shell 的使用情况

1. 如果大多数情况下调用其他实用程序，并且执行的操作相对较少，那么对于任务而言，shell是可接受的选择。
2. 如果性能很重要，可以使用shell以外的东西。
3. 如果你发现你需要使用数组来分配任何东西${PIPESTATUS}，你应该使Python。
4. 如果您正在编写超过100行的脚本，则应该用Python代替。请记住，脚本的增长。尽早用另一种语言重写脚本，以避免在以后的日子里耗费时间重写。

文件扩展名

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。

库文件，知道它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

在 shell 上还有很多安全问题，所以SUID/SGID 在 shell 上是禁止的。如果要使用更高的权限，必须使用 sudo。

STDOUT vs STDERR

所有的错误信息应该去STDERR。

这使得更容易将正常状态与实际问题分开。

推荐使用打印出错信息和其他状态信息的功能。

缩进

按照 2 个空格来缩进。

管道

如果一行不能容纳多个管道，请将多个管道拆分成一行一个。

循环

请将 ; do、; then 和 while、for 或者 if 放在同一行。

shell中的循环有点不同，但是在声明函数时我们遵循与花括号相同的原则。那就是：应该; then 和; doif / for / while在同一行。 else应该在自己的路线上，闭幕声明应该在自己的行与开幕声明垂直对齐。

例子：

```
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
```

行长度和长字符串

一行的长度最多是八十个字符.如果你必须要写一个长于该字符长度的字符串，你应该尽量嵌入一个新行，如果有一个文字字符串长度超过了该字符长度，并不能合理的分割文字字符串，这时候便强烈推荐找到一种办法让它更短一点。

TODO评论

使用TODO注释代码是暂时的，短期的解决方案，或者是足够好的但不完美的。这符合C ++指南中的约定。

TODO应在所有大写字母中包含字符串TODO，然后在括号中包含您的用户名。冒号是可选的。最好在TODO项目旁边放一个错误/票号。

例子：

＃TODO（mrmonkey）：处理不太可能的边缘情况（bug ####）

变量扩展

按优先顺序排列，引用你的变量。

命令替换

使用$(command)而不是反引号。嵌套反引号需要逃避内部的\。该$(command)嵌套，并更容易阅读格式时不会改变。

测试字符串

尽可能使用引号而不是填充字符

文件的通配符扩展

进行文件名的通配符扩展时使用显式路径。作为文件名可以用一个开始-，这是一个很大安全，扩大通配符./*代替*。

例子：

```
# 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

eval 命令应该被禁止执行。

```
# 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)"
```

函数名

使用小写字母，并用下划线分隔单词。
使用双冒号 :: 分隔库。
函数名后必须有圆括号。

函数位置

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 语句和设置常数可能在函数定义前完成。
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

对于一个很长的脚本而言，我们暂时需要一个 main 函数去调用其它的函数。
为了找到程序的起始位置，我们把主程序放到一个叫 main的函数中，并且放在其它函数的下面， 这当然是为了提供一致性。

```
main"$@"
```

源文件名

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

使用 readonly 或者 declare -r 来确保变量是只读的。
为全局变量广泛使用，所以在使用它们的过程中捕获错误是很重要的。当你声明了一个希望其只读的变量，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
```

使用局部变量

声明函数特定的变量。宣言和任务应该在不同的方面。

确保局部变量只能local在声明它们的时候才能在函数及其子元素中看到。这避免了污染全局名称空间，并且无意中设置了可能在函数外部具有意义的变量。

当分配值由命令替换提供时，声明和分配必须是单独的语句；因为“本地”内建并不会传播来自命令替换的退出代码。

例子：

```
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

  ...
}
```

检查返回值

始终检查返回值并提供丰富的返回值

例子：

```
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
```