Tree: 458626a814 hexo_blog / source / _posts / Bash文档7-重定向-命令执行-脚本.md

Find file Copy path

lifayi2008 test 3b8b69d on Dec 14, 2016

1 contributor

237 lines (145 sloc) | 18.3 KB

Raw Blame History

title	date	tags	categories
Bash文档7--重定向/命令执行/脚本	2016-09-13 07:02:54 -0700	bash	Bash

重定向

在命令执行之前，shell可以使用特定的语法来将其输入输出进行重定向。重定向可以让命令的文件句柄被复制，打开，关闭或者指向别的文件，可以改变命令要写入或者读取的文件。重定向也可以用来修改当前shell环境的文件句柄。下面的这些重定向操作符可以放在简单命令的任何地方：前置中间或者后面。重定向按照操作符出现的顺序从左向右进行

任何在重定向操作符之前加数字表示文件描述符的情况都可以使用`{varname}`来代替数字，这样shell会分配一个大于10的文件描述符给重定向操作，并且将这个数字赋给`varname`变量。有一种例外是`>&-`和`<&-`，如果上面两种形式前置一个数字或者变量表示文件描述符时，shell会关闭这个文件描述符或者关闭变量值对应的文件描述符

下面的描述中，如果文件描述符为空，则`<`和`>`分别表示前置一个文件描述符0或者1表示标准输入或者标准输出。

下面描述的操作元素`word`，除非特别指出，都要经历`brace expansion` `tilde expansion` `parameter expansion` `command subtitution` `arithmetic expansion` `quote removal` `filename expansion` `word splitting`。如果展开替换结果超过1个word，则shell会报错；这个结果要指代一个文件名

注意重定向的顺序非常重要。比如

```
ls > dirlist 2>&1
```

表示将ls命令的标准输出重定向到文件`dirlist`，然后将标准错误输出重定向到标准输出现在指向的地方(dirlist文件)

```
ls 2>&1 > dirlist
```

表示将ls命令的标准错误输出重定向到现在的标准输出指向的地方(terminal)，然后将标准输出重定向到文件`dirlist`

将下面的文件用作重定向操作时，有特殊的意义：

`/dev/fd/fd` 如果`fd`是一个合法整形，则复制文件描述符`fd` **`/dev/stdin`** 复制文件描述符0 **`/dev/stdout`** 复制文件描述符1 **`/dev/stderr`** 复制文件描述符2 **`/dev/tcp/host/port`** 如果host是一个合法主机或者Internet地址，

port是一个整形端口或者service名，则Bash会试图打开相对应的TCP socket `/dev/udp/host/port` 如果host是一个合法主机或者Internet地址，port是一个整形端口或者service名，则Bash会试图打开相对应的UDP socket

使用大于9的文件描述符需要小心，因为可能会和shell内部使用的文件描述符冲突

输入重定向

这表示命令在文件描述符 `n` 上打开文件 `word` 来准备读取， `word` 要经过前面提到的几种扩展，如果未指定 `n` 则默认是0表示将标准输入重定向为从 `word` 读 通常的格式是：

```
[n]<word
```

输出重定向

这表示命令在文件描述符n上打开文件 `word` 来准备写入， `word` 要经过前面提到的几种扩展，如果未指定 `n` 则默认是1表示将标准输出重定向到 `word` 。如果 `word` 指代文件不存在则会创建；如果存在则将文件截断为0，然后写入 通常的格式是：

```
[n]>[|]word
```

如果重定向操作符是 `>` ，并且shell选项 `noclobber` 启用，在打开文件 `word` 时如果文件存在并且是一个常规文件则shell会报错。如果操作符是 `>|` ，或者操作符是 `>` 并且 `noclobber` 选项未启用，在打开文件 `word` 时即使文件存在也进行截断写操作

输出重定向进行追加写

这表示命令在文件描述符 `n` 上打开文件 `word` 来进行准备追加， `word` 要经过前面提到的几种扩展，如果未指定 `n` 则默认是 `1` 表示将标准输出重定向到 `word` ；如果 `word` 指代文件不存在则会创建

```
通常的格式是：
[n]>>word
```

重定向标准输入和标准错误输出

下面这种结构可以将标准输出和标准错误输出都重定向到`word`，`word`要经过前面提到的几种扩展

```
有两种格式：
&>word
>&word
第一种是优先选择的，等同于
>word 2>&1
```

当使用第二种格式时，如果word扩展为`-`则表示关闭文件描述符`1`；如果`word`扩展为一个数字则表示复制文件描述符

重定向标准输入和标准错误输出进行追加写

```
&>>word
只有这一种形式等同于
word 2>&1
同上但是是追加写
```

文本重定向

这种方式的重定向可以让shell读取当前输入的内容，一直到一个顶格写的限定符出现。中间所有的行都被作为前面命令的标准输入

```
here document的格式为：
<<[-]wordhere-documentdelimiter
```

shell对`word`中不进行`parameter expansion` `command substiution` `arithmetic expansio` `filename expansion`。如果word中的任意字符加引号，并且执行完`quote removal`之后的结果是`delimiter`，则`here document`中的所有行不进行扩展。如果`word`未被引用，则`here document`中的所有行都进行`parameter expansion` `command substitution` `arithmetic expansion`，字符序列`\newline`被忽略，`\`只能用来引用字符`\` `$` ```

如果重定向操作符是`<<-`，则`here document`中和包含`delimiter`的每一行的前导`tab`都被从输入文本中去掉。这样在shell脚本中使用`here document`时可以和其他内容进行对齐

字符串重定向

这是here document的一种变体

```
<<< word
```

`word`经过`brace expansion` `tilde expansion` `parameter expansion` `command substitution` `arithmetic expansion`和`quote removal`。而`pathname expansion` `word splitting`则不进行。结果作为一个字符串从标准输入传给`command`

复制文件描述符

`[n]<&word`
用来复制输入文件描述符。如果`word`扩展为一个或者多个数字，则将这个文件描述符复制到n。如果`word`扩展后的数字指代的文件描述符不是一个已经为输入打开的文件描述符，则报错。如果`word`扩展为`-`则表示关闭文件描述符n。如果n为空则表示标准输入(文件描述符0)

`[n]>&word`
用来复制输出文件描述符。如果n未指定，则表示表示标准输出（文件描述符1）；如果`word`扩展后指代的文件描述符不是一个已经为输出打开的文件描述符，则报错。如果`word`扩展为`-`同意表示关闭文件描述符`n`。一种特殊情况是，如果n为空，并且`word`扩展后的结果不是数字或者`-`，则和3.6.4节一样，标准输出和标准错误输出被重定向到`word`

> 注意：复制操作的含义是`word`指代的文件描述符打开哪个文件，则在`n`文件描述符上也打开这个文件，这两次打开共享相同的偏移量

移动文件描述符

`[n]<&digit-`
将文件描述符`digit`移动到`n`，如果`n`为空则表示标准输入。`digit`复制为`n`之后被关闭

`[n]>&digit-`
将文件描述符`digit`移动到`n`，如果n为空则表示标准输出。

以读写的方式打开文件

```
[n]<>word
```

将在`n`上以读写的方式打开`word`指代的文件名，`word`要经过前面提到的扩展，如果`n`未指定则表示0。如果文件不存在则创建这个文件

执行命令

简单命令扩展

当执行一个简单命令时，shell从左到右进行如下扩展、变量分配和重定向

1. 语法分析结果为变量分配和重定向的words保留稍后进行处理
2. 不是变量分配和重定向的word按照前面章节的扩展规则进行扩展。扩展之后的第一个word作为命令名，后面其他的作为这个命令的参数
3. 按照前面章节中的重定向规则进行重定向操作
4. 变量赋值操作符`=`右边的`word`在赋值给变量之前要经过`tilde expansion` `parameter expansion` `command substitution` `arithmetic expansion` `qoute removal`

如果扩展的结果没有命令名，则变量的赋值影响当前shell环境。否则变量被加入到准备执行命令的子shell的环境中，而不会影响当前shell环境。任何试图给只读变量赋值的操作会报错，并且命令的退出状态为非0

如果扩展的结果没有命令名，则重定向被执行，但是不会影响当前的shell环境。重定向的错误会导致命令的退出状态为非0

如果扩展后有命令名，则按照下一节描述的方式执行。 否则的话命令退出。如果扩展中包含一个命令替换， 则整个命令的退出状态是进行替换的最后一个命令的退出状态。如果没有命令替换则命令扩展的退出状态是0

命令搜索和执行

命令被分隔为`words`之后，如果结果是一个简单命令和一串参数，则按照下面方式执行

1. 如果命令名不包含一个斜线，则shell尝试着去定位这个命令。如果和一个函数名相同，则按之前章节描述的函数调用的方法调用这个函数
2. 如果命令不匹配任何函数，则shell开始在内置命令中搜索这个命令。如果找到这执行这个内置命令
3. 如果命令名既不是一个函数也不是一个内置命令同时又不包含一个斜线的话，则shell在`$PATH`变量包含的目录中搜索这个命令名。bash使用一个hash表来记住可执行文件的路径名来避免每次都在文件系统中搜索；只有命令名没有在hash表中找到时才在这些路径中进行搜索。如果没有找到这个命令名，则shell执行预先定义的`command_not_found_handle`函数。如果这个函数存在，则这个函数替代命令来执行，命令的参数成为这个函数的参数，函数的退出状态作为执行这个命令的shell的退出状态。如果这个函数未定义则shell打印错误消息，并且退出状态为127

4. 如果命令被找到，或者命令中包含有一个或者多个斜线表示路径，则当前shell启动一个子shell来执行这个命令。参数0被设置为这个命令名，如果有参数的话，这些参数成为命令的参数
5. 如果命令因为非可执行文件格式而执行失败，并且命令名不是一个目录名的话，则shell假设这是一个`shell script`并且按照后面描述的`shell script`的执行方式来执行
6. 如果命令非异步执行，则shell等待命令执行完并且获取命令的退出状态

命令执行环境

Shell有一个执行环境(shell本身)，由下面这些部分组成：

- 从调用的shell中继承的打开文件，有可能已经使用内置命令exec加重定向操作改变
- 由`cd`或者`pushd`、`popd`命令设置的当前工作目录，或者是从调用shell中继承过来的
- 由内置命令`umask`设置的文件创建掩码，或者是从调用shell中继承的
- 使用内置trap命令设置的trap
- 使用set命令或者变量分配操作符设置的变量，或者是从调用的shell的环境中继承的
- 执行过程中定义的函数，或者是从调用的shell的环境中继承的
- 调用时设置的选项(默认的或者是命令行选项设置的)，或者是`set`命令设置的
- `shopt`命令设置的选项
- 使用`alias`命令定义的别名
- 各种进程ID，包括后台作业，`$$`的值，以及`$PPID`的值

当执行一个简单命令而不是内置或者函数时，命令会在一个单独的执行环境中被调用，这个环境包括下面这些部分。除非特别指出，这些都是从父shell中继承的：

- shell打开的文件，和使用重定向命令进行的任何重定向操作
- 当前工作目录
- 文件创建掩码

- 使用`export`命令导出的变量和函数；调用命令前的变量赋值也传给了子shell环境
- traps caught by the shell are reset to the values inherited from the shell's parent, and traps ignored by the shell are ignored

在这个单独的环境中执行的操作不会影响调用shell的环境

命令替换、使用小括号括起来的命令组和异步执行的命令在一个子shell环境中执行，这个子shell是从当前shell复制来的，但是shell信号捕获设置被重置为当前shell的父shell的设置。作为管道一部分执行的内置命令也是在一个子shell环境中执行的，对子shell环境的改变不会反应到当前的shell

bash孵化一个子shell执行命令替换，并从父shell中继承`-e`选项的值。如果Bash不是在POSIX模式，子shell清除`-e`选项的设置

如果一个命令以`&`符合结束，并且未启用作业控制，则命令的标准输入为`/dev/null`。否则的话命令会继承调用shell的文件描述符，和重定向操作对文件描述符的改动

环境

当程序被调用的时候，会被赋予一个字符串数组，这称为环境。这是一系列以`name=value`的名称-值对

Bash提供了几种方法来操作环境。在调用的时候，shell扫描自己的环境，并且未每个找到的环境变量名创建一个参数，并且自动导入到子进程的环境中。被执行的命令会继承环境。可以使用`export`和`declare -x`命令来将变量和函数添加或者从环境中删除。如果环境中的一个参数被改变，则新的值成为环境的一部分代替了旧的值。任何被执行的命令的环境都由shell的初始环境(初始环境变量的值可能已经被改变)，减去使用`unset`或者`export -n`命令移除的，加上使用`export`和`declare -x`命令增加的部分组成

简单命令或者函数的环境可以按照前面`shell parameter`章节介绍的方法，前置一个赋值语句来临时增加，这个增加的环境变量只存在于要执行的命令所创建的子shell的环境中

如果使用了`-k`shell选项，则命令中的任何位置的赋值语句都会成为环境的一部分，而不仅仅是前置的

当bash调用外部命令时，变量`$_`被设置为命令的绝对路径并且作为这个命令的环境

退出状态

命令的退出状态是系统调用waitpid或者其他类似函数返回的值。退出状态的值在0到255之间，然而，shell可能将大于125的值作为特殊用途，下面将详细描述。内置命令和组合命令的退出状态也是在这个范围之内。特定情形下，shell会用特定值来表示特定的错误

在shell中，通常一个命令返回0表示成功。一个非零的返回值表示失败。之所以使用这种和正常思维不太相符合的模式，因为我们可以让正确执行的命令返回一个统一的结果，而失败的结果可以用多种形式表示每种失败的情况。当命令被信号`N`终止时，bash使用`128+N`作为返回值

如果命令未找到，则shell创建的用来执行命令的子shell返回127。如果命令存在但是没有执行权限则返回126

如果命令在各种扩展或者重定向期间发生错误，则退出状态大于0

退出状态可以用作bash的条件命令或者列表结构

所有的bash内置命令如果执行成功则返回0，失败返回非0，所以也可以被用作条件命令和列表结构。所有的内置命令都返回2来表示参数或者选项不正确

信号

当bash是交互式，并且未设置任何`trap`时，shell会忽略`SIGTERM`(所以kill 0不会杀死一个交互式shell)，同时shell会捕捉`SIGINT`并进行处理(所以内置命令wait是可中断的)。当bash收到一个`SIGINT`信号时，会中断任何执行的循环。在所有情况下，bash会忽略`SIGQUIT`信号。如果作业控制生效的话，bash会忽略`SIGTTIN` `SIGTTOU` `SIGTSTP`信号

bash启动的非内置的命令从调用命令的shell继承信号处理的设置。当未启用作业控制时，除了这些继承的信号处理设置外异步命令忽略`SINGINT`和`SIGQUIT`信号。作为命令替换结果执行的命令会忽略键盘产生的作业控制信号`SIGTTING` `SIGTTOU`和`SIGTSTP`

默认shell直到收到一个`SIGHUP`信号才会退出。退出之前，交互式shell会重新发送`SIGHUP`信号给所有作业，停止的或者运行的。已经停止的作业会收到一个`SIGCONT`信号确保他们可以收到`SIGHUP`信号。要阻止shell发`SIGHUP`信号给特定的作业，我们可以使用`disown`命令将这个作业移除作业表，或者使用`-h`参数将其标记为不接收`SIGHUP`信号

如果使用shopt命令设置了`huponexit` shell选项，交互式shell退出时会发送一个`SGHUP`信号给所有的作业

如果bash正在等待一个命令执行完毕的期间接收到一个设置了`trap`的信号，则这个trap在命令完成之前不会被执行。如果bash使用内置命令`wait`等待一个异步执行的命令，一个设置了`trap`的信号的到来会会让`wait`命令在tarp执行完毕后立即返回，并且`wait`的退出状态是一个大于128的数字

shell脚本

shell脚本是一个包含shell命令的文本文件。如果这样一个文件作为调用bash的第一个非选项的参数，并且未使用`-c`或者`-s`选项，则bash会读取并执行文件的内容，然后退出。这种方式的操作创建了一个非交互式的shell。shell会首先在当前目录下寻找这个文件名，如果未找到的话，会在`$PATH`的目录下寻找

当bash执行一个脚本时，会将位置参数0设置为脚本的名字，而不是shell的名字，如果有其他的参数则成为位置参数。如果没有提供其他的参数，则表示未设置位置参数

shell脚本也可以被加上可执行的权限。当bash在搜索`$PATH`目录时找到这样一个文件，则会创建一个子shell来执行这个文件。换句话说，执行`filename arguments`相当于执行`bash filename arguments`如果文件名是一个可执行的shell脚本。则子shell会重新初始化自己，这样的效果是一个新的shell被调用来解释执行这个脚本，有一点区别是命令的位置被父shell记住并且可以在子shell中得到保持

大部分版本的Unix把这种情况也作为命令执行的一种。如果脚本文件的第一行已`#!`开始，则第一行的其余部分被作为可以解释文件中命令的解释器。所以，你可以指定`bash` `awk` `perl`或者其他的解释器

解释器参数包括一个可选的跟着解释器名的选项、然后是脚本文件名和其他的参数。如果一些操作系统自己不能处理这样的文件则bash会帮忙处理。注意有些老版本的Unix中，解释器和参数的总长度不能超过32个字符

Bash脚本通常以`#! /bin/bash`开头，这样即使在其他的shell中执行这个脚本文件，系统依然会使用bash解释器解释执行这个脚本文件