Приемы написания скриптов в Bash из песочницы

Программирование*, *nix*

Администраторам Linux писать скрипты на Bash приходится регулярно. Ниже я привожу советы, как можно ускорить эту работу, а также повысить надежность скриптов.

Совет 1

Не пишите скриптов, которые выполняют действия ничего не спрашивая. Такие скрипты нужны довольно редко. А вот всевозможного «добра» для копирования, синхронизации, запуска чего-либо, хоть отбавляй. И если в любимом Midnight Commander Вы вдруг нажали не на тот скрипт, то с системой может произойти все что угодно. Это как правила дорожного движения — «написано кровью».

Совет 2

Отталкиваясь от предыдущего, в начало каждого скрипта неплохо помещать что-то вроде:

```
read -n 1 -p "Ты уверен, что хочешь запустить это (y/[a]): " AMSURE
[ "$AMSURE" = "y" ] || exit
echo "" 1>&2
```

Команда echo, кстати, здесь нужна потому, что после нажатия кнопки <y> у вас не будет перевода строки, следовательно, следующий любой вывод пойдет в эту же строку.

Совет 3

Это ключевой совет из всех. Для того, чтобы не писать каждый раз одно и то же — пользуйтесь библиотеками функций. Прочитав много статей по Bash, я вынужден констатировать, что этой теме уделяется мало внимания. Возможно в силу очевидности. Однако я считаю необходимым напомнить об этом. Итак.
Заведите свою библиотеку функций, например myfunc.sh и положите ее, например в /usr/bin. При написании скриптов она не только поможет сократить ваш труд, но и позволит одним махом доработать множество скриптов, если Вы улучшите какую-либо функцию.
Например, в свете совета 2 можно написать такую функцию:

```
myAskYN()
{
local AMSURE
if [ -n "$1" ] ; then
   read -n 1 -p "$1 (y/[a]): " AMSURE
else
   read -n 1 AMSURE
fi
echo "" 1>&2
```

```
if [ "$AMSURE" = "y" ] ; then
   return 0
else
   return 1
fi
}
```

Единственным необязательным параметром эта функция принимает строку вопроса. Если строка не задана — молчаливое ожидание нажатия (в случаях, когда скрипт уже успел вывести все что нужно еще до вызова этой функции). Таким образом, применение возможно такое:

```
myAskYN "Ты уверен, что хочешь запустить это?" || exit
```

Можно написать и еще одну аналогичную функцию myAskYNE, с буквой E на конце, в которой return заменить на exit. Тогда запись будет еще проще:

```
myAskYNE "Ты уверен, что хочешь запустить это?"
```

Плюсы очевидны: а) пишете меньше кода, б) код легче читать, в) не отвлекаетесь на мелочи, вроде приставки " (y/[a]): " к тесту (замечу, что [a] означает any, а забранная в квадратные кавычки указывает, что это по умолчанию).

И последнее здесь. Для того, чтобы использовать функции из нашей библиотеки, ее надо не забыть включить в сам скрипт:

```
#!/bin/bash
a1=myfunc.sh ; source "$a1" ; if [ $? -ne 0 ] ; then echo "Ошибка —
нет библиотеки функций $a1" 1>&2 ; exit 1 ; fi

myAskYN "Ты уверен, что хочешь запустить это?"
echo Run!
```

Я намеренно уложил весь вызов и обработку ошибки в одну строку, поскольку это вещь стандартная и не относится напрямую к логике скрипта. Зачем же ее растягивать на пол-экрана? Обратите также внимание, что имя скрипта присваивается переменной. Это позволяет задавать имя скрипта один раз, а стало быть, можно дублировать строку и заменить имя библиотеки, чтобы подключить другую библиотеку функций, если надо.

Теперь любой скрипт, начинающийся с этих трех строчек никогда не выполнит что-то без подтверждения. Предоставляю вам самим написать аналогичную myAskYN функцию, называемую myAskYESNO.

Совет 4

Разовьем успех и продемонстрируем несколько очевидных функций с минимальными комментариями.

```
sayWait()
{
   local AMSURE
   [ -n "$1" ] && echo "$@" 1>&2
   read -n 1 -p "(нажмите любую клавишу для продолжения)" AMSURE
```

```
   echo "" 1>&2
}

 cdAndCheck()
{
   cd "$1"
   if ! [ "$(pwd)" = "$1" ] ; then
      echo "!!Не могу встать в директорию $1 - продолжение невозможно. Выходим." 1>&2
      exit 1
   fi
}

 checkDir()
{
   if ! [ -d "$1" ] ; then
      if [ -z "$2" ] ; then
         echo "!!Нет директории $1 - продолжение невозможно. Выходим." 1>&2
      else
         echo "$2" 1>&2
      fi
      exit 1
   fi
}
checkFile()
{
   if ! [ -f "$1" ] ; then
      if [ -z "$2" ] ; then
         echo "!!Нет файла $1 - продолжение невозможно. Выходим." 1>&2
      else
         echo "$2" 1>&2
      fi
      exit 1
   fi
}
checkParm()
{
   if [ -z "$1" ] ; then
      echo "!!$2. Продолжение невозможно.  Выходим." 1>&2
      exit 1
   fi
}
```

Здесь обращу ваше внимание на постоянно встречающееся сочетание 1>&2 после echo. Дело в том, что ваши скрипты, возможно, будут выводить некую ценную

информацию. И не всегда эта информация влезет в экран, а потому ее неплохо бывает сохранить в файл или отправить на less. Комбинация 1>&2 означает перенаправление вывода на стандартное устройство ошибок. И когда вы вызываете скрипт таким образом:

```
my-script.sh > out.txt
my-script.sh | less
```

в нем не окажется лишних ошибочных и служебных сообщений, а только то, что вы действительно хотите вывести.

Совет 5

В Bash не очень хорошо обстоят дела с возвратом значения из функции. Однако при помощи собственной библиотеки этот вопрос легко решается. Просто заведите переменную, в которую функция будет заносить значение, а по выходу из функции анализируйте эту переменную. Кстати, объявление переменной неплохо поместить в начало самой библиотеки ваших функций. Также, вы можете завести и другие переменные, которые будете использовать повсеместно. Вот начало вашей библиотеки функций:

```
curPath=  # переменная с текущим абсолютным путем, где находится скрипт
cRes=     # переменная для возврата текстовых значений из функций
pYes=     # параметр --yes, который обсудим позднее
```

Теперь можем добавить к коллекции еще такую полезную функцию:

```
input1()
{
   local a1

   if [ -n "$1" ] ; then
      read -p "$1" -sn 1 cRes
   else
      read -sn 1 cRes
   fi

   # Проверка допустимых выборов
   while [ "$2" = "${2#*$cRes}" ] ; do
      read -sn 1 cRes
   done
   echo $cRes 1>&2
}
```

Вот пример ее использования:

```
cat <<'EOF'
Выбери желаемое действие:
------------------------
   a) Действие 1
   b) Действие 2
```

```
   .) Выход
EOF
input1 "Твой выбор: " "ab."
echo "Выбор был: $cRes"
```

Эта функция ограничивает нажатие клавиш до списка указанных (в пример это a, b, и точка). Никакие иные клавиши восприниматься не будут и при их нажатии ничего выводиться тоже не будет. Пример также показывает использование переменной возврата ($cRes). В ней возвращается буква, нажатая пользователем.

Совет 6

Какой скрипт без параметров? Об их обработке написано тонны литературы. Поделюсь своим видением.

1. Крайне желательно, чтобы параметры обрабатывались независимо от их последовательности.
2. Я не люблю использовать однобуквенные параметры (а следовательно и getopts) по той простой причине, что скриптов очень много, а букв мало. И запомнить, что для одного скрипта -r означает replace, для другого replicate, а для третьего вообще remove практически невозможно. Поэтому я использую 2 нотации, причем одновременно: а) --show-files-only, б) -sfo (как сокращение от предыдущего). Практика показывает, что такие ключи запоминаются мгновенно и очень надолго.
3. Скрипт должен выдавать ошибку на неизвестный ему ключ. Это частично поможет выявить ошибки при написании параметров.
4. Из совета 2 возьмем правило: никогда не запускать скрипт без подтверждения. Но добавим к этому важное исключение — если не указан ключ --yes (ключ, конечно, может быть любым).
5. Ключи могут сопровождаться значением. В этом случае для длинных ключей действует такое правило: --source-file=my.txt (написание через равно), а для коротких такое: -sf my.txt (через пробел).

В этом свете обработка параметров может выглядеть так:

```
while [ 1 ] ; do
   if [ "$1" = "--yes" ] ; then
      pYes=1
   elif [ "${1#--source-file=}" != "$1" ] ; then
      pSourceFile="${1#--source-file=}"
   elif [ "$1" = "-sf" ] ; then
      shift ; pSourceFile="$1"
   elif [ "${1#--dest-file=}" != "$1" ] ; then
      pDestFile="${1#--dest-file=}"
   elif [ "$1" = "-df" ] ; then
      shift ; pDestFile="$1"
   elif [ -z "$1" ] ; then
```

```
      break # Ключи кончились
   else
      echo "Ошибка: неизвестный ключ" 1>&2
      exit 1
   fi
   shift
done

checkParm "$pSourceFile" "Не задан исходный файл"
checkParm "$pDestFile" "Не задан выходной файл"

if [ "$pYes" != "1" ] ; then
   myAskYNE "Ты уверен, что хочешь запустить это?"
fi
echo "source=$pSourceFile, destination=$pDestFile"
```

Этот код дает следующие возможности:

- `./test.sh -sf mysource -df mydest`
- `./test.sh --source-file=mysource --dest-file=mydest`
- `./test.sh --source-file=mysource --dest-file=mydest --yes`
- Параметры могут задаваться в любом порядке и комбинации полной и сокращенной формы ключей.
- Поскольку параметры обязательны, то присутствует проверка их наличия (но не корректности), благодаря checkParm.
- Если отсутствует ключ --yes, обязательно возникнет запрос подтверждения.

Это базовая часть, которую можно развивать и дальше. Например, добавим пару функций обработки параметров в нашу библиотеку:

```
procParmS()
{
   [ -z "$2" ] && return 1
   if [ "$1" = "$2" ] ; then
      cRes="$3"
      return 0
   fi
   return 1
}
procParmL()
{
   [ -z "$1" ] && return 1
   if [ "${2#$1=}" != "$2" ] ; then
      cRes="${2#$1=}"
      return 0
   fi
   return 1
```

```
}
```

При этом цикл обработки параметров будет выглядеть гораздо более удобоваримым:

```
while [ 1 ] ; do
   if [ "$1" = "--yes" ] ; then
      pYes=1
   elif procParmS "-sf" "$1" "$2" ; then
      pSourceFile="$cRes" ; shift
   elif procParmL "--source-file" "$1" ; then
      pSourceFile="$cRes"
   elif procParmS "-df" "$1" "$2" ; then
      pDestFile="$cRes" ; shift
   elif procParmL "--dest-file" "$1" ; then
      pDestFile="$cRes"
   elif [ -z "$1" ] ; then
      break # Ключи кончились
   else
      echo "Ошибка: неизвестный ключ" 1>&2
      exit 1
   fi
   shift
done
```

Фактически, этот цикл можно копировать из скрипта в скрипт не задумываясь ни о чем, кроме названий ключей и имени переменной для этого ключа. Причем они в данном случае не повторяются и возможность ошибки исключена.

Нет предела совершенству, и можно еще долго «улучшать» функции, например в procParmS проверить на непустое значение третий параметр и вывалиться по ошибке в таком случае. И так далее.

Файл библиотеки функций из этого примера можно скачать здесь.

Тестовый файл здесь.