脚本除错

网道（WangDoc.com），互联网文档计划

本章介绍如何对 Shell 脚本除错。

目录 [隐藏]

1. 常见错误

编写 Shell 脚本的时候，一定要考虑到命令失败的情况，否则很容易出错。

```
#! /bin/bash

dir_name=/path/not/exist

cd $dir_name
rm *
```

上面脚本中，如果目录 `$dir_name` 不存在， `cd $dir_name` 命令就会执行失败。这时，就不会改变当前目录，脚本会继续执行下去，导致 `rm *` 命令删光当前目录的文件。

如果改成下面的样子，也会有问题。

```
cd $dir_name && rm *
```

上面脚本中，只有 `cd $dir_name` 执行成功，才会执行 `rm *` 。但是，如果变量 `$dir_name` 为空， `cd` 就会进入用户主目录，从而删光用户主目录的文件。

下面的写法才是正确的。

```
[[ -d $dir_name ]] && cd $dir_name && rm *
```

上面代码中，先判断目录 `$dir_name` 是否存在，然后才执行其他操作。

如果不放心删除什么文件，可以先打印出来看一下。

```
[[ -d $dir_name ]] && cd $dir_name && echo rm *
```

上面命令中， `echo rm *` 不会删除文件，只会打印出来要删除的文件。

2. bash 的 -x 参数

`bash` 的 `-x` 参数可以在执行每一行命令之前，打印该命令。一旦出错，这样就比较容易追查。

下面是一个脚本 `script.sh` 。

```
# script.sh
echo hello world
```

加上 `-x` 参数，执行每条命令之前，都会显示该命令。

链接

本文源码

代码仓库

反馈

```
$ bash -x script.sh
+ echo hello world
hello world
```

上面例子中，行首为 `+` 的行，显示该行是所要执行的命令，下一行才是该命令的执行结果。

下面再看一个 `-x` 写在脚本内部的例子。

```
#! /bin/bash -x
# trouble: script to demonstrate common errors

number=1
if [ $number = 1 ]; then
  echo "Number is equal to 1."
else
  echo "Number is not equal to 1."
fi
```

上面的脚本执行之后，会输出每一行命令。

```
$ trouble
+ number=1
+ '[' 1 = 1 ']'
+ echo 'Number is equal to 1.'
Number is equal to 1.
```

输出的命令之前的 `+` 号，是由系统变量 `PS4` 决定，可以修改这个变量。

```
$ export PS4='$LINENO + '
$ trouble
5 + number=1
7 + '[' 1 = 1 ']'
8 + echo 'Number is equal to 1.'
Number is equal to 1.
```

另外， `set` 命令也可以设置 Shell 的行为参数，有利于脚本除错，详见《set 命令》一章。

3. 环境变量

有一些环境变量常用于除错。

3.1 LINENO

变量 `LINENO` 返回它在脚本里面的行号。

```
#!/bin/bash

echo "This is line $LINENO"
```

执行上面的脚本 `test.sh` ， `$LINENO` 会返回 `3` 。

```
$ ./test.sh
This is line 3
```

3.2 FUNCNAME

变量 `FUNCNAME` 返回一个数组，内容是当前的函数调用堆栈。该数组的0号成员是当前调用的函数，1号成员是调用当前函数的函数，以此类推。

```
#!/bin/bash

function func1()
{
  echo "func1: FUNCNAME0 is ${FUNCNAME[0]}"
  echo "func1: FUNCNAME1 is ${FUNCNAME[1]}"
  echo "func1: FUNCNAME2 is ${FUNCNAME[2]}"
  func2
}

function func2()
{
  echo "func2: FUNCNAME0 is ${FUNCNAME[0]}"
  echo "func2: FUNCNAME1 is ${FUNCNAME[1]}"
  echo "func2: FUNCNAME2 is ${FUNCNAME[2]}"
```

```
}

func1
```

执行上面的脚本 `test.sh` ，结果如下。

```
$ ./test.sh
func1: FUNCNAME0 is func1
func1: FUNCNAME1 is main
func1: FUNCNAME2 is
func2: FUNCNAME0 is func2
func2: FUNCNAME1 is func1
func2: FUNCNAME2 is main
```

上面例子中，执行 `func1` 时，变量 `FUNCNAME` 的0号成员是 `func1` ，1号成员是调用 `func1` 的主脚本 `main` 。执行 `func2` 时，变量 `FUNCNAME` 的0号成员是 `func2` ，1号成员是调用 `func2` 的 `func1` 。

3.3 BASH_SOURCE

变量 `BASH_SOURCE` 返回一个数组，内容是当前的脚本调用堆栈。该数组的0号成员是当前执行的脚本，1号成员是调用当前脚本的脚本，以此类推，跟变量 `FUNCNAME` 是一一对应关系。

下面有两个子脚本 `lib1.sh` 和 `lib2.sh` 。

```
# lib1.sh
function func1()
{
  echo "func1: BASH_SOURCE0 is ${BASH_SOURCE[0]}"
  echo "func1: BASH_SOURCE1 is ${BASH_SOURCE[1]}"
  echo "func1: BASH_SOURCE2 is ${BASH_SOURCE[2]}"
  func2
}
```

```
# lib2.sh
function func2()
```

```
{
  echo "func2: BASH_SOURCE0 is ${BASH_SOURCE[0]}"
  echo "func2: BASH_SOURCE1 is ${BASH_SOURCE[1]}"
  echo "func2: BASH_SOURCE2 is ${BASH_SOURCE[2]}"
}
```

然后，主脚本 `main.sh` 调用上面两个子脚本。

```
#!/bin/bash
# main.sh

source lib1.sh
source lib2.sh

func1
```

执行主脚本 `main.sh` ，会得到下面的结果。

```
$ ./main.sh
func1: BASH_SOURCE0 is lib1.sh
func1: BASH_SOURCE1 is ./main.sh
func1: BASH_SOURCE2 is
func2: BASH_SOURCE0 is lib2.sh
func2: BASH_SOURCE1 is lib1.sh
func2: BASH_SOURCE2 is ./main.sh
```

上面例子中，执行函数 `func1` 时，变量 `BASH_SOURCE` 的0号成员是 `func1` 所在的脚本 `lib1.sh` ，1号成员是主脚本 `main.sh` ；执行函数 `func2` 时，变量 `BASH_SOURCE` 的0号成员是 `func2` 所在的脚本 `lib2.sh` ，1号成员是调用 `func2` 的脚本 `lib1.sh` 。

3.4 BASH_LINENO

变量 `BASH_LINENO` 返回一个数组，内容是每一轮调用对应的行号。 `${BASH_LINENO[$i]}` 跟 `${FUNCNAME[$i]}` 是一一对应关系，表示 `${FUNCNAME[$i]}` 在调用它的脚本文件 `${BASH_SOURCE[$i+1]}` 里面的行号。

下面有两个子脚本 `lib1.sh` 和 `lib2.sh` 。

```
# lib1.sh
function func1()
{
  echo "func1: BASH_LINENO is ${BASH_LINENO[0]}"
  echo "func1: FUNCNAME is ${FUNCNAME[0]}"
  echo "func1: BASH_SOURCE is ${BASH_SOURCE[1]}"

  func2
}
```

```
# lib2.sh
function func2()
{
  echo "func2: BASH_LINENO is ${BASH_LINENO[0]}"
  echo "func2: FUNCNAME is ${FUNCNAME[0]}"
  echo "func2: BASH_SOURCE is ${BASH_SOURCE[1]}"
}
```

然后，主脚本 `main.sh` 调用上面两个子脚本。

```
#!/bin/bash
# main.sh

source lib1.sh
source lib2.sh

func1
```

执行主脚本 `main.sh` ，会得到下面的结果。

```
$ ./main.sh
func1: BASH_LINENO is 7
func1: FUNCNAME is func1
func1: BASH_SOURCE is main.sh
func2: BASH_LINENO is 8
```

```
func2: FUNCNAME is func2
func2: BASH_SOURCE is lib1.sh
```

上面例子中，函数 `func1` 是在 `main.sh` 的第7行调用，函数 `func2` 是在 `lib1.sh` 的第8行调用的。

set 命令，shopt 命令　:temp 命令，trap 命令

联系：contact@wangdoc.com