

NAME

pipeline – logical data pipeliner

SYNOPSIS

pipeline [-cdevsv] [-i file] [-o file] [--] [-c] cmdA [{conj}] [-c] cmdB [...]

conjunctions are: // && / ? -or -and -then -if

DESCRIPTION

Pipeline is a master control program for a data flow pipeline, in the manner of shell.

Pipeline essentially combines the Bourne shell notions of the pipeline:

```
cmdA | cmdB | cmdC
```

and of the logical command list:

```
cmdA || cmdB && cmdC
```

We say that these are combined because **pipeline** *always* passes a program's stdout to the next program's stdin, akin to the shell's pipeline. The difference is that stages of the pipeline can be excluded from execution based on the exit status of previous stages, as in the logical command list. Stages can also be excluded if no input is present from the previous stage.

Pipeline reads from its command line a sequence of pipelined stages. Each stage's exit status produces a *truth state*. The *or* and *and* conjunctions decide, based on this truth state, whether the next stage of the pipeline should execute.

Take, for example, this simple command:

```
pipeline cmdA || cmdB && cmdC
```

or, if you want to escape shell syntax mutilation:

```
pipeline cmdA -or cmdB -and cmdC
```

cmdA first executes, with *stdin* open to */dev/null*. (To force it to read *stdin* from **pipeline**'s *stdin*, use the *-i* option.) It does whatever it does; the output is accumulated inside the **pipeline** process. When *cmdA* completes, its exit status is evaluated. If it failed, *cmdB* is launched, and *cmdB* receives on *stdin* the accumulated stdout of *cmdA*. If *cmdA* succeeded, then *cmdB* is skipped, and stdout is passed to *stdin* of *cmdC* instead. If *cmdB* runs, and succeeds, that also is passed to *cmdC*.

The logical flow is exactly like Bourne shell's. Subject to that flow, the pipeline works similarly, with one significant difference: Bourne's approach is to immediately launch *cmdC*, *cmdB*, and *cmdA*, in that order. **Pipeline** will never start *cmdB* if *cmdA* fails. This can be a useful model.

The // and *-or* conjunctions are identical in function. // exists to make **pipeline**'s syntax read like Bourne's, but if you're running **pipeline** from Bourne, this can prove troublesome. *-or* is provided as an alternate. && and *-and* are likewise identical, as are / and *-then*.

The / or *-then* conjunction is a simple pipe. It disregards previous truth state; it always runs; and it always receives the previous stage's output as input.

The ? or *-if* conjunction disregards truth state, but considers instead whether the previous stage produced any output. If output was produced, the following stage will run. If no output was produced, the stage is ignored. (By implication, all following stages are also ignored, and it only makes sense to use one *-if* in a single invocation of **pipeline**.)

OPTIONS

The following options are supported:

- d** Produce *debugging* (or tracing) output: logical flow, operation, truth state, commands run, exit statuses.
- c** Causes all stages to be run using *sh -c* instead of being run directly via *execv()*. This allows you to use a variety of shell syntax in your pipeline stages, including input and output redirections.

- e** Merges the *stderr* stream with *stdout* for each stage. This is equivalent to running with *-c*, and adding **2>&1** to each stage.
- s** Suppress output, if the final truth state is *false*.
- v** Invert the final truth state, producing *true* for *false* and vice versa. This occurs immediately before output is produced or suppressed, so *-s* and *-sv* have precisely opposite effects.
- V** Shows *version* information -- the Id: tag(s) from CVS.
- i file** Source initial input data from *file*. If *file* is "-", source from *stdin*. Without any *-i* option, initial input to the first stage is from */dev/null*.
- o file** Sink final output data to *file*, rather than to *stdout*. (If *file* is "-", *stdout* is used anyway.)

Additionally, a *-c* can be given as the first term of any stage, and that stage will be executed via *sh -c*. Other stages will run normally.

EXAMPLES

Here's a simple *crontab* entry for conditional mailing:

```
0 * * * * /path/to/pipeline -sv somecommand -or mailx -s "oopsie" admin
```

This executes **somecommand**. If its exit status is true (success), then nothing happens, and any output produced is suppressed. But if it fails, output is released to **mailx**, which sends a nice and tidy message to *admin*, in place of the gruesome default **cron** mail.])

Maybe sometime I'll add more examples.

EXIT STATUS

The following exit values are returned:

- 0** By some path, the pipeline completed successfully. If all conjunctions in the pipeline were *and* conjunctions, then everything completed successfully. If some were *or* conjunctions, then some commands might have failed; but their failure was anticipated.
- 2** Command syntax was incorrect; refer to usage (**pipeline -h**).
- 10** The input file could not be opened.
- 15** The output file could not be opened or created.
- 20** Some stage of the pipeline could not be executed.

EMBED

Pipeline is compiled with **embed**. To extract the full source tree (including **embed** itself), simply run:

```
pipeline -explode # for a random subdirectory in $cwd
```

or

```
pipeline -explode=/other/path # for a specific location
```

Or, to list the files **embedded** in the program, run:

```
pipeline -list
```

ERRATA

Pipeline used to read *stdin* by default, passing the input to the first-stage process, unless the *-n* option was given. This made sense and was consistent with the model of mirroring Bourne's chaining functionality, but in practice most pipelines didn't care about existing *stdin* data: they were only concerned with whatever the first stage process produced itself.

The result was that if one forgot to use the *-n* option, **pipeline** would sit around waiting for *stdin* to close before ever launching the first-stage process. In almost all cases, this behavior was simply irritating. It seemed a good principle that the behavior most often desired should be the easiest to obtain, so this design was reversed. Removing the *-n* option and making its behavior the default (with *-i* - as an override) makes **pipeline** a little less logical, but more in line with expectation, and hopefully less aggravating over all.

LICENSE

Copyright (c) 2003, David Champion <dgc@uchicago.edu>

This is Free Software. It is released under the terms given in the accompanying file "LICENSE", which I cribbed from the Massachusetts Institute of Technology by way of <http://www.opensource.org/licenses/mit-license.php>.

BUGS

(*n.*) Creeping little problems that inhabit all software, no matter how conscientiously you test it.

SEE ALSO

sh(1), **cron**(1M, 8)

HAIKU

Bourne shell conditions
Are not always sufficient.
Specially with **cron**.