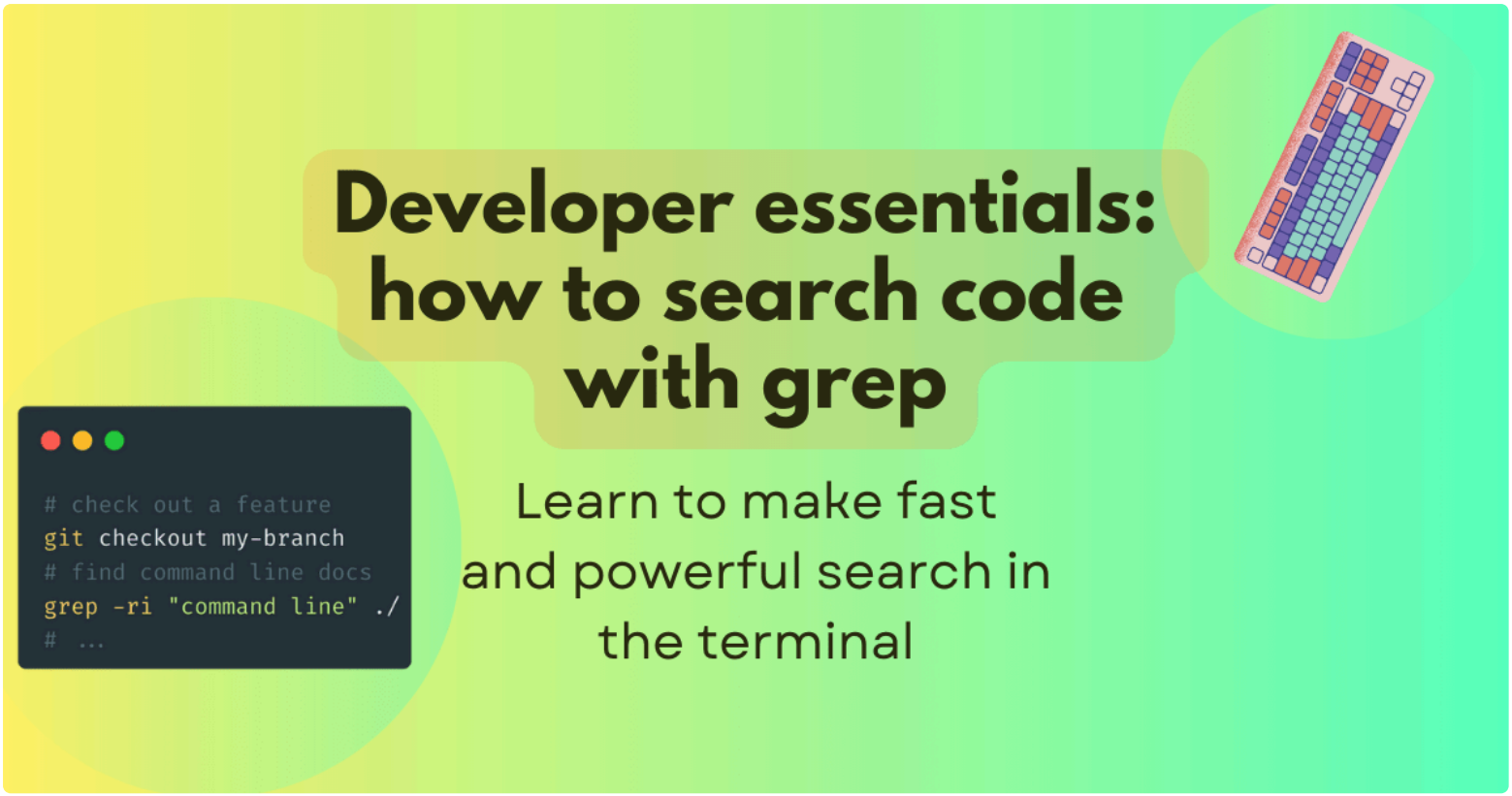

Developer essentials: How to search code using grep

Brian Smith 2023年7月3日 9 minute read

Wherever you are on your web development journey, you'll be searching for text or patterns in your code. You might want to search for a variable, where an error message originates, a CSS class, an image used in HTML or markdown source, logs from your application – the list is endless.

Searching through code and text is one of the most common tasks you'll be performing while you're building for the web. You might use your Integrated Development Environment (IDE) to search project files, your operating system's file search, or even code search through GitHub or another code hosting service. You'll quickly realize that you need an efficient tool to help you with the different kinds of searches you need to make; that's where grep comes in.

In this post, we'll see what grep is, what it can do, and why I think it's one of the most powerful command-line tools you will use when working with code. If you're unfamiliar with grep, this post will cover the basics, some common examples, including how I use it every day, and why I think it's an essential tool for developers. So let's dive in and find out how to put some grep in your step!

What is grep?

grep is a command-line tool that allows you to use regular expressions to search for patterns. The basic usage looks like this:

BASH

```
grep pattern file
```

There are different versions of grep available, with different options and features, but the core behaviour is the same for the most part. You can run `grep --version` to check which one you have installed:

BASH

```
grep --version
# (on macOS)
# grep (BSD grep, GNU compatible) 2.6.0-FreeBSD
```

Pro-tip: If you're ever stuck or if you need a reminder of how grep works, use the `man` (manual) command to see documentation and find out what options you can use:

BASH

```
man grep
# you can exit by pressing "q" 😁
```

Getting started with grep

Let's learn grep with some practical examples using the mdn/content GitHub repository . To follow along with the examples in this post, clone the `mdn/content` repository and `cd` into the directory (you can also download the repository as a zip file from GitHub if you don't use git):

BASH

```
git clone https://github.com/mdn/content.git
cd content
```

Once you're inside the content repository in your shell, you can search for some keywords and have a look at the output from grep. For example, here I am searching for the word "Communication" in the file `CONTRIBUTING.md`, the word "node" in the file `package.json`, and the phrase "Mozilla Community" in

BASH

```
grep "Communication" CONTRIBUTING.md
# [get in touch with us]: https://developer.mozilla.org/en-US/docs/MDN/Community/Communication_channels
grep "node" package.json
#    "node": ">=18.0.0"
grep "Mozilla Community" CODE_OF_CONDUCT.md
# [Mozilla Community Participation Guidelines]
(https://www.mozilla.org/about/governance/policies/participation/).
```

As you can see, each of the outputs from grep is the line or list of lines that matches the searched word or phrase. While you can run a command like `grep Communication CONTRIBUTING.md` and get the same output, it's recommended to wrap the pattern you want to search in double quotes to avoid any issues with whitespace or special characters being interpreted by the shell.

These examples are extremely useful to check things such as a node version or the URL we point the community to so they can get in touch with us. At this point, we know how to search for a specific word or group of words in a single file.

Searching recursively

A recursive search means searching for your pattern through multiple files, directories, and subdirectories. You will want to search for a pattern not only in a file but an entire tree. This is where grep starts to become really useful. To make a recursive search, use the `-r` flag before the pattern:

BASH

```
grep -r "my pattern" ./directory
```

In the context of searching in the content repository, we can search for a keyword in all markdown files:

BASH

```
grep -r "TOFU" ./files
# ./files/en-us/web/security/index.md:- TOFU# ./files/en-us/glossary/tofu/index.md:title: TOFU
# ./files/en-us/glossary/tofu/index.md:slug: Glossary/TOFU
```

The output shows us that the Web security page has a link to the glossary entry for TOFU, and the glossary entry has multiple matches for the pattern, as expected. I've left out a few other matches to make the output more readable, but the gist is that we can find occurrences of a pattern in multiple files and discover where it's used.

Excluding directories and files from search

In your development projects, you most likely have generated directories, such as `node_modules` or build directories such as `dist` or `build`, that you would like to ignore when doing a recursive search for a pattern. Some grep shell plugins can help with ignoring version control directories such as

`.git`, but it's always good to know how to explicitly ignore certain directories for search when you need to.

To ignore directories, you can use the `--exclude-dir` option. In this example, I am searching recursively for "cli-progress" starting from the current directory but excluding the `node_modules` directory from the search. The dot `.` at the end of the command is the path to search from, which in this case is the current directory:

BASH

```
grep -r --exclude-dir="node_modules" "cli-progress" .
# ./yarn.lock:    cli-progress "^3.12.0"
# ./yarn.lock:cli-progress@^3.12.0:
# ...
```

The output from this grep command is a good start, but there's a lot of extra matches in the `yarn.lock` file that are a little distracting. Let's ignore the `yarn.lock` file as well using the `--exclude` option:

BASH

```
grep -r --exclude-dir="node_modules" --exclude="yarn.lock" "cli-progress" .
# ./package.json:    "cli-progress": "^3.12.0",
# ./scripts/front-matter_linter.js:import cliProgress from "cli-progress";
```

This is much more useful as I have two relevant matches and I can tell at a glance that:

- We're using version `^3.12.0` of `cli-progress`

- We're importing `cli-progress` in `front-matter_linter.js` script as `cliProgress`

If I wanted to, I could continue my investigation by performing a search for `cliProgress` to see where and how it's used in the script.

Making a case-insensitive search

When searching through the content repository, I often need to check if a keyword occurs. However, because I don't know if it's the first word of a sentence, part of a URL, or follows a different casing convention, what pattern should I search for? Thankfully, this situation can be managed by doing a case-insensitive search using grep's `-i` flag:

BASH

```
grep -ri "github actions" ./files
# ./files/en-us/mdn/community/contributing/our_repositories/index.md:  A growing collection of reusable
GitHub Actions for use on MDN Web Docs repositories.
# ...
```

Ignoring binary files

If you're searching for a specific string and the project has binary files, you might run into some unexpected matches:

BASH

```
grep -ri "linux" ./files/en-us/web/http
# oh no:
```

```
# Binary file ./files/en-us/web/http/content_negotiation/httpnegotiation.png matches
```

This can happen by chance if you're looking for a short (two or three character) string and there's a lot of binary files such as images, PDFs, or other media files. Luckily, you can ignore binary files with the `--binary-files` option:

BASH

```
grep -ri --binary-files=without-match "linux" ./files/en-us/web/http
```

You can also use the `-I` flag to ignore binary files, but it's up to you to decide if you want to be explicit with the options or not. Do the flags below look readable to you or do you prefer to be explicit?

BASH

```
grep -riI "linux" ./files/en-us/web/http
```

Using regular expressions with grep

Of course, regular expressions are at the core of grep, so let's have a look at how we can use them to find variations of a pattern. Let's say I'm curious if we have pages where the titles start with numbers, but I'm not sure what the numbers are. Let's get an idea by matching the `\d` character class escape:

BASH

```
grep -r "title: \d\d\d" ./files/
# ./files//en-us/web/http/status/307/index.md:title: 307 Temporary Redirect
```

```
# ./files//en-us/web/http/status/300/index.md:title: 300 Multiple Choices
# ...
```

Now I know that we have a large number of pages that follow this pattern for HTTP status codes. Next up for me is to look for pages that use deprecated macros, specifically the `{{SpecName}}` and `{{spec2}}` macros.

I want to use a regex like `SpecName|spec2` that will match either `SpecName` or `spec2` using a disjunction. This isn't enabled on my version of grep by default, so I will need to enable extended regular expressions to use this with the `-E` flag:

BASH

```
grep -riE "SpecName|spec2" ./files
# files/en-us/mdn/writing_guidelines/howto/json_structured_data/index.md:The `{{SpecName}}` and
`{{Spec2}}` macros ...
```

This is great, the only places we found the macros are in our writing guidelines describing how to replace them. We've learned how to use more complex regular expressions and we've built a powerful search command combining these options:

- `-r` to search recursively
- `-i` for a case-insensitive search
- `-E` to enable extended regular expressions

Using unix pipes with grep

In unix land, there's a common convention of programs that perform one task very well. This allows us to build a kind of program that is a pipeline of commands, directing the output of one command as an input for another. You might see this referred to as "inter-process communication" but the idea is that you use the `|` character to pipe or chain commands together.

BASH

```
command1 | command2
```

In the interest of exploring pipes, why not use grep to find out how often I use grep? Let's use three commands together to find out how many times I've used `grep` in my shell history:

BASH

```
history | grep "grep" | wc -l
# 1164
```

To understand what's happening, let's look at the output of each command step by step. The `history` command outputs a list of all the commands I've run:

BASH

```
history
#    1   ls
#    2   mkdir ~/Code
```

```
#    ... 10000+ lines later
# 10098  man grep
```

Using `grep "grep"` , only output the lines that contain the string "grep":

BASH

```
history | grep "grep"
#    91  grep -r "prettier"
#    92  grep -r "inline-size" .
#    ...
# 10098  man grep
```

To get to the complete command, we're piping to `wc -l` (word count, lines) to count the number of lines in the output:

BASH

```
history | grep "grep" | wc -l
# 1164, magic ✨
```

We've built a small program that does the following using unix pipes:

- `history` : Outputs the history of commands I've run
- `grep "grep"` : Searches for the string "grep" in the output from the `history` command
- `wc -l` : Counts the number of lines in the output from the `grep "grep"` command

How fast is grep?

I search through `mdn/content` several times a day for occurrences of keywords, APIs, document titles or usage of a feature in code snippets. To search for a pattern using grep in approximately 13,500 files, it takes about 0.7 seconds:

BASH

```
time grep -r "\`\`\`plain" files/en-us/web/ | wc -l
     252
grep -r  0.43s user 0.29s system 99% cpu 0.730 total
wc -l  0.00s user 0.00s system 0% cpu 0.729 total
```

There are, of course, faster alternatives that you can use like ripgrep and ag, but grep is usually fast enough for most use cases. In the next section, I'll explain why I think it's worth learning grep over these alternatives.

Why should you learn grep?

If this post still hasn't convinced you that grep is the best code search tool you can spend your time learning, here's why I think it's worth it:

1. grep can handle searching a lot of text fairly quickly.
2. It's everywhere. It's probably available by default on servers or machines you SSH or log into.
3. Your shell history lets you reuse previous commands or use them as a starting point.

4. Beginners can start with simple use cases and advanced patterns can be learned over time.
5. You can direct output of other programs into grep and create powerful pipelines.
6. You learn regular expressions, which are useful in programming languages and other tools.

Summary

grep is so useful to me that it's muscle memory to type `grep -r` to prepare a search for a given pattern. I think that learning grep will be one of the best steps you can take for boosting productivity when writing code, debugging, inspecting new projects, or doing some quick analysis of a project.

If you think grep might be interesting for you, we've recently updated our regular expressions reference pages that will help you check patterns as you're searching. The recent blog post New reference pages on MDN for JavaScript regular expressions describes the updates we've made to the documentation to help you find what you're looking for and understand the syntax.

If you enjoyed this post, let us know in our community Discord or on GitHub to share your thoughts, ask questions, or just to say Hello! Let us know if this post has been helpful for you, if there are other ways you are using grep that I haven't mentioned here, or if I've missed something you think is important. Happy `/(hack|grep)ing/` ! 🖥️

Previous Post
Introducing AI Help (Beta): Your Companion for Web Development

Next Post
Reflections on AI Explain: A postmortem