Advanced Git log

The purpose of any version control system is to record changes to your code. This gives you the power to go back into your project history to see who contributed what, figure out where bugs were introduced, and revert problematic changes. But, having all of this history available is useless if you don't know how to navigate it. That's where the git log command comes in.

By now, you should already know the basic git log command for displaying commits. But, you can alter this output by passing many different parameters to git log.

The advanced features of git log can be split into two categories: formatting how each commit is displayed, and filtering which commits are included in the output. Together, these two skills give you the power to go back into your project and find any information that you could

Formatting Log Output

First, this article will take a look at the many ways in which `git log`'s output can be formatted. Most of these come in the form of flags that let you request more or less information from `git log`.

If you don't like the default `git log` format, you can use `git config`'s aliasing functionality to create a shortcut for any of the formatting options discussed below. Please see in The git config Command for how to set up an alias.

Oneline

The `--oneline` flag condenses each commit to a single line. By default, it displays only the commit ID and the first line of the commit message. Your typical `git log --oneline` output will look something like this:

```
0e25143 Merge branch 'feature'
ad8621a Fix a bug in the feature
16b36c6 Add a new feature
23ad9ad Add the initial code base
```

This is very useful for getting a high-level overview of your project.

Decorating

Many times it's useful to know which branch or tag each

Tutorials

This can be combined with other configuration options. For example, running `git log --oneline --decorate` will format the commit history like so:

```
0e25143 (HEAD, master) Merge branch 'feature'
ad8621a (feature) Fix a bug in the feature
16b36c6 Add a new feature
23ad9ad (tag: v0.9) Add the initial code base
```

This lets you know that the top commit is also checked out (denoted by `HEAD`) and that it is also the tip of the `master` branch. The second commit has another branch pointing to it called `feature`, and finally the 4th commit is tagged as `v0.9`.

Branches, tags, `HEAD`, and the commit history are almost all of the information contained in your Git repository, so this gives you a more complete view of the logical structure of your repository.

Diffs

The `git log` command includes many options for displaying diffs with each commit. Two of the most common options are `--stat` and `-p`.

The `--stat` option displays the number of insertions and deletions to each file altered by each commit (note that modifying a line is represented as 1 insertion and 1 deletion). This is useful when you want a brief summary of the changes introduced by each commit. For example, the following commit added 67 lines to the `hello.py` file and removed 38 lines:

Tutorials


```
    Add a new feature

 hello.py | 105 ++++++++++++++++++++++++++++---------
 1 file changed, 67 insertion(+), 38 deletions(-)
```

The amount of `+` and `-` signs next to the file name show the relative number of changes to each file altered by the commit. This gives you an idea of where the changes for each commit can be found.

If you want to see the actual changes introduced by each commit, you can pass the `-p` option to `git log`. This outputs the entire patch representing that commit:

```
commit 16b36c697eb2d24302f89aa22d9170dfe609855b
Author: Mary <mary@example.com>
Date:   Fri Jun 25 17:31:57 2014 -0500

    Fix a bug in the feature

diff --git a/hello.py b/hello.py
index 18ca709..c673b40 100644
--- a/hello.py
+++ b/hello.py
@@ -13,14 +13,14 @@ B
-print("Hello, World!")
+print("Hello, Git!")
```

For commits with a lot of changes, the resulting output can become quite long and unwieldy. More often than not, if you're displaying a full patch, you're probably searching for a specific change. For this, you want to use the pickaxe option.

The Shortlog

The `git shortlog` command is a special version of `git log` intended for creating release announcements.

Tutorials

For example, if two developers have contributed 5 commits to a project, the `git shortlog` output might look like the following:

```
Mary (2):
      Fix a bug in the feature
      Fix a serious security hole in our framewor

John (3):
      Add the initial code base
      Add a new feature
      Merge branch 'feature'
```

By default, `git shortlog` sorts the output by author name, but you can also pass the `-n` option to sort by the number of commits per author.

Graphs

The `--graph` option draws an ASCII graph representing the branch structure of the commit history. This is commonly used in conjunction with the `--oneline` and `--decorate` commands to make it easier to see which commit belongs to which branch:

```
git log --graph --oneline --decorate
```

For a simple repository with just 2 branches, this will produce the following:

```
*   0e25143 (HEAD, master) Merge branch 'feature'
|\
| * 16b36c6 Fix a bug in the new feature
| * 23ad9ad Start a new feature
* | ad8621a Fix a critical security issue
```

The asterisk shows which branch the commit was on, so the above graph tells us that the `23ad9ad` and `16b36c6` commits are on a topic branch and the rest are on the `master` branch.

While this is a nice option for simple repositories, you're probably better off with a more full-featured visualization tool like `gitk` or SourceTree for projects that are heavily branched.

Custom Formatting

For all of your other `git log` formatting needs, you can use the `--pretty=format:"<string>"` option. This lets you display each commit however you want using `printf`-style placeholders.

For example, the `%cn`, `%h` and `%cd` characters in the following command are replaced with the committer name, abbreviated commit hash, and the committer date, respectively.

```
git log --pretty=format:"%cn committed %h on %cd"
```

This results in the following format for each commit:

```
John committed 400e4b7 on Fri Jun 24 12:30:04 201
John committed 89ab2cf on Thu Jun 23 17:09:42 201
Mary committed 180e223 on Wed Jun 22 17:21:19 201
John committed f12ca28 on Wed Jun 22 13:50:31 201
```

The complete list of placeholders can be found in the Pretty Formats section of the `git log` manual page.

`git log` output into another command.

Filtering the Commit History

Formatting how each commit gets displayed is only half the battle of learning `git log`. The other half is understanding how to navigate the commit history. The rest of this article introduces some of the advanced ways to pick out specific commits in your project history using `git log`. All of these can be combined with any of the formatting options discussed above.

By Amount

The most basic filtering option for `git log` is to limit the number of commits that are displayed. When you're only interested in the last few commits, this saves you the trouble of viewing all the commits in a pager.

You can limit `git log`'s output by including the `-<n>` option. For example, the following command will display only the 3 most recent commits.

```
git log -3
```

By Date

If you're looking for a commit from a specific time frame, you can use the `--after` or `--before` flags for filtering

July 1st, 2014 (inclusive):

```
git log --after="2014-7-1"
```

You can also pass in relative references like `"1 week ago"` and `"yesterday"`:

```
get log --after="yesterday"
```

To search for a commits that were created between two dates, you can provide both a `--before` and `--after` date. For instance, to display all the commits added between July 1st, 2014 and July 4th, 2014, you would use the following:

```
git log --after="2014-7-1" --before="2014-7-4"
```

Note that the `--since` and `--until` flags are synonymous with `--after` and `--before`, respectively.

By Author

When you're only looking for commits created by a particular user, use the `--author` flag. This accepts a regular expression, and returns all commits whose author matches that pattern. If you know exactly who you're looking for, you can use a plain old string instead of a regular expression:

```
git log --author="John"
```

phrase.

You can also use regular expressions to create more complex searches. For example, the following command searches for commits by either *Mary* or *John*.

```
git log --author="John\|Mary"
```

Note that the author's email is also included with the author's name, so you can use this option to search by email, too.

If your workflow separates committers from authors, the `--committer` flag operates in the same fashion.

By Message

To filter commits by their commit message, use the `--grep` flag. This works just like the `--author` flag discussed above, but it matches against the commit message instead of the author.

For example, if your team includes relevant issue numbers in each commit message, you can use something like the following to pull out all of the commits related to that issue:

```
git log --grep="JRA-224:"
```

You can also pass in the `-i` parameter to `git log` to make it ignore case differences while pattern matching.

Tutorials

happened to a particular file. To show the history related to a file, all you have to do is pass in the file path. For example, the following returns all commits that affected either the `foo.py` or the `bar.py` file:

```
git log -- foo.py bar.py
```

The `--` parameter is used to tell `git log` that subsequent arguments are file paths and not branch names. If there's no chance of mixing it up with a branch, you can omit the `--`.

By Content

It's also possible to search for commits that introduce or remove a particular line of source code. This is called a *pickaxe*, and it takes the form of `-S"<string>"`. For example, if you want to know when the string *Hello, World!* was added to any file in the project, you would use the following command:

```
git log -S"Hello, World!"
```

If you want to search using a regular expression instead of a string, you can use the `-G"<regex>"` flag instead.

This is a very powerful debugging tool, as it lets you locate all of the commits that affect a particular line of code. It can even show you when a line was copied or moved to another file.

only the commits contained in that range. The range is specified in the following format, where `<since>` and `<until>` are commit references:

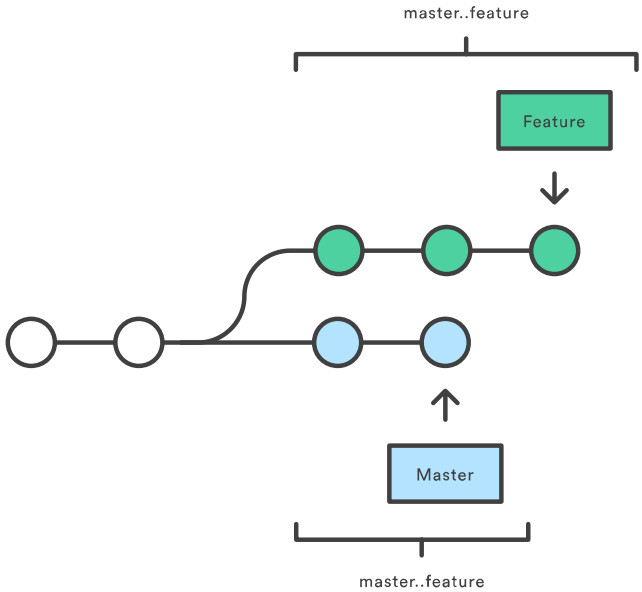
```
git log <since>..<until>
```

This command is particularly useful when you use branch references as the parameters. It's a simple way to show the differences between 2 branches. Consider the following command:

```
git log master..feature
```

The `master..feature` range contains all of the commits that are in the `feature` branch, but aren't in the `master` branch. In other words, this is how far `feature` has progressed since it forked off of `master`. You can visualize this as follows:

Detecting a fork in the history using ranges



for both versions, this tells you that your history has diverged.

Filtering Merge Commits

By default, `git log` includes merge commits in its output. But, if your team has an always-merge policy (that is, you merge upstream changes into topic branches instead of rebasing the topic branch onto the upstream branch), you'll have a lot of extraneous merge commits in your project history.

You can prevent `git log` from displaying these merge commits by passing the `--no-merges` flag:

```
git log --no-merges
```

On the other hand, if you're *only* interested in the merge commits, you can use the `--merges` flag:

```
git log --merges
```

This returns all commits that have at least two parents.

Summary

You should now be fairly comfortable using `git log`'s advanced parameters to format its output and select which commits you want to display. This gives you the power to pull out exactly what you need from your

Tutorials

but remember that `git log` is often used in conjunction other Git commands. Once you've found the commit you're looking for, you typically pass it off to `git checkout`, `git revert`, or some other tool for manipulating your commit history. So, be sure to keep on learning about Git's advanced features.

Next up:

Git Hooks

START NEXT TUTORIAL

Powered By

Tutorials

Enter Your Email For Git News

Getting Started

Collaborating

Migrating to Git

Advanced Tips