UNIX and Linux

GREP

elflord@pegasus.rutgers.edu

Tutorials

Useful information mostly written by me, the conspicuous exception being the bash manpage ...

Intro to Unix
UNIX command summary
grep tutorial powerful search tool
sed tutorial scripts to edit text files
Autofs in Linux automatically mounting removable media
procmail tutorial well known email filter
bash manpage the man page for the bash shell. Warning: this is long (~210k)

Contents

Why grep ?

Back to top

grep is not only one of the most useful commands, but also, mastery of *grep* opens the gates to mastery of other tools such as *awk* , *sed* and *perl* .

So what does it do ?

Back to top

grep basically searches. More precisely,

> `grep foo file` returns all the lines that *contain* a string matching the expression "foo" in the file "file".

For now, we will just think of an expression as a string. So *grep* returns all matching lines that contain foo as a substring.

Another way of using *grep* is to have it accept data through `STDIN`. instead of having it search a file. For example,

> `ls |grep blah` lists all files in the current directory whose names contain the string "blah"

Compatibility Notes

Back to top

This tutorial is based on the *GNU* version of grep. It is recommended that you use this version. To use it, firstly, it needs to be installed on your system. Secondly, your ***PATH*** needs to be set so that GNU grep is used in preference to the standard version.

Wildcards For Grep

Back to top

The Basics: Wildcards for grep

The Wildcard Character

So the first question that probably comes to mind is something like "does this *grep thing* support wildcards ? And the answer is better than yes. In fact saying that *grep* supports wildcards is a big understatement. *grep* uses *regular expressions* which go a few steps beyond wildcards. But we will start with wildcards. The canonical wildcard character is the dot "." Here is an example :

```
>cat file

big
bad bug
bag
bigger
boogy

>grep b.g file

big
bad bug
bag
bigger
```

notice that boogy didn't match, since the "." matches exactly one character.

The repetition character

To match repetitions of a character, we use the star, which works in the following way:

> *the expression consisting of a character followed by a star matches any number (possibly zero) of repetitions of that character. In particular, the expression ".*" matches any string, and hence acts as a "wildcard".*

To illustrate, we show some examples:

Examples: Wildcards

The File for These Examples	*Wildcards #1*	*Wildcards #2*	*repetition*
`>cat file`	`>grep "b.*g" file`	`>grep "b.*g." file`	`>grep "ggg*" file`
`big`	`big`	`bigger`	`bigger`
`bad bug`	`bad bug`	`boogy`	
`bag`	`bag`		
`bigger`	`bigger`		
`boogy`	`boogy`		

Read the repetion example carefully, and pay careful attention to the fact that the "*" in grep patterns denotes *repetition*. It does *not* behave as a wildcard in regular expression syntax (as it is in UNIX or DOS glob patterns). Recall that the pattern ".*" behaves as a wildcard (because .* means "repeat any character any number of times). The pattern "g*" matches the string "", "g", "gg", etc. Likewise, "gg*" matches "g", "gg", "ggg", so "ggg*" matches "gg", "ggg", "gggg", etc.

Taking it Further - Regular Expressions

Back to top The wildcards are a start, but the idea could be taken further. For example, suppose we want an expression that matches `Frederic Smith` or `Fred Smith`. In other words, the letters `eric` are "optional".

First, we introduce the concept of an "escaped" character.

> *An escaped character is a character preceded by a backslash. The preceding backslash does one of the following:
> (a) removes an implied special meaning from a character (b) adds special meaning to a "non-special" character*

Examples

To search for a line containing text `hello.gif`, the correct command is

```
grep 'hello\.gif' file
```

since `grep 'hello.gif' file` will match lines containing `hello-gif` , `hello1gif` , `helloagif` , etc.

Now we move on to grouping expressions, in order to find a way of making an expression to match `Fred` or `Frederic`

First, we start with the ? operator.

> *an expression consisting of a character followed by an escaped question mark matches one or zero instances of that character.*

Example

`bugg\?y` matches all of the following: `bugy` , `buggy` but not `bugggy` We move on to "grouping" expressions. In our example, we want to make the string "ederic" following "Fred" optional, we don't just want one optional character.

> *An expression surrounded by "escaped" parentheses is treated by a single character.*

Examples

`Fred\(eric\)\? Smith` matches `Fred Smith` or `Frederic Smith`
`\(abc\)*` matches `abc` , `abcabcabc` etc. (i.e. , any number of repetitions of the string `abc` , including the empty string.) Note that we have to be careful when our expressions contain white spaces or stars. When this happens, we need to enclose them in quotes so that the shell does not mis-interpret the command, because the shell will parse whitespace-separated strings as multiple arguments, and will expand an unquoted * to a glob pattern. So to use our example above, we would need to type

```
grep "Fred\(eric\)\? Smith" file
```

We now mention several other useful operators.

More on Regular Expressions

Back to top

Matching a list of characters

Back to top
To match a selection of characters, use [].

Example

> `[Hh]ello` matches lines containing `hello` or `Hello`

Ranges of characters are also permitted.

Example

> `[0-3]` is the same as `[0123]`
> `[a-k]` is the same as `[abcdefghijk]`
> `[A-C]` is the same as `[ABC]`
> `[A-Ca-k]` is the same as
> `[ABCabcdefghijk]`

There are also some alternate forms :

> `[[:alpha:]]` is the same as `[a-zA-Z]`
> `[[:upper:]]` is the same as `[A-Z]`
> `[[:lower:]]` is the same as `[a-z]`
> `[[:digit:]]` is the same as `[0-9]`
> `[[:alnum:]]` is the same as `[0-9a-zA-Z]`
> `[[:space:]]` matches any white space including tabs

These alternate forms such as `[[:digit:]]` are preferable to the direct method `[0-9]`

The [] may be used to search for *non-matches*. This is done by putting a carat ^ as the first character inside the square brackets.

Example

`grep "([^()]*)a" file` returns any line containing a pair of parentheses that are innermost and are followed by the letter "a". So it matches these lines

```
(hello)a
(aksjdhaksj d ka)a
```

But not this

```
x=(y+2(x+1))a
```

Matching a Specific Number Of Repetitions of a Pattern

Back to top

Suppose you want to match a specific number of repetitions of a pattern. A good example is phone numbers. You could search for a 7 digit phone number like this:

```
grep "[[:digit:]]\{3\}[ -]\?[[:digit:]]\{4\}" file
```

This matches phone numbers, possibly containing a dash or whitespace in the middle.

Nailing it Down to Start of the Line and End of the Line

Back to top
Here's the deal. Suppose you want to search for lines containing a line consisting of white space, then the word hello, then the end of the line. Let us start with an example.

```
>cat file

        hello
hello world
        hhello

>grep hello file

        hello
hello world
        hhello
```

This is *not* what we wanted. So what went wrong ? The problem is that grep searches for lines containing the string "hello" , and all the lines specified contain this. To get around this problem, we introduce the end and beginning of line characters

The $ character matches the end of the line. The ^ character matches the beginning of the line.

Examples

returning to our previous example,

```
grep "^[[:space:]]*hello[[:space:]]*$" file
```

does what we want (only returns one line) Another example:
`grep "^From.*mscharmi" /var/spool/mail/elflord` searches my inbox for headers from a particular person. This kind of regular expression is extremely useful, and mail filters such as procmail use it all the tims.

This or That: matching one of two strings

Back to top

The expression consisting of two expressions seperated by the or operator `\|` *matches lines containing either of those two expressions.*

Note that you *MUST* enclose this inside single or double quotes.

Example

`grep "cat\|dog" file` matches lines containing the word "cat" or the word "dog"
`grep "I am a \(cat\|dog\)"` matches lines containing the string "I am a cat" or the string "I am a dog".

Backpedalling and Backreferences

Back to top
Suppose you want to search for a string which contains a certain substring in more than one place. An example is the heading tag in HTML. Suppose I wanted to search for `<H1>some string</H1>` . This is easy enough to do. But suppose I wanted to do the same but allow `H2 H3 H4 H5 H6` in place of `H1`. The expression `<H[1-6]>.*</H[1-6]>` is not good enough since it matches `<H1>Hello world</H3>` but we want the opening tag to match the closing one. To do this, we use a *backreference*

> *The expression \n where n is a number, matches the contents of the n'th set of parentheses in the expression*

Woah, this really needs an example!

Examples

`<H\([1-6]\).*</H\1>` matches what we were trying to match before.
`"Mr \(dog\|cat\) came home to Mrs \1 and they went to visit Mr \(dog\|cat\) and Mrs \2 to discuss the meaning of life` matches ... well I'm sure you can work it out. the idea is that the cats and dogs should match up in such a way that it makes sense.

Some Crucial Details: Special Characters and Quotes

Back to top

Special Characters

Back to top
Here, we outline the special characters for grep. Note that in egrep (which uses extended regular expressions), which actually are no more functional than standard regular expressions if you use GNU grep) , the list of special characters increases (| in grep is the same as \| egrep and vice versa, there are also other differences. Check the man page for details) The following characters are considered special and need to be "escaped":

```
? \ . [ ] ^ $
```

Note that a $ sign loses its meaning if characters follow it (I think) and the carat ^ loses its meaning if other characters precede it.

Square brackets behave a little differently. The rules for square brackets go as follows:

- A closing square bracket loses its special meaning if placed first in a list. for example `[]12]` matches] , 1, or 2.
- A dash - loses it's usual meaning inside lists if it is placed last.
- A carat ^ loses it's special meaning if it is not placed first
- Most special characters lose their meaning inside square brackets

Quotes

Back to top
Firstly, single quotes are the safest to use, because they protect your regular expression from the shell. For example, `grep "!" file` will often produce an error (since the shell thinks that "!" is referring to the shell command history) while `grep '!' file` will not.

When should you use single quotes ? the answer is this: if you want to use shell variables, you need double quotes. For example,

```
grep "$HOME" file
```

searches file for the name of your home directory, while

```
grep '$HOME' file
```

searches for the string $HOME

Extended Regular Expression Syntax

Back to top

We now discuss egrep syntax as opposed to grep syntax. Ironically, despite the origin of the name (extended), egrep actually has *less* functionality as it is designed for compatibility with the traditional egrep. A better way to do an extended "grep" is to use `grep -E` which uses extended regular expression syntax without loss of functionality.

grep	grep -E	Available for egrep?
`a\+`	`a+`	yes
`a\?`	`a?`	yes
`expression1|expression2`	`expression1|expression2?`	yes
`\(expression\)`	`(expression1)`	yes
`\{m,n\}`	`{m,n}`	no

`\{,n\}`	`{,n}`	no
`\{m,}`	`{m,}`	no
`\{m}`	`{m}`	no