Version

1.0

UCSD Extension

Information Technology and Software Engineering

Software Configuration Management with Git

© 2013 Raymond Mitchell III

Contents

[Recommended Books 1](#_Toc367234600)

[Getting Started with Git 3](#_Toc367234601)

[1.1 About Version Control 4](#_Toc367234602)

[1.2 A Short History of Git 10](#_Toc367234603)

[1.3 Git Basics 11](#_Toc367234604)

[Git Basics 17](#_Toc367234605)

[1.4 Getting a Git Repository 19](#_Toc367234606)

[1.5 Recording Changes to the Repository 21](#_Toc367234607)

[1.6 Viewing the Commit History 34](#_Toc367234608)

[1.7 Undoing Things 42](#_Toc367234609)

[1.8 Working with Remotes 46](#_Toc367234610)

[1.9 Tagging 52](#_Toc367234611)

[Git Branching 58](#_Toc367234612)

[1.10 Git’s Killer Feature – Branching 59](#_Toc367234613)

[1.11 What a Branch Is 60](#_Toc367234614)

[1.12 Basic Branching and Merging 68](#_Toc367234615)

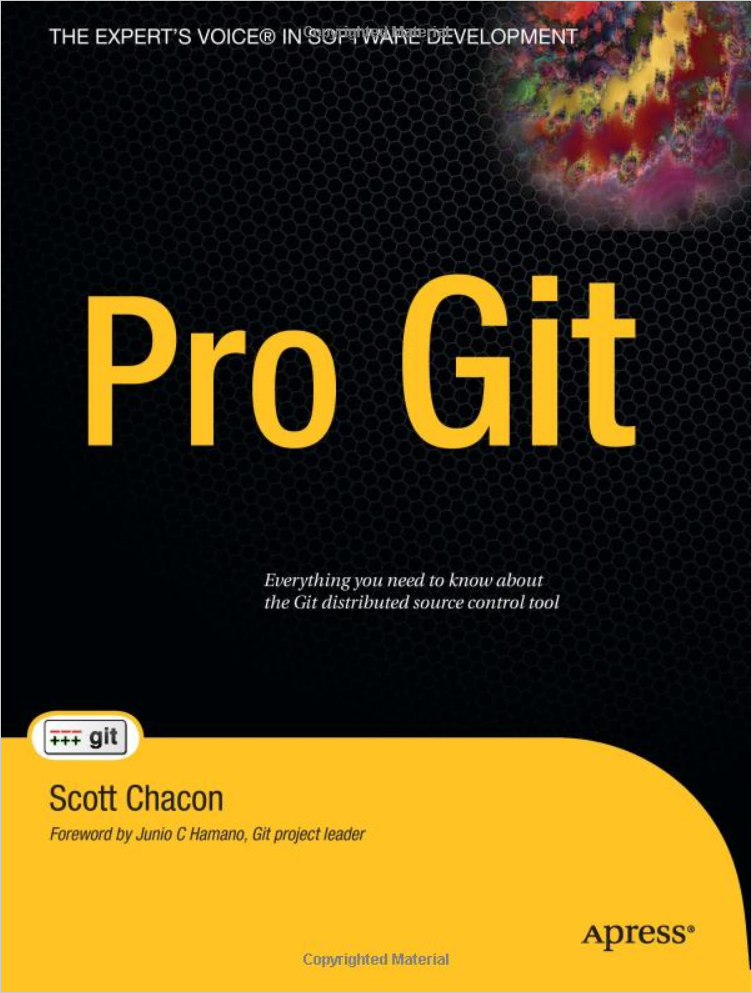
[1.13 Branch Management 79](#_Toc367234616)

[1.14 Branching Workflows 80](#_Toc367234617)

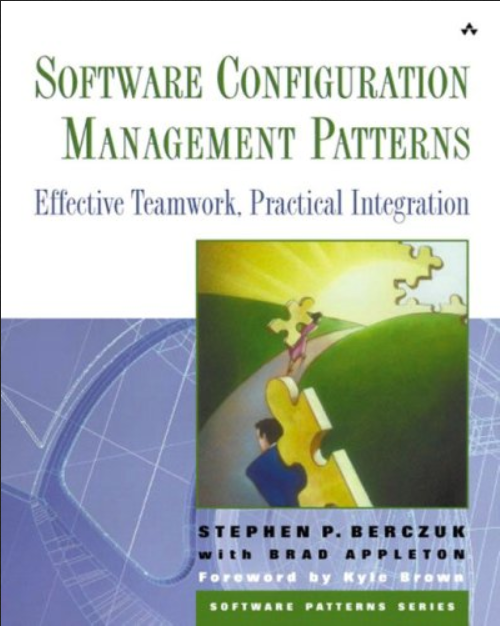
[1.15 Remote Branches 85](#_Toc367234618)

# Recommended Books

**Pro Git (Expert’s Voice in Software Development)**



**Software Configuration Management Patterns: Effective Teamwork, Practical Integration**

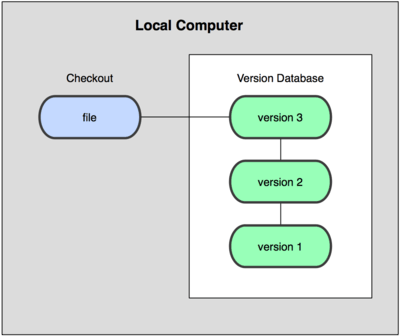
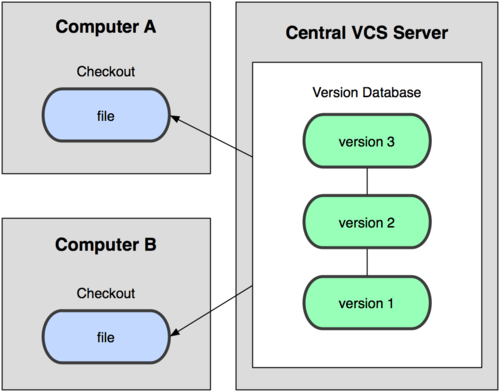


# Getting Started with Git

## About Version Control

* What is version control?
  + A version control system (VCS) is a system that records changes to a file or set of files over time so that you can recall specific versions later.
  + Any type of file on a computer can be placed under version control.
* A VCS allows you to:
  + Revert files back to a previous state
  + Revert the entire project back to a previous state
  + Review changes made over time
  + See who last modified something that might be causing a problem
  + See who introduced an issue and when
  + And more…
* Using a VCS also means that if you screw things up or lose files, you can generally recover easily.

**Types of Version Control Systems**

* *Manual VCS’s*
  + Many people’s version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they’re clever).
  + This approach is very common because it is so simple
  + But it is also incredibly error prone.
  + It is easy to forget which directory you’re in and accidentally write to the wrong file or copy over files you don’t mean to.
* *Local VCS’s*
  + Developed to deal with issues of manual source control.
  + Local VCSs are a simple database that keeps all the changes to files under revision control.
  + 
  + One of the more popular VCS tools was a system called **rcs**
    - Still distributed with many computers today.
    - Works by keeping patch sets (that is, the differences between files) from one revision to another in a special format on disk
    - Can then recreate what any file looked like at any point in time by adding up all the patches.
* *Centralized VCS’s*
  + The next major issue that people encounter is that they need to collaborate with developers on other systems.
  + To deal with this problem, Centralized Version Control Systems (CVCSs) were developed.
  + Have a single server that contains all the versioned files
    - A number of clients that check out files from that central place
  + 
  + Examples:
    - CVS
    - Subversion
    - Perforce
  + Offers many advantages, especially over local VCSs
    - Everyone knows to a certain degree what everyone else on the project is doing
    - Administrators have fine-grained control over who can do what
    - Far easier to administer a CVCS than it is to deal with local databases on every client.
  + Also has some serious downsides
    - Single point of failure that the centralized server represents
      * If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they’re working on
      * If the hard disk the central database is on becomes corrupted, and proper backups haven’t been kept, you lose absolutely everything
    - Local VCS systems suffer from this same problem
* *Distributed VCS’s*
  + Clients don’t just check out the latest snapshot of the files: they fully mirror the repository.
  + If any server dies any of the client repositories can be copied back up to the server to restore it.
  + Every checkout is really a full backup of all the data.
  + 
  + Examples:
    - Git
    - Mercurial
    - Bazaar
    - Darcs
  + These systems deal well with having several remote repositories they can work with
    - You can collaborate with different groups of people in different ways simultaneously within the same project
    - Allows you to set up several types of workflows that aren’t possible in centralized systems, such as hierarchical models.

## A Short History of Git

* Git began with a bit of creative destruction and fiery controversy
* The Linux kernel is an open source software project of fairly large scope
* For most of the lifetime of the Linux kernel maintenance (1991–2002), changes to the software were passed around as patches and archived files
* In 2002, the Linux kernel project began using a proprietary DVCS system called BitKeeper.
* In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down
  + Tool’s free-of-charge status was revoked
* Linux development community (and in particular Linus Torvalds, the creator of Linux) developed own tool based on some of the lessons they learned while using BitKeeper
* Goals of this new tool:
  + Speed
  + Simple design
  + Strong support for non-linear development (thousands of parallel branches)
  + Fully distributed
  + Able to handle large projects like the Linux kernel efficiently (speed and data size)
* Since its birth in 2005
  + Git has evolved and matured to be easy to use and yet retain these initial qualities
  + It’s incredibly fast
  + It’s very efficient with large projects
  + It has an incredible branching system for non-linear development

## Git Basics

**Snapshots, Not Differences**

* The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data.
* *Most other systems store information as a list of file-based changes*
  + These systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they keep as a set of files and the changes made to each file over time, as illustrated in this figure:
  + 
* *Git thinks of its data more like a set of snapshots of a mini filesystem*
  + Every time you commit Git takes a picture of what all your files look like at that moment and stores a reference to that snapshot
  + To be efficient, if files have not changed, Git doesn’t store the file again—just a link to the previous identical file it has already stored
  + Git thinks about its data more like this figure:
  + 
* This is an important distinction between Git and nearly all other VCSs
  + Git reconsiders almost every aspect of version control that most other systems copied from the previous generation
  + Git is more like a mini filesystem with some incredibly powerful tools built on top of it, rather than simply a VCS

**Nearly Every Operation is Local**

* Most operations in Git only need local files and resources to operate
  + Generally no information is needed from another computer on your network
  + If you’re used to a VCS where most operations have network latency overhead…
    - this aspect of Git will make you think that the gods of speed have blessed Git with unworldly powers
  + Possible because you have the entire history of the project right there on your local disk,
    - Most operations seem almost instantaneous
  + Examples:
    - To browse the history of the project, Git doesn’t need to go out to the server to get the history and display it for you,
      * it simply reads it directly from your local database
    - If you want to see the changes introduced between the current version of a file and the file a month ago,
      * Git can look up the file a month ago and do a local difference calculation
  + *There is very little you can’t do if you’re offline or off VPN*
    - If you get on an airplane or a train and want to do a little work,
      * you can commit happily until you get to a network connection to upload.
    - If you go home and can’t get your VPN client working properly,
      * you can still work.
    - In many other systems, doing so is either impossible or painful.
      * In Perforce you can’t do much when you aren’t connected to the server

**Git Has Integrity**

* Everything in Git is check-summed before it is stored and is then referred to by that checksum
  + This means it’s impossible to change the contents of any file or directory without Git knowing about it.
  + This is built into Git at the lowest levels and is integral to its philosophy.
  + You can’t lose information in transit or get file corruption without Git being able to detect it.
* Git uses an SHA-1 hash for checksumming.
  + 40 character string composed of hexadecimal characters
    - Calculated based on the contents of a file or directory structure in Git
  + Example:
    - 24b9da6552252987aa493b52f8696cd6d3b00373
* You will see these hash values all over the place in Git
  + Git stores everything not by file name but in the Git database addressable by the hash value of its contents.

**Git Generally Only Adds Data**

* When you do actions in Git, nearly all of them only add data to the Git database
  + It is very difficult to get the system to do anything that is not undoable or to make it erase data in any way
  + You can lose or mess up changes you haven’t committed yet; but…
    - After you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository

**The Three States**

* ***This is the main thing to remember about Git to make the rest of learning Git go smoothly***
* Git has three main states that your files can reside in:
  + *Committed*
    - the data is safely stored in your *local* database.
  + *Modified*
    - you have changed the file but have not committed it to your database yet
  + *Staged*
    - you have marked a modified file in its current version to go into your next commit snapshot.
* This leads to the three main sections of a Git project and the operations that move files between these sections:
* 
  + *Git Directory*
    - where Git stores the metadata and object database for your project
    - the most important part of Git
    - what is copied when you clone a repository from another compute
  + *Working Directory*
    - a single checkout of one version of the project
    - These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify
  + *Staging Area*
    - a simple file that stores information about what will go into your next commit
    - sometimes referred to as the index, but it’s becoming standard to refer to it as the staging area.

**Basic Git Workflow**

1. You modify files in your working directory
2. You stage the files, adding snapshots of them to your staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

**How Files Move through the 3 States**

* If a particular version of a file is in the git directory, it’s considered **committed**
* If it’s modified but has been added to the staging area, it is **staged**.
* If it was changed since it was checked out but has not been staged, it is **modified**

# Git Basics

## Getting a Git Repository

* Two ways to get a Git project:
  + Take an existing project or directory and import it into Git
  + Clone an existing Git repository from another server

**Initializing a Repository in an Existing Directory**

* To start to track an existing project in Git,
  + Go to project’s directory and type:

$ git init

* Creates a new subdirectory named .git that contains all of your necessary repository files
  + a Git repository skeleton
* At this point, nothing in your project is tracked yet
* If you want to start version-controlling existing files
  + you should probably begin tracking those files and do an initial commit
  + You can accomplish that as follows:

$ git add \*.c

$ git add README

$ git commit -m 'initial project version'

* We’ll go over what these commands do in just a minute

**Cloning an Existing Repository**

* If you want to get a copy of an existing Git repository — for example, a project you’d like to contribute to —
  + the command you need is git clone
  + notice that the command is clone and not checkout
  + This is an important distinction
  + Git receives a copy of nearly all data that the server has.
  + Every version of every file for the history of the project is pulled down when you run git clone
  + If your server disk gets corrupted, you can use any of the clones on any client to set the server back to the state it was in when it was cloned
  + You clone a repository with git clone [url]

$ git clone git://github.com/schacon/grit.git

* That command does the following:
  + creates a directory named grit
  + initializes a .git directory inside it
  + pulls down all the data for that repository
  + and checks out a working copy of the latest version
* If you want to clone the repository into a directory named something other than grit
  + you can specify that as the next command-line option:

$ git clone git://github.com/schacon/grit.git mygrit

* Does the same thing as the previous one, but the target directory is called mygrit
* Git has a number of different transfer protocols you can use:
  + git://
  + http(s)://
  + [user@server:/path.git](mailto:user@server:/path.git)
    - uses the SSH transfer protocol

## Recording Changes to the Repository

* You now have a working copy of the files for your project
* You need to make some changes and commit snapshots of those changes into your repository each time the project reaches a state you want to record
* Each file in your working directory can be in one of two states:
  + tracked
    - Files are files that were in the last snapshot; they can be:
      * unmodified
      * modified
      * staged
  + untracked
    - Everything else - any files in your working directory that were not in your last snapshot and are not in your staging area.
  + When you first clone a repository,
    - all of your files will be tracked and unmodified because you just checked them out and haven’t edited anything
* As you edit files, Git sees them as modified, because you’ve changed them since your last commit
* You stage these modified files and then commit all your staged changes, and the cycle repeats.
* This figure shows the lifecycle:
  + 

**Checking the Status of Your Files**

* The main tool you use to determine which files are in which state is
  + git status
* If you run this command directly after a clone, you should see something like this:

$ git status

# On branch master

nothing to commit (working directory clean)

* This means you have a clean working directory
  + in other words, no tracked files are modified
  + Git also doesn’t see any untracked files, or they would be listed here
  + Finally, the command tells you which branch you’re on
    - For now, that is always master
    - Later we’ll talk about branches
* Let’s say you add a new file to your project
  + a simple README file
  + If the file didn’t exist before, and you run git status, you see your untracked file like so:

$ vim README

$ git status

# On branch master

# Untracked files:

# (use "git add <file>..." to include in what will be committed)

#

# README

nothing added to commit but untracked files present (use "git add" to track)

* You can see that your new README file is untracked
  + because it’s under the “Untracked files” heading in your status output
* Untracked basically means that Git sees a file you didn’t have in the previous snapshot (commit)
* Git won’t start including it in your commit snapshots until you explicitly tell it to do so
  + It does this so you don’t accidentally begin including generated binary files or other files that you did not mean to include

**Tracking New Files**

* In order to begin tracking a new file, you use the command git add
* To begin tracking the README file, you can run this:

$ git add README

* If you run your status command again, you can see that your README file is now tracked and staged:

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

#

* You can tell that it’s staged because it’s under the “Changes to be committed” heading.
* If you commit at this point, the version of the file at the time you ran git add is what will be in the historical snapshot.
* The git add command takes a path name for either a file or a directory
  + if it’s a directory, the command adds all the files in that directory recursively.

**Staging Modified Files**

* Let’s change a file that was already tracked.
* If you change a previously tracked file called benchmarks.rb and then run your status command again,
  + you get something that looks like this:

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: benchmarks.rb

#

* The benchmarks.rb file appears under a section named “Changes not staged for commit”
  + means that a file that is tracked has been modified in the working directory but not yet staged
* To stage it, you run the git add command
* Let’s run git add now to stage the benchmarks.rb file, and then run git status again:

$ git add benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

# modified: benchmarks.rb

#

* Both files are staged and will go into your next commit.
* At this point, suppose you remember one little change that you want to make in benchmarks.rb before you commit it.
* You open it again and make that change, and you’re ready to commit.
* let’s run git status one more time:

$ vim benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

# modified: benchmarks.rb

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: benchmarks.rb

#

* Now benchmarks.rb is listed as both staged and unstaged.
  + Git stages a file exactly as it is when you run the git add command.
  + If you commit now, the version of benchmarks.rb as it was when you last ran the git add command is how it will go into the commit
    - not the version of the file as it looks in your working directory when you run git commit
  + If you modify a file after you run git add,
    - you have to run git add again to stage the latest version of the file:

$ git add benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

# modified: benchmarks.rb

#

* **Benefits of the staging area**
  + This staging area allows you to build up changes that will be committed together,
    - while still making other changes to your working copy
  + The staging area is useful
    - e.g. Suppose while fixing a bug you reformat a file
      * You stage the bug fix
      * You leave the reformat changes unstaged
      * Now you commit the bug fix by itself
        + …and can stage/commit the reformatting changes separately

**Ignoring Files**

* Often, you’ll have a class of files that you don’t want Git to automatically add or even show you as being untracked
  + These are generally automatically generated files such as log files or files produced by your build system.
  + In such cases, you can create a file listing patterns to match them named .gitignore
* Example .gitignore:

$ cat .gitignore

\*.[oa]

\*~

* First line tells Git to ignore any files ending in .o or .a
* Second line tells Git to ignore all files that end with a tilde (~)
* Setting up a .gitignore file before you get going is generally a good idea so you don’t accidentally commit files that you really don’t want in your Git repository.
* *Rules for patters you can put in .gitignore:*
  + Blank lines or lines starting with # are ignored.
  + Standard glob patterns work.
  + You can end patterns with a forward slash (/) to specify a directory.
  + You can negate a pattern by starting it with an exclamation point (!).

**Viewing Your Staged and Unstaged Changes**

* If the git status command is too vague for you
  + e.g. you want to know exactly what you changed, not just which files were changed
  + you can use the git diff command
* git diff usually used to answer two questions:
  + What have you changed but not yet staged?
  + What have you staged that you are about to commit?
  + git status answers those questions very generally
  + git diff shows you the exact lines added and removed
* Let’s say you edit and stage the README file again
  + and then edit the benchmarks.rb file without staging it
  + If you run your status command, you once again see something like this:

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: benchmarks.rb

#

* *To see what you’ve changed but not yet staged,*
  + type git diff with no other arguments

$ git diff

diff --git a/benchmarks.rb b/benchmarks.rb

index 3cb747f..da65585 100644

--- a/benchmarks.rb

+++ b/benchmarks.rb

@@ -36,6 +36,10 @@ def main

@commit.parents[0].parents[0].parents[0]

end

+ run\_code(x, 'commits 1') do

+ git.commits.size

+ end

+

run\_code(x, 'commits 2') do

log = git.commits('master', 15)

log.size

* + That command compares what is in your working directory with what is in your staging area.
* *To see what you’ve staged that will go into your next commit,*
  + use git diff –staged

$ git diff --staged

diff --git a/README b/README

new file mode 100644

index 0000000..03902a1

--- /dev/null

+++ b/README2

@@ -0,0 +1,5 @@

+grit

+ by Tom Preston-Werner, Chris Wanstrath

+ http://github.com/mojombo/grit

+

+Grit is a Ruby library for extracting information from a Git repository

* Another example,
  + if you stage the benchmarks.rb file and then edit it,
  + you can use git diff to see the changes in the file that are staged and the changes that are unstaged:

$ git add benchmarks.rb

$ echo '# test line' >> benchmarks.rb

$ git status

# On branch master

#

# Changes to be committed:

#

# modified: benchmarks.rb

#

# Changes not staged for commit:

#

# modified: benchmarks.rb

#

* Now you can use git diff to see what is still unstaged

$ git diff

diff --git a/benchmarks.rb b/benchmarks.rb

index e445e28..86b2f7c 100644

--- a/benchmarks.rb

+++ b/benchmarks.rb

@@ -127,3 +127,4 @@ end

main()

##pp Grit::GitRuby.cache\_client.stats

+# test line

* and git diff --cached to see what you’ve staged so far:

$ git diff --cached

diff --git a/benchmarks.rb b/benchmarks.rb

index 3cb747f..e445e28 100644

--- a/benchmarks.rb

+++ b/benchmarks.rb

@@ -36,6 +36,10 @@ def main

@commit.parents[0].parents[0].parents[0]

end

+ run\_code(x, 'commits 1') do

+ git.commits.size

+ end

+

run\_code(x, 'commits 2') do

log = git.commits('master', 15)

log.size

**Committing Your Changes**

* Now that your staging area is set up the way you want it, you can commit your changes.
  + Remember that anything that is still unstaged won’t go into this commit
* The simplest way to commit is to type git commit:

$ git commit

* Doing so launches your editor of choice.
  + You can configure which editor you’d like to use
* The editor displays the following text (this example is a Vim screen)

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# new file: README

# modified: benchmarks.rb

~

~

~

".git/COMMIT\_EDITMSG" 10L, 283C

* When you exit the editor, Git creates your commit with that commit message (with the comments and diff stripped out).
* Alternatively, you can type your commit message inline with the commit command by specifying it after a -m flag, like this:

$ git commit -m "Story 182: Fix benchmarks for speed"

[master]: created 463dc4f: "Fix benchmarks for speed"

2 files changed, 3 insertions(+), 0 deletions(-)

create mode 100644 README

* Now you’ve created your first commit!
* the commit has given you some output about itself:
  + which branch you committed to (master),
  + what SHA-1 checksum the commit has (463dc4f),
  + how many files were changed,
  + and statistics about lines added and removed in the commit.
* Every time you perform a commit, you’re recording a snapshot of your project that you can revert to or compare to later.

**Skipping the Staging Area**

* Although it can be amazingly useful for crafting commits exactly how you want them,
  + the staging area is sometimes a bit more complex than you need in your workflow.
* If you want to skip the staging area, Git provides a simple shortcut.
* Providing the -a option to the git commit command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the git add part:

$ git status

# On branch master

#

# Changes not staged for commit:

#

# modified: benchmarks.rb

#

$ git commit -a -m 'added new benchmarks'

[master 83e38c7] added new benchmarks

1 files changed, 5 insertions(+), 0 deletions(-)

* Notice how you don’t have to run git add on the benchmarks.rb file in this case before you commit.

**Removing Files**

* To remove a file from Git,
  + You have to remove it from your tracked files (remove it from your staging area) and then commit.
  + The git rm command does that
    - and also removes the file from your working directory so you don’t see it as an untracked file next time around.
* If you simply remove the file from your working directory,
  + it shows up under the “Changes not staged for commit” (that is, unstaged) area of your git status output:

$ rm grit.gemspec

$ git status

# On branch master

#

# Changes not staged for commit:

# (use "git add/rm <file>..." to update what will be committed)

#

# deleted: grit.gemspec

#

* Then, if you run git rm, it stages the file’s removal:

$ git rm grit.gemspec

rm 'grit.gemspec'

$ git status

# On branch master

#

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# deleted: grit.gemspec

#

* The next time you commit, the file will be gone and no longer tracked.
* If you modified the file and added it to the index already,
  + you must force the removal with the -f option.
  + This is a safety feature to prevent accidental removal of data that hasn’t yet been recorded in a snapshot and that can’t be recovered from Git.

**Moving Files**

* Unlike many other VCS systems, Git doesn’t explicitly track file movement.
* If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file.
* However, Git is pretty smart about figuring that out after the fact — we’ll deal with detecting file movement a bit later.
* If you want to rename a file in Git, you can run something like

$ git mv file\_from file\_to

* if you run something like this and look at the status, you’ll see that Git considers it a renamed file:

$ git mv README.txt README

$ git status

# On branch master

# Your branch is ahead of 'origin/master' by 1 commit.

#

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# renamed: README.txt -> README

#

* However, this is equivalent to running something like this:

$ mv README.txt README

$ git rm README.txt

$ git add README

* Git figures out that it’s a rename implicitly, so it doesn’t matter if you rename a file that way or with the mv command.
* The only real difference is that mv is one command instead of three — it’s a convenience function.

## Viewing the Commit History

* The most basic and powerful tool to view the commit history this is the git log command.
* Here is an example of running on git log on a sample project:

$ git log

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 10:31:28 2008 -0700

first commit

* By default, with no arguments, git log lists the commits made in that repository in reverse chronological order.
* This command lists each commit with its:
  + SHA-1 checksum
  + the author’s name and e-mail
  + the date written
  + the commit message.
* One of the more helpful options is -p,
  + which shows the diff introduced in each commit.
  + You can also use -2,
    - which limits the output to only the last two entries:

$ git log -p -2

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

diff --git a/Rakefile b/Rakefile

index a874b73..8f94139 100644

--- a/Rakefile

+++ b/Rakefile

@@ -5,5 +5,5 @@ require 'rake/gempackagetask'

spec = Gem::Specification.new do |s|

s.name = "simplegit"

- s.version = "0.1.0"

+ s.version = "0.1.1"

s.author = "Scott Chacon"

s.email = "schacon@gee-mail.com

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test code

diff --git a/lib/simplegit.rb b/lib/simplegit.rb

index a0a60ae..47c6340 100644

--- a/lib/simplegit.rb

+++ b/lib/simplegit.rb

@@ -18,8 +18,3 @@ class SimpleGit

end

end

-

-if $0 == \_\_FILE\_\_

- git = SimpleGit.new

- puts git.show

-end

\ No newline at end of file

* This option displays the same information but with a diff directly following each entry.
* Sometimes it's easier to review changes on the word level rather than on the line level.
  + There is a --word-diff option available in Git, that you can append to the git log -p command to get word diff instead of normal line by line diff.
  + Word diff format is quite useless when applied to source code, but it comes in handy when applied to large text files, like books or your dissertation.

$ git log -U1 --word-diff

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

diff --git a/Rakefile b/Rakefile

index a874b73..8f94139 100644

--- a/Rakefile

+++ b/Rakefile

@@ -7,3 +7,3 @@ spec = Gem::Specification.new do |s|

s.name = "simplegit"

s.version = **[-"0.1.0"-]{+"0.1.1"+}**

s.author = "Scott Chacon"

* There is no added and removed lines in this output as in a normal diff.
  + Changes are shown inline instead.
  + You can see the added word enclosed in {+ +} and removed one enclosed in [- -].
* You can also use a series of summarizing options with git log
  + For example, if you want to see some abbreviated stats for each commit, you can use the --stat option:

$ git log --stat

commit ca82a6dff817ec66f44342007202690a93763949

Author: Scott Chacon <schacon@gee-mail.com>

Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

Rakefile | 2 +-

1 files changed, 1 insertions(+), 1 deletions(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

removed unnecessary test code

lib/simplegit.rb | 5 -----

1 files changed, 0 insertions(+), 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 10:31:28 2008 -0700

first commit

README | 6 ++++++

Rakefile | 23 +++++++++++++++++++++++

lib/simplegit.rb | 25 +++++++++++++++++++++++++

3 files changed, 54 insertions(+), 0 deletions(-)

* the --stat option prints below each commit entry:
  + a list of modified files
  + how many files were changed
  + how many lines in those files were added and removed
* The most interesting option is format
  + which allows you to specify your own log output format.
  + This is especially useful when you’re generating output for machine parsing
  + Some of the more useful options that format takes:

| **Option** | **Description of Output** |
| --- | --- |
| %H | Commit hash |
| %h | Abbreviated commit hash |
| %T | Tree hash |
| %t | Abbreviated tree hash |
| %P | Parent hashes |
| %p | Abbreviated parent hashes |
| %an | Author name |
| %ae | Author e-mail |
| %ad | Author date (format respects the --date= option) |
| %ar | Author date, relative |
| %cn | Committer name |
| %ce | Committer email |
| %cd | Committer date |
| %cr | Committer date, relative |
| %s | Subject |

* + You may be wondering what the difference is between author and committer.
    - author is the person who originally wrote the patch
    - committer is the person who last applied the patch
    - If you send in a patch to a project and one of the core members applies the patch, both of you get credit — you as the author and the core member as the committer.
* The oneline and format options are particularly useful with another log option called –graph
  + This option adds a nice little ASCII graph showing your branch and merge history,
    - which we can see in our copy of the Grit project repository:

$ git log --pretty=format:"%h %s" --graph

\* 2d3acf9 ignore errors from SIGCHLD on trap

\* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit

|\

| \* 420eac9 Added a method for getting the current branch.

\* | 30e367c timeout code and tests

\* | 5a09431 add timeout protection to grit

\* | e1193f8 support for heads with slashes in them

|/

\* d6016bc require time for xmlschema

\* 11d191e Merge branch 'defunkt' into local

* Those are only some simple output-formatting options to git log — there are many more.
* The following lists the options we’ve covered so far and some other common formatting options that may be useful, along with how they change the output of the log command.

| **Option** | **Description** |
| --- | --- |
| -p | Show the patch introduced with each commit. |
| --word-diff | Show the patch in a word diff format. |
| --stat | Show statistics for files modified in each commit. |
| --shortstat | Display only the changed/insertions/deletions line from the --stat command. |
| --name-only | Show the list of files modified after the commit information. |
| --name-status | Show the list of files affected with added/modified/deleted information as well. |
| --abbrev-commit | Show only the first few characters of the SHA-1 checksum instead of all 40. |
| --relative-date | Display the date in a relative format (for example, “2 weeks ago”) instead of using the full date format. |
| --graph | Display an ASCII graph of the branch and merge history beside the log output. |
| --pretty | Show commits in an alternate format. Options include oneline, short, full, fuller, and format (where you specify your own format). |
| --oneline | A convenience option short for --pretty=oneline --abbrev-commit. |

**Limiting Log Output**

* In addition to output-formatting options, git log takes a number of useful limiting options
  + the time-limiting options such as --since and --until are very useful
  + this command gets the list of commits made in the last two weeks:

$ git log --since=2.weeks

* You can also filter the list to commits that match some search criteria
  + The --author option allows you to filter on a specific author
  + the --grep option lets you search for keywords in the commit messages.
* The last really useful option to pass to git log as a filter is a path.
  + If you specify a directory or file name, you can limit the log output to commits that introduced a change to those files
  + This is always the last option and is generally preceded by double dashes (--) to separate the paths from the options.
* Here are a few other common options:

| **Option** | **Description** |
| --- | --- |
| -(n) | Show only the last n commits |
| --since, --after | Limit the commits to those made after the specified date. |
| --until, --before | Limit the commits to those made before the specified date. |
| --author | Only show commits in which the author entry matches the specified string. |
| --committer | Only show commits in which the committer entry matches the specified string. |

* Example,
  + if you want to see which commits modifying test files in the Git source code history were committed by Junio Hamano in the month of October 2008 and were not merges, you can run something like this:

$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \

--before="2008-11-01" --no-merges -- t/

5610e3b - Fix testcase failure when extended attribute

acd3b9e - Enhance hold\_lock\_file\_for\_{update,append}()

f563754 - demonstrate breakage of detached checkout wi

d1a43f2 - reset --hard/read-tree --reset -u: remove un

51a94af - Fix "checkout --track -b newbranch" on detac

b0ad11e - pull: allow "git pull origin $something:$cur

**Using a GUI to Visualize History**

* If you like to use a more graphical tool to visualize your commit history, you may want to take a look at a Tcl/Tk program called gitk that is distributed with Git
* 
* We’ll cover more GUI options later

## Undoing Things

* At any stage, you may want to undo something
* Here, we’ll review a few basic tools for undoing changes that you’ve made.
* Be careful, because you can’t always revert some of these undos
* This is one of the few areas in Git where you may lose some work if you do it wrong.

**Changing Your Last Commit**

* One of the common undos takes place when you commit too early and possibly forget to add some files, or you mess up your commit message.
* If you want to try that commit again, you can run commit with the --amend option:

$ git commit --amend

* This takes your staging area and uses it for the commit.
* If you’ve made no changes since your last commit (for instance, you run this command immediately after your previous commit), then your snapshot will look exactly the same and all you’ll change is your commit message.
* The same commit-message editor fires up, but it already contains the message of your previous commit.
  + You can edit the message the same as always, but it overwrites your previous commit.
* Example,
  + if you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:

$ git commit -m 'initial commit'

$ git add forgotten\_file

$ git commit --amend

* After these three commands, you end up with a single commit —
  + the second commit replaces the results of the first.

**Unstaging a Staged File**

* The next two sections demonstrate how to wrangle your staging area and working directory changes.
  + The command you use to determine the state of those two areas also reminds you how to undo changes to them.
  + Example,
    - Let’s say you’ve changed two files and want to commit them as two separate changes, but you accidentally type git add \* and stage them both.
    - How can you unstage one of the two?
      * The git status command reminds you:

$ git add .

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: README.txt

# modified: benchmarks.rb

#

* + - * Right below the “Changes to be committed” text, it says "use git reset HEAD <file>... to unstage"
      * So, let’s use that advice to unstage the benchmarks.rb file:

$ git reset HEAD benchmarks.rb

benchmarks.rb: locally modified

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: README.txt

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: benchmarks.rb

#

* The benchmarks.rb file is modified but once again unstaged.

**Unmodifying a Modified File**

* What if you realize that you don’t want to keep your changes to the benchmarks.rb file?
  + How can you easily unmodify it — revert it back to what it looked like when you last committed
    - (or initially cloned, or however you got it into your working directory)?
    - Luckily, git status tells you how to do that, too. In the last example output, the unstaged area looks like this:

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# modified: benchmarks.rb

#

* + - It tells you pretty explicitly how to discard the changes you’ve made
    - Let’s do what it says:

$ git checkout -- benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: README.txt

#

* + - You can see that the changes have been reverted.
    - this is a dangerous command: any changes you made to that file are gone — you just copied another file over it.
    - Don’t ever use this command unless you absolutely know that you don’t want the file.
* Remember, anything that is committed in Git can almost always be recovered.
  + Even commits that were on branches that were deleted or commits that were overwritten with an --amend commit can be recovered
  + *However, anything you lose that was never committed is likely never to be seen again.*

## Working with Remotes

* To be able to collaborate on any Git project,
  + you need to know how to manage your remote repositories.
* Remote repositories are versions of your project that are hosted on the Internet or network somewhere.
* You can have several of them, each of which generally is either read-only or read/write for you.
* Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work.
* Managing remote repositories includes knowing how to
  + add remote repositories
  + remove remotes that are no longer valid
  + manage various remote branches and define them as being tracked or not
  + and more.

**Showing Your Remotes**

* To see which remote servers you have configured, you can run the git remote command.
  + It lists the shortnames of each remote handle you’ve specified.
  + If you’ve cloned your repository, you should at least see origin — that is the default name Git gives to the server you cloned from:

$ git clone git://github.com/schacon/ticgit.git

Initialized empty Git repository in /private/tmp/ticgit/.git/

remote: Counting objects: 595, done.

remote: Compressing objects: 100% (269/269), done.

remote: Total 595 (delta 255), reused 589 (delta 253)

Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.

Resolving deltas: 100% (255/255), done.

$ cd ticgit

$ git remote

origin

* You can also specify -v, which shows you the URL that Git has stored for the shortname to be expanded to:

$ git remote -v

origin git://github.com/schacon/ticgit.git (fetch)

origin git://github.com/schacon/ticgit.git (push)

* If you have more than one remote, the command lists them all:

$ cd grit

$ git remote -v

bakkdoor git://github.com/bakkdoor/grit.git

cho45 git://github.com/cho45/grit.git

defunkt git://github.com/defunkt/grit.git

koke git://github.com/koke/grit.git

origin git@github.com:mojombo/grit.git

**Adding Remote Repositories**

* To add a new remote Git repository as a shortname you can reference easily, run git remote add [shortname] [url]:

$ git remote

origin

$ git remote add pb git://github.com/paulboone/ticgit.git

$ git remote -v

origin git://github.com/schacon/ticgit.git

pb git://github.com/paulboone/ticgit.git

* Now you can use the string pb on the command line in lieu of the whole URL.
* Example:
  + If you want to fetch all the information that Paul has but that you don’t yet have in your repository, you can run git fetch pb *(fetch covered in in next section)*:

$ git fetch pb

remote: Counting objects: 58, done.

remote: Compressing objects: 100% (41/41), done.

remote: Total 44 (delta 24), reused 1 (delta 0)

Unpacking objects: 100% (44/44), done.

From git://github.com/paulboone/ticgit

\* [new branch] master -> pb/master

\* [new branch] ticgit -> pb/ticgit

* Paul’s master branch is accessible locally as pb/master
  + you can merge it into one of your branches,
  + or you can check out a local branch at that point if you want to inspect it.

**Fetching and Pulling from Your Remotes**

* To get data from your remote projects, you can run:

$ git fetch [remote-name]

* The command goes out to that remote project and pulls down all the data from that remote project that you don’t have yet.
* After you do this,
  + you should have references to all the branches from that remote,
  + which you can merge in or inspect at any time.
* If you clone a repository, the command automatically adds that remote repository under the name origin.
  + git fetch origin fetches any new work that has been pushed to that server since you cloned (or last fetched from) it.
  + the fetch command pulls the data to your local repository —
    - it doesn’t automatically merge it with any of your work or modify what you’re currently working on.
    - You have to merge it manually into your work when you’re ready.
* If you have a branch set up to track a remote branch
  + you can use the git pull command to automatically fetch and then merge a remote branch into your current branch.
  + Branching & merging will be covered later
  + Using git pull may be an easier workflow for you.
    - By default, the git clone command automatically sets up your local master branch to track the remote master branch on the server you cloned from (assuming the remote has a master branch).
  + *Running git pull generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you’re currently working on.*

**Pushing to Your Remotes**

* When you have your project at a point that you want to share, you have to push it upstream.
  + The command for this is simple: git push [remote-name] [branch-name]
* If you want to push your master branch to your origin server then you can run this to push your work back up to the server:

$ git push origin master

* *This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime.*
  + If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected.
  + You’ll have to pull down their work first and incorporate it into yours before you’ll be allowed to push.

**Inspecting a Remote**

* If you want to see more information about a particular remote, you can use the git remote show [remote-name] command.

$ git remote show origin

\* remote origin

URL: git://github.com/schacon/ticgit.git

Remote branch merged with 'git pull' while on branch master

master

Tracked remote branches

master

ticgit

* This command shows:
  + the URL for the remote repository as well as the tracking branch information.
  + that if you’re on the master branch and you run git pull,
    - it will automatically merge in the master branch on the remote after it fetches all the remote references.
  + all the remote references it has pulled down.
* When you’re using Git more heavily, however, you may see much more information from git remote show:

$ git remote show origin

\* remote origin

URL: git@github.com:defunkt/github.git

Remote branch merged with 'git pull' while on branch issues

issues

Remote branch merged with 'git pull' while on branch master

master

New remote branches (next fetch will store in remotes/origin)

caching

Stale tracking branches (use 'git remote prune')

libwalker

walker2

Tracked remote branches

acl

apiv2

dashboard2

issues

master

postgres

Local branch pushed with 'git push'

master:master

* This command shows:
  + which branch is automatically pushed when you run git push on certain branches.
  + which remote branches on the server you don’t yet have,
  + which remote branches you have that have been removed from the server,
  + multiple branches that are automatically merged when you run git pull.

**Removing and Renaming Remotes**

* If you want to rename a reference you can run git remote rename to change a remote’s shortname.

$ git remote rename pb paul

$ git remote

origin

paul

* This changes your remote branch names, too.
  + What used to be referenced at pb/master is now at paul/master.
* If you want to remove a reference for some reason
  + e.g. you’ve moved the server or are no longer using a particular mirror
  + e.g. or perhaps a contributor isn’t contributing anymore
  + you can use git remote rm:

$ git remote rm paul

$ git remote

origin

## Tagging

* Git has the ability to tag specific points in history as being important.
* Generally, people use this functionality to mark release points (v1.0, and so on).

**Listing Your Tags**

* Listing the available tags in Git is straightforward. Just type git tag:

$ git tag

v0.1

v1.3

* This command lists the tags in alphabetical order;
  + the order in which they appear has no real importance.
* You can also search for tags with a particular pattern.
  + The Git source repo, for instance, contains more than 240 tags.
  + If you’re only interested in looking at the 1.4.2 series, you can run this:

$ git tag -l 'v1.4.2.\*'

v1.4.2.1

v1.4.2.2

v1.4.2.3

v1.4.2.4

**Creating Tags**

* Git uses two main types of tags: lightweight and annotated.
  + lightweight tag is very much like a branch that doesn’t change — it’s just a pointer to a specific commit.
  + Annotated tags, however, are stored as full objects in the Git database.
    - They’re:
      * checksummed
      * contain the tagger name, e-mail, and date
      * have a tagging message
      * can be signed and verified with GNU Privacy Guard (GPG).
  + It’s generally recommended that you create annotated tags so you can have all this information
    - but if you want a temporary tag or for some reason don’t want to keep the other information, lightweight tags are available too.

**Annotated Tags**

* Creating an annotated tag in Git is simple. The easiest way is to specify -a when you run the tag command:

$ git tag -a v1.4 -m 'my version 1.4'

$ git tag

v0.1

v1.3

v1.4

* The -m specifies a tagging message, which is stored with the tag.
  + If you don’t specify a message for an annotated tag, Git launches your editor so you can type it in.
* You can see the tag data along with the commit that was tagged by using the git show command:

$ git show v1.4

tag v1.4

Tagger: Scott Chacon <schacon@gee-mail.com>

Date: Mon Feb 9 14:45:11 2009 -0800

my version 1.4

commit 15027957951b64cf874c3557a0f3547bd83b3ff6

Merge: 4a447f7... a6b4c97...

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'

* That shows:
  + the tagger information
  + the date the commit was tagged
  + the annotation message before showing the commit information.

**Signed Tags**

* You can also sign your tags with GPG, assuming you have a private key.
  + to do this use -s instead of -a:

$ git tag -s v1.5 -m 'my signed 1.5 tag'

You need a passphrase to unlock the secret key for

user: "Scott Chacon <schacon@gee-mail.com>"

1024-bit DSA key, ID F721C45A, created 2009-02-09

* If you run git show on that tag, you can see your GPG signature attached to it:

$ git show v1.5

tag v1.5

Tagger: Scott Chacon <schacon@gee-mail.com>

Date: Mon Feb 9 15:22:20 2009 -0800

my signed 1.5 tag

-----BEGIN PGP SIGNATURE-----

Version: GnuPG v1.4.8 (Darwin)

iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN

Ki0An2JeAVUCAiJ7Ox6ZEtK+NvZAj82/

=WryJ

-----END PGP SIGNATURE-----

commit 15027957951b64cf874c3557a0f3547bd83b3ff6

Merge: 4a447f7... a6b4c97...

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'

**Lightweight Tags**

* Another way to tag commits is with a lightweight tag.
  + This is basically the commit checksum stored in a file — no other information is kept.
  + To create a lightweight tag, don’t supply the -a, -s, or -m option:

$ git tag v1.4-lw

$ git tag

v0.1

v1.3

v1.4

v1.4-lw

v1.5

* This time, if you run git show on the tag, you don’t see the extra tag information.
* The command just shows the commit:

$ git show v1.4-lw

commit 15027957951b64cf874c3557a0f3547bd83b3ff6

Merge: 4a447f7... a6b4c97...

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'

**Verifying Tags**

* To verify a signed tag, you use git tag -v [tag-name].
* This command uses GPG to verify the signature.
* You need the signer’s public key in your keyring for this to work properly:

$ git tag -v v1.4.2.1

object 883653babd8ee7ea23e6a5c392bb739348b1eb61

type commit

tag v1.4.2.1

tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.

gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A

gpg: Good signature from "Junio C Hamano <junkio@cox.net>"

gpg: aka "[jpeg image of size 1513]"

Primary key fingerprint: 3565 2A26 2040 E066 C9A7 4A7D C0C6 D9A4 F311 9B9A

* If you don’t have the signer’s public key, you get something like this instead:

gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A

gpg: Can't check signature: public key not found

error: could not verify the tag 'v1.4.2.1'

**Tagging Later**

* You can also tag commits after you’ve moved past them.
* Suppose your commit history looks like this:

$ git log --pretty=oneline

15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'

a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support

0d52aaab4479697da7686c15f77a3d64d9165190 one more thing

6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'

0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function

4682c3261057305bdd616e23b64b0857d832627b added a todo file

166ae0c4d3f420721acbb115cc33848dfcc2121a started write support

9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile

964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo

8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme

* Now, suppose you forgot to tag the project at v1.2, which was at the "updated rakefile" commit.
* You can add it after the fact.
* To tag that commit, you specify the commit checksum (or part of it) at the end of the command:

$ git tag -a v1.2 -m 'version 1.2' 9fceb02

* You can see that you’ve tagged the commit:

$ git tag

v0.1

v1.2

v1.3

v1.4

v1.4-lw

v1.5

$ git show v1.2

tag v1.2

Tagger: Scott Chacon <schacon@gee-mail.com>

Date: Mon Feb 9 15:32:16 2009 -0800

version 1.2

commit 9fceb02d0ae598e95dc970b74767f19372d61af8

Author: Magnus Chacon <mchacon@gee-mail.com>

Date: Sun Apr 27 20:43:35 2008 -0700

updated rakefile

...

**Sharing Tags**

* By default, the git push command doesn’t transfer tags to remote servers.
* You will have to explicitly push tags to a shared server after you have created them.
* This process is just like sharing remote branches — you can run git push origin [tagname]

$ git push origin v1.5

Counting objects: 50, done.

Compressing objects: 100% (38/38), done.

Writing objects: 100% (44/44), 4.56 KiB, done.

Total 44 (delta 18), reused 8 (delta 1)

To git@github.com:schacon/simplegit.git

\* [new tag] v1.5 -> v1.5

* If you have a lot of tags that you want to push up at once,
  + you can also use the --tags option to the git push command.
  + This will transfer all of your tags to the remote server that are not already there.

$ git push origin --tags

Counting objects: 50, done.

Compressing objects: 100% (38/38), done.

Writing objects: 100% (44/44), 4.56 KiB, done.

Total 44 (delta 18), reused 8 (delta 1)

To git@github.com:schacon/simplegit.git

\* [new tag] v0.1 -> v0.1

\* [new tag] v1.2 -> v1.2

\* [new tag] v1.4 -> v1.4

\* [new tag] v1.4-lw -> v1.4-lw

\* [new tag] v1.5 -> v1.5

* Now, when someone else clones or pulls from your repository,
  + they will get all your tags as well.

# Git Branching

## Git’s Killer Feature – Branching

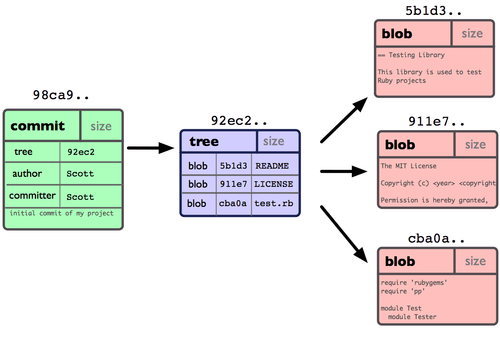
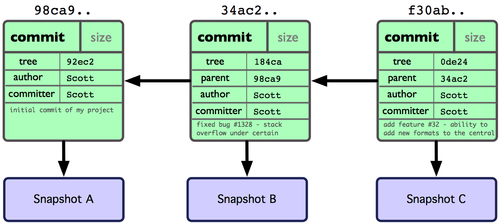
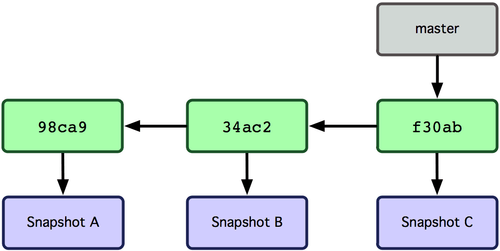
* Nearly every VCS has some form of branching support.
* Branching means you diverge from the main line of development
  + ...and continue to do work without messing with that main line.
* In many VCS tools,
  + this is a somewhat expensive process,
  + often requiring you to create a new copy of your source code directory,
  + which can take a long time for large projects.
* Some people refer to the branching model in Git as its “killer feature”
  + it certainly sets Git apart in the VCS community.
* Why is it so special?
  + The way Git branches is incredibly lightweight,
  + making branching operations nearly instantaneous
  + and switching back and forth between branches generally just as fast.
* Unlike many other VCSs,
  + Git encourages a workflow that branches and merges often,
  + even multiple times in a day.
* Understanding and mastering this feature gives you a powerful and unique tool and can literally change the way that you develop.

## What a Branch Is

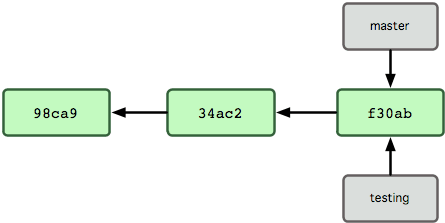
* To really understand the way Git does branching, we need to take a step back and examine how Git stores its data
* When you commit in Git, Git stores:
  + a commit object that contains a pointer to:
    - the snapshot of the content you staged,
    - the author and message metadata,
    - and zero or more pointers to the commit or commits that were the direct parents of this commit:
      * zero parents for the first commit,
      * one parent for a normal commit,
      * and multiple parents for a commit that results from a merge of two or more branches.
* To visualize this, let’s assume that you have a directory containing three files, and you stage them all and commit.
  + Staging the files checksums each one (SHA-1 hash),
  + stores that version of the file in the Git repository (Git refers to them as blobs),
  + and adds that checksum to the staging area:

$ git add README test.rb LICENSE

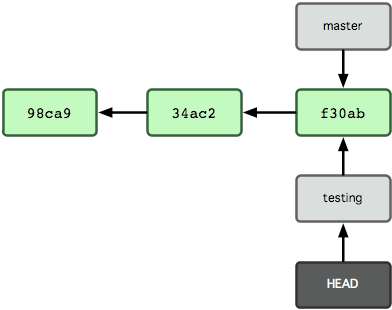
$ git commit -m 'initial commit of my project'

* When you create the commit by running git commit,
  + Git checksums each subdirectory (in this case, just the root project directory) and stores those tree objects in the Git repository.
  + Git then creates a commit object that has the metadata and a pointer to the root project tree so it can re-create that snapshot when needed.
* Your Git repository now contains five objects:
  + one blob for the contents of each of your three files,
  + one tree that lists the contents of the directory and specifies which file names are stored as which blobs,
  + and one commit with the pointer to that root tree and all the commit metadata.
* Conceptually, the data in your Git repository looks something like this:
* 
* If you make some changes and commit again,
  + the next commit stores a pointer to the commit that came immediately before it.
* After two more commits, your history might look something like this:
* 
* A branch in Git is simply a lightweight movable pointer to one of these commits.
* The default branch name in Git is master.
  + As you initially make commits, you’re given a master branch that points to the last commit you made.
  + Every time you commit, it moves forward automatically.
  + 
* What happens if you create a new branch?
  + Doing so creates a new pointer for you to move around.
* Let’s say you create a new branch called testing.
* You do this with the git branch command:

$ git branch testing

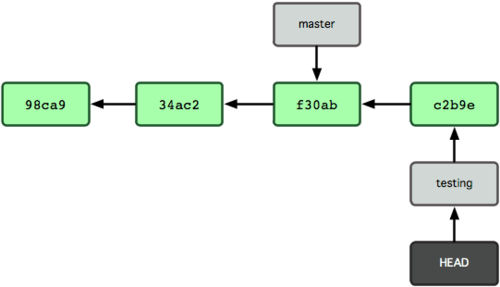
* This creates a new pointer at the same commit you’re currently on:
* 
* How does Git know what branch you’re currently on?
  + It keeps a special pointer called HEAD.
  + Note that this is a lot different than the concept of HEAD in other VCSs you may be used to, such as Subversion or CVS.
  + In Git, this is a pointer to the local branch you’re currently on.
  + In this case, you’re still on master.
  + The git branch command only created a new branch — it didn’t switch to that branch.
  + 
* To switch to an existing branch, you run the git checkout command.
* Let’s switch to the new testing branch:

$ git checkout testing

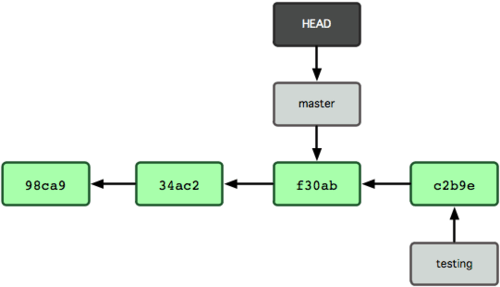
* This moves HEAD to point to the testing branch.
* 
* What is the significance of that?
* Well, let’s do another commit:

$ vim test.rb

$ git commit -a -m 'made a change'

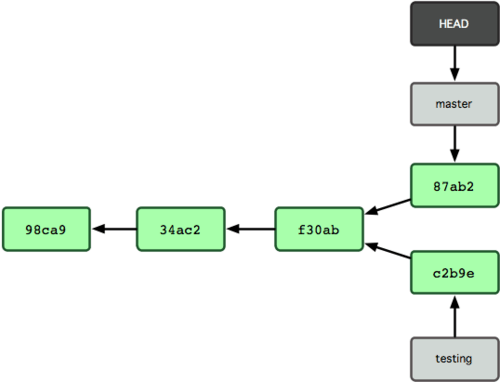
* 
* Now your testing branch has moved forward,
  + but your master branch still points to the commit you were on when you ran git checkout to switch branches.
* Let’s switch back to the master branch:

$ git checkout master

* 
* That command did two things.
  + It moved the HEAD pointer back to point to the master branch
  + It reverted the files in your working directory back to the snapshot that master points to.
  + This also means the changes you make from this point forward will diverge from an older version of the project.
  + It essentially rewinds the work you’ve done in your testing branch temporarily so you can go in a different direction.
* Let’s make a few changes and commit again:

$ vim test.rb

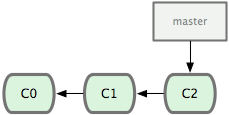
$ git commit -a -m 'made other changes'

* Now your project history has diverged.
  + You created and switched to a branch,
    - did some work on it,
    - and then switched back to your main branch and did other work.
  + Both of those changes are isolated in separate branches:
    - you can switch back and forth between the branches and merge them together when you’re ready.
  + You did all that with simple branch and checkout commands.
* 
* **Branches are cheap**
  + Because a branch in Git is in actuality a simple file that contains the 40 character SHA-1 checksum of the commit it points to,
    - ***branches are cheap to create and destroy.***
  + Creating a new branch is as quick and simple as writing 41 bytes to a file (40 characters and a newline).
  + This is in sharp contrast to the way most VCS tools branch,
    - which involves copying all of the project’s files into a second directory.
    - This can take several seconds or even minutes, depending on the size of the project, whereas in Git the process is always instantaneous.
  + Because we’re recording the parents when we commit,
    - finding a proper merge base for merging is automatically done for us and is generally very easy to do.
    - These features help encourage developers to create and use branches often.
  + Let’s see why you should do so...

## Basic Branching and Merging

* Let’s go through a simple example of branching and merging with a workflow that you might use in the real world.
* You’ll follow these steps:
  1. Do work on a web site.
  2. Create a branch for a new story you’re working on.
  3. Do some work in that branch.
* At this stage, you’ll receive a call that another issue is critical and you need a hotfix.
* You’ll do the following:
  1. Revert back to your production branch.
  2. Create a branch to add the hotfix.
  3. After it’s tested, merge the hotfix branch, and push to production.
  4. Switch back to your original story and continue working.

**Basic Branching**

* First, let’s say you’re working on your project and have a couple of commits already:
* 
* You’ve decided that you’re going to work on issue #53 in whatever issue-tracking system your company uses.
  + Because issue #53 is a focused topic that you want to work on,
    - you’ll create a new branch in which to work.
  + To create a branch and switch to it at the same time,
    - you can run the git checkout command with the -b switch:

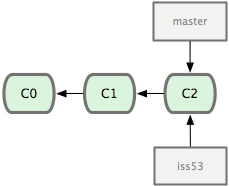
$ git checkout -b iss53

Switched to a new branch "iss53"

* + - This is shorthand for:

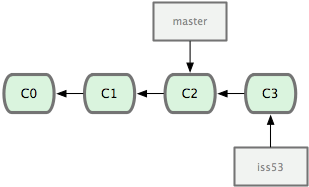
$ git branch iss53

$ git checkout iss53

* + - 
* You work on your web site and do some commits.
  + Doing so moves the iss53 branch forward,
    - because you have it checked out (that is, your HEAD is pointing to it)

$ vim index.html

$ git commit -a -m 'added a new footer [issue 53]'

* + - 
* Now you get the call that there is an issue with the web site, and you need to fix it immediately.
  + With Git, you don’t have to deploy your fix along with the iss53 changes you’ve made,
    - and you don’t have to put a lot of effort into reverting those changes before you can work on applying your fix to what is in production.
    - All you have to do is switch back to your master branch.
    - Before you do that,
      * if your working directory or staging area has uncommitted changes that conflict with the branch you’re checking out,
      * Git won’t let you switch branches
      * It’s best to have a clean working state when you switch branches.
      * There are ways to get around this (namely, stashing and commit amending).
      * For now, you’ve committed all your changes, so you can switch back to your master branch:

$ git checkout master

Switched to branch "master"

* At this point, your project working directory is exactly the way it was before you started working on issue #53
  + This is an important point to remember:
    - Git resets your working directory to look like the snapshot of the commit that the branch you check out points to.
    - It adds, removes, and modifies files automatically to make sure your working copy is what the branch looked like on your last commit to it.
* Next, you have a hotfix to make.
  + Let’s create a hotfix branch on which to work until it’s completed:

$ git checkout -b hotfix

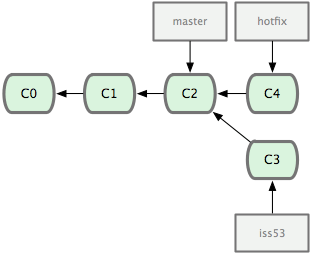
Switched to a new branch "hotfix"

$ vim index.html

$ git commit -a -m 'fixed the broken email address'

[hotfix]: created 3a0874c: "fixed the broken email address"

1 files changed, 0 insertions(+), 1 deletions(-)

* + 
* You can run your tests,
  + make sure the hotfix is what you want,
  + and merge it back into your master branch to deploy to production.
  + You do this with the git merge command:

$ git checkout master

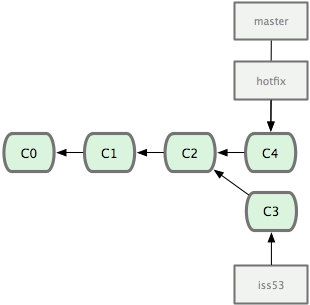
$ git merge hotfix

Updating f42c576..3a0874c

Fast forward

README | 1 -

1 files changed, 0 insertions(+), 1 deletions(-)

* + You’ll notice the phrase ***"Fast forward"*** in that merge.
    - Because the commit pointed to by the branch you merged in was directly upstream of the commit you’re on,
      * Git moves the pointer forward.
    - i.e. When you try to merge one commit with a commit that can be reached by following the first commit’s history,
      * Git simplifies things by moving the pointer forward because there is no divergent work to merge together — this is called a "fast forward".
* Your change is now in the snapshot of the commit pointed to by the master branch, and you can deploy your change
  + 
* After your super-important fix is deployed,
  + You’re ready to switch back to the work you were doing before you were interrupted.
  + However, first you’ll delete the hotfix branch,
    - because you no longer need it — the master branch points at the same place.
    - You can delete it with the -d option to git branch:

$ git branch -d hotfix

Deleted branch hotfix (3a0874c).

* Now you can switch back to your work-in-progress branch on issue #53 and continue working on it

$ git checkout iss53

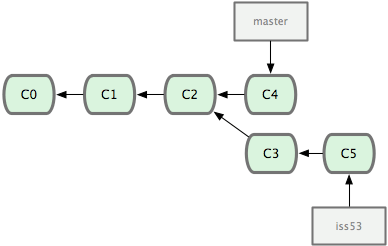
Switched to branch "iss53"

$ vim index.html

$ git commit -a -m 'finished the new footer [issue 53]'

[iss53]: created ad82d7a: "finished the new footer [issue 53]"

1 files changed, 1 insertions(+), 0 deletions(-)

* + 
* ***The work you did in your hotfix branch is not contained in the files in your iss53 branch***.
  + If you need to pull it in,
    - you can merge your master branch into your iss53 branch by running git merge master,
    - or you can wait to integrate those changes until you decide to pull the iss53 branch back into master later.

**Basic Merging**

* Suppose you’ve decided that your issue #53 work is complete and ready to be merged into your master branch.
  + In order to do that, you’ll merge in your iss53 branch, much like you merged in your hotfix branch earlier.
  + All you have to do is check out the branch you wish to merge into and then run the git merge command:

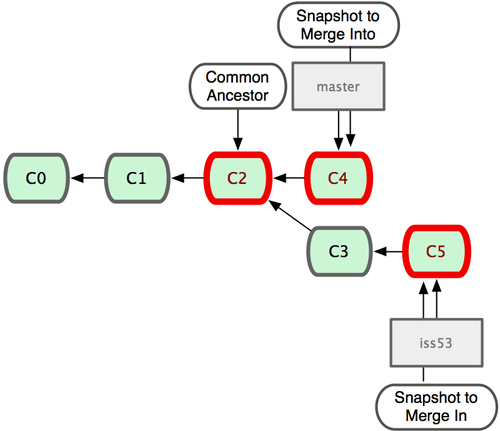
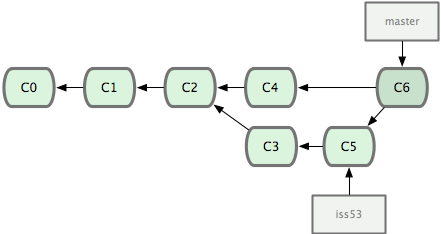
$ git checkout master

$ git merge iss53

Merge made by recursive.

README | 1 +

1 files changed, 1 insertions(+), 0 deletions(-)

* + This looks a bit different than the hotfix merge you did earlier.
  + In this case, your development history has diverged from some older point.
  + Because the commit on the branch you’re on isn’t a direct ancestor of the branch you’re merging in, Git has to do some work.
  + In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.
  + 
* *Git automatically identifies the best common-ancestor merge base for branch merging.*
  + Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it (see Figure 3-17).
    - This is referred to as a merge commit and is special in that it has more than one parent.
* Git determines the best common ancestor to use for its merge base;
  + This is different than CVS or Subversion (before version 1.5),
    - where the developer doing the merge has to figure out the best merge base for themselves.
    - ***This makes merging much easier in Git than in these other systems.***
    - The following figure shows the automatically created merge commit (C6) with two parents:
    - 
* Now that your work is merged in, you have no further need for the iss53 branch.
  + You can delete it and then manually close the ticket in your ticket-tracking system:

$ git branch -d iss53

**Basic Merge Conflicts**

* Occasionally, this process doesn’t go smoothly.
  + If you changed the same part of the same file differently in the two branches you’re merging together,
    - Git won’t be able to merge them cleanly.
  + If your fix for issue #53 modified the same part of a file as the hotfix, you’ll get a merge conflict that looks something like this:

$ git merge iss53

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

Automatic merge failed; fix conflicts and then commit the result.

* + Git hasn’t automatically created a new merge commit.
    - It has paused the process while you resolve the conflict.
  + If you want to see which files are unmerged at any point after a merge conflict, you can run git status:

[master\*]$ git status

index.html: needs merge

# On branch master

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

# (use "git checkout -- <file>..." to discard changes in working directory)

#

# unmerged: index.html

#

* + Anything that has merge conflicts and hasn’t been resolved is listed as unmerged.
  + Git adds standard conflict-resolution markers to the files that have conflicts,
    - so you can open them manually and resolve those conflicts.
  + Your file contains a section that looks something like this:

<<<<<<< HEAD:index.html

<div id="footer">contact : email.support@github.com</div>

=======

<div id="footer">

please contact us at support@github.com

</div>

>>>>>>> iss53:index.html

* + This means the version in HEAD
    - is the top part of that block (everything above the =======),
    - while the version in your iss53 branch looks like everything in the bottom part.
  + In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself.
  + For instance, you might resolve this conflict by replacing the entire block with this:

<div id="footer">

please contact us at email.support@github.com

</div>

* + This resolution has a little of each section, and the <<<<<<<, =======, and >>>>>>> lines have been fully removed.
  + After you’ve resolved each of these sections in each conflicted file, run ***git add*** on each file to mark it as resolved.
  + Staging the file marks it as resolved in Git.
  + If you want to use a ***graphical tool*** to resolve these issues, you can run git mergetool, which fires up an appropriate visual merge tool and walks you through the conflicts:

$ git mergetool

merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge vimdiff

Merging the files: index.html

Normal merge conflict for 'index.html':

{local}: modified

{remote}: modified

Hit return to start merge resolution tool (opendiff):

* + After you exit the merge tool,
    - Git asks you if the merge was successful.
    - If you tell the script that it was, it stages the file to mark it as resolved for you.
  + You can run git status again to verify that all conflicts have been resolved:

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: index.html

#

* + If you’re happy with that,
    - and you verify that everything that had conflicts has been staged,
    - you can type git commit to finalize the merge commit.
  + The commit message by default looks something like this:

Merge branch 'iss53'

Conflicts:

index.html

#

# It looks like you may be committing a MERGE.

# If this is not correct, please remove the file

# .git/MERGE\_HEAD

# and try again.

#

* + You can modify that message with details about how you resolved the merge if you think it would be helpful to others looking at this merge in the future —
    - why you did what you did, if it’s not obvious.

## Branch Management

* Now that you’ve created, merged, and deleted some branches,
  + let’s look at some branch-management tools that will come in handy when you begin using branches all the time.
* The git branch command does more than just create and delete branches.
  + If you run it with no arguments, you get a simple listing of your current branches:

$ git branch

iss53

\* master

testing

* + Notice the \* character that prefixes the master branch:
    - it indicates the branch that you currently have checked out.
    - This means that if you commit at this point, the master branch will be moved forward with your new work.
* To see the last commit on each branch, you can run git branch -v:

$ git branch -v

iss53 93b412c fix javascript issue

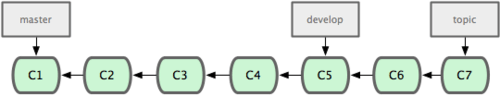
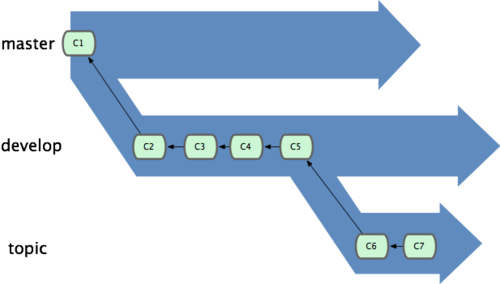
\* master 7a98805 Merge branch 'iss53'

testing 782fd34 add scott to the author list in the readmes

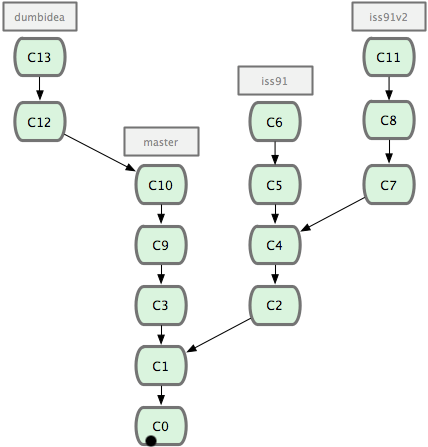
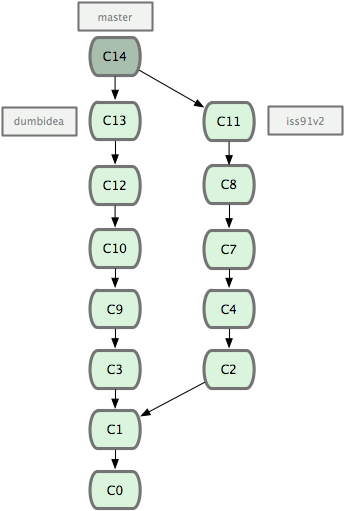
## Branching Workflows

* Now that you have the basics of branching and merging down, what can or should you do with them?
* In this section, we’ll cover some common workflows that this lightweight branching makes possible,
  + so you can decide if you would like to incorporate it into your own development cycle.

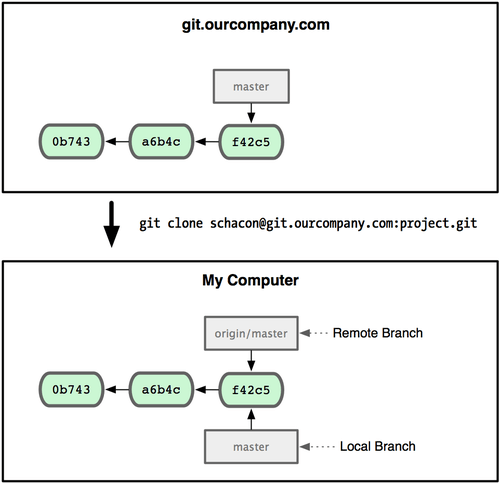
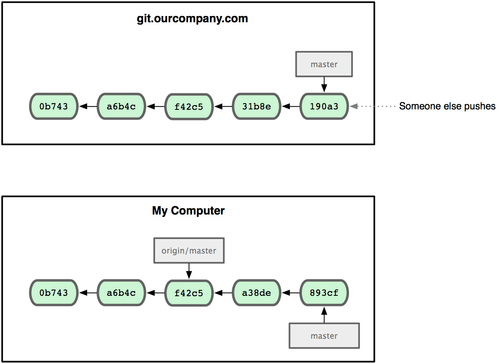
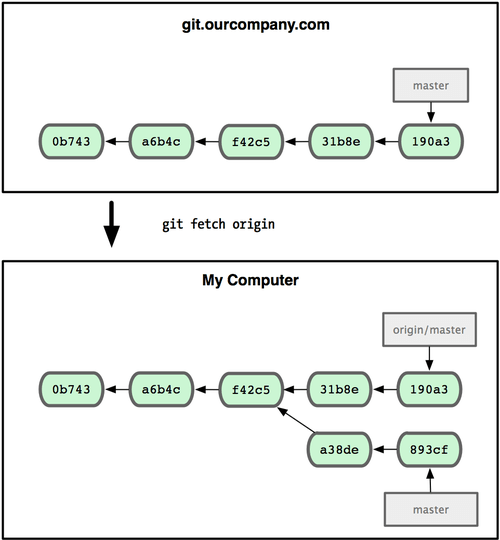
**Long-Running Branches**

* Because Git uses a simple three-way merge
  + merging from one branch into another multiple times over a long period is generally easy to do.
  + This means you can have several branches that are always open and that you use for different stages of your development cycle;
    - you can merge regularly from some of them into others.
* Many Git developers have a workflow that embraces this approach,
  + such as having only code that is entirely stable in their master branch
    - — possibly only code that has been or will be released.
  + They have another parallel branch named develop or next that they work from or use to test stability
    - — it isn’t necessarily always stable, but whenever it gets to a stable state, it can be merged into master.
    - It’s used to pull in topic branches (short-lived branches, like your earlier iss53 branch) when they’re ready,
      * to make sure they pass all the tests and don’t introduce bugs.
* In reality, we’re talking about pointers moving up the line of commits you’re making.
  + The stable branches are farther down the line in your commit history,
  + and the bleeding-edge branches are farther up the history
  + 
  + It’s generally easier to think about them as work silos, where sets of commits graduate to a more stable silo when they’re fully tested
  + 
* You can keep doing this for several levels of stability.
  + Some larger projects also have a proposed or pu (proposed updates) branch
    - that has integrated branches that may not be ready to go into the next or master branch.
  + The idea is that your branches are at various levels of stability;
    - when they reach a more stable level, they’re merged into the branch above them.
  + having multiple long-running branches isn’t necessary,
    - but it’s often helpful, especially when you’re dealing with very large or complex projects.

**Topic Branches**

* Topic branches are useful in projects of any size.
  + A topic branch is a short-lived branch that you create and use for a single particular feature or related work.
  + This is something you’ve likely never done with a VCS before because it’s generally too expensive to create and merge branches.
  + In Git it’s common to create, work on, merge, and delete branches several times a day.
* You saw this in the last section with the iss53 and hotfix branches you created.
  + You did a few commits on them and deleted them directly after merging them into your main branch.
  + This technique allows you to context-switch quickly and completely — because your work is separated into silos where all the changes in that branch have to do with that topic,
    - it’s easier to see what has happened during code review and such.
  + You can keep the changes there for minutes, days, or months, and merge them in when they’re ready,
    - regardless of the order in which they were created or worked on.
* Consider an example of doing some work (on master),
  + branching off for an issue (iss91),
  + working on it for a bit,
  + branching off the second branch to try another way of handling the same thing (iss91v2),
  + going back to your master branch and working there for a while,
  + and then branching off there to do some work that you’re not sure is a good idea (dumbidea branch).
  + Your commit history will look something like this:
    - 
  + Now, let’s say you decide you like the second solution to your issue best (iss91v2);
    - and you showed the dumbidea branch to your coworkers,
    - and it turns out to be genius.
    - You can throw away the original iss91 branch (losing commits C5 and C6) and merge in the other two.
    - Your history then looks like this:
      * 
  + It’s important to remember when you’re doing all this that these branches are completely local.
    - When you’re branching and merging, everything is being done only in your Git repository — no server communication is happening.

## Remote Branches

* Remote branches are references to the state of branches on your remote repositories.
  + They’re local branches that you can’t move;
    - they’re moved automatically whenever you do any network communication.
  + Remote branches act as bookmarks to remind you where the branches on your remote repositories were the last time you connected to them.
* They take the form (remote)/(branch)
  + e.g. If you wanted to see what the master branch on your origin remote looked like as of the last time you communicated with it,
    - you would check the origin/master branch.
  + e.g. If you were working on an issue with a partner and they pushed up an iss53 branch,
    - you might have your own local iss53 branch;
    - but the branch on the server would point to the commit at origin/iss53.
* Example:
  + Let’s say you have a Git server on your network at git.ourcompany.com.
  + If you clone from this,
    - Git automatically names it origin for you,
    - pulls down all its data,
    - creates a pointer to where its master branch is,
    - and names it origin/master locally;
    - and you can’t move it.
    - Git also gives you your own master branch starting at the same place as origin’s master branch,
    - so you have something to work from.
    - 
  + If you do some work on your local master branch,
    - and, in the meantime, someone else pushes to git.ourcompany.com and updates its master branch,
      * then your histories move forward differently.
  + Also, as long as you stay out of contact with your origin server,
    - your origin/master pointer doesn’t move.
    - 
  + To synchronize your work, you run a git fetch origin command.
    - This command looks up which server origin is (in this case, it’s git.ourcompany.com),
      * fetches any data from it that you don’t yet have,
      * and updates your local database, moving your origin/master pointer to its new, more up-to-date position.
    - 

**Pushing**

* When you want to share a branch with the world,
  + you need to push it up to a remote that you have write access to.
  + Your local branches aren’t automatically synchronized to the remotes you write to
    - — you have to explicitly push the branches you want to share.
  + That way, you can use private branches for work you don’t want to share,
    - and push up only the topic branches you want to collaborate on.
* If you have a branch named serverfix that you want to work on with others,
  + you can push it up the same way you pushed your first branch. Run git push (remote) (branch):

$ git push origin serverfix

Counting objects: 20, done.

Compressing objects: 100% (14/14), done.

Writing objects: 100% (15/15), 1.74 KiB, done.

Total 15 (delta 5), reused 0 (delta 0)

To git@github.com:schacon/simplegit.git

\* [new branch] serverfix -> serverfix

* The next time one of your collaborators fetches from the server,
  + they will get a reference to where the server’s version of serverfix is under the remote branch origin/serverfix:

$ git fetch origin

remote: Counting objects: 20, done.

remote: Compressing objects: 100% (14/14), done.

remote: Total 15 (delta 5), reused 0 (delta 0)

Unpacking objects: 100% (15/15), done.

From git@github.com:schacon/simplegit

\* [new branch] serverfix -> origin/serverfix

* It’s important to note that,
  + when you do a fetch that brings down new remote branches,
  + you don’t automatically have local, editable copies of them.
  + In other words, in this case,
    - you don’t have a new serverfix branch — you only have an origin/serverfix pointer that you can’t modify.
* To merge this work into your current working branch,
  + you can run git merge origin/serverfix.
  + If you want your own serverfix branch that you can work on, you can base it off your remote branch:

$ git checkout -b serverfix origin/serverfix

Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.

Switched to a new branch "serverfix"

* + This gives you a local branch that you can work on that starts where origin/serverfix is.

**Tracking Branches**

* Checking out a local branch from a remote branch automatically creates what is called a tracking branch.
  + Tracking branches are local branches that have a direct relationship to a remote branch.
  + If you’re on a tracking branch and type git push, Git automatically knows which server and branch to push to.
  + Also, running git pull while on one of these branches fetches all the remote references
    - and then automatically merges in the corresponding remote branch.
* When you clone a repository,
  + it generally automatically creates a master branch that tracks origin/master.
  + That’s why git push and git pull work out of the box with no other arguments.
  + However, you can set up other tracking branches if you wish
    - — ones that don’t track branches on origin and don’t track the master branch.
  + The simple case is the example you just saw,
    - running git checkout -b [branch] [remotename]/[branch].
    - If you have Git version 1.6.2 or later, you can also use the --track shorthand:

$ git checkout --track origin/serverfix

Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.

Switched to a new branch "serverfix"

* To set up a local branch with a different name than the remote branch,
  + you can easily use the first version with a different local branch name:

$ git checkout -b sf origin/serverfix

Branch sf set up to track remote branch refs/remotes/origin/serverfix.

Switched to a new branch "sf"

* + Now, your local branch sf will automatically push to and pull from origin/serverfix.

**Deleting a Remote Branch**

* Suppose you’re done with a remote branch
  + — say, you and your collaborators are finished with a feature and have merged it into your remote’s master branch.
  + You can delete a remote branch using the rather obtuse syntax git push [remotename] :[branch].
    - If you want to delete your serverfix branch from the server, you run the following:

$ git push origin :serverfix

To git@github.com:schacon/simplegit.git

- [deleted] serverfix