Иллюстрированное руководство по перенаправлению

Это руководство не является полным руководством по перенаправлению, оно не будет охватывать документы here, строки here, каналы имен и т. Д... Я просто надеюсь, что это поможет вам понять, что нравится `3>&2 2>&1` или `1>&3-` делать.

stdin, stdout, stderr

При запуске Bash обычно открываются 3 файловых дескриптора, `0` , `1` `2` также известные как стандартный ввод (`stdin`), стандартный вывод (`stdout`) и стандартная ошибка (`stderr`).

Например, при запуске Bash в эмуляторе терминала Linux вы увидите:

```
# lsof +f g -ap $ BASHPID -d 0,1,2
КОМАНДА PID ТИП FD ПОЛЬЗОВАТЕЛЯ ФАЙЛ-ФЛАГ РАЗМЕР УСТРОЙСТВА / ВЫКЛ. ИМЯ УЗЛА
bash 12135 root 0u CHR RW, LG 136,13 0t0 16 / dev/ pts / 5
bash 12135 root 1u CHR RW, LG 136,13 0t0 16 /dev/pts / 5
bash 12135 root 2u CHR RW, LG 136,13 0t0 16 / dev / pts / 5dev / pts /5
```

Это `/dev/pts/5` псевдотерминал, используемый для эмуляции реального терминала. Bash считывает (`stdin`) с этого терминала и печатает через `stdout` и `stderr` на этот терминал.

```
 --- +-----------------------+
стандартный ввод ( 0 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
стандартный выход ( 1 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
стандартная ошибка ( 2 ) ---->| / dev/очко/5 |
 --- +-----------------------+
```

Когда выполняется команда, составная команда, подоболочка и т. Д., Она наследует эти файловые дескрипторы. Например `echo foo` , отправит текст `foo` в файловый дескриптор `1` , унаследованный от оболочки, к которой подключен `/dev/pts/5` .

Простые перенаправления

Перенаправление вывода "n> file"

`>` вероятно, это самое простое перенаправление.

```
echo foo > file
```

`> file` после команды изменяются файловые дескрипторы, принадлежащие команде `echo` . Он изменяет дескриптор файла `1` (`> file` такой же, как `1>file`), чтобы он указывал на файл `file` . Они будут выглядеть так:

```
 --- +-----------------------+
стандартный ввод ( 0 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
стандартный вывод ( 1 ) ---->| file |
 --- +-----------------------+

 --- +-----------------------+
стандартная ошибка ( 2 ) ---->| / dev/очко/5 |
 --- +-----------------------+
```

Теперь символы, написанные нашей командой, `echo` , которые отправляются на стандартный вывод, то есть в дескриптор файла `1` , попадают в файл с именем `file` .

Таким же образом command `2> file` изменит стандартную ошибку и укажет на `file` нее. Стандартная ошибка используется приложениями для печати ошибок.

Что будет `command 3> file` делать? Он откроет новый файловый дескриптор, указывающий на `file` . Затем команда начнется с:

```
 --- +-----------------------+
стандартный ввод ( 0 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
стандартный выход ( 1 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
стандартная ошибка ( 2 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
новый дескриптор (3) ---->| file |
 --- +-----------------------+
```

Что команда будет делать с этим дескриптором? Это зависит. Часто ничего. Позже мы увидим, почему нам могут понадобиться другие файловые дескрипторы.

Введите перенаправление "n< file"

Когда вы запускаете команду `command < file` using , она изменяет дескриптор файла `0` так, чтобы он выглядел как:

```
 --- +-----------------------+
стандартный ввод ( 0 ) <----| file |
 --- +-----------------------+

 --- +-----------------------+
стандартный выход ( 1 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
стандартная ошибка ( 2 ) ---->| / dev/очко/5 |
 --- +-----------------------+
```

Если команда считывается из `stdin` , теперь она будет считываться из `file` , а не с консоли.

Как и `>` , `<` может использоваться для открытия нового файлового дескриптора для чтения, `command 3<file` . Позже мы увидим, как это может быть полезно.

Трубы |

Что это `|` делает? Среди прочего, он соединяет стандартный вывод команды слева со стандартным вводом команды справа. То есть он создает специальный файл, канал, который открывается как адрес назначения записи для левой команды и как источник чтения для правой команды.

```
 echo foo | cat

 --- +--------------+ --- +--------------+
( 0 ) ---->| / dev/оч/5 | ------> ( 0 ) ---->| труба (читать) |
 --- +--------------+ / --- +--------------+
 /
 --- +--------------+ / --- +--------------+
( 1 ) ---->| pipe (запись) | / ( 1 ) ---->| / dev/оч |
 --- +--------------+ --- +--------------+

 --- +--------------+ --- +--------------+
( 2 ) ---->| / dev/оч/5 | ( 2 ) ---->| / dev/оч/ |
 --- +--------------+ --- +--------------+
```

Это возможно, потому что перенаправления настраиваются командной оболочкой **перед** выполнением команд, и команды наследуют дескрипторы файлов.

Подробнее о файловых дескрипторах

Дублирование файлового дескриптора 2> &1

Мы видели, как открывать (или перенаправлять) файловые дескрипторы. Давайте посмотрим, как их дублировать, начиная с классического `2>&1` . Что это значит? Что-то, написанное в файловом дескрипторе `2` , будет идти туда, куда `1` идет файловый дескриптор. В оболочке `command 2>&1` не очень интересный пример, поэтому мы будем использовать `ls /tmp/ doesnotexist 2>&1 | less`

```
 ls /tmp/ не существует 2> и 1 | меньше

 --- +--------------+ --- +--------------+
( 0 ) ---->| / dev/оч/5 | ------> ( 0 ) ---->| из трубы |
 --- +--------------+ / ---> --- +--------------+
 / /
 --- +--------------+ / / --- +--------------+
(1) ---->| в канал | / / ( 1 ) ---->| / dev/оч |
 --- +--------------+ / --- +--------------+
 /
 --- +--------------+ / --- +--------------+
(2) ---->| в канал | / ( 2 ) ---->| / dev/оч/ |
 --- +--------------+ --- +--------------+
```

Почему это называется *дублированием*? Потому что после `2>&1` у нас есть 2 файловых дескриптора, указывающих на один и тот же файл. Будьте осторожны, чтобы не называть это "сглаживанием дескриптора файла"; если мы перенаправим `stdout` после `2>&1` на файл `B` , дескриптор файла `2` все равно будет открыт в файле `A` , где он был. Это часто неправильно понимают люди, желающие перенаправить как стандартный ввод, так и стандартный вывод в файл. Продолжайте читать, чтобы узнать больше об этом.

Итак, если у вас есть два файловых дескриптора `s` и `t` :

```
 --- +----------------------+
дескриптор ( ы ) ---->| /некоторый/файл |
 --- +----------------------+
 --- +----------------------+
дескриптор ( t ) ---->| /другой/файл |
 --- +----------------------+
```

Используя a `t>&s` (где `t` и `s` являются числами), это означает:

> Скопируйте все `s` , что содержит файловый дескриптор, в файловый дескриптор `t`

Итак, у вас есть копия этого дескриптора:

```
 --- +-----------------------+
дескриптор ( ы ) ---->| /некоторый/файл |
 --- +-----------------------+
 --- +-----------------------+
дескриптор ( t ) ---->| /некоторый/файл |
 --- +-----------------------+
```

Внутренне каждый из них представлен файловым дескриптором, открываемым `fopen` вызовами операционной системы, и, вероятно, является просто указателем на файл, который был открыт для чтения (`stdin` или файловый дескриптор `0`) или записи (`stdout` / `stderr`).

Обратите внимание, что позиции чтения или записи файла также дублируются. Если вы уже прочитали строку из `s` , то после `t>&s` того, как вы прочитаете строку из `t` , вы получите вторую строку файла.

Аналогично для выходных файловых дескрипторов, запись строки в файловый дескриптор `s` добавит строку к файлу, как и запись строки в файловый дескриптор `t` .

Синтаксис несколько сбивает с толку, поскольку вы могли бы подумать, что стрелка указывает в направлении копии, но она перевернута. Так что это `target>&source` эффективно.
Итак, в качестве простого примера (хотя и слегка надуманного) приведем следующее:

```
exec 3> & 1 # Скопируйте 1 в 3
exec 1> файл журнала # Сделайте 1 открытым для записи в файл журнала
lotsa_stdout # Выводит в fd 1, который записывает в файл журнала
exec 1> &3 # Скопируйте 3 обратно в 1
echo Done # Вывод в исходный стандартный вывод
```

Порядок перенаправления, т.е. "> file 2> & 1" против "2> & 1> file"

Хотя не имеет значения, где в командной строке отображаются перенаправления, их порядок имеет значение. Они настраиваются слева направо.

- `2>&1 >file`

Распространенная ошибка `command 2>&1 > file` заключается в том, чтобы перенаправлять оба `stderr` и `stdout` на `file` . Давайте посмотрим, что происходит. Сначала мы вводим команду в нашем терминале, дескрипторы выглядят следующим образом:

```
 --- +-----------------------+
стандартный ввод ( 0 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
стандартный выход ( 1 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
стандартная ошибка ( 2 ) ---->| / dev/очко/5 |
 --- +-----------------------+
```

Затем наша оболочка, Bash видит `2>&1` , что она дублирует 1, и дескриптор файла выглядит следующим образом:

```
 --- +-----------------------+
стандартный ввод ( 0 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
стандартный выход ( 1 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
стандартная ошибка ( 2 ) ---->| / dev/очко/5 |
 --- +-----------------------+
```

Верно, ничего не изменилось, 2 уже указывал на то же место, что и 1. Теперь Bash видит `> file` и, таким образом, изменяет `stdout` :

```
 --- +-----------------------+
стандартный ввод ( 0 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
стандартный вывод ( 1 ) ---->| file |
 --- +-----------------------+

 --- +-----------------------+
стандартная ошибка ( 2 ) ---->| / dev/очко/5 |
 --- +-----------------------+
```

И это не то, чего мы хотим.

- `>file 2>&1`

Теперь давайте посмотрим на правильное `command >file 2>&1` . Мы начинаем, как в предыдущем примере, и Bash видит `> file` :

```
 --- +-----------------------+
стандартный ввод ( 0 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
стандартный вывод ( 1 ) ---->| file |
 --- +-----------------------+

 --- +-----------------------+
стандартная ошибка ( 2 ) ---->| / dev/очко/5 |
 --- +-----------------------+
```

Затем он видит наше дублирование `2>&1` :

```
 --- +-----------------------+
стандартный ввод ( 0 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
стандартный вывод ( 1 ) ---->| file |
 --- +-----------------------+

 --- +-----------------------+
стандартная ошибка ( 2 ) ---->| file |
 --- +-----------------------+
```

И вуаля, оба `1` и `2` перенаправляются в файл.

Почему sed 's/foo/bar/' file >file не работает

Это распространенная ошибка, мы хотим изменить файл, используя что-то, что считывает из файла и записывает результат `stdout` . Для этого мы перенаправляем стандартный вывод в файл, который мы хотим изменить. Проблема здесь в том, что, как мы видели, перенаправления настраиваются до фактического выполнения команды.

Таким образом, **ПЕРЕД** запуском sed стандартный вывод уже перенаправлен, с дополнительным побочным эффектом, который, поскольку мы использовали>, "file" усекается. Когда `sed` начинается чтение файла, он ничего не содержит.

exec

В Bash `exec` встроенная программа заменяет оболочку указанной программой. Итак, какое это имеет отношение к перенаправлению? `exec` также позволяет нам манипулировать файловыми дескрипторами. Если вы не укажете программу, перенаправление после `exec` изменяет файловые дескрипторы текущей оболочки.

Например, все команды после `exec 2>file` будут иметь файловые дескрипторы, такие как:

```
 --- +-----------------------+
стандартный ввод ( 0 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
стандартный выход ( 1 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
стандартная ошибка ( 2 ) ---->| file |
 --- +-----------------------+
```

Все ошибки, отправленные `stderr` командами после `exec 2>file` , будут отправлены в файл, как если бы у вас была команда в скрипте и она была запущена `myscript 2>file` .

`exec` может использоваться, если, например, вы хотите регистрировать ошибки, создаваемые командами в вашем скрипте, просто добавьте `exec 2>myscript.errors` их в начале вашего скрипта.

Давайте рассмотрим другой вариант использования. Мы хотим прочитать файл построчно, это легко, мы просто делаем:

```
 при чтении -строка r; выполнить эхо "$line";готово < файл
```

Теперь мы хотим после печати каждой строки делать паузу, ожидая, пока пользователь нажмет клавишу:

```
 при чтении -строка r; повторите "$line"; чтение -p "Нажмите любую клавишу" -n 1;готово < фа
йл
```

И, к удивлению, это не работает. Почему? поскольку дескриптор оболочки цикла while выглядит следующим образом:

```
 --- +-----------------------+
стандартный ввод ( 0 ) ---->| file |
 --- +-----------------------+

 --- +-----------------------+
стандартный выход ( 1 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
стандартная ошибка ( 2 ) ---->| / dev/очко/5 |
 --- +-----------------------+
```

и наше чтение наследует эти дескрипторы, а наша команда (`read -p "Press any key" -n 1`) наследует их и, таким образом, считывает из файла, а не из нашего терминала.

Беглый взгляд на `help read` говорит нам, что мы можем указать дескриптор файла, из которого `read` следует читать. Прохладный. Теперь давайте используем `exec` для получения другого дескриптора:

```
 exec 3
```

Теперь дескрипторы файлов выглядят так:

```
 --- +-----------------------+
стандартный ввод ( 0 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
стандартный выход ( 1 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
стандартная ошибка ( 2 ) ---->| / dev/очко/5 |
 --- +-----------------------+

 --- +-----------------------+
новый дескриптор (3) ---->| file |
 --- +-----------------------+
```

и это работает.

Закрытие файловых дескрипторов

Закрыть файл с помощью файлового дескриптора легко, просто сделайте его дубликатом - . Например, давайте закроем `stdin <&-` и `stderr 2>&-` :

```
 bash -c '{ lsof -a -p $$ -d0,1,2 ;} <&- 2>&-'
 КОМАНДА PID ПОЛЬЗОВАТЕЛЬ FD ТИП РАЗМЕР УСТРОЙСТВА ИМЯ УЗЛА
bash 10668 pgas 1u CHR 136,2 4 /dev/pts/2
```

мы видим, что внутри `{}` that only `1` все еще здесь.

Хотя ОС (), вероятно, наведет порядок, возможно, хорошей идеей будет закрыть файловые дескрипторы, которые вы открываете. Например, если вы откроете файловый дескриптор с `exec 3>file` помощью , все команды впоследствии унаследуют его. Вероятно, лучше сделать что-то вроде:

```
exec 3> файл
.....
#команды, использующие 3
.....
exec 3>&-

# нам больше не нужно 3
```

Я видел, как некоторые люди использовали это как способ отбросить, скажем, stderr, используя что-то вроде: command 2> &- . Хотя это может сработать, я не уверен, что вы можете ожидать, что все приложения будут корректно работать с закрытым stderr.

Когда сомневаешься, я использую 2>/dev/null.

Пример

Этот пример взят из этого сообщения (ffe4c2e382034ed9) (http://groups.google.com/group/comp.unix.shell/browse_thread/thread/64206d154894a4ef/ffe4c2e382034ed9#ffe4c2e382034ed9) в группе comp.unix.shell:

```
{
{
cmd1 3>&- |
cmd2 2>&3 3>&-
 } 2>&1 >&4 4>&- |
cmd3 3>&- 4>&-

} 3>&2 4>&1
```

Перенаправления обрабатываются слева направо, но поскольку файловые дескрипторы наследуются, нам также придется работать от внешнего контекста к внутреннему. Предположим, что мы запускаем эту команду в терминале. Давайте начнем с внешнего `{ } 3>&2 4>&1` .

```
 --- +-------------+ --- +-------------+
( 0 ) ---->| / dev/оч/5 | ( 3 ) ---->| / dev/очко/5 |
 --- +-------------+ --- +-------------+

 --- +-------------+ --- +-------------+
( 1 ) ---->| / dev/оч/5 | ( 4 ) ---->| / dev/очко/5 |
 --- +-------------+ --- +-------------+

 --- +-------------+
( 2 ) ---->| / dev/очко/5 |
 --- +-------------+
```

Мы сделали только 2 копии `stderr` и `stdout` . `3>&1 4>&1` здесь был бы тот же результат, потому что мы выполнили команду в терминале и, таким образом `1` , `2` перешли к терминалу. В качестве упражнения вы можете начать с `1` указания на `file.stdout` и 2 указания на `file.stderr` , вы увидите, почему эти перенаправления очень хороши.

Давайте продолжим с правой частью второго канала: `| cmd3 3>&- 4>&-`

```
 --- +-------------+
(0) ---->| 2-й канал |
 --- +-------------+

 --- +-------------+
( 1 ) ---->| / dev/очко/5 |
 --- +-------------+

 --- +-------------+
( 2 ) ---->| / dev/очко/5 |
 --- +-------------+
```

Он наследует предыдущие файловые дескрипторы, закрывает 3 и 4 и настраивает канал для чтения. Теперь для левой части второго канала `{…} 2>&1 >&4 4>&- |`

```
 --- +-------------+ --- +-------------+
( 0 ) ---->| / dev/оч/5 | ( 3 ) ---->| / dev/очко/5 |
 --- +-------------+ --- +-------------+

 --- +-------------+
( 1 ) ---->| / dev/очко/5 |
 --- +-------------+

 --- +-------------+
( 2) ---->| 2-й канал |
 --- +-------------+
```

Сначала дескриптор файла `1` подключается к каналу (`|`), затем `2` создается копия `1` и, таким образом, создается fd для канала (`2>&1`), затем `1` создается копия `4` (`>&4`), затем `4` закрывается. Это файловые дескрипторы внутреннего `{}` . Давайте зайдем внутрь и посмотрим на правую часть первого канала: `| cmd2 2>&3 3>&-`

```
 --- +-------------+
( 0) ---->| 1-й канал |
 --- +-------------+

 --- +-------------+
( 1 ) ---->| / dev/очко/5 |
 --- +-------------+

 --- +-------------+
( 2 ) ---->| / dev/очко/5 |
 --- +-------------+
```

Он наследует предыдущие файловые дескрипторы, подключает 0 к 1-му каналу, файловый дескриптор 2 становится копией 3, а 3 закрывается. Наконец, для левой части канала:

```
 --- +-------------+
( 0 ) ---->| / dev/очко/5 |
 --- +-------------+

 --- +-------------+
( 1) ---->| 1-й канал |
 --- +-------------+

 --- +-------------+
( 2) ---->| 2-й канал |
 --- +-------------+
```

Он также наследует файловый дескриптор левой части 2-го канала, файловый дескриптор `1` подключен к первому каналу, `3` закрыт.

Цель всего этого становится ясной, если мы возьмем только команды:

```
 cmd2

 --- +-------------+
 -->( 0 ) ---->| 1 - я труба |
 / --- +-------------+
 /
 / --- +-------------+
cmd 1 / ( 1 ) ---->| / dev/очко/5 |
 / --- +-------------+
 /
 --- +-------------+ / --- +-------------+
( 0 ) ---->| / dev/оч/5 | / ( 2 ) ---->| / dev/очко/5 |
 --- +-------------+ / --- +-------------+
 /
 --- +-------------+ / cmd3
( 1) ---->| 1-й канал | /
 --- +-------------+ --- +-------------+
 ------------>( 0 ) ---->| 2 - я труба |
 --- +-------------+ / --- +-------------+
( 2) ---->| 2-й канал |/
 --- +-------------+ --- +-------------+
 ( 1 ) ---->| / dev/очко/5 |
 --- +-------------+

 --- +-------------+
 ( 2 ) ---->| / dev/очко/5 |
 --- +-------------+
```

Как говорилось ранее, в качестве упражнения вы можете начать с `1` открытия файла и `2` открыть другой файл, чтобы увидеть, как `stdin` из `cmd2` и `cmd3` переходит к оригиналу `stdin` и как `stderr` переходит к оригиналу `stderr` .

Синтаксис

Раньше у меня были проблемы с выбором между `0&<3 3&>1 3>&1 ->2 -<&0 &-<0 0<&-` etc… (Я думаю, вероятно, потому, что синтаксис более репрезентативен для результата, то есть перенаправления, чем то, что делается, то есть открытие, закрытие или дублирование файловых дескрипторов).

Если это соответствует вашей ситуации, то, возможно, вам помогут следующие "правила", перенаправление всегда выглядит следующим образом:

```
lhs op rhs
```

- `lhs` всегда является описанием файла, то есть номером:
  - Либо мы хотим открыть, дублировать, переместить, либо мы хотим закрыть. Если op < , то есть неявный 0, если это `>` или `>>` , то есть неявный 1.

- `op` является `<`, `>`, `>>`, `>|`, или `<>`:
  - `<` будет ли расшифровщик файла прочитан, `lhs` будет `>` ли он записан, `>>` будут ли данные добавлены к файлу, `>|` перезаписать существующий файл или `<>` он будет одновременно прочитан и записан.
- `rhs` это то, что будет описывать дескриптор файла:
  - Это может быть имя файла, место, куда переходит другой дескриптор (`&1`) , или, `&-` , который закроет дескриптор файла.

Вам может не понравиться это описание, и вы сочтете его немного неполным или неточным, но я думаю, что это действительно помогает легко найти, что, скажем `&->0` , неверно.

Замечание по стилю

Оболочка довольно свободно описывает, что она считает допустимым перенаправлением. Хотя мнения, вероятно, расходятся, у этого автора есть несколько (сильных) рекомендаций:

- **Всегда** сохраняйте перенаправления "плотно сгруппированными", то есть **не** включайте пробелы нигде в синтаксисе перенаправления, кроме как в кавычках, если это требуется в RHS (например, имя файла, содержащее пробел). Поскольку оболочки в основном используют пробелы для разделения полей в целом, визуально намного понятнее, чтобы каждое перенаправление было разделено пробелом, но сгруппировано в куски, которые не содержат ненужных пробелов.
- **Всегда** ставьте пробел между каждым перенаправлением, а также между списком аргументов и первым перенаправлением.
- **Всегда** размещайте перенаправления вместе в самом конце команды после всех аргументов. Никогда не добавляйте перенаправление перед командой. Никогда не ставьте перенаправление в середине аргументов.
- **Никогда** не используйте Csh `&>foo` и `>&foo` сокращенные перенаправления. Используйте длинную форму `>foo 2>&1` . (см.: устаревший)

```
# Хорошо! Очевидно, что это простая команда с двумя аргументами и 4 перенаправлениями
cmd arg1 arg2 <myFile 3<&1 2>/dev/null >&2

# Хорошо!
{cmd1 <<<'мой ввод'; cmd2; } >someFile

# Плохо. Является ли "1" дескриптором файла или аргументом cmd? (ответ: это FD). Является ли
пробел после herestring частью входных данных? (ответ: нет).
# Перенаправления также не разделены каким-либо очевидным образом.
cmd 2> & 1 <<< материал

# Ужасно плохо. Трудно сказать, где находятся перенаправления и являются ли они вообще допус
тимыми перенаправлениями.
# На самом деле это одна команда с одним аргументом, назначением и тремя перенаправлениями.
foo=bar<baz bork<<< blarg>bleh
```

Заключение

Я надеюсь, что это руководство сработало для вас.

Я солгал, я не объяснял `1>&3-` , иди проверь руководство

Спасибо Стефану Шазеласу, у которого я украл и вступление, и пример ....

Вступление вдохновлено этим введением, вы также найдете там хорошее упражнение:

- Подробное введение в ввод-вывод и перенаправление ввода-вывода (http://tldp.org/LDP/abs/html/ioredirintro.html)

Последний пример взят из этого поста:

- comp.unix.shell: передача stdout и stderr в разные процессы (http://groups.google.com/group/comp.unix.shell/browse_thread/thread/64206d154894a4ef/ffe4c2e382034ed9#ffe

Смотрите также

- Внутренний: обзор синтаксиса перенаправления

Обсуждение

Армин, 2010/04/28 10:24 (), 2010/12/16 00:25 ()

Здравствуйте,

спасибо за это исчерпывающее объяснение.

Я искал решение следующей проблемы: я хочу выполнить сценарий оболочки (как удаленно через RSH, так и локально). Вывод из stdout и stderr должен идти в файл, чтобы увидеть, как выполняются скрипты на терминале, я хотел перенаправить вывод некоторых команд echo в тот же файл и в терминал. На основе этого руководства я реализовал следующее решение ~~(я не знаю, как создать амперсанд, поэтому вместо этого я использую "amp;")~~:

```
# сохраните стандартный вывод, перенаправьте стандартный вывод и стандартный вывод в фай
л
exec 3> &1 1>logfile 2>&1

# сделайте что-нибудь, что создаст вывод в стандартный вывод и / или стандартный вывод
# "восстановить" стандартный вывод, отправить сообщение как на терминал, так и в файл
exec 1> & 3
echo "мое сообщение" | tee -a logfile
exec 1>>logfile

# сделайте что-нибудь, что создаст вывод в стандартный вывод и / или стандартный вывод

выход
```

Как только происходит вывод в stderr, он работает не так, как я ожидал. Например.

```
exec 3> & 1 1> лог-файл 2> & 1
эхо "Привет, мир"
ls filedoesnotexist
exec 1> & 3
эхо "мое сообщение" | tee -файл журнала
ls filedoesnotexistyet
exec 1>> лог
-файл эхо "Снова привет"
ls filestilldoesnotexist
завершается
```

в результате получается следующее содержимое файла logfile:

```
Привет, мир
ls: filedoesnotexist: нет такого файла или каталога
ls: filedoesnotexistyet: Нет такого файла или каталога
ls: filestilldoesnotexist: нет такого файла или каталога
```

Т.е. тексты "мое сообщение" и "Снова здравствуйте" были перезаписаны выводом stderr команд ls.

Если я изменю в 1-м exec, чтобы добавить стандартный вывод в logfile (`exec 3>&1 1>>logfile 2>&1`), результат будет правильным:

```
Привет, мир
ls: filedoesnotexist: нет такого файла или каталога
мое сообщение
ls: filedoesnotexistyet: нет такого файла или каталога
Еще раз здравствуйте
ls: filestilldoesnotexist: такого файла или каталога нет
```

Если я изменю 7-ю строку на `exec 1>>logfile` **`2>&1`** результат, он будет выглядеть лучше, но неверно (по-прежнему отсутствует строка с "моим сообщением"):

```
Привет, мир
ls: filedoesnotexist: нет такого файла или каталога
ls: filedoesnotexistyet: нет такого файла или каталога
Еще раз здравствуйте
ls: filestilldoesnotexist: такого файла или каталога нет
```

Я, наконец, реализовал следующее решение, которое, возможно, не так элегантно, но работает:

```
exec 3>&1 4>&2 1> лог-файл 2> и 1
эхо "Привет, мир"
ls filedoesnotexist
exec 1> & 3 2> & 4
эхо "мое сообщение" | файл журнала tee -a
ls filedoesnotexistyet
exec 1>> файл журнала 2> & 1
эхо "Снова привет"
ls filestilldoesnotexist
завершается
```

Это приводит к тому, что вывод ошибки 2-го ls отправляется на терминал, а не в файл, что абсолютно нормально для моего случая.

Кто-нибудь может объяснить, что именно происходит? Я предполагаю, что это как-то связано с указателями файлов. Каково предпочтительное решение моей проблемы?

С уважением

Армин

PS: У меня есть некоторые проблемы с форматированием, особенно. с переводами строк и пустыми строками.

Ян Шампера, 2010/04/28 20:02 (), 2010/12/16 00:26 ()

Попробуйте это. Короче говоря:

- нет последующего набора / сброса файловых дескрипторов
- `tee` получает замену процесса в качестве выходного файла внутри а `cat` и перенаправление на FD1 (файл журнала)
- `tee` стандартный вывод перенаправляется на FD3 (терминал / исходный стандартный вывод)

```
эхо "мое сообщение" | tee > (cat > &1) > &3
```

Это специфичная вещь для Bash.

Для вики-причуд: я окружил ваш код `<code>...</code>` тегами. Извините, у меня нет решения проблемы с амперсандом. Похоже, в этом плагине ошибка. Это происходит только при "предварительном просмотре", но работает для реального просмотра.

typedeaF, 08.08.2011 15:35 ()

Я хочу реализовать функции Expect с помощью bash. То есть разработать программу-оболочку, которая назначит вызываемой программе перенаправлять ее 0-2 в именованные каналы. Затем оболочка откроет другой конец именованных каналов. Что-то вроде этого:

```
exec 3<>pipe.out
exec 4<>pipe.in

( PS3="введите выбор:"; выберите выбор в один два три; повторите "вы выбираете \"$ choic
e \""; готово )0<&4 1>&3 2>&1

во время чтения -u pipe.out строка
делает
регистр $line в
* выбор :)
 echo "$line"
 прочитайте i; echo i > & 4
 ;;
 EOF)
 эхо "поймано"EOF", выход";
 перерыв;;
 *)
эхо "$line"
 ;;
esac
выполнено
```

Конечно, это вообще не работает. Первая проблема заключается в том, что при использовании канала процесс зависает, пока не будут установлены оба конца канала. Вторая часть проблемы заключается в том, что встроенная функция чтения bash возвращает значение новой строки или параметр из N символов или разделителя X - ни то, ни другое не было бы полезно в данном случае. Таким образом, ввод цикла while никогда не "видит" приглашение "введите выбор:", поскольку в нем нет новой строки. Есть и другие проблемы.

Возможно ли заставить Bash сделать это? Есть предложения? Вот кое-что, что действительно работает.

терминал 1:

```
(exec 3готово)
```

терминал 2:

```
(PS3 ="введите выбор:"; выберите выбор в один два три; повторите "вы выбираете $ choic
e"; готово)1> pipe.out 2> &1
```

В этом случае программа продолжается, когда оба конца подключаются к каналу, но поскольку мы не перенаправляем стандартный идентификатор из канала для выбора, вы должны ввести выбор в терминале 2. Вы также заметите, что даже в этом сценарии терминал 1 не видит приглашение PS3, поскольку оно не возвращает новую строку.

Тони, 2012/02/10 00:41 (), 2012/02/10 05:35 ()

Здравствуйте,

Большое спасибо за подробное руководство. Я многое узнал о перенаправлении.

В моем скрипте я хочу перенаправить stderr в файл, а stderr и stdout - в другой файл. Я обнаружил, что эта конструкция работает, но я не совсем понимаю, как. Не могли бы вы объяснить?

```
((./cmd 2>&1 1>&3 | tee /tmp/stderr.log) 3>&1 1>&2) > / tmp/оба.log 2>&1
```

Кроме того, если я хочу сделать то же самое в скрипте, используя exec, чтобы избежать такого рода перенаправления в каждой команде скрипта, что мне делать?

Спасибо

Тони

Ян Шампера, 2012/02/10 05:46 ()

Вы перекачиваете `STDERR` команду в дескриптор 1, чтобы ее можно было транспортировать по каналу и рассматривать как ввод `tee` команды. В то же время вы перенаправляете оригинал `STDOUT` в дескриптор 3. `tee` Команда записывает ваш исходный стандартный вывод ошибки в файл и выводит его в its `STDOUT` .

Вне всей конструкции вы собираете свой исходный стандартный вывод (дескриптор 3) и исходный стандартный вывод ошибок (дескриптор 1 - через `tee`) в обычные дескрипторы (1 и 2), остальное - это простое перенаправление файлов для обоих дескрипторов.

Короче говоря, вы используете третий дескриптор для переключения обхода `tee` .

Я не знаю глобального метода (`exec` или подобного) с моей головы. Я могу представить, что вы можете взломать что-то с помощью замены процесса, но я не уверен.

jack, 2012/03/02 16:41 ()

Большое спасибо за эти объяснения! Только один момент, который меня смутил. В примере из comp.unix.shell вы написали: "Теперь для левой части второго канала ..." Иллюстрация к результату смутила меня, потому что я предполагал, что fds исходят из предыдущей иллюстрации, но затем я понял, что они исходят из их родительского процесса и, следовательно, из предыдущей предыдущей иллюстрации. Я думаю, было бы немного понятнее, если бы вы поместили ярлык на каждую из своих иллюстраций и более четко обозначили переход от одной иллюстрации к другой. В любом случае, еще раз большое спасибо.) джек (

Р.У. Эмерсон II, 2012/12/09 16:30 ()

Кажется, что каналы вводят постороннюю строку в EOF. Это правда? Попробуйте это:

```
объявляем tT="A \nB \ nC \ n" # Здесь должно быть три строки
echo -e "tT($ tT)" # Три строки, подтвержденные
echo -e "сортировка ($(сортировка <<< $ tT))" # Сортировка выводит три строки
echo -e " $tT" | sort # Сортировка выводит четыре строки
```

Когда три строки попадают в канал, выходят четыре строки. Проблема отсутствует в объекте here-string .

Какой полезный и крайне необходимый сайт! Спасибо!

Ян Шампера, 2012/12/16 13:13 (), 2012/12/16 13:14 ()

Я вижу, что эти дополнительные строки исходят из предыдущего echo:

```
бонсай @ядро: ~ $ echo -e "$ tT"
A
B
C

бонсай @ядро: ~ $
```

Это дополнительная новая строка, которую echo добавляет для завершения вывода. В вашем первом echo это новая строка после закрывающей скобки.

Вы можете проверить это при использовании `echo -n` (подавляет само генерируемое эхо новой строки)

Ханс Гинзель, 2015/10/02 09:03 ()

Спасибо за подробное руководство.

Пожалуйста, добавьте этот пример, http://stackoverflow.com/questions/3141738/duplicating-stdout-to-stderr (http://stackoverflow.com/questions/3141738/duplicating-stdout-to-stderr) .

echo foo | tee /dev/stderr

Существуют ли лучшие / более чистые решения? Похоже, что /dev/stderr может иметь проблемы в cron.

Ян Шампера, 2015/10/21 04:51 ()

Это функциональность самой оболочки, оболочка дублирует соответствующие файловые дескрипторы, когда видит эти имена файлов. Таким образом, это может зависеть от оболочки (или уровня совместимости оболочки), которую вы используете в cron.