Chapter 2. Git Basics

Getting a Git Repository

- You can get a Git project using two main approaches.
- The first takes an existing project or directory and imports it into Git.
- The second clones an existing Git repository from another server.

Initializing a Repository in an Existing Directory

- If youre starting to track an existing project in Git, you need to go to the projects directory and type

```
$ git init
```

- This creates a new subdirectory named `.git` that contains all of your necessary repository files - a Git repository skeleton.
- At this point, nothing in your project is tracked yet.
- If you want to start version-controlling existing files (as opposed to an empty directory), you should probably begin tracking those files and do an initial commit.

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'initial project version'
```

Cloning an Existing Repository

- If you want to get a copy of an existing Git repository the command you need is `git clone`.
- Instead of getting just a working copy, Git receives a full copy of nearly all data that the server has.
- You may lose some server-side hooks and such, but all the versioned data would be there - see Getting Git on a Server for more details.
- You clone a repository with `git clone [url]`.

```
$ git clone https://github.com/libgit2/libgit2
```

- That creates a directory named libgit2, initializes a `.git` directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version.
- If you want to clone the repository into a directory named something other than libgit2, you can specify that as the next command-line option:

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

- Git has a number of different transfer protocols you can use.

Recording Changes to the Repository

- Each file in your working directory can be in one of two states: tracked or untracked.
- Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged.
- Untracked files are everything else - any files in your working directory that were not in your last snapshot and are not in your staging area.
- When you first clone a repository, all of your files will be tracked and unmodified because you just checked them out and havent edited anything.

- As you edit files, Git sees them as modified, because youve changed them since your last commit.
- You stage these modified files and then commit all your staged changes, and the cycle repeats.

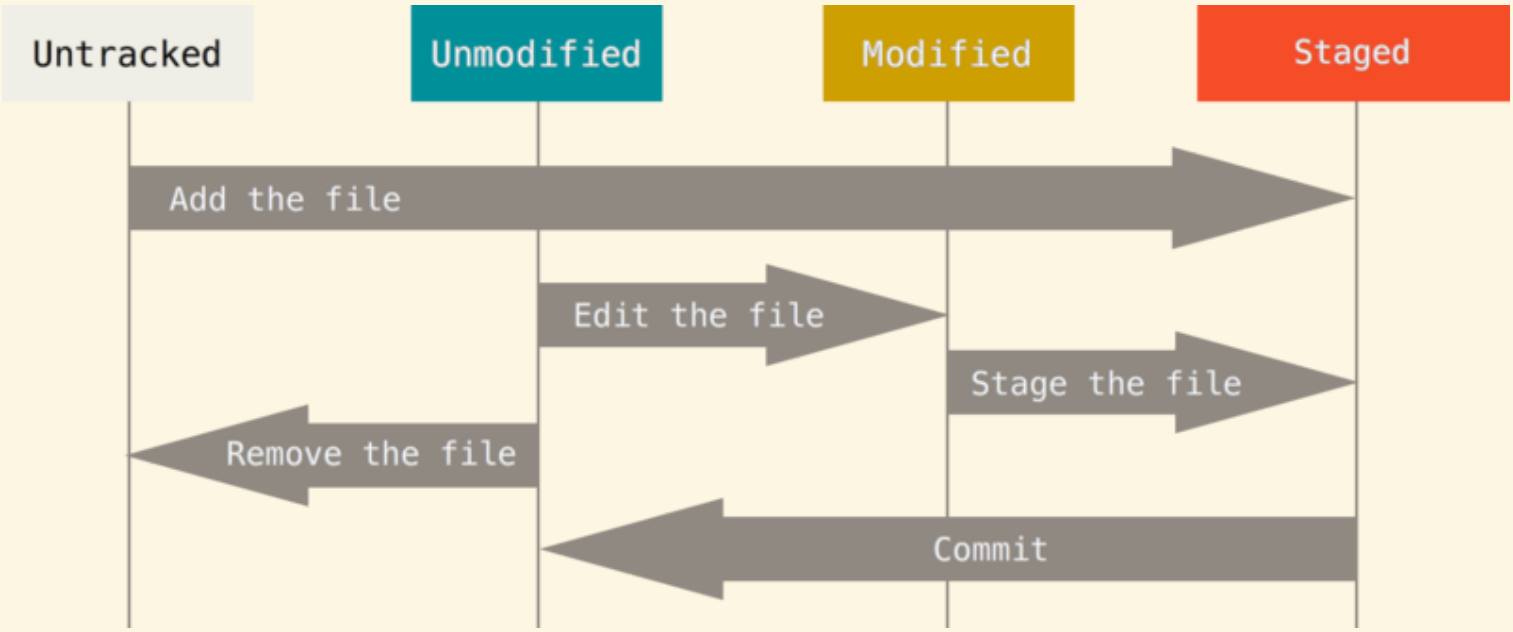

Figure 2-1. The lifecycle of the status of your files.

Checking the Status of Your Files

- The main tool you use to determine which files are in which state is the `git status` command.

```
$ git status
On branch master
nothing to commit, working directory clean
```

- There are no tracked and modified files.
- Git also doesnt see any untracked files, or they would be listed here.
- Finally, the command tells you which branch youre on and informs you that it has not diverged from the same branch on the server.
- Lets say you add a new file to your project, a simple README file.

```
$ echo 'My Project' > README
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

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

Tracking New Files

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

```
$ git add README
```

- If you run your status command again, you can see that your README file is now tracked and staged to be committed:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
```

- The `git add` command takes a path name for either a file or a directory; if its a directory, the command adds all the files in that directory recursively.

Staging Modified Files

- If you change a previously tracked file called CONTRIBUTING.md and then run your `git status` command again, you get something that looks like this:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

- Lets run `git add` now to stage the CONTRIBUTING.md file, and then run `git status` again:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

- Both files are staged and will go into your next commit.
- At this point, suppose you remember one little change that you want to make in `CONTRIBUTING.md` before you commit it.
- You open it again and make that change, and youre ready to commit.
- However, lets run `git status` one more time:

```
$ vim CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md

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:   CONTRIBUTING.md
```

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Short Status

- Git also has a short status flag so you can see your changes in a more compact way.
- If you run `git status -s` or `git status --short` you get a far more simplified output from the command.

```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

- New files that arent tracked have a `??` next to them, new files that have been added to the staging area have an `A`, modified files have an `M` and so on.
- There are two columns to the output - the left hand column indicates that the file is staged and the right hand column indicates that its modified.
- The `README` file is modified in the working directory but not yet staged, while the `lib/simplegit.rb` file is modified and staged.
- The `Rakefile` was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

Ignoring Files

- These are generally automatically generated files such as log files or files produced by your build system.

- In such cases, you can create a file listing patterns to match them named `.gitignore`. Here is an example `.gitignore` file:

```
$ cat .gitignore
*.[oa]
*~
```

- The first line tells Git to ignore any files ending in .o or .a - object and archive files that may be the product of building your code.
- The second line tells Git to ignore all files that end with a tilde (`~`), which is used by many text editors such as Emacs to mark temporary files.
- The rules for the patterns you can put in the `.gitignore` file are as follows:
    - Blank lines or lines starting with `#` are ignored.
    - Standard glob patterns work.
    - You can end patterns with a forward slash (`/`) to specify a directory.
    - You can negate a pattern by starting it with an exclamation point (`!`).
- Glob patterns are like simplified regular expressions that shells use.
- Here is another example .gitignore file:

```
# no .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the root TODO file, not subdir/TODO
/TODO

# ignore all files in the build/ directory
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .txt files in the doc/ directory
doc/**/*.txt
```

Viewing Your Staged and Unstaged Changes

- You want to know exactly what you changed, not just which files were changed - you can use the `git diff` command.
- What have you changed but not yet staged? And what have you staged that you are about to commit?

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

- To see what youve changed but not yet staged, type `git diff` with no other arguments:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if you patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

- The result tells you the changes youve made that you havent yet staged.
- If you want to see what youve staged that will go into your next commit, you can use `git diff --staged`.

- This command compares your staged changes to your last commit:

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

- Its important to note that `git diff` by itself doesnt show all changes made since your last commit - only changes that are still unstaged.

```
$ git add CONTRIBUTING.md
$ echo 'test line' >> CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   CONTRIBUTING.md
```

```
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:   CONTRIBUTING.md
```

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

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 643e24f..87f08c8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
 ## Starter Projects

 See our [projects list](https://github.com/libgit2/libgit2/blob/development/PROJECTS
+# test line
```

- and `git diff --cached` to see what youve staged so far (—staged and —cached are synonyms):

```
$ git diff --cached
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if you patch is
+longer than a dozen lines.

 If you are starting to work on a particular area, feel free to submit a PR
 that highlights your work in progress (and note in the PR title that it's
```

> *Git Diff in an External Tool*
> *We will continue to use the `git diff` command in various ways throughout the rest of the book. There is another way to look at these diffs if you prefer a graphical or external diff viewing program instead. If you run `git difftool` instead of `git diff`, you can view any of these diffs in software like Araxis, emerge, vimdiff and more. Run `git difftool --tool-help` to see what is available on your system.*

Committing Your Changes

- Remember that anything that is still unstaged - any files you have created or modified that you havent run `git add` on since you edited them - wont go into this commit.

- The simplest way to commit is to type `git commit`:

```
$ git commit
```

- Doing so launches your editor of choice.
- This is set by your shells `$EDITOR` environment variable - usually vim or emacs, although you can configure it with whatever you want using the `git config --global core.editor`.
- The editor displays the following text (this example is a Vim screen):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#    new file:   README
#    modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

- Can pass the `-v` option to `git commit`.
- Doing so also puts the diff of your change in the editor so you can see exactly what changes youre committing.
- When you exit the editor, Git creates your commit with that commit message (with the comments and diff stripped out).
- Alternatively, you can type your commit message inline with the `commit` command by specifying it after a -m flag, like this:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

- Remember that the commit records the snapshot you set up in your staging area.
- Anything you didnt stage is still sitting there modified; you can do another commit to add it to your history.
- Every time you perform a commit, youre recording a snapshot of your project that you can revert to or compare to later.

Skipping the Staging Area

- The staging area is sometimes a bit more complex than you need in your workflow.
- If you want to skip the staging area, Git provides a simple shortcut.
- Adding the `-a` option to the `git commit` command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the `git add` part:

```
$ git status
On branch master
```

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

- Notice how you dont have to run `git add` on the CONTRIBUTING.md file in this case before you commit.

Removing Files

- To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit.
- The `git rm` command does that, and also removes the file from your working directory so you dont see it as an untracked file the next time around.
- If you simply remove the file from your working directory, it shows up under the Changed but not updated (that is, *unstaged*) area of your `git status` output:

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    PROJECTS.md

no changes added to commit (use "git add" and/or "git commit -a")
```

- Then, if you run `git rm`, it stages the files removal:

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    deleted:    PROJECTS.md
```

- The next time you commit, the file will be gone and no longer tracked.
- If you modified the file and added it to the index already, you must force the removal with the `-f` option.

- Another useful thing you may want to do is to keep the file in your working tree but remove it from your staging area.
- `.gitignore` file and accidentally staged it, like a large log file or a bunch of `.a` compiled files.
- To do this, use the `\--cached` option:

```
$ git rm --cached README
```

- You can pass files, directories, and file-glob patterns to the `git rm` command. That means you can do things such as

```
$ git rm log/\*.log
```

- Note the backslash (`\\`) in front of the `*`.
- This is necessary because Git does its own filename expansion in addition to your shells filename expansion.
- This command removes all files that have the `.log` extension in the `log/` directory.
- Or, you can do something like this:

```
$ git rm \*~
```

Moving Files

- Thus its a bit confusing that Git has a `mv` command. If you want to rename a file in Git, you can run something like

```
$ git mv file_from file_to
```

- In fact, if you run something like this and look at the status, youll see that Git considers it a renamed file:

```
$ git mv README.md README
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

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

```
$ mv README.md README
$ git rm README.md
$ git add README
```

Viewing the Commit History

- The most basic and powerful tool to do this is the `git log` command.
- These examples use a very simple project called simplegit. To get the project, run

```
git clone https://github.com/schacon/simplegit-progit
```

- When you run `git log` in this project, you should get output that looks something like this:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

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

    first commit
```

- By default, with no arguments, `git log` lists the commits made in that repository in reverse chronological order - that is, the most recent commits show up first.
- One of the more helpful options is `-p`, which shows the difference introduced in each commit.
- You can also use `-2`, which limits the output to only the last two entries:

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.platform  =   Gem::Platform::RUBY
     s.name      =   "simplegit"
```

```
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com"
     s.summary   =   "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
     end

 end
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
\ No newline at end of file
```

- This option displays the same information but with a diff directly following each entry.
- For example, if you want to see some abbreviated stats for each commit, you can use the `\--stat` option:

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

 Rakefile | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

 lib/simplegit.rb | 5 -----
 1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
```

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

    first commit

 README           |  6 ++++++
 Rakefile         | 23 +++++++++++++++++++++
 lib/simplegit.rb | 25 +++++++++++++++++++++++++
 3 files changed, 54 insertions(+)
```

- As you can see, the `\--stat` option prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed.
- It also puts a summary of the information at the end.
- Another really useful option is `\--pretty`.
- This option changes the log output to formats other than the default.

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

- The most interesting option is `format`, which allows you to specify your own log output format.

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : changed the version number
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
a11bef0 - Scott Chacon, 6 years ago : first commit
```

Table 2-1 lists some of the more useful options that format takes.

TABLE 2-1. *Useful options for `git log --pretty=format`*

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

- The author is the person who originally wrote the work, whereas the committer is the person who last applied the work.
- The oneline and format options are particularly useful with another `log` option called `\--graph`.
- This option adds a nice little ASCII graph showing your branch and merge history:

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
*  11d191e Merge branch 'defunkt' into local
```

- Table 2-2 lists the options weve covered so far, as well as some other common formatting options that may be useful, along with how they change the output of the log command.

Table 2-2. Common options to `git log`

TABLE 2-2. *Common options to `git log`*

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

Limiting Log Output

- Youve seen one such option already - the `-2` option, which show only the last two commits.
- In reality, youre unlikely to use that often, because Git by default pipes all output through a pager so you see only one page of log output at a time.
- However, the time-limiting options such as `\--since` and `\--until` are very useful.
- For example, this command gets the list of commits made in the last two weeks:

```
$ git log --since=2.weeks
```

- The `\--author` option allows you to filter on a specific author, and the `\--grep` option lets you search for keywords in the commit messages.
- Another really helpful filter is the `-S` option which takes a string and only shows the commits that introduced a change to the code that added or removed that string.

```
$ git log -Sfunction_name
```

- The last really useful option to pass to `git log` as a filter is a path.
- If you specify a directory or file name, you can limit the log output to commits that introduced a change to those files.

- In Table 2-3 well list these and a few other common options for your reference.

Table 2-3. Options to limit the output of `git log`

TABLE 2-3. *Options to limit the output of `git log`*

Option	Description
`-(n)`	Show only the last n commits
`--since, --after`	Limit the commits to those made after the specified date.
`--until, --before`	Limit the commits to those made before the specified date.

Option	Description
`--author`	Only show commits in which the author entry matches the specified string.
`--committer`	Only show commits in which the committer entry matches the specified string.
`--grep`	Only show commits with a commit message containing the string
`-S`	Only show commits adding or removing code matching the string

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
   --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attributes are in use
acd3b9e - Enhance hold_lock_file_for_{update,append}() API
f563754 - demonstrate breakage of detached checkout with symbolic link HEAD
d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths
51a94af - Fix "checkout --track -b newbranch" on detached HEAD
b0ad11e - pull: allow "git pull origin $something:$current_branch" into an unborn branc
```

Undoing Things

- You cant always undo some of these undos.
- This is one of the few areas in Git where you may lose some work if you do it wrong.
- One of the common undos takes place when you commit too early and possibly forget to add some files, or you mess up your commit message.
- If you want to try that commit again, you can run commit with the `\--amend` option:

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

- As an example, if you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

Unstaging a Staged File

- For example, lets say youve changed two files and want to commit them as two separate changes, but you accidentally type `git add *` and stage them both.
- The `git status` command reminds you:

```
$ git add .
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
    modified:   CONTRIBUTING.md
```

- Right below the Changes to be committed text, it says use `git reset HEAD <file>...` to unstage.
- So, lets use that advice to unstage the `CONTRIBUTING.md` file:

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M    CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README

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:   CONTRIBUTING.md
```

- The command is a bit strange, but it works. The `CONTRIBUTING.md` file is modified but once again unstaged.
- While `git reset` *can* be a dangerous command if you call it with `\--hard`, in this instance the file in your working directory is not touched.
- Calling `git reset` without an option is not dangerous - it only touches your staging area.

Unmodifying a Modified File

- Can you easily unmodify it - revert it back to what it looked like when you last committed (or initially cloned, or however you got it into your working directory)?

```
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:   CONTRIBUTING.md
```

- It tells you pretty explicitly how to discard the changes youve made. Lets do what it says:

```
$ git checkout -- CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

- Remember, anything that is *committed* in Git can almost always be recovered.
- Even commits that were on branches that were deleted or commits that were overwritten with an `\--amend` commit can be recovered.

Working with Remotes

- Remote repositories are versions of your project that are hosted on the Internet or network somewhere.
- Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work.

Showing Your Remotes

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

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

- You can also specify `-v`, which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote:

```
$ git remote -v
origin    https://github.com/schacon/ticgit (fetch)
origin    https://github.com/schacon/ticgit (push)
```

- A repository with multiple remotes for working with several collaborators might look something like this.

```
$ cd grit
$ git remote -v
bakkdoor  https://github.com/bakkdoor/grit (fetch)
bakkdoor  https://github.com/bakkdoor/grit (push)
cho45     https://github.com/cho45/grit (fetch)
cho45     https://github.com/cho45/grit (push)
defunkt   https://github.com/defunkt/grit (fetch)
defunkt   https://github.com/defunkt/grit (push)
koke      git://github.com/koke/grit.git (fetch)
koke      git://github.com/koke/grit.git (push)
origin    git@github.com:mojombo/grit.git (fetch)
origin    git@github.com:mojombo/grit.git (push)
```

Adding Remote Repositories

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

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin    https://github.com/schacon/ticgit (fetch)
origin    https://github.com/schacon/ticgit (push)
pb    https://github.com/paulboone/ticgit (fetch)
pb    https://github.com/paulboone/ticgit (push)
```

- If you want to fetch all the information that Paul has but that you dont yet have in your repository, you can run `git fetch pb`:

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

- Pauls master branch is now accessible locally as `pb/master` - you can merge it into one of your branches, or you can check out a local branch at that point if you want to inspect it.

Fetching and Pulling from Your Remotes

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

```
$ git fetch [remote-name]
```

- If you clone a repository, the command automatically adds that remote repository under the name origin.
- So, `git fetch origin` fetches any new work that has been pushed to that server since you cloned it.
- Its important to note that the `git fetch` command pulls the data to your local repository - it doesnt automatically merge it with any of your work or modify what youre currently working on.
- You have to merge it manually into your work when youre ready.

- You can use the `git pull` command to automatically fetch and then merge a remote branch into your current branch.

- `git pull` generally fetches data from the server you originally cloned from and automatically tries to merge it into the code youre currently working on.

Pushing to Your Remotes

- `git push [remote-name] [branch-name]`

```
$ git push origin master
```

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

Inspecting a Remote

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

```
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/schacon/ticgit
  Push  URL: https://github.com/schacon/ticgit
  HEAD branch: master
  Remote branches:
    master                               tracked
    dev-branch                           tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

- The command helpfully tells you that if youre on the master branch and you run `git pull`, it will automatically merge in the master branch on the remote after it fetches all the remote

references.

```
$ git remote show origin
* remote origin
  URL: https://github.com/my-org/complex-project
  Fetch URL: https://github.com/my-org/complex-project
  Push  URL: https://github.com/my-org/complex-project
  HEAD branch: master
  Remote branches:
    master                           tracked
    dev-branch                       tracked
    markdown-strip                   tracked
    issue-43                         new (next fetch will store in remotes/origin)
    issue-45                         new (next fetch will store in remotes/origin)
    refs/remotes/origin/issue-11     stale (use 'git remote prune' to remove)
  Local branches configured for 'git pull':
    dev-branch merges with remote dev-branch
    master     merges with remote master
  Local refs configured for 'git push':
    dev-branch                     pushes to dev-branch                     (up to date
    markdown-strip                 pushes to markdown-strip                 (up to date
    master                         pushes to master                         (up to date
```

- This command shows which branch is automatically pushed to when you run `git push` while on certain branches.
- It also shows you which remote branches on the server you dont yet have, which remote branches you have that have been removed from the server, and multiple branches that are automatically merged when you run `git pull`.

Removing and Renaming Remotes

- rename a reference you can run `git remote rename` to change a remotes shortname.
- For instance, if you want to rename `pb` to `paul`, you can do so with `git remote rename`:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

- If you want to remove a remote for some reason - youve moved the server or are no longer using a particular mirror, or perhaps a contributor isnt contributing anymore - you can use `git remote rm`:

```
$ git remote rm paul
$ git remote
origin
```

Tagging

- Git has the ability to tag specific points in history as being important.

Listing Your Tags

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

```
$ git tag
v0.1
v1.3
```

- This command lists the tags in alphabetical order; the order in which they appear has no real importance.

```
$ git tag -l 'v1.8.5*'
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

Creating Tags

- Git uses two main types of tags: lightweight and annotated.
- A lightweight tag is very much like a branch that doesnt change - its just a pointer to a specific commit.
- Annotated tags, however, are stored as full objects in the Git database.
- Theyre checksummed; contain the tagger name, e-mail, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG).

Annotated Tags

- The easiest way is to specify `-a` when you run the `tag` command:

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

- The `-m` specifies a tagging message, which is stored with the tag.
- You can see the tag data along with the commit that was tagged by using the `git show` command:

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:19:12 2014 -0700

my version 1.4
```

```
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

Lightweight Tags

- To create a lightweight tag, dont supply the `-a`, `-s`, or `-m` option:

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

```
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

Tagging Later

- You can also tag commits after youve moved past them. Suppose your commit history looks like this:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

- You can add it after the fact.
- To tag that commit, you specify the commit checksum (or part of it) at the end of the command:

```
$ git tag -a v1.2 9fceb02
```

- You can see that youve tagged the commit:

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    updated rakefile
...
```

Sharing Tags

- By default, the `git push` command doesnt transfer tags to remote servers.
- You will have to explicitly push tags to a shared server after you have created them.
- This process is just like sharing remote branches - you can run `git push origin [tagname]`.

```
$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.5 -> v1.5
```

- If you have a lot of tags that you want to push up at once, you can also use the `\--tags` option to the `git push` command.

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]         v1.4 -> v1.4
 * [new tag]         v1.4-lw -> v1.4-lw
```

Checking out Tags

- You cant really check out a tag in Git, since they cant be moved around.

- If you want to put a version of your repository in your working directory that looks like a specific tag, you can create a new branch at a specific tag with `git checkout -b [branchname] [tagname]`:

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

Git Aliases

- Git doesnt automatically infer your command if you type it in partially.
- If you dont want to type the entire text of each of the Git commands, you can easily set up an alias for each command using `git config`.

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

- This means that, for example, instead of typing `git commit`, you just need to type `git ci`.
- You can add your own unstage alias to Git:

```
$ git config --global alias.unstage 'reset HEAD --'
```

- This makes the following two commands equivalent:

```
$ git unstage fileA
$ git reset HEAD fileA
```

- This seems a bit clearer. Its also common to add a `last` command, like this:

```
$ git config --global alias.last 'log -1 HEAD'
```

- This way, you can see the last commit easily:

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Author: Josh Goebel <dreamer3@example.com>
Date:   Tue Aug 26 19:48:51 2008 +0800

    test for current head

    Signed-off-by: Scott Chacon <schacon@example.com>
```

- Maybe you want to run an external command, rather than a Git subcommand.
- In that case, you start the command with a `!` character. This is useful if you write your own tools that work with a Git repository.
- We can demonstrate by aliasing `git visual` to run `gitk`:

```
$ git config --global alias.visual '!gitk'
```

Summary

- At this point, you can do all the basic local Git operations - creating or cloning a repository, making changes, staging and committing those changes, and viewing the history of all the changes the repository has been through.
- Next, well cover Gits killer feature: its branching model.