链滴

Linux Shell 操作符

作者：liudongdong

原文链接：https://ld246.com/article/1504236256775

来源网站：链滴

These are called shell operators and yes, there are more of them. I will give a brief overvi
w of the most common among the two major classes, control operators and <a href="https://
d246.com/forward?goto=https%3A%2F%2Fwww.gnu.org%2Fsoftware%2Fbash%2Fmanual%2
bashref.html%23Redirections" target="_blank" rel="nofollow ugc">redirection operators
 and how they work with respect to the bash shell.
<h2 id="A--Control-operators">A. Control operators</h2>
These are tokens that perform control functions, one of <code>||</code>, <code>!</co
e>, <code>&&</code>, <code>&</code>, <code>;</code>, <code>;;</co
e>, <code>|</code>, <code>|&</code>, <code>(</code>, or <code>)</code>.
<h3 id="A-1-List-terminators">A.1 List terminators</h3>

<code>;</code> : Will run one command after another has finished, irrespective of the
utcome of the first.
<pre><code class="highlight-chroma"><span class="highlight
cl">command1 ; command2
</code></pre>
First <code>command1</code> is run, in the foreground, and once it has finished, <cod
>command2</code> will be run.
A newline that isn't in a string literal or after certain keywords is not equival
nt to the semicolon operator. A list of <code>;</code> delimited simple commands is still a
em>list - as in the shell's parser must still continue to read in the simple commands th
t follow a <code>;</code> delimited simple command before executing, whereas a newline
an delimit an entire command list - or list of lists. The difference is subtle, but complicated: gi
en the shell has no previous imperative for reading in data following a newline, the newline m
rks a point where the shell can begin to evaluate the simple commands it has already read in,
whereas a <code>;</code> semi-colon does not.

<code>&</code> : This will run a command in the background, allowing you to co
tinue working in the same shell.
<pre><code class="highlight-chroma"><span class="highlight
cl"> command1 & command2
</code></pre>
Here, <code>command1</code> is launched in the background and <code>command
</code> starts running in the foreground immediately, without waiting for <code>command
</code> to exit.
A newline after <code>command1</code> is optional.

<h3 id="A-2-Logical-operators">A.2 Logical operators</h3>

<code>&&</code> : Used to build AND lists, it allows you to run one comma
d only if another exited successfully.
<pre><code class="highlight-chroma"><span class="highlight
cl"> command1 && command2
</code></pre>
Here, <code>command2</code> will run after <code>command1</code> has finished
nd only if <code>command1</code> was successful (if its exit code was 0). Bot
 commands are run in the foreground.
This command can also be written
<pre><code class="highlight-chroma"><span class="highlight
cl">if command1

then command2
else false
fi
</code></pre>
or simply <code>if command1; then command2; fi</code> if the return status is ignored

<code>||</code> : Used to build OR lists, it allows you to run one command only if anot
er exited unsuccessfully.
<pre><code class="highlight-chroma"><span class="highlight
cl"> command1 || command2
</code></pre>
Here, <code>command2</code> will only run if <code>command1</code> failed (if it
eturned an exit status other than 0). Both commands are run in the foreground.
This command can also be written
<pre><code class="highlight-chroma"><span class="highlight
cl">if command1
then true
else command2
fi
</code></pre>
or in a shorter way <code>if ! command1; then command2; fi</code>.
Note that <code>&&</code> and <code>||</code> are left-associative; see
a href="https://ld246.com/forward?goto=https%3A%2F%2Funix.stackexchange.com%2Fquest
ons%2F88850%2Fprecedence-of-the-shell-logical-operators" target="_blank" rel="nofollow
gc">Precedence of the shell logical operators &, ||for more information.

<code>!</code>: This is the “not” operator, used to negate the return status of a co
mand — return 0 if the command returns a nonzero status, return 1 if it returns the status 0.<
p>
<pre><code class="highlight-chroma"><span class="highlight
cl">! command1
</code></pre>

<h3 id="A-3-Pipe-operator">A.3 Pipe operator</h3>

<code>|</code> : The pipe operator, it passes the output of one command as input to a
other. A command built from the pipe operator is called a <a href="https://ld246.com/forwar
?goto=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FPipeline_%28Unix%29" target="_blank"
el="nofollow ugc">pipeline.
<pre><code class="highlight-chroma"><span class="highlight
cl"> command1 | command2
</code></pre>
Any output printed by <code>command1</code> is passed as input to <code>comma
d2</code>.

<code>|&</code> : This is a shorthand for <code>2>&1 |</code> in bash
nd zsh. It passes both standard output and standard error of one command as input to anoth
r.

<pre><code class="highlight-chroma"><span class="highlight
cl">command1 |& command2
</code></pre>

<h3 id="A-4-Other-list-punctuation">A.4 Other list punctuation</h3>
<code>;;</code> is used solely to mark the end of a <a href="https://ld246.com/forwar
?goto=https%3A%2F%2Fwww.gnu.org%2Fsoftware%2Fbash%2Fmanual%2Fbashref.html%23
onditional-Constructs" target="_blank" rel="nofollow ugc">case statement. Ksh, bash a
d zsh also support <code>;&</code> to fall through to the next case and <code>;;&a
p;</code> (not in ATT ksh) to go on and test subsequent cases.
<code>(</code> and <code>)</code> are used to <a href="https://ld246.com/forward
goto=https%3A%2F%2Fwww.gnu.org%2Fsoftware%2Fbash%2Fmanual%2Fbashref.html%23C
mmand-Grouping" target="_blank" rel="nofollow ugc">group commands and launch t
em in a subshell. <code>{</code> and <code>}</code> also group commands, but do not l
unch them in a subshell. See <a href="https://ld246.com/forward?goto=https%3A%2F%2Fsta
koverflow.com%2Fquestions%2F6270440%2Fsimple-logical-operators-in-bash%2F6270803%
36270803" target="_blank" rel="nofollow ugc">this answer for a discussion of the vario
s types of parentheses, brackets and braces in shell syntax.
<h2 id="B--Redirection-Operators">B. Redirection Operators</h2>
These allow you to control the input and output of your commands. They can appear an
where within a simple command or may follow a command. Redirections are processed in the
order they appear, from left to right.

<code><</code> : Gives input to a command.
<pre><code class="highlight-chroma"><span class="highlight
cl">command < file.txt
</code></pre>
The above will execute <code>command</code> on the contents of <code>file.txt</co
e>.

<code><></code> : same as above, but the file is open in read+write
mode instead of read-only:
<pre><code class="highlight-chroma"><span class="highlight
cl">command <> file.txt
</code></pre>
If the file doesn't exist, it will be created.
That operator is rarely used because commands generally only read from th
ir stdin, though <a href="https://ld246.com/forward?goto=https%3A%2F%2Funix.stackexcha
ge.com%2Fa%2F164449" target="_blank" rel="nofollow ugc">it can come handy in a number
of specific situations.

<code>></code> : Directs the output of a command into a file.
<pre><code class="highlight-chroma"><span class="highlight
cl">command > out.txt
</code></pre>
The above will save the output of <code>command</code> as <code>out.txt</code>. I
the file exists, its contents will be overwritten and if it does not exist it will be created.
This operator is also often used to choose whether something should be printed to <a hr
f="https://ld246.com/forward?goto=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FStandard_s
reams%23Standard_error_.28stderr.29" target="_blank" rel="nofollow ugc">standard error</

> or standard output:
<pre><code class="highlight-chroma">command >out.txt 2>error.txt
</code></pre>
In the example above, <code>></code> will redirect standard output and <code>2gt;</code> redirects standard error. Output can also be redirected using <code>1></cod> but, since this is the default, the <code>1</code> is usually omitted and it's written simply as <code>></code>.
So, to run <code>command</code> on <code>file.txt</code> and save its output in <ode>out.txt</code> and any error messages in <code>error.txt</code> you would run:</p

<pre><code class="highlight-chroma">command < file.txt > out.txt 2> error.txt
</code></pre>

<code>>|</code> : Does the same as <code>></code>, but will overwrite the target, even if the shell has been configured to refuse overwriting (with <code>set -C</code> or <code>set -o noclobber</code>).
<pre><code class="highlight-chroma">command >| out.txt
</code></pre>
If <code>out.txt</code> exists, the output of <code>command</code> will replace its ontent. If it does not exist it will be created.

<code>>></code> : Does the same as <code>></code>, except that if the target file exists, the new data are appended.
<pre><code class="highlight-chroma">command >> out.txt
</code></pre>
If <code>out.txt</code> exists, the output of <code>command</code> will be appended to it, after whatever is already in it. If it does not exist it will be created.

<code>&></code>, <code>>&</code>, <code>>>&</cod> and <code>&>></code> : (non-standard). Redirect both standard error and standard output, replacing or appending, respectively.
<pre><code class="highlight-chroma">command &> out.txt
</code></pre>
Both standard error and standard output of <code>command</code> will be saved in <ode>out.txt</code>, overwriting its contents or creating it if it doesn't exist.
<pre><code class="highlight-chroma">command &>> out.txt
</code></pre>
As above, except that if <code>out.txt</code> exists, the output and error of <code>cmmand</code> will be appended to it.
The <code>&></code> variant originates in <code>bash</code>, while the <cde>>&</code> variant comes from csh (decades earlier). They both conflict with oth r POSIX shell operators and should not be used in portable <code>sh</code> scripts.

<code><<</code> : A here document. It is often used to print multi-line strings.</
>
<pre><code class="highlight-chroma"><span class="highlight
cl"> command << WORD
     Text
 WORD
</code></pre>
Here, <code>command</code> will take everything until it finds the next occurrence of
<code>WORD</code>, <code>Text</code> in the example above, as input . While <code
WORD</code> is often <code>EoF</code> or variations thereof, it can be any alphanumeric
(and not only) string you like. When <code>WORD</code> is quoted, the text in the here do
ument is treated literally and no expansions are performed (on variables for example). If it is
nquoted, variables will be expanded. For more details, see the <a href="https://ld246.com/fo
ward?goto=https%3A%2F%2Fwww.gnu.org%2Fsoftware%2Fbash%2Fmanual%2Fbashref.htm
%23Here-Documents" target="_blank" rel="nofollow ugc">bash manual.
If you want to pipe the output of <code>command << WORD ... WORD</code> dir
ctly into another command or commands, you have to put the pipe on the same line as <cod
><< WORD</code>, you can't put it after the terminating WORD or on the line following
 For example:
<pre><code class="highlight-chroma"><span class="highlight
cl"> command << WORD | command2 | command3...
     Text
 WORD
</code></pre>

<code><<<</code> : Here strings, similar to here documents, but intended for a s
ngle line. These exist only in the Unix port or rc (where it originated), zsh, some implementati
ns of ksh, yash and bash.
<pre><code class="highlight-chroma"><span class="highlight
cl">command <<< WORD
</code></pre>
Whatever is given as <code>WORD</code> is expanded and its value is passed as input
o <code>command</code>. This is often used to pass the content of variables as input to a
ommand. For example:
<pre><code class="highlight-chroma"><span class="highlight
cl"> $ foo="bar"
 $ sed 's/a/A/' &lt
<< "$foo"
 bAr
 # as a short-cut f
r the standard:
 $ printf '%s\n' "$
oo" | sed 's/a/A/'
 bAr
 # or
 sed 's/a/A/' <&
t; EOF
 $foo
 EOF
</code></pre>

A few other operators (<code>>&-</code>, <code>x>&y</code> <co
e>x<&y</code>) can be used to close or duplicate file descriptors. For details on the
, please see the relevant section of your shell's manual (<a href="https://ld246.com/forward?
oto=https%3A%2F%2Fwww.gnu.org%2Fsoftware%2Fbash%2Fmanual%2Fbashref.html%23M
ving-File-Descriptors" target="_blank" rel="nofollow ugc">here for instance for bash).<
p>
That only covers the most common operators of Bourne-like shells. Some shells have a f
w additional redirection operators of their own.
Ksh, bash and zsh also have constructs <code><(...)</code>, <code>>(...)</code>
nd <code>=(...)</code> (that latter one in <code>zsh</code> only). These are not redirecti
ns, but <a href="https://ld246.com/forward?goto=http%3A%2F%2Fwww.gnu.org%2Fsoftwar
%2Fbash%2Fmanual%2Fbash.html%23Process-Substitution" target="_blank" rel="nofollow u
c">process substitution.
来源:stackexchange(<a href="https://ld246.com/forward?goto=https%3A%2F%2Funix.sta
kexchange.com%2Fquestions%2F159513%2Fwhat-are-the-shells-control-and-redirection-ope
ators" target="_blank" rel="nofollow ugc">https://unix.stackexchange.com/questions/159513
what-are-the-shells-control-and-redirection-operators)