Command-line arguments processing in bash

Matěj Týč, OpenAlt 2017

Outline

- How people tend to design interface of their bash scripts?
- What exactly is command-line interface?
- How is bash different from other languages?
- What is the bash way?
- What really is the bash way.

Bash scripts

Postgres backup script from postgres wiki

pg_backup.sh

```
#!/bin/bash

############################
####### LOAD CONFIG #######
############################

while [ $# -gt 0 ]; do
        case $1 in
                -c)
                        if [ -r "$2" ]; then
                                source "$2"
                                shift 2
                        else
                                ${ECHO} "Unreadable config file \"$2\"" 1>&2
                                exit 1
                        fi
                        ;;
                *)
                        ${ECHO} "Unknown Option \"$1\"" 1>&2
                        exit 2
                        ;;
        esac
done

if [ $# = 0 ]; then
        SCRIPTPATH=$(cd ${0%/*} && pwd -P)
        source $SCRIPTPATH/pg_backup.config
```

Bash scripts

The bash docker (bocker) – example of a good script (as Awesome Bash says).

```
function bocker_commit() { #HELP Commit a container to an image:\nBOCKER commit <cont
        [[ "$(bocker_check "$1")" == 1 ]] && echo "No container named '$1' exists" &
        [[ "$(bocker_check "$2")" == 1 ]] && echo "No image named '$2' exists" && ex
        bocker_rm "$2" && btrfs subvolume snapshot "$btrfs_path/$1" "$btrfs_path/$2"
        echo "Created: $2"
}

function bocker_help() { #HELP Display this message:\nBOCKER help
        sed -n "s/^.*#HELP\\s//p;" < "$1" | sed "s/\\\\n/\n\t/g;s/$/\n/;s!BOCKER!${1
}

[[ -z "${1-}" ]] && bocker_help "$0"
case $1 in
        pull|init|rm|images|ps|run|exec|logs|commit) bocker_"$1" "${@:2}" ;;
        *) bocker_help "$0" ;;
esac
```

Bash scripts

Ways how to pass data to your script:

- Env variables: `LS_LONG=true ls`
- “Config” files: `ls ls_config_long.cfg`
- **Command-line arguments**: `ls --long`

Command-line interface

Let’s write an `ls` help script `ls-help`. In `bash`!

- ```
  # default argument
  $ ls-help
  > List all files in directory .
  ```

- ```
  # Multiple arguments,
  # the default is replaced
  $ ls-help foo bar
  > List all files in directories ‘foo’, ‘bar’
  ```

Command-line interface

Let’s write an `ls` help script `ls-help`. In bash!

- ```
  # Short boolean option
  $ ls-help -A foo bar
  > List all files in directories ‘foo’, ‘bar’
  > Do not list implicit ‘.’ and ‘..’
  ```
- ```
  # Long boolean option, reversed order
  # does the same
  $ ls-help foo bar --almost-all
  > List all files in directories ‘foo’, ‘bar’
  > Do not list implicit ‘.’ and ‘..’
  ```

Command-line interface

Let’s write an `ls` help script `ls-help`. In `bash`!

- `# Long option that takes value`
  ```
  $ ls-help --width 10
  > List all files in directory ‘.’
  > Pad the width to 10 terminal columns
  ```

- `# Separated by = is the same`
  ```
  $ ls-help --width=10
  > List all files in directory ‘.’
  > Pad the width to 10 terminal columns
  ```

Command-line interface

Let’s write an `ls` help script `ls-help`. In bash!

- `# Short option and value can be stacked`
  `$ ls-help -w10`
  `> List all files in directory ‘.’`
  `> Pad the width to 10 terminal columns`

- `# Short options can be stacked`
  `$ ls-help -Aw10`
  `> List all files in directory ‘.’`
  `> Do not list implicit ‘.’ and ‘..’`
  `> Pad the width to 10 terminal columns`

Command-line interface

Let’s write an `ls` help script `ls-help`. In bash!

- ```
  # Optional and positional arguments
  # can be sparated by --
  $ ls-help -A -- -w10
  > List all files in directory ‘-w10’
  > Do not list implicit ‘.’ and ‘..’
  ```

- ```
  # Some arguments can be appended to a list
  $ ls-help -I ‘*.bak’ -I ‘*~’
  > List all files in directory ‘.’
  > Do not list item that match shell patterns
  ‘*.bak’ and ‘*~’
  ```

Command-line interface

Let’s write an `ls` help script `ls-help`. In `bash`!

- ```
  # We want a nice help message
  $ ls-help -h
  > ls-help [-h] [-A|--almost-all] ...
  >  -w|--width <int>: Padded width
  > ...
  ```
  ```

Command-line interface

What defines command-line interface:

- **Positional arguments**: Strings that have meaning on their own, is the position on command-line that identifies them
- **Optional argumetns**: They are either options or option-value pairs. Options are prefixed by single or double dash.

- `$ ls -w 10 something`
- `$ ls -A 10 something`

Command-line interface

What defines command-line interface:

POSIX standard:

```
$ script [opt. args] [--] pos. args
```

- Only short options were allowed.
- Order matters – no optional args after first positional arg

Command-line interface

What defines command-line interface:

GNU standard:

```
$ script [opt. args] [--] pos. args
```

- Same as POSIX, but more permissive.
- Long options were allowed.
- Order doesn’t matter – optional args may follow positional and still be treated as optional.

Non-bash

There are plenty of arg parsing libraries:

Language	Utility
Python	`Python argparse`
Perl	`Getopt::Long`
C	`Getopt, popt, C argparse`
C++	`Boost::ProgramOptions, QCommandLineParser`
Lua	`Lua Argparse`
Go	`Flag`

Bash experts

How do I parse command line arguments in Bash?

846

463

Say, I have a script that gets called with this line:

```
./myscript -vfd ./foo/bar/someFile -o /fizz/someOtherFile
```

or this one:

```
./myscript -v -f -d -o /fizz/someOtherFile ./foo/bar/someFile
```

What's the accepted way of parsing this such that in each case (or some combination of the two) `$v`, `$f`, and `$d` will all be set to `true` and `$outFile` will be equal to `/fizz/someOtherFile` ?

bash | command-line | scripting | arguments

Bash experts

1246

Preferred Method: Using straight bash without getopt[s]

I originally answered the question as the OP asked. This Q/A is getting a lot of attention, so I should also offer the non-magic way to do this. I'm going to expand upon guneysus's answer to fix the nasty sed and include Tobias Kienzler's suggestion.

Two of the most common ways to pass key value pair arguments are:

Straight Bash Space Separated

Usage `./myscript.sh -e conf -s /etc -l /usr/lib /etc/hosts`

```
#!/bin/bash
# Use -gt 1 to consume two arguments per pass in the loop (e.g. each
# argument has a corresponding value to go with it).
# Use -gt 0 to consume one or more arguments per pass in the loop (e.g.
```

Bash experts

How can I handle command-line arguments (options) to my script easily?

Well, that depends a great deal on what you want to do with them. There are several approaches, each with its strengths and weaknesses.

Contents

Manual loop

Manually parsing options without the use of a specialized function is the most flexible approach, and is sufficient for most simple scripts.

This example will handle a combination of short (POSIX) and long "GNU style" options with option arguments. Notice how both `--file FILE` and
scripts may also use functions and local variables, which can greatly improve your code. This example however illustrates a strictly POSIX co

Toggle line numbers

```
#!/bin/sh
# POSIX

# Reset all variables that might be set
file=
verbose=0 # Variables to be evaluated as shell arithmetic should be initialized to a default or validated

while :; do
    case $1 in
        -h|-\?|--help)   # Call a "show_help" function to display a synopsis, then exit.
            show_help
```

Non-bash

There are plenty of arg parsing libraries:

Library	Family	Note
Argbash	code generator	???
argparse-bash	module	Uses Python
EasyOptions	module	Lacks features
getopt	executable	Not portable
getopts	builtin	Short options only
shflags	library	Uses getopt
Bash-modules argument	module	Lacks features
...		

Why argbash

- Modules
  - are pain to distribute for users.
  - create a huge external dependency.
  - usually have bad documentation.
- Builtins are horrible to use,
- but still better than 100% manual parsing.

Why argbash

- Code generators:
  - Need to be installed.
  - Have to be able to regenerate modified script.
  - Should better generate nice code.
- You still have to ship the generated code.

Why use argbash

- Generated code is a template.
- Generates simple code for simple scripts, complex code for complex ones.
- Available as:
  - Local install: Github, Fedora, AUR
  - Docker image: Search the docker hub
  - Online service https://argbash.io/generate
- Feature-rich and documented.

DEMO

Clone/download sample files from

https://github.com/matejak/argbash-demo

Thank you attention!

Don’t forget to **`https://argbash.io`**

Any questions?