Even Wilder Wildcards

By Jerry Peek

In last December's column, "Wildcards Gone… Wild" (available online at http://www.linux-mag.com/2003-12/power_01.html), we looked at ways to match groups of files in a hurry, with a minimum of typing, by using *shell wildcards*. Last month, we saw some of the wilder things that the Z shell, *zsh*, can do to make computing easier. This month we'll mix those two topics and look at some of the most powerful *zsh* wildcards and similar, related features in other shells.

Even if you aren't interested in wildcards, you might want to read on. We'll dig deep into twisty corners of the shell. For instance, we'll see how to set a shell option temporarily and we'll use an alias as a sort of "preprocessor" for a shell function.

Most examples here are specific to one shell. If you don't normally use that shell, remember that there's no need to switch permanently. As we said in last month's Power Tip, "Changing Shells Temporarily" (available online at http://www.linux-mag.com/2004-02/power_03.html), you can drop into another shell temporarily. You can also leave a shell process suspended and bring it to the foreground whenever you need its extra power — power like the examples you'll see below. The December 2002 column, "Using Power Wisely," explains suspending shells. (It's online at http://www.linux-mag.com/2002-12/ power_01.html.)

Anything But…

It's easy to make wildcards match a lot. In the December column, we saw two ways to match "anything but." One is a character range (inside square brackets) that starts with `^` or `!`. For instance, `rm [^0-9]*` removes all filenames that *don't* start with a number. Many shells also have the "not" operator, like `!(`*`pattern`*`)`, which matches anything but *pattern*. So, `cp !(backup) backup` would copy everything in the current directory into the subdirectory named *backup*, except for *backup* itself.

The Z shell can also use a tilde (`~`), the *exclusion specifier*, as an "anything but" operator. (You have to set the `extended_glob` option first.) For instance, *`pat1~pat2`* matches *pat1* except for *pat2*. This has at least two advantages over the other methods: *pat2* can contain `/`, so you can exclude pathnames (not just filenames), and you can also exclude more than one pattern by using more than one tilde. For example, the expression *`pat1~pat2~pat3`* matches *pat1*, but not *pat2* or *pat3*.

Let's say you have a directory tree full of various XML files. You want to format all files whose names start with *009* and one more digit, like *0090.xml*, *0093.xml*, and so on. However, you don't want any files in subdirectories named *tmp*. The Z shell's built-in `**` recursive-matching wildcard and pattern-matching operators can do the job in a flash. *Listing One* shows the setup, using the *ls* command so you can see which files match the wildcards. (In real life, you'd use an XML formatter program instead of *ls*.)

The exclusion specifier `~` tells *zsh* not to expand any pathnames that match the wildcard pattern `*/tmp/*` — that is, pathnames containing the string `/tmp/`.

Only These, Not Those

Standard wildcards simply match the names of files (including directories, symbolic links, etc., which are actually files in the Linux filesystem). In other words, they don't look in the file's inode, which contains file attribute information.

Let's say your current directory has a mixture of files and subdirectories, and you want to print all the files but not the subdirectories themselves. (A directory file is full of gibberish.) A Z shell *glob qualifier* lets you write a wildcard expression that matches only files — or executable files, or files owned by a certain user, or files accessed *n* days ago, and so on.

The qualifier is written as a list in parentheses at the end of a wildcard expression. The *zshexpn man* page lists these. *Table One* has some examples.

Here's a simple example, listing all symbolic links in the current directory:

```
zsh% ls -l *(@)
lrwxrwxrwx    1 root        11 Mar   8 12:12
  rmt -> ../sbin/rmt
zsh%
```

LISTING ONE: Using the *zsh exclusion specifier* (~)

```
zsh% ls **/009?.xml
alfa/0092.xml       alfa/tmp/0097.xml
alfa/0093.xml       beta/0096.xml
alfa/0095.xml       beta/0098.xml
alfa/0097.xml       beta/0099.xml
alfa/tmp/0092.xml   beta/tmp/0096.xml
alfa/tmp/0093.xml   beta/tmp/0098.xml
alfa/tmp/0095.xml   beta/tmp/0099.xml
zsh% setopt extended_glob
zsh% ls **/009?.xml~*/tmp/*
alfa/0092.xml   alfa/0097.xml   beta/0099.xml
alfa/0093.xml   beta/0096.xml
alfa/0095.xml   beta/0098.xml
```

TABLE ONE: Some *zsh glob qualifiers*

QUALIFIER	DESCRIPTION
/	Directories
.	Plain files
@	Symbolic links
*	Executable plain files (mode 0100)
r	Owner-readable (mode 0400)
w	Owner-writable (mode 0200)
x	Owner-executable (mode 0100)
I	Group-writable (mode 0020)
E	Group-executable (mode 0010)
X	World-executable (mode 0001)
^	Negate all following qualifiers

If you use more than one qualifier, they're combined with "and" (that is, an object only matches if it matches *all* of the qualifiers.) Let's use some glob qualifiers to test whether there are any world-executable files in the directory:

```
zsh% ls -l *(.X)
zsh: no matches found: *(.X)
```

In that example, *zsh* didn't even run *ls*. There were no world-executable files, and the shell simply told you so.

You can use more than one list of qualifiers: just join the lists with a comma (`,`). For instance, to remove execute permission from any files in the *alfa* subdirectory that have "group" execute permission or "world" execute permission:

```
zsh% chmod go-x alfa/*(.E,.X)
```

When you combine qualifiers with the recursive wildcard operator `**`, you can do a lot of the work you'd do with the *find* utility, but in a much more concise syntax.

For example, here are two ways to add group-write permission to all subdirectories that don't already have it. The *zsh* command uses the "negate" qualifier, caret (`^`), to test for no group write permission:

```
$ find * -type d ! -perm -0020 \
  -exec chmod g+w '{}' \;

zsh% chmod g+w **/(/^I)
```

And that's not all! If you end the list of qualifiers with a colon (`:`), you can follow the colon with string editing operators to edit the expanded filenames. Here's a more advanced example from the *zshexpn man* page that combines attribute qualifiers with the string editing operator:

```
zsh% echo /tmp/foo*(u0^@:t)
```

That searches for all filenames starting with */tmp/foo* (`/tmp/foo*`, the wildcard expression) owned by root (`u0`, UID 0, which belongs to root) except symbolic links (`^@`, the negated qualifier for symlinks). It outputs the basename (the filename only, with no leading pathname) of those files by using `:t`, the string editing operator for "basename."

This is admittedly not simple to remember at first. Like so much of the Z shell, the syntax takes practice to learn! But if you do a lot of work with the filesystem, this shell's sometimes-steep learning curve is well worth the climb.

Sorting by Number

Your shell probably has options that let you control how wildcards do their job. Let's take a quick look at some Z shell options that control wildcard matching (called *globbing*). Your shell may have options like these; its *man* page should tell you.

The Linux filesystem is used for storing files, of course. But it also can be used as a kind of database. If you put each chunk of data in a separate file and give the files meaningful names, you can access the data in different ways by reading or sorting the files in different orders. (By the way, you can also use directories full of hard or symbolic links to make different "views" of those same data files. But that's getting off the topic.)

By default, all shells sort the globbed list of filenames in lexicographic order. This may be fine if your filenames are words, but if the names are numbers, that can be a problem.

For example, the MH email system, and its successor *nmh*, use the filesystem as a message database.

Each message is stored in a separate file, where the filename is the message number. (Message number 1 is in a file named *1*, and so on.) This makes it easy to write programs to read and manipulate messages.

For instance, here's how the shell's *for* loop can step through all of the messages in a directory (what MH calls a "folder"):

```
for msg in [1-9]*
do
   operate on message $msg ...
done
```

Wildcards aren't great with these numeric filenames, though, because a folder with 101 messages is sorted *1*, *10*, *100*, and so on. So, after message 1, the *for* loop operates on message 10 instead of what you want, which is 1, 2, 3. If you aren't using *zsh*, here are two workarounds:

```
for msg in [1-9] [1-9][0-9] [1-9][0-9][0-9]

for msg in 'ls [1-9]* | sort -g'
```

The first workaround matches all one-character filenames (*1*, *2*, …) followed by all two-character filenames (*10*, *11*, …), and so on. The second uses *ls* to list the messages with one filename per line, GNU `sort -g` to sort it in numeric order, and command substitution (in backquotes) to put the sorted filenames on to the command line. (Other versions of *sort* use `-n` instead of `-g`.) But *zsh* gives you a better way: simply set the `numeric_glob_sort` option:

```
setopt numeric_glob_sort
for msg in [1-9]*
...
```

When you want normal lexicographic sorting, run `unsetopt numeric_glob_sort`.

Setting Shell Options Temporarily

Here are two ways to set a shell option just when you need it. One way is by running a command line in a subshell. Changes to variable settings, current directory, and other attributes that you make inside a subshell don't affect the parent shell, so you can set an option inside a subshell without bothering to unset it. (When the subshell finishes, its attributes are all forgotten.)

All shells support the subshell operators `( )` — a pair of parentheses with *no* `$` before them. For example, let's set the *bash* option *nocaseglob* before trying to match all JPEG files. When the wildcard matching happens within the subshell, *bash* passes a case-insensitive list of JPEG filenames to *ls*. Next, for comparison, we run the same command without the temporary *nocaseglob:*

```
bash$ (shopt -s nocaseglob; ls *.jpg)
a.jpg  b.JPG  c.Jpg
bash$ ls *.jpg
a.jpg
```

If you want to do something like that often, typing the subshell operators and the option name can be a pain. But a shell function or alias can run a command line with whatever options you'd like to set temporarily, then reset the options after the command line runs.

For instance, let's set *bash* so that typing `ncg` before a command line will set the *nocaseglob* option for that one command. This can be more useful than a subshell because it lets your command change shell attributes — like the current directory — that a subshell couldn't. For example:

```
$ ncg ls *.jpg
a.jpg  b.JPG  c.Jpg
$ ls *.jpg
a.jpg
```

POWER TIP: Pathnames that only match directories

Let's say you want to move *afile* into the directory */a/b/backups*. But you type `mv afile /a/b/backupz`. To *mv*, that isn't an error. *mv* will rename *afile* to *backupz* and leave it in the (wrong) directory */a/b*. What to do?

Each directory has an entry named `.` — a single dot — which is a link to the directory itself. A pathname ending with `/.` can *only* match a directory. So, typing `mv afile /a/b/backupz/.` would have avoided the previous problem. (*mv* would complain that */a/b/backupz/* doesn't exist.)

If you're using the Z shell, a glob qualifier (which we saw earlier) can make sure a wildcard matches only a directory. But this tip works in all shells — in fact, it doesn't depend on the shell at all.

Typing the two characters `/.` after a directory name can be handy when you want to be sure you get *only* a directory.

Here's the setup:

```
alias ncg='shopt -s nocaseglob; ncgf'

ncgf() {
  shopt -u nocaseglob
  "$@"
}
```

We're using an alias named `ncg` that calls a function called `ncgf`. Why the two steps? A shell function is executed *after* the command line has been interpreted, but we want to set an option *before* the wildcards have been expanded. An alias does simple text substitution — and it's done before almost all other command-line interpretation. So when you type `ncg ls *.jpg`, the shell executes `shopt -s nocaseglob` before it expands the wildcard.

Next, the wildcard is expanded, and finally, the shell passes the expanded command line to the `ncgf()` function.

Within `ncgf()`, we first unset the *nocaseglob* option — before we run the command line.

Why? Because `"$@"` (which expands into the command line) is last in the function, the function returns the exit status from that command line. Also, if there's any syntax error in the command line, the `nocaseglob` option will already have been unset before the function aborts. (Thanks to Oliver Kiddle for this helpful tip.)

This is a dark corner of shell magic that's best used carefully. Different shells may interpret these differently. Still, this shows why understanding the shell's operation in depth can help you do just what you need to.

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for over 20 years. He's happy to hear from readers at jpeek@jpeek.com.