Git Magic =

Ben Lynn

Git Magic =

by Ben Lynn

Table of Contents

Chapter 1. Preface ==

Git (http://git.or.cz/) is a version control Swiss army knife. A reliable versatile multipurpose revision control tool whose extraordinary flexibility makes it tricky to learn, let alone master. I'm recording what I've figured out so far in these pages, because I initially had difficulty understanding the Git user manual (http://www.kernel.org/pub/software/scm/git/docs/user-manual.html).

As Arthur C. Clarke observed, any sufficiently advanced technology is indistinguishable from magic. This is a great way to approach Git: newbies can ignore its inner workings and view Git as a gizmo that can amaze friends and infuriate enemies with its wondrous abilities.

Rather than go into details, we provide rough instructions for particular effects. After repeated use, gradually you will understand how each trick works, and how to tailor the recipes for your needs.

Other Editions

- Single webpage (book.html): barebones HTML, with no CSS.
- PDF file (book.pdf): printer-friendly.

1.1. Thanks! ===

Kudos to Dustin Sallings, Alberto Bertogli, James Cameron, Douglas Livingstone, Michael Budde, Richard Albury, Tarmigan and Derek Mahar for suggestions and improvements. [If I've left you out, please tell me because I often forget to update this section.]

1.2. Links ===

I once listed some references, but it's too time-consuming to maintain. Besides, anyone can simply use a search engine (http://www.google.com/) to find Git tutorials (http://www.google.com/search?q=git+tutorial), guides (http://www.google.com/search?q=git+guide), and comparisons (http://www.google.com/search?q=git+comparison) with Subversion (http://www.google.com/search?q=git+subversion), Mercurial (http://www.google.com/search?q=git+mercurial), and other version control systems.

Free Git hosting

- http://repo.or.cz/ hosts free projects, including this guide (http://repo.or.cz/w/gitmagic.git).
- http://gitorious.org/ is another Git hosting site aimed at open-source projects.
- http://github.com/ hosts open-source projects for free, including this guide (http://github.com/blynn/gitmagic/tree/master), and private projects for a fee.

Chapter 2. Introduction ==

I'll use an analogy to introduce version control. See the Wikipedia entry on revision control (http://en.wikipedia.org/wiki/Revision_control) for a saner explanation.

2.1. Work is Play ===

I've played computer games almost all my life. In contrast, I only started using version control systems as an adult. I suspect I'm not alone, and comparing the two may make these concepts easier to explain and understand.

Think of editing your code or document, or whatever, as playing a game. Once you've made a lot of progress, you'd like to save. To do so, you click on the "Save" button in your trusty editor.

But this will overwrite the old version. It's like those old school games which only had one save slot: sure you could save, but you could never go back to an older state. Which was a shame, because your previous save might have been right at a exceptionally fun part of the game that you'd like to revisit one day. Or worse still, your current save is in an unwinnable state, and you have to start again.

2.2. Version Control ===

When editing, you can "Save As…" a different file, or copy the file somewhere first before saving if you want to savour old versions. Maybe compress them too to save space. This is a primitive and labour-intensive form of version control. Computer games improved on this long ago, many of them providing multiple automatically timestamped save slots.

Let's make the problem slightly tougher. Say you have a bunch of files that go together, such as source code for a project, or files for a website. Now if you want to keep an old version you have to archive a whole directory. Keeping many versions around by hand is inconvenient, and quickly becomes expensive.

With some computer games, a saved game really does consist of a directory full of files. These games hide this detail from the player and present a convenient interface to manage different versions of this directory.

Version control systems are no different. They all have nice interfaces to manage a directory of stuff. You can save the state of the directory every so often, and you can load any one of the saved states later on. Unlike most computer games, they're usually smart about conserving space. Typically, only a few files change between version to version, and not by much. Storing the differences instead of entire new copies saves room.

2.3. Distributed Control ===

Now imagine a very difficult computer game. So difficult to finish that many experienced gamers all over the world decide to team up and share their saved games to try to beat it. Speedruns are real-life examples: players specializing in different levels of the same game collaborate to produce amazing results.

How would you set up a system so they can get at each other's saves easily? And upload new ones?

In the old days, every project used centralized version control. A server somewhere held all the saved games. Nobody else did. Every player kept at most a few saved games on their machine. When a player wanted to make progress, they'd download the latest save from the main server, play a while, save and upload back to the server for everyone else to use.

What if a player wanted to get an older saved game for some reason? Maybe the current saved game is in an unwinnable state because somebody forgot to pick up an object back in level three, and they want to find the latest saved game where the game can still be completed. Or maybe they want to compare two older saved games to see how much work a particular player did.

There could be many reasons to want to see an older revision, but the outcome is the same. They have to ask the central server for that old saved game. The more saved games they want, the more they need to communicate.

The new generation of version control systems, of which Git is a member, are known as distributed systems, and can be thought of as a generalization of centralized systems. When players download from the main server they get every saved game, not just the latest one. It's as if they're mirroring the central server.

This initial cloning operation can be expensive, especially if there's a long history, but it pays off in the long run. One immediate benefit is that when an old save is desired for any reason, communication with the central server is unnecessary.

2.3.1. A Silly Superstition ====

A popular misconception is that distributed systems are ill-suited for projects requiring an official central repository. Nothing could be further from the truth. Photographing someone does not cause their soul to be stolen. Similarly, cloning the master repository does not diminish its importance.

A good first approximation is that anything a centralized version control system can do, a well-designed distributed system can do better. Network resources are simply costlier than local resources. While we shall later see there are drawbacks to a distributed approach, one is less likely to make erroneous comparisons with this rule of thumb.

A small project may only need a fraction of the features offered by such a system. But would you use Roman numerals when calculating with small numbers? Moreover, your project may grow beyond your original expectations. Using Git from the outset is like carrying a Swiss army knife even though you mostly use it to open bottles. On the day you desperately need a screwdriver you'll be glad you have more than a plain bottle-opener.

2.4. Merge Conflicts ===

An interesting problem arises in any type of revision control system. Suppose Alice and Bob have both independently downloaded the latest saved game. They both play the game a bit more and save. What if they both want to submit their new saves? Let's say Alice uploads first. Then when Bob tries to upload his save, the system realizes that his state does not chronologically follow Alice's state.

Let us revert to editing a text file. Our computer game analogy becomes too thinly stretched since most games never worry about this.

So suppose Alice has inserted a line at the beginning of a file, and Bob appends one at the end. They both upload their changes. Most systems will automatically deduce a reasonable course of action: accept and merge their changes, so both Alice's and Bob's edits are applied.

Now suppose both Alice and Bob have made distinct edits to the same line. Then it is impossible to resolve the conflict without human intervention. The second person to upload is informed of a merge conflict, and they must either choose one edit over another, or revise the line entirely.

More complex situations can arise. Version control systems handle the simpler cases themselves, and leave the difficult cases for humans. Usually their behaviour is configurable.

Chapter 3. Basic Tricks ==

Rather than diving into a sea of Git commands, use these elementary examples to get your feet wet. Despite their simplicity, each of them are useful.

3.1. Saving State ===

When I'm about to attempt something drastic I like to save the current state, so I can go back and try again should things go awry.

Take a snapshot of all files in the current directory with:

```
$ git init
$ git add .
$ git commit -m "My first backup"
```

The above sequence of commands should be memorized, or placed in a script, as they will be reused frequently.

Then if something goes wrong, run:

```
$ git reset --hard
```

to go back to where you were. To save the state again:

```
$ git commit -a -m "Another backup"
```

3.1.1. Add, Delete, Rename ====

The above will only keep track of the files that were present when you first ran **git add**. If you add new files or subdirectories, you'll have to tell Git:

```
$ git add NEWFILES...
```

Similarly, if you want Git to forget about certain files, maybe because you've deleted them

```
$ git rm OLDFILES...
```

Renaming a file is the same as removing the old name and adding the new name. There's also the shortcut **git mv** which has the same syntax as the **mv** command. For example:

```
$ git mv OLDFILE NEWFILE
```

3.2. Advanced Undo/Redo ===

Sometimes you just want to go back and forget about every change past a certain point because they're all wrong.

Then

```
$ git log
```

shows you a list of recent commits, and their SHA1 hashes. Next, type

```
$ git reset --hard SHA1_HASH
```

to restore the state to a given commit and erase all newer commits from the record permanently.

Other times you want to hop to an old state briefly. In this case, type:

```
$ git checkout SHA1_HASH
```

This takes you back in time, while preserving newer commits. However, like time travel in a science-fiction movie, if you now edit and commit, you will be in an alternate reality, because your actions are different to what they were the first time around.

This alternate reality is called a *'branch*', and we'll have more to say about this later. For now, just remember that

```
$ git checkout master
```

will take you back to the present.

Uncommitted changes travel in time with you when you run checkout.

To take the computer game analogy again:

- **`git reset --hard`**: load an old save and delete all saved games newer than the one just loaded.
- **`git checkout`**: load an old game, but if you play on, the game state will deviate from the newer saves you made the first time around. Any saved games you make now will end up in a separate branch representing the alternate reality you have entered. We deal with this later.

You can choose only to restore particular files and subdirectories by appending them after the command.

Don't like cutting and pasting hashes? Then use:

```
$ git checkout "@{10 minutes ago}" .
```

Other time specifications work too. For example, you can ask for the 5th-last saved state:

```
$ git checkout "@{5}" .
```

3.2.1. Reverting ====

In a court of law, events can be stricken from the record. Likewise, you can pick specific commits to undo.

```
$ git commit -a
$ git revert SHA1_HASH
```

will undo just the commit with the given hash. Running **git log** reveals the revert is recorded as a new commit.

3.3. Downloading Files ===

Get a copy of a project managed with Git by typing:

```
$ git clone git://server/path/to/files
```

For example, to get all the files I used to create this site:

```
$ git clone git://git.or.cz/gitmagic.git
```

We'll have much to say about the **clone** command soon.

3.4. The Bleeding Edge ===

If you've already downloaded a copy of a project using **git clone**, you can upgrade to the latest version with:

```
$ git pull
```

3.5. Instant Publishing ===

Let's say you've written a script you'd like to share with others. You could just tell them to download from your computer, but if they do so while you're improving the script or making experimental changes, they could wind up in trouble. Of course, this is why release cycles exist. Developers work on code, and when they feel it's suitable for others, they release the code.

To do this with Git, in the directory where your script resides:

```
$ git init
$ git add .
$ git commit -m "First release"
```

Then tell your users to run:

```
$ git clone your.computer:/path/to/script
```

to download your script. This assumes they have ssh access. If not, run **git daemon** and tell your users to instead run:

```
$ git clone git://your.computer/path/to/script
```

From now on, every time your script is ready for release, execute:

```
$ git commit -a -m "Next release"
```

and your users can upgrade their version by changing to the directory containing your script and typing:

```
$ git pull
```

Your users will never end up with a version of your script you don't want them to see. Obviously this trick works for anything, not just scripts.

3.6. What Have I Done? ===

Find out what changes you've made since the last commit with:

```
$ git diff
```

Or since yesterday:

```
$ git diff "@{yesterday}"
```

Or between a particular version and 2 versions ago:

```
$ git diff SHA1_HASH "@{2}"
```

Chapter 4. Cloning Around ==

In older version control systems, checkout is the standard operation to get files. You checkout a bunch of files in the requested saved state.

In Git and other distributed version control systems, cloning is the standard operation. To get files you create a clone of the entire repository. In other words, you practically create a mirror of the central server. Anything the main repository can do, you can do.

4.1. Sync Computers ===

This is the reason I first used Git. I can tolerate making tarballs or using **rsync** for backups and basic syncing. But sometimes I edit on my laptop, other times on my desktop, and the two may not have talked to each other in between.

Initialize a Git repository and commit your files as above on one machine. Then on the other:

```
$ git clone other.computer:/path/to/files
```

to create a second copy of the files and Git repository. From now on,

```
$ git commit -a
$ git pull other.computer:/path/to/files
```

will pull in the state of the files on the other computer into the one you're working on. If you've recently made conflicting edits in the same file, Git will let you know and you should commit again after resolving them.

4.2. Classic Source Control ===

Initialize a Git repository for your files:

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

On the central server, initialize an empty Git repository with some name, and start the Git daemon if necessary:

```
$ GIT_DIR=proj.git git init
$ git daemon --detach  # it might already be running
```

Some public hosts, such as repo.or.cz (http://repo.or.cz), will have a different method for setting up the initially empty Git repository, such as filling in a form on a webpage.

Push your project to the central server with:

```
$ git push git://central.server/path/to/proj.git HEAD
```

We're ready. To check out source, a developer types

```
$ git clone git://central.server/path/to/proj.git
```

After making changes, the code is checked in to the main server by:

```
$ git commit -a
$ git push
```

If the main server has been updated, the latest version needs to be checked out before the push. To sync to the latest version:

```
$ git commit -a
$ git pull
```

4.3. Forking a Project ===

Sick of the way a project is being run? Think you could do a better job? Then on your server:

```
$ git clone git://main.server/path/to/files
```

Next tell everyone about your fork of the project at your server.

At any later time, you can merge in the changes from the original project with:

```
$ git pull
```

4.4. Ultimate Backups ===

Want numerous tamper-proof geographically diverse redundant archives? If your project has many developers, don't do anything! Every clone of your code is effectively a backup. Not just of the current state of the project, but of your project's entire history. Thanks to cryptographic hashing, if anyone's clone becomes corrupted, it will be spotted as soon as they try to communicate with others.

If your project is not so popular, find as many servers as you can to host clones.

The truly paranoid should always write down the latest 20-byte SHA1 hash of the HEAD somewhere safe. It has to be safe, not private. For example, publishing it in a newspaper would work well, because it's hard for an attacker to alter every copy of a newspaper.

4.5. Light-Speed Multitask ===

Say you want to work on several features in parallel. Then after committing your project:

```
$ git clone . /some/new/directory
```

Git exploits hard links and file sharing as much as safely possible to create this clone, so it will be ready in a flash, and you can now work on two independent features simultaneously. For example, you can edit one clone while the other is compiling.

At any time, you can commit and pull changes from the other clone.

```
$ git pull /the/other/clone
```

4.6. Guerilla Version Control ===

Are you working on a project that uses some other version control system, and you sorely miss Git? Then initialize a Git repository in your working directory:

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

then clone it, at light speed:

```
$ git clone . /some/new/directory
```

Now go to the new directory and work here instead, using Git to your heart's content. Once in a while, you'll want to sync with everyone else, in which case go to the original directory, sync using the other version control system, and type:

```
$ git add .
$ git commit -m "Sync with everyone else"
```

Then go to the new directory and run:

```
$ git commit -a -m "Description of my changes"
$ git pull
```

The procedure for giving your changes to everyone else depends on the other version control system. The new directory contains the files with your changes. Run whatever commands of the other version control system are needed to upload them to the central repository.

Tip: If you want to interact with a Subversion repository, check out the **git svn** command, which can automate all this (and more) for you.

Chapter 5. Branch Wizardry ==

Instant branching and merging are the most lethal of Git's killer features.

Problem: External factors inevitably necessitate context switching. A severe bug manifests in the released version suddenly without warning, and must be fixed as soon as possible at all costs. The deadline for a certain feature is imminent. The guy who wrote a certain function is about to leave, so you should drop what you are doing and get him to help you understand it.

Interrupting your train of thought can be detrimental to your productivity, and the slower and less convenient it is to switch contexts, the greater the loss. With centralized version control we must download a fresh working copy from the central server. Distributed systems fare better, as we can clone the desired version locally.

But cloning still entails copying the whole working directory as well as the entire history up to the given point. Even though Git reduces the cost of this with file sharing and hard links, the project files themselves must be recreated in their entirety in the new working directory.

Solution: Git has a better tool for these situations that is much faster and more space-efficient than cloning: **git branch**.

With this magic word, the files in your directory suddenly shapeshift from one version to another. This transformation can do more than merely go back or forward in history. Your files can morph from the last release to the experimental version to the current development version to your friend's version and so on.

5.1. The Boss Key ===

Ever play one of those games where at the push of a button ("the boss key"), the screen would instantly display a spreadsheet or something? So if the boss walked in the office while you were playing the game you could quickly hide this fact?

In some directory:

```
$ echo "I'm smarter than my boss" > myfile.txt
$ git init
$ git add .
$ git commit -m "Initial commit"
```

We have created a Git repository that tracks one text file containing a certain message. Now type:

```
$ git checkout -b boss  # nothing seems to change after this
$ echo "My boss is smarter than me" > myfile.txt
```

```
$ git commit -a -m "Another commit"
```

It looks like we've just overwritten our file and committed it. But it's an illusion. Type:

```
$ git checkout master  # switch to original version of the file
```

and hey presto! The text file is restored. And if the boss decides to snoop around this directory, type:

```
$ git checkout boss  # switch to version suitable for boss' eyes
```

You can switch between the two versions of the file as much as you like, and commit to each independently.

5.2. Dirty Work ===

Say you're working on some feature, and for some reason, you need to go back to an old version and temporarily put in a few prints statements to see how something works. Then:

```
$ git commit -a
$ git checkout SHA1_HASH
```

Now you can add ugly temporary code all over the place. You can even commit these changes. When you're done,

```
$ git checkout master
```

to return to your original work. Observe that any uncommitted changes are carried over.

What if you wanted to save the temporary changes after all? Easy:

```
$ git checkout -b dirty
```

and commit before switching back to the master branch. Whenever you want to return to the dirty changes, simply type

```
$ git checkout dirty
```

We touched upon this command in an earlier chapter, when discussing loading old states. At last we can tell the whole story: the files change to the requested state, but we must leave the master branch. Any commits made from now on take your files down a different road, which can be named later.

In other words, after checking out an old state, Git automatically puts you in a new, unnamed branch, which can be named and saved with **git checkout -b**.

5.3. Quick Fixes ===

You're in the middle of something when you are told to drop everything and fix a newly discovered bug:

```
$ git commit -a
$ git checkout -b fixes SHA1_HASH
```

Then once you've fixed the bug:

```
$ git commit -a -m "Bug fixed"
$ git push  # to the central repository
$ git checkout master
```

and resume work on your original task.

5.4. Uninterrupted Workflow ===

Some projects require your code to be reviewed before you may submit it. To make life easier for those reviewing your code, if you have a big change to make you might break it into two or more parts, and get each part separately reviewed.

What if the second part cannot be written until the first part is approved and checked in? In many version control systems, you'd have to send the first part to the reviewers, and then wait until it has been approved before starting on the second part.

Actually that's not quite true, but in these systems editing Part II before Part I has been submitted involves a lot of suffering and hardship. In Git, branching and merging are painless (a technical term for fast and local). So after you've committed the first part and sent it for review:

```
$ git checkout -b part2
```

Next, code the second part of the big change without waiting for the first part to be accepted. When the first part is approved and submitted,

```
$ git checkout master
$ git merge part2
$ git branch -d part2  # don't need this branch anymore
```

and the second part of the change is ready to review.

But wait! What if it wasn't that simple? Say you made a mistake in the first part, which you have to correct before submitting. No problem! First, switch back to the master branch with

```
$ git checkout master
```

Fix the issue with the first part of the change and hope it gets approved. If not we simply repeat this step. You'll probably want to merge the fixed version of Part I into Part II as well:

```
$ git checkout part2
$ git merge master
```

Now it's the same as before. Once the first part has been approved and submitted:

```
$ git checkout master
$ git merge part2
$ git branch -d part2
```

and again, the second part is ready to be reviewed.

It's easy to extend this trick for any number of parts.

5.5. Reorganizing a Medley ===

Perhaps you like to work on all aspects of a project in the same branch. You want to keep works-in-progress to yourself and want others to see your commits only when they have been neatly organized. Start a couple of branches:

```
$ git checkout -b sanitized
$ git checkout -b medley
```

Next, work on anything: fix bugs, add features, add temporary code, and so forth, committing often along the way. Then:

```
$ git checkout sanitized
$ git cherry-pick SHA1_HASH
```

applies a given commit to the "sanitized" branch. With appropriate cherry-picks you can construct a branch that contains only permanent code, and has related commits grouped together.

5.6. Managing Branches ===

Type:

```
$ git branch
```

to list all the branches. There is always a branch named "master", and you start here by default. Some advocate leaving the "master" branch untouched and creating new branches for your own edits.

See **git help branch** for other branch operations.

5.7. Work How You Want ===

Applications such as Mozilla Firefox (http://www.mozilla.com/) allow you to open multiple tabs and multiple windows. Switching tabs gives you different content in the same window. Git branching is like tabs for your working directory. Continuing this analogy, Git cloning is like opening a new window. Being able to do both easily makes for a better user experience.

On a higher level, several Linux window managers allow you to have multiple desktops, which means you can instantly view a different state of the desktop. This is similar to branching in Git, whereas Git cloning would be like attaching another monitor where you can open more windows.

Yet another example is the **screen** (http://www.gnu.org/software/screen/) utility. This gem allows you to create, destroy and switch between multiple terminal sessions in the same terminal. Instead of opening new terminals (clone), you can use the same one if you run **screen** (branch). In fact, you can do a lot more with **screen** but that's a topic for another text.

Cloning, branching and merging are fast and local in Git, encouraging you to use the combination that best suits you. Git allows you to work exactly how you want.

5.7.1. Personal Experience ====

The equivalent of branching and cloning in centralized version control systems, when it exists, requires network communication. This precludes working offline, and means these systems need more expensive network infrastructure, especially as the number of users grows.

Most importantly, these operations will be slower to some degree, usually to the point where users won't bother using them unless absolutely necessary. And when they absolutely have to run slow commands, productivity suffers because of an interrupted work flow.

I experienced these phenomena first-hand. Git was the first version control system I used, and I grew accustomed to it, taking many features for granted. I did not know what centralized systems were like, and assumed they were similar. Later, I was forced to use one.

Often I have a flaky internet connection. This matters little with Git, but makes development unbearable with a centralized system.

Additionally, I found myself conditioned to avoid certain commands because of the latencies involved, which ultimately prevented me from following the work flow I wanted.

When I had to run a slow command, the interruption to my train of thought dealt a disproportionate amount of damage. While waiting for server communcation to complete, I'd do something else to pass the time, such as check email or write documentation. By the time I returned to the original task, the command had already finished long ago. I would then spend a long time trying to remember what I was doing.

Chapter 6. Git Grandmastery ==

This pretentiously named page is my dumping ground for uncategorized Git tricks.

6.1. Source Releases ===

For my projects, Git tracks exactly the files I'd like to archive and release to users. To create a tarball of the source code, I run:

```
$ git archive --format=tar --prefix=proj-1.2.3/ HEAD
```

6.2. Changelog Generation ===

It's good practice to keep a changelog (http://en.wikipedia.org/wiki/Changelog), and some projects even require it. If you've been committing frequently, which you should, generate a Changelog by typing:

```
$ git log > ChangeLog
```

6.3. Git Over SSH, HTTP ===

Suppose you have ssh access to your web server, but it does not have Git installed. Then download, compile and install Git in your account.

Create a repository in your web directory:

```
$ GIT_DIR=proj.git git init
```

and in the "proj.git" directory, run

```
$ git --bare update-server-info
$ chmod a+x hooks/post-update
```

From your computer, push via ssh:

```
$ git push web.server:/path/to/proj.git master
```

and others get your project via:

```
$ git clone http://web.server/proj.git
```

6.4. Commit What Changed ===

Telling Git when you've added, deleted and renamed files gets tedious. Instead, you can type:

```
$ git add .
$ git add -u
```

Git will look at the files in the current directory and work out the details by itself. Instead of the second add command, run `git commit -a` if you also intend to commit at this time.

You can perform the above in a single pass with:

```
$ git ls-files -d -m -o -z | xargs -0 git update-index --add --remove
```

The **-z** and **-0** options prevent ill side-effects from filenames containing strange characters. Note this command adds ignored files. You may want to use the `-x` or `-X` option.

6.5. I Stand Corrected ===

Did you just commit, but wish you had typed a different message? Realized you forgot to add a file? Then:

```
$ git commit --amend
```

can help you out.

Since this changes the history, only do this if you have yet to push your changes, otherwise your tree will diverge from other trees. Of course, if you control all the other trees too, then there is no problem since you can overwrite them.

6.6. . . . And Then Some ===

Let's suppose the previous problem is ten times worse. After a lengthy session you've made a bunch of commits. But you're not quite happy with the way they're organized, and some of those commit messages could use rewording. This is quite likely if you've been saving early and saving often. Then type

```
$ git rebase -i HEAD~10
```

and the last 10 commits will appear in your favourite $EDITOR. A sample excerpt:

```
pick 5c6eb73 Added repo.or.cz link
```

```
pick a311a64 Reordered analogies in "Work How You Want"
pick 100834f Added push target to Makefile
```

Then:

- Remove commits by deleting lines.
- Reorder commits by reordering lines.
- Replace "pick" with "edit" to mark a commit for amending.
- Replace "pick" with "squash" to merge a commit with the previous one.

Next run **git commit --amend** if you marked a commit for editing. Otherwise, run:

```
$ git rebase --continue
```

Again, only do this if no one else has a clone of your tree.

6.7. Local Changes Last ===

You're working on an active project. You make some local commits over time, and then you sync with the official tree with a merge. This cycle repeats itself a few times before you're ready to push to the central tree.

But now the history in your local Git clone is a messy jumble of your changes and the official changes. You'd prefer to see all your changes in one contiguous section, and after all the official changes.

This is a job for **git rebase** as described above. In many cases you can use the **--onto** flag and avoid interaction.

Also see the manpage for other amazing uses of this command, which really deserves a chapter of its own. You can split commits. You can even rearrange branches of a tree!

6.8. My Commit Is Too Big! ===

Have you neglected to commit for too long? Been coding furiously and forgotten about source control until now? Made a series of unrelated changes, because that's your style?

No worries, use **git add -i** or **git commit -i** to interactively choose which edits should belong to the next commit.

6.9. Don’t Lose Your HEAD ===

The HEAD tag is like a cursor that normally points at the latest commit, advancing with each new commit. Some Git commands let you move it. For example:

```
$ git reset HEAD~3
```

will move the HEAD three commits backwards in time. Thus all Git commands now act as if you hadn’t made those last three commits, while your files remain in the present. See the git reset man page for some applications.

But how can you go back to the future? The past commits do not know anything of the future.

If you have the SHA1 of the original HEAD then:

```
$ git reset SHA1
```

But suppose you never took it down? Don’t worry, for commands like these, Git saves the original HEAD as a tag called ORIG_HEAD, and you can return safe and sound with:

```
$ git reset ORIG_HEAD
```

6.10. HEAD-hunting ===

Perhaps ORIG_HEAD isn’t enough. Perhaps you’ve just realized you made a monumental mistake last month and you need to go back to an ancient commit in a long-forgotten branch.

It’s hard to lose Git commits permanently, even after deleting branches. As long as you never run **git gc --prune**, your commits are preserved forever and can be restored at any time.

The trouble is finding the appropriate hash. You could look at all the hash values in `.git/objects` and use trial and error to find the one you want. But there’s a much easier way.

Git records every hash of a commit it computes in `.git/logs`. The subdirectory `refs` contains the history of all activity on all branches, while the file `HEAD` shows every hash value it has ever taken. The latter can be used to find hashes of commits on branches that have been accidentally lopped off.

The reflog command provides a friendly interface to these log files. Try

```
$ git reflog
```

and see its manpage for more information.

Eventually, you may want to run **git gc --prune** to recover space. Be aware that doing so prevents you from recovering lost HEADs.

6.11. Making History ===

Want to migrate a project to Git? If it's managed with one of the more well-known systems, then chances are someone has already written a script to export the whole history to Git.

Otherwise, take a look at **git fast-import**. This command takes text input in a specific format and creates Git history from scratch. Typically a script is cobbled together and run once to feed this command, migrating the project in a single shot.

As an example, paste the following listing into temporary file, such as `/tmp/history`:

```
commit refs/heads/master
committer Alice <alice@example.com> Thu, 01 Jan 1970 00:00:00 +0000
data <<EOT
Initial commit.
EOT

M 100644 inline hello.c
data <<EOT
#include <stdio.h>

int main() {
  printf("Hello, world!\n");
  return 0;
}
EOT


commit refs/heads/master
committer Bob <bob@example.com> Tue, 14 Mar 2000 01:59:26 -0800
data <<EOT
Replace printf() with write().
EOT

M 100644 inline hello.c
data <<EOT
#include <unistd.h>

int main() {
  write(1, "Hello, world!\n", 14);
  return 0;
}
```

```
EOT
```

Then create a Git repository from this temporary file by typing:

```
$ mkdir project; cd project; git init
$ git fast-import < /tmp/history
```

You can checkout the latest version of the project with:

```
$ git checkout master .
```

6.12. Building On Git ===

In true UNIX fashion, Git's design allows it to be easily used as a low-level component of other programs. There are GUI interfaces, web interfaces, alternative command-line interfaces, and perhaps soon you will have a script or two of your own that calls Git.

One easy trick is to use built-in git aliases shorten your most frequently used commands:

```
$ git config --global alias.co checkout
$ git config --global --get-regexp alias  # display current aliases
alias.co checkout
$ git co foo                              # same as 'git checkout foo'
```

Another is to print the current branch in the prompt, or window title. Invoking

```
$ git symbolic-ref HEAD
```

shows the current branch name. In practice, you most likely want to remove the "refs/heads/" and ignore errors:

```
$ git symbolic-ref HEAD 2> /dev/null | cut -b 12-
```

See the Git homepage (http://git.or.cz/) for more examples.

6.13. Daring Stunts ===

Recent versions of Git make it difficult for the user to accidentally destroy data. This is perhaps the most compelling reason to upgrade.

Nonetheless, there are times you truly want to destroy data. We show how to override the safeguards for common commands. Only use them if you know what you are doing.

Checkout: If you have uncommitted changes, a plain checkout fails. To destroy your changes, and checkout a given commit anyway, use the force flag:

```
$ git checkout -f COMMIT
```

On the other hand, if you specify particular paths for checkout, then there are no safety checks. The supplied paths are quietly overwritten. Take care if you use checkout in this manner.

Reset: Reset also fails in the presence of uncommitted changes. To force it through, run:

```
$ git reset --hard [COMMIT]
```

Branch: Deleting branches fails if this causes changes to be lost. To force a deletion, type:

```
$ git branch -D BRANCH  # instead of -d
```

Similarly, attempting to overwrite a branch via a move fails if data loss would ensue. To force a branch move, type:

```
$ git branch -M [SOURCE] TARGET  # instead of -m
```

Unlike checkout and reset, the destruction is deferred. The changes are still stored in the .git subdirectory, and can be retrieved by recovering the appropriate hash from `.git/logs` (see "HEAD-hunting" above). The data is only deleted the next time garbage is collected.

Chapter 7. Secrets Revealed ==

We take a peek under the hood and explain how Git performs its miracles. I will skimp over details. For in-depth descriptions refer to the user manual (http://www.kernel.org/pub/software/scm/git/docs/user-manual.html).

7.1. Invisibility ===

How can Git be so unobtrusive? Aside from occasional commits and merges, you can work as if you were unaware that version control exists. That is, until you need it, and that's when you're glad Git was watching over you the whole time.

Other version control systems don't let you forget about them. Permissions of files may be read-only unless you explicitly tell the server which files you intend to edit. The central server might be keeping track of who's checked out which code, and when. When the network goes down, you'll soon suffer. Developers constantly struggle with virtual red tape and bureaucracy.

The secret is the `.git` directory in your working directory. Git keeps the history of your project here. The initial "." stops it showing up in `ls` listings. Except when you're pushing and pulling changes, all version control operations operate within this directory.

You have total control over the fate of your files because Git doesn't care what you do to them. Git can easily recreate a saved state from `.git` at any time.

7.2. Integrity ===

Most people associate cryptography with keeping information secret, but another equally important goal is keeping information safe. Proper use of cryptographic hash functions can prevent accidental or malicious data corruption.

A SHA1 hash can be thought of as a unique 160-bit ID number for every string of bytes you'll encounter in your life. Actually more than that: every string of bytes that any human will ever use over many lifetimes. The hash of the whole contents of a file can be viewed as a unique ID number for that file.

An important observation is that a SHA1 hash is itself a string of bytes, so we can hash strings of bytes containing other hashes.

Roughly speaking, all files handled by Git are referred to by their unique ID, not by their filename. All data resides in files in the ".git/objects" subdirectory, where you won't find any normal filenames. The

contents of files are strings of bytes we call *'blobs*' and they are divorced from their filenames.

The filenames are recorded somewhere though. They live in *'tree*' objects, which are lists of filenames along with the IDs of their contents. Since the tree itself is a string of bytes, it too has a unique ID, which is how it is stored in the ".git/objects" subdirectory. Trees can appear on the lists of other trees, hence a directory tree and all the files within may be represented by trees and blobs.

Lastly, a *'commit*' contains a message, a few tree IDs and information on how they are related to each other. A commit is also a string of bytes, hence it too has a unique ID.

You can see for yourself: take any hash you see in the `.git/objects` directory, and type

```
$ git cat-file -p SHA1_HASH
```

Now suppose somebody tries to rewrite history and attempts to change the contents of a file in an ancient version. Then the ID of the file will change since it's now a different string of bytes. This changes the ID of any tree object referencing this file, which in turn changes the ID of all commit objects involving this tree. The corruption in the bad repository is exposed when everyone realizes all the commits since the mutilated file have the wrong IDs.

I've ignored details such as file permissions and signatures. But in short, so long as the 20 bytes representing the last commit are safe, it's impossible to tamper with a Git repository.

7.3. Intelligence ===

How does Git know you renamed a file, even though you never mentioned the fact explicitly? Sure, you may have run **git mv**, but that is exactly the same as a **git rm** followed by a **git add**.

Git heuristically ferrets out renames and copies between successive versions. In fact, it can detect chunks of code being moved or copied around between files! Though it cannot cover all cases, it does a decent job, and this feature is always improving. If it fails to work for you, try options enabling more expensive copy detection, and consider upgrading.

7.4. Indexing ===

For every tracked file, Git records information such as its size, creation time and last modification time in a file known as the **index**. To determine whether a file has changed, Git compares its current stats with that held the index. If they match, then Git can skip reading the file again.

Since stat calls are vastly cheaper than reading file contents, if you only edit a few files, Git can update its state in almost no time.

7.5. Bare Repositories ===

You may have been wondering what format those online Git repositories use. They're plain Git repositories, just like your `.git` directory, except they've got names like `proj.git`, and they have no working directory associated with them.

Most Git commands expect the Git index to live in `.git`, and will fail on these bare repositories. Fix this by setting the `GIT_DIR` environment variable to the path of the bare repository, or running Git within the directory itself with the `--bare` option.

7.6. Git's Origins ===

This Linux Kernel Mailing List post (http://lkml.org/lkml/2005/4/6/121) describes the chain of events that led to Git. The entire thread is a fascinating archaeological site for Git historians.

Chapter 8. Git Shortcomings ==

There are some Git issues I've swept under the carpet. Some can be handled easily with scripts and hooks, some require reorganizing or redefining the project, and for the few remaining annoyances, one will just have to wait. Or better yet, pitch in and help!

I've been playing around with some version control system ideas, and wrote an experimental system based on Git (/~blynn/gg/), which addresses some of these problems.

8.1. Microsoft Windows ===

Git on Microsoft Windows can be cumbersome:

- Cygwin (http://cygwin.com/), a Linux-like environment for Windows, contains a Windows port of Git (http://cygwin.com/packages/git/).
- Git on MSys (http://code.google.com/p/msysgit/) is an alternative requiring minimal runtime support, though a few of the commands need some work.

8.2. Unrelated Files ===

If your project is very large and contains many unrelated files that are constantly being changed, Git may be disadvantaged more than other systems because single files are not tracked. Git tracks changes to the whole project, which is usually beneficial.

A solution is to break up your project into pieces, each consisting of related files. Use **git submodule** if you still want to keep everything in a single repository.

8.3. Who's Editing What? ===

Some version control systems force you to explicitly tag a file before editing. While this is especially annoying when this involves talking to a central server, it does have two benefits:

1. Diffs are quick because only the tagged files need be examined.
2. When a central server stores the tags, one can discover who else is working on the file.

With appropriate scripting, you can achieve the same with Git. This requires cooperation from the programmer, who should execute particular scripts when editing a file.

8.4. File History ===

Since Git records project-wide changes, reconstructing the history of a single file requires more work than in version control systems that track individual files.

The penalty is typically slight, and well worth having as other operations are incredibly efficient. For example, `git checkout` is faster than `cp -a`, and project-wide deltas compress better than collections of file-based deltas.

8.5. Initial Clone ===

Creating a clone is more expensive than checking out code in other version control systems when there is a lengthy history.

The initial cost is worth paying in the long run, as most future operations will then be fast and offline. However, in some situations, it may be preferable to create a shallow clone with the `--depth` option. This is much faster, but the resulting clone has reduced functionality.

8.6. Volatile Projects ===

Git was written to be fast with respect to the size of the changes. Humans make small edits from version to version. A one-liner bugfix here, a new feature there, emended comments, and so forth. But if your files are radically different in successive revisions, then on each commit, your history necessarily grows by the size of your whole project.

There is nothing any version control system can do about this, but standard Git users will suffer more since normally histories are cloned.

The reasons why the changes are so great should be examined. Perhaps file formats should be changed. Minor edits should only cause minor changes to at most a few files.

Or perhaps a database or backup/archival solution is what is actually being sought, not a version control system. For example, version control may be ill-suited for managing photos periodically taken from a webcam.

If the files really must be constantly morphing and they really must be versioned, a possibility is to use Git in a centralized fashion. One can create shallow clones, which checks out little or no history of the project. Of course, many Git tools will be unavailable, and fixes must be submitted as patches. This is probably fine as it's unclear why anyone would want the history of wildly unstable files.

Another example is a project depending on firmware, which takes the form of a huge binary file. The history of the firmware is uninteresting to users, and updates compress poorly, so firmware revisions would unnecessarily blow up the size of the repository.

In this case, the source code should be stored in a Git repository, and the binary file should be kept separately. To make life easier, one could distribute a script that uses Git to clone the code, and rsync or a Git shallow clone for the firmware.

8.7. Global Counter ===

Some centralized version control systems maintain a positive integer that increases when a new commit is accepted. Git refers to changes by their hash, which is better in many circumstances.

But some people like having this integer around. Luckily, it's easy to write scripts so that with every update, the central Git repository increments an integer, perhaps in a tag, and associates it with the hash of the latest commit.

Every clone could maintain such a counter, but this would probably be useless, since only the central repository and its counter matters to everyone.

8.8. Empty Subdirectories ===

Empty subdirectories cannot be tracked. Create dummy files to work around this problem.

The current implementation of Git, rather than its design, is to blame for this drawback. With luck, once Git gains more traction, more users will clamour for this feature and it will be implemented.

8.9. Initial Commit ===

A stereotypical computer scientist counts from 0, rather than 1. Unfortunately, with respect to commits, git does not adhere to this convention. Many commands are unfriendly before the initial commit. Additionally, some corner cases must be handled specially, such as rebasing a branch with a different initial commit.

Git would benefit from defining the zero commit: as soon as a repository is constructed, HEAD would be set to the string consisting of 20 zero bytes. This special commit represents an empty tree, with no parent, at some time predating all Git repositories.

Then running git log, for example, would inform the user that no commits have been made yet, instead of exiting with a fatal error. Similarly for other tools.

Every initial commit is implicitly a descendant of this zero commit. For example, rebasing an unrelated branch would cause the whole branch to be grafted on to the target. Currently, all but the initial commit is applied, resulting in a merge conflict. One workaround is to use `git checkout` followed by `git commit -C` on the initial commit, then rebase the rest.

There are worse cases unfortunately. If several branches with different initial commits are merged together, then rebasing the result requires substantial manual intervention.