Git from the bottom up

Fri, 2 May 2008

by John Wiegley

In my pursuit to understand Git, it's been helpful for me to understand it from the bottom up — rather than look at it only in terms of its high-level commands. And since Git is so beautifully simple when viewed this way, I thought others might be interested to read what I've found, and perhaps avoid the pain I went through finding it.

I used Git version 1.5.4.5 for each of the examples found in this document.

Contents

Introduction

Welcome to the world of Git. I hope this document will help to advance your understanding of this powerful content tracking system, and reveal a bit of the simplicity underlying it — however dizzying its array of options may seem from the outside.

Before we dive in, there are a few terms which should be mentioned first, since they'll appear repeatedly throughout this text:

repository A **repository** is an archive of what the **working tree** looked like at different times in the past — whether on your machine or someone else's. It's a history of snapshots,and allows you to compare your working tree to those past states. You can even "checkout" a copy of the past, make changes you should have made long ago, and then merge those changes forward into your current branch.

the index Unlike other, similar tools you may have used, Git does not commit changes directly from the **working tree** into the **repository**. Instead, changes are first registered in something called the **index**. Think of it as a way of "confirming" your changes, one by one, before doing a commit (which records all your approved changes at once). Some find it helpful to call it instead as the "staging area", instead of the index.

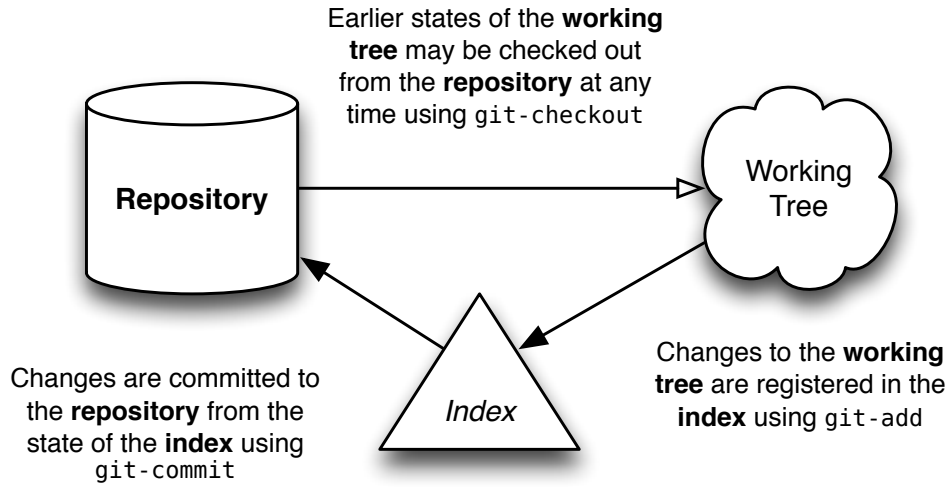
working tree A **working tree** is any directory on your filesystem which has a **repository** associated with it (typically indicated by the presence of a sub-directory within it named `.git`.). It includes all the files and sub-directories in that directory.

branch A **branch** is just a name for a commit (and much more will be said about commits in a moment), also called a reference. It's the parentage of a commit which defines its history, and thus the typical notion of a "branch of development".

master The mainline of development in most repositories is done on a branch called "master". Although this is a typical default, it is in no way special.

HEAD The word **HEAD** refers to the most recent commit of the last branch you checked out. Thus, if you are currently on the master branch, the words master and HEAD would be equivalent.

The usual flow of events is this: After creating a repository, your work is done in the working tree. Once your work reaches a significant point — the completion of a bug, the end of the working day, a moment when everything compiles — you add your changes successively to the index. Once the index contains everything you intend to commit, you record its content in the repository. Here's a simple diagram that shows a typical project's life-cycle:



With this basic picture in mind[1], the following sections shall attempt to describe how each of these different entities is important to the operation of Git.

Repository: Directory content tracking

As mentioned above, what Git does is quite rudimentary: it maintains snapshots of a directory's contents. Much of its internal design can be understood in terms of this basic task.

The design of a Git repository in many ways mirrors the structure of a UNIX filesystem: A *filesystem* begins with a root directory, which typically consists of other directories, most of which have leaf nodes, or *files,* that contain data. Meta-data about these files' contents is stored both in the directory (the names), and in the *i-nodes* that reference the contents of those file (their size, type, permissions, etc). Each i-node has a unique number that identifies the contents of its related file. And while you may have many directory entries pointing to a particular i-node (i.e., hard-links), it's the i-node which "owns" the contents stored on your filesystem.

1. In reality, a checkout causes contents from the repository to be copied into the index, which is then written out to the working tree. But since the user never sees this usage of the index during a checkout operation, I felt it would make more sense not to depict it in the diagram.

Internally, Git shares a strikingly similar structure, albeit with one or two key differences. First, it represents your file's contents in *blobs*, which are also leaf nodes in something awfully close to a directory, called a *tree*. Just as an i-node is uniquely identified by a system-assigned number, a blob is named by computing the SHA1 hash id of its size and contents. For all intents and purposes this is just an arbitrary number, like an i-node, except that it has two additional properties: first, it verifies the blob's contents will never change; and second, the same contents shall always be represented by the same blob, no matter where it appears: across commits, across repositories — even across the whole Internet. If multiple trees reference the same blob, this is just like hard-linking: the blob will not disappear from your repository as long as there is at least one link remaining to it.

The difference between a Git blob and a filesystem's file is that a blob stores no metadata about its content. All such information is kept in the tree that holds the blob. One tree may know those contents as a file named "foo" that was created in August 2004, while another tree may know the same contents as a file named "bar" that was created five years later. In a normal filesystem, two files with the same contents but with such different metadata would always be represented as two independent files. Why this difference? Mainly, it's because a filesystem is designed to support files that change, whereas Git is not. The fact that data is immutable in the Git repository is what makes all of this work and so a different design was needed. And as it turns out, this design allows for much more compact storage, since all objects having identical content can be shared, no matter where they are.

Introducing the blob

Now that the basic picture has been painted, let's get into some practical examples. I'm going to start by creating a sample Git repository, and showing how Git works from the bottom-up in that repository. Feel free to follow along as you read:

```
$ mkdir sample; cd sample
$ echo 'Hello, world!' > greeting
```

Here I've created a new filesystem directory named "sample" which contains a file whose contents are prosaically predictable. I haven't even created a repository yet, but already I can start using some of Git's commands to understand what it's going to do. First of all, I'd like to know which hash id Git is going to store my greeting text under:

```
$ git hash-object greeting
af5626b4a114abcb82d63db7c8082c3c4756e51b
```

If you run this command on your system, you'll get the same hash id. Even though we're creating two different repositories (possibly a world apart, even) our greeting blob in those two

repositories will have the same hash id. I could even pull commits from your repository into mine, and Git would realize that we're tracking the same content — and so would only store one copy of it! Pretty cool.

The next step is to initialize a new repository and commit the file into it. I'm going to do this all in one step right now, but then come back and do it again in stages so you can see what's going on underneath:

```
$ git init
$ git add greeting
$ git commit -m "Added my greeting"
```

At this point our blob should be in the system exactly as we expected, using the hash id determined above. As a convenience, Git requires only as many digits of the hash id as are necessary to uniquely identify it within the repository. Usually just six or seven digits is enough:

```
$ git cat-file -t af5626b
blob

$ git cat-file blob af5626b
Hello, world!
```

There it is! I haven't even looked at which commit holds it, or what tree it's in, but based solely on the contents I was able to assume it's there, and there it is. It will always have this same identifier, no matter how long the repository lives or where the file within it is stored. These particular contents are now verifiably preserved, forever.

In this way, a Git blob represents the fundamental data unit in Git. Really, the whole system is about blob management.

Blobs are stored in trees

The contents of your files are stored in blobs, but those blobs are pretty featureless. They have no name, no structure — they're just "blobs", after all.

In order for Git to represent the structure and naming of your files, it attaches blobs as leaf nodes within a tree. Now, I can't discover which tree(s) a blob lives in just by looking at it, since it may have many, many owners. But I know it must live somewhere within the tree held by the commit I just made:

```
$ git ls-tree HEAD
100644 blob af5626b4a114abcb82d63db7c8082c3c4756e51b greeting
```

There it is! This first commit added my greeting file to repository. It contains one Git tree, which has a single leaf: the greeting content's blob.

Although I can look at the tree containing my blob by passing HEAD to `ls-tree`, I haven't yet seen the underlying tree object referenced by that commit. Here are a few other commands to highlight that difference and thus discover my tree:

```
$ git rev-parse HEAD
588483b99a46342501d99e3f10630cfc1219ea32   # different on your system

$ git cat-file -t HEAD
commit

$ git cat-file commit HEAD
tree 0563f77d884e4f79ce95117e2d686d7d6e282887
author John Wiegley <johnw@newartisans.com> 1209512110 -0400
committer John Wiegley <johnw@newartisans.com> 1209512110 -0400

Added my greeting
```

The first command decodes the HEAD alias into the commit it references, the second verifies its type, while the third command shows the hash id of the tree held by that commit, as well as the other information stored in the commit object. The hash id for the commit is unique to my repository — because it includes my name and the date when I made the commit — but the hash id for the tree should be common between your example and mine, containing as it does the same blob under the same name.

Let's verify that this is indeed the same tree object:

```
$ git ls-tree 0563f77
100644 blob af5626b4a114abcb82d63db7c8082c3c4756e51b greeting
```

There you have it: my repository contains a single commit, which references a tree that holds a blob — the blob containing the contents I want to record. There's one more command I can run to verify that this is indeed the case:

```
$ find .git/objects -type f
.git/objects/05/63f77d884e4f79ce95117e2d686d7d6e282887
.git/objects/58/8483b99a46342501d99e3f10630cfc1219ea32
.git/objects/af/5626b4a114abcb82d63db7c8082c3c4756e51b
```

From this output I see that the whole of my repo contains three objects, each of whose hash id has appeared in the preceding examples. Let's take one last look at the types of these objects, just to satisfy curiosity:

```
$ git cat-file -t 588483b99a46342501d99e3f10630cfc1219ea32
commit
$ git cat-file -t 0563f77d884e4f79ce95117e2d686d7d6e282887
tree
$ git cat-file -t af5626b4a114abcb82d63db7c8082c3c4756e51b
blob
```

I could have used the `show` command at this point to view the concise contents of each of these objects, but I'll leave that as an exercise to the reader.

How trees are made

Every commit holds a single tree, but how are trees made? We know that blobs are created by stuffing the contents of your files into blobs — and that trees own blobs — but we haven't yet seen how the tree that holds the blob is made, or how that tree gets linked to its parent commit.

Let's start with a new sample repository again, but this time by doing things manually, so you can get a feeling for exactly what's happening under the hood:

```
$ rm -fr greeting .git
$ echo 'Hello, world!' > greeting
$ git init
$ git add greeting
```

It all starts when you first add a file to the index. For now, let's just say that the index is what you use to initially create blobs out of files. When I added the file `greeting`, a change occurred in my repository. I can't see this change as a commit yet, but here is one way I can tell what happened:

```
$ git log               # this will fail, there are no commits!
fatal: bad default revision 'HEAD'

$ git ls-files --stage   # list blob in the index
100644 af5626b4a114abcb82d63db7c8082c3c4756e51b 0 greeting
```

What's this? I haven't committed anything to the repository yet, but already an object has come into being. It has the same hash id I started this whole business with, so I know it represents the contents of my `greeting` file. I could use `cat-file -t` at this point on the hash id, and I'd see that it was a blob. It is, in fact, the same blob I got the first time I created this sample repository. The same file will always result in the same blob (just in case I haven't stressed that enough).

This blob isn't referenced by a tree yet, nor are there any commits. At the moment it is only known by a file named `.git/index`, which references the blobs and trees that make up the current index. So now let's make a tree in the repo for our blob to hang off of:

```
$ git write-tree        # record the contents of the index in a tree
0563f77d884e4f79ce95117e2d686d7d6e282887
```

This number should look familiar as well: a tree containing the same blobs (and sub-trees) will always have the same hash id. I don't have a commit object yet, but now there is a tree object in that repository which holds the blob. The purpose of the low-level `write-tree` command is to take whatever the contents of the index are and tuck them into a new tree for the purpose of creating a commit.

I can manually make a new commit object by using this tree directly, which is just what the `commit-tree` command does:

```
$ echo "Initial commit" | git commit-tree 0563f77
5f1bc85745dcccce6121494fdd37658cb4ad441f
```

The raw `commit-tree` command takes a tree's hash id and makes a commit object to hold it. If I had wanted the commit to have a parent, I would have had to specify the parent commit's hash id explicitly using the `-p` option. Also, note here that the hash id differs from what will appear on your system: This is because my commit object refers to both my name, and the date at which I created the commit, and these two details will always be different from yours.

Our work is not done yet, though, since I haven't registered the commit as the new head of the current branch:

```
$ echo 5f1bc85745dcccce6121494fdd37658cb4ad441f > .git/refs/heads/master
```

This command establishes the HEAD of the "master" branch to be our recent commit. This works right now because brand new repositories always start out on an empty "master" branch. Another, much safer way to set master's HEAD commit in this situation is to use `update-ref`:

```
$ git update-ref HEAD 5f1bc85745dcccce6121494fdd37658cb4ad441f
```

It's hard to believe it's this simple, but yes, I can now use `log` to see my newly minted commit:

```
$ git log
commit 5f1bc85745dcccce6121494fdd37658cb4ad441f
Author: John Wiegley <johnw@newartisans.com>
Date:   Mon Apr 14 11:14:58 2008 -0400

    Initial commit
```

As a side note: if I hadn't updated the HEAD of the current branch to point to the new commit, it would have been considered "unreachable", since nothing currently refers to it nor is it the parent of an reachable commit. When this is the case, the commit object can be removed from the repository, along with its tree and all its blobs (this happens automatically by a command called `gc`, which you rarely need to use manually). By linking the commit to a name within `refs/heads`, as we did above, it becomes a reachable commit, which ensures that it's kept around from now on.

The beauty of commits

Most version control systems make "branches" into magical things, often distinguishing them from the "main line" or "trunk". But in Git there are no branches as separate entities: there are only blobs, trees and commits[2]. Since a commit can have one or more parents, and those commits can have parents, this is what allows a single commit to be treated like a branch: because it knows the whole history that led up to it.

You can examine all the top-level, referenced commits at any time using the `branch` command:

```
$ git branch -v
* master 5f1bc85 Initial commit
```

Say it with me: A branch is nothing more than a named reference to a single commit. In this way, branches and tags are identical, with the sole exception that tags can have their own descriptions, just like the commits they reference. Branches are just names, but tags are descriptive, well, "tags".

But the fact is, we don't really need to use aliases at all. For example, if I wanted to, I could reference everything in the repository using only the hash ids of its commits. Here's me being straight up loco and resetting the head of my working tree to a particular commit:

```
$ git reset --hard 5f1bc85
```

The `--hard` option says to erase all changes currently in my working tree, whether they've been registered for a checkin or not (more will be said about this command later). A safer way to do the same thing is by using `checkout`:

```
$ git checkout 5f1bc85
```

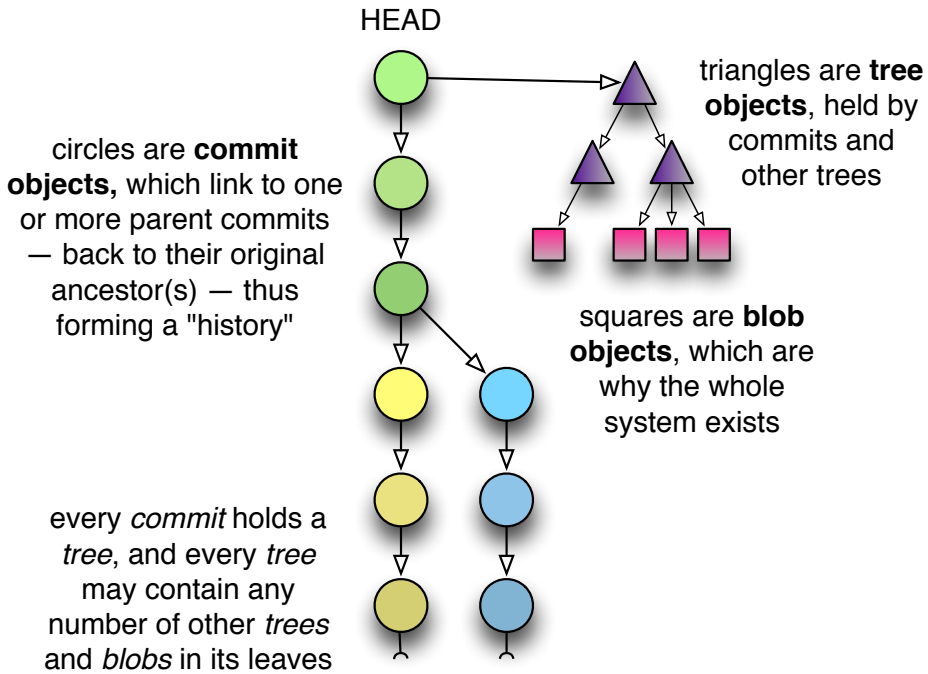
2. Well, and tags, but these are just fancy references to commits and can be ignored for the time being.

The difference here is that changed files in my working tree are preserved. If I pass the `-f` option to `checkout`, it acts the same in this case to `reset --hard`, except that `checkout` only ever changes the working tree, whereas `reset --hard` may change the current branch's HEAD reference.

The convenience of branch naming, specified by commit ids stored in `.git/refs/heads`, is that it makes life much easier, and it lets us know that these commits (and their history) are to be kept around. But never forget that they are simply aliases for specific commits. You could always work entirely with commit ids at any time, thus leaving names behind you forever. Want to see what your source tree looked like at a specific point in your commit log? Just checkout that commit's hash id.

Another joy of the commit-based system is that you can rephrase even the most complicated version control terminology using a single vocabulary. For example, if a commit has multiple parents, it's a "merge commit" — since it merged multiple commits into one. Or, if a commit has multiple children, it represents the ancestor of a "branch", etc. But really there is no difference between these things to Git: to it, the world is simply a hegemony of commit objects, each of which holds a tree that references other trees and blobs, which store your data. Anything more complicated than this is simply a device of nomenclature.

Here is a picture of how all these pieces fit together:



A commit by any other name…

Understanding commits is the key to grokking Git. You'll know you have reached the Zen plateau of branching wisdom when your mind contains only commit topologies, leaving behind the confusion of branches, tags, local and remote repositories, etc. Hopefully such understanding will not require lopping off your arm[3] — although I can appreciate if you've considered it by now.

If commits are the key, how you name commits is the doorway to mastery. There are many, many ways to name commits, ranges of commits, and even some of the objects held by commits, which are accepted by most of the Git commands. Here's a summary of some of the more basic usages:

branchname	As has been said before, the name of any branch is simply an alias for the most recent commit on that "branch". This is the same as using the word **HEAD** whenever that branch is checked out.
tagname	A tag-name alias is identical to a branch alias in terms of naming a commit. The major difference between the two is that tag aliases never change, whereas branch aliases change each time a new commit is checked in to that branch.
HEAD	The currently checked out commit is always called HEAD. If you check out a specific commit — instead of a branch name — then HEAD refers to that commit only *and not to any branch*. Note that this case is somewhat special, and is called "using a detached HEAD" (I'm sure there's a joke to be told here...).
c82a22c39cbc32...	A commit may always be referenced using its full, 40-character SHA1 hash id. Usually this happens during cut-and-pasting, since there are typically other, more convenient ways to refer to the same commit.
c82a22c	You only need use as many digits of a hash id as are needed for a unique reference within the repository. Most of the time, six or seven digits is enough.

3. Cf. the example of the second Zen patriarch, Hui-k'o.

name^ The parent of any commit is referenced using the caret symbol. If a commit has more than one parent, the first is used.

name^^ Carets may be applied successively. This alias refers to "the parent of the parent" of the given commit name.

name^2 If a commit has multiple parents (such as a merge commit), you can refer to the *nth* parent using **name^*n***.

name~10 A commit's *nth* ancestor may be referenced using a tilde (~) followed by the ordinal number. This type of usage is common with `rebase -i`, for example, to mean "show me a bunch of recent commits". This is the same as **name**^^^^^^^^^^.

name:path To reference a certain file within a commit's content tree, specify that file's name after a colon. This is helpful with `show`, or to show the difference between two versions of a committed file:

```
$ git diff HEAD^1:Makefile HEAD^2:Makefile
```

name^{tree} You can reference just the tree held by a commit, rather than the commit itself.

name1..name2 This and the following aliases indicate **commit ranges**, which are supremely useful with commands like `log` for seeing what's happened during a particular span of time.

The syntax to the left refers to all the commits reachable from **name2** back to, but not including, **name1**. If either **name1** or **name2** is omitted, HEAD is used in its place.

name1...name2 A "triple-dot" range is quite different from the two-dot version above. For commands like `log`, it refers to all the commits referenced by **name1** or **name2**, but not by both. The result is then a list of all the unique commits in both branches.

For commands like `diff`, the range expressed is between **name2** and the common ancestor of **name1** and **name2**. This differs from the `log` case in that changes introduced by **name1** are not shown.

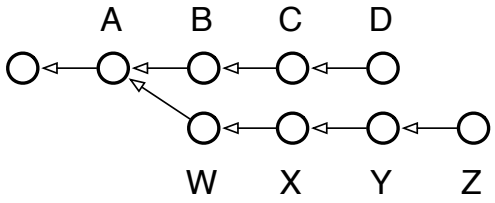
master..	This usage is equivalent to "`master..HEAD`". I'm adding it here, even though it's been implied above, because I use this kind of alias constantly when reviewing changes made to the current branch.
..master	This, too, is especially useful after you've done a `fetch` and you want to see what changes have occurred since your last `rebase` or `merge`.
--since="2 weeks ago"	Refers to all commits since a certain date.
--until="1 week ago"	Refers to all commits up to a certain date.
--grep=pattern	Refers to all commits whose commit message matches the regular expression *pattern*.
--committer=pattern	Refers to all commits whose committer matches *pattern*.
--author=pattern	Refers to all commits whose author matches *pattern*. The author of a commit is the one who created the changes it represents. For local development this is always the same as the committer, but when patches are being sent by e-mail, the author and the committer usually differ.
--no-merges	Refers to all commits in the range that have only one parent — that is, it ignores all merge commits.

Most of these options can be mixed-and-matched. Here is an example which shows the following log entries: changes made to the current branch (branched from `master`), by myself, within the last month, which contain the text "foo":

```
$ git log --grep='foo' --author='johnw' --since="1 month ago" master..
```

Branching and the power of rebase

One of Git's most capable commands for manipulating commits is the innocently-named `rebase` command. Basically, every branch you work from has one or more "base commits": the commits that branch was born from. Take the following typical scenario, for example. Note that the arrows point back in time because each commit references its parent(s), but not its children. Therefore, the `D` and `Z` commits represent the heads of their respective branches:



In this case, running `branch` would show two "heads": `D` and `Z`. The "base" of the `Z` branch is `A`, while the base of the `D` branch continues back to some unlabeled commit in the past. The output of `show-branch` shows us just this information:

```
$ git branch
  Z
* D

$ git show-branch
! [Z] Z
 * [D] D
--
 * [D] D
 * [D^] C
 * [D~2] B
+  [Z] Z
+  [Z^] Y
+  [Z~2] X
+  [Z~3] W
+* [D~3] A
```

Reading this output takes a little getting used to, but essentially it's no different from the diagram above. Here's what it tells us:

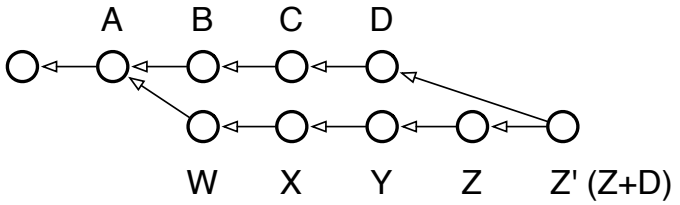
- The branch we're on (it doesn't matter if it's `D` or `Z` in this case), experienced its first divergence at commit `A` (also known as commit `D~3`, and even `Z~4` if you feel so inclined). The syntax `commit^` is used to refer to the parent of a commit, while `commit~3` refers to its third parent, or great-grandparent.
- Reading from bottom to top, the first column (the plus signs) shows a divergent branch named `Z` with four commits: `W`, `X`, `Y` and `Z`.
- The second column (the asterisks) show the commits which happened on the current branch, namely three commits: `B`, `C` and `D`.
- The top of the output, separated from the bottom by a dividing line, identifies the branches displayed, which column their commits are labelled by, and the character used for the labeling.

The action we'd like to perform is to bring the working branch `Z` back up to speed with the main branch, `D`. In other words, we want to incorporate the work from `B`, `C`, and `D` into `Z`.

In other version control systems this sort of thing can only be done using a "branch merge". In fact, a branch merge can still be done in Git, using `merge`, and remains needful in the case where `Z` is a published branch and we don't want to alter its commit history. Here are the commands to run:

```
$ git checkout Z        # switch to the Z branch
$ git merge D           # merge commits B, C and D into Z
```

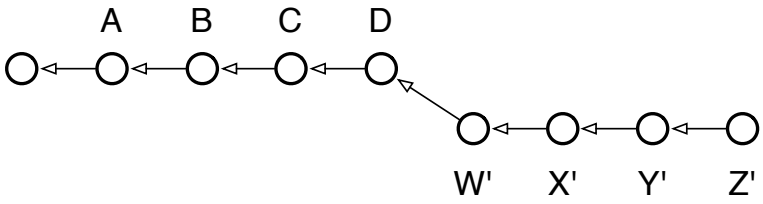
This is what the repository looks like afterward:



If we checked out the `Z` branch now, it would contain the contents of the previous `Z` (now referenceable as `Z^`), merged with the contents of `D`. (Though note: a real merge operation would have required resolving any conflicts between the states of `D` and `Z`).

Although the new `Z` now contains the changes from `D`, it also includes a new commit to represent the merging of `Z` with `D`: the commit now shown as `Z'`. This commit doesn't add anything new, but represents the work done to bring `D` and `Z` together. In a sense it's a "meta-commit", because its contents are related to work done solely in the repository, and not to new work done in the working tree.

There is a way, however, to transplant the `Z` branch straight onto `D`, effectively moving it forward in time: by using the powerful `rebase` command. Here's the graph we're aiming for:



This state of affairs most directly represents what we'd like done: for our local, development branch `Z` to be based on the latest work in the main branch `D`. That's why the command is called "rebase", because it changes the base commit of the branch it's run from. If you run it repeatedly,

you can carry forward a set of patches indefinitely, always staying up-to-date with the main branch, but without adding unnecessary merge commits to your development branch. Here are the commands to run, compared to the merge operation performed above:

```
$ git checkout Z        # switch to the Z branch
$ git rebase D          # change Z's base commit to point to D
```

Why is this only for local branches? Because every time you rebase, you're potentially changing every commit in the branch. Earlier, when `W` was based on `A`, it contained only the changes needed to transform `A` into `W`. After running `rebase`, however, `W` will be rewritten to contain the changes necessary to transform `D` into `W'`. Even the transformation from `W` to `X` is changed, because `A+W+X` is now `D+W'+X'` — and so on. If this were a public branch, and any of your downstream consumers had created their own local branches off of `Z`, their branches would now point to the old `Z`, not the new `Z'`.

Generally, the following rule of thumb can be used: Use `rebase` if you have a local branch with no other sub-branches or users, and use `merge` for all other cases. `merge` is also useful when you're ready to pull your local branch's changes back into the main branch.

Interactive rebasing

When `rebase` was run above, it automatically rewrote all the commits from W to Z in order to rebase the Z branch onto the D commit (i.e., the head commit of the D branch). You can, however, take complete control over how this rewriting is done. If you supply the -i option to `rebase`, it will pop you into an editing buffer where you can choose what should be done for every commit in the local `Z` branch:

pick This is the default behavior chosen for every commit in the branch if you don't use interactive mode. It means that the commit in question should be applied to its (now rewritten) parent commit. For every commit that involves conflicts, the `rebase` command gives you an opportunity to resolve them.

squash A squashed commit will have its contents "folded" into the contents of the commit preceding it. This can be done any number of times. If you took the example branch above and squashed all of its commits (except the first, which must be a **pick** in order to **squash**), you would end up with a new `Z` branch containing only one commit on top of `D`. Useful if you have changes spread over multiple commits, but you'd like the history rewritten to show them all as a single commit.

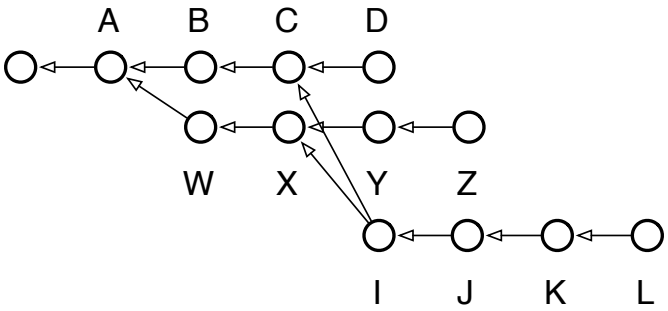
edit If you mark a commit as **edit**, the rebasing process will stop at that commit and leave you at the shell with the current working tree set to reflect that commit. The index will have all the commit's changes registered for inclusion when you run `commit`. You can thus make whatever changes you like: amend a change, undo a change, etc.; and after committing, and running `rebase --continue`, the commit will be rewritten as if those changes had been made originally.

(drop) If you remove a commit from the interactive rebase file, or if you comment it out, the commit will simply disappear as if it had never been checked in. Note that this can cause merge conflicts if any of the later commits in the branch depended on those changes.

The power of this command is hard to appreciate at first, but it grants you virtually unlimited control over the shape of any branch. You can use it to:

- Collapse multiple commits into single ones.
- Re-order commits.
- Remove incorrect changes you now regret.
- Move the base of your branch *onto any other commit in the repository*.
- Modify a single commit, to amend a change long after the fact.

I recommend reading the man page for `rebase` at this point, as it contains several good examples how the true power of this beast may be unleashed. To give you one last taste of how potent a tool this is, consider the following scenario and what you'd do if one day you wanted to migrate the secondary branch `L` to become the new head of `Z`:

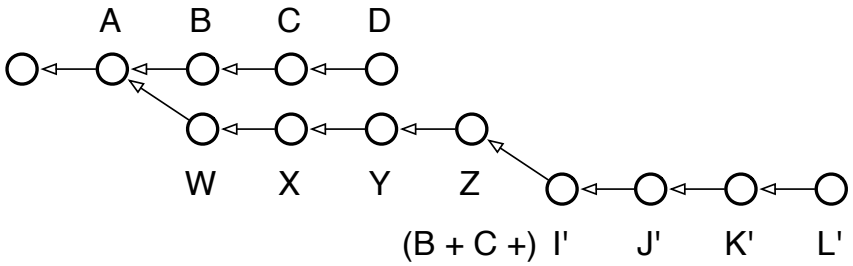


The picture reads: we have our main-line of development, `D`, which three commits ago was branched to begin speculative development on `Z`. At some point in the middle of all this, back

when C and X were the heads of their respective branches, we decided to begin another speculation which finally produced L. Now we've found that L's code is good, but not quite good enough to merge back over to the main-line, so we decide to move those changes over to the development branch Z, making it look as though we'd done them all on one branch after all. Oh, and while we're at it, we want to edit J real quick to change the copyright date, since we forgot it was 2008 when we made the change! Here are the commands needed to untangle this knot:

```
$ git checkout L
$ git rebase -i Z
```

After resolving whatever conflicts emerge, I now have this repository:

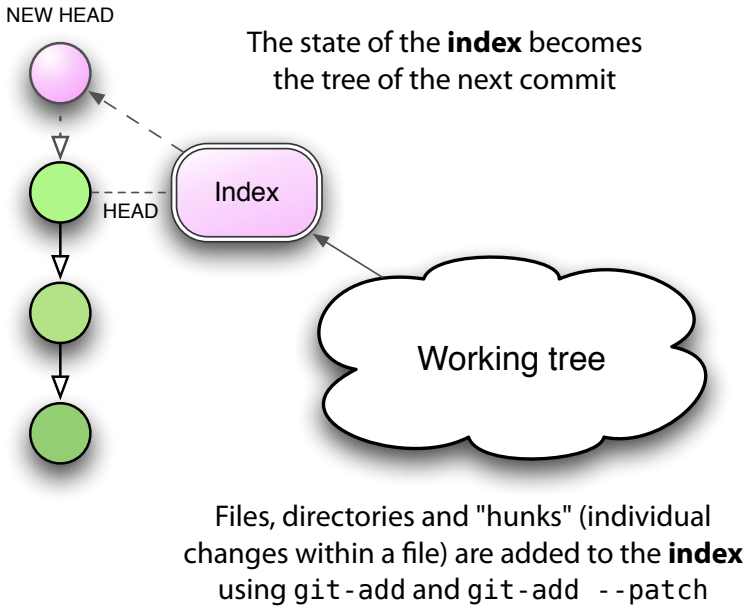


As you can see, when it comes to local development (public development requires the use of merge), rebasing gives you unlimited control over how your commits appear in the repository.

The Index: Meet the middle man

Between your data files, which are stored on the filesystem, and your Git blobs, which are stored in the repository, there stands a somewhat strange entity: the Git index. Part of what makes this beast hard to understand is that it's got a rather unfortunate name. It's an index in the sense that it refers to the set of newly created trees and blobs which you created by running add. These new objects will soon get bound into a new tree for the purpose of committing to your repository — but until then, they are only referenced by the index. That means that if you unregister a change from the index with reset, you'll end up with an orphaned blob that will get deleted at some point at the future.

The index is really just a staging area for your next commit, and there's a good reason why it exists: it supports a model of development that may be foreign to users of CVS or Subversion, but which is all too familiar to Darcs users: the ability to build up your next commit in stages.



First, let me say that there is a way to ignore the index almost entirely: by passing the `-a` flag to `commit`. Look at the way Subversion works, for example. When you type `svn status`, what you'll see is a list of actions to be applied to your repository on the next call to `svn commit`. In a way, this "list of next actions" is a kind of informal index, determined by comparing the state of your working tree with the current head of the repository. If the file `foo.c` has been changed, on your next commit those changes will be saved. If an unknown file has a question mark next to it, it will be ignored; but a new file which has been added with `svn add` will get added to the repository.

This is no different from what happens if you use `commit -a`: new, unknown files are ignored, but new files which have been added with `add` are added to the repository, as are any changes to existing files. This interaction is nearly identical with the Subversion way of doing things.

The real difference is that in the Subversion case, your "list of next actions" is always determined by looking at the current working tree. In Git, the "list of next actions" *is* the contents of the index, which can be manipulated by the user manually before executing `commit`. This gives you an extra layer of control over what's going to happen, by allowing you to stage those changes in advance.

If this isn't clear yet, consider the following example: you have a trusty source file, `foo.c`, and you've made two sets of unrelated changes to it. What you'd like to do is to tease apart these

changes into two different commits, each with its own description. Here's how you'd do this in Subversion:

```
$ svn diff foo.c > foo.patch
$ vi foo.patch
<edit foo.patch, keeping the changes I want to commit later>
$ patch -p1 -R < foo.patch    # remove the second set of changes
$ svn commit -m "First commit message"
$ patch -p1 < foo.patch       # re-apply the remaining changes
$ svn commit -m "Second commit message"
```

Sounds gross? Yeah, if it reminds you of vomiting in your mouth, you're close. Now repeat that many times over for a complex, dynamic set of changes.

Here's the Git version, making use of the index:

```
$ git add --patch foo.c
<select the hunks I want to commit first>
$ git commit -m "First commit message"
$ git add foo.c               # add the remaining changes
$ git commit -m "Second commit message"
```

What's more, it gets even easier! If you like Emacs, the superlative tool `gitsum.el`[4], by Christian Neukirchan, puts a beautiful face on this potentially tedious process. I recently used it to tease apart 11 separate commits from a set of conflated changes. Thank you, Christian!

Taking the index farther

Let's see, the index... With it you can pre-stage a set of changes, thus iteratively building up a patch before committing it to the repository. Now, where have I have heard that concept before...

If you're thinking "Quilt!", you're exactly right. In fact, the index is little different from Quilt[5], it just adds the restriction of allowing only one patch to be constructed at a time.

But what if, instead of two sets of changes within `foo.c`, I had four? With plain Git, I'd have to tease each one out, commit it, and then tease out the next. This is made much easier using the index, but what if I wanted to test those changes in various combination with each other before checking them in? That is, if I labelled the patches A, B, C and D, what if I wanted to

4. http://chneukirchen.org/blog/archive/2008/02/introducing-gitsum.html

5. http://savannah.nongnu.org/projects/quilt

test A + B, then A + C, then A + D, etc., before deciding if any of the changes were truly complete?

There is no mechanism in Git itself that allows you to mix and match parallel sets of changes on the fly. Sure, multiple branches can let you do parallel development, and the index lets you stage multiple changes into a series of commits, but you can't do both at once: staging a series of patches while at the same time selectively enabling and disabling some of them, to verify the integrity of the patches in concert before finally committing them.

What you'd need to do something like this is an index which allows for greater depth than one commit at a time. This is exactly what Stacked Git[6] provides.

Here's how I'd commit two different patches into my working tree using plain Git:

```
$ git add -i             # select first set of changes
$ git commit -m "First commit message"
$ git add -i             # select second set of changes
$ git commit -m "Second commit message"
```

This works great, but I can't selectively disable the first commit in order to test the second one alone. To do that, I'd have to do the following:

```
$ git log                # find the hash id of the first commit
$ git checkout -b work <first commit's hash id>^
$ git cherry-pick <second commit's hash id>
<...  run tests ...>
$ git checkout master    # go back to the master "branch"
$ git branch -D work     # remove my temporary branch
```

Surely there has to be a better way! With `stg` I can queue up both patches and then re-apply them in whatever order I like, for independent or combined testing, etc. Here's how I'd queue the same two patches from the previous example, using `stg`:

```
$ stg new patch1
$ git add -i             # select first set of changes
$ stg refresh --index
$ stg new patch2
$ git add -i             # select second set of changes
$ stg refresh --index
```

6. http://procode.org/stgit

Now if I want to selectively disable the first patch to test only the second, it's very straightforward:

```
$ stg applied
patch1
patch2
<...  do tests using both patches ...>
$ stg pop patch1
<...  do tests using only patch2 ...>
$ stg pop patch2
$ stg push patch1
<...  do tests using only patch1 ...>
$ stg push -a
$ stg commit -a          # commit all the patches
```

This is definitely easier than creating temporary branches and using `cherry-pick` to apply specific commit ids, followed by deleting the temporary branch.

To reset, or not to reset

One of the more difficult commands to master in Git is `reset`, which seems to bite people more often than other commands. Which is understandable, giving that it has the potential to change both your working tree and your current HEAD reference. So I thought a quick review of this command would be useful.

Basically, `reset` is a reference editor, an index editor, and a working tree editor. This is partly what makes it so confusing, because it's capable of doing so many jobs. Let's examine the difference between these three modes, and how they fit into the Git commit model.

Doing a mixed reset

If you use the `--mixed` option (or no option at all, as this is the default), `reset` will revert parts of your index along with your HEAD reference to match the given commit. The main difference from `--soft` is that `--soft` only changes the meaning of HEAD and doesn't touch the index. This is the default mode of operation for `reset`.

```
$ git add foo.c              # add changes to the index as a new blob
$ git reset HEAD foo.c       # delete blob from the index
$ git add foo.c              # made a mistake, add it back
```

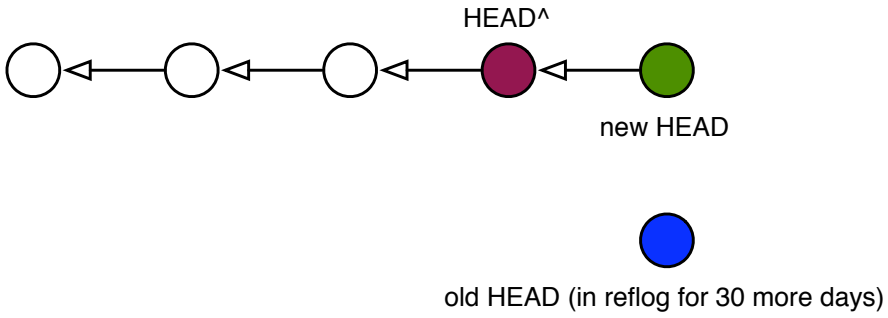
Doing a soft reset

If you use the `--soft` option to `reset`, this is the same as simply changing your HEAD reference to a different commit. Your working tree changes are left untouched. This means the following two commands are equivalent:

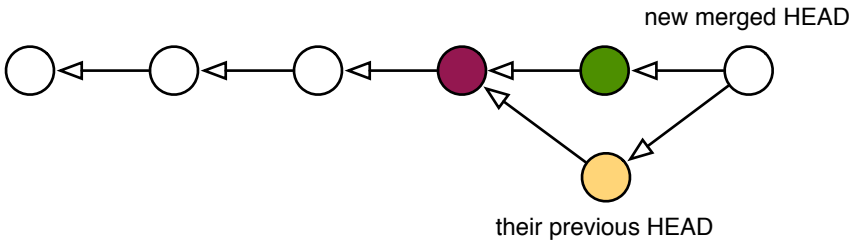
```
$ git reset --soft HEAD^     # backup HEAD to its parent,
                             # effectively ignoring the last commit
$ git update-ref HEAD HEAD^  # does the same thing, albeit manually
```

In both cases, your working tree now sits on top of an older HEAD, so you should see more changes if you run `status`. It can give you a chance to create a new commit in place of the old one. In fact, if the commit you want to change is the most recent one checked in, you can use `commit --amend` to add your latest changes to the last commit as if you'd done them together.

But please note: if you have downstream consumers, and they've done work on top of your previous head — the one you threw away — changing HEAD like this will force a merge to happen automatically after their next pull. Below is what your tree would look like after a soft reset and a new commit:



And here's what your consumer's HEAD would look like after they pulled again, with colors to show how the various commits match up:



Doing a hard reset

A hard reset (the `--hard` option) has the potential of being very dangerous, as it's able to do two different things at once: First, if you do a hard reset against your current HEAD, it will erase all changes in your working tree, so that your current files match the contents of HEAD.

There is also another command, `checkout`, which operates just like `reset --hard` if the index is empty. Otherwise, it forces your working tree to match the index.

Now, if you do a hard reset against an earlier commit, it's the same as first doing a soft reset and then using `reset --hard` to reset your working tree. Thus, the following commands are equivalent:

```
$ git reset --hard HEAD~3   # Go back in time, throwing away changes
```

```
$ git reset --soft HEAD~3   # Set HEAD to point to an earlier commit
$ git reset --hard          # Wipe out differences in the working tree
```

As you can see, doing a hard reset can be very destructive. Fortunately, there is a safer way to achieve the same effect, using the Git stash (see the next section):

```
$ git stash
$ git checkout -b new-branch HEAD~3   # head back in time!
```

This approach has two distinct advantages if you're not sure whether you really want to modify the current branch just now:

1. It saves your work in the stash, which you can come back to at any time. Note that the stash is not branch specific, so you could potentially stash your changes while on one branch, and later apply them to another.
2. It reverts your working tree back to a past state, but on a new branch, so if you decide to commit your changes against the past state, you won't have altered your original branch.

If you do make changes to `new-branch` and then decide you want it to become your new master branch, run the following commands:

```
$ git branch -D master            # goodbye old master (still in reflog)
$ git branch -m new-branch master  # the new-branch is now my master
```

The moral of this story is: although you can do major surgery on your current branch using `reset --soft` and `reset --hard` (which changes the working tree too), why would you want to? Git makes working with branches so easy and cheap, it's almost always worth it to do your

destructive modifications on a branch, and then move that branch over to take the place of your old master. It has an almost Sith-like appeal to it...

And what if you do accidentally run `reset --hard`, losing not only your current changes but also removing commits from your master branch? Well, unless you've gotten into the happen of using `stash` to take snapshots (see next section), there's nothing you can do to recover your lost working tree. But you can restore your branch to its previous state by again using `reset --hard` with the reflog (this will also be explained in the next section):

```
$ git reset --hard HEAD@{1}   # restore from reflog before the change
```

To be on the safe side, never use `reset --hard` without first running `stash`. It will save you many white hairs later on. If you did run `stash`, you can now use it to recover your working tree changes as well:

```
$ git stash                   # because it's always a good thing to do
$ git reset --hard HEAD~3     # go back in time

$ git reset --hard HEAD@{1}   # oops, that was a mistake, undo it!
$ git stash apply             # and bring back my working tree changes
```

Last links in the chain: Stashing and the reflog

Until now we've described two ways in which blobs find their way into Git: first they're created in your index, both without a parent tree and without an owning commit; and then they're committed into the repository, where they live as leaves hanging off of the tree held by that commit. But there are two other ways a blob can dwell in your repository.

The first of these is the Git *reflog*, a kind of meta-repository that records — in the form of commits — every change you make to your repository. This means that when you create a tree from your index and store it under a commit (all of which is done by `commit`), you are also, inadvertently adding that commit to the reflog, which can be viewed using the following command:

```
$ git reflog
5f1bc85...  HEAD@{0}: commit (initial): Initial commit
```

The beauty of the reflog is that it persists independently of other changes in your repository. This means I could unlink the above commit from my repository (using `reset`), yet it would still be referenced by the reflog for another 30 days, protecting it from garbage collection. This gives me a month's chance to recover the commit should I discover I really need it.

The other place blobs can exist, albeit indirectly, is in your working tree itself. What I mean is, say you've changed a file `foo.c` but you haven't added those changes to the index yet. Git may

not have created a blob for you, but those changes do exist, meaning the content exists — it just lives in your filesystem instead of Git's repository. The file even has its own SHA1 hash id, despite the fact no real blob exists. You can view it with this command:

```
$ git hash-object foo.c
<some hash id>
```

What does this do for you? Well, if you find yourself hacking away on your working tree and you reach the end of a long day, a good habit to get into is to stash away your changes:

```
$ git stash
```

This takes all your directory's contents — including both your working tree, and the state of the index — and creates blobs for them in the git repository, a tree to hold those blobs, and a pair of stash commits to hold the working tree and index and record the time when you did the stash.

This is a good practice because, although the next day you'll just pull your changes back out of the stash with `stash apply`, you'll have a reflog of all your stashed changes at the end of every day. Here's what you'd do after coming back to work the next morning (WIP here stands for "Work in progress"):

```
$ git stash list
stash@{0}: WIP on master: 5f1bc85...  Initial commit

$ git reflog show stash  # same output, plus the stash commit's hash id
2add13e...  stash@{0}: WIP on master: 5f1bc85...  Initial commit

$ git stash apply
```

Because your stashed working tree is stored under a commit, you can work with it like any other branch — at any time! This means you can view the log, see when you stashed it, and checkout any of your past working trees from the moment when you stashed them:

```
$ git stash list
stash@{0}: WIP on master: 73ab4c1...  Initial commit
...
stash@{32}: WIP on master: 5f1bc85...  Initial commit

$ git log stash@{32}                # when did I do it?
$ git show stash@{32}               # show me what I was working on

$ git checkout -b temp stash@{32}   # let look at that old working tree!
```

This last command is particularly powerful: behold, I'm now playing around in an uncommitted working tree from over a month ago. I never even added those files to the index; I just

used the simple expedient of calling `stash` before logging out each day, and `stash apply` when I logged back in. You could even automate these actions with `.login` and `.logout` scripts.

If you ever want to clean up your stash list — say to keep only the last 30 days of activity — don't use `stash clear`; use the `reflog expire` command instead:

```
$ git stash clear        # DON'T!  You'll lose all that history

$ git reflog expire --expire=30.days refs/stash
<outputs the stash bundles that've been kept>
```

The beauty of `stash` is that it lets you apply unobtrusive version control to your working process itself: namely, the various stages of your working tree from day to day. You can even use `stash` on a regular basis if you like, with something like the following `snapshot` script:

```
$ cat <<EOF > /usr/local/bin/git-snapshot
#!/bin/sh
git stash && git stash apply
EOF
$ chmod +x $_

$ git snapshot
```

There's no reason you couldn't run this from a `cron` job every hour, along with running the `reflog expire` command every week or month.

Conclusion

Over the years I've used many version control systems, and many backup schemes. They all have facilities for retrieving the past contents of a file. Most of them have ways to show how a file has differed over time. Many permit you to go back in time, begin a divergent line of reasoning, and then later bring these new thoughts back to the present. Still fewer offer fine-grained control over that process, allowing you to collect your thoughts however you feel best to present your ideas to the public. Git lets you do all these things, and with relative ease — once you understand its fundamentals.

It's not the only system with this kind of power, nor does it always employ the best interface to its concepts. What it does have, however, is a solid base to work from. In the future, I imagine many new methodologies will be created to take advantage of the flexibilities Git allows. Most other systems have led me to believe they've reached their conceptual plateau — that all else from now will be only a slow refinement of what I've seen before. Git gives me the opposite

impression, however. I feel we've only begun to see the potential its deceptively simple design promises.

THE END

Further reading

If your interest to learn Git more has been piqued, please check out the following articles:

- *A tour of Git: the basics*
  http://cworth.org/hgbook-git/tour/
- *Manage source code using Git*
  http://www.ibm.com/developerworks/linux/library/l-git/
- *A tutorial introduction to git*
  http://www.kernel.org/pub/software/scm/git/docs/tutorial.html
- *GitFaq — GitWiki*
  http://git.or.cz/gitwiki/GitFaq
- *A git core tutorial for developers*
  http://www.kernel.org/pub/software/scm/git/docs/core-tutorial.html
- *git for the confused*
  http://www.gelato.unsw.edu.au/archives/git/0512/13748.html
- *The Thing About Git*
  http://tomayko.com/writings/the-thing-about-git