## 一、提取错误日志脚本说明

完成整个功能需求的一共由四个脚本组成，分别如下：

1、FindErrorLogt.sh——这个脚本是最先也是最主要的，实现提取指定目录下的错误日志。

注：如果以后要提取其他路径下的日志，直接打开此文件修改路径变量就行。

2、SoloErrorLog.sh——这个脚本是后来加的，因为出现了这样一种情况就是：有些错误日志不包含在任何的用户工号下，也即就是用户还没登陆就已经产生了这样的错误，所以为了不漏掉这样的错误，特此加了这样一个脚本。

注：之所以不把这个功能也一并写在FindErrorLogt.sh这个文件中，是因为这样要造成FindErrorlogt.sh文件比较大的改动，同时为了解耦性，方便以后维护脚本。

3、AutoExec.sh——这个脚本也是后来加的，是为了实现在ATM上自动定期的执行脚本。

这个脚本的主要功能就是将所有实现自动化执行需要用到的命令综合在一起。

注：目前是提取的公网上的错误日志，如果要换成其他的服务器，可以打开这个脚本修改对应的服务器ip就行了。

## 二、shell编程总结

另外总结一下在实现这些脚本的过程中一些注意的地方，也算是对自己学习shell脚本以来的一个归纳。

1、在写脚本的时候，需特别注意添加空格或换行或分号

eg text命令： $[ 5 -lt 10 ]左方括号后须有空格，右方括号前也须有空格

2、shell脚本中函数定义： foo() { 后须有空格或换行

3、shell脚本中，若同一行内写有两条命令须加分号

4、与C语言不同的是，shell脚本中，返回值0表真，1表假。

5、不同系统编码格式引起的：在windows系统中编辑的.sh文件可能有不可见字符，所以在Linux系统下执行会报异常信息。

解决：1）在windows下转换：

利用一些编辑器如UltraEdit或EditPlus等工具先将脚本编码转换，再放到Linux中执行。转换方式如下（UltraEdit）：File-->Conversions-->DOS->UNIX即可。

2）也可在Linux中转换：

首先要确保文件有可执行权限

#sh>chmod a+x filename

然后修改文件格式

#sh>vi filename

利用如下命令查看文件格式

:set ff 或 :set fileformat

可以看到如下信息

fileformat=dos 或 fileformat=unix

利用如下命令修改文件格式

:set ff=unix 或 :set fileformat=unix

6、在写shell脚本时一定要注意相对路径与绝对路径

7、不要将$()写成$[]

8、注意单引号与双引号的区别

9、注意数值与字符串的比较的差异

10、多参数函数调用

11、字符串替换的问题：

[chengmo@localhost ~]$ test='c:/windows/boot.ini'

[chengmo@localhost ~]$ echo ${test/\//\\}

c:\windows/boot.ini

[chengmo@localhost ~]$ echo ${test//\//\\}

c:\windows\boot.ini

${变量/查找/替换值} 一个“/”表示替换第一个，”//”表示替换所有,当查找中出现了：”/”请加转义符”\/”表示。

12、shell脚本的三种调试方法：

-n 读一遍脚本中的命令但不执行，用来检查脚本中的语法错误

-v 一边执行脚本，一边将执行过的脚本命令打印到标准输出端

-x 提供跟踪执行信息，将执行的每一条命令和结果一次打印出来

这应该就像C语言的gdb一样有用吧，有助于检查错误

使用这些选项有三种方法

1.在命令行提供参数：$sh -x script.sh

2.脚本开头提供参数：#!/bin/sh -x

3.在脚本中用set命令启用or禁用参数：其中set -x表启用，set +x表禁用

13、shell文件互相调用

fork ( /directory/script.sh)

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

exec (exec /directory/script.sh)

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

source (source /directory/script.sh)

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。

可以通过下面这两个脚本来体会三种调用方式的不同:

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"

注：这两个脚本中的参数$$用于返回脚本的pid值，这个是例子是想通过显示pid号区别，两个脚本是分开执行还是同一进程里执行。当执行完脚本2后，脚本1后面的内容是否还执行。

执行情况：

$ ./1.sh

PID for 1.sh before exec/source/fork:5845

1.sh: $A is B

using fork by default…

PID for 2.sh: 5242

2.sh get $A=B from 1.sh

2.sh: $A is C

PID for 1.sh after exec/source/fork:5845

1.sh: $A is B

fork方式可以看出，两个脚本都执行了，运行顺序为1-2-1，从两者的PID值，可以看出，两个脚本是分成两个进程运行的。

$ ./1.sh exec

PID for 1.sh before exec/source/fork:5562

1.sh: $A is B

using exec…

PID for 2.sh: 5562

2.sh get $A=B from 1.sh

2.sh: $A is C

exec方式运行的结果是，2执行完成后，不再回到1。运行顺序为1-2。从pid值看，两者是在同一进程中运行的。

$ ./1.sh source

PID for 1.sh before exec/source/fork:5156

1.sh: $A is B

using source…

PID for 2.sh: 5156

2.sh get $A=B from 1.sh

2.sh: $A is C

PID for 1.sh after exec/source/fork:5156

1.sh: $A is C

source方式的结果是两者在同一进程里运行。该方式相当于把两个脚本先合并再运行。