http://www.coincoin.fr.eu.org/?KSH93-cool-features-for-scripting-19840

KSH93 cool features for scripting

- 6- Webographie -

Date de mise en ligne : lundi 30 novembre 2015

From time to time, I'm involved into a trolling conversation when any linux kiddie tells me :

Bash is really the superior shell

I totally disagree, but as I'm getting older, I don't argue anymore.

Anyway, in this post I will expose two arguments, or I should say two reasons, why I usually use ksh93 to run my scripts.

Note I'm really talking about the engine of the script, (the shebang definition). set I'm used to the bourn shell syntax therefore I also exclude any C shell from the comparison. My $SHELL for interactivity is zsh because it's efficient enough and it has a bunch of really cool features I won't discuss in this post (maybe later)

Read, loops, forks and efficiency...

More than 10 years ago, as I was working for a project at IBM, my excellent team leader told me to refer to this book : Unix Power Tools . I did learn a lot with it.

And one feature I've always used is the while read loop.

The use case

Let's take this script as example : $ cat test for i in $(seq 1 500) do echo $i | read a echo -ne "$a\r" done echo ""

It simply iterate 500 times and display the counter on the screen.

The result of execution

Let's execute it in different shells for i in bash zsh ksh do echo "$i =>" eval $i test done bash => zsh => 500 ksh => 500

Bash is the only one which does not display the expected result. The explanation is that the shell sees a pipe and the fork the process. The assignation to the variable a is in another context and therefore, when the father wants to display $a in the current shell, the variable is empty.

Wait, but why does ksh (and zsh) do display the correct result ? Simply because ksh and zsh have noticed that the command after the pipe was a builtin, and therefore that it was un-useful to fork.

Strace to the rescue...

To prove it, let's check for syscalls with the strace tool, and count how many clones and calls are performed : $ for i in bash zsh ksh do echo "$i =>" strace -c $i test 2>&1 | egrep "clone|calls" done bash => % time seconds usecs/call calls errors syscall 56.05 0.067081 67 1001 clone zsh => % time seconds usecs/call calls errors syscall 71.57 0.057681 115 501 clone ksh => % time seconds usecs/call calls errors syscall 68.50 0.042059 84 500 clone

quod erat demonstrandum, twice as much clone in bash thant in ksh|zsh.

Efficiency

Of course this as an impact on performances, because fork are expensive, let's query the execution time : for i in bash zsh ksh do echo "$i =>" eval time $i test done bash => bash test 0,17s user 0,86s system 95% cpu 1,079 total zsh => 500 zsh test 0,08s user 0,46s system 82% cpu 0,648 total ksh => 500 ksh test 0,07s user 0,46s system 65% cpu 0,819 total

This sounds clear to me...

The KSH93 Getopts unknown feature

Another cool feature I've discovered recently is the little addon of the getopts feature.

I wanted to use the getopts built in in a script. As usual, I did *RTFM* (because I never know when to use colon etc.).

Here is the extract of the man page of ksh93 relative to the getopts function : **getopts** [**-a** *name*] *optstring vname* [*arg* ...] Checks *arg* for legal options. If *arg* is omitted, the positional parameters are used. An option argument begins with a **+** or a **-**. An option not beginning with **+** or **-** or the argument **â€”** ends the options. Options beginning with **+** are only recognized when *optstring* begins with a **+**. *optstring* contains the letters that **getopts** recognizes. If a letter is followed by a **:**, that option is expected to have an argument. The options can be separated from the argument by blanks. The option **- ?** causes **getopts** to generate a usage message on standard error. The **-a** argument can be used to specify the name to use for the usage message, which defaults to **$0**. **getopts** places the next option letter it finds inside variable *vname* each time it is invoked. The option letter will be prepended with a **+** when *arg* begins with a **+**. The index of the next *arg* is stored in **OPTIND**. The option argument, if any, gets stored in **OPTARG**. A leading **:** in *optstring* causes **getopts** to store the letter of an invalid option in **OPTARG**, and to set *vname* to **?** for an unknown option and to **:** when a required option argument is missing. Otherwise, **getopts** prints an error message. The exit status is non-zero when there are no more options.

There is no way to specify any of the options **:**, **+**, **-**, **?**, **[**, and **]**. The option # can only be specified as the first option.

This particular sentence, in the middle of the documentation peaked my interest

The option - ? causes getopts to generate a usage message on standard error.

What ? We can generate usage with getopts ?

Cool, any script should be documented, but any documentation should not be difficult to implement.
[PNG - 16.8Â kio]

https://xkcd.com/1343/

I did googled and found this web page which is an extract from this book Learning the Korn Shell

An example is sometimes better than an explanation (and the book is complete on this subject)

The example

The script

```
 # !/bin/ksh ENV=dev MPATH=/tmp ## ### Man usage and co... USAGE="[- ?The example script v1.0]" USAGE+="[-author ?Olivier Wulveryck]" USAGE+="[-copyright ?Copyright (C) My Blog]" USAGE+="[+NAME ?$0 --- The Example Script]" USAGE+="[+DESCRIPTION ?The description of the script]" USAGE+="[u:user] :[user to run the command as :=$USER ?Use the name of the user you want to sudo to : ]" USAGE+="[e:env] :[environnement :=$ENV ?environnement to use (eg : dev, prod) ]" USAGE+="[p:path] :[Execution PATH :=$MPATH ?prefix of the chroot]" USAGE+="[+EXAMPLE ?$0 action2]" USAGE+='[+SEE ALSO ?My Blog Post : http://blog.owulveryck.info/2015/11/30/ksh93-cool-features-for-scripting]' USAGE+="[+BUGS ?A few, maybe...]" ### Option Checking while getopts "$USAGE" optchar ; do case $optchar in u) USER=$OPTARG ; ; e) ENV=$OPTARG ; ; p) PATH=$OPTARG ; ; esac done shift OPTIND-1 ACTION=$1
```

The invocation

Here are two *singing* examples of the usage output (sorry, I'm tired)

Ballad of a thin man

```
 $ ./blog.ksh â€"man NAME ./blog.ksh --- The Example Script SYNOPSIS ./blog.ksh [ options ] DESCRIPTION The description of the script OPTIONS -u, â€"user=user to run the command as Use the name of the user you want to sudo to : The default value is owulveryck. -e, â€"env=environnement environnement to use (eg : dev, prod) The default value is dev. -p, â€"path=Execution PATH prefix of the chroot The default value is /tmp. EXAMPLE ./blog.ksh action2 SEE ALSO My Blog Post : http://blog.owulveryck.info/2015/11/30/ksh93-cool-features-for-scripting BUGS A few, maybe... IMPLEMENTATION version The example script v1.0 author Olivier Wulveryck copyright Copyright (C) My Blog I'm gonna try with a little help (from my friends)
```

```
 $ ./blog.ksh â€"help Usage : ./blog.ksh [ options ] OPTIONS -u, â€"user=user to run the command as Use the name
```

of the user you want to sudo to : The default value is owulveryck. -e, â€”env=environnement environnement to use (eg : dev, prod) The default value is dev. -p, â€”path=Execution PATH prefix of the chroot The default value is /tmp.

And let's try with an invalid option... ./blog.ksh -t ./blog.ksh : -t : unknown option Usage : ./blog.ksh [-u user to run the command as] [-e environnement] [-p Execution PATH]

Conclusion

By now, KSH93 remains my favorite engine for shell scripts, but is sometimes replaced by ZSH.

Actually, ZSH seems as "smart" and efficient, but this getopts feature is really nice for any script aim to be distributed widely.