The mapfile builtin command

Synopsis

```
mapfile [-n COUNT] [-O ORIGIN] [-s COUNT] [-t] [-u FD] [-C CALLBACK]
[-c QUANTUM] [ARRAY]
```

```
readarray [-n COUNT] [-O ORIGIN] [-s COUNT] [-t] [-u FD] [-C CALLBAC
K] [-c QUANTUM] [ARRAY]
```

Description

This builtin is also accessible using the command name `readarray` .

`mapfile` is one of the two builtin commands primarily intended for handling standard input (the other being `read`). `mapfile` reads lines of standard input and assigns each to the elements of an indexed array. If no array name is given, the default array name is `MAPFILE` . The target array must be a "normal" integer indexed array.

`mapfile` returns success (0) unless an invalid option is given or the given array `ARRAY` is set readonly.

Option	Description
`-c QUANTUM`	Specifies the number of lines that have to be read between every call to the callback specified with `-C` . The default QUANTUM is 5000
`-C CALLBACK`	Specifies a callback. The string `CALLBACK` can be any shell code, the index of the array that will be assigned, and the line is appended at evaluation time.
`-n COUNT`	Reads at most `COUNT` lines, then terminates. If `COUNT` is 0, then all lines are read (default).
`-O ORIGIN`	Starts populating the given array `ARRAY` at the index `ORIGIN` rather than clearing it and starting at index 0.
`-s COUNT`	Discards the first `COUNT` lines read.
`-t`	Remove any trailing newline from a line read, before it is assigned to an array element.

Option	Description
`-u FD`	Read from filedescriptor `FD` rather than standard input.

While `mapfile` isn't a common or portable shell feature, it's functionality will be familiar to many programmers. Almost all programming languages (aside from shells) with support for compound datatypes like arrays, and which handle open file objects in the traditional way, have some analogous shortcut for easily reading all lines of some input as a standard feature. In Bash, `mapfile` in itself can't do anything that couldn't already be done using read and a loop, and if portability is even a slight concern, should never be used. However, it does *significantly* outperform a read loop, and can make for shorter and cleaner code - especially convenient for interactive use.

Examples

Here's a real-world example of interactive use borrowed from Gentoo workflow. Xorg updates require rebuilding drivers, and the Gentoo-suggested command is less than ideal, so let's Bashify it. The first command produces a list of packages, one per line. We can read those into the array named "args" using `mapfile`, stripping trailing newlines with the '-t' option. The resulting array is then expanded into the arguments of the "emerge" command - an interface to Gentoo's package manager. This type of usage can make for a safe and effective replacement for xargs(1) in certain situations. Unlike xargs, all arguments are guaranteed to be passed to a single invocation of the command with no wordsplitting, pathname expansion, or other monkey business.

```
# eix --only-names -IC x11-drivers | { mapfile -t args; emerge -av1
"${args[@]}" <&1; }
```

Note the use of command grouping to keep the emerge command inside the pipe's subshell and within the scope of "args". Also note the unusual redirection. This is because the -a flag makes emerge interactive, asking the user for confirmation before continuing, and checking with isatty(3) to abort if stdin isn't pointed at a terminal. Since stdin of the entire command group is still coming from the pipe even though mapfile has read all available input, we just borrow FD 1 as it just so happens to be pointing where we want it. More on this over at greycat's wiki: http://mywiki.wooledge.org/BashFAQ/024 (http://mywiki.wooledge.org/BashFAQ/024)

The callback

This is one of the more unusual features of a Bash builtin. As far as I'm able to tell, the exact behavior is as follows: If defined, as each line is read, the code contained within the string argument to the -C flag is evaluated and executed *before* the assignment of each array element. There are no restrictions to this string, which can be any arbitrary code, however, two additional "words" are automatically appended to the end before evaluation: the index, and corresponding line of data to be assigned to the next array element. Since all this happens before assignment, the callback feature cannot be used to modify the element to be assigned, though it can read and modify any array elements already assigned.

A very simple example might be to use it as a kind of progress bar. This will print a dot for each line read. Note the escaped comment to hide the appended words from printf.

```
$ printf '%s\n' {1..5} | mapfile -c 1 -C 'printf . \#' )
.....
```

Really, the intended usage is for the callback to just contain the name of a function, with the extra words passed to it as arguments. If you're going to use callbacks at all, this is probably the best way because it allows for easy access to the arguments with no ugly "code in a string".

```
$ foo() { echo "|$1|"; }; mapfile -n 11 -c 2 -C 'foo' <file
|2|
|4|
etc..
```

For the sake of completeness, here are some more complicated examples inspired by a question asked in #bash - how to prepend something to every line of some input, and then output even and odd lines to separate files. This is far from the best possible answer, but hopefully illustrates the callback behavior:

```
$ { printf 'input%s\n' {1..10} | mapfile -c 1 -C '>&$(( (${#x[@]} %
2) + 3 )) printf -- "%.sprefix %s"' x; } 3>outfile0 4>outfile1
$ cat outfile{0,1}
prefix input1
prefix input3
prefix input5
prefix input7
prefix input9
prefix input2
prefix input4
prefix input6
prefix input8
prefix input10
```

Since redirects are syntactically allowed anywhere in a command, we put it before the printf to stay out of the way of additional arguments. Rather than opening "outfile<n>" for appending on each call by calculating the filename, open an FD for each first and calculate which FD to send output to by measuring the size of x mod 2. The zero-width format specification is used to absorb the index number argument.

Another variation might be to add each of these lines to the elements of separate arrays. I'll leave dissecting this one as an exercise for the reader. This is quite the hack but illustrates some interesting properties of printf -v and mapfile -C (which you should probably never use in real code).

```
$ y=( 'odd[j]' 'even[j++]' ); printf 'input%s\n' {1..10} | { mapfile
-tc 1 -C 'printf -v "${y[${#x[@]} % 2]}" -- "%.sprefix %s"' x; printf
'%s\n' "${odd[@]}" '' "${even[@]}"; }
prefix input1
prefix input3
prefix input5
prefix input7
prefix input9

prefix input2
prefix input4
prefix input6
prefix input8
prefix input10
```

This example based on yet another #bash question illustrates mapfile in combination with read. The sample input is the heredoc to `main`. The goal is to build a "struct" based upon records in the input file made up of the numbers following the colon on each line. Every 3rd line is a key followed by 2 corresponding fields. The showRecord function takes a key and returns the record.

```
#!/usr/bin/env bash

showRecord() {
    printf 'key[%d] = %d, %d\n' "$1" "${vals[@]:keys[$1]*2:2}"
}

parseRecords() {
    trap 'unset -f _f' RETURN
    _f() {
        local x
        IFS=: read -r _ x
        ((keys[x]=n++))
    }
    local n

    _f
    mapfile -tc2 -C _f "$1"
    eval "$1"'=("${'"$1"'[@]##*:}")' # Return the array with some mod
ification
}

main() {
    local -a keys vals
    parseRecords vals
    showRecord "$1"
}

main "$1" <<-"EOF"
fabric.domain:123
routex:1
routey:2
fabric.domain:321
routex:6
routey:4
EOF
```

For example, running `scriptname 321` would output `key[321] = 6, 4`. Every 2 lines read by `mapfile`, the function `_f` is called, which reads one additional line. Since the first line in the file is a key, and `_f` is responsible for the keys, it gets called first so that `mapfile` starts by reading the second line of input, calling `_f` with each subsequent 2 iterations. The RETURN trap is unimportant.

Bugs

- Early implementations were buggy. For example, `mapfile` filling the readline history buffer with calls to the `CALLBACK`. This was fixed in 4.1 beta.
- `mapfile -n` reads an extra line beyond the last line assigned to the array, through Bash. Fixed in 4.2.35 (ftp://ftp.gnu.org/gnu/bash/bash-4.2-patches/bash42-035).
- `mapfile` callbacks could cause a crash if the variable being assigned is manipulated in certain ways. https://lists.gnu.org/archive/html/bug-bash/2013-

01/msg00039.html (https://lists.gnu.org/archive/html/bug-bash/2013-01/msg00039.html). Fixed in 4.3.

To Do

- Create an implementation as a shell function that's portable between Ksh, Zsh, and Bash (and possibly other bourne-like shells with array support).

See also

- Arrays
- The read builtin command - If you don't know about this yet, why are you reading this page?
- http://mywiki.wooledge.org/BashFAQ/001 (http://mywiki.wooledge.org/BashFAQ/001) - It's FAQ () 1 for a reason.

Discussion

commands/builtin/mapfile.txt Last modified: 2013/08/19 08:09 by ormaaj

This site is supported by Performing Databases - your experts for database administration

Bash Hackers Wiki

Except where otherwise noted, content on this wiki is licensed under the following license:
GNU Free Documentation License 1.3