Git

by Scott Chacon

Git

OTHER PEEPCODE PRODUCTS

- RSpec (http://peepcode.com/products/rspec-basics) – A three part series on the popular behavior-driven development framework.

- Rails from Scratch (http://peepcode.com) – Learn Rails!

- RESTful Rails (http://peepcode.com/products/restful-rails) – Teaches the concepts of application design with REST.

- Subscription pack of 10 (http://peepcode.com/products/subscription-pack-of-10) – Save money! Buy 10 PeepCode credits.

- Javascript with Prototype (http://peepcode.com/products/javascript-with-prototypejs) – Code confidently with Javascript!

- Rails Code Review PDF (http://peepcode.com/products/draft-rails-code-review-pdf) – Common mistakes in Rails applications, and how to fix them.

CONTENTS

About this book

It took me a pretty long time to really get git. As we've continued to use git more and more in our company, we've found ourselves having 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 some 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 then is aimed at the developer who does not particularly like SVN 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 and graphically as possible in a clean, concise, easily readable volume. I would like you to be able to get through this book and understand git at a fundamental level, very quickly.

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

Installing Git

** TODO ** clean up this whole section, and actually run through each of these (yes, even windows!) ** TODO **

Before we can start playing with git, we'll have to install it. I'll quickly cover installing git on Linux, Mac and Windows.

You can download git from http://git.or.cz/

http://kernel.org/pub/software/scm/git/

Installing on Linux

wget http://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

or : apt-get git-core, yum install git-core

Installing on Mac

You are likely going to want to install git without the asciidoc depen-
dency because it is a pain to install. When you compile from source,
you will see

MAC 10.4 – TIGER

There are some requirements you'll have to install before you can
compile git.

http://sourceforge.net/projects/expat/

However, if you want an easier path, you can use the excellent Mac-
Ports software:

http://www.macports.org/

and just run

port install git-core

All the requirements are installed with the developer CD, so you can just download source and compile pretty easily if the developer tools are installed.

MacPorts is also an easy option if you have that installed.

http://code.google.com/p/git-osx-installer/

Windows

Cygwin http://www.cygwin.com/setup.exe

http://code.google.com/p/msysgit/

A TortoiseCVS lookalike for Git

http://repo.or.cz/w/git-cheetah.git/

A Short History of Git

Git 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 it's 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 all existing source code management systems, he decided to write his own.

7

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.

> "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 perl scripts. Recently (since 1.0), more and more of the perl scripts have been re-written in C, increasing portability and speed.

** TODO ** growth and adoption – list some projects (kernel, X.Org, Mesa3D, Compiz, Wine, OLPC, Fedora, Samba | ruby: rubinius, merb, god, github, rails?) ** TODO **

http://git.fedoraproject.org/git/

Section One – 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.

When I learned Git, as many people do, I learned it in the context of other SCMs I had used – Subversion or CVS. I really think 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 SVN, but...' context, but more like a really interesting file system.

Git tracks content – files and directories. It is at it's 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.

"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 (http://marc. info/?l=linux-kernel&m=111314792424707)

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 content and subdirectories. If you want a diff between two versions, it doesn't add up all the deltas, it simply looks at the two full directories 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 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 it's content using delta-compressed packfiles – but in concept, it is a very simple beast. Git is at it's 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.

NON-LINEAR DEVELOPMENT

branching and merging

DISTRIBUTED DEVELOPMENT

local copy of the entire development history, no central repository

Efficiency

seriously faster than nearly all other SCM systems for most operations.

** TODO ** table of speed benchmarks for git vs. svn vs. ??

– add, commit, push, diff, merge

– directory/checkout sizes? ** TODO **

A Toolkit Design

Following the Unix tradition, Git is a collection of many small tools written in C [1]

** TODO ** expand this whole chapter ** TODO **

Git Object Types

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 [see git directory]. Each object is compressed (with Zlib) and referenced by the SHA1 value of its contents plus a small header. In the examples, I will use the first 6 char-

acters of the SHA for simplicity, but the actual value is 40 characters long.

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:

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

The Blob

sidebar of git object sha algorithm (git-object-sidebar.textile)

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

It is important to note that it is the *contents* that are stored, not the files. The names and permissions 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.

The Tree

sidebar of tree object encoding algorithm (git-tree-sidebar.textile)

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

A tree is a list of trees and blobs that the tree contains, along with the names and permissions of those blobs and trees. The contents section of a tree object consists of a very simple text file that lists the *mode*, *type*, *name* and *sha* of each entry (see the sidebar for details).

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.

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

Notice how the *parent* in that commit is the same SHA1 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.

> the current record for number of commit parents in the Linux kernel is 12 – *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 a 'commit' and the object is the sha of the commit you're tagging. The tag can also be GPG signed, providing cryptographic integrity to a release or version.

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 section on [the git data model], where we'll pull all of this together into how all these objects relate to each other conceptually.

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 your current working directory 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 (/opt/git/myproject.git) and just make sure to set the GIT_DIR variable when you switch projects you are working on (/home/user-name/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. (http://www.kernel.org/pub/software/scm/git/docs/repository-layout.html)

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 [config section] 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 index] or [advanced index operations] sections for more information on the git index.

.git/objects/

This is the main directory that holds the data of your git objects and pack files – 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 SHA1 values. The first two characters make up the subdirectory and the last 38 is the filename. For example, if the SHA 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 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 [hooks section] or the online hooks documentation (http://www.kernel.org/pub/software/scm/git/docs/hooks.html) for more information on what you can do with hooks.

The Git Data Model

In computer science speak, the Git object data store is a Directed Acyclic Graph. That is, starting at any commit you can traverse it's 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 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. Then, as you continue to commit, one of the branches will keep changing to point to the new commit shas, while the other one can stay where it was.

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 store 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 3 trees, 3 blobs, 1 commit that points to the top of the tree, the current branch pointing to our last commit and the HEAD file pointing to the branch we're currently on to let 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 it's parent and the new tree and 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 re-use those references. Again, the branch reference will move forward and the new commit will point to it's 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 commit object.

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 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.

Branching and Merging

Here we come to one of the real strengths of Git, cheap 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 of the last commit for that branch.

.note Branching is just writing 40 characters to a file

Switching to that branch simply means making your working directory look like the tree that sha points to and updating the HEAD file so each commit from that point on moves that branch pointer forward (makes the 40 characters in '.git/refs/heads/[current_branch_name]' be the SHA of your last commit).

Merging is also simple, compared to most SCM systems – is just merging the trees that the commits you are trying to merge are pointing to, which is much simpler than resolving a bunch of deltas.

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.

Simple Case

Let's say 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, then 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 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 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.

Remotes

Now lets take a look at remotes. Remotes are basically other peoples branches of the same repository. 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 persons repository you cloned from when you did so, and a 'master' branch that points the most recent local commit.

Now lets 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. Lets say 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. 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 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 file, as if you had forked your work off at that point and done all your changes, 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. Git will automatically remove these eventually, since

nothing points to them, when you run the garbage collector (see git-gc).

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.

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.

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 effect anything else. When you do get back to them, you rebase them to the current 'development' and it's 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. No emailing patches around and applying them – 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.

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 (http://www.kernel.org/pub/software/scm/git/docs/git-rev-parse.html) for full descriptions of all the available syntaxes.

FULL SHA

```
dae86e1950b1277e545cee180551750029cfe735
```

You can always list out the entire SHA1 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

```
dae86e
```

Just about anything you can reference with the full SHA can be referenced fine with the first 6 or 7 characters. Even though the SHA

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 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. Git will go through the commit parents until it found one that matched that date.

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.

CARROT PARENT

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

This refers to the Nth parent of that commit. This is only really help-

ful 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 carrot 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 it's tree.

BLOB SPEC

```
master:/path/to/file
```

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

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 the Subversion world have a hard time understanding and tends to scare them mightily. If you check out a different branch, git will basically 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.

> the files in your working directory are temporary, they are not your repository!

Most long time Subversion 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

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 is also used to speed up some operations. It acts as sort of a middle ground between your working directory and your repository.

- keeping track of the state of working directory files

- staging changes

- generating commits

Non-SCM Uses of Git

I keep saying that Git is primarily a content tracking system with an 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.

(uses for slowly changing distributed trees)

Peer to Peer Content Distribution Network

Imagine you have a network of ...

http://gittorrent.utsl.gen.nz/rfc.html

Distributed Document Oriented Database

replication, search w/grep, no history loss

DISTRIBUTED WIKI

(offline writing) Branches, merges – distributed, adhoc documentation, book writing (git-wiki)

DISTRIBUTED ISSUE TRACKER

(offline working on tickets) (http://www.distract.wellquite.org/ Monotone) (http://bugseverywhere.org/be/show/HomePage)

Backup Tool

Let's say you want to build something like a distributed Time-Machine (Apple all rights reserved).

(http://eigenclass.org/hiki/gibak-backup-system-introduction)

Section Two – Using Git

Now that you understand what Git does 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 section, there will be a link to the full documenta-tion for each of the commands used in that section, in case you want to learn more or see all the options for that command.

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 to version control 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'
```

34

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 (http://www.kernel.org/pub/software/scm/git/docs/git-init.html)

- git commit (http://www.kernel.org/pub/software/scm/git/docs/git-commit.html)

- git add (http://www.kernel.org/pub/software/scm/git/docs/git-add.html)

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 it's history and published branches.

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, you can specify that on the command line, too.

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

- git clone (http://www.kernel.org/pub/software/scm/git/docs/git-clone.html)

Normal Workflow Examples

Now that we have our repository, let's go through some normal work-flow 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 (http://www.kernel.org/pub/software/scm/git/docs/gitignore.html)

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.

> 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 (http://www.kernel.org/pub/software/scm/git/docs/git-status.html)

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 let's 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 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 – 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'

'git rm'

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 will 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.

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 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 orders them by date)

Formatting Log Output

This all 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 shas and the first line of the message with '—pretty=oneline':

```
$> git log --pretty=oneline
cf25cc3bfb0ece7dc3609b8dc0cc4a1e19ffbcd4 committing all
changes
0c8a9ec46029a4e92a428cb98c9693f09f69a3ff changed the verison
number
0576fac355dd17e39fd2671b010e36299f713b4d my second commit,
which is better...
a11bef06a3f659402fe7563abf99ad00de2209e6 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.

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 the last 30 commits between yesterday and a month ago by me :

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

• git log (http://www.kernel.org/pub/software/scm/git/docs/git-log.html)

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 it's 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 it's 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 shas of all the blobs and trees that it points to.

```
$>git ls-tree master^{tree}
100644 blob 569b350811e7bfcb2cc781956641c37189e120d9
README
100644 blob 8f94139338f9404f26296befa88755fc2598c289
Rakefile
040000 tree ae850bd698b2b5dfbac1ab5fd95a48c4f4d5275b    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 of any blob any-where in the tree without having to walk it one node at a time.

```
$>git ls-tree -r -t master^{tree}
100644 blob 569b350811e7bfcb2cc781956641c37189e120d9
```

```
README
100644 blob 8f94139338f9404f26296befa88755fc2598c289
Rakefile
040000 tree ae850bd698b2b5dfbac1ab5fd95a48c4f4d5275b     lib
100644 blob 7e92ed361869246dc76f0cd0e526efd6a13bd87a     lib/
simplegit.rb
```

The '-t' makes it also show the shas 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 is, if you happen to not know.
It is sort of a catch-all command that you can use to inspect objects.

```
$>git cat-file -t ae850bd698b2b5dfbac1ab5fd95a48c4f4d5275b
tree
$>git cat-file -p ae850bd698b2b5dfbac1ab5fd95a48c4f4d5275b
100644 blob 7e92ed361869246dc76f0cd0e526efd6a13bd87a
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.

http://www.kernel.org/pub/software/scm/git/docs/git-instaweb.html

- git show (http://www.kernel.org/pub/software/scm/git/docs/git-show.html)

- git ls-tree (http://www.kernel.org/pub/software/scm/git/docs/git-ls-tree.html)

- git cat-file (http://www.kernel.org/pub/software/scm/git/docs/git-cat-fileß.html)

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 do that 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 my index, that is, the last time I ran 'git add' on my 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 com-mits 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 (http://www.kernel.org/pub/software/scm/git/docs/git-diff.html)

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 'new-func' 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 (http://www.kernel.org/pub/software/scm/git/docs/git-branch.html)

• git checkout (http://www.kernel.org/pub/software/scm/git/docs/git-checkout.html)

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 4 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 two different ways? 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.

```
rakefile.rb
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:

```
rakefile-post.rb
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 some-
one on your team hasn't commited 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 direc-
tory. 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 covered in [History
Manipulation]

- git branch (http://www.kernel.org/pub/software/scm/git/docs/git-branch.html)

- git merge (http://www.kernel.org/pub/software/scm/git/docs/git-merge.html)

- git reset (http://www.kernel.org/pub/software/scm/git/docs/git-reset.html)

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. 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.

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

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 just leave this one patch out.

.note Git rebase options for a conflict :—continue : trys to keep going once you've resolved it—abort : gives up altogether and returns to the state before the rebase—skip : skips this patch, but applies all the others

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:

Then all we have to do is switch to the master branch and merge 'story84' to get this:

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 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.

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 com-mits, because the second one is just a tweak and could be squashed into 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 neccesary:

The rebase command is one of the most useful and unique in the git workflow. To learn more about some spiffy things you can do with it, check out the [History Manipulation] and [Advanced Merging] sections.

• git rebase (http://www.kernel.org/pub/software/scm/git/docs/git-rebase.html)

• git reset (http://www.kernel.org/pub/software/scm/git/docs/git-reset.html)

Stashing

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:

and will store that in the '.git/objects/' directory and then will create a permanent reference to it in '.git/refs/tags/v0.1' that contains the sha of that tag.

Then you can use that as a reference to that commit at any time in commands like 'diff' or 'archive' (see next chapter).

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 of the current HEAD commit itself, not the sha 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 it's 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 (http://www.kernel.org/pub/software/scm/git/docs/git-tag.html)

Archiving 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 tar-ball, 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:

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

• git archive (http://www.kernel.org/pub/software/scm/git/docs/git-archive.html)

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 it's 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, it's 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 http://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 (http://www.kernel.org/pub/software/scm/git/docs/git-clone.html)

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 auto-matically 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 (http://www.kernel.org/pub/software/scm/git/docs/git-fetch.html)
- git pull (http://www.kernel.org/pub/software/scm/git/docs/git-pull.html)

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 (http://www.kernel.org/pub/software/scm/git/docs/git-push.html)

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.

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 (http://www.kernel.org/pub/software/scm/git/docs/git-remote.html)

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.

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.

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.

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 – say your company has an internal fork of the Linux kernel that is managed by the Integration Manager model, in addition to pulling in changes occasionally from Linus's branch. In the end, you (or you and your team) will have to sit down for a second and think about what will work best for you.

Sharing Repositories

Over Git

Git provides it's 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 (http://www.kernel.org/pub/software/scm/git/docs/git-daemon.html) 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 (http://www.kernel.org/pub/software/scm/git/docs/git-daemon.html)

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, so if the user they are ssh'ing in as has access to write to the repo, then that user can push over ssh as well.

```
$\> git clone --bare; scp
```

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.

.sidebar 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 that 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 (http://github.com) here, but there is also an open source project called Gitorious (http://gitorious.com) that has many of the same features.

My focus on GitHub is only because it has private repos for commercial projects, the open source ones are free, and many popular projects featured in the Peepcode series, including Ruby on Rails, Merb, RSpec, and Capistrano are hosted there.

GitHub

open source software repos are free

closed source or multiple committers cost

social networking – watching people and projects – forking a project
– sending a pull request

Section Five –
Commands Overview

This section is meant to be a really quick reference to the common commands 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.

The section is broken into two parts, the Porcelain and the Plumbing. The Porcelain commands are generally run by people on the command line, the Plumbing commands are generally used by scripts and programs to do complicated things at a lower level – use them if you want to write your own tool on top of git.

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 it's GitHub repository (http://github.com/schacon/simplegit)

Web Documentation

Main Git Documentation (http://www.kernel.org/pub/software/scm/git/docs) – fantastic reference for all the command line programs

Git User's Manual (http://www.kernel.org/pub/software/scm/git/docs/user-manual.html)

Git for Computer Scientists (http://eagain.net/articles/git-for-computer-scientists) – good detail about the DAG object model

Junio Hamano New Git Maintainer (http://kerneltrap.org/node/5496) – some history on git and Junio becoming the new maintainer

Git Rebase Explained (http://wincent.com/knowledge-base/Git_rebase_explained)

Screencasts

Git Peepcode Screencast (http://peepcode.com/products/git)

RailsCasts Git Screencast (http://railscasts.com/episodes/96)

Using Git to Manage and Deploy Rails Apps (http://www.jointheconversa-tion.org/railsgit)

IRC

#git channel on irc.freenode.net