Heather Lapointe Follow
Feb 4 · 3 min read · Listen

Error Handling in Bash

If you are going to write Bash, it should be robust. This covers a few robustness patterns as well as a few pitfalls you may run into while writing your scripts.

Return Codes

As a quick Unix refresher, it is good to know that most programs have a return code when they exit. They only do not when they don't exit.

The universal success error code is `0`. Almost all other exit codes indicate some kind of error.

Explicit Exit

You can use the `exit $rc` to end your script with a specified exit code.

An example of a successful exit would be `exit 0`, while an unsuccessful example may be `exit 1`.

There are plenty of other implicit cases that can trigger exits as well.

Note that if you are using a `source`'d file, you probably do not want to use exit at all! This is because `exit` will terminate the active subshell (which could be the parent script, or even the terminal you are running in!).

Checking a Return Code (RC)

In Bash, the `$?` variable (dollar-sign + question mark) refers to the return code of the last executed command/pipeline.

```
#!/usr/bin/env bash
set +e

true
# prints: true rc=0
echo "true rc=$?"

false
# prints: false rc=1
echo "false rc=$?"
```

check_rc.sh hosted with ❤ by GitHub view raw

`$PIPESTATUS` also contains status codes as an array for a pipeline, but **this behavior is Bash specific.**

```
#!/usr/bin/env bash
set +e

true | true | false | true
# prints: 0 0 1 0
echo "${PIPESTATUS[@]}"
```

check_pipestatus.sh hosted with ❤ by GitHub view raw

Implicit Exit

The return code of the last call of a script is the implicit exit code.

```
#!/usr/bin/env bash
set +e

true

false

true
# implicit rc=0
```

implicit_exit_0.sh hosted with ❤ by **GitHub** **view raw**

```
#!/usr/bin/env bash
set +e

true

false
# implicit rc=1
```

implicit_exit_1.sh hosted with ❤ by **GitHub** **view raw**

Error Handling

Error handlers can be enabled in Bash to perform some actions when a command or pipeline fails with a non-zero return code.

Ignored Conditions

There are a few conditions which would not trigger error handlers by design.

These are mostly the condition parts of conditionals or loops and the non-last part of an AND (`&&`) or OR (`||`) list. You can think of these as the parts where the return code is already handled by something else.

```bash
#!/usr/bin/env bash
set +e

# show the failed command
trap 'echo $BASH_COMMAND' ERR

# print: false
false

# no print
if false; then
    echo "won't run"
fi

# no print
while false; then
    echo "won't run"
fi

# no print
false || true

# no print
! false
```

ignored_errors.sh hosted with ❤ by GitHub view raw

errexit Flag (set -e)

If you are writing a script, you may have seen some variant of `set -e` floating around. This tells your script that you should stop execution when you run any unhandled command with a non-zero return code.

Tip: you can use `help set` to view set flags in your bash shell.

```
#!/usr/bin/env bash
set -e

true
echo "first true ran"

# script aborts here with rc=1
false
echo "first false didn't run"

true
echo "second false didn't run"
```

errexit.sh hosted with ❤ by GitHub view raw

pipefail flag (set -o pipefail)

This flag causes a pipeline to fail with the last non-zero exit code if there is one. The default behavior is to only return with the return code of the last command.

```
#!/usr/bin/env bash
set +e
# enable default pipeline behavior
set +o pipefail

false | true
# print: 0
echo $?

# enable pipefail
set -o pipefail

false | true
# print: 1
echo $?
```

pipefail.sh hosted with ❤ by GitHub view raw

ERR trap

You can use an ERR trap to handle a failed command or pipeline.

```
#!/usr/bin/env bash
set +e

# show the command and return code
trap 'echo $BASH_COMMAND: $?' ERR

# no print
true

# print: false: 1
false

# print: ( exit 3 ): 3
(exit 3)

# no print
false || true

# print: false: 1
true && false
```

err_trap.sh hosted with ❤ by GitHub view raw

Bash Traceback

You can combine some of these topics into a traceback with some special bash variables that track state:

- FUNCNAME: Array of called functions (0 = current func, last = “main”)
- BASH_SOURCE: Source filename (corresponding to one more than frame)
- BASH_LINENO: Source line number (corresponding to frame)
- BASH_COMMAND: The current command at the time of trap

The following implements a helpful `_show_traceback` and `_register_traceback` that can be inserted into a program and modified as necessary:

```bash
#!/usr/bin/env bash
# exit on error (errexit)
set -e
# pass ERR trap to subshells (errtrace)
set -E

# Callback function for when set -e (ERREXIT) is triggered)
# This shows a small stacktrace for the current shell along with the failing
# function call.
function _show_traceback() {
    # The very first line captures the return code in $?.
    local rc=$?
    # We pass "${BASH_SOURCE[0]}" and $LINENO (note: not $BASH_LINENO) from
    # the trap so they refer to the failing call before this is run.
    # Otherwise, they refer to this _show_traceback itself.
    local fail_file=$1
    local file_line=$2
    # traceback locals
    local line_idx;
    local filename;
    # FUNCNAME is an array of functions
    local frame=${#FUNCNAME[@]}
    # skip "main" frame
    frame=$(( frame - 1 ))

    _warning "Subcommand failed with code=$rc. Bash traceback:"
    # Roughly an enumeration like::
    #   for frame_id, func in reversed(enumerate(FUNCNAME[1:])):
    #       filename = BASH_SOURCE[frame_id+1]
    #       line_idx = BASH_LINENO[frame_id]
    while [ "$frame" -gt 1 ]; do
        # BASH_SOURCE is +1 from BASH_LINENO and FUNCNAME index
        filename="${BASH_SOURCE[$frame]:-"<script>"}"
        # decrement after getting source, which is +1 of the rest
        frame=$(( frame - 1 ))
        line_idx="${BASH_LINENO[$frame]}"
        func="${FUNCNAME[$frame]}"
        _log "In $filename, line $line_idx:"
        _log "[#$frame]\t$func"
    done
    # frame 0 is _show_traceback, so we instead show $1 and $2 from our trap
    _log "In $fail_file, line $file_line:"
    _error "\t$BASH_COMMAND"
    _error "\t^-- returned $rc"
    _log ""
    return "$rc"
}
```

```
function _register_traceback() {
    # Cause shell to exit whenever any command fails.
    # Allows the ERR trap to be called before exit. (errexit)
    set -e
    # Ensure _show_traceback is propagated through functions (errtrace)
    set -E
    # Invoke _show_traceback whenever a failure occurs (via set -e)
    # Pass the current script and lineno (man (1) bash for LINENO usage)
    trap '_show_traceback "${BASH_SOURCE[0]}" "$LINENO"' ERR
}
```

Linux Bash Shell Scripting

```
# $* is the message to be echo'd.
function _log() {
    # >&2 means that the default (stdout) is being redirected (>) to &2 (stderr)
    echo -e "$@" >&2
}

# Write a message to stderr with Yellow foreground. WARNING: will be prepended with a
#   $* - Message
function _warning() {
    _log "\\033[1;33m\\033[41mWARNING\\033[49m: $*\\033[0m"
```



```
#   $* - Message
function _error() {
    _log "\\033[1;91mERROR: $*\\033[0m"
}


function _test_something_that_fails() {
    false  # a failing exit code
}

function _test_subfunction() {
    _test_something_that_fails  # this is where we fail
    echo "success" # we shouldn't get here
}

# Register our error handler
_register_traceback
# Should generate a traceback:
# WARNING: Subcommand failed with code=1. Bash traceback:
# In bash_traceback.sh, line 101:
# [#2]    _test_subfunction
```

```
# In bash_traceback.sh, line 86:
# [#1]    _test_something_that_fails
# In bash_traceback.sh, line 82:
# ERROR:  false
# ERROR:  ^-- returned 1
_test_subfunction
```