Speaking UNIX: Extreme shell makeover

Skill Level: Intermediate

Martin Streicher (martin.streicher@gmail.com)
Software Developer
Pixel, Byte, and Comma

25 May 2011

Break out the caffeine, elbow grease, and your text editor. It's time to turn your UNIX shell into a hot rod. It's time for an extreme shell makeover.

No matter what car you drive, you have to appreciate the hot rod. There's nothing quite like a 1969 Dodge Charger with five-spoke mag wheels, wide tires, a midnight black finish, lots of chrome, and a snarling big block. It's a masterpiece—the car Rodin's *The Thinker* would drive. A classic.

The UNIX® shell is a classic, too (sans the big fenders). But with a little elbow grease, wrench time, and after-market modifications, it can look and perform like a hot rod, too. You might call this "Top Gear-head" or "Extreme Shell Makeover." I call it cool!

It's got moving parts

Beneath that blinking prompt, your shell—be it Bash, Z shell, or something else—has lots of moving parts that you can augment, tune, and tweak. Here's a summary (as many of these have been discussed in detail on prior columns):

- An *alias* can abbreviate command names and command-line phrases. For example, creating `alias ll 'ls -hlt'` reduces a commonly used command line to just two letters—less to type and less to remember. Most often, an alias abbreviates literals. However, much like a shell script or a shell function, an alias can also accept parameters. The command `alias print 'lpr -h -Pps5 \!*'` replaces `\!*` with command-line arguments. Thus, typing `print manual.ps schematic.ps form.ps`

is expanded to `lpr -h -Pps5 manual.ps schematic.ps form.ps`.

- *Environment variables* persist settings across commands, even commands that spawn new processes. Some environment variables have special meaning because of convention: PRINTER is recognized by commands and applications as your preferred output device, while EDITOR and PAGER indicate which text editor and viewer to use for modifying and displaying text, respectively. Other environment variables are germane to individual utilities. One example is PS1. It tells your shell what to render for your initial prompt. Another workhorse environment variable is the shell's PATH, which enumerates directories to search for an executable. (To find which environment variables are recognized in a specific application or command, look for a section called "Environment variables" in its man page.)
- *Functions* are callable from the command line, too, and fill the gap between an alias and a full-blown script. For example, if you need to manipulate arguments or apply logic, an alias is likely insufficient; yet, an entire script may be overkill. Functions can also be combined to achieve cumulative effects. You need a modicum of programming savvy to create functions, but nothing onerous.
- *Shell options* control the behavior of the shell. Shell options typically vary from shell to shell, and a rich shell, such as Bash or Z shell, can have hundreds of tunable parameters. (The Z shell has a dedicated man page, *zshoptions*, just to document configurability.) For example, if you enable the Z shell option `pushd_ignore_dups`, `pushd` won't push a directory onto the directory stack if it's already there. You can literally spend hours exploring options. Once you find a combination you want to keep, type `set` (Bash) or `setopt` and redirect the output to a file. You can copy some or all of the captured settings to a startup file to recreate your workspace each time you open a shell.

In addition to these gears and levers, you can change how your shell *looks*. The nondescript dollar sign (`$`) prompt can sport colors, reflect your current working directory, and even show the weather. If you can capture bits of information with a command, you can probably display that data at the prompt.

Like pinstripes

Much like other operations in the shell, an environment variable controls what's drawn each time the shell presents a prompt. The variable PS1, or "prompt string level 1," is interpreted when rendered much like the command line itself. PS1 can contain other shell and environment variables, in-place command evaluation (via backquotes), and specialized literals. Here's an initial example:

```
$ export PS1="\u@\h \w >"
strike@nostromo ~ > whoami; hostname; pwd
strike
nostromo
/home/strike
strike@nostromo ~ >
```

Initially, the prompt was the simple `$`. Setting the environment variable PS1 (hence, `export` instead of `set`) to `\u@\h \w >` renders your user name (the meaning of `\u`), the literal `@`, your host name (`\h`), your present working directory (`\w`), and the literal string >.

Other special literals include `\t` for the time (in 24-hour format), `\d` for the date (in weekday, month, date format), and `\!` for the shell history number. If you keep your shell window open for extended periods, you can quickly repeat a previous command with `!`*`nnn`*, where *`nnn`* is the number shown in the prompt for the command.

```
strike@nostromo ~ > export PS1="\! $ "
776 $ find . -name ...
...
999 $ !776
find . -name ...
1000 $
```

Have you ever wondered how the shell prompt changes to the octothorpe (#) whenever you are or behave like the superuser, root? By default, the standard prompt is actually the special operator `\$`. If your effective user ID is 0, this emits #; otherwise, it emits a `$`.

Abbreviations also exist to change colors. Here is another sample to demonstrate:

```
strike@nostromo ~ > export PS1='$ '
$ blue='\e[0;34m'
$ none='\e[m'
$ export PS1="$blue\u@\h$none\w>"
<span style="color: blue;">strike@nostromo </span>~ ><br />
```

The (rather unwieldy) abbreviation `\e[0;34m` enables blue. Its complement, `\e[m`, resets the rendering color to the default for your shell window. Both codes are enclosed in single quotation marks to prevent the shell from interpolating characters that have special meaning. This example also shows that you can use variables in a prompt, too. Here, the variables `blue` and `none` are expanded *when the command line is interpreted*, and the prompt is set to the string that results from the expansion. If you want to expand a variable dynamically each time the prompt is rendered, you must escape its interpretation when set. Let's look at just that:

```
<span style="color: blue;">strike@nostromo
    </span>~ >export PS1="\$somevar $ "
$ somevar="hello"
hello $
```

The phrase `\$somevar` escapes the dollar sign, avoiding interpretation of the variable in the command where the prompt is set. Instead, the interpretation occurs whenever the prompt is drawn. If `somevar` changes because of other commands, the prompt displays its new value.

You can also call any command or function in the prompt, as mentioned earlier. Simply use the backquotes (``). For example, if you use the Ruby Version Manager to switch between Ruby language versions and interpreters, you may want to know which Ruby binary is active. There are two approaches:

```
hello $ export PS1="(`which ruby`) \w $ "
(/Users/strike/.rvm/rubies/ruby-1.9.2-p0/bin/ruby) ~ $
```

This first technique applies `which` to find the first Ruby executable in the current PATH. This next technique achieves the same goal but demonstrates how complex an in-place command evaluation can be. If you set the prompt to the string:

```
"(`rvm info | grep 'ruby:' | grep bin | cut -f2 -d: | tr \\" ' '`)"
```

you get the same result, albeit through a more circuitous route.

By the way, you may have wondered why PS1 is "prompt string level 1." Are there other prompt levels? The answer is yes. Environment variables exist for PS2, PS3, and PS4, and these prompts appear anytime you open a new block. Here is one occurrence:

```
$ for i in [A-Z]*
>
```

After you type `for i in [A-Z]*` and press Return, the shell presents the PS2 prompt (the default is the >) to highlight that you are now within the body of the `for` loop. In other words, you are now "nested" in the loop or at the next deeper level. If you finish the loop with `done`, the first-level prompt reappears.

```
$ for i in [A-Z]*
> do
> echo $i
> done
Gemfile
Gemfile.lock
README
```

```
Rakefile
$
```

In fact, a new prompt appears anytime you do not complete the command line at the previous prompt. This explains why an unmatched single or double quotation mark causes a new prompt to appear. The shell is prompting you (no pun intended) to continue what you started.

Embedding information in the prompt is a great way to track state, such as your current host, working directory, and more. Once you create a prompt you like, distribute your shell startup files to all your accounts. If, like most modern users, your screen fills with remote shell windows, the prompts can help you discern one from another. You might craft a function to change the prompt to a different color on each host.

Modders and rodders

Customizing the shell is a perennial geek pastime, and you can find a great deal of inspiration and source code online to find your own work. Two sources deserve special mention: Oh My Z Shell! and Bash It!. The former is for Z shell users, and the latter is for Bash aficionados. Both are a collection of shell modifications, including completions, themes (colors and prompts), functions, and ready-made "dot" files. Both are open source and available from github. Here, let's try Oh, My Z Shell! (or OMZ! for short). You must have Z shell version 4.3.9 or later to use the code.

You can install the package and change your shell to Z shell automatically with `wget`, as Listing 1 shows.

Listing 1. Install and change to the Z shell

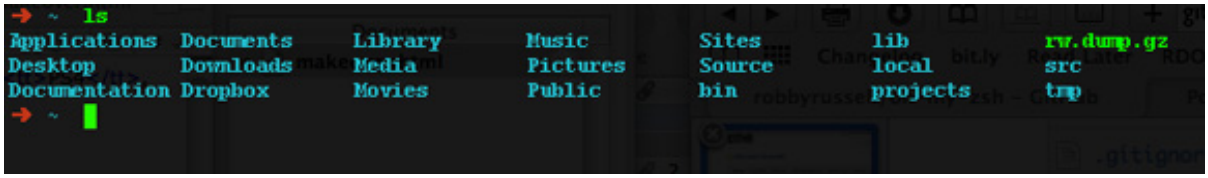
```
$ wget http://github.com/robbyrussell/oh-my-zsh/raw/master/tools/install.sh -O - | sh
...
Cloning Oh My Zsh...
Cloning into /Users/strike/.oh-my-zsh...
remote: Counting objects: 1312, done.
remote: Compressing objects: 100% (750/750), done.
remote: Total 1312 (delta 796), reused 944 (delta 520)
Receiving objects: 100% (1312/1312), 153.63 KiB, done.
Resolving deltas: 100% (796/796), done.
Looking for an existing zsh config...
Using the Oh My Zsh template file and adding it to ~/.zshrc
Copying your current PATH and adding it to the end of ~/.zshrc for you.
Time to change your default shell to zsh!
Changing shell for strike.
Password for strike:
chsh: /usr/bin/env zsh: non-standard shell

         __                                     __
  ____  / /_     ____ ___  __  __   ____  _____/ /_
 / __ \/ __ \   / __ `__ \/ / / /  / __ \/ ___/ __ \
/ /_/ / / / /  / / / / / / /_/ /  / /_/ (__  ) / / /
\____/_/ /_/  /_/ /_/ /_/\__, /   \____/____/_/ /_/
                        /____/
```

Optionally, you can install the kit manually. Doing so is likely preferable if you are running another shell and just want to try the Z shell. Use Git to clone the package, and then run `zsh`:

```
$ git clone git://github.com/robbyrussell/oh-my-zsh.git ~/.oh-my-zsh
$ cp ~/.oh-my-zsh/templates/zshrc.zsh-template ~/.zshrc
$ zsh
```

You should see a screen and prompt that resembles Figure 1. The default then for OMZ! is called "robbyrussell," named eponymously after the steward of OMZ! You can change this to any of the themes listed in ~/.oh-my-zsh/themes. To change a theme, open the ~/.zshrc file and set the ZSH_THEME variable to the base name of the theme file. For example, to use `cloud.zsh-theme`, set `ZSH_THEME=cloud`.

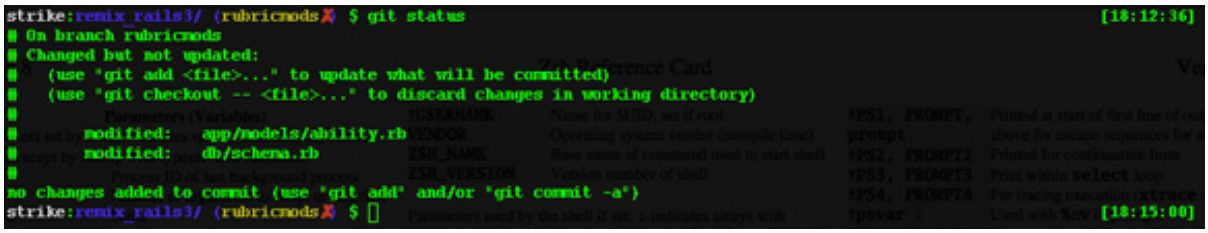
Figure 1. Changing your shell's theme



You'll likely notice that many of the theme's print status information is in the prompt *and to the far right of the prompt*. For example, the Clean theme prints the current time at the far right. Screen emulators typically don't include a status line at the bottom (unlike their hardware progenitors) but can use the real estate found to the right of the prompt for dynamic feedback. Recall the funny `\e[` codes to set colors in the prompt? There is an extensive set of such "escapes" to move the cursor around a window. Further, rather than use arcane symbols and numbers, modern UNIX systems use `tput` to look up and emit escapes by name or purpose. Z shell uses all this trickery to provide an RPS1 and RPS2 for right-side prompts on the initial and subsequent lines, respectively.

Beyond thematic treatments of colors and prompts, OMZ! includes plug-ins that group like functions and features. If you use Git for source control, for instance, you can enable the Git OMZ! plug-in to amend your prompt with Git status. Again, open the ~/.zshrc file and edit the plug-ins line to include `git>`. Now, when you switch to a Git repo, the prompt reflects state. For instance, Figure 2 shows the Clean theme and a repository with modifications that are not yet staged.

Figure 2. The Clean theme and repository



Smartly, the prompt shows the current branch ("rubricmods") and a red X, indicating that the current repository is dirty or that local files have been modified but not committed. Once the revisions are committed, the X is elided. The Git plug-in also adds helpful aliases for common Git command combinations and context-sensitive completions for Git options. Try one: If you type `git branch` followed by the Tab key, the Git plug-in lists the available branches. Other plug-ins exist for Mac OS X, Ruby on Rails development, MySQL, and more. Some of the plug-ins are slight, perhaps nothing more than a function or two or some aliases; other plug-ins are more extensive.

Dirpersist is a good showcase of the intent of OMZ! plug-ins. Dirpersist saves and restores your directory stack across shell invocations, thus preserving important state so that you can resume work where you left off. To use it, add the plug-in to your ~/.zshrc. The source of the plug-in is brief and shown in Listing 2.

Listing 2. OMZ! plug-in source

```
#!/bin/zsh
#
# Make the dirstack more persistent
#
# Add dirpersist to $plugins in ~/.zshrc to load
#

# $zdirstore is the file used to persist the stack
zdirstore=~/.zdirstore

dirpersistinstall () {
  if grep 'dirpersiststore' ~/.zlogout > /dev/null; then
  else
    if read -q \?"Would you like to set up your .zlogout file for
        use with dirpersist? (y/n) "; then
      echo "# Store dirs stack\n\
        # See ~/.oh-my-zsh/plugins/dirspersist.plugin.zsh\ndirpersiststore" >> ~/.zlogout
    else
      echo "If you don't want this message to appear, remove dirspersist from \$plugins"
    fi
  fi
}

dirpersiststore () {
  dirs -p | perl -e 'foreach (reverse <STDIN>) {\
    chomp;s/([& ])/\\$1/g ;print "if [ -d $_ ]; then pushd -q $_; fi\n"}' > $zdirstore
}

dirpersistrestore () {
  if [ -f $zdirstore ]; then
    source $zdirstore
  fi
}
```

```
DIRSTACKSIZE=10
setopt autopushd pushdminus pushdsilent pushdtohome pushdignoredups

dirpersistinstall
dirpersistrestore

# Make popd changes permanent without having to wait for logout
alias popd="popd;dirpersiststore"
```

The plug-in is composed of three functions, a handful of shell options, and an alias for `popd` that persists the directory stack with each pop operation. The plug-in also initializes its environment, creating a place to store the directory stack and, by modifying the Z shell logout file, ~/.zlogout, to persist the directory stack any time you log out. As you can see, a new plug-in is easy to create, and you can build one around any set of shell commands.

If you like OMZ! but use the Bash shell, try Bash It! It was inspired by OMZ! and has similar and operation. Bash It also offers plug-ins for Subversion and nginx.

Burning virtual rubber

Pop open the hood of OMZ! or Bash It! to see what makes shell customizations tick. You can also find wisdom and novelty by searching the Web for the phrase "dot files": Many UNIX gearheads post personal shell configurations online for others to cull from. Here's a tip, too: Keep your dot files in some form of version control. Given a repository, you're likely to feel more comfortable experimenting with change and with other shells. You can revert to a prior version if you make a mistake or keep variations for particular shells or scenarios. The shell is flexible and dynamic, and one size need not fit all. Once you have a set of cool "dot files," share with others. Showing off your dot files may not land you a starring role in the next "The Fast and the Furious," but it'll likely boost your geek cred.

Resources

Learn

- Z shell: You can find the source of the Z shell and extensive documentation and tutorials at the shell's project page.
- Bash documentation: Read more about the Bash shell and terminal cursor and rendering options.
- Oh My Z Shell!: Check out the open source project to add extensions and helpful shortcuts to the Z shell.
- Bash It!: This community project provides a framework of extensions and convenience for the Bash shell.
- Many developers and open source projects keep code under version control with Git and github.
- Speaking UNIX: Check out other parts in this series.
- AIX and UNIX developerWorks zone: The AIX and UNIX zone provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.
- New to AIX and UNIX? Visit the New to AIX and UNIX page to learn more.
- Technology bookstore: Browse the technology bookstore for books on this and other technical topics.

Get products and technologies

- Bash shell: Download the source of the Bash shell from the GNU Software Foundation.

Discuss

- developerWorks blogs: Check out our blogs and get involved in the developerWorks community.
- Participate in the AIX and UNIX forums:
  - AIX 5L—technical forum
  - AIX for Developers Forum
  - Cluster Systems Management
  - IBM Support Assistant
  - Performance Tools—technical
  - More AIX and UNIX forums

About the author

Martin Streicher

Martin Streicher is a freelance Ruby on Rails developer and the former Editor-in-Chief of *Linux Magazine*. Martin holds a Masters of Science degree in computer science from Purdue University and has programmed UNIX-like systems since 1986. He collects art and toys. You can reach Martin at martin.streicher@gmail.com.