Git Internals

Source code control and beyond

by Scott Chacon & the communit

Figure 1: Git Internals book cover

version: 2.2.1-83-g9cace88

2024-11-21 10:33:32

Abstract

This document explains the internal workings of the Git source code control system.

It was originally written by Scott Chacon and published at PeepCode (now a part of Pluralsight).

Scott and Pluralsight have generously open sourced it. It is now maintained by the community - including you!

Contents

Summary

Understanding Git

Using Git

Annex

.

Commands Overview

References and Endnotes

Revisions

About this book

It took me a pretty long time to really get Git. As I've continued to use Git more and more where I work, I've found myself trying to teach people what it is and why we use it over and over again, and the reality is that Git generally has a pretty steep learning curve compared to many other systems. I've seen case after case of developers who love Git after they finally understand it, but getting to that point is often somewhat painstaking.

This book is aimed at the developer who does not particularly like Subversion, Perforce or whatever SCM system they are currently using, has heard good things about Git, but doesn't know where to start or why it's so wonderful. It is meant to explain Git as simply as possible in a clean, concise, easily readable volume. My goal is to help you understand Git internals as well as usage at a fundamental level by the time you finish this book.

To accomplish this, I'm starting the book out (after the introduction) with a section about what Git actually **does**, rather than how to use it. I found that I didn't really understand Git and had many problems using it until I understood what it was actually doing at a low level, rather than thinking of it as a different, weird SVN-like system.

Installing Git

Before we can start playing with Git, we'll have to install it. I'll quickly cover installing Git on Linux, Mac and Windows. I will not get into really fine detail, because others have done that much better, but I will give you an overview and links as to where to find more detailed instructions on each platform.

For any of these examples, you can find a link to the most current Git source code at https://github.com/git/git.

I would recommend compiling from source if possible, simply because Git is lately making big strides in usability, so more current versions may be a bit easier to use.

Installing on Linux

If you are installing from source, it will go something like the standard:

```
wget https://kernel.org/pub/software/scm/git/git-1.5.4.4.tar.bz2
tar jxpvf git-1.5.4.4.tar.bz2
cd git-1.5.4.4
make prefix=/usr all doc info
sudo make prefix=/usr install install-doc install-info
```

If you are running Ubuntu or another Debian based system, you can run

```
apt-get install git-core
```

or on yum based systems, you can often run:

```
yum install git-core
```

Installing on Mac

TODO This section is way outdated; needs revising

You are likely going to want to install Git without the asciidoc dependency, because it is a pain to install. Other than that, what you basically need is Curl and Expat. With the exception of the Leopard binary OS X installer, you will also need the Developer Tools installed. If you don't have the OS X install discs anymore, you can get the tools from **TODO**.

Windows

There are two options on Windows currently, but the popular one is https://gitforwindows.org/ (formerly MSysGit), which installs easily and can be run on the Windows command line. Simply download the exe file from the website, execute it and follow the on-screen instructions.

A Short History of Git

The Git project started with Linus Torvalds scratching the very serious itch of needing a fast, efficient and massively distributed source code management system for Linux kernel development.

The kernel team had moved from a patch emailing system to the proprietary BitKeeper SCM in 2002. That ended in April 2005 when BitMover stopped providing a free version of its tool to the open source community, because they felt some developers had reverse engineered it in violation of the license.

Since Linus had (and still has) a passionate dislike of just about all existing source code management systems, he decided to write his own. Thus, in April of 2005, Git was born. A few months later, in July, maintenance was turned over to Junio Hamano, who has maintained the project ever since.

> **NOTE**
> “I’m an egotistical bastard, and I name all my projects after myself. First Linux, now git.” - Linus

Git started out as a collection of lower level functions used in various combinations by shell and perl scripts. Recently (since 1.0), more and more of the scripts have been re-written in C (referred to as built-ins), increasing portability and speed.

Though originally used for just the Linux kernel, the Git project spread rapidly, and quickly became used to manage a number of other Linux projects, such as the X.org, Mesa3D, Wine, Fedora and Samba projects. Recently it has begun to spread outside the Linux world to manage projects such as Rubinius, Merb, Ruby on Rails, Nu, Io and many more major open source projects.

Understanding Git

In this section, we will go over what Git was built for and how it works, hopefully laying the groundwork to properly understand what it is doing when we run the commands.

> **NOTE**
> The first commit message for the Git project was 'initial version of "git", the information manager from hell' - Linus, 4/7/05

When I learned Git, as many people do, I learned it in the context of other SCMs I had used - Subversion or CVS. I have come to believe that this is a horrible way to learn Git. I felt far more comfortable with it when I stopped thinking that `git add` was sort of like 'svn add', but instead understood what it was actually doing. Then I found I could find new and interesting ways to use what is really a very powerful and cool toolset.

So, let's see what it's doing behind the scenes first.

What is Git?

Git is a stupid content tracker. That is probably the best description of it - don't think of it in a 'like (insert favorite SCM system), but. . . ' context, but more like a really interesting file system.

Git tracks content - files and directories. It is at its heart a collection of simple tools that implement a tree history storage and directory content management system. It is simply used as an SCM, not really designed as one.

> **NOTE**
> "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." - Linus

When most SCMs store a new version of a project, they store the code delta or diff. When Git stores a new version of a project, it stores a new *tree* - a bunch of blobs of content and a collection of pointers that can be expanded back out into a full directory of files and subdirectories. If you want a diff between two versions, it doesn't add up all the deltas, it simply looks at the two trees and runs a new diff on them.

This is what fundamentally allows the system to be easily distributed - it doesn't have issues figuring out how to apply a complex series of deltas, it simply transfers all the directories and content that one user has and another does not have but is requesting. It is efficient about it - it only stores identical files and directories once and it can compress and transfer its content using delta-compressed packfiles - but in concept, it is a very simple beast. Git is at its heart very stupid-simple.

Focus and Design

There are a number of areas that the developers of Git, including and especially Linus, have focused on in conceiving and building Git. There may be a lot of things that Git is not good at, but these things are what Git is *very* good at.

Non-Linear Development

Git is optimized for cheap and efficient branching and merging. It is built to be worked on simultaneously by many people, having multiple branches developed by individual developers, being merged, branched and re-merged constantly. Because of this, branching is incredibly cheap and merging is incredibly easy.

Distributed Development

Git is built to make distributed development simple. No repository is special or central in Git - each clone is basically equal and could generally replace any other one at any time. It works completely offline or with hundreds of remote repositories that can push to and/or fetch from each other over several simple and standard protocols.

Efficiency

Git is very efficient. Compared to many popular SCM systems, it seems downright unbelievably fast. Most operations are local, which reduces unnecessary network overhead. Repositories are generally packed very efficiently, which often leads to surprisingly small repo sizes.

> **NOTE**
> The Ruby on Rails Git repository download, which includes the full history of the project - every version of every file, weighs in at around 13M, which is not even twice the size of a single checkout of the project (~9M). The Subversion server repository for the same project is about 115M.

Git also is efficient in its network operations - the common Git transfer protocols transfer only packed versions of only the objects that have changed. It also won't try to transfer content twice, so if you have the same file under two different names, it will only transfer the content once.

A Toolkit Design

Git is not really a single binary, but a collection of dozens of small specialized programs, which is sometimes annoying to people trying to learn Git, but is pretty cool when you want to do anything non-standard with it. Git is less a program and more a toolkit that can be combined and chained to do new and interesting things.

> **NOTE**
> For a long time, Git was just the raw toolkit and the project to wrap those into a user friendly SCM was called *Cogito*. That project has since been abandoned as Git itself became easier to use.

The tools can be more or less divided into two major camps, often referred to as the *porcelain* and the *plumbing*. The plumbing is not really meant to be used by people on the command line, but rather to do simple things flexibly and are combined by programs and scripts into porcelain programs. The porcelain programs are largely what we will be focusing on in this book — the user-oriented interfaces to do SCM type things — hiding the low-level fun.

Git Object Types

Git *objects* are the actual data of Git, the main thing that the repository is made up of. There are four main object types in Git, the first three being the most important to really understand the main functions of Git.

All of these types of objects are stored in the Git **Object Database**, which is kept in the **Git Directory**. Each object is compressed (with Zlib) and referenced by the SHA-1 value of its contents plus a small header. In the examples, I will use the first 6 characters of the SHA-1 for simplicity, but the actual value is 40 characters long.

> **NOTE**
> SHA stands for Secure Hash Algorithm. A SHA creates an identifier of fixed length

that uniquely identifies a specific piece of content. SHA-1 succeeded SHA-0 and is the most commonly used algorithm. Wikipedia has more on the topic.

To demonstrate these examples, we will develop a small ruby library that provides very simple bindings to Git, keeping the project in a Git repository. The basic layout of the project is this:

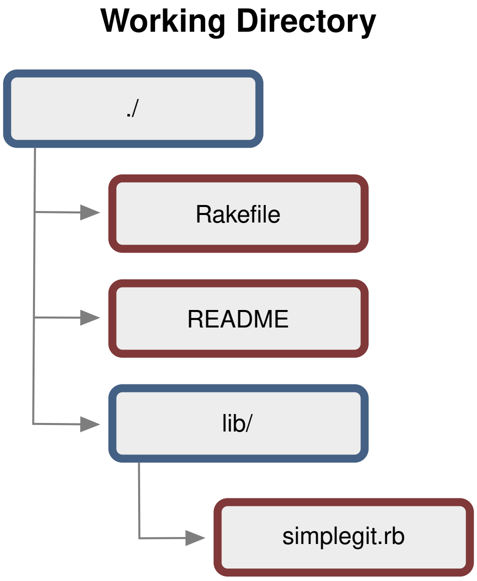


Figure 2: Sample project with files and directories

Let's take a look at what Git does when this is committed to a repository.

The Blob

In Git, the contents of files are stored as **blobs**.

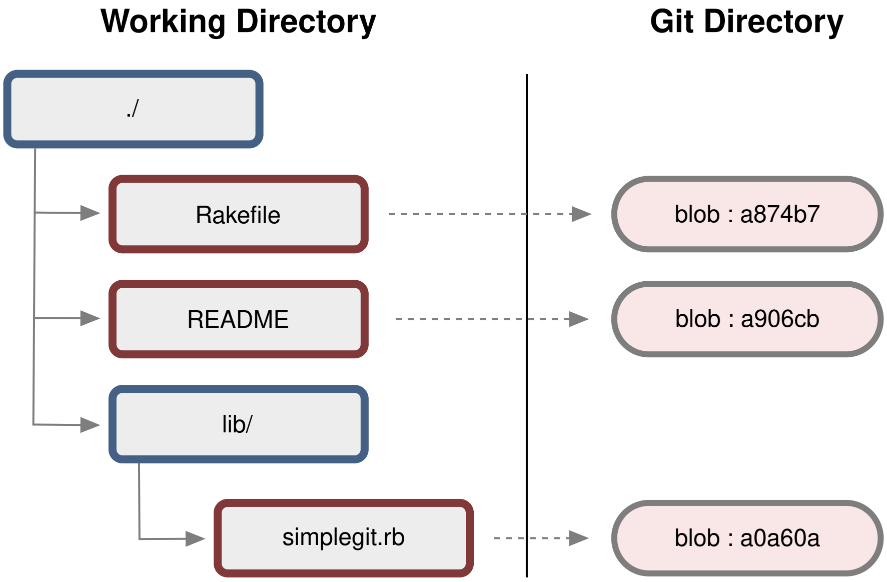
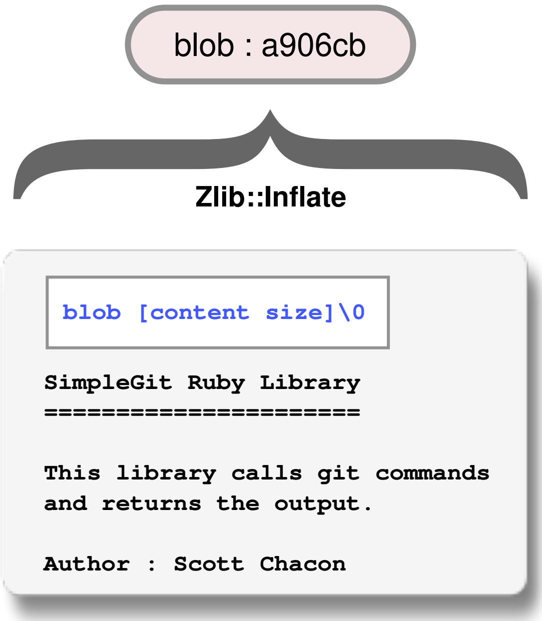


Figure 3: Files are stored as blobs

It is important to note that it is the *contents* that are stored, not the files. The names and modes of the files are not stored with the blob, just the contents.

This means that if you have two files anywhere in your project that are exactly the same, even if they have different names, Git will only store the blob once. This also means that during repository transfers, such as clones or fetches, Git will only transfer the blob once, then expand it out into multiple files upon checkout.



```
blob [content size]\0

SimpleGit Ruby Library
======================

This library calls git commands
and returns the output.

Author : Scott Chacon
```

Figure 4: The contents of a blob, uncompressed

The Tree

Directories in Git basically correspond to **trees**.

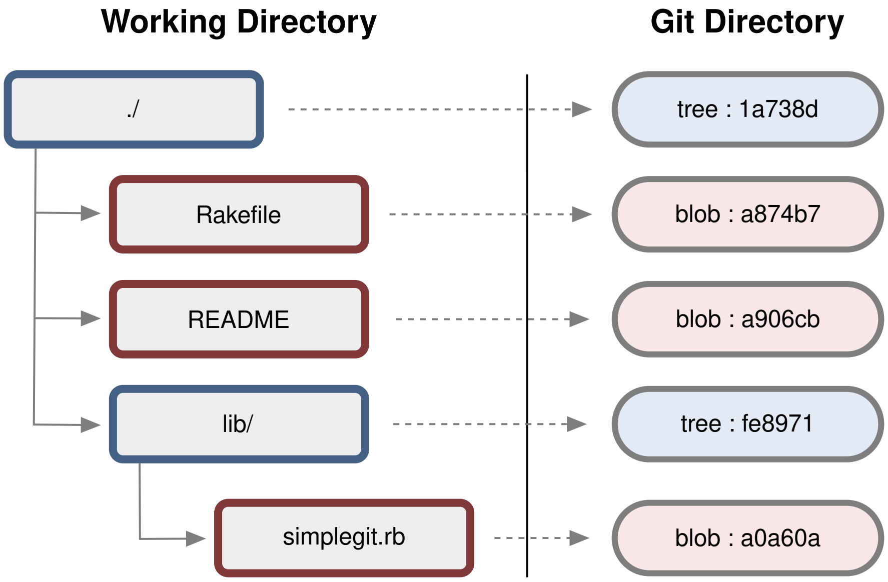


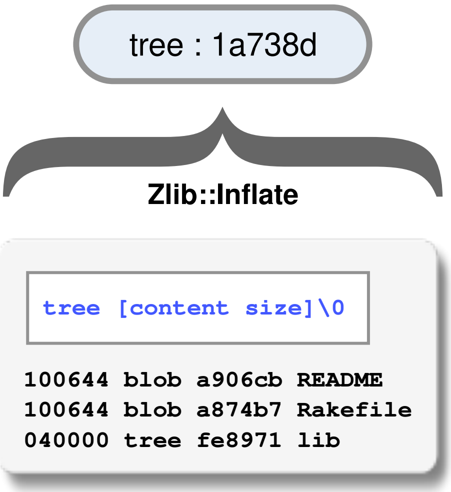
Figure 5: Trees are pointers to blobs and other trees

A tree is a simple list of trees and blobs that the tree contains, along with the names and modes of those trees and blobs. The contents of a tree object file is a list of entries in a compact, binary format (`git cat-file tree <sha>`), which simply consists of *mode*, *type*, *name* and *sha* for each entry. You can see it in a clear-text, human-readable form with `git cat-file -p <sha>`, which shows *mode*, *type*, *sha* and *name* for each entry (same fields, different order).

The Commit

So, now that we can store arbitrary trees of content in Git, where does the 'history' part of 'tree history storage system' come in? The answer is the **commit** object.

The commit is very simple, much like the tree. It simply points to a tree and keeps an *author*, *committer*, *message* and any *parent* commits that directly preceded it.



```
tree [content size]\0

100644 blob a906cb README
100644 blob a874b7 Rakefile
040000 tree fe8971 lib
```

Figure 6: An uncompressed tree

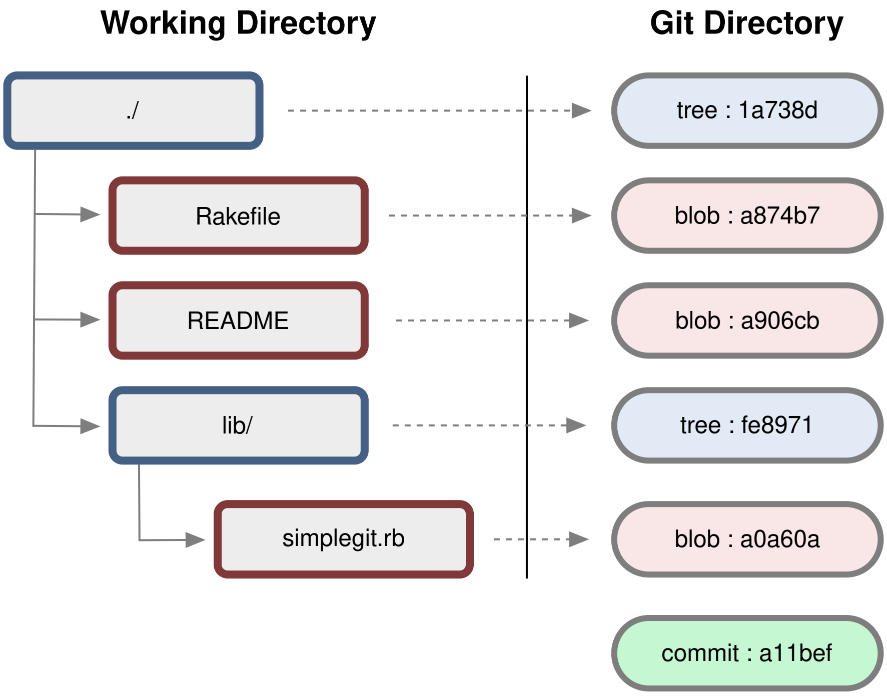
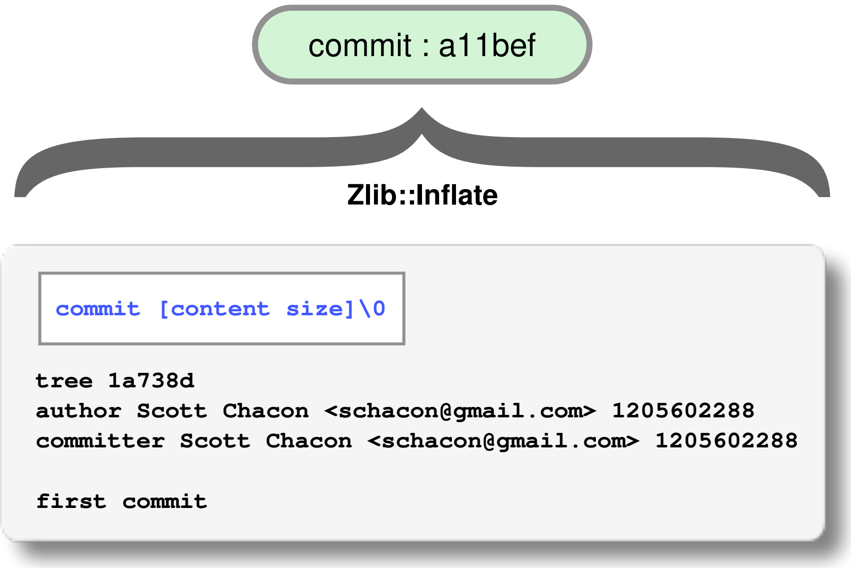


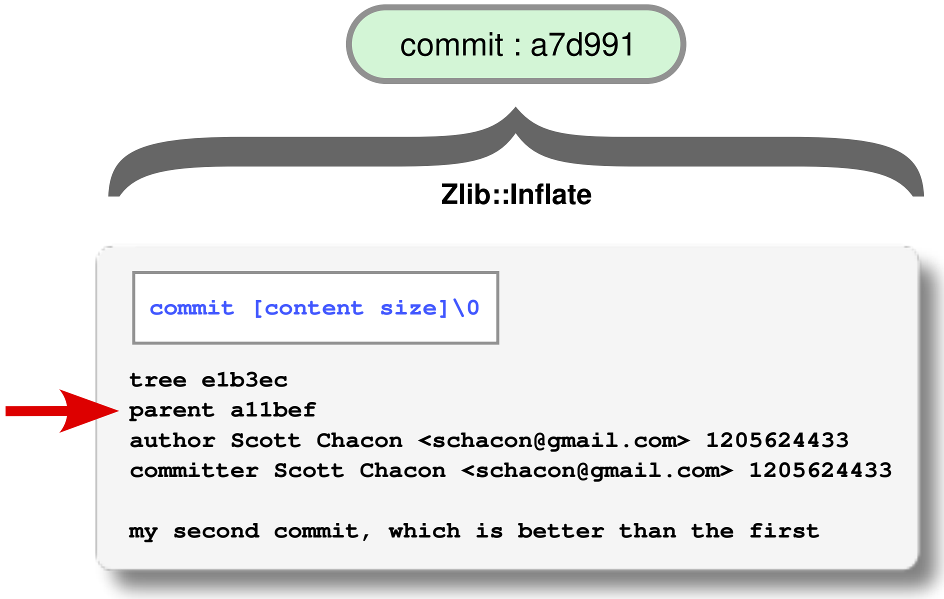
Figure 7: A commit references a tree



```
commit [content size]\0

tree 1a738d
author Scott Chacon <schacon@gmail.com> 1205602288
committer Scott Chacon <schacon@gmail.com> 1205602288

first commit
```

Figure 8: Uncompressed initial commit

Since this was my first commit, there are no parents. If I commit a second time, the commit object will look more like this:



```
commit [content size]\0

tree e1b3ec
parent a11bef
author Scott Chacon <schacon@gmail.com> 1205624433
committer Scott Chacon <schacon@gmail.com> 1205624433

my second commit, which is better than the first
```

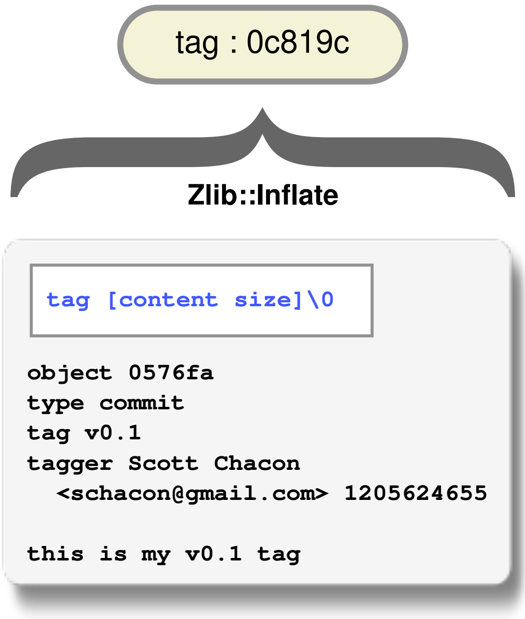
Figure 9: A commit with a parent

Notice how the *parent* in that commit is the same SHA-1 value of the last commit we did? Most times a commit will only have a single parent like that, but if you merge two branches, the next commit will point to both of them.

> **NOTE**
> The current record for number of commit parents in the Linux kernel is 12 branches merged in a single commit!

The Tag

The final type of object you will find in a Git database is the **tag**. This is an object that provides a permanent shorthand name for a particular commit. It contains an *object*, *type*, *tag*, *tagger* and a *message*. Normally the *type* is `commit` and the *object* is the SHA-1 of the commit you're tagging. The tag can also be GPG signed, providing cryptographic integrity to a release or version.



```
tag [content size]\0

object 0576fa
type commit
tag v0.1
tagger Scott Chacon
  <schacon@gmail.com> 1205624655

this is my v0.1 tag
```

Figure 10: Uncompressed tag

We'll talk a little bit more about tags and how they differ from *branches* (which also point to commits, but are not stored as objects) in the next section, where we'll pull all of this together into how all these objects relate to each other conceptually.

The Git Data Model

In computer science speak, the Git object data is a *directed acyclic graph*. That is, starting at any commit you can traverse its parents in one direction and there is no chain that begins and ends with the same object.

All commit objects point to a tree and optionally to previous commits. All trees point to one or many blobs and/or trees. Given this simple model, we can store and retrieve vast histories of complex trees of arbitrarily changing content quickly and efficiently.

This section is meant to demonstrate how that model looks.

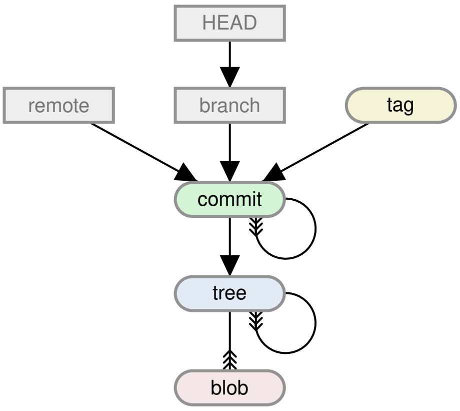
References

In addition to the Git objects, which are immutable - that is, they cannot ever be changed, there are references also stored in Git. Unlike the objects, references can constantly change. They are simple pointers to a particular commit, something like a tag, but easily moveable.

Examples of references are branches and remotes. A branch in Git is nothing more than a file in the `.git/refs/heads/` directory that contains the SHA-1 of the most recent commit of that branch. To branch that line of development, all Git does is create a new file in that directory that points to the same SHA-1. As you continue to commit, one of the branches will keep changing to point to the new commit SHA-1s, while the other one can stay where it was. If this is confusing, don't worry. We'll go over it again later.

The Model

The basic data model I've been explaining looks something like this:



The cheap references I've represented as the grey boxes, the immutable objects are the colored round cornered boxes.

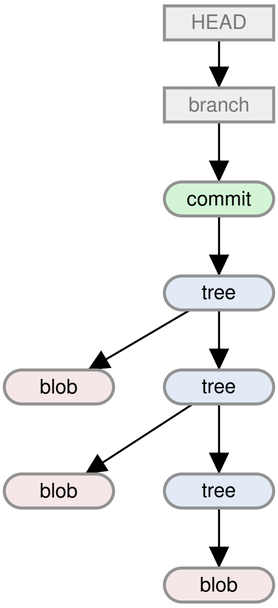
An Example

Lets look at an example of simple usage of Git and which objects are stored in the Git object database as we go.

To begin with, we commit an initial tree of three files and two subdirectories, each directory with one file in it. Possibly something like this:

```
.
|-- init.rb
`-- lib
    |-- base
    |   `-- base_include.rb
    `-- my_plugin.rb
```

When we first commit this tree, our Git model may look something like this:

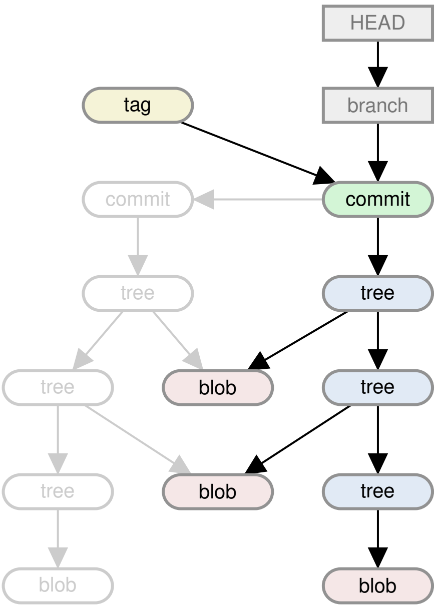


We have three trees, three blobs and a single commit that points to the top of the tree. The current branch points to our last commit and the `HEAD` file points to the branch we're currently on. This lets Git know which commit will be the parent for the next commit.

Now let's assume that we change the `lib/base/base_include.rb` file and commit again. At this point, a new blob is added, which changes the tree that points to it, which changes the tree that points to that tree and so on, to the top of the entire directory. Then a new commit object is added which points to its parent and the new tree, then the branch reference is moved forward.

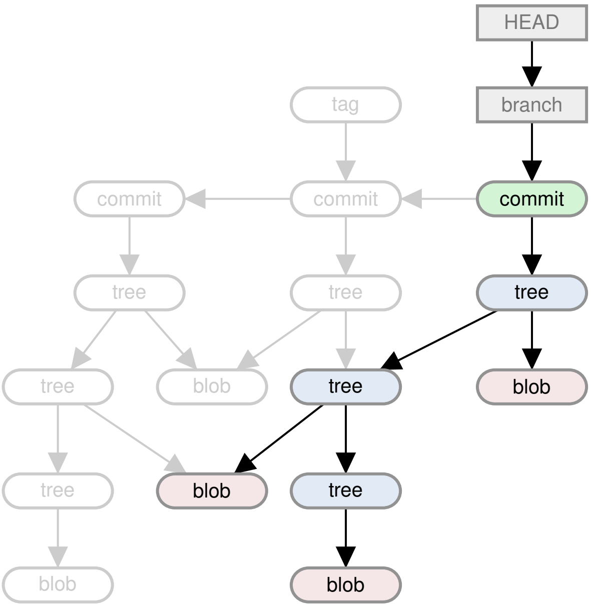
Let's also say at this point we tag this commit as a release, which adds a new tag object.

At this point, we'll have the following in Git:



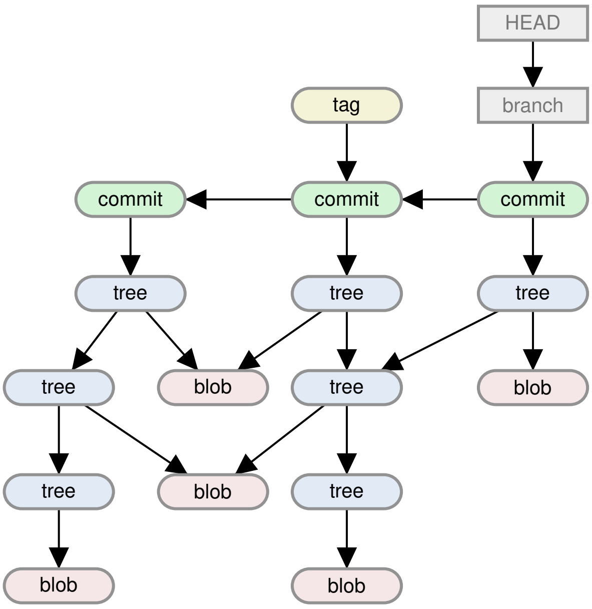
Notice how the other two blobs that were not changed were not added again. The new trees that were added point to the same blobs in the data store that the previous trees pointed to.

Now let's say we modify the `init.rb` file at the base of the project. The new blob will have to be added, which will add a new top tree, but all the subtrees will not be modified, so Git will reuse those references. Again, the branch reference will move forward and the new commit will point to its parent.



At this point, let's stop to look at the objects we now have in our repository. From this, we can easily recreate any of the three directories we committed by following the graph from the most recent commit object, and having Git expand the trees that are pointed to.

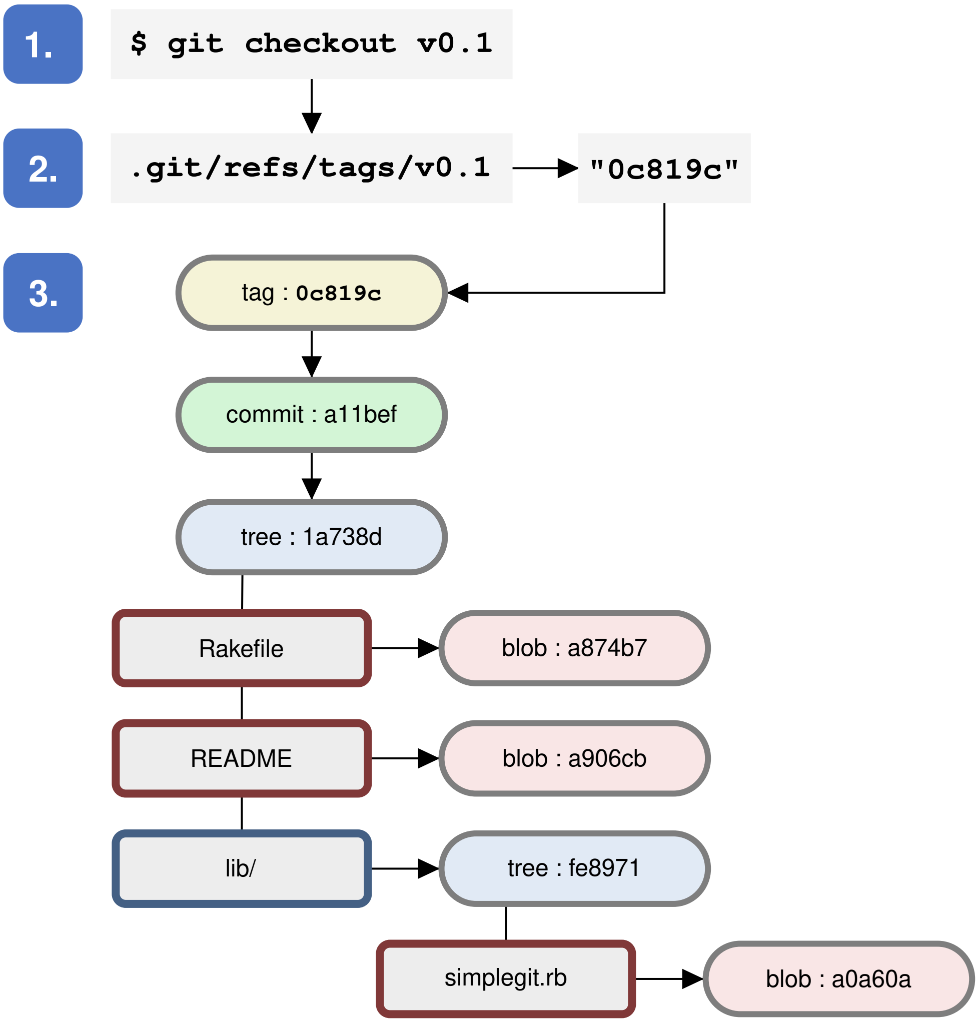
For instance, if we wanted the first tree, we could look for the parent of the parent of the HEAD, or the parent of the tag. If we wanted the second tree, we could ask for the commit pointed to by the tag, and so on.



So, to keep all the information and history on the three versions of this tree, Git stores 16 immutable, signed, compressed objects.

Traversal

So, what do all the arrows in these illustrations really mean? How does Git actually retrieve these objects in practice? Well, it gets the initial SHA-1 of the starting commit object by looking in the `.git/refs` directory for the branch, tag or remote you specify. Then it traverses the objects by walking the trees one by one, checking out the blobs under the names listed.



1.

```
$ git checkout v0.1
```

2.

```
.git/refs/tags/v0.1
```

```
"0c819c"
```

3.

Branching and Merging

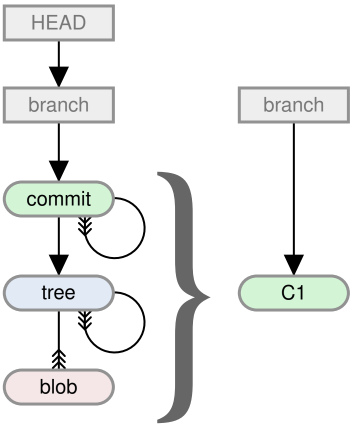
Here we come to one of the real strengths of Git, cheap inline branching. This is a feature that truly sets it apart and will likely change the way you think about developing code once you get used to it.

When you are working on code in Git, storing trees in any state and keeping pointers to them is very simple, as we've seen. In fact, in Git, the act of creating a new branch is simply writing a file in the `.git/refs/heads` directory that has the SHA-1 of the last commit for that branch.

> **NOTE**
> Creating a branch is nothing more than just writing 40 characters to a file.

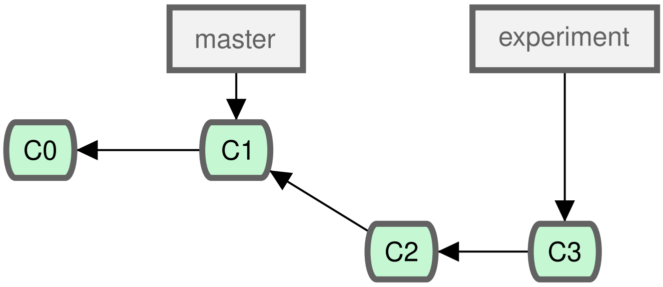
Switching to that branch simply means having Git make your working directory look like the tree that SHA-1 points to and updating the `HEAD` file so each commit from that point on moves that branch pointer forward (in other words, it changes the 40 characters in `.git/refs/heads/[current_branch_name]` to be the SHA-1 of your last commit).

Simple Case

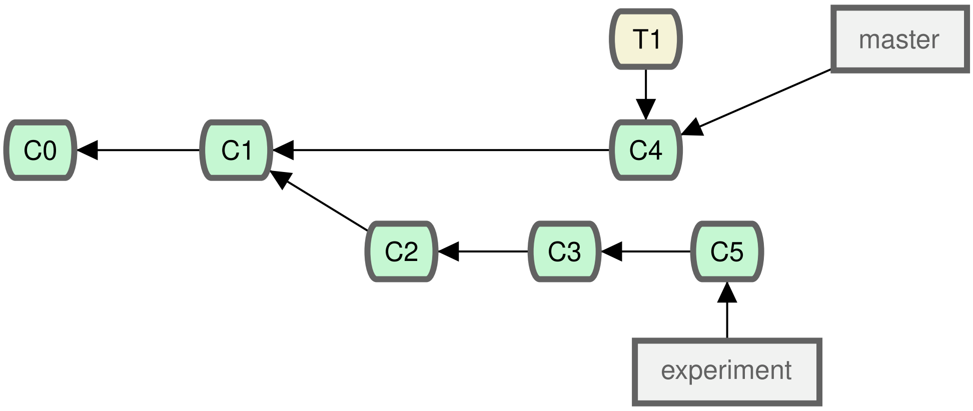


Now, let's see how Git handles branching, fetching and merging operations abstractly. For the following illustrations, we will represent the entire tree and the commit it points to as a single object.

Suppose that we work on a project for a while, then we get an idea for something that may not work out, but we want to do a quick proof-of-concept. We create a new branch called `experiment` off of our main branch, which is by convention called `master`. We then switch to the new branch and create a few commits.

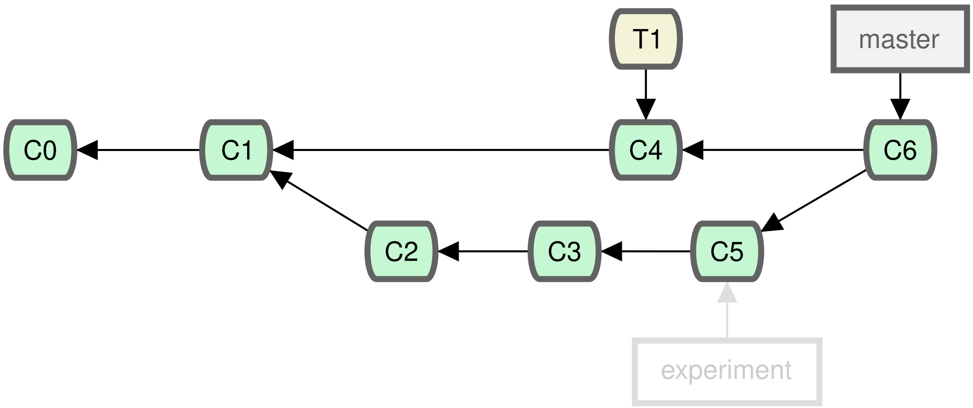


Then, our boss comes in and says we need a hot fix to production. So we switch back to our `master` branch, make the change, push the release and then tag the new commit with the release number. Then we go back to our `experiment` branch, continue working and commit again.



At this point, we show the new branch code to our co-workers and everyone likes the new changes. We decide we want to merge them back into our main branch, so we merge the changes and delete our `experiment` branch.

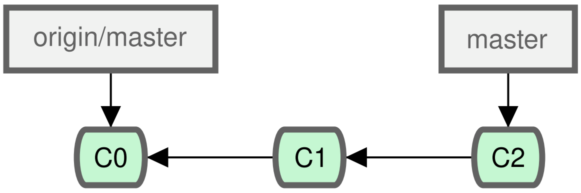
Our history of commit objects now looks like this:



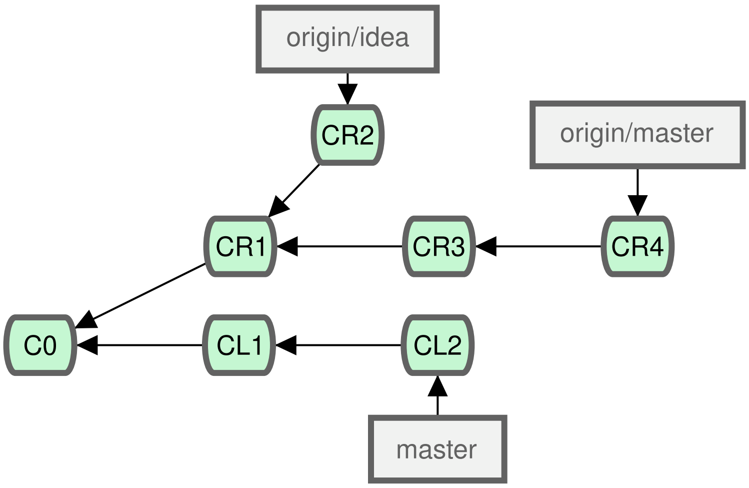
Remotes

Now lets take a look at remotes. Remotes are basically pointers to branches in other peoples copies of the same repository, often on other computers. If you got your repository by cloning it, rather than initializing it, you should have a remote branch of where you copied it from automatically added as `origin` by default. Which means the tree that was checked out during your initial clone would be referenced as `origin/master`, which means “the master branch of the origin remote.”

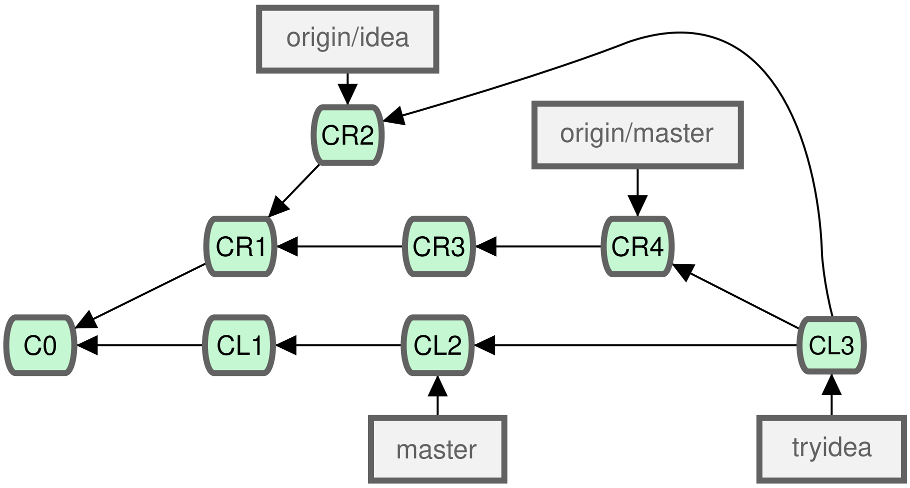
Lets say you clone someone’s repository and make a few changes. You would have two references, one to `origin/master` which points to where the master branch was on the person’s repository you cloned from when you did so, and a `master` branch that points to the most recent local commit.



Now let’s say you run a `fetch`. A fetch pulls all the refs and objects that you don’t already have from the remote repository you specify. By default, it is origin, but you can name your remotes anything, and you can have more than one. Suppose we fetch from the repository that we originally cloned from and they had been doing some work. They have now committed a few times on their master branch, but they also branched off at one point to try an idea, and they named the branch `idea` locally, then pushed that branch. We now have access to those changes as `origin/idea`.



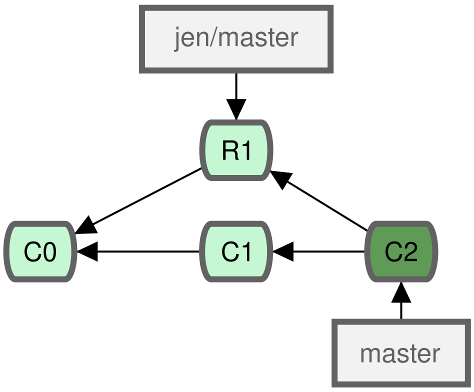
We look at the `idea` branch and like where they're going with it, but we also want the changes they've made on their `master` branch, so we do a 3-way merge of their two branches and our `master`. We don't know how well this is going to work, so we make a `tryidea` branch first and then do the merge there.



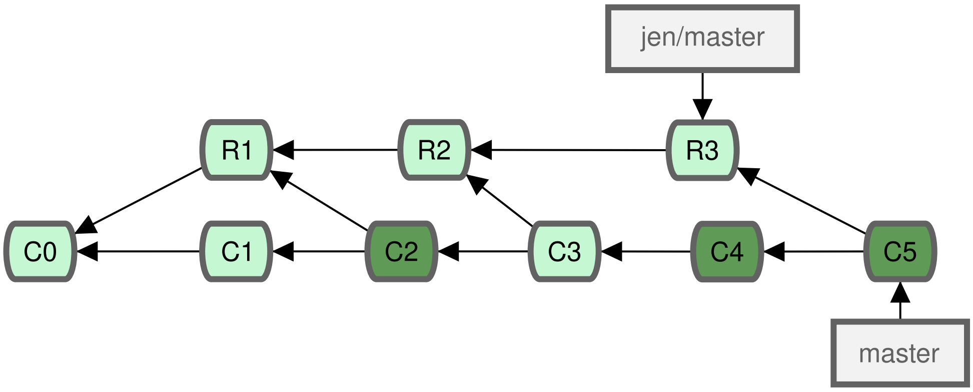
Now we can run our tests and merge everything back into our `master` branch if we want. Then we can tell our friend we cloned from to fetch from our repository, where we've merged their two branches for them and integrated some of our changes as well. They can choose to accept or reject that patch.

Rebasing

Let's say you and another developer, Jen, are working on the same project simultaneously. She clones from you, and works for a while and commits. You have committed in the meantime and want to get your work in sync, so you add her repository as the remote `jen`, do a fetch and merge her changes in, creating a new merge commit. (All commits that are simply merges are given a darker color in this example)



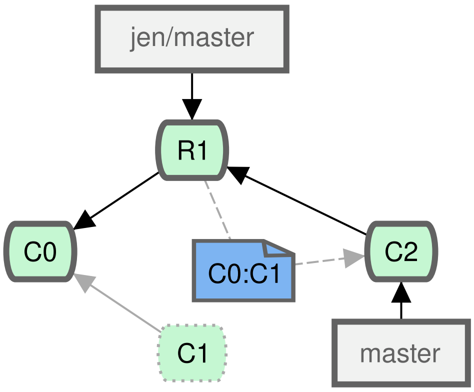
At this point, you both do work and commit changes and then you fetch and merge from her again. Then she does another commit and you fetch and merge once more. At this point, you'll have a commit history that looks something like this:



Perfectly fine, but it can get a little confusing when you litter the history with all those commits that do nothing but merge unshared changes. The longer you keep out of sync, the worse this can get.

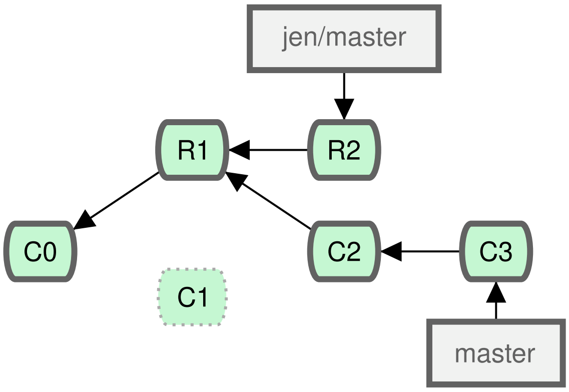
This is where the rebasing command comes in. With rebase, Git will checkout the *upstream branch*, in this case, Jen's `master` branch, and then replay all the changes you've done since you forked on top of those files, as if you had forked your work off at *that* point and done all your changes later, rather than earlier.

Rebase will literally produce a series of patch files of your work and start applying them to the upstream branch, automatically making new commits with the same messages as before and orphaning your older ones. These objects can then be removed, since nothing points to them, when you run the garbage collection tools (see "The Care and Feeding of Git" section).

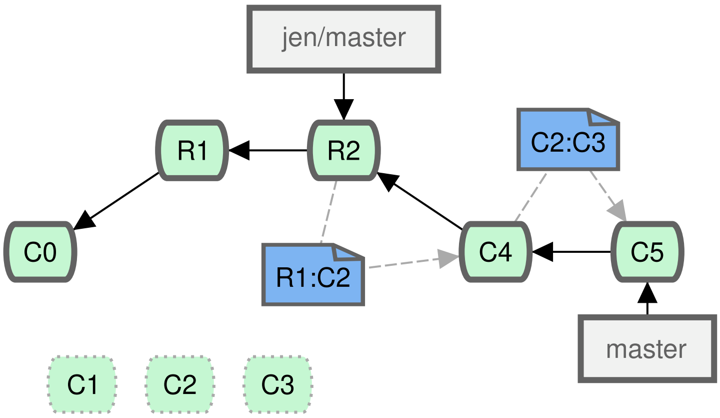


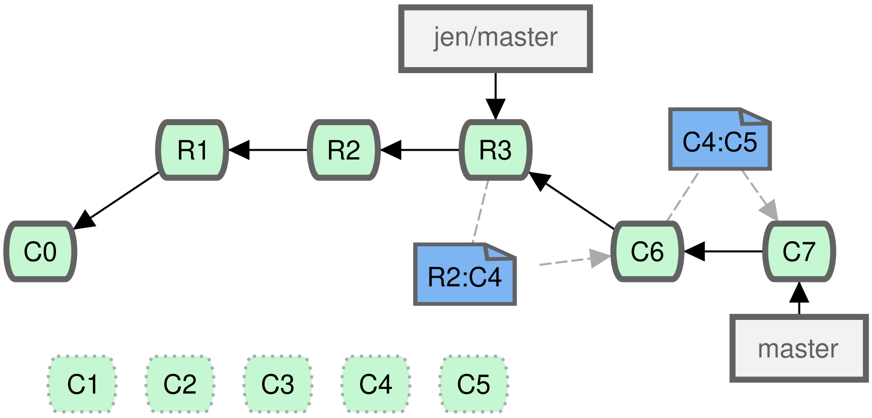
So let's see what happens if we rebase rather than merge in the same scenario. Here we have our first merge and we can see that it orphans *Commit 1* and applies the changes between *Commit 0* and *Commit 1* to the files in *Remote Commit 1*, creating a new *Commit 2*.

Then, as you'll remember, you and Jen both commit again. You'll notice that now it looks like she cloned you and committed and then you changed that code, rather than you both working at the same time and merging.



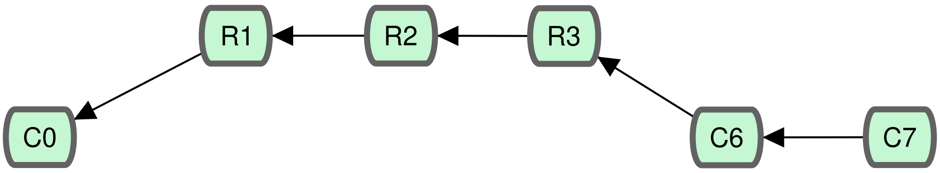
At this point, instead of merging two more times like we did originally, we rebase the next two commits she makes.



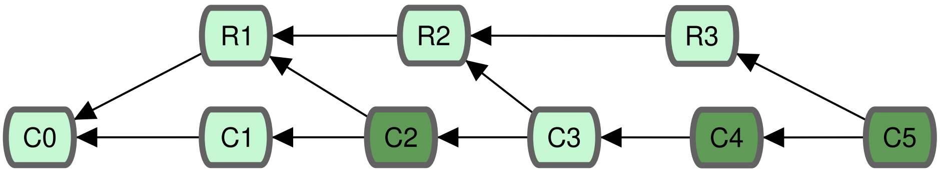


And finally, we are left with a commit history that looks like Figure 1, rather than Figure 2, which is what we would have if we had merged instead.

1



2



> **NOTE**
> You should remember to only do this on local branches before you push or on repositories that nobody has fetch access to - if anyone pulls down the objects that will become abandoned during a rebase, it gets a bit frustrating.

Use Cases

So why is this helpful, exactly? It means that you can keep your development cycles loosely coupled. Here is an example of a common workflow with cheap branches.

You have a `master` branch that is *always* stable - you never merge anything into it that you wouldn't put into production. Then you have a `development` branch that you merge any experimental code into before you imagine pulling it into the `master` branch.

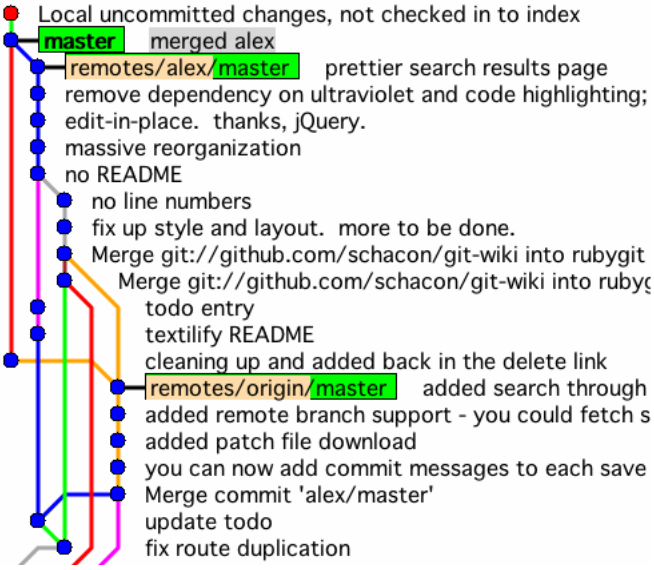
> **NOTE**
> It's a common error to think of the `master` branch as being equivalent to Subversion's

trunk. However, a custom `development` branch is much closer in practice to the Subversion `trunk`, where experimental work is done.

You create a new branch each time you begin to work on a story or feature, branching it off your current `development` branch each time, so if you get blocked and need to put it on hold, it doesn't affect anything else. When you do get back to them, you rebase them to the current `development` and it is just like you started from there. Often times you merge the branch back into `development` and delete it the same day that you created it.

If you get a huge project or idea - say refactoring the entire code base to the newest version of your framework or switching database vendors or something, you create a long-term branch, continuously rebase it to keep it in line with other development, and once everything is tested and ready, merge it in with your master.

Working with others is unbelievably easy. You ask in an IRC room if someone has implemented a feature in a library you are using. Turns out that someone has, and you are sent the URL of their public Git repo for that project. You add it as a remote, fetch it, create a new `merge-feature` branch off your `development` branch, merge in the new changes and you're done. There's no awkward emailing of patches - you can just add contributors as a remote and try out their branches before deciding to merge them in. If it breaks things or is not a good patch, you simply delete the `merge-feature` branch and that's it.



```
Local uncommitted changes, not checked in to index
master   merged alex
remotes/alex/master   prettier search results page
remove dependency on ultraviolet and code highlighting;
edit-in-place.  thanks, jQuery.
massive reorganization
no README
no line numbers
fix up style and layout.  more to be done.
Merge git://github.com/schacon/git-wiki into rubygit
Merge git://github.com/schacon/git-wiki into rubyg
todo entry
textilify README
cleaning up and added back in the delete link
remotes/origin/master   added search through
added remote branch support - you could fetch s
added patch file download
you can now add commit messages to each save
Merge commit 'alex/master'
update todo
fix route duplication
```

Figure 11: Project collaboration using multiple branch forks and merges

NOTE
A common problem with open source projects managed with Subversion is that a patch is sent which was made against and older version of the code base. With Git, it's as easy as applying the outdated patch in a new branch, then rebasing from `master` to bring it up to date with the current code base.

You branch and rebase or merge several times a day in and out of several different branches, some of which last for hours and some are continually there. Once you get used to this pattern, it completely changes the way you approach your development and the way you contribute and collaborate.

The Treeish

Besides branch heads, there are a number of shorthand ways to refer to particular objects in the Git data store. These are often referred to as a *treeish*. Any Git command that takes an object - be it a commit, tree or blob - as an argument can take one of these shorthand versions as well.

I will list here the most common, but please read the
rev-parse command
for full descriptions of all the available syntaxes.

Full SHA-1

```
dae86e1950b1277e545cee180551750029cfe735
```

You can always list out the entire SHA-1 value of the object to reference it. This is sometimes easy if you're copying and pasting values from a tree listing or some other command.

Partial SHA-1

```
dae86e
```

Just about anything you can reference with the full SHA-1 can be referenced fine with the first 6 or 7 characters. Even though the SHA-1 is always 40 characters long, it's very uncommon for more than the first few to actually be the same. Git is smart enough to figure out a partial SHA-1 as long as it's unique.

Branch or Tag Name

```
master
```

Anything in *.git/refs/heads* or *.git/refs/tags* can be used to refer to the commit it points to.

Date Spec

```
master@{yesterday}
master@{1 month ago}
```

This example would refer to the value of that branch yesterday. Importantly, this is the value of that branch *in your repository* yesterday. This value is relative to your repo - your 'master@{yesterday}' will likely be different than someone else's, even on the same project, whereas the SHA-1 values will *always* point to the same commit in every copy of the repository.

Ordinal Spec

```
master@{5}
```

This indicates the 5th prior value of the master branch. Like the *Date Spec*, this depends on special files in the *.git/log* directory that are written during commits, and is specific to *your* repository.

Carrot Parent

```
e65s46^2
master^2
```

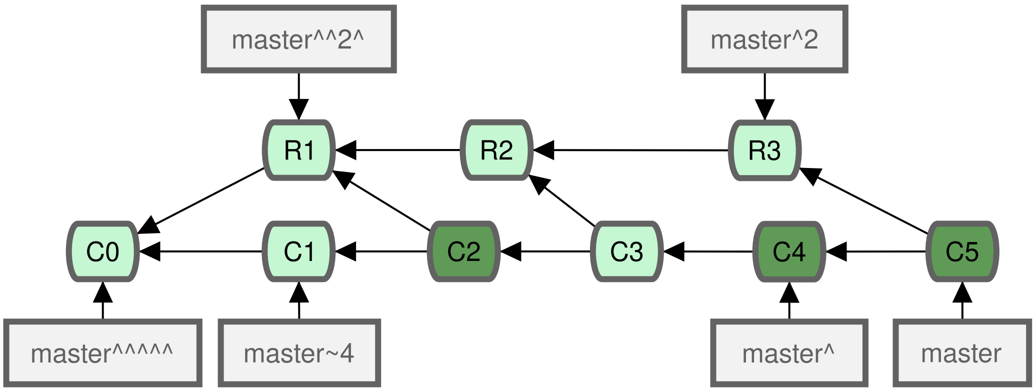
This refers to the Nth parent of that commit. This is only really helpful for commits that merged two or more commits - it is how you can refer to a commit other than the first parent.

Tilde Spec

```
e65s46~5
```

The tilde character, followed by a number, refers to the Nth generation grandparent of that commit. To clarify from the carrot, this is the equivalent commit in caret syntax:

```
e65s46^^^^^
```



Tree Pointer

`e65s46^{tree}`

This points to the tree of that commit. Any time you add a `^{tree}` to any commit-ish, it resolves to its tree.

Blob Spec

`master:/path/to/file`

This is very helpful for referring to a blob under a particular commit or tree.

The Git Directory

When you initialize a Git repository, either by cloning an existing one or creating a new one, the first thing Git does is create a Git directory. This is the directory that stores all the object data, tags, branches, hooks and more. Everything that Git permanently stores goes in this single directory. When you clone someone else's repository, it basically just copies the contents of this directory to your computer. Without a checkout (called a *working directory*) this is called a *bare* Git repo and moving it to another computer backs up your entire project history. It is the soul of Git.

When you run `git init` to initialize your repository, the Git directory is by default installed in the directory you are currently in as `.git`. This can be overridden with the *GIT_DIR* environment variable at any time. In fact, the Git directory does not need to be in your source tree at all. It's perfectly acceptable to keep all your Git directories in a central place (ex: `/opt/git/myproject.git`) and just make sure to set the `GIT_DIR` variable when you switch projects you are working on (ex: `/home/username/projects/myproject`).

The Git directory for our little project looks something like this:

```
.
|-- HEAD
|-- config
|-- hooks
|   |-- post-commit
|   `-- pre-commit
|-- index
|-- info
|   `-- exclude
|-- objects
|   |-- 05
|   |   `-- 76fac355dd17e39fd2671b010e36299f713b4d
|   |-- 0c
|   |   `-- 819c497e4eca8e08422e61adec781cc91d125d
|   |-- 1a
|   |   `-- 738da87a85f2b1c49c1421041cf41d1d90d434
|   |-- 47
|   |   `-- c6340d6459e05787f644c2447d2595f5d3a54b
|   |-- 99
|   |   `-- f1a6d12cb4b6f19c8655fca46c3ecf317074e0
|   |-- a0
|   |   `-- a60ae62dd2244a68d78151331067c5fb5d6b3e
|   |-- a1
|   |   `-- 1bef06a3f659402fe7563abf99ad00de2209e6
|   |-- a8
|   |   `-- 74b732e12a5c04b5a73d7f1123c249997b0b2d
|   |-- a9
|   |   `-- 06cb2a4a904a152e80877d4088654daad0c859
|   |-- e1
|   |   `-- b3ececb0cbaf2320ca3eebb8aa2beb1bb45c66
|   |-- fe
|   |   `-- 897108953cc224f417551031beacc396b11fb0
|   |-- info
|   `-- pack
`-- refs
    |-- heads
    |   `-- master
    `-- tags
        `-- v0.1
```

For more in-depth information on the Git directory layout, see the git repository layout docs..

For now, let's go over some of the more important contents of this directory.

.git/config

This is the main Git configuration file. It keeps your project specific Git options, such as your remotes, push configurations, tracking branches and more. Your configuration will be loaded first from this file, then from a `~/.gitconfig` file and then from an `/etc/gitconfig` file, if they exist.

Here is an example of what a config file might look like:

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[remote "origin"]
        url = git@github.com:username/myproject.git
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
        remote = origin
        merge = refs/heads/master
```

See the git-config docs for more information on available configuration options.

.git/index

This is the default location of the `index` file for your Git project. This location can be overridden with the *GIT_INDEX* environment variable, which is sometimes useful for temporary tree operations. See the chapter on the "Git Index" for more information on what this file is used for.

.git/objects/

This is the main directory that holds the data of your Git objects - that is, all the contents of the files you have ever checked in, plus your commit, tree and tag objects.

The files are stored by their SHA-1 values. The first two characters make up the sub-directory and the last 38 is the filename. For example, if the SHA-1 for a blob we've checked in was

```
a576fac355dd17e39fd2671b010e36299f713b4d
```

the file we would find the Zlib compressed contents in is:

```
[GIT_DIR]/objects/a5/76fac355dd17e39fd2671b010e36299f713b4d
```

.git/refs/

This directory normally has three subdirectories in it - *heads*, *remotes* and *tags*. Each of these directories will hold files that correspond to your local branches, remote branches and tags, respectively.

For example, if you create a `development` branch, the file `.git/refs/heads/development` will be created and will contain the SHA-1 of the commit that is the latest commit of that branch.

.git/HEAD

This file holds a reference to the branch you are currently on. This basically tells Git what to use as the parent of your next commit. The contents of it will generally look like this:

```
ref: refs/heads/master
```

.git/hooks

This directory contains shell scripts that are invoked after the Git commands they are named after. For example, after you run a commit, Git will try to execute the `post-commit` script, if it has executable permissions.

See the online hooks documentation for more information on what you can do with hooks.

Working Directory

The working directory is the checkout of the current branch you are working on. What is really important to note here is that this code is a working copy - it is not really important.

This is something that developers from most other SCMs have a hard time understanding and tends to scare them mightily. If you check out a different branch, Git will make your working directory look like that branch, removing any checked in content that is currently in your working directory that is not in the new tree. This is why Git will only let you checkout another branch if everything is checked in - there are no uncommitted modified files. The reason for this is that Git will remove files that are not necessary in the branch you are checking out - it needs to make sure that you can get them back again.

Most users don't like to see content automatically removed from their directories, but that's one of the mental shifts you'll need to make. Your working directory is temporary - everything is stored permanently in your Git repository. Your working directory is just a copy of a tree so you can edit it and commit changes.

The Index

Git has two places that content states are stored. The working directory stores one state at a time in a human-editable format. When committed, that state becomes permanent and repeatable by being stored in the object database. However, how do you determine what changes in your working directory will go into your next commit? Perhaps you have edited three files and you want two to go into the first commit (because they are related changes) and the other file into a second commit. This is where the index file comes in.

The index was called the cache for a while, because that's largely what it does. It is a staging area for changes that are made to files or trees that are not committed to your repository yet. It acts as sort of a middle ground between your working directory and your repository. When you run `git commit`, the resulting tree and commit object will be built based on the contents of the index.

> **NOTE**
> Early on, the index was called the *cache* or the *current directory cache.*

It is also used to speed up some operations. It keeps track of the state of all the files in your working directory, so you can quickly see what has been modified since the last commit.

Non-SCM Uses of Git

I keep saying that Git is primarily a content tracking toolkit with SCM tools built on top of it. So, if it's not built specifically to be an SCM, perhaps it would be useful to see some other examples of things it might be good for.

This is a simple listing of some other tools that have been built on Git internals to demonstrate that an SCM is the bundled application, but Git can also be a useful toolkit for any application needing to track and manage slowly changing distributed trees of content.

Peer to Peer Content Distribution Network

Imagine you are a retail chain or university campus and have a network of digital signage displays that play flash content advertisements or show campus news, etc. You have to get new content out to them every day or two, which may consist of any combination of XML files, images, animations, text and sound.

You need to build a content distribution framework that will easily and efficiently transfer all the necessary content to the machines on your network. You need to constantly determine what content each machine has and what it needs to have and transfer the difference as efficiently as possible, because networking to these units may be spotty.

It turns out that Git is an excellent solution to this problem. You can simply check all the needed content into Git, create a branch for each unit and point that branch to the exact subtree of content it needs. Then at some interval, you have the unit fetch its branch. If nothing has changed, nothing happens - if content has changed somehow, it gets only the files it does not already have in a delta compressed package and then expands them locally. Log and status files could even be transferred back by a push.

An example of a media company actually using this approach is Reactrix, which also happens to be where I work.

> **NOTE**
> Somewhat interestingly, Git being a good solution to this problem is what exposed me to the tool in the first place. We were using Git for content distribution on our network since the 1.0 release back in 2005, but using Perforce for version control internally. It wasn't until nearly a year later that we switched to actually using it to manage our source code.

Distributed Document Oriented Database

Using Git as a backend for a document oriented database could have some interesting applications. Git provides a number of features such as replication, searching with `grep`, and full versioning history for free.

Distributed Wiki

Suppose we wanted to have a wiki for documentation on a project. If we create a wiki that works on files, we can simply write those files into a Git repository and run a commit after every change. This gives us good performance, since it's just reading the content off the disk. We also get full file version history and easy 'recently changed' data. Searching is built in and we can edit the wiki offline.

The other cool feature we could use is the distributed nature of Git. We could add other people on the project as remote repositories and push to and fetch from them, merging changes to write a book or documentation collaboratively. We could branch to try out a re-write and then either merge it in or throw it away if we don't like it. We could send a pull request to our colleagues for them to try out the branch to preview it before we decide whether to merge it in or not.

It's possible the entire wiki project could even live in a bare branch (that is, a branch with no common ancestors with any existing branch) in the same repository as our project, so clones can get the documentation as well, without it muddying up our code tree.

See the git-wiki project for an example of this.

Distributed Issue Tracker

Another similar project might be a distributed ticketing system, where all the tickets (bugs and features) for a project could be stored in a Git repository, worked on offline and transferred with a project.

Examples of projects trying to do this are git-bug (which is the most advanced as of 2023), Ditz and my own TicGit.

Backup Tool

Let's suppose that you want to build something like a distributed Time Machine[TM] backup system (Apple all rights reserved) that efficiently packs up its backups and transfers them to multiple machines. I'm hoping by now that you could see the benefits of using the Git toolkit to accomplish this, but this particular problem is interesting because of something that Git doesn't do, which is permissions. Git stores the mode of its content in the tree, but it doesn't store any permissions data, which means it's not good for backing up directories in which permissions are important. A good example would be the `/etc` directory on a Unix machine.

One project that has tackled this is Gibak (TODO: This is old, maybe there is something more recent). It implements a metastore in OCaml and it's worth a look if this topic interests you.

> **NOTE**
> If you are interested in using Git in a non-standard way and plan to use Ruby, you might be interested in my git gem, which provides an object oriented interface in Ruby to the Git tools, including several of the lower level functions.

Using Git

Now that you *hopefully* understand what Git is designed to do at a fundamental level - how it tracks and stores content, how it stores branches and merges and tracks remote copies of the repository, let's see how to actually use it. This next section presents some of the basic commands that you will need to know in order to use Git effectively.

At the end of each chapter, there will be a link to the official Git documentation for each of the commands used in that section, in case you want to learn more or see all the options for that command.

Setting Up Your Profile

For every commit you do, Git will try to associate a name and email address. One of the first things you'll want to do in Git is to set these values. You can set them as global configuration values with the `git config` command:

```
git config --global user.name "Scott Chacon"
git config --global user.email "schacon@gmail.com"
```

This will create a new `~/.gitconfig` file that will look like this:

```
$ cat ~/.gitconfig
[user]
        name = Scott Chacon
        email = schacon@gmail.com
```

You can change those variables at any time, either by editing that file, or running the `git config` commands again.

If you want to set different values for a specific project, just leave out the `--global` and it will write the same snippet into your `.git/config` file in that repository, which will overwrite your global values.

- git config

Getting a Git Repository

There are two major ways you will get a Git repository - you will either clone an existing project, or you will initialize a new one.

New Repositories

To create a new Git repository somewhere, simply go to the directory you want to add version control to and type:

```
git init
```

This will create a `.git` directory in your current working directory that is entirely empty. If you have existing files you want to add to your new repository, type:

```
git add .
git commit -m 'my first commit'
```

This will add all of your current files into your new repository and index and then create your first commit object, pointing your new `master` branch to it. Congratulations, you have now added your source code to Git.

- git init
- git commit
- git add

Cloning a Repository

Many times you will be *cloning* a repository, however. This means that you are creating a complete copy of another repo, including all of its history and published branches.

> **NOTE**
> A clone is, for all intents and purposes, a full backup. If the server that you cloned from has a hard disk failure or something equally catastrophic, you can basically take any of the clones and stick it back up there when the server is restored without anyone really the worse for wear.

In order to do this, you simply need a URL that has a Git repository hosted there, which can be over HTTP, HTTPS, SSH or the special *git* protocol. We will use the public hosted repository of the simple library I mentioned at the beginning of the book.

```
git clone git://github.com/schacon/simplegit.git
```

This will by default create a new directory called `simplegit` and do an initial checkout of the `master` branch. If you want to put it in a different directory than the name of the project, you can specify that on the command line, too.

```
git clone git://github.com/schacon/simplegit.git my_directory
```

- git clone

Normal Workflow Examples

Now that we have our repository, let's go through some normal workflow examples of a single person developing.

Ignoring

First off, we will often want Git to automatically ignore certain files - often ones that are automatically generated during our development. For example, in Rails development we often want to ignore the log files, the production specific configuration files, etc. To do this, we can add patterns into the `.gitignore` file to tell Git that we don't want it to track them.

Here is an example `.gitignore` file.

```
tmp/*
log/*
config/database.yml
config/environments/production.rb
```

- .gitignore

Adding and Committing

Now we'll do some development and periodically commit our changes. We have a few options here - we can commit individual files or we can tell the *commit* command to automatically add all modified files in our working directory to the index, then commit it.

A good way to find out what you're about to commit (that is, what is in your index) is to use the `status` command.

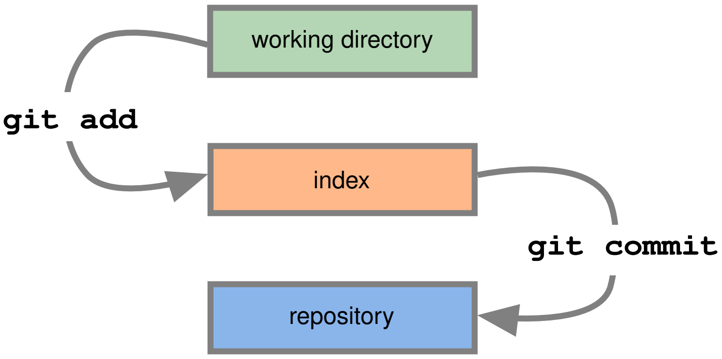
```
$ git status
# On branch master
# Changed but not updated:
```

```
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   README
#       modified:   Rakefile
#       modified:   lib/simplegit.rb
#
no changes added to commit (use "git add" and/or "git commit -a")
```

In this example, I can see that I've modified three files in my working directory, but none of them have been added to the index yet - they are not staged and ready to be committed. If I want to make these changes in two separate commits, or I have completed work on some of them and would like to push that out, I can specify which files to add individually and then commit.

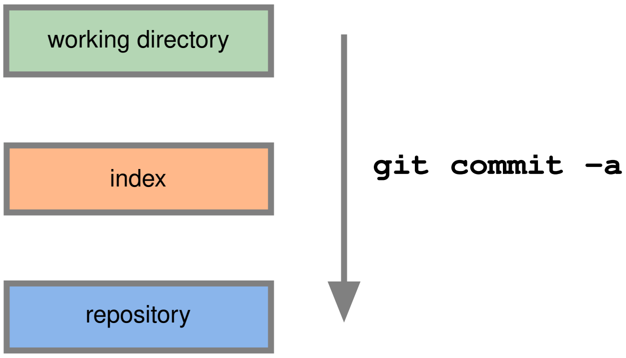
```
$ git add Rakefile
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   Rakefile
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   README
#       modified:   lib/simplegit.rb
#
```

You can see that if we commit at this point, only the Rakefile will show up as changed in the commit.



If we want to commit all our changes, we can use this shorthand, which will automatically run a `git add` on every modified file to our index, then commit the whole thing:

```
git commit -a -m 'committing all changes'
```



If you would like to give a more useful commit message, you can leave out the `-m` option. That will fire up your `$EDITOR` to add your commit message.

> **NOTE**
> Give special care to the first line of your commit message - it will often be the only

thing people see when they are looking through your commit history.

Now we can continue this loop - modifying, adding and committing - during our development.

- git status

Interactive Adding

Although that will work for all of your development needs - many developers simply use `-a` nearly every time they commit to just automatically add everything to the index - there is another way of adding files that makes for a more controlled and thematic set of commits. This is called *interactive* adding, and it is a very powerful tool to controlling what goes into each commit.

Let's say we add a new function to our `lib/simplegit.rb` file, add a new task to our `Rakefile` and then add a new `TODO` file to our project. Later we come back and want to commit, but we don't remember which files had to do with each other and we don't just want to commit them all together because that's confusing for collaborators trying to review our code. Interactive mode lets us modify our index interactively before committing. To fire it up, type `git add -i`:

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +5/-0 Rakefile
  2:    unchanged        +4/-0 lib/simplegit.rb

*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now>
```

We can see that we have two files that are being tracked (have been added at some point in the past) that have been modified. We cannot yet see our new TODO file, though. To add that, type `4` for the `add untracked` option and hit `Enter`.

```
What now> 4
  1: TODO
Add untracked>> 1
* 1: TODO
Add untracked>>
added one path

*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now>
```

You will see all the untracked files in your working directory. Type the numbers of the files you want to add, or a range (i.e.: `1-5`), and hit enter twice when you're done. This will drop you back to the main menu. You can then type `1` to see what your index looks like now.

```
What now> 1
           staged     unstaged path
  1:    unchanged        +5/-0 Rakefile
  2:        +5/-0      nothing TODO
  3:    unchanged        +4/-0 lib/simplegit.rb
```

You can see that the TODO file is now staged (in the index), but the other two are not. Let's add the Rakefile, but not the `lib/simplegit.rb` file and commit it. To do that, we hit `2`, which lists the files we can update, type `1` and enter to add the `Rakefile`, then hit enter again to go back to the main menu. Then we hit `7` to exit and run the `git commit` command

```
What now> 2
           staged     unstaged path
  1:    unchanged        +5/-0 Rakefile
  2:    unchanged        +4/-0 lib/simplegit.rb
Update>> 1
           staged     unstaged path
```

```
* 1:    unchanged        +5/-0 Rakefile
  2:    unchanged        +4/-0 lib/simplegit.rb
Update>>
updated one path

*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now> 7
Bye.

$ git commit -m 'rakefile and todo file added'
Created commit 4b0780c: rakefile and todo file added
 2 files changed, 9 insertions(+), 0 deletions(-)
 create mode 100644 TODO
```

The interactive shell is pretty simple and very powerful - playing with it instead of running `git add` commands directly may help in understanding what's happening, since you can see the status of your files in the index versus the working directory more clearly. It helps visualize that what is in your index (the `staged` column) is what will be committed when you run `git commit`.

You can also do more complicated things, like going through all of your change patches hunk by hunk, deciding if each hunk should be applied to the next commit or not. This means that if you've made a bunch of changes to one file, you can commit *part* of those changes in one commit, and the rest in a second. Try the `patch` menu option in the *Interactive Adding* menu to try this out.

> **NOTE**
> Beware of using interactive adding if you are already used to running `git commit -a`. If you run through the whole interactive add process and then run `git commit -a`, it will basically ignore everything you just did and just add all modified files.

Removing

For removing files from your tree, you can simply run:

```
git rm <filename>
```

This will remove that file from the index (and thus from the next commit) as well as from your working directory. On your next commit, the tree that commit points to will simply not contain that file anymore.

Log - the Commit History

So, now we have all this history in our Git repository. So what? What can we do with it? How can we see this history?

The answer is the very powerful `git log` command. The `log` command can show you nearly anything you want to know about your commit history. Also, since the entire history is stored locally, it's really fast compared with most other SCM systems, especially if your repository is packed (see the "Care and Feeding" section).

If you just run *git log*, you will get output like this:

```
$ git log
commit cf25cc3bfb0ece7dc3609b8dc0cc4a1e19ffbcd4
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Mar 17 21:52:20 2008 -0700

    committing all changes

commit 0c8a9ec46029a4e92a428cb98c9693f09f69a3ff
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700
```

```
    changed the verison number

commit 0576fac355dd17e39fd2671b010e36299f713b4d
Author: Scott Chacon <schacon@gmail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    my second commit, which is better than the first

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gmail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

This will show you the SHA-1 of each commit, the committer and date of the commit, and the full message, starting from the last commit on your current branch and going backward in reverse chronological order (so if there are multiple parents, it just squishes them together, interleaving the commits ordered by date)

Formatting Log Output

The default format takes up a lot of space though, so there are ways to limit and format this output differently. `--pretty` is a useful option for formatting the output in different ways.

For example, we can list the commit SHA-1s and the first line of the message with `--pretty=oneline`:

```
$ git log --pretty=oneline
cf25cc3bfb0ece7dc3609b8dc0c committing all changes
0c8a9ec46029a4e92a428cb98c9 changed the verison number
0576fac355dd17e39fd2671b010 my second commit, which is..
a11bef06a3f659402fe7563abf9 first commit
```

With `--pretty`, you can choose between *oneline*, *short*, *medium*, *full*, *fuller*, *email*, *raw* and *format:(string)*, where (string) is a format you specify with variables (ex: `--format:"%an added %h on %ar"` will give you a bunch of lines like "Scott Chacon added `f1cc9df` 4 days ago").

Filtering Log Output

There are also a number of options for filtering the log output. You can specify the maximum number of commits you want to see with `-n`, you can limit the range of dates you want to see commits for with `--since` and `--until`, you can filter it on the author or committer, text in the commit message and more. Here is an example showing at most 30 commits between yesterday and a month ago by me :

```
git log -n 30 --since="1 month ago" --until=yesterday --author="schacon"
```

- git log

Browsing Git

Git also gives you access to a number of lower level tools that can be used to browse the repository, inspect the status and contents of any of the objects, and are generally helpful for inspection and debugging.

Showing Objects

The `git show` command is really useful for presenting any of the objects in a very human readable format. Running this command on a file will simply output the contents of the file. Running it on a tree will just give you the filenames of the contents of that tree, but none of its subtrees. Where it's most useful is using it to look at commits.

Showing Commits

If you call it on a tree-ish that is a commit object, you will get simple information about the commit (the author, message, date, etc.) and a diff of what changed between that commit and its parents.

```
$ git show master^
commit 0c8a9ec46029a4e92a428cb98c9693f09f69a3ff
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the verison number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.platform  =   Gem::Platform::RUBY
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gmail.com"
     s.summary   =   "A simple gem for using Git in Ruby code."
```

Showing Trees

Instead of the `git show` command, it's generally more useful to use the lower level `git ls-tree` command to view trees, because it gives you the SHA-1s of all the blobs and trees that it points to.

```
$ git ls-tree master^{tree}
100644 blob 569b350811e7bfcb2cc781956641c3   README
100644 blob 8f94139338f9404f26296befa88755   Rakefile
040000 tree ae850bd698b2b5dfbac1ab5fd95a48   lib
```

You can also run this command recursively, so you can see all the subtrees as well. This is a great way to get the SHA-1 of any blob anywhere in the tree without having to walk it one node at a time.

```
$ git ls-tree -r -t master^{tree}
100644 blob 569b350811e7bfcb2cc781956641c    README
100644 blob 8f94139338f9404f26296befa8875    Rakefile
040000 tree ae850bd698b2b5dfbac1ab5fd95a4    lib
100644 blob 7e92ed361869246dc76f0cd0e526e    lib/simplegit.rb
```

The `-t` makes it also show the SHA-1s of the subtrees themselves, rather than just all the blobs.

Showing Blobs

Lastly, you may want to extract the contents of individual blobs. The `cat-file` command is an easy way to do that, and can also serve to let you know what type of object a SHA-1 is, if you don't know. It is sort of a catch-all command that you can use to inspect objects.

```
$ git cat-file -t ae850bd698b2b5dfbac
tree
$ git cat-file -p ae850bd698b2b5dfbac
100644 blob 7e92ed361869246dc76    simplegit.rb
$ git cat-file -t 569b350811
blob
$ git cat-file -p 569b350811
SimpleGit Ruby Library
======================

This library calls git commands and returns the output.
```

```
It is an example for the Git Peepcode book.

Author : Scott Chacon
```

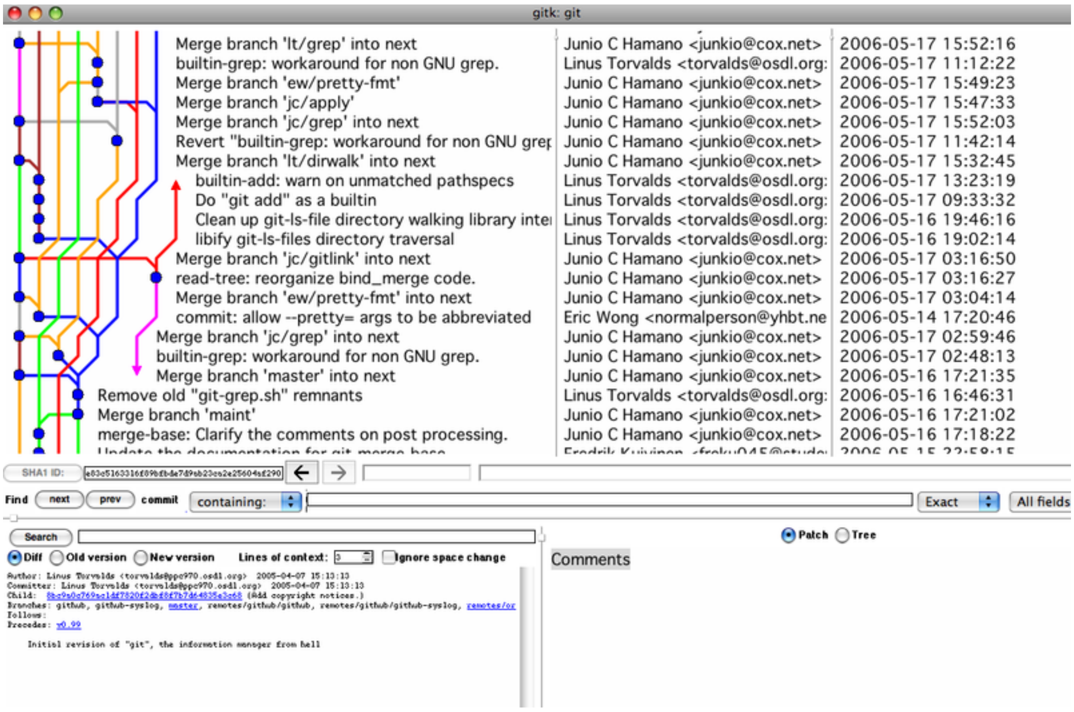
With those basic commands, you should be able to explore and inspect any object in any git repository relatively easily.

Graphical Interfaces

There are two major graphical interfaces that come with Git as tools to browse the repository.

Gitk

A very popular choice for browsing Git repositories is the Tcl/Tk based browser called `gitk`. If you want to see a simple visualization of your repository, `gitk` is a great tool.



Merge branch 'lt/grep' into next	Junio C Hamano <junkio@cox.net>	2006-05-17 15:52:16
builtin-grep: workaround for non GNU grep.	Linus Torvalds <torvalds@osdl.org:	2006-05-17 11:12:22
Merge branch 'ew/pretty-fmt'	Junio C Hamano <junkio@cox.net>	2006-05-17 15:49:23
Merge branch 'jc/apply'	Junio C Hamano <junkio@cox.net>	2006-05-17 15:47:33
Merge branch 'jc/grep' into next	Junio C Hamano <junkio@cox.net>	2006-05-17 15:52:03
Revert "builtin-grep: workaround for non GNU grep	Junio C Hamano <junkio@cox.net>	2006-05-17 11:42:14
Merge branch 'lt/dirwalk' into next	Junio C Hamano <junkio@cox.net>	2006-05-17 15:32:45
builtin-add: warn on unmatched pathspecs	Linus Torvalds <torvalds@osdl.org:	2006-05-17 13:23:19
Do "git add" as a builtin	Linus Torvalds <torvalds@osdl.org:	2006-05-17 09:33:32
Clean up git-ls-file directory walking library inte	Linus Torvalds <torvalds@osdl.org:	2006-05-16 19:46:16
libify git-ls-files directory traversal	Linus Torvalds <torvalds@osdl.org:	2006-05-16 19:02:14
Merge branch 'jc/gitlink' into next	Junio C Hamano <junkio@cox.net>	2006-05-17 03:16:50
read-tree: reorganize bind_merge code.	Junio C Hamano <junkio@cox.net>	2006-05-17 03:16:27
Merge branch 'ew/pretty-fmt' into next	Junio C Hamano <junkio@cox.net>	2006-05-17 03:04:14
commit: allow --pretty= args to be abbreviated	Eric Wong <normalperson@yhbt.ne	2006-05-14 17:20:46
Merge branch 'jc/grep' into next	Junio C Hamano <junkio@cox.net>	2006-05-17 02:59:46
builtin-grep: workaround for non GNU grep.	Junio C Hamano <junkio@cox.net>	2006-05-17 02:48:13
Merge branch 'master' into next	Junio C Hamano <junkio@cox.net>	2006-05-16 17:21:35
Remove old "git-grep.sh" remnants	Linus Torvalds <torvalds@osdl.org:	2006-05-16 16:46:31
Merge branch 'maint'	Junio C Hamano <junkio@cox.net>	2006-05-16 17:21:02
merge-base: Clarify the comments on post processing.	Junio C Hamano <junkio@cox.net>	2006-05-16 17:18:22

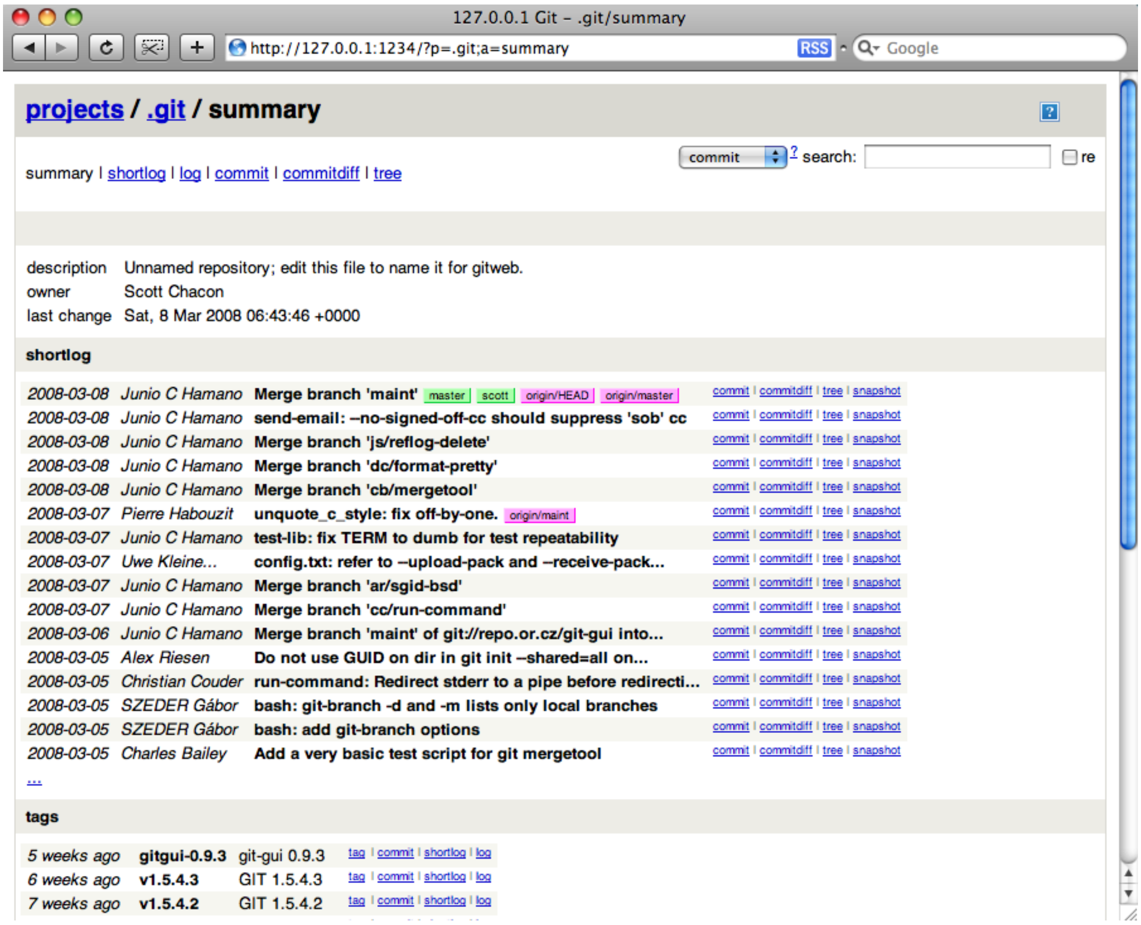
```
Author: Linus Torvalds <torvalds@ppc970.osdl.org>  2005-04-07 15:13:13
Committer: Linus Torvalds <torvalds@ppc970.osdl.org>  2005-04-07 15:13:13
Child:  8bc9a0c769ac1df7820f2dbf8f7b7d64835e3c68 (Add copyright notices.)
Branches: github, github-syslog, master, remotes/github/github, remotes/github/github-syslog, remotes/or
Follows:
Precedes: v0.99

    Initial revision of "git", the information manager from hell
```

Gitk will also take most of the same arguments that `git log` will take, including `--since`, `--until`, `--max-count`, revision ranges and path limiters. One of the most interesting visualizations that I regularly use is `gitk --all`, which will show all of your branches next to each other (rather than just the one you are currently on).

Instaweb

If you don't want to fire up Tk, you can also browse your repository quickly via the `git instaweb` command. This will basically fire up a web server running the gitweb CGI script using lighttpd, apache or webrick. It then tries to automatically fire up your default web browser and points it at the new server.

```
$ git instaweb --httpd=webrick
[2008-04-08 20:32:29] INFO  WEBrick 1.3.1
[2008-04-08 20:32:29] INFO  ruby 1.8.4 (2005-12-24) [i686-darwin8.8.2]
```



127.0.0.1 Git – .git/summary

http://127.0.0.1:1234/?p=.git;a=summary

projects / .git / summary

summary | shortlog | log | commit | commitdiff | tree

commit ? search: □ re

description Unnamed repository; edit this file to name it for gitweb.
owner Scott Chacon
last change Sat, 8 Mar 2008 06:43:46 +0000

shortlog

2008-03-08	*Junio C Hamano*	**Merge branch 'maint'** master scott origin/HEAD origin/master	commit | commitdiff | tree | snapshot
2008-03-08	*Junio C Hamano*	**send-email: –no-signed-off-cc should suppress 'sob' cc**	commit | commitdiff | tree | snapshot
2008-03-08	*Junio C Hamano*	**Merge branch 'js/reflog-delete'**	commit | commitdiff | tree | snapshot
2008-03-08	*Junio C Hamano*	**Merge branch 'dc/format-pretty'**	commit | commitdiff | tree | snapshot
2008-03-08	*Junio C Hamano*	**Merge branch 'cb/mergetool'**	commit | commitdiff | tree | snapshot
2008-03-07	*Pierre Habouzit*	**unquote_c_style: fix off-by-one.** origin/maint	commit | commitdiff | tree | snapshot
2008-03-07	*Junio C Hamano*	**test-lib: fix TERM to dumb for test repeatability**	commit | commitdiff | tree | snapshot
2008-03-07	*Uwe Kleine...*	**config.txt: refer to –upload-pack and –receive-pack...**	commit | commitdiff | tree | snapshot
2008-03-07	*Junio C Hamano*	**Merge branch 'ar/sgid-bsd'**	commit | commitdiff | tree | snapshot
2008-03-07	*Junio C Hamano*	**Merge branch 'cc/run-command'**	commit | commitdiff | tree | snapshot
2008-03-06	*Junio C Hamano*	**Merge branch 'maint' of git://repo.or.cz/git-gui into...**	commit | commitdiff | tree | snapshot
2008-03-05	*Alex Riesen*	**Do not use GUID on dir in git init –shared=all on...**	commit | commitdiff | tree | snapshot
2008-03-05	*Christian Couder*	**run-command: Redirect stderr to a pipe before redirecti...**	commit | commitdiff | tree | snapshot
2008-03-05	*SZEDER Gábor*	**bash: git-branch -d and -m lists only local branches**	commit | commitdiff | tree | snapshot
2008-03-05	*SZEDER Gábor*	**bash: add git-branch options**	commit | commitdiff | tree | snapshot
2008-03-05	*Charles Bailey*	**Add a very basic test script for git mergetool**	commit | commitdiff | tree | snapshot

...

tags

5 weeks ago	**gitgui-0.9.3**	git-gui 0.9.3	tag | commit | shortlog | log
6 weeks ago	**v1.5.4.3**	GIT 1.5.4.3	tag | commit | shortlog | log
7 weeks ago	**v1.5.4.2**	GIT 1.5.4.2	tag | commit | shortlog | log

When you are done, you can run the following to shut down the server:

```
git instaweb --stop
```

This is a quick way to throw up a web interface on your git repository for sharing with others or simply browsing it in a different way.

For a more long term web interface to your repository, you can put the gitweb Perl files that come with Git into your `cgi-bin` directory.

- git show
- git ls-tree
- git cat-file
- gitk
- git instaweb

Searching Git

Git has an easy way for searching through trees in your repository without having to check them out into your working directory to do it manually. It is called 'git-grep' and works very much like the traditional UNIX 'grep' command, with the difference that instead of listing the files you want to search as an argument, you list the trees you want to search.

For example, if we wanted to search for the string 'log_syslog' in versions 1.0 and 1.5.3.8 of the Git source code in the C files only, we can find that very easily.

```
$ git grep -n 'log_syslog' v1.5.3.8 v1.0.0 -- *.c
v1.5.3.8:daemon.c:16:static int log_syslog;
v1.5.3.8:daemon.c:92:   if (log_syslog) {
v1.5.3.8:daemon.c:768:        if (log_syslog)
v1.5.3.8:daemon.c:1055:         log_syslog = 1;
v1.5.3.8:daemon.c:1063:         log_syslog = 1;
v1.5.3.8:daemon.c:1112:         log_syslog = 1;
v1.5.3.8:daemon.c:1177: if (log_syslog) {
```

```
v1.0.0:daemon.c:13:static int log_syslog;
v1.0.0:daemon.c:45:     if (log_syslog) {
v1.0.0:daemon.c:423:          if (log_syslog)
v1.0.0:daemon.c:615:            log_syslog = 1;
v1.0.0:daemon.c:623:            log_syslog = 1;
v1.0.0:daemon.c:653:    if (log_syslog)

$ git grep -n -c 'log_syslog' v1.5.3.8 v1.0.0 -- *.c
v1.5.3.8:daemon.c:7
v1.0.0:daemon.c:6
```

You can see that you can view the number of lines that match, or the actual lines, and you can list as many trees (in this case, I used tags to reference them) as you want to search.

Another interesting example is to see which files in these versions do not contain the string 'git' anywhere in them:

```
$ git grep -L -v git v1.5.3.8 v1.0.0
v1.5.3.8:contrib/fast-import/git-p4.bat
v1.5.3.8:contrib/p4import/README
v1.5.3.8:t/t5100/patch0007
v1.5.3.8:t/t5100/patch0008
v1.5.3.8:templates/this--description
v1.0.0:debian/git-arch.files
v1.0.0:debian/git-cvs.files
v1.0.0:debian/git-doc.files
v1.0.0:debian/git-email.files
v1.0.0:debian/git-svn.files
v1.0.0:debian/git-tk.files
v1.0.0:templates/this--description
```

- git grep

Git Diff

Git has a great diff utility built in that can give you statistics or a patch file given any combination of tree objects, working directory and index.

Two common uses of this include seeing what you've worked on but not committed yet, and creating a patch file to send to someone over email (though there is a much preferred way to share changes which we will learn about in the "distributed workflow" section a bit later).

What has changed?

If you simply run `git diff` with no arguments, it will show you the differences between your current working directory and your index, that is, the last time you ran `git add` on your files.

For example, if I add my email to the README file and run it, I will see this:

```
$ git diff
diff --git a/README b/README
index 569b350..26c6ac8 100644
--- a/README
+++ b/README
@@ -5,4 +5,4 @@ This library calls git commands and returns the output.

 It is an example for the Git Peepcode book.

-Author : Scott Chacon
+Author : Scott Chacon (schacon@gmail.com)
```

You can also use `git diff` to show you some spiffy stats for a diff, rather than a patch file, if you want to see a wider overview of what changed, then drill down into specific files later. Here are some examples getting stats, the first for the differences between two commits and the second a summary between a commit and the current HEAD.

```
$ git diff --numstat a11bef06a3f65..cf25cc3bfb0
3       1       README
1       1       Rakefile
4       5       lib/simplegit.rb

$ git diff --stat 0576fac35..
 README           |    4 +++-
 Rakefile         |    2 +-
 lib/simplegit.rb |    4 ++++
 3 files changed, 8 insertions(+), 2 deletions(-)
```

If you want to see what the specific difference is in one of those files, you can just add a path limiter to the diff command.

```
$ git diff a11bef06a3f65..cf25cc3bfb0 -- Rakefile
diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.platform  =   Gem::Platform::RUBY
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gmail.com"
     s.summary   =   "A simple gem for using Git in Ruby code."
```

You can use this command to detect changes between your index and any tree, or your working directory and any tree, your working directory and your index, etc.

Generating Patch Files

The default output of the `git diff` command is a valid patch file. If you pipe the output into a file and email it to someone, they can apply it with the 'patch' command. If you've done some work off of a project in an 'experiment' branch, you could create a patch file this way:

```
git diff master..experiment > experiment.patch
```

You can then email that file to anyone, who could apply it with the '-p1' argument:

```
$ patch -p1 < ~/experiment.patch
patching file lib/simplegit.rb
```

- git diff

Branching

This is the fun part of Git that you'll come to love like a child. When you first initialize a git repository, or clone one, you'll get a 'master' branch by default if you don't specify something else. This is really just a git suggestion and you don't have to use it - like just about everything in Git, it can be overridden.

Switching Branches

However, let's say we're working on our project and we want to add a new function to our library, so we'll make a new branch called 'newfunc' and switch to it. There are two ways we can do this, one is to create the branch and then switch to it:

```
git branch newfunc; git checkout newfunc
```

The other way is to checkout a branch that doesn't exist yet and tell git you want to create it by passing the '-b' flag:

```
git checkout -b newfunc
```

Now, to check which branch we are on, we just type `git branch`:

```
$ git branch
  master
* newfunc
```

We can see we are now on our new branch. This means that if we modify a file and commit it, this branch will include that change, but the 'master' branch will not have it yet. So, we add a new method to our library and commit it.

```
$  vim lib/simplegit.rb; git commit -a -m 'added lstree function'
Created commit 1a8c32e: added lstree function
 1 files changed, 4 insertions(+), 0 deletions(-)
```

Now we want to change something in the README in the 'master' branch, but we haven't tested this function yet so we don't want to merge our new branch in yet. That's fine, we just switch back and make the change.

```
$ git checkout master

$  vim README         # add some text

$ git commit -a -m 'added more description'
Created commit d6fad7d: added more description
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Now lets see what the differences in our branches are.

```
$ git diff --stat master newfunc
 README           |    4 ++--
 lib/simplegit.rb |    4 ++++
 2 files changed, 6 insertions(+), 2 deletions(-)
```

We could also get a patch file for one to apply to the other, but what we really want to do next is merge the two.

- git branch
- git checkout

Simple Merging

So now we want to move the changes in our `newfunc` branch back into our `master` branch and remove it. This will require us merging one branch into another. Since we're already in our `master` branch, we'll merge in the `newfunc` branch like this:

```
git merge newfunc
```

Easy peasy. We can see that the `simplegit.rb` file now has four new lines and the `README` file was auto-merged.

Now we can get rid of our 'newfunc' branch with a simple:

```
$ git branch -d newfunc
Deleted branch newfunc.
```

Resolving Conflicts

That was a fairly simple problem, but what if we branch our code and then edit the same place in a file in different ways in each branch? In that case, we'll get a conflict when we try to merge them back together. Git is not too aggressive in trying to resolve conflicts, since you don't want it to make assumptions that are not necessarily correct, so bugs aren't introduced without your knowledge.

Let's say that we created a `versioning` branch and then modified the version in the `Rakefile` to different versions in both the new branch and the `master` branch, then tried to merge them together.

```
$ git merge versioning
Auto-merged Rakefile
CONFLICT (content): Merge conflict in Rakefile
Automatic merge failed; fix conflicts and then commit the result.
```

It tells us that there was a conflict and so the new commit object was not created. We will have to merge the conflicted file manually and then commit it again. The output tells us the files that had conflicts, in this case it was the `Rakefile`.

```
spec = Gem::Specification.new do |s|
    s.platform  =   Gem::Platform::RUBY
    s.name      =   "simplegit"
<<<<<<< HEAD:Rakefile
    s.version   =   "0.1.2"
=======
    s.version   =   "0.2.0"
>>>>>>> versioning:Rakefile
    s.author    =   "Scott Chacon"
    s.email     =   "schacon@gmail.com"
    s.summary   =   "A simple gem for using Git in Ruby code."
    s.files     =   FileList['lib/**/*'].to_a
    s.require_path  =   "lib"
end
```

We can see that in the `master` branch, the version was changed to `0.1.2` and in the `versioning` branch, the same line was changed to `0.2.0`. All we have to do is choose which one is correct and remove the rest of the lines, like so:

```
spec = Gem::Specification.new do |s|
    s.platform  =   Gem::Platform::RUBY
    s.name      =   "simplegit"
    s.version   =   "0.2.0"
    s.author    =   "Scott Chacon"
    s.email     =   "schacon@gmail.com"
    s.summary   =   "A simple gem for using Git in Ruby code."
    s.files     =   FileList['lib/**/*'].to_a
    s.require_path  =   "lib"
end
```

Now we add and commit that file, and we're good.

```
$ git add Rakefile
$ git commit -m 'fixed conflict'
Created commit 47c668a: fixed conflict
```

Undoing a Merge

Assume we have gone through some massive merge because someone on your team hasn't committed in a while, or you have a branch that was created some time ago but hasn't been rebasing and you want to pull it in. So you try to `git merge old_branch` it and you get conflict after conflict and it is just too much trouble to deal with and you just want to undo it all.

This is where `git reset` comes in. To reset your working directory and index back to what it was before you tried the merge, simply run:

```
git reset --hard HEAD
```

The `--hard` makes sure both your index file and working directory are changed to match what it used it be. By default it will only reset your index, leaving the partially merged files in your working directory. If you happen to have worked through it all and committed, then decided that it was a mistake because all of your tests break or something, you can still go back (and throw away that commit) by running:

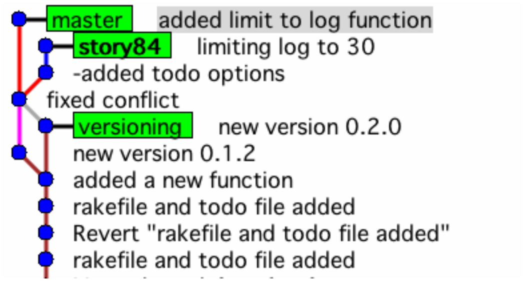
```
git reset --hard ORIG_HEAD
```

This is only helpful if you want to undo the latest change or changes. If you happen to commit again then decide that you want to keep the latest commit, but undo a commit that was added sometime before that, you'll need to use `git revert`, which is a bit too dangerous to cover here.

- git branch
- git merge
- git reset
- git revert

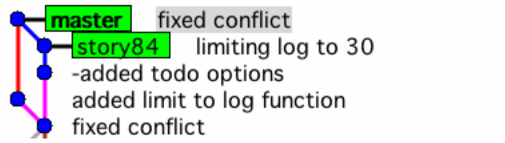
Rebasing

To review, rebasing is an alternative to merging that takes all the changes you've done since you branched off and applies those changes as patches to where the branch you are rebasing to is now, abandoning your original commit objects. For clean merges, this is a relatively simple process. Say we have been working in a branch called 'story84' and it's completed and we want to merge it into the master branch.



master added limit to log function
story84 limiting log to 30
-added todo options
fixed conflict
versioning new version 0.2.0
new version 0.1.2
added a new function
rakefile and todo file added
Revert "rakefile and todo file added"
rakefile and todo file added

If we do a simple merge, our history will look like this:



master fixed conflict
story84 limiting log to 30
-added todo options
added limit to log function
fixed conflict

But we don't want to mess up our history with a bunch of branches and merges when it can be clearer. Instead of running `git merge story84` from the master branch, we can stay in the 'story84' branch and run `git rebase master`

```
$ git rebase master
First, rewinding head to replay your work on top of it...
HEAD is now at 2c0d4d7... added limit to log function

Applying -added todo options

Adds trailing whitespace.
.dotest/patch:12:* add
warning: 1 line adds whitespace errors.
Wrote tree 2d0bd54dc9e4c398769cdcb59256ca03bb482ccb
Committed: b669c78acffaafd5ba34449e7faf88217394864a

Applying limiting log to 30

error: patch failed: lib/simplegit.rb:14
error: lib/simplegit.rb: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merged lib/simplegit.rb
CONFLICT (content): Merge conflict in lib/simplegit.rb
Failed to merge in the changes.
Patch failed at 0002.
```

```
When you have resolved this problem run "git rebase --continue".
If you would prefer to skip this patch, instead run "git rebase --skip".
To restore the original branch and stop rebasing run "git rebase --abort".
```

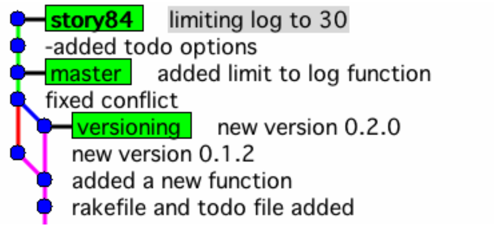
Many times this goes very smoothly and you can see all the new commits and trees written in place of the old ones. In this case, I had edited the 'lib/simplegit.rb' file differently in each branch, which caused a conflict. I will have to resolve this conflict before I can continue the rebase.

This gives us some options, since the rebase can do this at any point - say you have 8 commits to move onto the new branch - each one could cause a conflict and you will have to resolve them each manually. The 'rebase' command will stop at each patch if it needs to and let you do this.

You have three things you can do here, you can either fix the file, run a `git add` on it and then run a `git rebase --continue`, which will move on to the next patch until it's done. Our second option is to run `git rebase --abort`, which will reset us to what our repo looked like before we tried the rebase. Or, we can run `git rebase --skip`, which will leave this one patch out, abandoning the change forever.

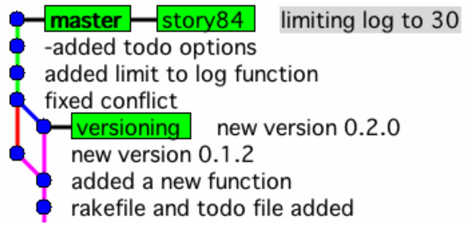
> **NOTE**
> Git rebase options for a conflict: `--continue` : tries to keep going once you've resolved it, `--abort` : gives up altogether and returns to the state before the rebase, `--skip` : skips this patch, abandoning it forever

In this case we will simply fix the conflict, run `git add` on the file and then run `git rebase --continue` which then makes our history look like this:



story84 limiting log to 30
-added todo options
master added limit to log function
fixed conflict
versioning new version 0.2.0
new version 0.1.2
added a new function
rakefile and todo file added

Then all we have to do is switch to the master branch and merge in 'story84' (which is called a 'fast-forward', since 'master' is now a direct ancestor of 'story84') to get this:

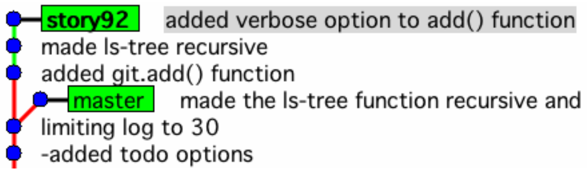


master story84 limiting log to 30
-added todo options
added limit to log function
fixed conflict
versioning new version 0.2.0
new version 0.1.2
added a new function
rakefile and todo file added

Interactive Rebasing

Much like Git provides a nicer way to work with your index before committing with `git add --interactive`, there is an interactive rebasing option that can only be fairly described as the "bee's knees".

Assume we have started working on a story to add the `git add` functionality to our library and so we've started a new branch called 'story92' and done the work there. Then we decide that the 'ls-tree' function needs to be recursive and make that change, then we tweak the library again, committing each time. Meanwhile, we've pulled in a change that implements the same 'ls-tree' change differently into our 'master' branch.



story92 added verbose option to add() function
made ls-tree recursive
added git.add() function
master made the ls-tree function recursive and
limiting log to 30
-added todo options

We can see before we try the merge that the same change is in each branch, and I can see that the master branch version is better, so I don't even really want to merge it, I just want to throw my change away. Also, I don't really need the other two commits to be two commits, because the second one is just a tweak and should be included in the first one. Lets use `git rebase -i` to rebase this branch and make those changes. When we run the command, our editor comes up, showing this:

```
# Rebasing c110d7f..c4f10f2 onto c110d7f
#
# Commands:
#  pick = use commit
#  edit = use commit, but stop for amending
#  squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
pick 2b6ae91 added git.add() function
pick bdfd292 made ls-tree recursive
pick c4f10f2 added verbose option to add()
```

Now we can see all of the commits that we are going to rebase. If we remove the 'made ls-tree recursive' line, it effectively ditches that commit so we'll avoid a conflict and not have to worry about it. Changing the action on the last line to 'squash', tells git to just make this and the previous commit into a single commit. So if we exit the editor with this as the new text:

```
pick 2b6ae91 added git.add() function
squash c4f10f2 added verbose option to add() function
```

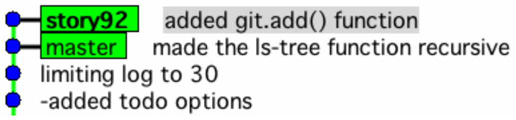
Then git sees we have squashed two commits and wants us to pick a commit message for it, giving us the commit messages of both, for us to create a new one for.

```
# This is a combination of two commits.
# The first commit's message is:

added git.add() function

# This is the 2nd commit message:

added verbose option to add() function

# Please enter the commit message for your changes.
# (Comment lines starting with '#' will not be included)
# Not currently on any branch.
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   lib/simplegit.rb
#
```

So we stick with the first message, save and exit the editor.

```
$ git rebase -i master
Created commit 8341085: added git.add() function
 1 files changed, 4 insertions(+), 0 deletions(-)
Successfully rebased and updated refs/heads/story92.
```

Now we've rebased and instead of three commits on top of our master and having to reconcile a useless conflict, we've just added a single commit with no resolving necessary:



story92 added git.add() function
master made the ls-tree function recursive
limiting log to 30
-added todo options

The rebase command is one of the most useful and unique in the git workflow.

- git rebase
- git reset

Stashing

Stashing is a pretty simple concept that is incredibly useful and very easy to use. If you are working on your code and you need to switch to another branch for some reason and don't want to commit your current state because it is only partially completed, you can run `git stash`, which will basically take the changes from your last commit to the current state of your working directory and store it temporarily.

In the following example, I have a change to my 'lib/simplegit.rb' file, but it's not complete.

```
$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   lib/simplegit.rb
#
no changes added to commit (use "git add" and/or "git commit -a")

$ git stash
Saved working directory and index state "WIP on master: c110d7f... made the ls-tree function recur
(To restore them type "git stash apply")
HEAD is now at c110d7f made the ls-tree function recursive and list trees

$ git status
# On branch master
nothing to commit (working directory clean)
```

Now I can see that my working directory is clean, as if I had committed, but I did not. Now I can switch branches, work for a while somewhere else, then switch back. So where did that change go? How do I get it back? Well, I can see my stashes by running `git stash list`.

```
$ git stash list
stash@{0}: WIP on experiment: 89e6d12... trying git archive
stash@{1}: WIP on master: c110d7f... made the ls-tree function recursive
stash@{2}: WIP on master: c110d7f... made the ls-tree function recursive
```

I see I have two stashes on the 'master' branch, both saved off of working from the same commit, and I have one stashed change off the 'experiment' branch. However, I can't remember which stash was the one I want, so I can use `git stash show` to figure it out.

```
$ git stash show stash@{1}
 lib/simplegit.rb |    4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)

$ git stash show stash@{2}
 lib/simplegit.rb |    8 ++++++++
 1 files changed, 8 insertions(+), 0 deletions(-)
```

I can also use any normal git tools that will take a tree on it, for instance, `git diff`:

```
$ git diff stash@{1}
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index e939f77..b03bc9c 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -21,10 +21,6 @@ class SimpleGit
    command("git ls-tree -r -t #{treeish}")
  end

-  def commit(message = 'commit message')
-    command("git commit -m #{message}")
-  end
-
  private
```

```
    def command(git_cmd)
```

And finally, I can apply it:

```
$ git stash apply stash@{1}
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   lib/simplegit.rb
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Now we can see that our working directory is back to where it was, with one file in an unstaged state. Now I would have to `git add` and `git commit` it if I wanted to keep the change.

Normally it's not even this complicated. If you run `git stash apply` without the actual stash reference, it will just apply the last stash you saved on that branch. Normally I will just use `git stash` to save something, go work elsewhere, then come back and run `git stash apply` to get back to where I was.

- git stash

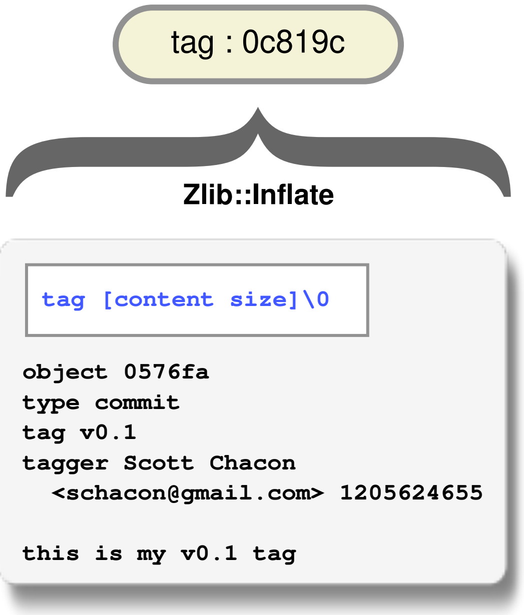
Tagging

As we previously covered, creating a tag in Git is much like making a branch. A tag in Git serves is basically a signed branch that never moves - it is simply an arbitrary string that points to a specific commit.

For example, if you wanted to tag your code base every time you released to production or created a new binary to release, you would run something like this:

```
git tag -a v0.1 -m 'this is my v0.1 tag'
```

As we recall from section one, that command will create a git object that looks something like this:



```
tag [content size]\0

object 0576fa
type commit
tag v0.1
tagger Scott Chacon
  <schacon@gmail.com> 1205624655

this is my v0.1 tag
```

and).

Lightweight Tags

You can also create a tag that doesn't actually add a Tag object to the database, but just creates a reference to it in the '.git/refs/tags' directory. If you run the following command:

```
git tag v0.1
```

Git will create the same file as before, '.git/refs/tags/v0.1', but it will contain the SHA-1 of the current HEAD commit itself, not the SHA-1 of a Tag object pointing to that commit. Unlike object Tags, these can be moved around easily, which is generally undesirable.

Signed Tags

At the other end of the tagging spectrum, you can sign Tag object with a GPG key to ensure its cryptographic integrity. Replacing '-a' with '-s' in the command will create a Tag object and sign it with the current users email address GPG key. If you want to specify a key, you can run it with '-u' instead:

```
git tag -u <key-id> v0.1 -m 'the 0.1 release'
```

Then, you or others can later verify that signed tag with a '-v'

```
git tag -v v0.1
```

- git tag

Exporting Git

If you want to create a release of your code, or provide some poor non-git user with a snapshot of just a specific tree, you can use the **git-archive** command.

> **NOTE**
> 'git-archive' used to be called 'git-tar-tree', in case you ever see that command around
> in older articles

You can create the archive in either 'tar' or 'zip' formats, the default being 'tar'. You can use the '—prefix' argument to determine what directory, if any, the files are expanded into. To create a gzipped tarball, you'll have to pipe the output through 'gzip' first.

```
git-archive --prefix=simplegit/ v0.1 | gzip > simple-git-0.1.tgz
```

Then, if you email that tarball to someone, they would get this when they opened it:

```
$ tar zxpvf simple-git-0.1.tgz
simplegit/README
simplegit/Rakefile
simplegit/TODO
simplegit/lib/
simplegit/lib/simplegit.rb
```

You can also archive parts of your project. This command will create a zip file of just the 'lib' directory of the first parent of your master branch that will expand out into the current directory:

```
git-archive --format=zip master^ lib/ > simple-git-lib.zip
```

Which will unzip like this:

```
$ unzip simple-git-lib.zip
Archive:  simple-git-lib.zip
ce9b0d5551762048735dd67917046b44176317e0
   creating: lib/
  inflating: lib/simplegit.rb
```

- git archive

The Care and Feeding of Git

Git requires a bit of tender loving care from time to time. It may seem a bit odd, but occasionally you should run a few commands on your repositories to make sure they're healthy and running as quickly as possible.

garbage collection

The `git gc` command is an important one to remember. It will pack up your objects into the delta-compressed format, saving you a lot of space and seriously speeding up several commands.

```
$ git gc
Generating pack...
Done counting 91 objects.
Deltifying 91 objects...
 100% (91/91) done
Writing 91 objects...
 100% (91/91) done
Total 91 (delta 33), reused 0 (delta 0)
Pack pack-9ca918b18abca509b2de71439522a62178415ebd created.
Removing unused objects 100%...
Done.
```

If can turn gc'ing automatically on and off by setting a configuration setting to '1' or '0':

```
git config --global gc.auto 1
```

This will make git automatically gc itself occasionally. You may want to setup a cron to do this at night, however, as it can take a while sometimes on really large repositories.

fsck and prune

If you want to check the health of your repository, you can run 'git-fsck', which will tell you if you have any unreachable or corrupted objects in your database and help you fix them.

```
$ git fsck
dangling tree 8276318347b8e971733ca5fab77c8f5018c75261
dangling blob 2302a5a4baec369fb631bb89cfe287cc002dc049
dangling blob cb54512d0a989dcfb2d78a7f3c8909f76ad2326a
dangling tree 8e1088e1cc1bc67e0ef01e018707dcb07a2a562b
dangling blob 5e069ed35afae29015b6622fe715c0aee10112ad
```

Which you can then remove with 'git-prune' (you can run it with '-n' first to see what it will do)

```
$ git prune -n
2302a5a4baec369fb631bb89cfe287cc002dc049 blob
5e069ed35afae29015b6622fe715c0aee10112ad blob
8276318347b8e971733ca5fab77c8f5018c75261 tree
8e1088e1cc1bc67e0ef01e018707dcb07a2a562b tree
cb54512d0a989dcfb2d78a7f3c8909f76ad2326a blob
$ git prune
$ git fsck
```

- git gc
- git fsck
- git prune

Distributed Workflow Examples

Now we've gone over most of the basic commands that you'll use on a day to day basis as a single developer. This chapter covers some examples of what you will use in order to collaborate with other developers on a code base.

Cloning

If you want to begin working on an existing project, you will need to get an initial version of it - copy its repository of objects and references to your machine. This is done with a clone. Git can clone a repository over several transports, including local, HTTP, HTTPS, SSH, its own `git` protocol, and `rsync`. The `git` protocol and SSH are preferred, because they are more efficient and not difficult to set up.

When you clone a repository, it in essence copies all the git objects to a new directory, checks you out a single local branch named the same as the HEAD branch on the cloned repo (normally 'master'), and stores all the other branches under a remote reference, by default named 'origin'.

That means that if we cloned the repo in the previous examples, instead of 'story84' being a local branch you can switch to, it becomes 'origin/story84' that you have to create a local branch to pull into, in order to work on (eg: `git checkout --track story84 origin/story84`)
The '—track' indicates that you may want to pull from or push to the origin of this branch later, so remember where it came from.

Local Clones

Local clones are the simplest types of clones - it is basically the equivalent of copying the .git directory and doing a checkout. The only major difference is that it adds all the original branches in as origin branches. Often you will do this when creating a bare repository (that is, a repository without a working directory) for putting on a public server, or if you're working with people using a shared repository over NFS or something similar.

```
git clone --bare simplegit/.git simplegit-bare.git
```

SSH and Git Transports

Cloning over SSH requires that you have user credentials on the machine you are cloning from. The git transport does not have this authentication and so is normally used for fetching only.

```
git clone git@github.com:schacon/ticgit.git ticgit_directory
```

HTTP and HTTPS Transports

Another popular way to clone a repository is over HTTP, just because it is so simple. You don't need to setup any special service or give out user credentials (made easier by services like gitosis), you simply scp your bare repository into any web server's static content directory. It is not as efficient as the other protocols - it will transfer loose objects and packfiles over a number of calls instead of packing them up, but it is simple.

```
git clone https://git.gitorious.org/piston/mainline.git piston
```

Once you have run one of these commands, you will have a copy of the git repository, full of all the history - basically every blob and tree and commit that project has ever had. This is really a full backup of the repository - if the main server ever goes down or gets corrupted, everyone who has ever cloned it has a fully capable backup that can replace it. With Git, there is really no single point of failure.

- git clone

Fetching and Pulling

So let's say that we're going to hack on TicGit, our git based ticket tracking system project. After we clone it, we look through the source code but don't do anything right away. After some time passes we come back to the project but it may not still be up to date - changes may have occurred in the meantime. So we fetch an update.

```
git fetch origin
```

This will contact the server over the same protocol we used to clone it and grab all of the objects and references that have been added since our clone and update our 'origin/[branch]' branches to point to what the server is pointing at now.

So, if we did create a tracking branch on 'story84' and it was changed on the server (someone pushed an update), before we fetch, our local 'story84' branch and our remote 'origin/story84' branch will be the same. After we fetch, they will be different. 'origin/story84' will now point to one of the new commit objects we downloaded during the fetch.

At this point, we may want to merge 'origin/story84' into our local 'story84' branch. That's easy enough, but if we want to do it automatically every time we fetch, we can use `git pull`, which is just a 'fetch' and then a 'merge' command.

So, these commands are functionally equivalent:

```
git pull origin/story84
```

```
git fetch origin/story84; git merge origin/story84
```

- git fetch
- git pull

Pushing

Now we can get updates from other repositories, but how can we push changes to them? If we have commit rights on the repository (normally over SSH), we can simply run `git push`.

```
git push origin master
```

The 'origin' in that case will be inferred if you leave it out, but if you've used a different name for your remote or you are trying to push one of your other branches, you can do that, too.

```
git push scott-public experimental
```

If you don't specify a branch, it will infer that you want to push every branch that you and the server have in common. So, if you have pushed your 'master' branch and your 'experimental' branch to the 'scott-public' server at any point, running this will update the server to have the newest versions of **both** of them:

```
git push scott-public
```

Whereas this will only update the 'master' branch:

```
git push scott-public master
```

> **NOTE**
> In Git, the opposite of 'push' is not 'pull', but 'fetch'. A 'pull' is a 'fetch' and then a 'merge'.

- git push

Multiple Remotes

Although a bit different in syntax maybe, most of that should seem familiar to any users of other SCM systems. However, this is where the 'decentralized' part comes in. In Git, there is really no special repository. You can add as many remote repositories that are related to your codebase in some way as you want. You can add each of your co-workers repositories as read-only repositories, you can have a centralized one you all share, one out on your production servers outside the firewall, a public one for stable or sanitized pushes on your personal webserver, one on your build server, etc, etc.

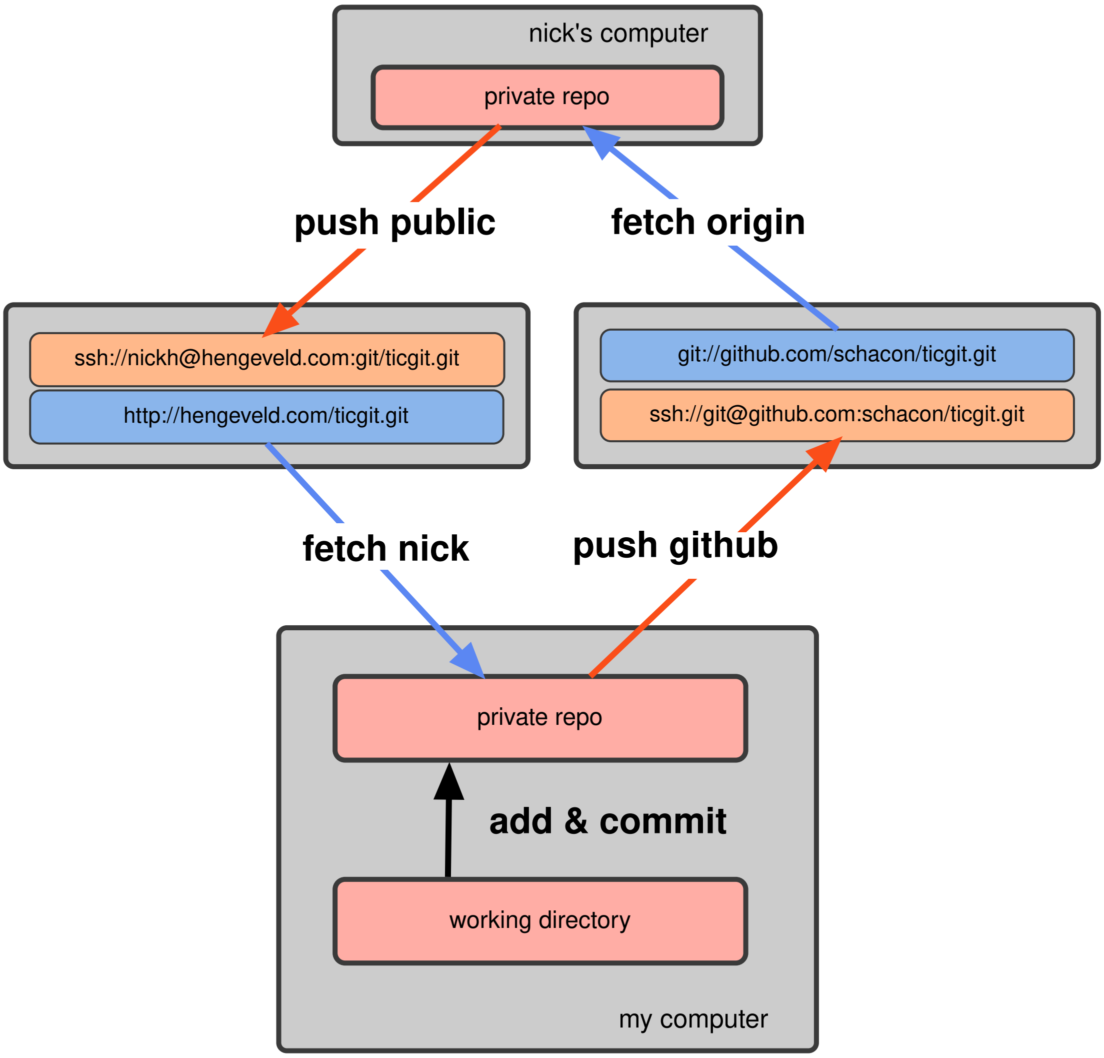
Pushing to and pulling from multiple sources is easy and straightforward. You simply add remotes :

```
git remote add mycap git@github.com:schacon/capistrano.git
git remote add official git://github.com/jamis/capistrano.git
```

Then, if the the project is updated, I can pull in the changes from one remote, merge them locally, and then push to another remote.

```
git fetch official
git merge official/master
git push mycap master
```

I can also add several remotes to pull and merge from, in this case, one for every developer with a public fork of that project that might push changes I care to try.


nick's computer
private repo
push public
fetch origin
ssh://nickh@hengeveld.com:git/ticgit.git
http://hengeveld.com/ticgit.git
git://github.com/schacon/ticgit.git
ssh://git@github.com:schacon/ticgit.git
fetch nick
push github
private repo
add & commit
working directory
my computer

You can also remove remotes at any time, which simply removes the lines that contain the URL in your `.git/config` file and the references to their remote branches in `.git/refs/[remote_name]` directory. It will not remove any of the git objects, so if you decide to add it again and fetch, very little will be transferred.

You can also view useful information about a remote branch by using the (**remote?**) show@ command. For example, if I run this on a checkout of the Git source code itself, I will see this:

```
$ git remote show origin
* remote origin
  URL: git://git.kernel.org/pub/scm/git/git.git
  Remote branch(es) merged with 'git pull' while on branch master
    master
  Stale tracking branches in remotes/origin (use 'git remote prune')
    old-next
  Tracked remote branches
    html maint man master next pu todo
```

- git remote

Possible Workflows

The idea of having multiple remote repositories that you can push to and/or pull from is probably new to you, and many people have a hard time figuring out what their workflow should look like, especially if they are moving from a centralized SCM system. I will present a couple of possible workflows that I have seen, so you can determine what will work best for you and your team.

Most of these are simply a matter of convention, not even configuration. Each of the models can pretty easily change to another with minimal configuration changes - maybe some permissions tweaked here or there.

Central Repository Model

There is a single repository that all developers push to and pull from.

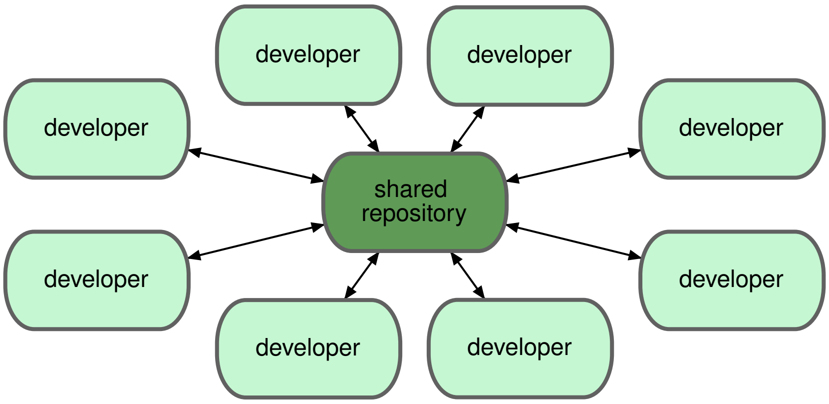


Figure 12: All developers push to a single server

This model works just like a centralized SCM and Git can work that way just fine. If you setup a repository for your team on a server that everyone has SSH or NFS access to, Git can very easily function as a centralized repository. This may be common on small teams with non-public projects where you don't want to worry about a hierarchy - the strength of this model is that it forces everyone to stay up to date with each other and it doesn't depend on a single role.

Even large teams could use this, but in general there are a lot of gains to be made in larger teams with a different or hybrid model.

Dictator and Lieutenant Model

This is a highly hierarchical model where one individual has commit rights to a blessed repository that everyone else fetches from. Changes are fetched from developers by lieutenants responsible for specific subsystems and merged and tested. Lieutenant branches are then fetched by the dictator and merged and pushed into the blessed repository, where the cycle starts over again.

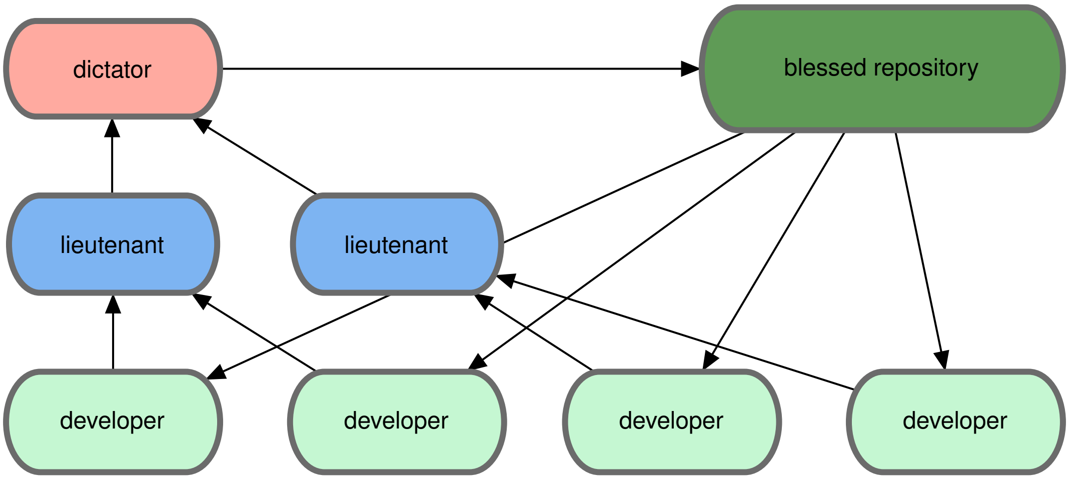


Figure 13: Approved features gradually make their way up the ladder

This is a model something like the Linux kernel uses, Linus being the benevolent dictator. This model is much better for large teams, and can be implemented with multiple and varied levels of lieutenants and sub-lieutenants in charge of various subsystems. At any stage in this process, patches or commits can be rejected - not merged in and sent up the chain.

Integration Manager Model

This is where each developer has a public repository, but one is considered the 'official' repository - it is used to create the packages and binaries. A person or core team has commit rights to it, but many other developers have public forks of that repository. When they have changes, they issue a pull request to an integration manager, who adds them as a remote if they haven't already - then merges, tests, accepts and pushes.

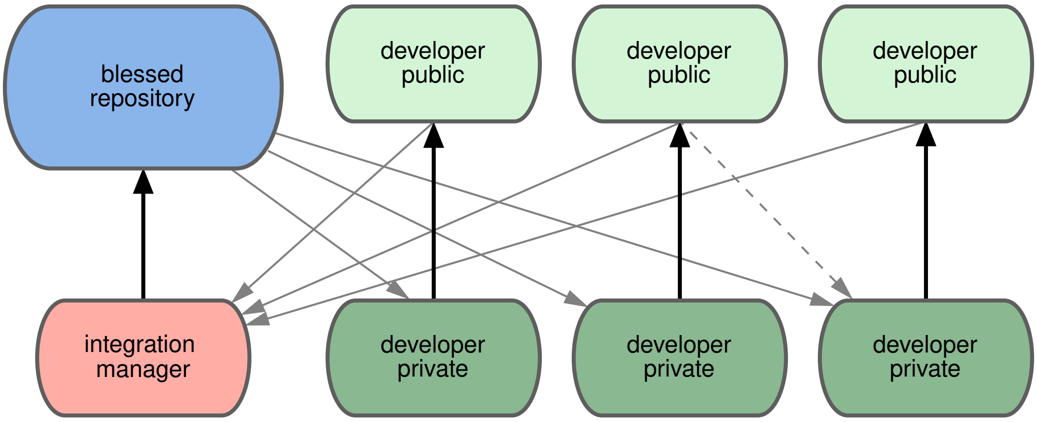


Figure 14: Private and public repositories driven by read-only pull requests

This is largely how community-based git repositories like GitHub were built to work and how many smaller open source projects operate.

In the end, there is really no single right way to do it - being a decentralized system, you can have a model with all of these aspects to it, or any combination you can think of. You can also have subgroups using different models on the same codebase. For example, if your company might have an internal fork of the Linux kernel that is managed by the Integration Manager model in addition to pulling in changes occasionally from Linuses branch. In the end, you and your team will have to think about what will work best for you.

Sharing Repositories

Over Git

Git provides its own special protocol, which is basically just a really thin wrapper over the 'git-upload-pack' command that will tell you what is available, then you tell it what you have and it gives you a packfile of the difference. To start it up manually, run something like the following command:

```
git-daemon --detach --export-all --base-path=/opt/git /opt/git/ambition
```

Though for long term running, you'll likely want to add this to your inet.d configuration. See the git-daemon docs for specific information on how to set that up.

The git protocol has no built in authentication, so generally you cannot push over it (although some people have open push policies and so allow that - I would not recommend setting that up, so you'll have to look up how to do that). If you want your users to have push access, it's recommended to use the SSH protocol. Many repositories, like GitHub, have SSH enabled for account owners to push over, and git-daemon enabled for the public to pull over - which is often the most efficient combination.

- git-daemon

Over SSH

Git can work entirely over SSH - it actually does much the same thing that happens over the git protocol, except it implicitly has authentication built in. If a user has SSH and write access to the repo and its subdirectories, then that user can push over SSH as well.

```
$ git clone --bare
scp -r project.bare user@host:/repos/project.git
git clone user@host:/repos/project.git
```

Git ships with the `git-shell`, a slightly more secure shell that restricts the user to performing only Git-related actions. If you use SSH public keys, you won't need to give out any passwords or create user accounts for other developers who need to have write-access to the repository. However, the restricted `git` user needs to have write access to the files and directories in the repository, which can be slightly awkward to setup and maintain.

Over HTTP

This is pretty easy to setup because there is no requirement other than a static web server - it is not like SVN which requires a DAV server to run. If you want to push over http, however, you will need DAV setup - it's generally a much better idea to use SSH to do so, however.

The only caveat is that you need to run `git update-server-info` each time you commit to the repository.

```
git update-server-info
```

It is generally a good idea to put this in the post-commit hook file on any server that you want to be able to fetch from over HTTP.

> **NOTE**
> Actually, you can run this without `update-server-info` if you never pack objects and you have a pre-defined list of branches you are always fetching. All `update-server-info` does is put a list of branches into one file (`.git/info/refs`) and a list of pack files into another (`.git/objects/info/pack`) to get around the fact that you cannot reliably list contents of a directory over HTTP - it's not built into the protocol. So for Git to know what packfiles and branches are available, it needs to have one URL it can get those from.

Hosted Repositories

If you don't want to deal with setting up and maintaining your own server for your git repository, you can use one of the growing number of public Git hosted servers.

I will focus on some interesting features of a commercial service called GitHub here, but there is also an open source project called CodeBerg - using the Gitea software - that has many of the same features.

> **NOTE**
> One of the compelling features of GitHub is the ability to create a private repository and share it with only a few developers. However, the complete source to Gitorious is available as a Ruby on Rails application. It could be modified to run on your company's server if your project needs to remain private.

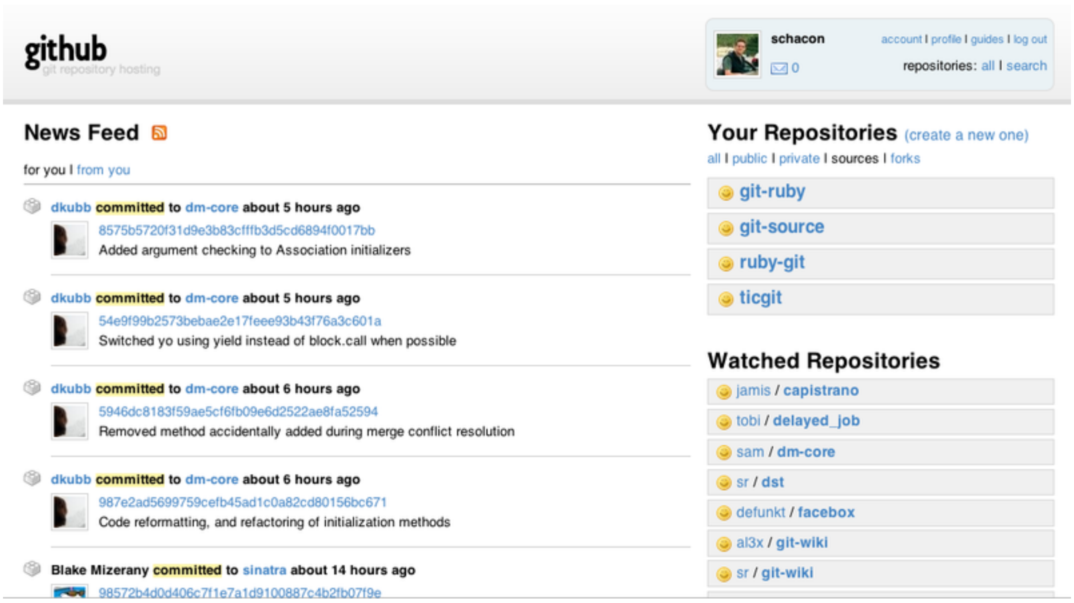
GitHub hosts many popular projects featured in the PeepCode series, including Ruby on Rails, Merb, RSpec, and Capistrano.

GitHub

GitHub is interesting as a source code hosting service because it includes some social networking features, which is not what most people imagine when they think of hosting source code.

You can follow friends, other developers, or just individual projects. You then subscribe to a single Atom feed and are kept up to date on what all those projects and people are doing, code-wise.

More interestingly, you can publicly fork a project to get your own copy of it. GitHub is unique in concept in that it is really centered around individuals rather than projects. For instance, if you want to follow or work on Merb, you would follow or fork wycats's Merb. There is really no *official* Merb page in GitHub. It's simply that wycats is known to be the blessed repository by the Merb project.



github
git repository hosting

schacon | 0

account | profile | guides | log out

repositories: all | search

News Feed

for you | from you

dkubb committed to dm-core about 5 hours ago
8575b5720f31d9e3b83cfffb3d5cd6894f0017bb
Added argument checking to Association initializers

dkubb committed to dm-core about 5 hours ago
54e9f99b2573bebae2e17feee93b43f76a3c601a
Switched yo using yield instead of block.call when possible

dkubb committed to dm-core about 6 hours ago
5946dc8183f59ae5cf6fb09e6d2522ae8fa52594
Removed method accidentally added during merge conflict resolution

dkubb committed to dm-core about 6 hours ago
987e2ad5699759cefb45ad1c0a82cd80156bc671
Code reformatting, and refactoring of initialization methods

Blake Mizerany committed to sinatra about 14 hours ago
98572b4d0d406c7f1e7a1d9100887c4b2fb07f9e

Your Repositories (create a new one)

all | public | private | sources | forks

- git-ruby
- git-source
- ruby-git
- ticgit

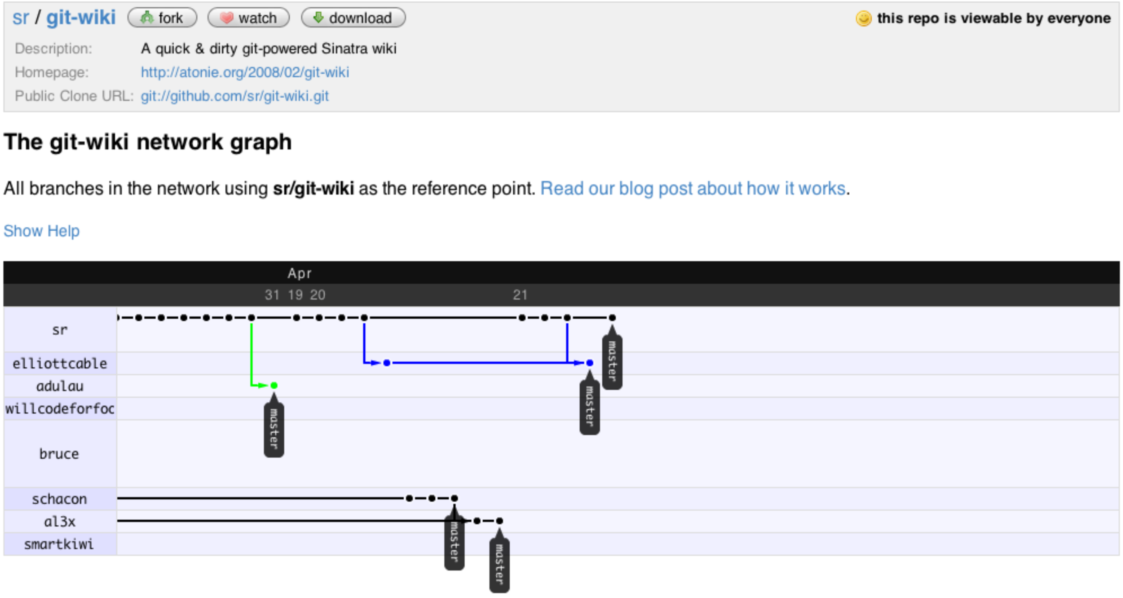
Watched Repositories

- jamis / capistrano
- tobi / delayed_job
- sam / dm-core
- sr / dst
- defunkt / facebox
- al3x / git-wiki
- sr / git-wiki

Figure 15: Custom GitHub home page with your projects

NOTE
In fact, Rails itself has at one point been forked and significantly enhanced by Ezra Zygmuntowicz. In other contexts this would be a bold divisive gesture, but with Git it's common practice.

You could just as easily follow and fork someone else's repository of the project. An interesting example of this is the git-wiki project, which was started by a user named `sr`, then was forked and greatly modified by `al3x`. `sr` wanted to keep the project simple, so now there are two major versions of the project. I have a checkout of the project and remotes added for each one, so I could work on and contribute to either.



sr / git-wiki   fork   watch   download   this repo is viewable by everyone

Description: A quick & dirty git-powered Sinatra wiki
Homepage: http://atonie.org/2008/02/git-wiki
Public Clone URL: git://github.com/sr/git-wiki.git

The git-wiki network graph

All branches in the network using **sr/git-wiki** as the reference point. Read our blog post about how it works.

Show Help

Figure 16: Project network visualization

Let's say there is a popular feature request that is either not completed or not accepted by the main project maintainer. You can very easily fork the project and keep your patch or patches up to date with the head on a regular basis and people can pull from yours instead. Perhaps enough demand builds up or enough testing is done from all these users that the patch is accepted. You can then delete your fork and revert to the original project head.

This will also really change patch submission for large projects. Instead of emailing patches around, you can fork the project, add your patch and submit a pull request for your branch with the fix to one of the core members or through the ticketing system. A member of the core team can add you as a remote easily, create a new testing branch, merge in or rebase your branch, test and accept or reject. If your patch is ignored or the team doesn't have time to deal with it yet, it's easy to keep up to date by continually rebasing and re-sending the pull requests until it's either rejected or accepted. It doesn't just go stale until it's difficult to apply the patch anymore.

Services like GitHub and an increased adoption of distributed SCM systems will dramatically change open source development workflows on teams of all sizes, in addition to changing the way individual developers work.

Commands Overview

This section is meant to be a really quick reference to the commands we have reviewed in Git and a quick description of what they do, where we have talked about them and where to find out more information on them.

Basic Git

git config

Sets configuration values for things like your user name, email, and gpg key, your preferred diff algorithm, file formats to use, proxies, remotes and tons of other stuff. For a full list, see the git-config docs

git init

Initializes a git repository - creates the initial '.git' directory in a new or existing project.

git clone

Copies a Git repository from another place and adds the original location as a remote you can fetch from again and possibly push to if you have permission.

git add

Adds changes in files in your working directory to your index, or staging area.

git rm

Removes files from your index and your working directory so they will stopped being tracked.

git commit

Takes all of the changes staged in the index (that have been `git added`), creates a new commit object pointing to it, and advances the branch to point to that new commit.

git status

Shows you the status of files in your index versus your working directory. It will list out files that are untracked (only in your working directory), modified (tracked but not yet updated in your index), and staged (added to your index and ready for committing).

git branch

Lists existing branches, including remote branches if '-a' is provided. Creates a new branch if a branch name is provided. Branches can also be created with '-b' option to `git checkout`.

git checkout

Checks out a different branch - makes your working directory look like the tree of the commit that branch points to and updates your HEAD to point to this branch now, so your next commit will modify it.

git merge

Merges one or more branches into your current branch and automatically creates a new commit if there are no conflicts.

git reset

Resets your index and working directory to the state of your last commit, in the event that something screwed up and you just want to go back.

git rebase

An alternative to merge that rewrites your commit history to move commits since you branched off, to apply to the current head instead. A bit dangerous as it discards existing commit objects.

git stash

Temporarily saves changes that you don't want to commit immediately for later. Can re-apply the saved changes at any time.

git tag

Tags a specific commit with a simple, human readable handle that never moves.

git fetch

Fetches all the objects that a remote version of your repository has that you do not yet have, so you can merge them into yours or simply inspect them.

git pull

Runs a `git fetch` then a `git merge`.

git push

Pushes all the objects that you have that a remote version does not yet have to that repository and advances its branches.

git remote

Lists all the remote versions of your repository, or can be used to add and delete them.

Inspecting Repositories

git log

Shows a listing of commits on a branch or involving a specific file and optionally details about what changed between it and its parents.

git show

Shows information about a git object, normally used to view commit information.

git ls-tree

Shows a tree object, including the mode and name of each node and the SHA-1 value of the blob or tree that it points to. Can also be run recursively to see all subtrees as well.

git cat-file

Used to view the type of an object if you only have the SHA-1 value, or used to redirect contents of files or view raw information about any object.

git grep

Lets you search through your trees of content for words and phrases without having to actually check them out.

git diff

Generates patch files or statistics of differences between paths or files in your git repository, or your index or your working directory.

gitk

Graphical Tcl/Tk based interface to a local Git repository.

git instaweb

Wrapper script to quickly run a web server with an interface into your repository and automatically directs a web browser to it.

Extra Tools

git archive

Creates a tar or zip file of the contents of a single tree from your repository. Easiest way to export a snapshot of content from your repository.

git gc

Garbage collector for your repository. Packs all your loose objects for space and speed efficiency and optionally removes unreachable objects as well. Should be run occasionally on each of your repos.

git fsck

Does an integrity check of the Git “filesystem”, identifying dangling pointers and corrupted objects.

git prune

Removes objects that are no longer pointed to by any object in any reachable branch.

git-daemon

Runs a simple, unauthenticated wrapper on the git-upload-pack program, used to provide efficient, anonymous and unencrypted fetch access to a Git repository.

References and Endnotes

Here are some references that I used or that you may use to find out more about Git.

The example git repository that I was working with throughout this book can be cloned from its GitHub repository.

For anything you cannot find in this book or these references, be sure to ask the fantastic people hanging out at the '#git' channel on `irc.freenode.net`.

Web Documentation

Main Git Documentation -
fantastic reference for all the command line programs

Git User's Manual

Git for Computer Scientists -
good detail about the DAG object model

A Tutorial Introduction to Git

Git Rebase Explained

A Tour of Git, the Basics

Junio Hamano New Git Maintainer -
some history on git and Junio becoming the new maintainer

Screencasts

The one pertaining to this book is sadly not available anymore. It could previously be found at `http://peepcode.com/products/git`.

RailsCasts Git Screencast

Using Git to Manage and Deploy Rails Apps

Revisions

All verions can be found unter the tags section on the source hosting site of this document.
As of April 2023, you will only find the sources there. Generated documents (PDF, HTML, EPub) are linked to in the README at the sources.

The original version from 6. September 2013 is available as a PDF from the original repo.