Перенаправление

Исправь меня: продолжение следует

Перенаправление позволяет контролировать, куда отправляется вывод команды, и откуда поступает ввод команды. Это мощный инструмент, который вместе с конвейерами делает оболочку мощной. Операторы перенаправления проверяются всякий раз, когда выполняется простая команда.

При нормальных обстоятельствах открыты 3 файла, доступные по файловым дескрипторам 0, 1 и 2, все они подключены к вашему терминалу:

Имя	FD	Описание
`stdin`	0	стандартный поток ввода (например, клавиатура)
`stdout`	1	стандартный выходной поток (например, монитор)
`stderr`	2	стандартный поток вывода ошибок (обычно также на мониторе)

Термины "монитор" и "клавиатура" относятся к одному и тому же устройству, **терминалу** здесь. Проверьте предпочитаемый UNIX®-FAQ () для получения подробной информации, мне лень объяснять, что такое терминал 😉

Оба, `stdout` и `stderr` являются дескрипторами выходных файлов. Их разница заключается в **соглашении** о том, что программа выводит полезную нагрузку `stdout` и диагностические сообщения и сообщения об ошибках `stderr` . Если вы пишете скрипт, который выводит сообщения об ошибках, пожалуйста, убедитесь, что вы следуете этому соглашению!

Всякий раз, когда вы **называете** такой filedescriptor, то есть хотите перенаправить этот дескриптор, вы просто используете номер:

```
# это выполняет cat-команду и перенаправляет ее сообщения об ошибках
(stderr) на
cat-бит-корзину some_file.txt 2>/dev/null
```

Всякий раз, когда вы **ссылаетесь** на дескриптор, чтобы указать на его текущий целевой файл, вы используете " `&` ", за которым следует номер дескриптора:

```
# это выполняет эхо-команду и перенаправляет ее обычный вывод (станда
ртный вывод) на стандартное целевое
эхо-сообщение об ошибке "Произошла ошибка" 1> & 2
```

Операция перенаправления может быть где **угодно** в простой команде, поэтому эти примеры эквивалентны:

```
кот foo.txt bar.txt >new.txt
кошка >new.txt foo.txt bar.txt
>new.txt кошка foo.txt bar.txt
```

Каждый оператор перенаправления принимает одно или два слова в качестве операндов. Если вам нужно использовать операнды (например, имена файлов для перенаправления), которые содержат пробелы, вы **должны** заключить их в кавычки!

Допустимые цели и источники перенаправления

Этот синтаксис распознается всякий `TARGET` раз, когда используется спецификация или `SOURCE` спецификация (как показано ниже в подробных описаниях).

Синтаксис	Описание
`FILENAME`	ссылается на обычное, обычное имя файла из файловой системы (которое, конечно, тоже может быть FIFO. Просто все, на что вы можете ссылаться в файловой системе)
`&N`	ссылается на текущую цель / источник filedescriptor `N` ("дублирует" filedescriptor)
`&-`	закрывает перенаправленный filedescriptor, полезный вместо `> /dev/null` constructs (`> &-`)
`/dev/fd/N`	дублирует filedescriptor `N` , если `N` является допустимым целым числом
`/dev/stdin`	дублирует filedescriptor 0 (`stdin`)
`/dev/stdout`	дублирует filedescriptor 1 (`stdout`)
`/dev/stderr`	дублирует filedescriptor 2 (`stderr`)
`/dev/tcp/HOST/PORT`	предполагается `HOST` , что это допустимое имя хоста или IP-адрес, а `PORT` также допустимый номер порта или имя службы: перенаправление с / на соответствующий сокет TCP
`/dev/udp/HOST/PORT`	предполагается `HOST` , что это допустимое имя хоста или IP-адрес, а `PORT` также допустимый номер порта или имя службы: перенаправление из / в соответствующий сокет UDP

Если целевая / исходная спецификация не открывается, вся операция перенаправления завершается неудачей. Избегайте ссылок на файловые дескрипторы выше 9, поскольку вы можете столкнуться с файловыми дескрипторами, которые Bash использует внутри.

Перенаправление вывода

```
N > ЦЕЛЬ
```

Это перенаправляет номер дескриптора файла `N` на цель `TARGET`. Если `N` опущено, `stdout` предполагается (FD 1). Перед `TARGET` началом записи **усекается**.

Если опция `noclobber` установлена с помощью встроенного набора, это приведет к сбою перенаправления, когда `TARGET` имя обычного файла, который уже существует. Вы можете вручную переопределить это поведение, принудительно перезаписав с помощью оператора перенаправления `>|` вместо `>`.

Добавление перенаправленного вывода

```
N >> ЦЕЛЬ
```

Это перенаправляет номер дескриптора файла `N` на цель `TARGET`. Если `N` опущено, `stdout` предполагается (FD 1). `TARGET` **Не усекается** перед началом записи.

Перенаправление вывода и вывода ошибок

```
&> ЦЕЛЬ
```

```
> и ЦЕЛЬ
```

Этот специальный синтаксис перенаправляет оба, `stdout` и `stderr` на указанную цель. Это **эквивалентно**

```
> ЦЕЛЬ 2> &1
```

Начиная с Bash4, существует `&>>TARGET`, что эквивалентно `>> TARGET 2>&1`.

Этот синтаксис устарел и не должен использоваться. Смотрите страницу об устаревшем и устаревшем синтаксисе.

Добавление перенаправленного вывода и вывода с ошибкой

Чтобы добавить совокупное перенаправление `stdout` и `stderr` в файл, вы просто делаете

```
>> ФАЙЛ 2>&1
```

```
&>> ФАЙЛ
```

Транспортировка stdout и stderr по каналу

```
КОМАНДА1 2>&1 | КОМАНДА2
```

```
COMMAND1 |& COMMAND2
```

Перенаправление ввода

```
N < ИСТОЧНИК
```

Входной дескриптор `N` используется `SOURCE` в качестве источника данных. Если `N` значение опущено, предполагается filedescriptor 0 (`stdin`).

Здесь документы

```
<
...
ТЕГ
```

```
<<-ТЕГ
...
ТЕГ
```

Здесь-документ представляет собой перенаправление ввода с использованием исходных данных, указанных непосредственно в командной строке (или в скрипте), без "внешнего" источника. Оператор перенаправления `<<` используется вместе с тегом `TAG` , который используется для обозначения конца ввода позже:

```
# показать справку

кот <
Извините...
Для $PROGRAM помощь пока недоступна.
Хе-хе...
EOF
```

Как вы видите, замены возможны. Чтобы быть точным, в данных здесь-документа выполняются следующие замены и расширения:

- Расширение параметров
- Замена команд

- Арифметическое расширение

Вы можете избежать этого, указав тег:

```
cat <<"EOF"
Это не будет расширено: $PATH
EOF
```

И последнее, но не менее важное: если за оператором перенаправления `<<` следует `-` (тире), все **начальные вкладки** из данных документа будут проигнорированы. Это может быть полезно для создания оптического кода также при использовании here-documents.

Используемый вами тег **должен** быть единственным словом в строке, чтобы его можно было распознать как маркер конца документа.

Похоже, что here-documents (проверенные на версиях `1.14.7` `2.05b` и `3.1.17`) корректно завершаются, когда перед тегом end-of-here-document есть EOF () . Причина неизвестна, но, похоже, это сделано специально. В Bash 4 появилось предупреждающее сообщение, когда конец файла отображается до достижения тега.

Здесь строки

```
<<< СЛОВО
```

Строки here являются разновидностью документов here. Слово `WORD` берется для перенаправления ввода:

```
кот <<< "Привет, мир... $NAME здесь ... "
```

Просто остерегайтесь цитировать `WORD` , если оно содержит пробелы. В противном случае остальное будет задано как обычные параметры.

Строка here добавит к данным символ новой строки (`\n`).

Множественные перенаправления

Конечно, в строке может выполняться больше операций перенаправления. Порядок **важен**! Они оцениваются **слева направо**. Если вы хотите перенаправить оба `stderr` и `stdout` в один и тот же файл (например `/dev/null` , чтобы скрыть его), это **неправильный путь**:

```
echo
{
# вы можете протестировать с его  помощью # { echo OUTPUT; echo ERROR
S >&2; } - это имитация чего-то, что выводится на стандартный вывод и
ВЫВОД STDERR; ОШИБКИ echo >&2; } 2>&1  1>/ dev/null
```

Почему? Относительно просто:

- изначально `stdout` указывает на ваш терминал (вы его читаете)
- то же самое относится к `stderr` , он подключен к вашему терминалу
- `2>&1` перенаправляет `stderr` с терминала на цель для `stdout` : **терминала** (опять же ...)
- `1>/dev/null` перенаправляет `stdout` с вашего терминала на файл `/dev/null`

Что остается? `stdout` переходит к `/dev/null` , `stderr` все еще (или лучше: "снова") переходит к терминалу. Вы должны поменять порядок, чтобы заставить его делать то, что вы хотите:

```
echo { ВЫВОД; ОШИБКИ эха >&2; } 1>/dev/null 2>&1
```

Примеры

Как сделать программу тихой (предполагая, что весь вывод идет в `STDOUT` и `STDERR` ?

```
команда> /dev/null 2>&1
```

Смотрите также

- Внутреннее: иллюстрированное руководство по перенаправлению
- Внутренний: опция noclobber
- Внутренняя: встроенная команда exec
- Внутренний: простой синтаксический анализ и выполнение команд
- Внутренний: синтаксис подстановки процесса
- Внутренний: устаревший и устаревший синтаксис
- Внутренний: непереносимый синтаксис и использование команд

Обсуждение

syntax/redirection.txt Последнее редактирование: 2013/04/14 12:30 автор thebonsai

Bash Hackers Wiki