You are here / ft / Syntax / Redirection

Trace / Redirection

< y 🔀 😝 🖨

Redirection

Fix me: To be continued

Redirection makes it possible to control where the output of a command goes to, and where the

input of a command comes from. It's a mighty tool that, together with pipelines, makes the shell powerful. The redirection operators are checked whenever a simple command is about to be executed.

Under normal circumstances, there are 3 files open, accessible by the file descriptors 0, 1 and 2, all connected to your terminal:

Name FD **Description** stdin standard input stream (e.g. keyboard) standard output stream (e.g. monitor) stdout 1 2 standard error output stream (usually also on monitor) stderr

preferred UNIX®-FAQ for details, I'm too lazy to explain what a terminal is 100 Both, stdout and stderr are output file descriptors. Their difference is the convention that a program outputs payload on stdout and diagnostic- and error-messages on stderr. If you write a script that outputs error messages, please make sure you follow this convention!

The terms "monitor" and "keyboard" refer to the same device, the **terminal** here. Check your

Whenever you **name** such a filedescriptor, i.e. you want to redirect this descriptor, you just use the number: # this executes the cat-command and redirects its error messages (stderr) to the bit bucket

cat some_file.txt 2>/dev/null Whenever you **reference** a descriptor, to point to its current target file, then you use a " & " followed by a the descriptor number:

this executes the echo-command and redirects its normal output (stdout) to the standard error target echo "There was an error" 1>&2

The redirection operation can be **anywhere** in a simple command, so these examples are equivalent:

>new.txt cat foo.txt bar.txt

Every redirection operator takes one or two words as operands. If you have to use operands (e.g. filenames to redirect to) that contain spaces you must quote them!

Valid redirection targets and sources

Syntax Description

everything you can reference in the filesystem)

	everything you can reference in the mesystem,
&N	references the current target/source of the filedescriptor N ("duplicates" the filedescriptor)
&-	closes the redirected filedescriptor, useful instead of > /dev/null constructs (> &-)
/dev/fd/N	duplicates the filedescriptor N, if N is a valid integer
/dev/stdin	duplicates filedescriptor 0 (stdin)
/dev/stdout	duplicates filedescriptor 1 (stdout)
/dev/stderr	duplicates filedescriptor 2 (stderr)
/dev/tcp/HOST/PORT	assuming H0ST is a valid hostname or IP address, and P0RT is a valid port number or service name: redirect
	from/to the corresponding TCP socket
/dev/udp/H0ST/P0RT	assuming H0ST is a valid hostname or IP address, and P0RT is a valid port number or service name: redirect
	from/to the corresponding UDP socket
If a target/source specification fails to open, the whole redirection operation fails. Avoid referencing file descriptors above 9, since you may collide with file descriptors Bash uses internally.	
•	

This redirects the file descriptor number N to the target TARGET. If N is omitted, stdout is assumed (FD 1). The TARGET is truncated

If the option noclobber is set with the set builtin, with cause the redirection to fail, when TARGET names a regular file that already exists.

before writing starts.

N > TARGET

You can manually override that behaviour by forcing overwrite with the redirection operator > | instead of > .

Appending redirected output

This redirects the file descriptor number N to the target TARGET. If N is omitted, stdout is assumed (FD 1). The TARGET is not truncated before writing starts.

>> FILE 2>&1

Redirecting output and error output

>& TARGET

> TARGET 2>&1

This syntax is deprecated and should not be used. See the page about obsolete and deprecated syntax.

&>> FILE

Transporting stdout and stderr through a pipe

Redirecting input

The input descriptor N uses SOURCE as its data source. If N is omitted, filedescriptor 0 (stdin) is assumed. Here documents

TAG

TAG

Hehe...

E0F

A here-document is an input redirection using source data specified directly at the command line (or in the script), no "external" source. The redirection-operator << is used together with a tag TAG that's used to mark the end of input later:

As you see, substitutions are possible. To be precise, the following substitutions and expansions are performed in the here-document data: • Parameter expansion Command substitution Arithmetic expansion

when end-of-file is seen before the tag is reached.

cat <<< "Hello world... \$NAME is here..."</pre>

You can avoid that by quoting the tag:

No help available yet for \$PROGRAM.

Last but not least, if the redirection operator << is followed by a - (dash), all leading TAB from the document data will be ignored. This might be useful to have optical nice code also when using here-documents.

It seems that here-documents (tested on versions 1.14.7, 2.05b and 3.1.17) are correctly terminated when there is an EOF before

🕠 the end-of-here-document tag. The reason is unknown, but it seems to be done on purpose. Bash 4 introduced a warning message

<<< WORD

The here-strings are a variation of the here-documents. The word WORD is taken for the input redirection:

The tag you use **must** be the only word in the line, to be recognized as end-of-here-document marker.

Just beware to quote the WORD if it contains spaces. Otherwise the rest will be given as normal parameters.

More redirection operations can occur in a line of course. The order is **important**! They're evaluated from **left to right**. If you want to redirect both, stderr and stdout to the same file (like /dev/null, to hide it), this is the wrong way:

Here strings

Multiple redirections

Why? Relatively easy:

• 2>&1 redirects stderr away from the terminal to the target for stdout: the terminal (again...) 1>/dev/null redirects stdout away from your terminal to the file /dev/null What remains? stdout goes to /dev/null, stderr still (or better: "again") goes to the terminal. You have to swap the order to make it

do what you want:

Examples

initially, stdout points to your terminal (you read it)

same applies to stderr, it's connected to your terminal

command >/dev/null 2>&1

- - Internal: The noclobber option
 - Internal: Simple commands parsing and execution

Internal: Illustrated Redirection Tutorial

Internal: Nonportable syntax and command uses

You could leave a comment if you were logged in.

Internal: Obsolete and deprecated syntax

 \odot 个

[[syntax:redirection]]

≡ Table of Contents **▼** Redirection Valid redirection targets and sources Redirecting output Appending redirected output Redirecting output and error output Appending redirected output and error output Transporting stdout and stderr through a pipe Redirecting input

Here documents Here strings Multiple redirections **Examples** See also

Discussion

cat foo.txt bar.txt >new.txt cat >new.txt foo.txt bar.txt

This syntax is recognized whenever a TARGET or a SOURCE specification (like below in the details descriptions) is used.

FILENAME references a normal, ordinary filename from the filesystem (which can of course be a FIFO, too. Simply

C Redirecting output

N >> TARGET

&> TARGET

This special syntax redirects both, stdout and stderr to the specified target. It's equivalent to

Since Bash4, there's &>>TARGET, which is equivalent to >> TARGET 2>&1.

Appending redirected output and error output To append the cumulative redirection of stdout and stderr to a file you simply do

COMMAND1 2>&1 | COMMAND2 COMMAND1 |& COMMAND2

N < SOURCE

<<TAG

<<-TAG

display help cat <<E0F Sorry

cat <<"E0F" This won't be expanded: \$PATH E0F

The here-string will append a newline (\n) to the data.

you can test with it { echo OUTPUT; echo ERRORS >&2; } 2>&1 1>/dev/null

{ echo OUTPUT; echo ERRORS >&2; } is to simulate something that outputs to STDOUT and STDERR

echo OUTPUT; echo ERRORS >&2; } 1>/dev/null 2>&1

How to make a program quiet (assuming all output goes to STDOUT and STDERR?

See also

 Internal: The exec builtin command Internal: Process substitution syntax