科学大数据管理系统

编 号

密 级

阶段标记 S

页 数

**名称：科学大数据管理系统**

**Shell代码规范（V1.0）**

编写单位/部门 中国科学院计算机网络信息中心

编 写

校 对

审 核

标 审

批 准

中国科学院计算机网络信息中心

2016年 12 月 15 日

**会 签**

|  |  |  |
| --- | --- | --- |
| 姓名 | 单位 | 时间 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**文档变更记录**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **修改**  **版本号** | **修改位置** | | **修改内容** | **修改人** | **修改日期** |
| **页** | **节** |
| V0.1 |  |  | 建立文档 |  | 2016.12.15 |
| V0.5 |  |  | 完成初稿 |  |  |
| V1.0 |  |  | 形成评审稿 |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**目 录**

[1 Shell文件和解释器调用 3](#_Toc469583810)

[1.1 文件扩展名 3](#_Toc469583811)

[1.2 SUID / SGID 3](#_Toc469583812)

[2 环境 3](#_Toc469583813)

[2.1 STDOUT vs STDERR 3](#_Toc469583814)

[3 注释 4](#_Toc469583815)

[3.1 文件头 4](#_Toc469583816)

[3.2 功能注释 4](#_Toc469583817)

[3.3 实现部分的注释 5](#_Toc469583818)

[3.4 TODO注释 5](#_Toc469583819)

[4 格式 5](#_Toc469583820)

[4.1 缩进 5](#_Toc469583821)

[4.2 行的长度和长字符串 6](#_Toc469583822)

[4.3 管道 6](#_Toc469583823)

[4.4 循环 7](#_Toc469583824)

[4.5 case语句 7](#_Toc469583825)

[4.6 变量扩展 8](#_Toc469583826)

[4.7 引用 10](#_Toc469583827)

[5 特性及错误 12](#_Toc469583828)

[5.1 命令替换 12](#_Toc469583829)

[5.2 test，[和[[ 12](#_Toc469583830)

[5.3 测试字符串 13](#_Toc469583831)

[5.4 文件名的通配符扩展 14](#_Toc469583832)

[5.5 Eval 14](#_Toc469583833)

[5.6 管道导向while循环 15](#_Toc469583834)

[6 命名转换 16](#_Toc469583835)

[6.1 函数名 16](#_Toc469583836)

[6.2 变量名 17](#_Toc469583837)

[6.3 常量和环境变量名 17](#_Toc469583838)

[6.4 源文件名 18](#_Toc469583839)

[6.5 只读变量 18](#_Toc469583840)

[6.6 使用本地变量 18](#_Toc469583841)

[6.7 函数位置 19](#_Toc469583842)

[6.8 主函数main 19](#_Toc469583843)

[7 调用命令 20](#_Toc469583844)

[7.1 检查返回值 20](#_Toc469583845)

[7.2 内建命令和外部命令 21](#_Toc469583846)

# 

# Shell文件和解释器调用

## 文件扩展名

Tip: 可执行文件应该没有扩展名（强烈建议）或者使用.sh扩展名。库文件必须使用.sh作为扩展名，而且应该是不可执行的。

当执行一个程序时，并不需要知道它是用什么语言编写的。而且shell脚本也不要求有扩展名。所以我们更喜欢可执行文件没有扩展名。

然而，对于库文件，知道其用什么语言编写的是很重要的，有时候会需要使用不同语言编写的相似的库文件。使用.sh这样特定语言后缀作为扩展名，就使得用不同语言编写的具有相同功能的库文件可以采用一样的名称。

## SUID / SGID

Tip: SUID(Set User ID)和SGID(Set Group ID)在shell脚本中是被禁止的。

shell存在太多的安全问题，以致于如果允许SUID/SGID会使得shell几乎不可能足够安全。虽然bash使得运行SUID非常困难，但在某些平台上仍然有可能运行，这就是为什么我们明确提出要禁止它。

如果你需要较高权限的访问请使用 sudo 。

# 环境

## STDOUT vs STDERR

Tip: 所有的错误信息都应该被导向STDERR。

这使得从实际问题中分离出正常状态变得更容易。

推荐使用类似如下函数，将错误信息和其他状态信息一起打印出来。

err() {

echo "[$(date +'%Y-%m-%dT%H:%M:%S%z')]: $@" >&2

}

if ! do\_something; then

err "Unable to do\_something"

exit "${E\_DID\_NOTHING}"

fi

# 注释

## 文件头

Tip: 每个文件的开头是其文件内容的描述。

每个文件必须包含一个顶层注释，对其内容进行简要概述。版权声明和作者信息是可选的。

例如：

#!/bin/bash

#

# Perform hot backups of Oracle databases.

## 功能注释

Tip: 任何不是既明显又短的函数都必须被注释。任何库函数无论其长短和复杂性都必须被注释。

其他人通过阅读注释（和帮助信息，如果有的话）就能够学会如何使用你的程序或库函数，而不需要阅读代码。

所有的函数注释应该包含：

* 函数的描述
* 全局变量的使用和修改
* 使用的参数说明
* 返回值，而不是上一条命令运行后默认的退出状态

例如：

#!/bin/bash

#

# Perform hot backups of Oracle databases.

export PATH='/usr/xpg4/bin:/usr/bin:/opt/csw/bin:/opt/goog/bin'

#######################################

# Cleanup files from the backup dir

# Globals:

# BACKUP\_DIR

# ORACLE\_SID

# Arguments:

# None

# Returns:

# None

#######################################

cleanup() {

...

}

## 实现部分的注释

Tip: 注释你代码中含有技巧、不明显、有趣的或者重要的部分。

这部分遵循谷歌代码注释的通用做法。不要注释所有代码。如果有一个复杂的算法或者你正在做一些与众不同的，放一个简单的注释。

## TODO注释

Tip: 使用TODO注释临时的、短期解决方案的、或者足够好但不够完美的代码。

这与C++指南中的约定相一致。

TODOs应该包含全部大写的字符串TODO，接着是括号中你的用户名。冒号是可选的。最好在TODO条目之后加上 bug或者ticket 的序号。

例如：

# TODO(mrmonkey): Handle the unlikely edge cases (bug ####)

# 格式

## 缩进

Tip: 缩进两个空格，没有制表符。

在代码块之间请使用空行以提升可读性。缩进为两个空格。无论你做什么，请不要使用制表符。对于已有文件，保持已有的缩进格式。

## 行的长度和长字符串

Tip: 行的最大长度为80个字符。

如果你必须写长度超过80个字符的字符串，如果可能的话，尽量使用here document或者嵌入的换行符。长度超过80个字符的文字串且不能被合理地分割，这是正常的。但强烈建议找到一个方法使其变短。

# DO use 'here document's

cat <<END;

I am an exceptionally long

string.

END

# Embedded newlines are ok too

long\_string="I am an exceptionally

long string."

## 管道

Tip: 如果一行容不下整个管道操作，那么请将整个管道操作分割成每行一个管段。

如果一行容得下整个管道操作，那么请将整个管道操作写在同一行。

否则，应该将整个管道操作分割成每行一个管段，管道操作的下一部分应该将管道符放在新行并且缩进2个空格。这适用于使用管道符’|’的合并命令链以及使用’||’和’&&’的逻辑运算链。

# All fits on one line

command1 | command2

# Long commands

command1 \

| command2 \

| command3 \

| command4

## 循环

Tip: 请将 ; do , ; then 和 while , for , if 放在同一行。

shell中的循坏略有不同，但是我们遵循跟声明函数时的大括号相同的原则。也就是说， ; do , ; then 应该和 if/for/while 放在同一行。 else 应该单独一行，结束语句应该单独一行并且跟开始语句垂直对齐。

例如：

for dir in ${dirs\_to\_cleanup}; do

if [[ -d "${dir}/${ORACLE\_SID}" ]]; then

log\_date "Cleaning up old files in ${dir}/${ORACLE\_SID}"

rm "${dir}/${ORACLE\_SID}/"\*

if [[ "$?" -ne 0 ]]; then

error\_message

fi

else

mkdir -p "${dir}/${ORACLE\_SID}"

if [[ "$?" -ne 0 ]]; then

error\_message

fi

fi

done

## case语句

Tip: 通过2个空格缩进可选项。

在同一行可选项的模式右圆括号之后和结束符 ;; 之前各需要一个空格。

长可选项或者多命令可选项应该被拆分成多行，模式、操作和结束符 ;; 在不同的行。

匹配表达式比 case 和 esac 缩进一级。多行操作要再缩进一级。一般情况下，不需要引用匹配表达式。模式表达式前面不应该出现左括号。避免使用 ;& 和 ;;& 符号。

case "${expression}" in

a)

variable="..."

some\_command "${variable}" "${other\_expr}" ...

;;

absolute)

actions="relative"

another\_command "${actions}" "${other\_expr}" ...

;;

\*)

error "Unexpected expression '${expression}'"

;;

esac

只要整个表达式可读，简单的命令可以跟模式和 ;; 写在同一行。这通常适用于单字母选项的处理。当单行容不下操作时，请将模式单独放一行，然后是操作，最后结束符 ;; 也单独一行。当操作在同一行时，模式的右括号之后和结束符 ;; 之前请使用一个空格分隔。

verbose='false'

aflag=''

bflag=''

files=''

while getopts 'abf:v' flag; do

case "${flag}" in

a) aflag='true' ;;

b) bflag='true' ;;

f) files="${OPTARG}" ;;

v) verbose='true' ;;

\*) error "Unexpected option ${flag}" ;;

esac

done

## 变量扩展

Tip: 按优先级顺序：保持跟你所发现的一致；引用你的变量；推荐用 ${var} 而不是 $var ，详细解释如下。

这些仅仅是指南，因为作为强制规定似乎饱受争议。

以下按照优先顺序列出。

1. 与现存代码中你所发现的保持一致。
2. 引用变量参阅下面一节，引用。
3. 除非绝对必要或者为了避免深深的困惑，否则不要用大括号将单个字符的shell特殊变量或定位变量括起来。推荐将其他所有变量用大括号括起来。

# Section of recommended cases.

# Preferred style for 'special' variables:

echo "Positional: $1" "$5" "$3"

echo "Specials: !=$!, -=$-, \_=$\_. ?=$?, #=$# \*=$\* @=$@ \$=$$ ..."

# Braces necessary:

echo "many parameters: ${10}"

# Braces avoiding confusion:

# Output is "a0b0c0"

set -- a b c

echo "${1}0${2}0${3}0"

# Preferred style for other variables:

echo "PATH=${PATH}, PWD=${PWD}, mine=${some\_var}"

while read f; do

echo "file=${f}"

done < <(ls -l /tmp)

# Section of discouraged cases

# Unquoted vars, unbraced vars, brace-quoted single letter

# shell specials.

echo a=$avar "b=$bvar" "PID=${$}" "${1}"

# Confusing use: this is expanded as "${1}0${2}0${3}0",

# not "${10}${20}${30}

set -- a b c

echo "$10$20$30"

## 引用

Tip:

* 除非需要小心不带引用的扩展，否则总是引用包含变量、命令替换符、空格或shell元字符的字符串。
* 推荐引用是单词的字符串（而不是命令选项或者路径名）。
* 千万不要引用整数。
* 注意 [[ 中模式匹配的引用规则。
* 请使用 $@ 除非你有特殊原因需要使用 $\* 。

# 'Single' quotes indicate that no substitution is desired.

# "Double" quotes indicate that substitution is required/tolerated.

# Simple examples

# "quote command substitutions"

flag="$(some\_command and its args "$@" 'quoted separately')"

# "quote variables"

echo "${flag}"

# "never quote literal integers"

value=32

# "quote command substitutions", even when you expect integers

number="$(generate\_number)"

# "prefer quoting words", not compulsory

readonly USE\_INTEGER='true'

# "quote shell meta characters"

echo 'Hello stranger, and well met. Earn lots of $$$'

echo "Process $$: Done making \$\$\$."

# "command options or path names"

# ($1 is assumed to contain a value here)

grep -li Hugo /dev/null "$1"

# Less simple examples

# "quote variables, unless proven false": ccs might be empty

git send-email --to "${reviewers}" ${ccs:+"--cc" "${ccs}"}

# Positional parameter precautions: $1 might be unset

# Single quotes leave regex as-is.

grep -cP '([Ss]pecial|\|?characters\*)$' ${1:+"$1"}

# For passing on arguments,

# "$@" is right almost everytime, and

# $\* is wrong almost everytime:

#

# \* $\* and $@ will split on spaces, clobbering up arguments

# that contain spaces and dropping empty strings;

# \* "$@" will retain arguments as-is, so no args

# provided will result in no args being passed on;

# This is in most cases what you want to use for passing

# on arguments.

# \* "$\*" expands to one argument, with all args joined

# by (usually) spaces,

# so no args provided will result in one empty string

# being passed on.

# (Consult 'man bash' for the nit-grits ;-)

set -- 1 "2 two" "3 three tres"; echo $# ; set -- "$\*"; echo "$#, $@")

set -- 1 "2 two" "3 three tres"; echo $# ; set -- "$@"; echo "$#, $@")

# 特性及错误

## 命令替换

Tip: 使用 $(command) 而不是反引号。

嵌套的反引号要求用反斜杠转义内部的反引号。而 $(command) 形式嵌套时不需要改变，而且更易于阅读。

例如：

# This is preferred:

var="$(command "$(command1)")"

# This is not:

var="`command \`command1\``"

## test，[和[[

Tip: 推荐使用 [[ ... ]] ，而不是 [ , test , 和 /usr/bin/ [ 。

因为在 [[ 和 ]] 之间不会有路径名称扩展或单词分割发生，所以使用 [[ ... ]] 能够减少错误。而且 [[ ... ]] 允许正则表达式匹配，而 [ ... ] 不允许。

# This ensures the string on the left is made up of characters in the

# alnum character class followed by the string name.

# Note that the RHS should not be quoted here.

# For the gory details, see

# E14 at http://tiswww.case.edu/php/chet/bash/FAQ

if [[ "filename" =~ ^[[:alnum:]]+name ]]; then

echo "Match"

fi

# This matches the exact pattern "f\*" (Does not match in this case)

if [[ "filename" == "f\*" ]]; then

echo "Match"

fi

# This gives a "too many arguments" error as f\* is expanded to the

# contents of the current directory

if [ "filename" == f\* ]; then

echo "Match"

fi

## 测试字符串

Tip: 尽可能使用引用，而不是过滤字符串。

Bash足以在测试中处理空字符串。所以，请使用空（非空）字符串测试，而不是过滤字符，使得代码更易于阅读。

# Do this:

if [[ "${my\_var}" = "some\_string" ]]; then

do\_something

fi

# -z (string length is zero) and -n (string length is not zero) are

# preferred over testing for an empty string

if [[ -z "${my\_var}" ]]; then

do\_something

fi

# This is OK (ensure quotes on the empty side), but not preferred:

if [[ "${my\_var}" = "" ]]; then

do\_something

fi

# Not this:

if [[ "${my\_var}X" = "some\_stringX" ]]; then

do\_something

fi

为了避免对你测试的目的产生困惑，请明确使用`-z`或者`-n`

# Use this

if [[ -n "${my\_var}" ]]; then

do\_something

fi

# Instead of this as errors can occur if ${my\_var} expands to a test

# flag

if [[ "${my\_var}" ]]; then

do\_something

fi

## 文件名的通配符扩展

Tip: 当进行文件名的通配符扩展时，请使用明确的路径。

因为文件名可能以 - 开头，所以使用扩展通配符 ./\* 比 \* 来得安全得多。

# Here's the contents of the directory:

# -f -r somedir somefile

# This deletes almost everything in the directory by force

psa@bilby$ rm -v \*

removed directory: `somedir'

removed `somefile'

# As opposed to:

psa@bilby$ rm -v ./\*

removed `./-f'

removed `./-r'

rm: cannot remove `./somedir': Is a directory

removed `./somefile'

## Eval

Tip: 应该避免使用eval。

当用于给变量赋值时，Eval解析输入，并且能够设置变量，但无法检查这些变量是什么。

# What does this set?

# Did it succeed? In part or whole?

eval $(set\_my\_variables)

# What happens if one of the returned values has a space in it?

variable="$(eval some\_function)"

## 管道导向while循环

Tip: 请使用过程替换或者for循环，而不是管道导向while循环。在while循环中被修改的变量是不能传递给父shell的，因为循环命令是在一个子shell中运行的。

管道导向while循环中的隐式子shell使得追踪bug变得很困难。

last\_line='NULL'

your\_command | while read line; do

last\_line="${line}"

done

# This will output 'NULL'

echo "${last\_line}"

如果你确定输入中不包含空格或者特殊符号（通常意味着不是用户输入的），那么可以使用一个for循环。

total=0

# Only do this if there are no spaces in return values.

for value in $(command); do

total+="${value}"

done

使用过程替换允许重定向输出，但是请将命令放入一个显式的子shell中，而不是bash为while循环创建的隐式子shell。

total=0

last\_file=

while read count filename; do

total+="${count}"

last\_file="${filename}"

done < <(your\_command | uniq -c)

# This will output the second field of the last line of output from

# the command.

echo "Total = ${total}"

echo "Last one = ${last\_file}"

当不需要传递复杂的结果给父shell时可以使用while循环。这通常需要一些更复杂的“解析”。请注意简单的例子使用如awk这类工具可能更容易完成。当你特别不希望改变父shell的范围变量时这可能也是有用的。

# Trivial implementation of awk expression:

# awk '$3 == "nfs" { print $2 " maps to " $1 }' /proc/mounts

cat /proc/mounts | while read src dest type opts rest; do

if [[ ${type} == "nfs" ]]; then

echo "NFS ${dest} maps to ${src}"

fi

done

# 命名转换

## 函数名

Tip: 使用小写字母，并用下划线分隔单词。使用双冒号 :: 分隔库。函数名之后必须有圆括号。关键词 function 是可选的，但必须在一个项目中保持一致。

如果你正在写单个函数，请用小写字母来命名，并用下划线分隔单词。如果你正在写一个包，使用双冒号 :: 来分隔包名。大括号必须和函数名位于同一行（就像在Google的其他语言一样），并且函数名和圆括号之间没有空格。

# Single function

my\_func() {

...

}

# Part of a package

mypackage::my\_func() {

...

}

当函数名后存在 () 时，关键词 function 是多余的。但是其促进了函数的快速辨识。

## 变量名

Tip: 如函数名。

循环的变量名应该和循环的任何变量同样命名。

for zone in ${zones}; do

something\_with "${zone}"

done

## 常量和环境变量名

Tip: 全部大写，用下划线分隔，声明在文件的顶部。

常量和任何导出到环境中的都应该大写。

# Constant

readonly PATH\_TO\_FILES='/some/path'

# Both constant and environment

declare -xr ORACLE\_SID='PROD'

第一次设置时有一些就变成了常量（例如，通过getopts）。因此，可以在getopts中或基于条件来设定常量，但之后应该立即设置其为只读。值得注意的是，在函数中 declare 不会对全局变量进行操作。所以推荐使用 readonly 和 export 来代替。

VERBOSE='false'

while getopts 'v' flag; do

case "${flag}" in

v) VERBOSE='true' ;;

esac

done

readonly VERBOSE

## 源文件名

Tip: 小写，如果需要的话使用下划线分隔单词。

这是为了和在Google中的其他代码风格保持一致： maketemplate 或者 make\_template ，而不是 make-template 。

## 只读变量

Tip: 使用 readonly 或者 declare -r 来确保变量只读。

因为全局变量在shell中广泛使用，所以在使用它们的过程中捕获错误是很重要的。当你声明了一个变量，希望其只读，那么请明确指出。

zip\_version="$(dpkg --status zip | grep Version: | cut -d ' ' -f 2)"

if [[ -z "${zip\_version}" ]]; then

error\_message

else

readonly zip\_version

fi

## 使用本地变量

Tip: 使用 local 声明特定功能的变量。声明和赋值应该在不同行。

使用 local 来声明局部变量以确保其只在函数内部和子函数中可见。这避免了污染全局命名空间和不经意间设置可能具有函数之外重要性的变量。

当赋值的值由命令替换提供时，声明和赋值必须分开。因为内建的 local 不会从命令替换中传递退出码。

my\_func2() {

local name="$1"

# Separate lines for declaration and assignment:

local my\_var

my\_var="$(my\_func)" || return

# DO NOT do this: $? contains the exit code of 'local', not my\_func

local my\_var="$(my\_func)"

[[ $? -eq 0 ]] || return

...

}

## 函数位置

Tip: 将文件中所有的函数一起放在常量下面。不要在函数之间隐藏可执行代码。

如果你有函数，请将他们一起放在文件头部。只有includes， set 声明和常量设置可能在函数声明之前完成。不要在函数之间隐藏可执行代码。如果那样做，会使得代码在调试时难以跟踪并出现意想不到的讨厌结果。

## 主函数main

Tip: 对于包含至少一个其他函数的足够长的脚本，需要称为 main 的函数。

为了方便查找程序的开始，将主程序放入一个称为 main 的函数，作为最下面的函数。这使其和代码库的其余部分保持一致性，同时允许你定义更多变量为局部变量（如果主代码不是一个函数就不能这么做）。文件中最后的非注释行应该是对 main 函数的调用。

main "$@"

显然，对于仅仅是线性流的短脚本， main 是矫枉过正，因此是不需要的。

# 调用命令

## 检查返回值

Tip: 总是检查返回值，并给出信息返回值。

对于非管道命令，使用 $? 或直接通过一个 if 语句来检查以保持其简洁。

例如：

if ! mv "${file\_list}" "${dest\_dir}/" ; then

echo "Unable to move ${file\_list} to ${dest\_dir}" >&2

exit "${E\_BAD\_MOVE}"

fi

# Or

mv "${file\_list}" "${dest\_dir}/"

if [[ "$?" -ne 0 ]]; then

echo "Unable to move ${file\_list} to ${dest\_dir}" >&2

exit "${E\_BAD\_MOVE}"

fi

Bash也有 PIPESTATUS 变量，允许检查从管道所有部分返回的代码。如果仅仅需要检查整个管道是成功还是失败，以下的方法是可以接受的：

tar -cf - ./\* | ( cd "${dir}" && tar -xf - )

if [[ "${PIPESTATUS[0]}" -ne 0 || "${PIPESTATUS[1]}" -ne 0 ]]; then

echo "Unable to tar files to ${dir}" >&2

fi

可是，只要你运行任何其他命令， PIPESTATUS 将会被覆盖。如果你需要基于管道中发生的错误执行不同的操作，那么你需要在运行命令后立即将 PIPESTATUS 赋值给另一个变量（别忘了 [ 是一个会将 PIPESTATUS 擦除的命令）。

tar -cf - ./\* | ( cd "${DIR}" && tar -xf - )

return\_codes=(${PIPESTATUS[\*]})

if [[ "${return\_codes[0]}" -ne 0 ]]; then

do\_something

fi

if [[ "${return\_codes[1]}" -ne 0 ]]; then

do\_something\_else

fi

## 内建命令和外部命令

Tip: 可以在调用shell内建命令和调用另外的程序之间选择，请选择内建命令。

我们更喜欢使用内建命令，如在 bash(1) 中参数扩展函数。因为它更强健和便携（尤其是跟像 sed 这样的命令比较）

例如：

# Prefer this:

addition=$((${X} + ${Y}))

substitution="${string/#foo/bar}"

# Instead of this:

addition="$(expr ${X} + ${Y})"

substitution="$(echo "${string}" | sed -e 's/^foo/bar/')"