shell 的使用情况：

shell 是用于编写各种实用程序的脚本，它将复杂的项目分解为简单的子任务，将组件和实用程序链接在一起。

使用 shell 脚本，有一些准则如下所示：

使用 shell 脚本：

调用其它工具并且做一些相对很小数据量的操作，那么使用 shell 来完成任务是可以接受的

不使用 shell 脚本的情况：

1：当需要高性能的程序时

2：复杂的应用程序的编写

3：操作大量的本地文件

4：需要跨平台的可移植性

5：复杂的数学运算程序以及复杂的排序任务

6：需要数据结构，如链表、树

7：需要生成或者操作图形或 GUI

8：对于安全有着较高要求的话

文件扩展名(File Extensions)：

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

因为在 shell 上还有一些安全问题，因此 SUID/SGID 在 shell 上时禁止的。

如果要使用更高的权限，使用 sudo。

标准输出对比错误输出(STDOUT vs STDERR)

所有的错误信息应该输入到标准错误输出（STDERR）中。

这使得从实际问题中分离出正常状态变得更容易。

下面这个函数是用于打印出错误信息以及其他状态信息的功能，值得推荐。

```
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(Comments)

文件头(File Header)：

每个文件的开头处添加一段描述内容。

每个文件都必须要有一个顶层注释，其中包含了内容的简要概述。版权声明和作者信息是可选的。

例如：

```
#!/bin/bash
#
# Perform hot backups of Oracle databases.
```

函数注释

任何不明显和短的函数都必须注释。无论长度或复杂度如何，库中的任何函数都必须进行注释。

应该可以让其他人学习如何使用你的程序，或者通过阅读注释（如果提供了自助功能）而不用阅读代码来使用你的程序库中的函数。

所有功能注释应该包含：

- 函数的描述信息
- 使用的和修改的全局变量
- 参数信息

返回值而不是最后一条命令的缺省退出状态码

```
#!/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() {
   ...
}
```

代码实现过程中的注释：
对代码中含有技巧的，不明显得或者是一些重要的部分添加注释。
遵循 Google 的通用编码注释的做法。不要所有代码都加注释。如果有一个复杂的算法，或者是你在做一个与众不同的功能，在这些地方放置一个简单的注释即可。
TODO 的注释(TODO Comments)
在临时的、短期解决方案的、或者足够好但不够完美的代码处添加 TODO 注释。

这与 C++ 指南中的约定相一致。
所有的 TODO 类别的注释，应该包含一个全部大写的字符串 TODO，后面用括号包含您的用户名。冒号是可选的。这里最好把 bug 号，或者是 ticket 号放在 TODO 注释后面。
例如：

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

格式(Formatting)
虽然你需要遵循你正在修改的文件的风格，但是新的代码必须要遵循下面的风格。

缩进(Indentation)
按照 2 个空格来缩进，不使用 tab 来缩进。
在两个语句块中间使用空白行来提高可读性。缩进使用两个空格。无论你做什么，不要使用制表符（tab）。对于现有的文件，保留现有使用的缩进，

行长度和长字符串(Line Length and Long Strings)
一行的长度最多是 80 个字符.
如果你必须要写一个长于 80 个字符的字符串，如果可能的话，你应该尽量使用 here document 或者嵌入一个新行，如果有一个文字字符串长度超过了 80 个字符，并且不能合理的分割文字字符串，但是强烈推荐你找到一种办法让它更短一点。

多个管道（Pipelines）
如果一行不能容纳多个管道操作，那么请将多个管道拆分成一行一个。
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，那么应该分割成每行一个管道，新的一行应该缩进 2 个空格。这条规则适用于那些通过使用”|”或者是一个逻辑运算符”||”和”&&”等组合起来的链式命令。

```
# All fits on one line
command1 | command2

# Long commands
command1 \
    | command2 \
    | command3 \
    | command4
```

循环(Loops)

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

Shell 中的循环略有不同，但是我们遵循像声明函数时用大括号同样的原则，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(Case statement)

- 缩进可用 2 个空格替代。
- 可用一行替代的，需要在右括号后面和 ;; 号前面添加一个空格。
- 对于长的，有多个命令的，应该分割成多行，其中匹配项，对于匹配项的处理以及 ;; 号各自在单独的行。

case 和 esac 中匹配项的表达式应该都在同一个缩进级别，匹配项的（多行）处理也应该在另一个缩进级别。通常来说，没有必要给匹配项的表达式添加引号。匹配项的表达式不应该在前面加一个左括号，避免使用 ;& 和 ;;s& 等符号.

```
case "${expression}" in
    a)
        variable="..."
        some_command "${variable}" "${other_expr}" ...
        ;;
    absolute)
        actions="relative"
        another_command "${actions}" "${other_expr}" ...
        ;;
    *)
        error "Unexpected expression '${expression}'"
        ;;
esac
```

对于一些简单的匹配项处理操作，可以和匹配项表达式以及 ;; 号在同一行,只要表达式仍然可读。这通常适合单字符的选项处理，当匹配项处理操作不能满足单行的情况下，可以将匹配项表达式单独放在一行，匹配项处理操作和 ;; 放在同一行，当匹配项操作和匹配项表达式以及 ;; 放在同一行的时候在匹配项表达式右括号后面以及 ;; 前面放置一个空格。

```
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(Variable expansion)

按优先级顺序：保持跟你所发现的一致；把你的变量用括号印起来；推荐用 "${var}" 而不是 "$var"，详细解释如下。
这些仅仅是指南，因为按标题作为强制的规定饱受争议。

以下按照优先顺序列出。
与现存代码中你所发现的保持一致。
把变量用（大）扩号引起来，参阅下面一节：引用。
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 特殊变量或位置参数括起来。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(Quoting)

除非需要小心不带引用的扩展，否则总是将包含变量、命令替换符、空格或 Shell 元字符的字符串引起来。

优先引用是单词的字符串（而不是命令选项或者路径名）。

不要对整数进行引用。

千万小心 [[中模式匹配的引用规则。

特征和错误(Features and Bugs)

命令替换(Command Substitution)

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 [[(Test, [and [[)
优先使用 [[...]]，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(Testing Strings)

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(Wildcard Expansion of Filenames)

当做文件名通配符扩展的时候，使用显式路径。

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

```
# Here's the contents of the directory:
#  当前目录下又-f -r somedir somefile 等文件和目录
# -f    -r    somedir    somefile

#  使用 rm -v *将会扩展成 rm -v -r -f somedir simefile，这将导致删除当前目录所有的文件和目录
# This deletes almost everything in the directory by force
psa@bilby$ rm -v *
removed directory: `somedir'
removed `somefile'

#相反如果你使用./*则不会，因为-r -f 就不会变成 rm 的参数了
# 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。

eval 用于给变量赋值的时候，可以设置变量，但是不能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(Pipes to While)

优先使用过程替换或者 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'
```

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
```

命名约定(Naming Conventions)

函数名(Function Names)

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(Variable Names)

如函数名。

循环的变量名应该和要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(Constants and Environment Variable Names)

要大写、用下划线分割、声明在文件的开头。

常量和任何导出到环境的变量都应该大写。

```
# Constant
readonly PATH_TO_FILES='/some/path'

# Both constant and environment
# declare -r 设置只读变量，-x 设置为环境变量
declare -xr ORACLE_SID='PROD'
```

有些第一次设置时(例如使用 getopts 情况下)就变成了常量。也就是说，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。需要注意的是，declare 不能在函数内部操作全局变量，所以这时推荐使用 readonly 和 export 来代替。

```
VERBOSE='false'
while getopts 'v' flag; do
    case "${flag}" in
        v) VERBOSE='true' ;;
    esac
done
```

源文件名(Source Filenames)

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(Read-ony Variables)

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
```

使用本地变量(Use Local Variables)

使用 local 声明函数内部变量。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意义的变量。

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(Function Location)

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(main)

对于足够长的脚本来说，至少需要一个名为 main 的函数来调用其它的函数。

为了便于找到程序的起始位置,把主程序放在一个叫 main 的函数中,放在其它函数的下面，为了提供一致性你应该定义更多的变量为本地变量(如果主程序不是一个程序，那么不能这么做)，文件中最后一句非注释行应该是一个 main 函数的调用。

```
main "$@"
```

调用命令(Calling Commands)
检查返回值(Checking Return Values)

总是应该检查返回值，给出返回值相关的信息。
对于一个未使用管道的命令，可以使用 $? 或者直接指向 if 语句来检查其返回值
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
```

个

管道执行成功与否。下面的例子是被接受的。

```
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
```

然后当你使用任何其它命令的时候 PIPESTATUS 将会被覆盖，如果你需要根据管道发生错误的地方来进行不同的操作，那么你将需要在运行完管道命令后立即将 PIPESTATUS 的值赋给另外一个变量(不要忘了[这个符号也是一个命令，将会把 PIPESTATUS 的值给覆盖掉．)

```
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
```

内置命令对比外部命令(Builtin Commands vs. External Commands)

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/')"
```