grep

by example

Anton Zhiyanov

Table of contents

Preface

grep is the ultimate text search tool available on virtually all Linux machines. While there are now better alternatives (such as `ripgrep`), you will still often find yourself on a server where grep is the only search tool available. So it's nice to have a working knowledge of it.

That's why is I've created this step-by-step guide to grep operations. You can read it from start to finish to (hopefully) learn more about grep, or jump to a specific use case that interests you.

The guide is available in the following formats:

- Interactive online cookbook
- Bookmarkable playground
- PDF minibook

1. Basics

Basically, grep works like this:

- You give it a search pattern and a file.
- grep reads the file line by line, printing the lines that match the pattern and ignoring others.

Let's look at an example. We'll search the httpurr source code, which I've already downloaded to the `/opt/httpurr` directory.

Search in file

Let's find all occurrences of the word `codes` in `README.md`:

```
grep -n codes README.md
```

```
3:    <strong><i> >> HTTP status codes on speed dial << </i></str
30:* List the HTTP status codes:
54:* Filter the status codes by categories:
124:          Print HTTP status codes by category with --list;
131:          Print HTTP status codes
```

grep read the contents of `README.md`, and for each line that contained `codes`, grep printed it to the terminal.

grep also included the line number for each line, thanks to the `-n` (`--line-number`) option.

> Not all grep versions support the long option syntax (e.g. `--line-number`). If you get an error using the long version, try the short one (e.g. `-n`) — it may work fine.

Matches

grep uses *partial matches* by default:

```
grep -n descr README.md
```

```
81:* Display the description of a status code:
127:          Print the description of an HTTP status code
```

The word `description` matches the `descr` search pattern.

To search for *whole words* instead, use the `-w` (`--word-regexp`) option:

```
grep -n --word-regexp code README.md
```

```
81:* Display the description of a status code:
84: httpurr --code 410
94: Transfer Protocol (HTTP) 410 Gone client error response code
99: code should be used instead.
126:      -c, --code [status code]
127:          Print the description of an HTTP status code
```

grep found strings containing the word `code`, but not `codes`. Try removing `--word-regexp` and see how the results change.

> When using multiple short options, you can combine them like this: `grep -nw code README.md`. This gives exactly the same result as using the separate options (`-n -w`).

To search for *whole lines* instead of partial matches of whole words, use the `-x` (`--line-regexp`) option:

```
grep -n --line-regexp end httpurr.rb
```

```
47:end
```

Regular expressions

To make grep use regular expressions (*Perl-compatible regular expressions* in grep terminology), use the `-P` (`--perl-regexp`) option.

Let's find all lines with a word that contains `res` followed by other letters:

```
grep -Pn 'res\w+' README.md
```

```
94: Transfer Protocol (HTTP) 410 Gone client error response code
95: indicates that access to the target resource is no longer ava
152:of the rest.
```

`\w+` means "one or more word-like characters" (e.g. letters like `p` or `o`, but not punctuation like `.` or `!`), so `response`, `resource`, and `rest` all match.

> **Regular expression dialects in grep**
>
> Without `--perl-regexp`, grep treats the search pattern as something called a *basic regular expression*. While regular expressions are quite common in the software world, the *basic* dialect is really weird, so it's better not to use it at all.
>
> Another dialect supported by grep is extended regular expressions. You can use the `-E` (`--extended-regexp`) option to enable them. Extended regular expressions are *almost* like normal regular expressions, but not quite. So I wouldn't use them either.
>
> Some grep versions do not support `--perl-regexp`. For those, `--extended-regexp` is the best you can get.

Suppose we are only interested in 4 letter words starting with `res`:

```
grep -Pn 'res\w\b' README.md
```

```
152:of the rest.
```

`\b` means “word boundary” (e.g. a space, a punctuation character, or the end of a line), so `rest` matches, but `response` and `resource` don’t.

Finally, let’s search for 3-digit numbers (showing first 10 matches with `head`):

```
grep -Pn '\d\d\d' README.md | head
```

```
45: 100    Continue
46: 101    Switching Protocols
47: 102    Processing
48: 103    Early Hints
69: 200    OK
70: 201    Created
71: 202    Accepted
72: 203    Non-Authoritative Information
73: 204    No Content
74: 205    Reset Content
```

A full tutorial on regular expressions is beyond the scope of this guide, but grep’s “Perl-compatible” syntax is documented in the PCRE2 manual.

Fixed strings

What if we want to search for a *literal* string instead of a regular expression? Suppose we are interested in a word `code` followed by a dot:

```
grep -Pn 'code.' src/data.go | head
```

```
8:The HTTP 100 Continue informational status response code indica
14:status code in response before sending the body.
31:The HTTP 101 Switching Protocols response code indicates a pro
53:Deprecated: This status code is deprecated. When used, clients
56:The HTTP 102 Processing informational status response code ind
59:This status code is only sent if the server expects the reques
112:The HTTP 200 OK success status response code indicates that t
141:The HTTP 201 Created success status response code indicates t
149:The common use case of this status code is as the result of a
165:The HTTP 202 Accepted response status code indicates that the
```

Since `.` means “any character” in regular expressions, our pattern also matches `code` , `codes` and other cases we are not interested in.

To treat the pattern as a literal string, use the `-F` (`--fixed-strings`) option:

```
grep -Fn 'code.' src/data.go
```

```
197:to responses with any status code.
283:accessing web pages will never encounter this status code.
695:to an error code.
1027:happens, they will handle it as a generic 400 status code.
1051:web servers will normally not return this status code. But s
1418:then the server responds with the 510 status code.
```

Much better!

Multiple patterns

To search for multiple patterns, list them with the `-e` (`--regexp`) option. grep will output lines matching at least one of the specified patterns.

For example, search for `make` or `run`:

```
grep -En -e make -e run README.md
```

```
139:* Go to the root directory and run:
141:    make init
145:    make lint
149:    make test
```

> Unfortunately, grep can't use Perl-compatible regular expressions (`-P`) with multiple patterns. So we are stuck with the extended (`-E`) dialect.

If you have many patterns, it may be easier to put them in a file and point grep to it with `-f` (`--file`):

```
echo 'install' > /tmp/patterns.txt
echo 'make' >> /tmp/patterns.txt
echo 'run' >> /tmp/patterns.txt

grep -En --file=/tmp/patterns.txt README.md
```

```
13:* On MacOS, brew install:
17:     && brew install httpurr
20:* Or elsewhere, go install:
23: go install github.com/rednafi/httpurr/cmd/httpurr
139:* Go to the root directory and run:
141:    make init
145:    make lint
149:    make test
```

2. Recursive search

grep searches directories recursively when called with the `-r` (`--recursive`) option.

Search in directory

Let's find all unexported functions (they start with a lowercase letter):

```
grep -Pnr 'func [a-z]\w+' .
```

```
./cmd/httpurr/main.go:12:func main() {
./src/cli.go:16:func formatStatusText(text string) string {
./src/cli.go:21:func printHeader(w *tabwriter.Writer) {
./src/cli.go:35:func printStatusCodes(w *tabwriter.Writer, catego
./src/cli.go:105:func printStatusText(w *tabwriter.Writer, code s
```

This search returned matches from both the `cmd` and `src` directories. If you are only interested in `cmd`, specify it instead of `.`:

```
grep -Pnr 'func [a-z]\w+' cmd
```

```
cmd/httpurr/main.go:12:func main() {
```

To search multiple directories, list them all like this:

```
grep -Pnr 'func [a-z]\w+' cmd src
```

```
cmd/httpurr/main.go:12:func main() {
src/cli.go:16:func formatStatusText(text string) string {
src/cli.go:21:func printHeader(w *tabwriter.Writer) {
src/cli.go:35:func printStatusCodes(w *tabwriter.Writer, category
src/cli.go:105:func printStatusText(w *tabwriter.Writer, code str
```

File globs

Let's search for `httpurr`:

```
grep -Pnr --max-count=5 httpurr .
```

```
./README.md:2:    <h1>ᕦ httpurr</h1>
./README.md:16: brew tap rednafi/httpurr https://github.com/redna
./README.md:17:     && brew install httpurr
./README.md:23: go install github.com/rednafi/httpurr/cmd/httpurr
./README.md:33: httpurr --list
./cmd/httpurr/main.go:4:    "github.com/rednafi/httpurr/src"
./go.mod:1:module github.com/rednafi/httpurr
./httpurr.rb:7:  homepage "https://github.com/rednafi/httpurr"
./httpurr.rb:12:      url "https://github.com/rednafi/httpurr/rel
./httpurr.rb:16:        bin.install "httpurr"
./httpurr.rb:20:      url "https://github.com/rednafi/httpurr/rel
./httpurr.rb:24:        bin.install "httpurr"
./src/cli.go:24:    fmt.Fprintf(w, "\nᕦ httpurr\n")
./src/cli_test.go:64:   want := "\nᕦ httpurr\n═══════\n\n"
```

Note that I have limited the number of results per file to 5 with the `-m` (`--max-count`) option to keep the results readable in case there are many matches.

Quite a lot of results. Let's narrow it down by searching only in `.go` files:

```
grep -Pnr --include='*.go' httpurr .
```

```
./cmd/httpurr/main.go:4:    "github.com/rednafi/httpurr/src"
./src/cli.go:24:    fmt.Fprintf(w, "\nᕦ httpurr\n")
./src/cli_test.go:64:   want := "\nᕦ httpurr\n═══════\n\n"
```

The `--include` option (there is no short version) takes a *glob* (filename pattern), typically containing a fixed part (`.go` in our example) and a wildcard `*` ("anything but the path separator").

Another example — search in files named `http`-something:

```
grep -Pnr --include='http*' httpurr .
```

```
./httpurr.rb:7:  homepage "https://github.com/rednafi/httpurr"
./httpurr.rb:12:      url "https://github.com/rednafi/httpurr/rel
./httpurr.rb:16:        bin.install "httpurr"
./httpurr.rb:20:      url "https://github.com/rednafi/httpurr/rel
./httpurr.rb:24:        bin.install "httpurr"
./httpurr.rb:31:      url "https://github.com/rednafi/httpurr/rel
./httpurr.rb:35:        bin.install "httpurr"
./httpurr.rb:39:      url "https://github.com/rednafi/httpurr/rel
./httpurr.rb:43:        bin.install "httpurr"
```

To *negate* the glob, use the `--exclude` option. For example, search everywhere except the `.go` files:

```
grep -Pnr --exclude '*.go' def .
```

```
./.goreleaser.yml:1:# easer.yml file with some sensible defaults.
./httpurr.rb:15:      def install
./httpurr.rb:21:      sha256 "82adefd1222f6228636f2cda6518e0316f4
./httpurr.rb:23:      def install
./httpurr.rb:34:      def install
./httpurr.rb:42:      def install
```

To apply multiple filters, specify multiple glob options. For example, find all functions except those in test files:

```
grep -Pnr --include '*.go' --exclude '*_test.go' 'func ' .
```

```
./cmd/httpurr/main.go:12:func main() {
./src/cli.go:16:func formatStatusText(text string) string {
./src/cli.go:21:func printHeader(w *tabwriter.Writer) {
./src/cli.go:35:func printStatusCodes(w *tabwriter.Writer, catego
./src/cli.go:105:func printStatusText(w *tabwriter.Writer, code s
./src/cli.go:123:func Cli(w *tabwriter.Writer, version string, ex
```

Binary files

By default, grep does not ignore binary files:

```
grep -Pnr aha .
```

```
grep: ./data.bin: binary file matches
```

Most of the time, this is probably not what you want. If you're searching in a directory that might contain binary files, it's better to ignore them altogether with the `-I` (`--binary-files=without-match`) option:

```
grep -Pnr --binary-files=without-match aha .
```

```
(not found)
```

If for some reason you want grep to search binary files and print the actual matches (as it does with text files), use the `-a` (`--text`) option.

3. Search options

grep supports a couple of additional search options you may find handy.

Ignore case

Let's find all occurrences of the word `codes` in `README.md`:

```
grep -Pnr codes README.md
```

```
3:    <strong><i> >> HTTP status codes on speed dial << </i></str
30:* List the HTTP status codes:
54:* Filter the status codes by categories:
124:          Print HTTP status codes by category with --list;
131:          Print HTTP status codes
```

It returns `codes` matches, but not `Codes` because grep is case-sensitive by default. To change this, use `-i` (`--ignore-case`):

```
grep -Pnr --ignore-case codes README.md
```

```
3:    <strong><i> >> HTTP status codes on speed dial << </i></str
30:* List the HTTP status codes:
40: Status Codes
54:* Filter the status codes by categories:
64: Status Codes
124:          Print HTTP status codes by category with --list;
131:          Print HTTP status codes
```

Inverse matching

To find lines that *do not* contain the pattern, use `-v` (`--invert-match`). For example, find the non-empty lines without the `@` symbol:

```
grep -Enr --invert-match -e '@' -e '^$' Makefile
```

```
1:.PHONY: lint
2:lint:
8:.PHONY: lint-check
9:lint-check:
14:.PHONY: test
15:test:
20:.PHONY: clean
21:clean:
27:.PHONY: init
28:init:
```

4. Output options

grep supports a number of additional output options you may find handy.

Count matches

To count the number of matched lines (per file), use `-c` (`--count`). For example, count the number of functions in each `.go` file:

```
grep -Pnr --count --include '*.go' 'func ' .
```

```
./cmd/httpurr/main.go:1
./src/cli.go:5
./src/cli_test.go:10
./src/data_test.go:2
```

Note that `--count` counts the number of *lines*, not the number of matches. For example, there are 6 words `string` in `src/cli.go`, but two of them are on the same line, so `--count` reports 5:

```
grep -nrw --count string src/cli.go
```

```
5
```

Limit matches

To limit the number of matching lines per file, use the `-m` (`--max-count`) option:

```
grep -Pnrw --max-count=5 func .
```

```
./cmd/httpurr/main.go:12:func main() {
./src/cli.go:16:func formatStatusText(text string) string {
./src/cli.go:21:func printHeader(w *tabwriter.Writer) {
./src/cli.go:35:func printStatusCodes(w *tabwriter.Writer, catego
./src/cli.go:105:func printStatusText(w *tabwriter.Writer, code s
./src/cli.go:123:func Cli(w *tabwriter.Writer, version string, ex
./src/cli_test.go:15:func TestFormatStatusText(t *testing.T) {
./src/cli_test.go:54:func TestPrintHeader(t *testing.T) {
./src/cli_test.go:71:func TestPrintStatusCodes(t *testing.T) {
./src/cli_test.go:159:      t.Run(want, func(t *testing.T) {
./src/cli_test.go:168:func TestPrintStatusText(t *testing.T) {
./src/data_test.go:9:func TestStatusCodes(t *testing.T) {
./src/data_test.go:99:func TestStatusCodeMap(t *testing.T) {
```

With `--max-count=N`, grep stops searching the file after finding the first N matching lines (or non-matching lines if used with `--invert-match`).

Show matches only

By default, grep prints the entire line containing the match. To show only the matching part, use `-o` (`--only-matching`).

Suppose we want to find functions named `print`-something:

```
grep -Pnr --only-matching --include '*.go' 'func print\w+' .
```

```
./src/cli.go:21:func printHeader
./src/cli.go:35:func printStatusCodes
./src/cli.go:105:func printStatusText
```

The results are much cleaner than without `--only-matching` (try removing the option in the above command and see for yourself).

Show files only

If there are too many matches, you may prefer to show only the files where the matches occurred. Use `-l` (`--files-with-matches`) to do this:

```
grep -Pnr --files-with-matches 'httpurr' .
```

```
./README.md
./cmd/httpurr/main.go
./go.mod
./httpurr.rb
./src/cli.go
./src/cli_test.go
```

Show context

Let's search for GitHub action jobs:

```
grep -Pnr 'jobs:' .github/workflows
```

```
.github/workflows/automerge.yml:8:jobs:
.github/workflows/lint.yml:11:jobs:
.github/workflows/release.yml:10:jobs:
.github/workflows/test.yml:11:jobs:
```

These results are kind of useless, because they don't return the actual job name (which is on the next line after `jobs`). To fix this, let's use `-C` (`--context`), which shows `N` lines around each match:

```
grep -Pnr --context=1 'jobs:' .github/workflows
```

```
.github/workflows/automerge.yml-7-
.github/workflows/automerge.yml:8:jobs:
.github/workflows/automerge.yml-9-  dependabot:
--
.github/workflows/lint.yml-10-
.github/workflows/lint.yml:11:jobs:
.github/workflows/lint.yml-12-  golangci:
--
.github/workflows/release.yml-9-
.github/workflows/release.yml:10:jobs:
.github/workflows/release.yml-11-  goreleaser:
--
.github/workflows/test.yml-10-
.github/workflows/test.yml:11:jobs:
.github/workflows/test.yml-12-  test:
```

It might be even better to show only the *next* line after the match, since we are not interested in the previous one. Use `-A` (`--after-context`) for this:

```
grep -Pnr --after-context=1 'jobs:' .github/workflows
```

```
.github/workflows/automerge.yml:8:jobs:
.github/workflows/automerge.yml-9-  dependabot:
--
.github/workflows/lint.yml:11:jobs:
.github/workflows/lint.yml-12-  golangci:
--
.github/workflows/release.yml:10:jobs:
.github/workflows/release.yml-11-  goreleaser:
--
.github/workflows/test.yml:11:jobs:
.github/workflows/test.yml-12-  test:
```

> There is also `-B` (`--before-context`) for showing N lines *before* the match.

Nice!

Silent mode

Sometimes you just want to know if a file contains a certain string; you don't care about the number or positions of the matches.

To make grep quit immediately after the first match and not print anything, use the `-q` (`--quiet` or `--silent`) option. Use the return code (`$?`) to see if grep found anything (0 — found, 1 — not found):

```
grep -Pnrw --quiet main cmd/httpurr/main.go
if [ $? = "0" ]; then echo "found!"; else echo "not found"; fi
```

```
found!
```

Try changing the search pattern from `main` to `Main` and see how the results change.

When searching in multiple files with `--quiet`, grep stops after the first match in any file and does not check other files:

```
grep -Pnrw --quiet main .
if [ $? = "0" ]; then echo "found!"; else echo "not found"; fi
```

```
found!
```

Colors

To highlight matches and line numbers, use the `--color=always` option:

```
grep -Pnr --color=always codes README.md
```

```
3:    <strong><i> >> HTTP status codes on speed dial << </i></str
30:* List the HTTP status codes:
54:* Filter the status codes by categories:
124:        Print HTTP status codes by category with --list;
131:        Print HTTP status codes
```

Use `--color=auto` to let grep decide whether to use colors based on your terminal. Use `--color=never` to force no-color mode.

Final thoughts

That's it! We've covered just about everything grep can do. Unfortunately, it doesn't support replacing text, reading options from a configuration file, or other fancy features provided by grep alternatives like `ack` or `ripgrep`. But grep is still quite powerful, as you can probably see now.

If you find this book useful — please spread the word and subscribe to my other projects at antonz.org/subscribe.

Have fun grepping!