

Bash и дерево процессов

Дерево процессов

Процессы в UNIX®, в отличие от других систем, **организованы в виде дерева**. У каждого процесса есть родительский процесс, который запустил или отвечает за него. У каждого процесса есть своя **контекстная память** (не память, в которой процесс хранит свои данные, а память, в которой хранятся данные, которые не принадлежат непосредственно процессу, но необходимы для запуска процесса), то есть **среда**.

У каждого процесса есть свое **собственное** пространство среды.

Среда хранит, среди прочего, полезные для нас данные, **переменные среды**. Это строки в обычной NAME=VALUE форме, но они не связаны с переменными оболочки.

LANG Например, переменная с именем используется каждой программой, которая ищет ее в своей среде для определения текущей локали.

Внимание: переменная, которая устанавливается, например, с MYVAR=hello помощью , автоматически **не** является частью среды. Вам нужно поместить его в среду с помощью встроенной команды `bash export` :

```
экспорт MYVAR
```

Общие системные переменные, такие как PATH или HOME, обычно являются частью среды (как задается сценариями входа или программами).

Выполнение программ

На всех диаграммах дерева процессов используются имена типа "xterm" или "bash", но это просто для того, чтобы было легче понять, что происходит, это не означает, что эти процессы действительно выполняются.

Давайте кратко рассмотрим, что происходит, когда вы "выполняете программу" из командной строки Bash, программу типа "ls":

```
$ ls
```

Теперь Bash выполнит **два шага**:

- Он создаст копию самого себя
- Копия заменит себя программой "ls"

Копия Bash унаследует среду от основного процесса Bash: все переменные среды также будут скопированы в новый процесс. Этот шаг называется **разветвлением**.

На короткое время у вас есть дерево процессов, которое может выглядеть следующим образом...

```
xterm ----- bash ----- bash(копия)
```

... и после того, как "второй Bash" (копия) заменяет себя программой (копия выполняет ее), это может выглядеть так ls

```
xterm ----- bash ----- ls
```

Если все было в порядке, два шага приводили к запуску одной программы. Копия среды с первого шага (разветвления) становится средой для окончательной запущенной программы (в данном случае ls).

Что в этом такого важного? В нашем примере то, что программа ls делает внутри своей собственной среды, не может повлиять на среду ее родительского процесса (в данном случае, bash). Среда была скопирована при выполнении ls. При завершении ничто не "копируется обратно" в родительскую среду ls.

Bash играет с трубами

Каналы - очень мощный инструмент. Вы можете подключить выходные данные одного процесса к входным данным другого процесса. Мы не будем углубляться в конвейер на этом этапе, мы просто хотим посмотреть, как это выглядит в дереве процессов. Снова мы выполняем некоторые команды, на этот раз мы запустим ls и grep:

```
$ ls | grep myfile
```

В результате получается дерево, подобное этому:

```
+-- ls
xterm ----- bash --|
+-- grep
```

Обратите внимание еще раз, ls не может влиять на окружающую grep среду, grep не может влиять на окружающую ls среду, и grep ни ls или не может влиять на окружающую bash среду.

Как это связано с программированием оболочки ?!?

Ну, представьте себе некоторый код Bash, который считывает данные из канала. Например, внутренняя команда read, которая считывает данные из *stdin* и помещает их в переменную. Мы запускаем его в цикле здесь для подсчета входных строк:

```
счетчик =0

cat /etc/passwd | во время чтения; do ((counter++)); готово
повторить "Строки: $counter"
```

Что? Это 0? Да! Количество строк может быть не равно 0, но переменная \$counter все равно равна 0. Почему? Помните диаграмму сверху? Немного переписав его, мы имеем:

```
+-- cat /etc/passwd
xterm ----- bash --|
+-- bash (во время чтения; do ((counter++)); готово)
```

Видите взаимосвязь? Разветвленный процесс Bash будет считать строки как заклинание. Он также установит переменную `counter` в соответствии с указаниями. Но если все закончится, этот дополнительный процесс будет завершен - **ваша переменная "counter" исчезнет**. Вы видите 0, потому что в основной оболочке оно было 0 и не было изменено дочерним процессом!

Итак, как мы считаем строки? Легко: **избегайте подоболочки**. Детали не имеют значения, важно то, что оболочка, которая устанавливает счетчик, должна быть "основной оболочкой". Например:

```
счетчик =0

во время чтения; do ((counter++)); готово
```

Это почти не требует пояснений. `while` Цикл выполняется в **текущей оболочке**, счетчик увеличивается в **текущей оболочке**, все жизненно важное происходит в **текущей оболочке**, также `read` команда устанавливает переменную `REPLY` (значение по умолчанию, если ничего не задано), хотя мы ее здесь не используем.

Действия, создающие подоболочку

Bash создает **подоболочки** или **подпроцессы** для различных действий, которые он выполняет:

Выполнение команд

Как показано выше, Bash будет создавать подпроцессы каждый раз, когда он выполняет команды. В этом нет ничего нового.

Но если ваша команда является подпроцессом, который задает переменные, которые вы хотите использовать в своем основном скрипте, это не сработает.

Именно для этой цели существует `source` команда (также: команда `dot` `.`). Исходный код не выполняет скрипт, он импортирует код другого скрипта в текущую оболочку:

```
источник ./myvariables.sh
# эквивалентно:
. ./myvariables.sh
```

Трубы

Последний большой раздел был посвящен каналам, поэтому здесь нет примера.

Явная подболочка

Если вы группируете команды, заключая их в круглые скобки, эти команды выполняются внутри подболочки:

```
(следует echo PASSWD; cat /etc/passwd; следует echo GROUP; cat /etc/group) >output.txt
```


Замена команд

С помощью подстановки команд вы повторно используете вывод другой команды в виде текста в командной строке, например, для установки переменной. Другая команда выполняется в подболочке:

```
number_of_users=$(cat /etc/passwd | wc -l)
```

Обратите внимание, что в этом примере вторая подболочка была создана с помощью канала при подстановке команды:

```
+-- cat /etc/passwd
xterm ----- bash ----- bash (cmd. subst.) --|
+-- wc -l
```

 **Fix Me!** продолжение следует

Обсуждение

Роб Хаббард, [2015/06/26 09:25 \(\)](#)

У меня возник вопрос о том, как определить, какие переменные являются частью среды. Эта статья косвенно ответила на этот вопрос.

Итак, явно:

В команде `set` перечислены все переменные, которые были установлены с их значениями (выходные данные, как правило, подробные, поскольку они включают функции).

С другой стороны, команда `env` показывает только те переменные (опять же с их значениями), которые были `export` изменены в среде.

Этот сайт поддерживается Performing Databases - вашими
экспертами по администрированию баз данных

Bash Hackers Wiki



Если не указано иное, содержимое этой вики лицензируется по следующей лицензии:
Лицензия на бесплатную документацию GNU 1.3