Zsh Native Scripting Handbook

Sebastian Gniazdowski

Version 1.19, 10/21/2018

Table of Contents

This document has been created after 2.5 years of creating software for Zshell and receiving help from IRC channel #zsh. Avoiding forks was the main guideline when creating the projects and this lead to multiple discoveries of Zsh constructs that are fast, robust and do not depend on external tools. Such code is more like Ruby or Perl code, less like top-down shell scripts.

Information

@ is about keeping array form

How to access all array elements in a shell? The standard answer: `use @ subscript`, i.e. `${array[@]}`. However, this is the Bash & Ksh way (and with the option `KSH_ARRAYS`, Zsh also works this way, i.e. needs `@` to access whole array). Zshell **is different**: it is `$array` that refers to all elements anyway. There is no need of `@` subscript.

So what use has `@` in the Zsh-world? It is: “keep array form” or “do not join”. When is it activated? When user quotes the array, i.e. invokes `"$array"`, he induces **joining** of all array elements (into a single string; again, `KSH_ARRAYS` induces the Bash behavior: `"$array"` contains only the first element). `@` is to have elements still quoted (so empty elements are preserved), but not joined.

Two forms are available, `"$array[@]"` and `"${(@)array}"`. First form has additional effect – when option `KSH_ARRAYS` is set, it indeed induces referencing to whole array instead of first element only. It should then use braces, i.e. `${array[@]}`, `"${array[@]}"` (`KSH_ARRAYS` requirement).

In practice, if you’ll use `@` as a subscript – `[@]`, not as a flag – `${(@)…}`, then you’ll make the code `KSH_ARRAYS`-compatible.

extended_glob

Glob-flags `#b` and `#m` require `setopt extended_glob`. Patterns utilizing `~` and `^` also require it. Extended-glob is one of the main features of Zsh.

Constructs

Reading a file

```
declare -a lines; lines=( "${(@f)"$(<path/file)"}" )
```

This preserves empty lines because of double-quoting (the outside one). `@`-flag is used to obtain array instead of scalar. If you don’t want empty lines preserved, you can also skip `@`-splitting, as it is explained in Information section:

```
declare -a lines; lines=( ${(f)"$(<path/file)"} )
```

Note: `$(<⋯)` construct strips trailing empty lines.

Reading from stdin

This topic is governed by the same principles a the previous paragraph (`Reading a file`), with the single difference that instead of the substitution `"$(<file-path)"` the substitution that should be used is `"$(command arg1 ⋯)"`, i.e.:

```
declare -a lines; lines=( ${(f)"$(command arg1 ...)"} )
```

This will read the `command's` output into the array `lines`. The version that does `@` splitting and retains any empty lines is:

```
declare -a lines; lines=( "${(f@)$(command arg1 ...)}" )
```

Note that instead of four double-quotes `"`, an idiom that is justified (simply suggested) by the Zsh documentation (and was used in the previous paragraph, in the snippet `⋯ "${(@f)"$(<path/file)"}" ⋯`), only **two** double-quotes are being used. I've investigated this form with the main Zsh developers on the `zsh-workers@zsh.org` mailing list, and it was clearly stated that single, outside quoting of `${(f@)⋯}` substitution works as if it was also separately applied to `$(command ⋯)` (or to `$(<file-path)`) inner substitution, so the second double-quoting isn't actually needed.

Skipping grep

```
declare -a lines; lines=( "${(@f)"$(<path/file)"}" )
declare -a grepped; grepped=( ${(M)lines:#*query*} )
```

To have `grep -v` effect, skip `M`-flag. To grep case insensitively, use `#i` glob flag (`⋯:#(#i)*query*}`).

As it can be seen, `${⋯:#⋯}` substitution is a filtering of array, which by default filters-out elements (`(M)` flag induces the opposite behavior). When used with string, not an array, it behaves similarily: returns empty string when `{input_string_var:#pattern}` matches whole input string.

Side-note: `(M)` flag can be used also with `${(M)var#pattern}` and other substitutions, to retain what's matched by the pattern instead of removing that.

Multi-line matching like with grep

Suppose you have a Subversion repository and want to check if it contains files being not under version control. You could do this in Bash style like follows:

```
local svn_status="$(svn status)"
if [[ -n "$(echo "$svn_status" | \grep \^\?)" ]]; then
    echo found
fi
```

That are 3 forks: for `svn status`, for `echo` and for `grep`. This can be solved by `:#` substitution and `(M)` flag described above in this section (just check if the number of matched lines is greater than 0). However, there’s a more direct approach:

```
local svn_status="$(svn status)" nl=$'\n'
if [[ "$svn_status" = *((#s)|$nl)\?* ]]; then
  echo found
fi
```

This requires `extendedglob`. The `(#s)` means: "start of the string". So `((#s)|$nl)` means "start of the string OR preceded by a new-line".

If the `extendedglob` option cannot be used for some reason, this can be achieved also without it, but essentially it means that alternative (i.e. `|`) of two versions of the pattern will have to be matched:

```
setopt localoptions noextendedglob
local svn_status="$(svn status)" nl=$'\n'
if [[ "$svn_status" = (\?*|*$nl\?*) ]]; then
  echo found
fi
```

In general, multi-line matching falls into the following idiom (`extendedglob` version):

```
local needle="?" required_preceding='[[:space:]]#'
[[ "$(svn status)" = *((#s)|$nl)${~required_preceding}${needle}* ]] && echo found
```

It does a single fork (calls `svn status`). The `${~variable}` means (the `~` in it): "the variable is holding a pattern, interpret it". All in all, instead of regular expressions we were using patterns (globs) (see this section).

Pattern matching in AND-fashion

```
[[ "abc xyz efg" = *abc*~^*efg* ]] && print Match found
```

The `~` is a negation — `match *abc* but not …`. Then, `^` is also a negation. The effect is: `*abc* but not those that don’t have *efg*` which equals to: `*abc* but those that have also *efg*`. This is a regular pattern and it can be used with `:#` above to search arrays, or with `R`-subscript flag to search hashes (`${hsh[(R)*pattern*]}`), etc. Inventor of those patterns is Mikael Magnusson.

Skipping tr

```
declare -A map; map=( a 1 b 2 );
text=( "ab" "ba" )
text=( ${text[@]//(#m)?/${map[$MATCH]}} )
print $text ▶ 12 21
```

`#m` flag enables the `$MATCH` parameter. At each `//` substitution, `$map` is queried for character-replacement. You can substitute a text variable too, just skip `[@]` and parentheses in assignment.

Ternary expressions with `\+,-,:+,:-` substitutions

```
HELP="yes"; print ${${HELP:+help enabled}:-help disabled} ▶ help enabled
HELP=""; print ${${HELP:+help enabled}:-help disabled} ▶ help disabled
```

Ternary expression is known from `C` language but exists also in Zsh, but directly only in math context, i.e. `(( a = a > 0 ? b : c ))`. Flexibility of Zsh allows such expressions also in normal context. Above is an example. `:+` is "if not empty, substitute …" `:-` is "if empty, substitute …". You can save great number of lines of code with those substitutions, it's normally at least 4-lines `if` condition or lenghty `&&`/`||` use.

Ternary expressions with `:#` substitution

```
var=abc; print ${${${(M)var:#abc}:+is abc}:-not abc} ▶ is abc
var=abcd; print ${${${(M)var:#abc}:+is abc}:-not abc} ▶ not abc
```

An one-line "if var = x, then …, else …". Again, can spare a great amount of boring code that makes 10-line function a 20-line one.

Using built-in regular expressions engine

```
[[ "aabbb" = (#b)(a##)*(b(#c2,2)) ]] && print ${match[1]}-${match[2]} ▶ aa-bb
```

`##` is: "1 or more". `(#c2,2)` is: "exactly 2". A few other constructs: `#` is "0 or more", `?` is "any character", `(a|b|)` is "a or b or empty match". `#b` enables the `$match` parameters. There's also `#m` but it has one parameter `$MATCH` for whole matched text, not for any parenthesis.

Zsh patterns are basically a custom regular expressions engine. They are slightly faster than `zsh/regex` module (used for `=~` operator) and don't have that dependency (regex module can be not present, e.g. in default static build of Zsh). Also, they can be used in substitutions, for example in `//` substitution.

Skipping uniq

```
declare -aU array; array=( a a b ); print $array ▶ a b
declare -a array; array=( a a b ); print ${(u)array} ▶ a b
```

Enable `-U` flag for array so that it guards elements to be unique, or use `u`-flag to uniquify elements of any array.

Skipping awk

```
declare -a list; list=( "a,b,c,1,e" "p,q,r,2,t" );
print "${list[@]/(#b)([^,]##,)(#c3,3)([^,]##)*/${match[2]}}" ▶ 1 2
```

The pattern specifies 3 blocks of `[^,]##,` so 3 "not-comma multiple times, then comma", then single block of "not-comma multiple times" in second parentheses — and then replaces this with second parentheses. Result is 4th column extracted from multiple lines of text, something `awk` is often used for. Other method is use of `s`-flag. For single line of text:

```
text="a,b,c,1,e"; print ${${(s:,:)text}[4]} ▶ 1
```

Thanks to in-substitution code-execution capabilities it's possible to use `s`-flag to apply it to multiple lines:

```
declare -a list; list=( "a,b,c,1,e" "p,q,r,2,t" );
print "${list[@]/(#m)*/${${(s:,:)MATCH}[4]}}" ▶ 1 2
```

There is a problem with the `(s::)` flag that can be solved if Zsh is version `5.4` or higher: if there will be single input column, e.g. `list=( "column1" "a,b")` instead of two or more columns (i.e. `list=( "column1,column2" "a,b" )`), then `(s::)` will return **string** instead of 1-element **array**. So the index `[4]` in above snippet will index a string, and show its 4-th letter. Starting with Zsh 5.4, thanks to a patch by Bart Schaefer (`40640: the (A) parameter flag forces array result even if…`), it is possible to force **array**-kind of result even for single column, by adding `(A)` flag, i.e.:

```
declare -a list; list=( "a,b,c,1,e" "p,q,r,2,t" "column1" );
print "${list[@]/(#m)*/${${(As:,:)MATCH}[4]}}" ▶ 1 2
print "${list[@]/(#m)*/${${(s:,:)MATCH}[4]}}" ▶ 1 2 u
```

Side-note: `(A)` flag is often used together with `::=` assignment-substitution and `(P)` flag, to assign arrays and hashes by-name.

Searching arrays

```
declare -a array; array=( a b " c1" d ); print ${array[(r)[[:space:]][[:alpha:]]*]} ▶
c1
```

`[[:space:]]` contains unicode spaces. This is often used in conditional expression like `[[ -z ${array[(r)…]} ]]`.

Note that Skipping grep that uses `:#` substitution can also be used to search arrays.

Code execution in `//` substitution

```
append() { gathered+=( $array[$1] ); }
functions -M append 1 1 append
declare -a array; array=( "Value 1" "Other data" "Value 2" )
declare -a gathered; integer idx=0
: ${array[@]/(#b)(Value ([[:digit:]]##)|*)/$(( ${#match[2]} > 0 ? append(++idx) :
++idx ))}
print $gathered ▶ Value 1 Value 2
```

Use of `#b` glob flag enables math-code execution (and not only) in `/` and `//` substitutions. Implementation is very fast.

Serializing data

```
declare -A hsh deserialized; hsh=( key value )
serialized="${(j: :)${(qkv@)hsh}}"
deserialized=( "${(Q@)${(z@)serialized}}" )
print ${(kv)deserialized} ▶ key value
```

`j`-flag means join — by spaces, in this case. Flags `kv` mean: keys and values, interleaving. Important `q` -flag means: quote. So what is obtained is each key and value quoted, and put into string separated by spaces.

`z`-flag means: split as if Zsh parser would split. So quoting (with backslashes, double quoting and other) is recognized. Obtained is array `( "key" "value")` which is then dequoted with `Q`-flag. This yields original data, assigned to hash `deserialized`. Use this to e.g. implement array of hashes.

Note: to be compatible with `setopt ksharrays`, use `[@]` instead of `(@)`, e.g.: `…( "${(Q)${(z)serialized[@]}[@]}" )`

Tip: serializing with Bash

```
array=( key1 key2 )
printf -v serialized "%q " "${array[@]}"
eval "deserialized=($serialized)"
```

This method works also with Zsh. The drawback is use of `eval`, however it's impossible that any problem will occurr unless someone compromises variable's value, but as always, `eval` should be avoided if possible.

Real world examples

Testing for Git subcommand

Following code checks if there is a `git` subcommand `$mysub`:

```
if git help -a | grep "^  [a-z]" | tr ' ' '\n' | grep -x $mysub > /dev/null >
/dev/null; then
```

That are `4` forks. The code can be replaced according to this guide:

```
local -a lines_list
lines_list=( ${(f)"$(git help -a)"} )
lines_list=( ${(M)${(s: :)${(M)lines_list:#  [a-z]*}}:#$mysub} )
if (( ${#lines_list} > 0 )); then
```

The result is just `1` fork.

Counting unquoted-only apostrophes

A project was needing this to do some Zle line-continuation tricks (when you put a backslash-\ at the end of the line and press enter – it is the line-continuation that occurs at that moment).

The required functionality is: in given string, count the number of apostrophes, but *only the unquoted ones*. This means that only apostrophes with null or an even number of preceding backslashes should be accepted into the count:

```
buf="word'continue\'after\\\'afterSecnd\\''afterPair"
integer count=0
: ${buf//(#b)((#s)|[^\\])([\\][\\])#(\'\'#)/$(( count += ${#match[3]} ))}
echo $count ▶ 3
```

The answer (i.e. the output) to the above presentation and example is: `3` (there are `3` unquoted apostrophes in total in the string kept in the variable `$buf`).

Below follows a variation of above snippet that doesn't use math-code execution:

```
buf="word'continue\'after\\\'afterSecnd\\''afterPair"
buf="${(S)buf//(#b)*((#s)|[^\\])([\\][\\])#(\'\'#)*/${match[3]}}"; buf=${buf%%[^\']##}
integer count=${#buf}
echo $count ▶ 3
```

This is possible thanks to `(S)` flag – non-greedy matching, `([\\][\\])` trick – it matches only unquoted following `(\'\'#)` characters (which are the apostrophes) and a general strategy to replace `anything-apostrope(s)` (unquoted ones) with `the-apostrope(s)` (and then count them with `${#buf}`).

Tips and Tricks

Parsing INI file

With Zshell's `extended_glob` parsing an `ini` file is an easy task. It will not result in a nested-arrays data structure (Zsh doesn't support nested hashes), but the hash keys like `$DB_CONF[db1_<connection>_host]` are actually really intuitive.

The code should be placed in file named `read-ini-file`, in `$fpath`, and `autoload read-ini-file` should be invoked.

read-ini-file

```
# Copyright (c) 2018 Sebastian Gniazdowski
#
# $1 - path to the ini file to parse
# $2 - name of output hash
# $3 - prefix for keys in the hash
#
# Writes to given hash under keys built in following way: ${3}<section>_field.
# Values are values from ini file. Example invocation:
#
# read-ini-file ./database1-setup.ini DB_CONF db1_
# read-ini-file ./database2-setup.ini DB_CONF db2_
#

setopt localoptions extendedglob

local __ini_file="$1" __out_hash="$2" __key_prefix="$3"
local IFS='' __line __cur_section="void" __access_string
local -a match mbegin mend

[[ ! -r "$__ini_file" ]] && { builtin print -r "read-ini-file: an ini file is
unreadable ($__ini_file)"; return 1; }

while read -r -t 1 __line; do
    if [[ "$__line" = [[:blank:]]#\;* ]]; then
        continue
    # Match "[Section]" line
    elif [[ "$__line" = (#b)[[:blank:]]#\[([^\]]##)\][[:blank:]]# ]]; then
        __cur_section="${match[1]}"
    # Match "string = string" line
    elif [[ "$__line" =
(#b)[[:blank:]]#([^[:blank:]=]##)[[:blank:]]#[=][[:blank:]]#(*) ]]; then
        match[2]="${match[2]%"${match[2]##*[! $'\t']}"}" # severe trick - remove
trailing whitespace
        __access_string="${__out_hash}[${__key_prefix}<$__cur_section>_${match[1]}]"
        : "${(P)__access_string::=${match[2]}}"
    fi
done < "$__ini_file"

return 0
```

Appendix A: Revision history (history of updates to the document)

v1.19, 11/05/2018: Less abstract but more true function of `@` (flag, subscript)
v1.18, 10/21/2018: Multi-line matching like with `grep`
v1.16, 10/21/2018: Count apostrophes example – additional version without math-code

v1.15, 10/21/2018: Inform about potential (s::)-flag problems ("Skipping awk")
v1.1, 10/21/2018: Elaborate on (M)-flag in "Skipping grep" section
v1.05, 10/21/2018: New section "Reading from stdin"
v1.0, 09/29/2018: New real world examples (`git help -a` and "Counting apostrophes")