furthermore, there's no notion of change or history represented by either blob or tree objects. Instead, the commit object combines the current tree state together with the previous, parent commit objects that led to this state, and a user-supplied log message. The sequence of commits forms an annotated *directed acyclic graph* (DAG) of complete source tree revisions. Naturally, there is at least one root commit object that has no parent commit, obtained from the initial commit.

Without the commit object, there'd be no notion of nor any other way to represent the history and changes within a repository. It's the sequence of commits that forms a progression of changes and development within a repository. Various tools can traverse the *commit graph* and extract information such as change logs, revision history, and common merge-base ancestry.

The final object type is a tag, used to assign a symbolic name to a particular object reference, typically the commit associated with a named release. For integrity and trust, a tag object can be digitally signed with *PGP*.

Keeping Track of It All

git maintains a mutable, virtual directory structure along with meta-data such as time-stamps and permissions in an *Index*. Historically, the Index was called the *Cache* as many of the operations on the Index manipulated it as if it was a write-through cache. However, as its purpose was refined with implementation, it was renamed the Index.

For *git* to properly distinguish file content that should be tracked by the SCM from non-tracked files, the Index is instructed what files are important and where the files are in the directory structure. Incidental information, such as permissions and time stamps, are automatically collected or inspected from the underlying file system when needed.

The contents of the Index can be thought of as describing a tree object with a bit more supporting information. Adding files to the repository effectively involves two steps: adding the file contents to the file system in the directory structure managed by the repository, and altering the Index to track the file. Similarly, removing a file from the repository requires two tasks: removing the file from the file system, and removing the file's entry from the Index. In both cases, these two steps are not necessarily directly linked. Part of *git*'s power comes from separating these two steps and allowing arbitrary manipulation of either the file system or the Index and then synchronizing them again.

> *git* supports a fully-distributed development model: There's no need for a sole "master" repository

Essentially, the Index acts as a staging-ground for collecting changes and directory structure for past, present, or future tree objects. These changes can come from direct manipulation of files as described above, or they can come from larger scale operations such as branch merging or fetching from other repositories. Once the Index has been changed, though, its current state can be captured, converted to a tree object, and placed into the Object Database.

To obtain any particular revision of a file or an entire directory structure, the reverse process is performed. The correct tree object is identified, read from the object database, and placed in the index. To realize the file contents in the working directory structure, the file system must be synchronized with the Index — that is, the Index dictates what blob objects must be extracted from the object store and placed into the users working directory hierarchy. Furthermore, any *other* files known to the Index within the working directory must be removed for the Index and the working directory to be fully synchronized.

The Index also plays a critical role during the important process of merging. When a tree object is read into the Index, it can either replace the entire contents of the Index, as described above, or it can be *merged* into the Index. There are fairly complex merge resolution rules that are detailed in the *git-read-tree* and *git-merge* manual pages, and each is designed to provide an intuitive, non-destructive merge that is likely to succeed, unless there are direct, conflicting changes that require human intervention to resolve.

Managing Merges

So, where do all of these changes come from? Feel free to make changes in the kernel yourself, or, with any luck, someone else will help and supply the location of a published repository from which you can *fetch* changes. To fetch changes, all that's needed is a reference to a repository.

git supports several remote repository names and transport protocols. For example, all of these forms are valid repository references:

```
rsync://example.com/path/to/repo.git/
http://example.com/path/to/repo.git/
https://example.com/path/to/repo.git/
git://example.com/path/to/repo.git/
ssh://example.com/path/to/repo.git/
/path/to/repo.git/
```

The last reference, the path name, can be used to reference a repository on the same host. As an interesting special case, `.` ("dot") can be used to fetch changes from the very same repository, likely from an alternate branch.

> In the long-standing tradition of *UNIX* tool philosophy, *git* provides a large suite of fairly simple, yet powerful tools

Again, keeping with *git*'s goal to be flexible, the *fetch* operation can be combined with an explicit *merge* operation called *pull*. While not quite entirely symmetric, the *push* operation can be used to publish changes in one repository directly to another repository.

Getting git

git can be built and installed on your system in a variety of ways. It is fairly portable, and currently runs on Linux, *xBSD*, *Mac OS X*, *Cygwin*, and *Solaris*. While there are several pre-built binary packages available, *git* is easily built from source, too.

Unless it is already installed on your system, though, you'll have to bootstrap an install from somewhere. Start with the most recent *tarball* from http://www.kernel.org/pub/software/scm/git/, place it in your favorite build directory, and unpack it:

```
$ cd /tmp
$ wget \
http://www.kernel.org/pub/software/scm/git/
git-1.0.7.tar.gz
$ tar xvzf git-1.0.7.tar.gz
$ cd git-1.0.7
```

Read the *README*, and follow the instructions in the *INSTALL* file. Here's what the latter says:

```
$ make prefix=/usr          ;# as yourself
# make prefix=/usr install ;# as root
```

Given `prefix=/usr`, *git* is installed into */usr/bin/*. If you plan to install into a different `prefix=` path, build the code (the first *make* command) with the same `prefix=`. (If you do not provide a `prefix`, *git* is installed in `$HOME` / *bin/*.)

There are several packages that are prerequisites for a successful build, and several build options allow for available package variations as well. These packages and alternates are called out in the *INSTALL* and top-level *Makefile* files.

Using git

In the long-standing tradition of *UNIX* tool philosophy, *git* provides a large suite of fairly simple, yet powerful tools that can be coordinated and combined with each other and with other traditional commands. Covering them all is beyond the scope of this article, but many of the key concepts discussed earlier can be shown with just a few of the basic commands.

While it may seem somewhat self-referential and even unnecessary since you've already obtained *git*, here's how to use the `git clone` command to obtain the current, development sources directly from the master *git* repository:

```
$ cd /tmp
$ git clone \
git://git.kernel.org/pub/scm/git/git.git
$ cd /tmp/git
```

If you run the command, you should now have a real *git* repository located in */tmp/git/*. (The top-level directory is fairly large, so it will abbreviated to the first several entries here and throughout the rest of the examples.)

You can identify a *git* repository because it has a hidden sub-directory called *.git* that contains both the object database, *.git/objects*, the index, *.git/index*, and several other files. For example, the URL used to clone the repository is kept in *.git/remotes/origin*:

```
$ cat .git/remotes/origin
URL:
git://git.kernel.org/pub/scm/git/git.git
Pull: master:origin
Pull: pu:pu
Pull: man:man
```

```
Pull: todo:todo
Pull: maint:maint
Pull: html:html
```

The `URL:` line specifies the remote repository that forms the basis of the clone operation. All of the `Pull:` lines specify *reference specifications* and describe the mapping between remote branches and local branches. (You can find details in the the manual page for *git-pull* or *git-fetch*.)

The *.git/HEAD* file contains the SHA1 of an object that is considered the current top, or *HEAD*, of the active development branch. Normally, after a *git clone* operation, it is the HEAD of the *master* branch:

```
$ ls -lsa .git/HEAD
0 lrwxrwxrwx  1 jdl jdl 17 Jan  8 14:06
  .git/HEAD -> refs/heads/master

$ cat .git/refs/heads/master
026351a03507dc3a2e89e068c01234dc55914df2
```

The `git cat-file -t` command can be used to determine the object type of any object reference:

```
$ git cat-file -t \
  026351a03507dc3a2e89e068c01234dc55914df2
commit
```

It's a `commit` object. So, what does that really look like? The *git cat-file commit* command extracts the given object out of the object store, treats it like a commit object, and dumps it:

```
$ git cat-file commit \
  026351a03507dc3a2e89e068c01234dc55914df2
tree
4e1ac5c4269f2e6e568390cc6128a88c2f924cbe
parent
5df466c507ee2dd81c2e9002c3fedf3536cde0dc
author John Ellson
<ellson@research.att.com> 1135959797 -0500
committer Junio C Hamano <junkio@cox.net>
1136523721 -0800

Make GIT-VERSION-GEN tolerate missing git
describe command

I think it is probably a bug that "git
non_existent_command" returns its error
message to stdout without an error, where
"git-non_existent_command" behaves differ-
ently and does return an error.

Older versions of git did not implement
"git describe" and GIT-VERSION-GEN pro-
duces an empty version string if run on a
system with such a git installed. The con-
sequence is that "make rpm" fails.

This patch fixes GIT-VERSION-GEN so that
it works in the absence of a working "git
describe"

Signed-off-by: John Ellson
<ellson@research.att.com>
Signed-off-by: Junio C Hamano
<junkio@cox.net>
```

The first line of output identifies the tree object that was captured from the index at the time the commit was performed. The prior, or *parent*, commit object is identified in the second line of output. While just one parent commit is listed here, it is possible that more than one parent contributed to the state of the repository leading to this point in time.

The tree object referenced above, `4e1ac5...` should somehow look like the state of the Index. But the Index, as described above, is a fairly complex data structured sorted by path name. In fact, it is stored as a binary file, and a special command `git ls-tree` is available to interpret its data. Note that *git* is, thankfully, usually pretty good about allowing shortened SHA1 references!

```
$ git ls-tree 4e1ac5
100644 blob 1a90…  .gitignore
100644 blob 6ff8…  COPYING
040000 tree 0a02…  Documentation
100755 blob 845b…  GIT-VERSION-GEN
100644 blob 916d…  INSTALL
100644 blob 1b6c…  Makefile
100644 blob cee7…  README
100644 blob c471…  apply.c
040000 tree 7316…  arm
100644 blob ea52…  blob.c
100644 blob ea5d…  blob.h
…
```

The output goes on and on, listing the entire contents of the *git* directory! Each line is one entry, and contains a tuple with mode information, an object type, the referenced object's SHA1, and the *path name* for the object within the directory structure. But also notice that *just* the *git* directory is listed; no sub-directory contents are listed. That's because each "tree" object represents just one level of the directory hierarchy.

The two types of objects present are `blob` and `tree`. For each tree object, the SHA1 references another tree recur-

sively one level deeper in the directory structure. The `Documentation` and `arm` are two such entries, and the contents of each can be obtained again using the `git ls-tree` command:

```
$ git ls-tree 0a0208 | head -3
100644 blob 9fef…  .gitignore
100644 blob a1ff…  Makefile
100644 blob 9ccb…  SubmittingPatches

$ git ls-tree 73167
100644 blob 11b1…  sha1.c
100644 blob 3952…  sha1.h
100644 blob da92…  sha1_arm.S
```

That was pretty painful. Luckily, the `-r` option requests a recursive listing, descending into sub-tree objects directly:

```
$ git ls-tree -r 4e1ac5 | head -5
100644 blob 1a90…  .gitignore
100644 blob 6ff8…  COPYING
100644 blob 9fef…  Documentation/.gitignore
100644 blob a1ff…  Documentation/Makefile
100644 blob 9ccb…
Documentation/SubmittingPatches
```

The other type of objects referenced by tree objects are blobs. Blobs are leaf nodes and represent raw file contents. As an example, the `INSTALL` blob is obtained like this:

```
$ git cat-file blob 916dd | head -10

Git installation

Normally you can just do "make" followed
by "make install", and that will install
the git programs in your own ~/bin/ direc-
tory.  If you want to do a global install,
you can do

$ make prefix=/usr ;# as yourself
# make prefix=/usr install ;# as root
```

That should look somewhat familiar!

Finally, tags are named and stored under *.git/refs/tags*. There are several of them in the *git* tree. Here is one:

```
$ cat .git/refs/tags/v1.0.7
c90d90…

$ git cat-file -t c90d90
tag
```

```
$ git cat-file tag c90d90
object 92e8…
type commit
tag v1.0.7
tagger Junio C Hamano <junkio@cox.net>
1136523576 -0800

GIT 1.0.7
——-BEGIN PGP SIGNATURE——-
Version: GnuPG v1.4.2 (GNU/Linux)

iD8D…
——-END PGP SIGNATURE——-
```

The sidebar" Peering into the git Object Database" illustrates the structure of the Object Database after a handful of operations.

Plumbing and Porcelain

Likely, all that looked daunting. Large numbers, weird references, and no intuition about how it is used on a daily basis.

Don't worry! Kick back, have a home-brew and revel in the knowledge that all those bits are at the very bottom layer of the *git* tool suite. Those details are like the plumbing in your house: With any luck, it remains hidden and it just works.

If the core *git* tool suite provided a "plumbing," then anything layered over that, providing a better user interface and experience, might be considered "porcelain." (Let's not take that analogy too much further, OK?) There are, however, several so-called pieces of "porcelain" that bear mention, investigation, and likely use.

Petr Baudis is the main pipe-fitter behind *Cogito* (http://www.kernel.org/pub/scm/cogito/cogito.git), a tool primarily designed to simplify the *git* interface and provide a more intuitive, traditional SCM feel. A key goal of Cogito, or *cg* as it's also called, is to abstract away the presence of the Index away from the user. If you're familiar with *CVS*, then *cg* may be a good starting point for you.

Stacked Git (http://www.procode.org/stgit/) spearheaded by Catalin Marinas, is a Python application that allows pushing and popping of patches on a *git* repository using a stack. In the end, the selected patches are stored as commit objects that are readily merged into a development branch. *stgit* has features similar to *quilt* (http://savannah.nongnu.org/projects/quilt). If you manage, manipulate, or coordinate a large number of patches for a repository, *stgit* might be for you.

If you visit http://www.kernel.org/git, review (http://www.kernel.org/pub/scm/git/gitweb.git). Written by Kay Sievers, *gitweb* is the essential tool for publishing an *HTTP*-accessible page of *git* repositories.

There are currently two graphical interfaces to *git* repositories. The first, *gitk*, written by Paul Mackerras, is part of the core *git* distribution. The other, *qgit* (http://sourceforge.net/projects/qgit), is maintained as an external SourceForge project by Marco Costalba. Both *gitk* and *qgit* provide a GUI that allows inspection of *git* repositories with views into the branches, logs, commits, and files. *qgit* also provides patch selection and application abilities.

Naturally, other pipe-fittings are available to help users convert repositories from other SCM systems to *git*. There are importers for CVS, *subversion*, and *arch*. Rumors have even circulated that other SCMs (for instance, *darcs*) could be written as a user front-end over top of *git*.

Furthermore, fine sparkling faucet handles in the form of *emacs* modes and *bash* command-line completions are also available. It's the expected Linux behavior, after all!

Plumbers and Pipe-fitters

There are thousands of commits in the *git* tree itself, with commits from perhaps 100 different individual contributors. Not all of the contributors can be named here.

However, the two plumbers who should be named are Linus Torvalds for instigating the whole *git* effort in the first place, and Junio Hamano, who shepherded the *git* tree through the majority of its development process.

Jon Loeliger currently works at Freescale Semiconductor in the Open Source group where he primarily helps develop Linux for PowerPC, yet occasionally strays into other Open Source projects. He has a sordid past developing vectorizing compilers, networks, 802.11 wireless boards, and the occasional game. In his spare time he's been known to brew, climb rocks, and contemplate pinot noir.

PEERING INTO THE GIT OBJECT DATABASE

To help you visualize the contents and dynamics of the *git Object Database, Figure One* shows the four objects in the Object Database after performing these steps:

1. Start with a new, empty repository.
2. Create *file1* with content `Here is some stuff in file1!`
3. Create *file2* with content `Other stuff is found in the second file.`
4. Update the index to add *file1* and *file2*.
5. Make an initial commit.

After moving *file1* and *file2* into a new subdirectory, introducing a new *file3* with contents `New top level file.` at the top level, and making a new commit, the object database would look like *Figure Two*.

Even though *file1* and *file2* have moved within the filesystem, neither their blob contents nor their SHA1 have changed. Furthermore, the tree object `b5ac`... that represents the directory containing both *file1* and *file2* also did not change.

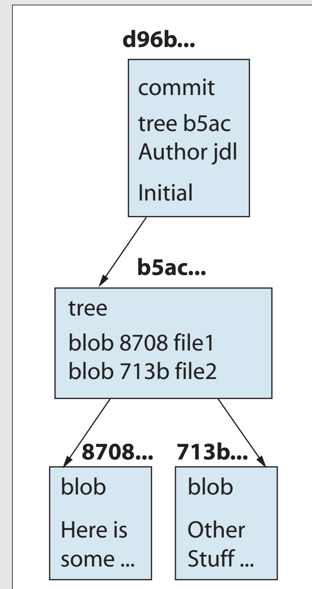

FIGURE ONE: The state of the *git* repository after committing two new files

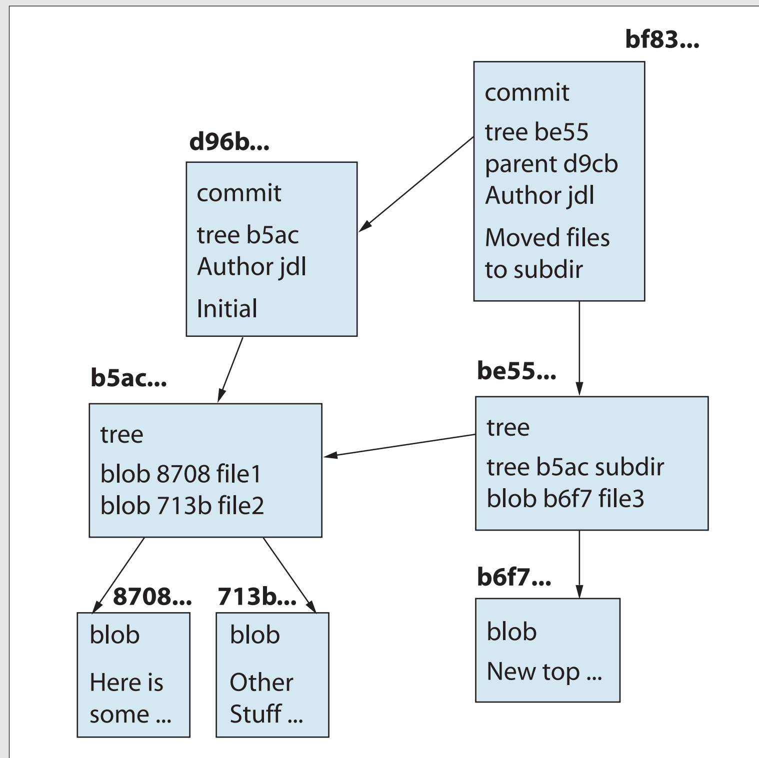

FIGURE TWO: The *git* repository of *Figure One* after adding a file and moving the previous two files