Learn Git The Hard Way

Chapter 1. Core Git

This git course has been written to help git users get a clearer understanding of source control and git concepts. After following it they will be able to:

- use git to contribute to projects
- understand how to troubleshoot git when it gets confusing
- not be fazed when people talk about things like rebasing, bisecting, remotes, merges

It assumes:

- No prior knowledge of source control
- No prior knowledge of git

It aims to give users:

- A hands-on, practical understanding of git
- Enough information to understand what is going on as they go deeper into git
- A familiarity with advanced git usage

1.1. Introduction

This section covers:

- source control
- git vs trad SCs
- the four phases of the git lifecycle
- benefits of git
- git config

If you’re not familiar with source control, it solves a simple problem: how do we keep track of changes in our codebase? It’s a database of files and the histories of their states.

I’m old enough to remember a time when people complained about using source control at all!

Traditional source control tools such as CVS and SVN were centralised. You communicated with a server which maintained the state of the system. This could mean several things:

- The source control database could get very big
- The history could get very messy
- Managing your 'checkouts' of code could get complicated and painful

These code databases are known as 'repositories'.

In the old world, if you checked out source code, that was a copy of some code that was 'inferior' in status to the centralised version.

As far as the user was concerned, code was in one of two states:

- Local changes ('dirty')
- Committed - pushed to server

My local changes could not be shared with anyone else until

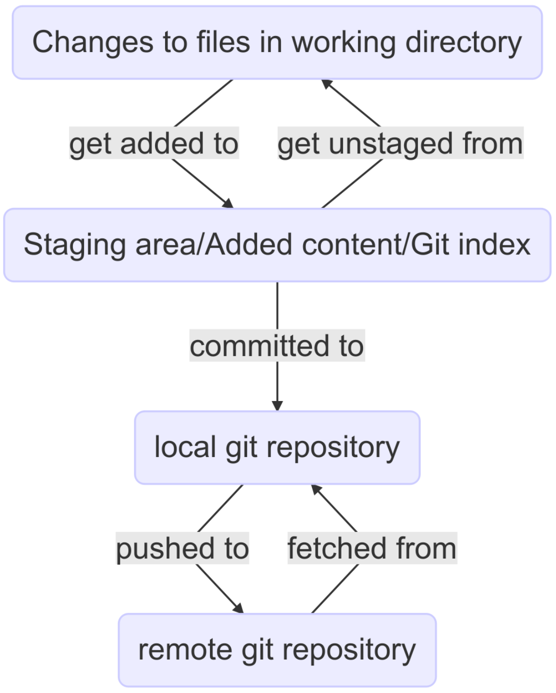
Git, by contrast, is fundamentally distributed. Each git repository is a full copy of each other git repositories.

ALL GIT REPOSITORIES ARE EQUAL.

Remember, git was created so people could work on the linux kernel across the globe, and offline. So there is no concept of a central server that holds the golden source.

1.1.1. The Four Phases of Git Content

In the git world you have four phases your code can go through:



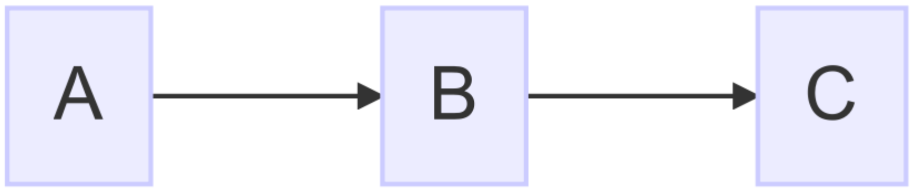
Understanding these 4) stages are key to understanding git.

If this seems over-complicated now, it won't as you grow to know and love git. If you've ever been confused by git, it's likely because these stages were not understood.

1.1.2. Branches

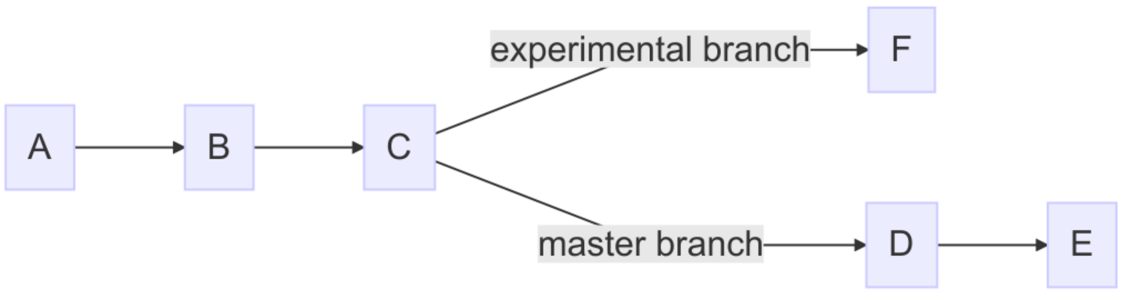
In case you've not looked at a SC tool before, a branch is a core concept.

A series of changes to a repository might look like this:



Change A is made, then B, then C. This might be informally called the 'main line'.

But let’s say someone wants to make an experimental change but not affect the 'main line'. Then they might 'branch' the code at point C:



That way users can choose to get a view of the source on the 'main line' branch or the 'experimental' one.

1.1.3. What About GitHub?

In practice, some repositories are more equal than others (eg GitHub). This is a matter of convention. I can use a GitHub repo as a secondary repo for my workflow - it’s up to me (indeed I do this for some of my private repos).

This is why every GitHub outage causes a flurry of smart-alec comments about git being decentralised.

Keeping your local repo sync’d with others is one of the challenges of git, but the first step to mastering git is understanding this equality of repos.

1.1.4. Other differences with git vs traditional VCSes

- History is more malleable.

You can change the history in your own copy of the repo and others' (if you have the appropriate permission).

- Branching is cheap

In CVS it’s very slow to branch a (O(n) to number of files). In git it’s an O(1) step. This makes experimentation much easier. Branch deletion is a common and cheap operation.

- Commits are across the whole poject

In contrast to other source control tools, changes are made across the whole project, not per file.

- No version numbers

Git does not automatically number versions of files/changes. It instead assigns a hash (effectively random) to the change.

- File renaming

You can rename files in git and the content will be tracked. This is a massive win compared to CVS.

1.1.5. Assumptions

At this point I assume you have git installed, and that you have set your details up as per the below. Replace with your email address and username:

```
$ git config --global user.email "you@example.com" && git config --global user.name
"Your Name"
```

1.1.6. What we learned

- setting up git
- what git is - the four stages
- differences to other SC systems

1.1.7. Exercises

1) Install git and set up your config. Set up user.email and user.name using the --global flag. 2) Find out where the config is stored. 3) Research the other config items that are in the file and some of those that are not.

1.2. Git basics

This section covers:

- git init
- the .git folder

- git log
- git status
- git add
- git commit
- git diff

This section is important because these are the basic tools you will most often use with git.

To initialise a git repository, run 'git init' from within the root folder of the source you want to manage.

```
$ rm -rf 1.2.1
$ mkdir 1.2.1
$ cd 1.2.1
$ git init
```

This initialises a database in the folder '.git' locally. Your repository is entirely stored within this .git folder. There are no other files elsewhere on your filesystem you need to be concerned about. (There are config files for git, but these are global to the host. We can ignore them for now.)

```
$ cd .git
$ ls
config
description
HEAD
hooks
info
objects
refs
$ cd ..
```

It’s not part of the scope of this course to go into detail about the git internals files seen here.

What is worth being aware of here are:

- the 'HEAD' file
- config

1.2.1. HEAD

The HEAD is key - it points to the current branch you are 'on'.

If we look at the file, we see it points to the refs/heads/master.

This is an internal representation of the default 'master' branch.

```
$ cat HEAD
ref: refs/heads/master
```

1.2.2. config

'config' stores information about your repository's local configuration, eg what branches and remote repositories your repository is aware of. It's a plain text file:

```
$ cat config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
    ignorecase = true
    precomposeunicode = true
```

1.2.3. git log

If we want to look at the history of this repository, you can run the git log command:

```
$ cd ..
$ git log
fatal: bad default revision 'HEAD'
```

We have a problem! This repository has no history to look at.

Git has followed the 'HEAD' pointer to the refs/heads/master entry and found nothing there! And indeed there is nothing there:

```
$ ls .git/refs/heads/master
ls: .git/refs/heads/master: No such file or directory
```

You need to create a history.

1.2.4. git status

As is often the case, git status is your friend:

```
$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

It tells you where the HEAD is pointed at (the non-existent master branch), and that there is 'nothing to commit'.

Create a file and check status again:

```
$ touch mycode.py
$ git status
On branch master

Initial commit

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

    mycode.c
```

We are now advised that we have an 'untracked' file. Git has detected that it exists but the repository is not 'aware' of it.

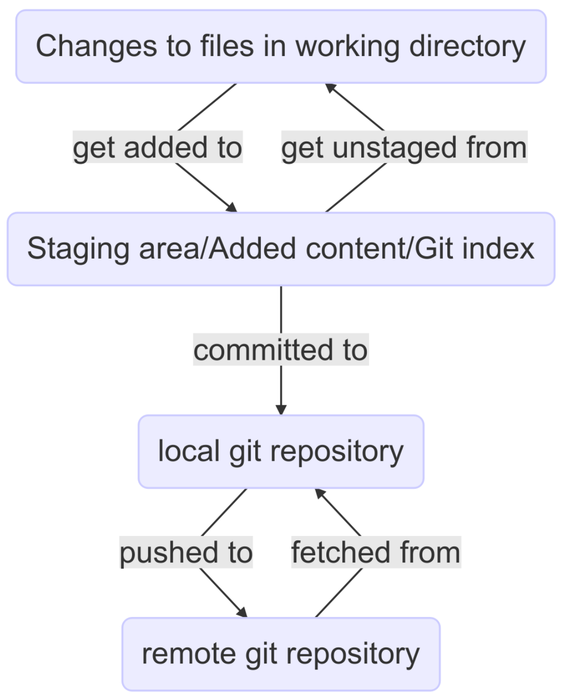
Make git aware of it by adding it to the repository.

1.2.5. git add

```
$ git add mycode.py
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   mycode.py
```

We have added a file to the index ready to be committed to the repository.

Remember the four stages we talked about before:



We create our file (1- local changes), then added/staged it to the index (2) and then committed to the local repository.

Still we have no history!

```
$ git log
fatal: bad default revision 'HEAD'
```

So we need to commit it to the repository to get a history.

1.2.6. git commit

```
$ git commit
$ git log
commit e5fb099e952e8754b54f9b99be93d62e3fce0fca
Author: ianmiell <ian.miell@gmail.com>
Date:   Tue Apr 26 07:46:58 2016 +0100

    Some message
```

Now that git is aware of this file you can make a change to it and show how the local change looks using git diff.

1.2.7. git diff

```
$ vi mycode.py
$ git diff
```

Again, you can see what's going on by looking at the status. You can commit changes to files and add at the same time by doing 'commit -a'

```
$ git status
$ git commit -a
$ git status
```

git log now shows the history of the file:

```
$ git log
```

1.2.8. What you learned

- git init
- the .git folder
- HEAD - a pointer to where in the history we are
- git log
- git status
- git add
- git commit
- git diff

1.2.9. Exercises

1) Create a git repo

2) Add and commit a file to the repo

3) Commit a few more changes, and then run git log to view the history

1.3. Cloning a repo

- git clone
- git reset

1.3.1. Clone

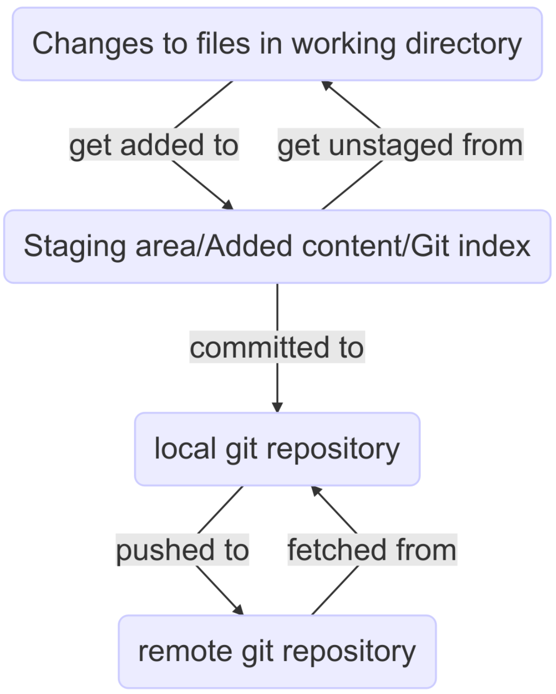
In this section we're going to play with the contents of our repository by deleting the content and seeing what our options are to recover from the repository.

```
$ git clone https://github.com/ianmiell/shutit
$ cd shutit
$ ls .git
```

There’s .git, just as before.

1.3.2. Accidental Deletion

Recall the 4 stages of data in a git repo:



```
$ git log                          # default history of this repo
$ git log --oneline                # more concise history of this repo
$ git log --oneline --graph        # graphical view of the history of this repo
$ cd ..                            # exit this repo's root folder
$ git clone shutit cloned_shutit   # clone the repository
$ cd cloned_shutit                 # enter the repository
$ ls .git                          # we have a copy of the repository's history
$ rm -rf *                         # delete all the files!
$ ls .git                          # The .git folder is still there
```

We have cloned the repository, and 'accidentally' deleted all the files.

1.3.3. git reset

You can use 'git reset' to recover the state of the

```
$ git status       # reports that we have deleted files in working tree/directory
$ git add .        # added to staging/index area
$ git status       # reports that . Note there's a helpful message about resetting
now! Let's explore that.
$ git reset --mixed # --mixed is the default. out of staging/index area, but still
deleted in the working directory!
$ git status       # we are back to 'deleted in the working directory' with a message
about being ready to add
$ rm -rf *         # delete all the files again
$ git add .        # added to staging/index area ready to commit again
$ git reset --hard  # does a re-check out of the whole repository, discarding working
directory and changes to the index
$ git status       # we now have a consistent state between 1 (local changes) and 3
(committed)
```

1.3.4. What you learned

- git clone
- git reset

1.3.5. Exercises

1) Check out a git repo from either your company repository or github

2) Browse the git log for that repo

3) Look at the man page for git log and explore the options. Don’t worry about understanding everything in there, but play with the options and try to work out what is going on.

1.4. Git branching

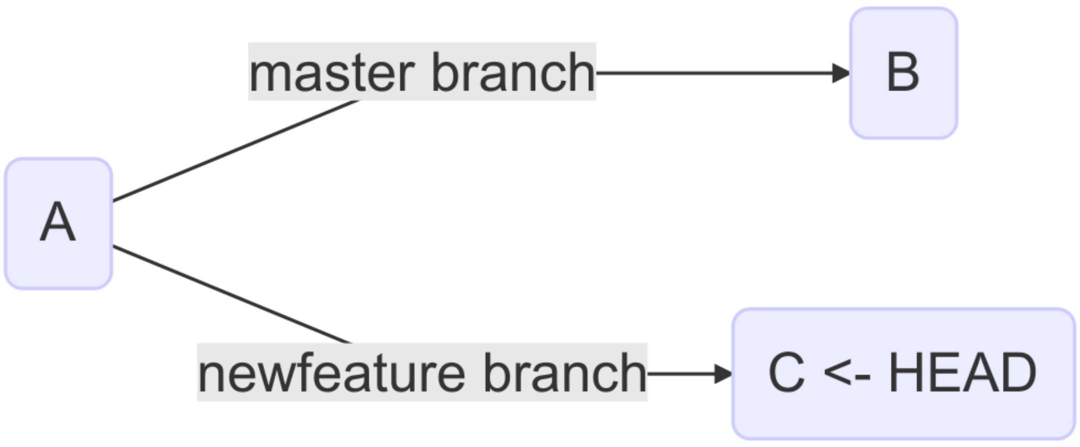
In this section you will learn about:

- git branch
- git checkout

In the next section of code we will create a git repository with a single file. This file will have separate changes made on two branches - master and newfeature.

```
$ rm -rf 1.4.1
$ mkdir 1.4.1
$ cd 1.4.1
$ git init
$ echo newfile > file1
$ git add file1
$ git commit -am 'new file1'
$ git status
$ git branch newfeature                      # Create the 'newfeature' branch
$ git status                                 # You are still on the master branch!
$ git branch                                 # git branch shows the branches in your
repository
$ echo Line_master1 >> file1                 # add Line_master1
$ git commit -am 'master change'             # add, commit and message
$ git log --decorate --graph --oneline       # graphical view of this branch
$ git log --decorate --graph --oneline --all # graphical view of all branches
$ git checkout newfeature                    # Check out the newfeature branch
$ cat file1                                  # This has been checked out at the
'branch point'
$ echo Line_feature1 >> file1                # add Line_feature1
$ git commit -am 'feature change'            # add, commit and message
$ git log --decorate --graph --oneline --all # graphical view of all branches
$ git checkout master                        # checkout the master branch
$ cat file1                                  # The feature change is not there
```

This is the final state of the commit tree.



which reflects the output of the last 'git log' command.

Note that the HEAD (and branch) moves forward with each commit.

The head is where git is pointed at right now, the branch is where that branch reference is pointed to.

HEAD can be moved to an arbitrary point (git checkout does this)

```
$ git log
$ git checkout
$ git status
```

TODO: more here

Detached head means you are not associated with a branch.

It 'feels wrong' to be on a detached head because you have no pointer to a branch to reference.

SO: A 'branch' is just a pointer on a line of changes HEAD is just your 'current' branch

It’s worth also pointing out here that apart from its default status, there is nothing special about the 'master' branch. It’s just a name. Your principal branch might be called 'live', 'alice', 'pristine', or whatever you like.

1.4.1. What you learned

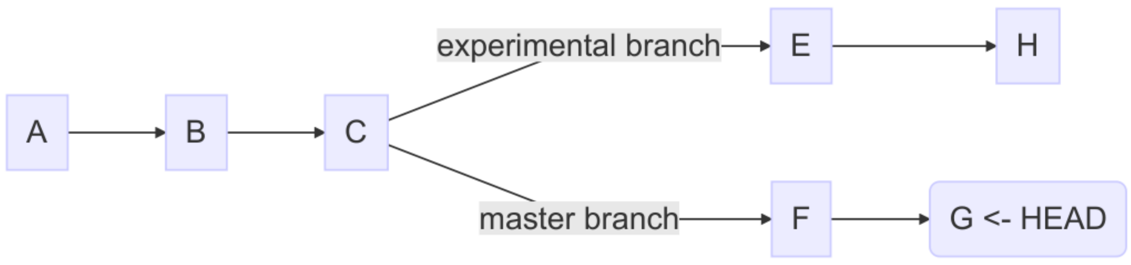
- git branch
- git checkout
- Detached head
- git log decoration

1.4.2. Exercises

TODO

1.5. Merging

We’ve already covered basic branching in previous sections. As you will recall, branching gives you the ability to work on parallel streams of development in the same codebase.

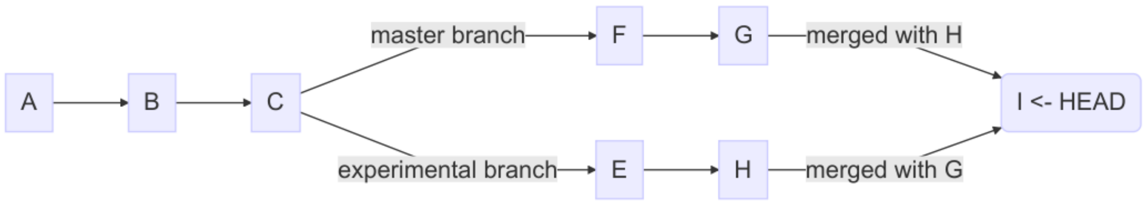


In a sense, merging is the opposite of branching. When we merge, we take two separate points in our development tree and fuse them together.

It’s important to understand merging as it’s a routine job of a repository maintainer to merge branches together.

In the above diagram, the repository is positioned at the tip of master (G). We know this because the HEAD is pointed at it.

If you merge the experimental branch into master with a 'git merge experimental', you end up with a tree that looks like this:

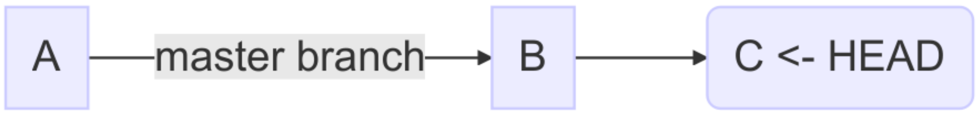


A new change has been made (I). This change merges together the changes made on experimental with the changes made on master.

You can run through the above scenario step-by-step by following these commands:

```
$ rm -rf 1.5.1
$ mkdir -p 1.5.1
$ cd 1.5.1
$ git init
$ echo A > file1
$ git add file1
$ git commit -am 'A'
$ echo B >> file1
$ git commit -am 'B'
$ echo C >> file1
$ git commit -am 'C'
```

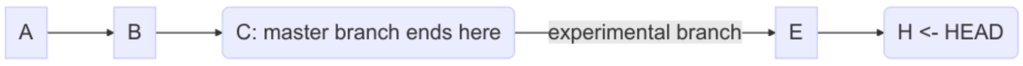
Now you are at this point:



you can branch to experimental and make your changes:

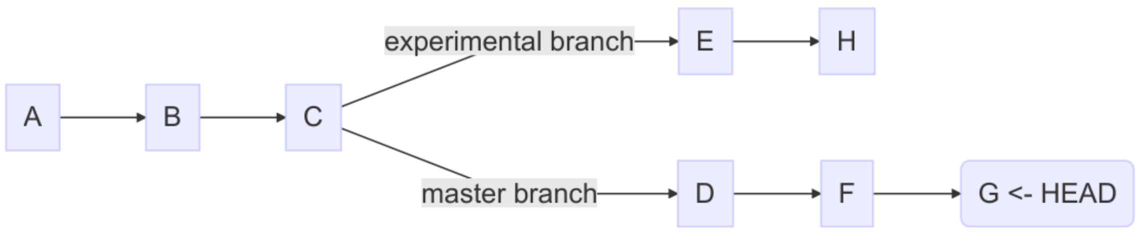
```
$ git branch experimental
$ git checkout experimental
$ git branch
$ echo E >> file1
$ git commit -am 'E'
$ echo H >> file1
$ git commit -am 'H'
```

and the repository is now in this state:



Return to master and make changes D, F and G:

```
$ git checkout master
$ git branch
$ echo D >> file1
$ git commit -am 'D'
$ echo F >> file1
$ git commit -am 'F'
$ echo G >> file1
$ git commit -am 'G'
```



and you are ready to merge!

```
$ git merge experimental
Auto-merging file1
CONFLICT (content): Merge conflict in file1
Automatic merge failed; fix conflicts and then commit the result.
```

Oh dear, that does not look good. The merge failed with a CONFLICT.

1.5.1. What’s going on?

So what exactly happens when you perform a merge?

When you run a merge, git looks at the branch you are on (here it is master), and the branch you are merging in, and works out what the first common ancestor is. In this case, it’s point C, as that’s where you branched experimental.

It then takes the changes on the branch you are merging in from that point and applies them to the branch you are on in one go.

These changes create a new commit, and the git log graph shows the branches joined back up.

Sometimes though, the changes made on the branches conflict with one another. In this case, the D, F and G of the master changed the same lines as the E and H of experimental.

Git doesn’t know what to do with these lines. Should it put the E and H in instead of the D, F and G, or put them all in? If it should put them all in, then what order should they go in?

Changing lines around the same area in code can have disastrous effects, so git does not make a decision when this happens. Instead it tells you that there was a conflict, and asks you to 'fix conflicts and then commit the result'.

If you look at file1 now:

```
A
B
C
<<<<<<< HEAD
D
F
G
=======
E
H
>>>>>>> experimental
```

all the lines from both branches are in the file. There are three sections here. The file up to line C is untouched, as there was no conflict. Then we see a line with arrows indicating the start of a conflicting section, followed by the point in the repo that those changes were made on (in this case, HEAD) '<<<<<<< HEAD'. Then a line of just equals signs indicates the end of a conflicting set of changes, followed by the changes on the other conflicting branch (the E and H on experimental).

What you choose to do here is up to you as maintainers of this repository. You could add or remove lines as you wish until you were happy the merge has been completed. At that point you can commit your change, and the merge has taken place.

You could even leave the file as is (including the '<<<<<<<','=======', and '>>>>>>>' lines, though this is unlikely to be what you want! It’s surprising how easily you can forget to resolve all the conflicting sections in your codebase when doing a merge.

When you are done you can commit the change, and view the history with the git log command.

```
$ git commit -am 'merged experimental in'
$ git log --all --oneline --graph --decorate
*   69441b0 (HEAD, master) merged
|\
| * b3d54fe (experimental) H
| * 4a013db E
* | d9d3722 G
* | bf0fc3e F
* | ccedaee D
|/
* 8835191 C
* f9e5b4f B
* 38471fe A
```

Reading this from bottom to top, you can read commit C and commit H as being merged into the HEAD of master.

NOTE git prefers to show the history from most recent to oldest, which is the opposite of the diagrams in this section. The git man pages like to show time from left to right, like this:

```
              A'--B'--C' topic
             /
D---E---F---G master
```

If you think this is confusing, I won't disagree. However, for git log it makes some sense: if you are looking at a repository with a long history, you are more likely to be interested in recent changes than older ones.

1.5.2. What you learned

- What a merge is
- What a merge conflict is
- How to resolve a merge conflict
- How to read a merged log history

1.5.3. Exercises

1) Initialise a repository, commit a file, make changes on two branches and merge

2) Read over git merge's man page, and research what you don't understand

3) Create a merge knowing there will be a conflict and understand what you need to do to resolve

Chapter 2. Advanced local repository management

Part 1 dealt with core git concepts, and setting up and managing code within a local git repository.

Part 2 deals with advanced local repository management and techniques, before moving onto to Part 3, which looks at working with other git repositories.

In Part 2 you will cover:

- git stash
- git cherry-pick
- git rebase
- git bisect

2.1. Git Stash

Next we introduce a concept that you may end up using a lot!

Often when you are working you want to return to a pristine state, but not lose the work you have done so far.

Traditionally with other source control tools we’ve copied files changed locally aside, then updated our repo, and diffed and re-applied the changed files.

However, git has a concept of the stash to store all local changes ready to reapply at will.

Here is a basic example of a change I want to 'stash':

```
rm -rf 2.1.1
mkdir 2.1.1
cd 2.1.1
git init
echo 'Some content' > file1
git add file1
git commit -am initial
echo 'Some changes I'm not sure about' >> file1
```

You can get very sophisticated with the stash, but 99% of the time I use it like this:

```
[do some work]

[get interupted]

git stash

[deal with interruption]

git stash pop
```

Let’s imagine I’m in the middle of some work, and Alice lets me know that there’s an important update to the code I need to pull from BitBucket.

```
git diff
# diff --git a/file1 b/file1
# index 0ee3895..5554e0f 100644
# --- a/file1
# +++ b/file1
# @@ -1 +1,2 @@
#  Some content
# +Some changes I'm not sure about...
git stash
# Saved working directory and index state WIP on master: 34509a0 initial
# HEAD is now at 34509a0 initial
git status
# On branch master
# nothing to commit, working directory clean
git log --graph --all --decorate
# *   commit 6a2fda32eaf55fedf90c3aa237a528cf7cf50a95 (refs/stash)
# |\  Merge: 34509a0 9ff137c
# | | Author: Ian Miell <ian.miell@gmail.com>
# | | Date:   Tue Jun 28 12:02:45 2016 +0100
# | |
# | |     WIP on master: 34509a0 initial
# | |
# | * commit 9ff137cd51373afe6db37cbac4f1011b0db78ace
# |/  Author: Ian Miell <ian.miell@gmail.com>
# |   Date:   Tue Jun 28 12:02:45 2016 +0100
# |
# |       index on master: 34509a0 initial
# |
# * commit 34509a0afaf3eb9b7ff31dee3ab804903c8d36b0 (HEAD, master)
#   Author: Ian Miell <ian.miell@gmail.com>
#   Date:   Tue Jun 28 12:01:49 2016 +0100
#
#       initial
```

As you can see, it’s committed the state of the index (9ff…) and then committed the local change to the refs/stash branch, and merged them as a child of the HEAD.

Don’t worry too much about the details: it’s basically stored all the changes we’ve made but not *committed* ready to be re-applied.

'stash' is a special branch which is kept local to our repository. The message 'WIP on master' is added automatically for us.

The master branch is still where it was and the HEAD pointer is pointed at it (that is where our repo now is).

I can now do my other work (in this case pulling the latest changes) without concern for whether it conflicts with those changes.

```
git stash list
# stash@{0}: WIP on master: 34509a0 initial
```

Once done, I can reapply those changes by running 'git stash pop':

```
git stash pop
# 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)
#
#   modified:   file1
#
# no changes added to commit (use "git add" and/or "git commit -a")
# Dropped refs/stash@{0} (279ee87c68798caaf2ea3d45fcfa0ac42df6ba4b)
```

which pops the change off the stash stack and restores me to where I was.

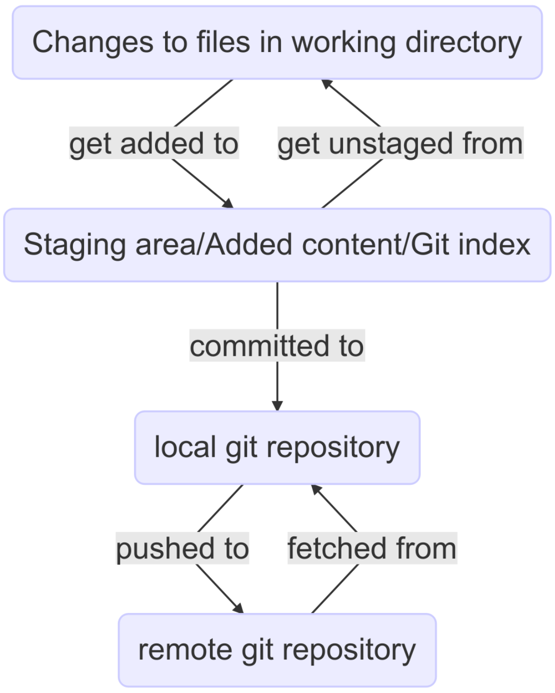
There are more sophisticated ways to deal with stash, but these are not covered here.

2.1.1. What you learned

- What the stash is
- How it works
- How to re-apply changes

2.2. Git add interactive

Previously we discussed the four stages of working in Git:



So far we've shown a difference between adding (staging) and committing, but this still causes confusion for people - what's the point of this?

Let's demonstrate how you might want to use this with a simple example:

```
mkdir 2.2.1
cd 2.2.1
git init
echo 'This is file1' > file1
echo 'This is file2' > file2
git add file1 file2
git commit -am 'files added'
cat > file1 << END
Good change
This is file1
Experimental change
END
cat > file2 << END
All good
This is file2
END
```

```
git add -i
p
1

s
```

```
y
n
q
#imiell@Ians-Air:~/tmpgit$ git add -i
#           staged     unstaged path
#  1:    unchanged        +2/-0 file1
#
#*** Commands ***
#  1: status      2: update   3: revert   4: add untracked
#  5: patch   6: diff     7: quit     8: help
#What now> p
#           staged     unstaged path
#  1:    unchanged        +2/-0 file1
#Patch update>> 1
#           staged     unstaged path
#* 1:    unchanged        +2/-0 file1
#Patch update>>
#diff --git a/file1 b/file1
#index 6a00e12..014f6e4 100644
#--- a/file1
#+++ b/file1
#@@ -1 +1,3 @@
#+Good change
# This is file 1
#+Experimental change
#Stage this hunk [y,n,q,a,d,/,s,e,?]? s
#Split into 2 hunks.
#@@ -1 +1,2 @@
#+Good change
# This is file 1
#Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]? y
#@@ -1 +2,2 @@
# This is file 1
#+Experimental change
#Stage this hunk [y,n,q,a,d,/,K,g,e,?]? n
#
#*** Commands ***
#  1: status      2: update   3: revert   4: add untracked
#  5: patch   6: diff     7: quit     8: help
#What now> q
#Bye.
git status # There are both staged and unstaged changes
git diff   # One change has been added (but not committed), and the other is still a
change only in section 1
#1) Local changes
#2) Staging/adding
#3) Committing to local Repo
#4) Push
```

Now we have staged the good change, but not lost the other changes we have made. This gives us more granular control over the changes committed.

If we are happy with the changes we can go ahead and commit all the changes we made.

NOTE: Committing will still commit all the changes we have made. What is the point of staging then? It is to confirm that you want to commit some changes made locally, but not others.

These changes are added to the 'index' (as opposed to the repository). Remember: index==staging==adding Committing: goes to the repository, which can then be pushed to remote repositories

2.2.1. What you learned

- Difference between staging and committing
- Why the distinction exists
- How to stage specific 'hunks' of code to the index

2.3. Reflog

In this section we're going to look at the reflog.

The reflog gives us references to a sequential history of what we did to the repo. This can come in very handy when you play with your local repo's history, as we will see here.

First set up a repo with two commits:

```
rm -rf 2.3.1
mkdir 2.3.1
cd 2.3.1
git init
echo first commit > file1
git add file1
git commit -am file1
echo second commit >> file1
git commit -am file1.1
git log
```

Then do some magic to effectively remove the last commit:

```
git checkout HEAD^
git branch -f master
git checkout master
git log
```

NOTE don't worry about what we just did - it's a more advanced set of commands that mess with git's history.

The last commit has disappeared! We have fully reverted the master branch to where it was before.

Don’t worry about what we did there, the point here is what we do if we get ourselves into a mess, what do we do to get out of it?

This is where git reflog can help.

Git reflog records all *movements* of branches in the repo. Like stashes, it is local to your repo.

```
git reflog
#66cdcd2 HEAD@{0}: checkout: moving from 66cdcd23c5c005edecd7cd7b162d7b42b7a02ab4 to
master
#66cdcd2 HEAD@{1}: checkout: moving from master to HEAD^
#40e99f7 HEAD@{2}: commit: file1.1
#66cdcd2 HEAD@{3}: commit (initial): file1
```

Git reflog is a history of the changes made to the HEAD (remember the head is a pointer to the current location of the repository).

If we 'reset --hard' the repository to the reference given:

```
git reset --hard 40e99f7
# HEAD is now at 40e99f7 file1.1
git log
```

we are returned to where we were.

The --hard updates both the index (staging/added) and the working tree, as we saw previously.

The refog contains refernces to the state of the repository at various points even if those points are no longer apparently reachable within the repo.

2.3.1. What you have learned

- git reflog
- git reset (--mixed)
- git reset --hard

2.4. Cherry Picking

Next we look at 'cherry-picking'.

Since every commit in git is a change set with a reference id, we can easily port changes from one branch to another.

To demonstrate this, create a simple repository with two changes:

```
rm -rf 2.4.1
mkdir 2.4.1
cd 2.4.1
git init
echo change1 > file1
git add file1
git commit -am change1
echo change2 >> file1
git commit -am change2
git log
cd -
```

At this point we branch off into two branches, master and experimental.

```
cd 2.4.1
git branch experimental
git checkout experimental
ex -sc '1i|crazy change' -cx file1  # Magic to insert before the first line
cat file1
git commit -am crazy
echo more sensible change >> file1
cat file1
git commit -am sensible
```

We decide that the sensible change is the one we want to keep.

First get the reference id with a git log:

```
git log
```

then checkout the master and run a cherry-pick command:

```
git checkout master
git cherry-pick ID
git log
```

NOTE while researching this I came across complex scenarios where the diff was not easily applied (hence the insertion of the 'crazy change' at the top).

Sometimes the cherry-pick might fail because the diff cannot easily be applied, as in this case:

```
rm -rf 2.4.3
mkdir 2.4.3
cd 2.4.3
git init
echo change1 > file1
git add file1
git commit -am change1
echo change2 >> file1
git commit -am change2
git log
git branch experimental
git checkout experimental
echo crazy change >> file1
cat file1
git commit -am crazy
echo more sensible change >> file1
cat file1
git commit -am sensible
git log
git checkout master
git cherry-pick ID
git log
```

```
error: could not apply 743d18e... sensible
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
```

in which case you need to follow the instructions above.

As ever, a git status helps you see what's going on.

```
imiell@Ians-MacBook-Air:~/tmpgit$ git status
#On branch master
#You are currently cherry-picking commit 743d18e.
#  (fix conflicts and run "git cherry-pick --continue")
#  (use "git cherry-pick --abort" to cancel the cherry-pick operation)
#
#Unmerged paths:
#  (use "git add <file>..." to mark resolution)
#
#   both modified:   file1
#
#no changes added to commit (use "git add" and/or "git commit -a")
```

Cherry-picking is often a simple and easy to follow way to move changes between different branches, which can be very useful.

2.4.1. What you learned

- what git cherry pick does

2.5. Git Rebase

Rebasing is one of the most commonly-discussed advanced git topics.

In essence it’s quite simple, but it can get very confusing.

Following this you’re going to learn about rebasing and fast-forwarding with a simple example, so pay attention!

Let’s say we have a set of changes on a master branch:

```
A
|
B
|
C
```

and at this point we branch off to 'feature1' and make another change:

```
A
A
|
B
|
C (HEAD, master)
 \
  D (feature1)
```

Now we go back to master and make a couple more changes:

```
A
A
|
B
|
C
|\
E D (feature1)
|
F (HEAD, master)
```

Now think about this from the point of view of feature1. It made a change from point C on the master branch, but the situation has moved on. Now if master wants to merge in the change on feature1, it could merge it in, and the tree would look like this:

```
A
|
B
|
C
|\
E |
| |
F D (feature1)
| |
|/
G (HEAD, master)
```

That is OK, but not entirely desirable for two reasons:

- The history has just got more complicated
- We have introduced an extra new change (G), which is the merge of D and F

Wouldn’t it be better if the history looked like this?

```
A
|
B
|
C
|
E
|
F
|
D (head, master, feature1)
```

This is much cleaner and easier to follow. If, for example, a bug was introduced in D, it’s easier to find (eg using bisect, which we will come onto). Also, the feature1 branch can be safely deleted without any significant information being lost, making the history tidier and simpler.

If we remind ourselves of the situation pre-merge (above)then we can visualise 'picking up' the changes on the feature1 branch and moving them to the HEAD. So from this:

```
A
|
B
|
C
|\
E D (feature1)
|
F (HEAD, master)
```

To this:

```
A
|
B
|
C
|
E
|
F (HEAD, master)
 \
  D (feature1)
```

This is what a rebase is: you take a set of changes from a particular point and apply them from a different point - re-base!

NOTE be aware that people also talk about rebasing to 'squash' commits. This is a slightly different scneario that uses the same rebase command.

Let’s walk through the above scenario with git commands.

```
rm -rf 2.5.1
mkdir 2.5.1
cd 2.5.1
git init
echo A > file1
git add file1
git commit -am A
echo B >> file1
git commit -am B
echo C >> file1
git commit -am C

git checkout -b feature1
echo D >> file1
git commit -am D
git checkout master
echo E >> file1
git commit -am E
echo F >> file1
git commit -am F

git log --all --decorate --graph
# * commit baacf6fb432967a9d404858268928278df40c7a3 (feature1)
# | Author: Ian Miell <ian.miell@gmail.com>
# | Date:   Wed Jun 29 19:02:09 2016 +0100
# |
# |     D
# |
# | * commit cb548ab427a50028f2dbd721f4c285cbd6ad595d (HEAD, master)
# | | Author: Ian Miell <ian.miell@gmail.com>
# | | Date:   Wed Jun 29 19:02:09 2016 +0100
# | |
# | |     F
# | |
# | * commit 9a9a81060dd74ded8306e7c1a49400529188df70
# |/  Author: Ian Miell <ian.miell@gmail.com>
# |   Date:   Wed Jun 29 19:02:09 2016 +0100
# |
# |       E
# |
# * commit 44954ddfb91d96aaa3bbedab3ae7bcb47aa833be
# | Author: Ian Miell <ian.miell@gmail.com>
# | Date:   Wed Jun 29 19:02:09 2016 +0100
# |
# |     C
# |
# * commit a63e4ff9ba95ab478a5755ed4e3c9c9bc3ddbc37
# | Author: Ian Miell <ian.miell@gmail.com>
# | Date:   Wed Jun 29 19:02:09 2016 +0100
# |
# |     B
# |
# * commit b1fd27851324ed88caa958e2da9d7a36e24277dc
#   Author: Ian Miell <ian.miell@gmail.com>
#   Date:   Wed Jun 29 19:02:09 2016 +0100
#
#       A
```

We are now in this state:

```
A
|
B
|
C
|\
E D (feature1)
|
F (HEAD, master)
```

We go to feature1 and rebase:

```
git checkout feature1
git rebase master
# First, rewinding head to replay your work on top of it...
# Applying: D
# Using index info to reconstruct a base tree...
# M file1
# Falling back to patching base and 3-way merge...
# Auto-merging file1
# CONFLICT (content): Merge conflict in file1
# Failed to merge in the changes.
# Patch failed at 0001 D
# The copy of the patch that failed is found in:
#    /Users/imiell/gitcourse/tmprebase/.git/rebase-apply/patch
#
# When you have resolved this problem, run "git rebase --continue".
# If you prefer to skip this patch, run "git rebase --skip" instead.
# To check out the original branch and stop rebasing, run "git rebase --abort".
vi file1
git add file1
git rebase --continue
# Applying: D
git log --all --decorate --graph
* commit eff7c3a62c8a2ce74302207db014b0db82c22d4e (HEAD, feature1)
| Author: Ian Miell <ian.miell@gmail.com>
| Date:   Wed Jun 29 19:02:09 2016 +0100
|
|     D
|
* commit cb548ab427a50028f2dbd721f4c285cbd6ad595d (master)
| Author: Ian Miell <ian.miell@gmail.com>
| Date:   Wed Jun 29 19:02:09 2016 +0100
|
|     F
|
* commit 9a9a81060dd74ded8306e7c1a49400529188df70
| Author: Ian Miell <ian.miell@gmail.com>
| Date:   Wed Jun 29 19:02:09 2016 +0100
```

```
|
|     E
|
* commit 44954ddfb91d96aaa3bbedab3ae7bcb47aa833be
| Author: Ian Miell <ian.miell@gmail.com>
| Date:   Wed Jun 29 19:02:09 2016 +0100
|
|     C
|
* commit a63e4ff9ba95ab478a5755ed4e3c9c9bc3ddbc37
| Author: Ian Miell <ian.miell@gmail.com>
| Date:   Wed Jun 29 19:02:09 2016 +0100
|
|     B
|
* commit b1fd27851324ed88caa958e2da9d7a36e24277dc
  Author: Ian Miell <ian.miell@gmail.com>
  Date:   Wed Jun 29 19:02:09 2016 +0100

      A
```

Now the changes are in one line we can merge the feature1 master branch.

```
git checkout master
git merge feature1
# Updating cb548ab..eff7c3a
# Fast-forward
#  file1 | 1 +
#  1 file changed, 1 insertion(+)
git log --all --decorate --graph
# * commit eff7c3a62c8a2ce74302207db014b0db82c22d4e (HEAD, master, feature1)
# | Author: Ian Miell <ian.miell@gmail.com>
# | Date:   Wed Jun 29 19:02:09 2016 +0100
# |
# |     D
# |
# * commit cb548ab427a50028f2dbd721f4c285cbd6ad595d
# | Author: Ian Miell <ian.miell@gmail.com>
# | Date:   Wed Jun 29 19:02:09 2016 +0100
# |
# |     F
# |
# * commit 9a9a81060dd74ded8306e7c1a49400529188df70
# | Author: Ian Miell <ian.miell@gmail.com>
# | Date:   Wed Jun 29 19:02:09 2016 +0100
# |
# |     E
# |
# * commit 44954ddfb91d96aaa3bbedab3ae7bcb47aa833be
# | Author: Ian Miell <ian.miell@gmail.com>
# | Date:   Wed Jun 29 19:02:09 2016 +0100
# |
# |     C
# |
# * commit a63e4ff9ba95ab478a5755ed4e3c9c9bc3ddbc37
# | Author: Ian Miell <ian.miell@gmail.com>
# | Date:   Wed Jun 29 19:02:09 2016 +0100
# |
# |     B
# |
# * commit b1fd27851324ed88caa958e2da9d7a36e24277dc
#   Author: Ian Miell <ian.miell@gmail.com>
#   Date:   Wed Jun 29 19:02:09 2016 +0100
#
#       A
```

2.5.1. Fast-forwarding

What’s interesting about the above is this:

```
git merge feature1
# Updating cb548ab..eff7c3a
# Fast-forward
#  file1 | 1 +
#  1 file changed, 1 insertion(+)
```

Because the changes are in a line, no new changes need to be made - the master branch pointer merely needs to be 'fast-forwarded' to the same point as feature1! The HEAD pointer, naturally, moves with the branch we’re on (master).

2.5.2. What you learned

- What a rebase is
- What fast-forward means

2.6. Git Bisect

Bisecting is a very powerful tool for finding bugs.

Let’s say you have a set of changes on a master branch:

```
A1
|
A2
|
A3
|
...
|
A50
|
A51
|
...
|
A100
```

You discover a previously-unseen bug at point A100 and want to debug it. One way to do this is to read over the code, add logging etc.. This can be time-consuming, and there is another simpler way to gather information about what change caused the bug.

Git bisect is a very useful tool for finding out where a bug was introduced. If you know where a bug was introduced, you can look at the diff of the commit that caused it and

It works by picking a start point where the bug definitely did not exist (the 'good' point). In this case we’ll choose point A1. Then you pick a point where the bug definitely did exist (the 'bad' point). In this case, that’s A100.

```
A1     GOOD
|
A2     ?
|
A3     ?
|
...
|
A50    ?
|
A51    ?
|
...
|
A100   BAD
```

Once the git bisect session has that information, it can hand you a version at the hafway point between the 'good' and 'bad' points and asks you to run whatever you need to run to determine whether it’s good or bad. If you tell it it’s 'good' it will mark all version at that point and before as 'good'.

```
A1     GOOD
|
A2     GOOD
|
...
|
A49    GOOD
|
A50    GOOD
|
A51    ?
|
...
|
A99    ?
|
A100   BAD
```

It then repeats the process, giving you a version at the halfway point between 'good' and 'bad', asking you for its status. In this sequence, we are given A75:

```
A50    GOOD
|
A51    ?
|
A75    GOOD OR BAD?
|
A76    ?
|
...
|
A100   BAD
```

If we determine that this version was 'bad', then all the versions after it are marked as bad:

```
A1     GOOD
...
A50    GOOD
|
A51    ?
|
A52    ?
...
A74    ?
|
A75    BAD
|
A76    BAD
...
A99    BAD
|
A100   BAD
```

This binary search process repeats until we know which versions were good and bad. One outcome might be:

```
A1    GOOD
|
A2    GOOD
...
A62   GOOD
|
A63   BAD
|
A64   BAD
...
A100  BAD
```

Once we know that the first 'bad' commit was A63, we can examine the difference between A62 and A63, and this gives us a clue.

2.6.1. A 'real' git bisect session

Let’s make this more realistic with an actual git bisect session.

What you’re going to do is create a git repo with one file (projectfile). In this file you are going to add a line for each commit. The first line will be 1, the second 2, and so on until the hundredth commit which adds the line '100'.

In this scenario the 'bug' is the line '63', but we don’t know that yet. All we know is that we can tell if the bug is in the code with the shell script:

```
rm -rf git-bisect && mkdir -p git-bisect && cd git-bisect
git init
touch projectfile
git add projectfile
for ((i=1;i<=100;i++)); do echo $i >> projectfile; git commit -am "A$i"; done
git log
git bisect start
git bisect bad
git status
git checkout HEAD~99   # Check out the first checkout
git log
git status
git bisect good
git log                # Now at A50
git status
git bisect good
git log                # Now at A75
git bisect bad
git log                # Now at A62
git bisect good
git log                # Now at A68
git bisect bad
git log                # Now at A65
git bisect bad
git log                # Now at A64
git bisect bad
git log                # Now at A63
git bisect bad
# 79583459dc6061bd91d55cfcf8c34fae845f836b is the first bad commit
# commit 79583459dc6061bd91d55cfcf8c34fae845f836b
# Author: Ian Miell <ian.miell@gmail.com>
# Date:   Sun Jul 10 11:53:47 2016 +0100
#
#     A63
#
# :100644 100644 aea6bd8ad6845cca3804a87230fee1b69651643d
55200b3d5d7c0e515eaccaf8465a295017e88249 M    projectfile
```

The bisect is complete, and has reported 79583459dc6061bd91d55cfcf8c34fae845f836b as the first bad commit (this may differ for you).

We can get the diff between this commit and its parent by using the '^' operator with diff:

```
git diff 79583459dc6061bd91d55cfcf8c34fae845f836b^
79583459dc6061bd91d55cfcf8c34fae845f836b
# diff --git a/projectfile b/projectfile
# index aea6bd8..55200b3 100644
# --- a/projectfile
# +++ b/projectfile
# @@ -60,3 +60,4 @@
#  60
#  61
#  62
# +63
```

2.6.2. WHat you learned

- How to bisect a git repo

Chapter 3. Remote Repositories

3.1. Fetching and Pulling Content

In part one we emphasized the point that all git repositories are equal.

This section covers how git repositories communicate with each other and manage their differences.

We have already covered git clone, but let’s create a simple git repo and then clone it:

```
mkdir git_origin
cd git_origin
git init
echo 'first commit' > file1
git add file1
git commit -am file1
cd ..
git clone git_origin git_cloned
```

These two repositories are now contain identical content:

```
diff git_origin/file1 git_cloned/file1
```

However, their .git/config files differ in instructive ways.

The git_origin folder has this:

```
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
    ignorecase = true
    precomposeunicode = true
```

while the git_cloned has this:

```
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
    ignorecase = true
    precomposeunicode = true
[remote "origin"]
    url = /Users/imiell/gitcourse/git_origin
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
```

While the git_origin has no visibility of any 'remotes', the cloned one does.

Its url is (in this case) pointed at the file. URLs can be http(s), ssh, or git.

If I go to the cloned repo and ask it for information about remotes:

```
git remote
# origin
```

I get the name 'origin' back.

The name 'origin' is the default name for a remote, but it has no special meaning. It could be renamed to 'bitbucket', or 'gitlab' for example.

To get more defailed information, I run with -v:

```
git remote -v
# origin    /Users/imiell/gitcourse/git_origin (fetch)
# origin    /Users/imiell/gitcourse/git_origin (push)
```

which gives me the information about the URLs we saw.

3.1.1. fetch

The above remotes are divided into fetch and push. First we look at fetch.

'git fetch' gets the latest changes from the remote repository and copies them into the local repository.

Crucially, these changes are kept in a separate place. You can see this if you use the 'git branch -a' command.

First make a change to the origin’s repo:

```
cd git_origin
echo 'fetchable change' >> file1
git commit -am fetchable
```

Then go to the cloned repository and fetch the changes on master:

```
cd ../git_cloned
git fetch origin master
# remote: Counting objects: 3, done.
# remote: Total 3 (delta 0), reused 0 (delta 0)
# Unpacking objects: 100% (3/3), done.
# From /Users/imiell/gitcourse/git_origin
#  * branch            master     -> FETCH_HEAD
#    ceed883..056dd2c  master     -> origin/master
```

What the above output means is that the origin’s master has been brought into this repo’s references (origin/master).

This has not affected the local master branch at all:

```
git log
```

We can see this repository’s view of all the branches by running a git branch command with '--all':

```
git branch --all
# * master
#   remotes/origin/HEAD -> origin/master
#   remotes/origin/master
```

Here we see the local 'master' branch, followed by the remotes/origin/HEAD pointer (remember HEAD is a pointer to a location in the repository), which is linked to remotes/origin/master.

If you want to dig into the internals at this point, you can peek at the .git folder again:

```
ls .git/refs/
# heads remotes tags
```

which has 'heads' which contains references to local branch:

```
cat .git/refs/heads/master
# ceed883eec5a797471cd1c62365d9f2899b857c7
```

and similarly for remote branches:

```
cat .git/refs/remotes/origin/master
# 056dd2ce64da1e746214107b74866c375a85ffc2
```

So we've 'fetch'ed the remote branch and have it locally.

To apply the remote master's changes to the local one we merge it just as we would for any other reference:

```
git merge origin/master
# Updating ceed883..056dd2c
# Fast-forward
#  file1 | 1 +
#  1 file changed, 1 insertion(+)
git log
# commit 056dd2ce64da1e746214107b74866c375a85ffc2
# Author: Ian Miell <ian.miell@gmail.com>
# Date:   Tue Jun 28 18:41:41 2016 +0100
#
#     fetchable
#
# commit ceed883eec5a797471cd1c62365d9f2899b857c7
# Author: Ian Miell <ian.miell@gmail.com>
# Date:   Tue Jun 28 17:30:44 2016 +0100
#
#     file1
```

3.1.2. What you've learned

- a fetch followed by
- a merge

is what 'git pull' does in one go. A pull fetches the mapped branch, and then merges it into the local branch.

We will cover what your branch locally is mapped to remotely in the next section, where we cover remote repository management in more depth.

3.2. Working with multiple repos

Now you are going to work with multiple repos.

Let's do the same as you did before, but create alice_cloned and bob_cloned:

---

mkdir git_origin cd git_origin git init echo 'first commit' > file1 git add file1 git commit -am file1 cd .. git clone git_origin alice_cloned git clone git_origin bob_cloned ---

Now alice_cloned and bob_cloned have git_origin as the origin:

```
cd alice_cloned
git remote -v
# origin    /Users/imiell/gitcourse/git_origin (fetch)
# origin    /Users/imiell/gitcourse/git_origin (push)
cd ../bob_cloned
git remote -v
# origin    /Users/imiell/gitcourse/git_origin (fetch)
# origin    /Users/imiell/gitcourse/git_origin (push)
```

Now alice makes a change in her master branch:

```
echo alice_change >> file1
# imiell@Ians-MacBook-Air:~/gitcourse/alice_cloned$ git commit -am 'alice change'
# [master 9077a48] alice change
#  1 file changed, 1 insertion(+)
```

Alice <→ Origin <→ Bob

The question is: how does Bob get Alice’s change into his master branch without going to origin?

This is a common scenario in distributed teams. If you consider that git was created for managing the codebase of the Linux operating system, it’s easy to imagine the git_origin as Linus Torvalds' repository, Alice as a contributor and Bob as a so-called lieutenant.

So here is how:

1) ADD alice’s repository as a remote to Bob’s 2) FETCH alice’s updated master branch 3) MERGE alice’s master branch into Bob’s local one

As we have already seen, steps 2) and 3) can be collapsed into a 'git pull', but it is more instructive to keep these separate.

1) ADD alice’s repository as a remote to Bob’s First, Bob needs to add alice’s repository as a remote.

```
git remote add alice ../alice_cloned
git remote -v
# alice ../alice_cloned/ (fetch)
# alice ../alice_cloned/ (push)
# origin    /Users/imiell/gitcourse/git_origin (fetch)
# origin    /Users/imiell/gitcourse/git_origin (push)
```

We have now linked up our repository to alice’s, and given it the name 'alice'.

2) FETCH alice’s updated master branch

```
git fetch alice master
# remote: Counting objects: 3, done.
# remote: Total 3 (delta 0), reused 0 (delta 0)
# Unpacking objects: 100% (3/3), done.
# From ../alice_cloned
#  * branch            master     -> FETCH_HEAD
#  * [new branch]      master     -> alice/master
```

Alice’s master branch is now fetched to our local repository.

```
git branch -vv -a
# * master                fdc7132 [origin/master] file1
#   remotes/alice/master  9077a48 alice change
#   remotes/origin/HEAD   -> origin/master
#   remotes/origin/master fdc7132 file1
```

3) MERGE alice’s master branch into Bob’s local one

```
git merge alice/master
# Updating fdc7132..9077a48
# Fast-forward
#  file1 | 1 +
#  1 file changed, 1 insertion(+)
cat file1
# first commit
# alice_change
```

You may be wondering why we use alice/master and not remotes/alice/master, as the output of 'git branch -vv -a' tells us. You can run:

```
git merge remotes/alice/master
```

which will do the same. Git assumes that the branch is a remote (presumably from seeing the '/' in the branch) and adds the 'remotes' for you.

This 'Lieutenants' model is one example of a git workflow. Although it was the one git was originally created for, it is still common for developers to use a traditional centralised model around a repository such as GitLab or BitBucket.

This is why people make jokes when GitHub is down about the fact that it should not be a problem because it is a distributed source control tool.

3.2.1. What we learned

- How to add a remote repo

- How to fetch changes from the remote repo
- How to merge changes from the remote repo

imiell@Ians-MacBook-Air:/space/git/shutit$ git push origin run:run Counting objects: 3, done. Delta compression using up to 4 threads. Compressing objects: 100% (3/3), done. Writing objects: 100% (3/3), 733 bytes | 0 bytes/s, done. Total 3 (delta 2), reused 0 (delta 0) To git@github.com:ianmiell/shutit.git * [new branch] run → run

3.3. Git Submodules

Submodules are a useful concept, and often seen in real projects.

Git submodules can be very confusing if you stumble into them without much preparation or experience. Following this tutorial, you should have a good understanding for a simple submodule workflow and what is going on when you run the core submodule commands.

Sometimes you want to 'include' one repository in another, but not simply copy it over. Submodules allow you to manage the separate codebase with your repository without changing the other repository.

Let’s look at a concrete example.

Let’s say Alice maintains a library:

```
rm -rf alicelib && mkdir alicelib && cd alicelib
git init
echo 'A' > file1
git add file1
git commit -am 'A'
git checkout -b experimental      # Branch to experimental
echo 'C - EXPERIMENTAL' >> file1
git commit -am EXPERIMENTAL
git checkout master
echo 'B' >> file1
git commit -am 'B'
```

Alice’s library’s history looks like this:

```
A
|\
| C (experimental)
|
B (master)
```

Now Bob wants to use Alice’s library, but specifically wants to use what’s on the experimental branch.

One option is to copy the code over directly, but that seems to be against the spirit of git.

If an improvement is made on the experimental branch, or Bob wants to move later to follow what’s on the master branch, then he must copy over the code he wants. For one file it might be manageable, but for a more realistic and large project, managing this will be completely impractical.

Another option is to check out the code in another folder and link to it in some predictable way in the code (eg 'source ../alice_experimental). Again, this causes management problems, as the user checking out the source must remember to keep code outside this git repository in a certain place for it all to work.

Git submodules solve these management issues, with a little overhead.

They use git commands to track copies of other repositories within your repository. The tracking is under your control (so you decide when it gets 'updated'), and the tracking is done within a file that is stored with your git repository.

WARNING git submodules can be confusing if you don’t follow a few basic patterns or understand how they work, so it’s worth paying attention to this.

Let’s make this clearer with a walkthrough.

We are going to assume you have the 'alicelib' repository created as above.

Now create Bob’s repository:

```
rm -rf bob_repo && mkdir bob_repo && cd bob_repo
git init
echo 'source alicelib' > file1
git add file1
git commit -am 'sourcing alicelib'
echo 'do something with alicelib experimental' >> file1
git commit -am 'using alicelib experimental'
cat file1
# source alicelib
# do something with alicelib
```

Now we have alice’s repo referenced in bob_repo’s code, but bob_repo has no link to alice_repo’s code.

The first step to including alicelib in bob_repo is to initialise submodules:

```
git submodule init
```

Once a git submodule init has been performed, you can 'add' the submodule you want:

```
git submodule add ../alicelib
# Cloning into 'alicelib'...
# done.
```

A new file has been created (.gitmodules), and the folder alicelib has been created:

```
ls -a
# .     ..      .git        .gitmodules alicelib    file1
```

alicelib has been clone just as any other git repository would be anywhere else:

```
ls -a alicelib/
.   ..  .git    file1
```

but the .gitmodules file tracks where the submodule comes from:

```
cat .gitmodules
# [submodule "alicelib"]
#   path = alicelib
#   url = ../alicelib
```

If you get confused, git provides a useful 'status' command for gitmodules:

```
git submodule status
# ff75b7fc52c3a7d52d89a47fd27d7d22ed280b6f alicelib (heads/master)
```

Now, you may have some questions at this point, such as:

- How do you get to the experimental branch?
- What happens if alice's branch changes? Does my code automatically update?
- What if I make a change to alicelib within my repository submodule checkout? Can I push those to alice's? Can I keep those private to my repository?
- What if there are conflicts between these repositories?
    i. and so on. I certainly had these questions when I came to git submodules, and with some trial and error it took me some time to understand what was going on, so I really recommend playing with these simple examples to get the relationships clear in your mind.

3.3.1. Get the experimental branch

Since your 'alicelib' submodule is a straightforward clone of the remote 'alicelib' origin, you have the master branch and the origin's experimental branch:

```
git branch -a -vv
# * master                      ff75b7f [origin/master] B
#   remotes/origin/HEAD         -> origin/master
#   remotes/origin/experimental 969b840 C EXPERIMENTAL
#   remotes/origin/master       ff75b7f B
```

You are on the master branch (indicated with a *), which is mapped to remotes/origin/master.

NOTE the refs (eg ff75b7f) may be different in your output

You do not have an experimental branch locally. However, if you checkout a branch that does not exist locally but does exist remotely, git will assume you want to track that remote branch.

```
git checkout experimental
# Branch experimental set up to track remote branch experimental from origin.
# Switched to a new branch 'experimental'
git branch -a -vv
# * experimental                969b840 [origin/experimental] C EXPERIMENTAL
#   remotes/origin/HEAD         -> origin/master
#   remotes/origin/experimental 969b840 C EXPERIMENTAL
#   remotes/origin/master       ff75b7f B
```

NOTE If more than one remote has the same name, git will not perform this matching. In that case you would have to run the full command:

```
git checkout -b experimental --track origin/master
```

assuming it’s the origin’s master branch you want to track.

3.3.2. Git tracks the submodule’s state

Now that you’ve checked out and tracked the remote experimental branch in your submodule, a change has taken place in bob_repo. If you return to bob_repo’s root folder and run 'git diff' you will see that the subproject commit of 'alicelib' has changed:

```
cd ..
git diff
# diff --git a/alicelib b/alicelib
# index ff75b7f..969b840 160000
# --- a/alicelib
# +++ b/alicelib
# @@ -1 +1 @@
# -Subproject commit ff75b7fc52c3a7d52d89a47fd27d7d22ed280b6f
# +Subproject commit 969b840142f389de55357350a6f26f0825e02393
```

The commit identifier now matches the experimental.

Note that bob_repo tracks the *specific commit* and not the remote branch. This means that changes to alicelib in the origin repository are not automatically tracked within bob_repo's submodule.

We want to commit this change to the submodule:

```
git commit -am 'alicelib moved to experimental'
# [master 1f67953] alicelib moved to experimental
#  2 files changed, 4 insertions(+)
#  create mode 100644 .gitmodules
#  create mode 160000 alicelib
```

3.3.3. Alice makes a change

Alice now spots a bug in her experimental branch that she wants to fix:

```
cd ../alicelib
git checkout experimental
echo 'D' >> file1
git commit -am 'D - a fix added'
```

Now there is a mismatch between alicelib's experimental branch and bob_repo's experimental branch.

```
cd ../bob_repo/alicelib
git status
# On branch experimental
# Your branch is up-to-date with 'origin/experimental'.
# nothing to commit, working directory clean
```

git status reports that bob_repo's alicelib is up-to-date with origin/experimental. Remember that origin/experimental is the locally stored representation of alicelib's experimental branch. Since you have not contacted alicelib to see if there are any updates, this is still the case.

To get the latest changes you can perform a fetch and merge, or save time by running a 'pull', which does both:

```
git pull
# remote: Counting objects: 3, done.
# remote: Total 3 (delta 0), reused 0 (delta 0)
# Unpacking objects: 100% (3/3), done.
# From /Users/imiell/gitcourse/alicelib
#    969b840..1a725f6  experimental -> origin/experimental
# Updating 969b840..1a725f6
# Fast-forward
#  file1 | 1 +
#  1 file changed, 1 insertion(+)
```

GOTCHAS: Generally I would advise not editing repositories that are checked out as submodules until you are more experienced with git. You quickly may find yourself in a 'detached HEAD' state and confused about what you’ve done.

3.3.4. Checking out a project with submodules

Submodules have a special status within git repositories. Since they are both included within a repository and at the same time referencing a remote repository, a simple clone will not check out the included submodule:

```
cd ../..
rm -rf bob_repo_cloned
git clone bob_repo bob_repo_cloned
cd bob_repo_cloned
ls -1
# alicelib
# file1
cd alicelib
ls ## No output
```

Alicelib is not there. Confusingly, 'git submodule status' gives us little clue what’s going on here.

```
git submodule status
# -969b840142f389de55357350a6f26f0825e02393 alicelib
```

The dash (or minus sign) at the front indicates the submodule is not cheked out. Only by running a 'git submodule init' and a 'git submodule update' can you retrieve the appropriate submodule repository:

```
git submodule init
# Submodule 'alicelib' (/Users/imiell/gitcourse/alicelib) registered for path
'alicelib'
git submodule update
# Submodule path 'alicelib': checked out '969b840142f389de55357350a6f26f0825e02393'
git submodule status
# 969b840142f389de55357350a6f26f0825e02393 alicelib (969b840)
```

Now the submodule status has no dash, and a commit ID has been added to the output (969b840).

3.3.5. git clone --recursive

Fortunately there is an easier way. You can clone the repository with a --recursive flag to automatically init and update any submodules (and submodules of those submodules ad infinitum) within the cloned repo:

```
cd ..
git clone --recursive bob_repo bob_repo_cloned_recursive
# Cloning into 'bob_repo_cloned'...
# done.
# Submodule 'alicelib' (/Users/imiell/gitcourse/alicelib) registered for path
'alicelib'
# Cloning into 'alicelib'...
# done.
# Submodule path 'alicelib': checked out '969b840142f389de55357350a6f26f0825e02393'
```

3.3.6. You have learned

- How to set up git submodules
- How to add a submodule to a repo
- How to track remote branches
- How to checkout submodules with init and update
- How to checkout submodules with recursive