Git (software)

In software development, **Git** (/gɪt/) is a distributed revision control and source code management system with an emphasis on speed.[4] Git was initially designed and developed by Linus Torvalds for Linux kernel development. Every Git working directory is a full-fledged repository with complete history and full revision tracking capabilities, not dependent on network access or a central server. Git is free software distributed under the terms of the GNU General Public License version 2.

Name

Linus Torvalds has quipped about the name "git", which is English slang for a stupid or unpleasant person. Torvalds said: "I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'."[5][6] The man page describes git as "the stupid content tracker".[7]

History

Git development began after many Linux kernel developers chose to give up access to BitKeeper, a proprietary SCM system that had previously been used to maintain the project.[8] The copyright holder of BitKeeper, Larry McVoy, had withdrawn free use of the product after he claimed that Andrew Tridgell had reverse-engineered the BitKeeper protocols.

Torvalds wanted a distributed system that he could use like BitKeeper, but none of the available free systems met his needs, particularly his performance needs. From an email he wrote on 7 April 2005 while writing the first prototype:[9]

However, the SCMs I've looked at make this hard. One of the things (the main thing, in fact) I've been working at is to make that process really *efficient*. If it takes half a minute to apply a patch and remember the changeset boundary etc. (and quite frankly, that's *fast* for most SCMs around for a project the size of Linux), then a series of 250 emails (which is not unheard of at all when I sync with Andrew, for example) takes two hours. If one of the patches in the middle doesn't apply, things are bad bad bad.

Now, BK wasn't a speed demon either (actually, compared to everything else, BK *is* a speed demon, often by one or two orders of magnitude), and took about 10–15 seconds per email when I merged with Andrew. HOWEVER, with BK that wasn't as big of an issue, since the BK<->BK merges were so easy, so I never had the slow email merges with any of the other main developers. So a patch-application-based SCM "merger" actually would need to be *faster* than BK is. Which is really really really hard.

So I'm writing some scripts to try to track things a whole lot faster. Initial indications are that I should be able to do it almost as quickly as I can just apply the patch, but quite frankly, I'm at most half done, and if I hit a snag maybe that's not true at all. Anyway, the reason I can do it quickly is that my scripts will *not* be an SCM, they'll be a very specific "log Linus' state" kind of thing. That will make the linear patch merge a lot more time-efficient, and thus possible.

(If a patch apply takes three seconds, even a big series of patches is not a problem: if I get notified within a minute or two that it failed half-way, that's fine, I can then just fix it up manually. That's why latency is critical—if I'd have to do things effectively "offline", I'd by definition not be able to fix it up when problems happen).

Torvalds had several design criteria:

1. Take CVS as an example of what *not* to do; if in doubt, make the exact opposite decision. To quote Torvalds, speaking somewhat tongue-in-cheek:
2. For the first 10 years of kernel maintenance, we literally used tarballs and patches, which is a much superior source control management system than CVS is, but I did end up using CVS for 7 years at a commercial company [Transmeta[10]] and I hate it with a passion. When I say I hate CVS with a passion, I have to also say that if there are any SVN [Subversion] users in the audience, you might want to leave. Because my hatred of CVS has meant that I see Subversion as being the most pointless project ever started. The slogan of

Subversion for a while was "CVS done right", or something like that, and if you start with that kind of slogan, there's nowhere you can go. There is no way to do CVS right.[11]

3. Support a distributed, BitKeeper-like workflow:
4. BitKeeper was not only the first source control system that I ever felt was worth using at all, it was also the source control system that taught me why there's a point to them, and how you actually can do things. So Git in many ways, even though from a technical angle it is very very different from BitKeeper (which was another design goal, because I wanted to make it clear that it wasn't a BitKeeper clone), a lot of the flows we use with Git come directly from the flows we learned from BitKeeper.[11]
5. Very strong safeguards against corruption, either accidental or malicious.[11][12]
6. Very high performance.

The first three criteria eliminated every pre-existing version control system, except for Monotone, and the fourth excluded everything.[11] So, immediately after the 2.6.12-rc2 Linux kernel development release,[11] he set out to write his own.[11]

The development of Git began on 3 April 2005.[13] The project was announced on 6 April,[14] and became self-hosting as of 7 April.[13] The first merge of multiple branches was done on 18 April.[15] Torvalds achieved his performance goals; on 29 April, the nascent Git was benchmarked recording patches to the Linux kernel tree at the rate of 6.7 per second.[16] On 16 June, the kernel 2.6.12 release was managed by Git.[17]

While strongly influenced by BitKeeper, Torvalds deliberately attempted to avoid conventional approaches, leading to a unique design.[18] He developed the system until it was usable by technical users, then turned over maintenance on 26 July 2005 to Junio Hamano, a major contributor to the project.[19] Hamano was responsible for the 1.0 release on 21 December 2005,[20] and remains the project's maintainer.

Design

Git's design was inspired by BitKeeper and Monotone.[21][22] Git was originally designed as a low-level version control system engine on top of which others could write front ends, such as Cogito or StGIT.[22] However, the core Git project has since become a complete revision control system that is usable directly.[23]

Characteristics

Git's design is a synthesis of Torvalds's experience with Linux in maintaining a large distributed development project, along with his intimate knowledge of file system performance gained from the same project and the urgent need to produce a working system in short order. These influences led to the following implementation choices:

Strong support for non-linear development

Git supports rapid branching and merging, and includes specific tools for visualizing and navigating a non-linear development history. A core assumption in Git is that a change will be merged more often than it is written, as it is passed around various reviewers. Branches in git are very lightweight: A branch in git is only a reference to a single commit. With its parental commits, the full branch structure can be constructed.

Distributed development

Like Darcs, BitKeeper, Mercurial, SVK, Bazaar and Monotone, Git gives each developer a local copy of the entire development history, and changes are copied from one such repository to another. These changes are imported as additional development branches, and can be merged in the same way as a locally developed branch.

Compatibility with existing systems/protocols

Repositories can be published via HTTP, FTP, rsync, or a Git protocol over either a plain socket or ssh. Git also has a CVS server emulation, which enables the use of existing CVS clients and IDE plugins to access Git repositories. Subversion and svk repositories can be used directly with git-svn.

Efficient handling of large projects

Torvalds has described Git as being very fast and scalable,[24] and performance tests done by Mozilla showed it was an order of magnitude faster than some revision control systems, and fetching revision history from a locally stored repository can be one hundred times faster than fetching it from the remote server.[25][26] In particular, Git does not get slower as the project history grows larger.[27]

Cryptographic authentication of history

The Git history is stored in such a way that the name of a particular revision (a "commit" in Git terms) depends upon the complete development history leading up to that commit. Once it is published, it is not possible to change the old versions without it being noticed. The structure is similar to a hash tree, but with additional data at the nodes as well as the leaves.[28] (Mercurial and Monotone also have this property.)

Toolkit-based design

Git was designed as a set of programs written in C, and a number of shell scripts that provide wrappers around those programs.[29] Although most of those scripts have since been rewritten in C for speed and portability, the design remains, and it is easy to chain the components together.[30]

Pluggable merge strategies

As part of its toolkit design, Git has a well-defined model of an incomplete merge, and it has multiple algorithms for completing it, culminating in telling the user that it is unable to complete the merge automatically and manual editing is required.

Garbage accumulates unless collected

Aborting operations or backing out changes will leave useless dangling objects in the database. These are generally a small fraction of the continuously growing history of wanted objects. Git will automatically perform garbage collection when enough loose objects have been created in the repository. Garbage collection can be called explicitly using git gc --prune.[31]

Periodic explicit object packing

Git stores each newly created object as a separate file. Although individually compressed, this takes a great deal of space and is inefficient. This is solved by the use of "packs" that store a large number of objects in a single file (or network byte stream), delta-compressed among themselves. Packs are compressed using the heuristic that files with the same name are probably similar, but do not depend on it for correctness. Newly created objects (newly added history) are still stored singly, and periodic repacking is required to maintain space efficiency. The process of packing the repository can be very computationally expensive. By allowing objects to exist in the repository in a loose, but quickly generated format, git allows the expensive pack operation to be deferred until later when time does not matter (e.g. the end of the work day). Git does periodic repacking automatically but manual repacking is also possible with the git gc command.

Another property of Git is that it snapshots directory trees of files. The earliest systems for tracking versions of source code, SCCS and RCS, worked on individual files and emphasized the space savings to be gained from interleaved deltas (SCCS) or delta encoding (RCS) the (mostly similar) versions. Later revision control systems maintained this notion of a file having an identity across multiple revisions of a project. However, Torvalds rejected this concept.[32] Consequently, Git does not explicitly record file revision relationships at any level below the source code tree.

Inexplicit revision relationships has some significant consequences:

- It is slightly more expensive to examine the change history of a single file than the whole project.[33] To obtain a history of changes affecting a given file, Git must walk the global history and then determine whether each change modified that file. This method of examining history does, however, let Git produce with equal efficiency a single history showing the changes to an arbitrary set of files. For example, a subdirectory of the source tree plus an associated global header file is a very common case.
- Renames are handled implicitly rather than explicitly. A common complaint with CVS is that it uses the name of a file to identify its revision history, so moving or renaming a file is not possible without either interrupting its history, or renaming the history and thereby making the history inaccurate. Most post-CVS revision control systems solve this by giving a file a unique long-lived name (a sort of inode number) that survives renaming. Git does not record such an identifier, and this is claimed as an advantage.[34][35] Source code files are sometimes split or merged as well as simply renamed,[36] and recording this as a simple rename would freeze an inaccurate description of what happened in the (immutable) history. Git addresses the issue by detecting renames while browsing the history of snapshots rather than recording it when making the snapshot.[37] (Briefly, given a file in revision *N,* a file of the same name in revision *N−1* is its default ancestor. However, when there is no like-named file in revision *N−1,* Git searches for a file that existed only in revision *N−1* and is very similar to the new file.) However, it does require more CPU-intensive work every time history is reviewed, and a number of options to adjust the heuristics.

Git implements several merging strategies; a non-default can be selected at merge time:[38]

- **resolve**: the traditional three-way merge algorithm.

- **recursive**: This is the default when pulling or merging one branch, and is a variant of the three-way merge algorithm. "When there are more than one common ancestors that can be used for three-way merge, it creates a merged tree of the common ancestors and uses that as the reference tree for the three-way merge. This has been reported to result in fewer merge conflicts without causing mis-merges by tests done on actual merge commits taken from Linux 2.6 kernel development history. Additionally this can detect and handle merges involving renames."[39]
- **octopus**: This is the default when merging more than two heads.

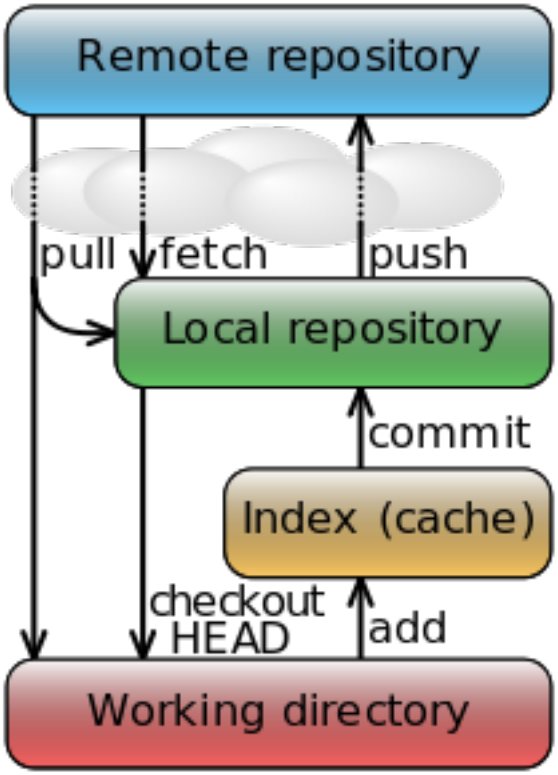
Implementation

Git's primitives are not inherently a software configuration management (SCM) system. Torvalds explains,[40]
In many ways you can just see git as a filesystem — it's content-addressable, and it has a notion of versioning, but I really really designed it coming at the problem from the viewpoint of a *filesystem* person (hey, kernels is what I do), and I actually have absolutely *zero* interest in creating a traditional SCM system.
From this initial design approach, Git has developed the full set of features expected of a traditional SCM,[23] with features mostly being created as needed, then refined and extended over time.

Git has two data structures: a mutable *index* that caches information about the working directory and the next revision to be committed; and an immutable, append-only *object database.*
The object database contains four types of objects:

- A *blob* object is the content of a file. Blob objects have no file name, time stamps, or other metadata.
- A *tree* object is the equivalent of a directory. It contains a list of file names, each with some type bits and the name of a blob or tree object that is that file, symbolic link, or directory's contents. This object describes a snapshot of the source tree.
- A *commit* object links tree objects together into a history. It contains the name of a tree object (of the top-level source directory), a time stamp, a log message, and the names of zero or more parent commit objects.
- A *tag* object is a container that contains reference to another object and can hold additional meta-data related to another object. Most commonly, it is used to store a digital signature of a commit object corresponding to a particular release of the data being tracked by Git.

The index serves as connection point between the object database and the working tree.
Each object is identified by a SHA-1 hash of its contents. Git computes the hash, and uses this value for the object's name. The object is put into a directory matching the first two characters of its hash. The rest of the hash is used as the file name for that object.
Git stores each revision of a file as a unique blob object. The relationships between the blobs can be found through examining the tree and commit objects. Newly added objects are stored in their entirety using zlib compression.
This can consume a large amount of disk space quickly, so objects can be combined into *packs*, which use delta compression to save space, storing blobs as their changes relative to other blobs.
Git servers typically listen on TCP port 9418.[41]



Terminology and structure

A Git repository — data and metadata — is completely contained within its directory, so a normal system-level copy (or rename, or delete) of an entire Git repository is a safe operation. The resulting copy is both independent of and unaware of the original.

Data

Working tree

The files and directories — the things typically considered to be "under source control, but checked out" that the user changes in an editor or otherwise — live under the top repository directory, but outside of its .git/ directory.
During a normal work session, one can choose to ask Git to note for later some or all of the differences between how the files are now and how Git's permanent repository remembers them. The place it remembers them is called "the staging area" (or index) and remembering them is called "staging". One can also remove or exclude selected changes from staging if it suits. Staging diffs is easy, and is similar in concept to creating a patch file. Staging is the precursor to committing those changes as a permanent record in the Git repository.

.gitignore

This is a plain text configuration file which contains filename patterns that git should consider invisible for the purposes of source control — *e.g.*, *.o, *.class, bin/

- it is outside the .git directory with the "normal" files, in order to keep it under source control: everyone in a project typically needs the same exclusions.
- it can refer to itself, to excuse itself from source control!

Metadata

The "behind the scenes" source control elements live within the .git/ directory at the root of the repository.

Commits

- commits, representing checked-in versions of files and directories live in .git/objects
    - each commit represents a snapshot of the full tree of files current at the point the commit was created.

    - each commit is uniquely identified to Git commands by a 40-character identifier, but it's usually sufficiently unique to provide only the first seven characters of it.
        - git will complain if a shortened identifier is ambiguous.
    - each commit has a reference to its parent commit(s).
        - collectively, all the commits form a tree-like structure based on the parent relationship, whence the concept of branches derives.

Index (stage)

- the index — a set of file changes being collected (with "git add") to create in the repository as a single commit (version)
    - file changes explicitly added to the index are termed "staged changes".
    - file changes made but not (or not yet) added to the index are termed "unstaged changes".
    - staged and unstaged changes coexist harmlessly (and can be seen with "git status").
    - creating a commit (with "git commit") copies the staged changes into a new commit and then clears the index
        - creating a commit does not affect the working tree.
        - the new commit becomes the current commit and incorporates the staged changes, so they no longer show up as differences between the commit and the working tree.
        - unstaged changes thus continue to show as differences, until added to the index (or reverted).
    - the index can even be told to stage selected "hunks" (patches/blocks of change) within a file, while leaving other hunks within the very same file unstaged.

Branches and tags

- these are ways to refer to particular commits, and they live in .git/refs and .git/packed-refs
    - a tag marks a particular commit and is only ever moved manually; other SCMs might call it a label.
    - a branch is a special mobile tag that is updated whenever a commit is made to that branch.
        - a branch is thus really a label that is always on the commit that most SCMs would call the branch/HEAD version.
        - local branches live in .git/refs/heads
        - branches from other repositories are maintained in .git/refs/remotes
- Branches and tags are very cheap and almost identical — they can be made whenever one wants to remember a particular point in development.
    - .git/refs/heads/master contains a 40 character identifier for the commit that is the master branch. Much Git metadata is in plain text files.

Stash

- the stash ("git stash") is a special list of commits used to hold temporary changes.

Configuration

- information about which remote repositories this repository can talk to, how local branches relate to remote branches, etc., lives in .git/config, in plain text.

Hooks

- special scripts that run for various source-control events to enforce policy.
- hooks are within the .git directory and not therefore under source control, *i.e.*, not cloned from remote repositories.

Global settings

User preferences — real name, email address, preferences, command aliases, etc. — are held in a user's home directory in a file called ~/.gitconfig.

Working environment

The working environment looks much like it would as if Git wasn't around: the working tree is a normal directory with only the subtle presence of a .git directory to suggest otherwise.
After the first "git clone" or "git init" operation, git sets the "current branch" to be "master" and will make sure that the working tree reflects the commit to which the "master" branch points. Each commit will have its parent set to the commit currently marked "master", and "master" will be moved to that new commit, ready for the next one.

Portability

Git is primarily developed on Linux, but can be used on other Unix-like operating systems including BSD, Solaris and Mac OS X, along with Microsoft Windows[42] .

Adoption

The Eclipse Foundation reported in its annual community survey that as of May 2012 Git is the primary source control system used by over 27% of professional software developers, an increase from 12.8% in 2011.[43] Open source directory Ohloh reports a similar uptake among open source projects.[44]
The UK IT jobs website itjobswatch.co.uk reports that as of July 2012, approximately 7% of UK permanent software development job openings list Git as a requirement,[45] compared to 19.9% for Subversion,[46] 9.1% for Microsoft Team Foundation Server,[47] and 2.6% for Visual SourceSafe.[48]

Source code hosting

See also: Comparison of open source software hosting facilities
The following Web sites provide free source code hosting for Git repositories:[49]

- Backlog
- BerliOS
- Bitbucket
- CodePlex[50]
- GitHub
- Gitorious
- GNU Savannah
- Google Code
- JavaForge (with pull requests to control source code contribution)
- SourceForge

See also

Free software portal

- Comparison of revision control software
- Comparison of open source software hosting facilities
- Distributed revision control
- List of revision control software
- Repo (script)
- Gerrit (software)

References

1. ^ Hamano, Junio (2012-07-22). "[ANNOUNCE] Git 1.7.11.3". *kernel mailing list*. Retrieved 2012-07-23.

2. ^ Hamano, Junio (2012-07-24). "[ANNOUNCE] Git v1.7.12-rc0". *kernel mailing list*. Retrieved 2012-07-24.
3. ^ "git/git.git/tree". git.kernel.org. Retrieved 2009-06-15.
4. ^ Linus Torvalds (2005-04-07). "Re: Kernel SCM saga..". *linux-kernel mailing list*. "So I'm writing some scripts to try to track things a whole lot faster."
5. ^ "GitFaq: Why the 'git' name?". Git.or.cz. Retrieved 2012-07-14.
6. ^ "After controversy, Torvalds begins work on 'git'". PC World. 2012-07-14. "Torvalds seemed aware that his decision to drop BitKeeper would also be controversial. When asked why he called the new software, "git", British slang meaning "a rotten person", he said. "I'm an egotistical bastard, so I name all my projects after myself. First Linux, now git""
7. ^ "git(1) Manual Page". Retrieved 2012-07-21.
8. ^ Feature: No More Free BitKeeper | KernelTrap.org
9. ^ Linus Torvalds (2005-04-07). "Re: Kernel SCM saga..". *linux-kernel mailing list*.
10. ^ Linus Torvalds (2005-10-31). "Re: git versus CVS (versus bk)". *git mailing list*.
11. ^ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** Linus Torvalds (2007-05-03). *Google tech talk: Linus Torvalds on git*. Event occurs at 02:30. Retrieved 2007-05-16.
12. ^ Linus Torvalds (2007-06-10). "Re: fatal: serious inflate inconsistency". *git mailing list*. A brief description of Git's data integrity design goals.
13. ^ ***a*** ***b*** Linus Torvalds (2007-02-27). "Re: Trivia: When did git self-host?". *git mailing list*.
14. ^ Linus Torvalds (2005-04-06). "Kernel SCM saga..". *linux-kernel mailing list*.
15. ^ Linus Torvalds (2005-04-17). "First ever real kernel git merge!". *git mailing list*.
16. ^ Matt Mackall (2005-04-29). "Mercurial 0.4b vs git patchbomb benchmark". *git mailing list*.
17. ^ Linus Torvalds (2005-06-17). "Linux 2.6.12". *git-commits-head mailing list*.
18. ^ Linus Torvalds (2006-10-20). "Re: VCS comparison table". *git mailing list*. A discussion of Git vs. BitKeeper
19. ^ Linus Torvalds (2005-07-27). "Meet the new maintainer...". *git mailing list*.
20. ^ Junio C Hamano (2005-12-21). "ANNOUNCE: GIT 1.0.0". *git mailing list*.
21. ^ Linus Torvalds (2006-05-05). "Re: [ANNOUNCE] Git wiki". *linux-kernel mailing list*. "Some historical background" on git's predecessors
22. ^ ***a*** ***b*** Linus Torvalds (2005-04-08). "Re: Kernel SCM saga". *linux-kernel mailing list*. Retrieved 2008-02-20.
23. ^ ***a*** ***b*** Linus Torvalds (2006-03-23). "Re: Errors GITtifying GCC and Binutils". *git mailing list*.
24. ^ Linus Torvalds (2006-10-19). "Re: VCS comparison table". *git mailing list*.
25. ^ Stenback, Johnny (2006-11-30). "bzr/hg/git performance". *Jst's Blog*. Retrieved 2008-02-20., benchmarking "git diff" against "bzr diff", and finding the former 100x faster in some cases.
26. ^ Roland Dreier (2006-11-13). "Oh what a relief it is"., observing that "git log" is 100x faster than "svn log" because the latter has to contact a remote server.
27. ^ Fendy, Robert (2009-01-21). *DVCS Round-Up: One System to Rule Them All?—Part 2*. Linux Foundation. Retrieved 2009-06-25. "One aspect that really sets Git apart is its speed. ...dependence on repository size is very, very weak. For all facts and purposes, Git shows nearly a flat-line behavior when it comes to the dependence of its performance on the number of files and/or revisions in the repository, a feat no other VCS in this review can duplicate (although Mercurial does come quite close)."
28. ^ "Trust". *Git Concepts*. Git User's Manual. 2006-10-18.
29. ^ Linus Torvalds. "Re: VCS comparison table". *git mailing list*. Retrieved 2009-04-10., describing Git's script-oriented design
30. ^ iabervon (2005-12-22). "Git rocks!"., praising Git's scriptability
31. ^ "Git User's Manual". 2007-08-05.
32. ^ Linus Torvalds (2005-04-10). "Re: more git updates..". *linux-kernel mailing list*.
33. ^ Bruno Haible (2007-02-11). "how to speed up "git log"?". *git mailing list*.
34. ^ Linus Torvalds (2006-03-01). "Re: impure renames / history tracking". *git mailing list*.
35. ^ Junio C Hamano (2006-03-24). "Re: Errors GITtifying GCC and Binutils". *git mailing list*.
36. ^ Junio C Hamano (2006-03-23). "Re: Errors GITtifying GCC and Binutils". *git mailing list*.
37. ^ Linus Torvalds (2006-11-28). "Re: git and bzr". *git mailing list*., on using git-blame to show code moved between source files
38. ^ Linus Torvalds (2007-07-18). "git-merge(1)".

39. ^ Linus Torvalds (2007-07-18). "CrissCrossMerge".
40. ^ Linus Torvalds (2005-04-10). "Re: more git updates...". *linux-kernel mailing list*.
41. ^ "Git Installation". git.wiki.kernel.org. Retrieved 2012-01-25.
42. ^ "downloads". Retrieved 14 May 2012.
43. ^ "Results of Eclipse Community Survey 2012".
44. ^ "Ohloh: Compare Repositories".
45. ^ http://www.itjobswatch.co.uk/jobs/uk/git%20%28software%29.do
46. ^ http://www.itjobswatch.co.uk/jobs/uk/subversion.do
47. ^ http://www.itjobswatch.co.uk/jobs/uk/team%20foundation%20server.do
48. ^ http://www.itjobswatch.co.uk/jobs/uk/vss/sourcesafe.do
49. ^ https://git.wiki.kernel.org/articles/g/i/t/GitHosting_2036.html
50. ^ Bright, Peter (22 March 2012). "Microsoft brings git support to its CodePlex hosting service". *Ars Technica*. Retrieved 23 March 2012.

External links

- Official website
- Git Community Book
- Introduction to git-svn for Subversion/SVK users and deserters by Sam Vilain
- Git for computer scientists
- Git Magic: a comprehensive listing of Git tips & tricks
- Git Quick Reference
- Linus Torvalds hosting a Google Tech Talk on Git
- Git Wiki at kernel.org
- gitref.org — Git quick reference site for most commonly used commands
- Introduction to Git with Scott Chacon of GitHub — Marakana
- Pro Git book
- A Note from the Maintainer