stdin, stdout, stderr以及重定向

[收藏](javascript:void(0);)

[Start-up](https://my.oschina.net/qihh/home)

* 发表于 5年前

* 阅读 17172

* 收藏 6

* 点赞 1

* [评论 0](https://my.oschina.net/qihh/blog/55308#comment-list)

[**【粉丝福利】-《Java+web 前端基础到实战系列课程》免费在线直播教学>>>**](http://web.tanzhouedu.com/index/osc.html)  https://my.oschina.net/img/hot3.png

stdin, stdout, stderr以及重定向

作者：Sam(甄峰) [sam\_code@hotmail.com](mailto:sam_code@hotmail.com)

stdin, stdout,stderr: standard I/O streams

介绍：

在通常情况下，UNIX每个程序在开始运行的时刻，都会有3个已经打开的stream. 分别用来输入，输出，打印诊断和错误信息。通常他们会被连接到用户终端（tty(4)）. 但也可以改变到其它文件或设备。这取决于双亲进程的选择和设置。

这3个symbols都是stdio(3) macro，类型为指向FILE的指针。可以被fprintf() fread（）等函数使用。

当一个程序开始启动后，stdin, stdout, and stderr are 0, 1,and 2，其它的文件描述符则排在其后。

 Linux的本质就是一切皆文件，输入输出设备也是以文件形式存在和管理的。

内核启动的时候默认打开这三个I/O设备文件：标准输入文件stdin，标准输出文件stdout，标准错误输出文件stderr，分别得到文件描述符 0, 1, 2。

stderr是不缓存的，stdout是行间缓存的。请注意：

所以：

for(i = 0; i < 10; i++)  
    {  
      fprintf(stdout, "This is stdout[%d]", i);  
      fprintf(stderr, "This is stderr[%d]", i);  
    }

会全部显示stderr之后，再显示stdout.

又因为stdout是行内缓存，所以加 \n 后会立刻显示。

在程序中使用stdin,stdout,stderr,可以先：

extern FILE \*stdout;  
extern FILE \*stderr;

重定向：

在实际应用中，可以分别使用stdout,stderr来输出。

之后再重定向：

如：

./example 1 > /dev/null

这样就将 stdout(1) 的输出丢弃。只显示stderr.

./example 2 > /dev/null

这样就将 stderr(2) 的输出丢弃。只显示stdout.

./example **>> /dev/null 2>&1**

**将2（stderr）重定向到1（stdout）. 但又将stdout(1) 丢弃。**

**所以即为丢弃stdout和stderr.**

[stdin, stdout, stderr 详解](http://www.cnblogs.com/puputu/archive/2010/06/02/1749769.html)

NAME

stdin, stdout, stderr - 标准 I/O 流

SYNOPSIS 览

#include <stdio.h> extern FILE \*stdin; extern FILE \*stdout; extern FILE \*stderr;

DESCRIPTION 述

通常，每个 Unix 程序在启动时都会打开三个流，一个用于输入，一个用于输出，一个用于打印诊断或错误消息。典型的，他们被连接到用户的终端

(参见 [tty(4)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man4/tty.4.html)) 但是也有可能指向文件或是其他设备，取决于父进程选择设置了什么 (参见 [sh(1)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man1/sh.1.html) 的重定向 (``Redirection'') 章节。)

输入流被称为 ``standard input''; 输出流被称为 ``standard output''; 错误流被称为 ``standard error''。这些名词通常简写为符号，用于引用这些文件，它们是 stdin, stdout, 和 stderr.

这些符号中，每一个都是 [stdio(3)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man3/stdio.3.html) 中的一个宏，类型是指向 FILE 的指针，可以用于类似 [fprintf(3)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man3/fprintf.3.html) 或 [fread(3)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man3/fread.3.html) 等函数中。

由于 FILE 是一个对 Unix 文件描述符加以缓冲的包装，下层的文件也可以使用- 始的 Unix 文件接口来存取。也就是，类似 [read(2)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man2/read.2.html) 和 [lseek(2)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man2/lseek.2.html) 的函数。与流

stdin, stdout, 和 stderr 关联的整数形式的文件描述符分别是 0，1 还有

2。预处理器符号 STDIN\_FILENO，STDOUT\_FILENO 和 STDERR\_FILENO 分别以它们为值，定义在 <unistd.h> 中。

注意混合使用 FILE 和- 始的文件描述符可能带来不可预料的结果，一般应当避免。(对于喜欢追根问底的人：POSIX.1 规范的 8.2.3 节详细地描述了这样的混合使用怎样才能不出错。) 一个简单的规则是，文件描述符由内核控制，而 stdio 仅仅是一个库。它的意思是，例如当调用 exec 之后，子进程可以继承所有打开的文件描述符，但是任何杂械牧鞫疾豢稍俅嫒×恕

由于符号 stdin, stdout, 和 stderr 被指定为宏，为它们赋值将导致不可移植。利用库函数 [freopen(3)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man3/freopen.3.html) ，标准流可以用来指向不同的文件。引进这个函数专门用来为 stdin, stdout, 和

stderr 重新赋值。标准流在调用 [exit(3)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man3/exit.3.html) 和程序正常中止时被关闭。

[sh(1)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man1/sh.1.html), [csh(1)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man1/csh.1.html), [open(2)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man2/open.2.html), [fopen(3)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man3/fopen.3.html), [stdio(3)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man3/stdio.3.html)

CONSIDERATIONS

错误流 stderr 是非缓冲的。输出流 stdout 是行缓冲的，如果它指向一个终端。不完全的行只有在调用 [fflush(3)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man3/fflush.3.html) 或 [exit(3)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man3/exit.3.html) ，或者打印了新行符之后才会显示。这样可能带来无法预料的结果，尤其是调试输出时。标准流

(或任何其他流) 的缓冲模式可以用函数 [setbuf(3)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man3/setbuf.3.html) 或 [setvbuf(3)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man3/setvbuf.3.html) 来切换。注意当 stdin 与一个终端关联时，也许终端驱动中存在输入缓冲，与

stdio 缓冲完全无关。(确实如此，一般的终端输入在内核中是行缓冲的。) 内核对输入的控制可以通过对 [tcsetattr(3)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man3/tcsetattr.3.html) 的调用来修改，参见 [stty(1)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man1/stty.1.html), 和

[termios(3)](http://manpages.ubuntu.com/manpages/gutsy/zh_CN/man3/termios.3.html) 。

宏 stdin, stdout, 和 stderr 遵 ANSI X3.159-1989 (``ANSI C'') 标准，这个标准同时规定了这三个流应当在程序启动时打开。

Advanced Bash-Script-3.9.1\_cn

16. I/O重定向

**目录**

16.1. [使用](http://www.tsnc.edu.cn/default/tsnc_wgrj/doc/abs-3.9.1_cn/html/x13380.html) [**exec**](http://www.tsnc.edu.cn/default/tsnc_wgrj/doc/abs-3.9.1_cn/html/x13380.html)

16.2. [代码块重定向](http://www.tsnc.edu.cn/default/tsnc_wgrj/doc/abs-3.9.1_cn/html/redircb.html)

16.3. [重定向的应用](http://www.tsnc.edu.cn/default/tsnc_wgrj/doc/abs-3.9.1_cn/html/redirapps.html)

默认情况下始终有3个"文件"处于打开状态,stdin(键盘),stdout(屏幕),和stderr(错误消息输出到屏幕上).这3个文件和其他打开的文件都可以被重定向.对于重定向简单的解释就是捕捉一个文件,命令, 程序,脚本, 或者是脚本中的代码块(请参考[例子3-1](http://www.tsnc.edu.cn/default/tsnc_wgrj/doc/abs-3.9.1_cn/html/special-chars.html#EX8)和[例子3-2](http://www.tsnc.edu.cn/default/tsnc_wgrj/doc/abs-3.9.1_cn/html/special-chars.html#RPMCHECK))的输出,然后将这些输出作为输入发送到另一个文件,命令, 程序,或脚本中.

每个打开的文件都会被分配一个文件描述符.[[1]](http://www.tsnc.edu.cn/default/tsnc_wgrj/doc/abs-3.9.1_cn/html/io-redirection.html#FTN.AEN13310) stdin, stdout,和stderr的文件描述符分别是0, 1, 和 2. 除了这3个文件,对于其他那些需要打开的文件,保留了文件描述符3到9.在某些情况下,将这些额外的文件描述符分配给stdin,stdout, 或stderr作为临时的副本链接是非常有用的.[[2]](http://www.tsnc.edu.cn/default/tsnc_wgrj/doc/abs-3.9.1_cn/html/io-redirection.html#FTN.AEN13320) 在经过复杂的重定向和刷新之后需要把它们恢复成正常状态(请参考[例子16-1](http://www.tsnc.edu.cn/default/tsnc_wgrj/doc/abs-3.9.1_cn/html/x13380.html#REDIR1)).

|  |
| --- |
| 1  COMMAND\_OUTPUT > 2  # 将stdout重定向到一个文件. 3  # 如果这个文件不存在, 那就创建, 否则就覆盖. 4  5  ls -lR > dir-tree.list 6  # 创建一个包含目录树列表的文件. 7  8  : > filename 9  # >操作, 将会把文件"filename"变为一个空文件(就是size为0). 10  # 如果文件不存在, 那么就创建一个0长度的文件(与'touch'的效果相同). 11  # :是一个占位符, 不产生任何输出. 12  13  > filename 14  # >操作, 将会把文件"filename"变为一个空文件(就是size为0). 15  # 如果文件不存在, 那么就创建一个0长度的文件(与'touch'的效果相同). 16  # (与上边的": >"效果相同, 但是某些shell可能不支持这种形式.) 17  18  COMMAND\_OUTPUT >> 19  # 将stdout重定向到一个文件. 20  # 如果文件不存在, 那么就创建它, 如果存在, 那么就追加到文件后边. 21  22  23  # 单行重定向命令(只会影响它们所在的行): 24  # -------------------------------------------------------------------- 25  26  1>filename 27  # 重定向stdout到文件"filename". 28  1>>filename 29  # 重定向并追加stdout到文件"filename". 30  2>filename 31  # 重定向stderr到文件"filename". 32  2>>filename 33  # 重定向并追加stderr到文件"filename". 34  &>filename 35  # 将stdout和stderr都重定向到文件"filename". 36  37  M>N 38  # "M"是一个文件描述符, 如果没有明确指定的话默认为1. 39  # "N"是一个文件名. 40  # 文件描述符"M"被重定向到文件"N". 41  M>&N 42  # "M"是一个文件描述符, 如果没有明确指定的话默认为1. 43  # "N"是另一个文件描述符. 44  45  #============================================================================== 46  47  # 重定向stdout, 一次一行. 48  LOGFILE=script.log 49  50  echo "This statement is sent to the log file, \"$LOGFILE\"." 1>$LOGFILE 51  echo "This statement is appended to \"$LOGFILE\"." 1>>$LOGFILE 52  echo "This statement is also appended to \"$LOGFILE\"." 1>>$LOGFILE 53  echo "This statement is echoed to stdout, and will not appear in \"$LOGFILE\"." 54  # 每行过后, 这些重定向命令会自动"reset". 55  56  57  58  # 重定向stderr, 一次一行. 59  ERRORFILE=script.errors 60  61  bad\_command1 2>$ERRORFILE # Error message sent to $ERRORFILE. 62  bad\_command2 2>>$ERRORFILE # Error message appended to $ERRORFILE. 63  bad\_command3 # Error message echoed to stderr, 64  #+ and does not appear in $ERRORFILE. 65  # 每行过后, 这些重定向命令也会自动"reset". 66  #============================================================================== 67  68  69  70  2>&1 71  # 重定向stderr到stdout. 72  # 将错误消息的输出, 发送到与标准输出所指向的地方. 73  74  i>&j 75  # 重定向文件描述符**i**到**j**. 76  # 指向**i**文件的所有输出都发送到**j**. 77  78  >&j 79  # 默认的, 重定向文件描述符**1**(stdout)到**j**. 80  # 所有传递到stdout的输出都送到**j**中去. 81  82  0< FILENAME 83  < FILENAME 84  # 从文件中接受输入. 85  # 与">"是成对命令, 并且通常都是结合使用. 86  # 87  # grep search-word <filename 88  89  90  [j]<>filename 91  # 为了读写"filename", 把文件"filename"打开, 并且将文件描述符"j"分配给它. 92  # 如果文件"filename"不存在, 那么就创建它. 93  # 如果文件描述符"j"没指定, 那默认是fd 0, stdin. 94  # 95  # 这种应用通常是为了写到一个文件中指定的地方. 96  echo 1234567890 > File # 写字符串到"File". 97  exec 3<> File # 打开"File"并且将fd 3分配给它. 98  read -n 4 <&3 # 只读取4个字符. 99  echo -n . >&3 # 写一个小数点. 100  exec 3>&- # 关闭fd 3. 101  cat File # ==> 1234.67890 102  # 随机访问. 103  104  105  106  | 107  # 管道. 108  # 通用目的处理和命令链工具. 109  # 与">", 很相似, 但是实际上更通用. 110  # 对于想将命令, 脚本, 文件和程序串连起来的时候很有用. 111  cat \*.txt | sort | uniq > result-file 112  # 对所有.txt文件的输出进行排序, 并且删除重复行. 113  # 最后将结果保存到"result-file"中. |

可以将输入输出重定向和(或)管道的多个实例结合到一起写在同一行上.

|  |
| --- |
| 1 command < input-file > output-file 2  3 command1 | command2 | command3 > output-file |

请参考[例子12-28](http://www.tsnc.edu.cn/default/tsnc_wgrj/doc/abs-3.9.1_cn/html/filearchiv.html#DERPM)和[例子A-15](http://www.tsnc.edu.cn/default/tsnc_wgrj/doc/abs-3.9.1_cn/html/contributed-scripts.html#FIFO).

可以将多个输出流重定向到一个文件上.

|  |
| --- |
| 1 ls -yz >> command.log 2>&1 2 # 将错误选项"yz"的结果放到文件"command.log"中. 3 # 因为stderr被重定向到这个文件中, 4 #+ 所有的错误消息也就都指向那里了. 5  6 # 注意, 下边这个例子就不会给出相同的结果. 7 ls -yz 2>&1 >> command.log 8 # 输出一个错误消息, 但是并不写到文件中. 9  10 # 如果将stdout和stderr都重定向, 11 #+ 命令的顺序会有些不同. |

**关闭文件描述符**

n<&-

关闭输入文件描述符 *n*.

0<&-, <&-

关闭 stdin.

n>&-

关闭输出文件描述符 *n*.

1>&-, >&-

关闭 stdout.

子进程继承了打开的文件描述符.这就是为什么管道可以工作.如果想阻止fd被继承,那么可以关掉它.

|  |
| --- |
| 1 # 只将stderr重定到一个管道. 2  3 exec 3>&1 # 保存当前stdout的"值". 4 ls -l 2>&1 >&3 3>&- | grep bad 3>&- # 对'grep'关闭fd 3(但不关闭'ls'). 5 # ^^^^ ^^^^ 6 exec 3>&- # 对于剩余的脚本来说, 关闭它. 7  8 # 感谢, S.C. |

如果想了解关于I/O重定向更多的细节,请参考[Appendix E](http://www.tsnc.edu.cn/default/tsnc_wgrj/doc/abs-3.9.1_cn/html/ioredirintro.html).

注意事项

|  |  |
| --- | --- |
| [[1]](http://www.tsnc.edu.cn/default/tsnc_wgrj/doc/abs-3.9.1_cn/html/io-redirection.html#AEN13310) | 一个**文件描述符**说白了就是文件系统为了跟踪这个打开的文件而分配给它的一个数字. 也可以的将其理解为文件指针的一个简单版本. 与C语言中**文件句柄**的概念很相似. |
| [[2]](http://www.tsnc.edu.cn/default/tsnc_wgrj/doc/abs-3.9.1_cn/html/io-redirection.html#AEN13320) | 使用*文件描述符5*可能会引起问题. 当Bash使用[exec](http://www.tsnc.edu.cn/default/tsnc_wgrj/doc/abs-3.9.1_cn/html/internal.html#EXECREF)创建一个子进程的时候, 子进程会继承fd5(参考Chet Ramey的归档e-mail, [SUBJECT: RE: File descriptor 5 is held open](http://www.geocrawler.com/archives/3/342/1996/1/0/1939805/)). 最好还是不要去招惹这个特定的fd. |

Linux Tips  
IO Redirection

<http://www.linuxsa.org.au/tips/io-redirection.html>

UNIX had the concept of IO redirection long before DOS copied and bastardised the concept. The UNIX IO redirection concept is fundamental to many of the things that you can do with UNIX, and it is quite a well-developed idea, so we will explore this concept here.

Why do I mention UNIX at all? Well, Linux **is** a UNIX operating system!

Under UNIX, all programs that run are given three open files when they are started by a shell:

|  |  |
| --- | --- |
| 0. | **Standard in, or STDIN.**  This is where input comes from, and it normally points at your terminal device.  To find out what device is your terminal, use the tty(1) command. Note, the (1) after command names in UNIX refers to the section of the man pages that the documentation for the command exists in.  You can arrange to run any command and pass it input from a file in the following way:  $ some-command < /path/to/some/file  Note, the '$' is your prompt. Note also, you can always specify a complete path name for a file.  For example:  $ grep -i Fred < /etc/passwd  Would search for the string 'fred' in /etc/passwd, regardless of the case of the characters.  But wait a minute, you object, I always use:  $ grep -i Fred /etc/passwd  This is true, but you can also pass the file in on STDIN, and you will get different results if you do. Can you see what the difference is? |
| 1. | **Standard out, or STDOUT.**  This is where the normal output from a program goes. It normally points at your terminal as well, but you can redirect it.  You can redirect output in the following way:  $ some-program > /path/to/some/file  For example:  $ grep -i Fred /etc/passwd > /tmp/results |
| 2. | **Standard error, or STDERR.**  This is where error output from your program goes. This normally points at your terminal as well, but you can redirect it.  Why have different output places for standard out and standard error?  Well, as you will see when you come to writing shell scripts, you often do not want error messages cluttering up the normal output from a program. |

You will forgive me for starting the above list at 0, I am sure, when you learn that each of these IO 'channels' are represented by small numbers, called file descripters (FDs), that have exactly those numbers. That is, STDIN is FD 0, while STDOUT is FD 1, and STDERR is FD 2.

When the shell runs a program for you, it opens STDIN as FD 0, STDOUT as FD 1, and STDERR as FD 2, and then runs the program (technically, it almost always does a fork(2) and then anexec(3) or one of the exec?? calls). If you have redirected one of STDIN, STDOUT or STDERR, your shell opens that file as the appropriate FD before running the program.

Now, what does this all have to do with you, I hear you ask?

Well, there are lots of neat things you can do, but some things to watch out for as well.

A lot of inexperienced UNIX users assume that they can redirect a file into a program and use the same name for redirecting the output:

$ some-program < mega-important-data-file > mega-important-data-file

They become very upset after doing the above, especially if that mega-important data file has never been backed up anywhere. Why is this?

The shell opens the mega-important-data-file for reading and associates it with FD 0 (or STDIN), and then opens it for writing, but truncates it to zero length, and associates it with FD 1 (or STDOUT) as well.

So, if you want to do something like the above, use a different file name for the output file. Oh, you should also back up files as well :-).

Now, there are lots of redirection symbols that you can use, and here are some of them:

|  |  |
| --- | --- |
| < *file* | means open a file for reading and associate with STDIN. |
| << *token* | Means use the current input stream as STDIN for the program until *token* is seen. We will ignore this one until we get to scripting. |
| > *file* | means open a file for writing and truncate it and associate it with STDOUT. |
| >> *file* | means open a file for writing and seek to the end and associate it with STDOUT. This is how you append to a file using a redirect. |
| *n*>&*m* | means redirect FD *n* to the same places as FD *m*. Eg, 2>&1 means send STDERR to the same place that STDOUT is going to. |

OK, here are some tricks that you might want to use in various places.

If you are gathering evidence for a bug report, you might want to redirect the output from a series of programs to a text file (never mind that you can use the script command to do the same :-). So you might do the following:

$ some-buggy-program > important-evidence.txt $ echo '---------MARKER-------' >> important-evidence.txt $ some-buggy-program *some-params* >> important-evidence.txt

The second and subsequent lines append the output from the commands issues to the evidence file rather than overwriting them. Try the following:

$ echo This is a line of text > /tmp/file.txt $ echo This is another line > /tmp/file.txt

What do you get?

Now try:

$ echo This is a line of text > /tmp/file.txt $ echo This is another line >> /tmp/file.txt

What do you get this time?

OK, for the last few tricks here. Sometimes you want to append STDOUT and STDERR to a file. How do you do it?

$ some-command >> /tmp/log.log 2>&1

The 2>&1 says make STDERR point to the same places as STDOUT. Since STDOUT is open already, and the shell has done a seek to the end, STDERR will also be appended to STDOUT.

If you want to append a line to a file, you can echo the line you want with a redirect, rather than firing up an editor:

$ echo *Some text* >> /path/to/some/file

It turns out that you can cause the shell to redirect to other file descriptors as well, and if you look in the configure scripts that come with many UNIX software packages, you will see examples of this.

Why is redirecting so important? Well, it is used in many shell scripts, it is a simple and conventient mechanism to sending output to any file without the programmer having to add code for handling command line instructions, and it is the UNIX way of doing things :-).

It is also the same as piping, where you redirect output to, or input from, a pipe device. The pipe device has a process living on the other side, but we will look at this later.

Regards

------- Richard Sharpe, [sharpe@ns.aus.com](mailto:sharpe@ns.aus.com), Master Linux Administrator :-), Samba (Team member, [www.samba.org](http://www.samba.org/)), Ethereal (