【Shell】linux 下source、sh、bash、./執行指令碼的區別

linux 下source、sh、bash、./執行指令碼的區別 - IT閱讀 (itread01.com)

1、source命令用法：
source FileName
作用:在當前bash環境下讀取並執行FileName中的命令。該filename檔案可以無"執行許可權"
注：該命令通常用命令“.”來替代。
如：source .bash_profile
. .bash_profile兩者等效。
source(或點)命令通常用於重新執行剛修改的初始化文件。
source命令(從 C Shell 而來)是bash shell的內建命令。
點命令，就是個點符號，(從Bourne Shell而來)。
source的程式主體是bash，指令碼中的$0變數的值是bash，而且由於作用於當前bash環境，指令碼中set的變數將直接起效

2、 sh, bash的命令用法：
sh/bash FileName
作用:開啟一個子shell來讀取並執行FileName中命令。該filename檔案可以無"執行許可權"
注：執行一個shell指令碼時會啟動另一個命令直譯器.
每個shell指令碼有效地執行在父shell(parent shell)的一個子程序裡.
這個父shell是指在一個控制終端或在一個xterm視窗中給你命令指示符的程序.
shell指令碼也可以啟動他自己的子程序.
這些子shell(即子程序)使指令碼並行地，有效率地地同時執行指令碼內的多個子任務.
在ubuntu中sh只是bash的一個連結。
由於是在子shell中執行，指令碼設定的變數不會影響當前shell。

3、./的命令用法：
./FileName
作用:開啟一個子shell來讀取並執行FileName中命令。該filename檔案需要"執行許可權"
注：執行一個shell指令碼時會啟動另一個命令直譯器.
每個shell指令碼有效地執行在父shell(parent shell)的一個子程序裡.
這個父shell是指在一個控制終端或在一個xterm視窗中給你命令指示符的程序.
shell指令碼也可以啟動他自己的子程序.
這些子shell(即子程序)使指令碼並行地，有效率地地同時執行指令碼內的多個子任務.
由於是在子shell中執行，指令碼設定的變數不會影響當前shell。
exec和source都屬於bash內部命令（builtins commands），在bash下輸入man exec或man source可以檢視所有的內部命令資訊。
bash shell的命令分為兩類：外部命令和內部命令。外部命令是通過系統呼叫或獨立的程式實現的，如sed、awk等等。內部命令是由特殊的檔案格式（.def）所實現，如cd、history、exec等等。

在說明exe和source的區別之前，先說明一下fork的概念。

fork是linux的系統呼叫，用來建立子程序（child process）。子程序是父程序(parent process)的一個副本，從父程序那裡獲得一定的資源分配以及繼承父程序的環境。子程序與父程序唯一不同的地方在於pid（process id）。

環境變數（傳給子程序的變數，遺傳性是本地變數和環境變數的根本區別）只能單向從父程序傳給子程序。不管子程序的環境變數如何變化，都不會影響父程序的環境變數。

shell script:

有兩種方法執行shell scripts，一種是新產生一個shell，然後執行相應的shell scripts；一種是在當前shell下執行，不再啟用其他shell。
新產生一個shell然後再執行scripts的方法是在scripts檔案開頭加入以下語句
#!/bin/sh
一般的script檔案(.sh)即是這種用法。這種方法先啟用新的sub-shell（新的子程序）,然後在其下執行命令。
另外一種方法就是上面說過的source命令，不再產生新的shell，而在當前shell下執行一切命令。
source:

source命令即點(.)命令。

在bash下輸入man source，找到source命令解釋處，可以看到解釋"Read and execute commands from filename in the current shell environment and ..."。從中可以知道，source命令是在當前程序中執行引數檔案中的各個命令，而不是另起子程序(或sub-shell)。

exec:

在bash下輸入man exec，找到exec命令解釋處，可以看到有"No new process is created."這樣的解釋，這就是說exec命令不產生新的子程序。那麼exec與source的區別是什麼呢？

exec命令在執行時會把當前的shell process關閉，然後換到後面的命令繼續執行。

* fork (/directory/script.sh)
　fork是最普通的，就是直接在腳本里面用/directory/script.sh來呼叫script.sh這個指令碼.執行的時候開一個sub-shell執行呼叫的指令碼，sub-shell執行的時候，parent-shell還在。sub-shell執行完畢後返回parent-shell. sub-shell從parent-shell繼承環境變數.但是sub-shell中的環境變數不會帶回parent-shell
* source（source /directory/script.sh)
　與fork的區別是不新開一個sub-shell來執行被呼叫的指令碼，而是在同一個shell中執行. 所以被呼叫的指令碼中宣告的變數和環境變數，都可以在主指令碼中得到和使用.
* exec (exec /directory/script.sh)
　exec與fork不同，不需要新開一個sub-shell來執行被呼叫的指令碼. 被呼叫的指令碼與父指令碼在同一個shell內執行。但是使用exec呼叫一個新指令碼以後，父指令碼中exec行之後的內容就不會再執行了。這是exec和source的區別。

下面用一個例子來講解

1.sh

```
#!/bin/bash

A=B

echo "PID for 1.sh before exec/source/fork:$"
export A

echo "1.sh: \$A is $A"

case $1 in
	exec)
		echo "using exec..."
		exec ./2.sh ;;
	source)
		echo "using source..."
		. ./2.sh ;;
	*)
		echo "using fork by default..."
		./2.sh ;;
esac

echo "PID for 1.sh after exec/source/fork:$"
echo "1.sh: \$A is $A"
```

2.sh

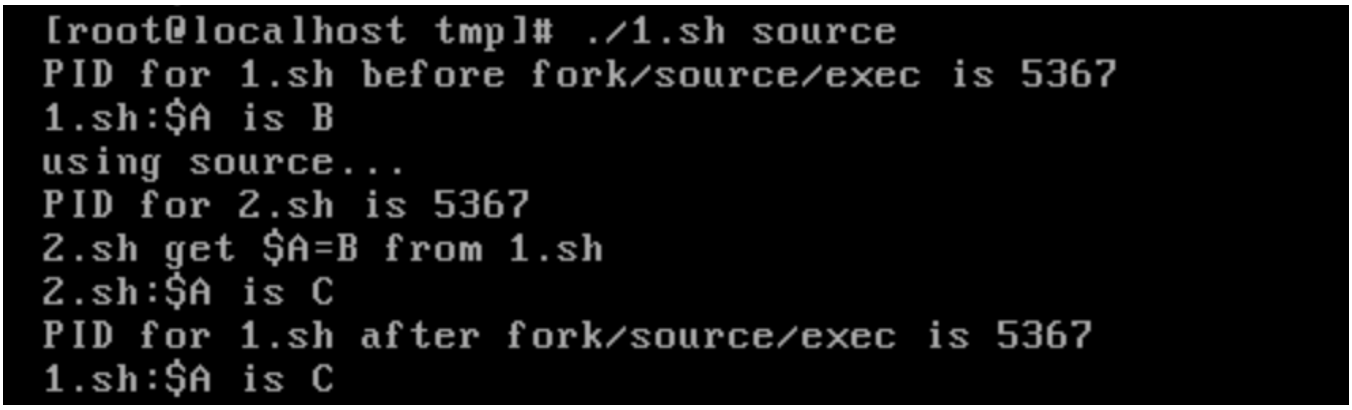
```
#!/bin/bash

echo "PID for 2.sh: $"
echo "2.sh get \$A=$A from 1.sh"

A=C

export A

echo "2.sh: \$A is $A"
```

下面在命令列中去執行
./1.sh fork

```
[root@localhost tmp]# ./1.sh fork
PID for 1.sh before fork/source/exec is 5344
1.sh:$A is B
using fork...
PID for 2.sh is 5345
2.sh get $A=B from 1.sh
2.sh:$A is C
PID for 1.sh after fork/source/exec is 5344
1.sh:$A is B
```

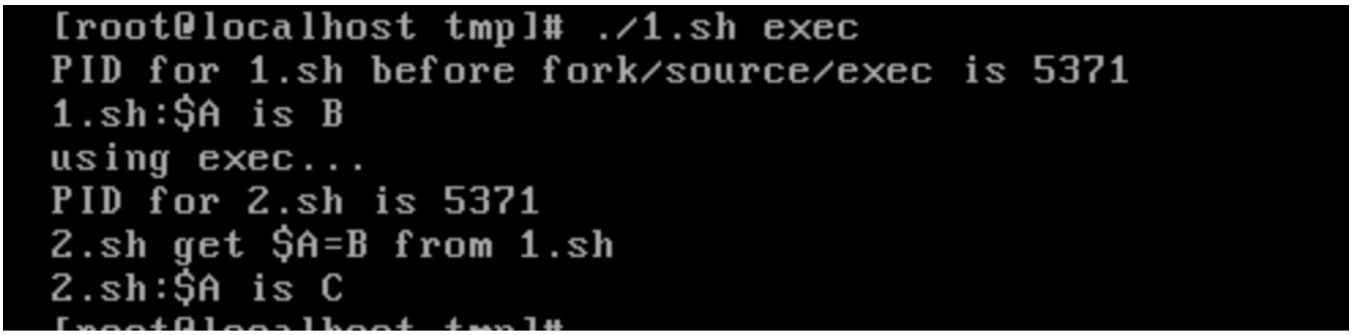
可以看到，1.sh是在父程序中執行，2.sh是在子程序中執行的，父程序的PID是5344，而子程序的是5345，當子程序執行完畢後，控制權返回到父程序。同時，在子程序改變環境變數A的值不會影響到父程序。

./1.sh source



由結果可知，1.sh和2.sh都是在同一程序中執行的，PID為5367

./1.sh exec



可知，兩個指令碼都是在同一程序中執行，但是請注意，使用exec終止了原來的父程序，因此，可以看到

```
echo "PID for 1.sh after exec/source/fork:$"

echo "1.sh: \$A is $A"
```

這兩個命令沒有執行

由這個例子，便大致可瞭解它們的區別了

修訂版本 #2
★由 treeman 建立於 21 ﷺ 2022 10:53:46
✎由 treeman 更新於 10 ﷺ 2024 15:22:45