Shell 脚本执行的几种方式

2016 年 07 月 16 日

1 Shell 脚本执行的方式

bash[1] 的命令分为两类：外部命令和内部命令。外部命令是通过系统调用或独立的程序实现的，如 sed、awk 等。内部命令是由特殊的文件格式（.def）所实现，如 cd、history、exec 等。fork 是 linux 的系统调用，用来创建子进程。exec 和 source 都属于 bash 内部命令（builtins commands）。

1.1 fork

fork(/directory/script.sh)：如果 shell 中包含执行命令，那么子命令并不影响父级的命令，在子命令执行完后再执行父级命令。子级的环境变量不会影响到父级。

fork 是最普通的，就是直接在脚本里面用 /directory/script.sh 来调用 script.sh 这个脚本。运行的时候开一个 sub-shell 执行调用的脚本，sub-shell 执行的时候，parent-shell 还在。sub-shell 执行完毕后返回 parent-shell。sub-shell 从 parent-shell 继承环境变量，但是 sub-shell 中的环境变量不会带回 parent-shell。

子进程是父进程 (parent process) 的一个副本，从父进程那里获得一定的资源分配以及继承父进程的环境。子进程与父进程唯一不同的地方在于 pid（process id）。

环境变量（传给子进程的变量，遗传性是本地变量和环境变量的根本区别）只能单向从父进程传给子进程。不管子进程的环境变量如何变化，都不会影响父进程的环境变量。

1.2 exec

exec(exec /directory/script.sh)：执行子级的命令后，不再执行父级命令。

exec 命令在执行时会把当前的 shell process 关闭，然后换到后面的命令继续执行。exec 与 fork 不同，不需要新开一个 sub-shell 来执行被调用的脚本。被调用的脚本与父脚本在同一个 shell 内执行。但是使用 exec 调用一个新脚本以后，父脚本中 exec 行之后的内容就不会再执行了。这是 exec 和 source 的区别。

1.3 source/（.）

source(source /directory/script.sh)：执行子级命令后继续执行父级命令，同时子级设置的环境变量会影响到父级的环境变量。 source 命令即点 (.) 命令。

与 fork 的区别是不新开一个 sub-shell 来执行被调用的脚本，而是在同一个 shell 中执行。所以被调用的脚本中声明的变量和环境变量, 都可以在主脚本中得到和使用。

[1] 本文档以 bash 为例进行说明

2 代码说明参考

以下代码取自于网络，但是能够很清晰地说明 Shell 脚本的几种执行方式。

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"
```

执行情况：

$ ./1.sh

```
PID for 1.sh before exec/source/fork:5845364
```

```
2 1.sh: $A is B
3 using fork by default…
4 PID for 2.sh: 5242940
5 2.sh get $A=B from 1.sh
6 2.sh: $A is C
7 PID for 1.sh after exec/source/fork:5845364
8 1.sh: $A is B
```

$ ./1.sh exec

```
1 PID for 1.sh before exec/source/fork:5562668
2 1.sh: $A is B
3 using exec…
4 PID for 2.sh: 5562668
5 2.sh get $A=B from 1.sh
6 2.sh: $A is C
```

$ ./1.sh source

```
1 PID for 1.sh before exec/source/fork:5156894
2 1.sh: $A is B
3 using source…
4 PID for 2.sh: 5156894
5 2.sh get $A=B from 1.sh
6 2.sh: $A is C
7 PID for 1.sh after exec/source/fork:5156894
8 1.sh: $A is C
```

3 其他说明

系统调用 exec 是以新的进程去代替原来的进程，但进程的 PID 保持不变。因此，可以这样认为，exec 系统调用并没有创建新的进程，只是替换了原来进程上下文的内容。原进程的代码段，数据段，堆栈段被新的进程所代替。

一个进程主要包括以下几个方面的内容:

1. 一个可以执行的程序;

2. 与进程相关联的全部数据 (包括变量，内存，缓冲区);

3. 程序上下文 (程序计数器 PC, 保存程序执行的位置)。

exec 是一个函数簇，由 6 个函数组成，分别是以 excl 和 execv 打头的。执行 exec 系统调用，一般都是这样，用 fork() 函数新建立一个进程，然后让进程去执行 exec 调用。我们知道，在 fork() 建立新进程之后，父进各与子进程共享代码段，但数据空间是分开的，但父进程会把自己数据空间的内容 copy 到子进程中去，还有上下文也会 copy 到子进程中去。而为了提高效率，采用一种写时 copy 的策略，即创建子进程的时候，并不 copy 父进程的地址空间，父子进程拥有共同的地址空间，只有当子进程需要写入数据时 (如向缓冲区写入数据), 这时候会复制地址空间，复制缓冲区到子进程中去。从而父子进程拥有独立的地址空间。而对于 fork() 之后执行 exec 后，这种策略能够很好的提高效率，如果一开始就 copy, 那么 exec 之后，子进程的数据会被放弃，被新的进程所代替。