中软国际公司内部技术规范

Shell语言编程规范

中软国际科技服务有限公司

修订声明

参考：业界 Shell 编码规范指南，华为 Shell 编码规范等。

日期	修订版本	修订描述	作者
2015-03-12	1.0	整理创建初稿	徐庆阳
2015-03-27	1.1	合入 VGS 交付部自动安装开发组的 SHELL 编程规范文档	顾行举
2017-08-02	1.2	统一调整格式	陈丽佳

目录

前言：

建立 Linux/Unix 下脚本语言编程规范的主要目的是：

- 统一编程风格
- 提高代码的可阅读性
- 减少错误产生
- 减少性能漏洞
- 提高代码可靠性
- 减少错误的编码设计
- 作为代码检查的依据
- 建立可维护的 shell 语言编程规范

本规范采用以下的术语描述：

- ★ 规则：编程时强制必须遵守的原则。
- ★ 建议：编程时必须加以考虑的原则。
- ★ 说明：对此规则或建议进行必要的解释、说明。
- ★ 示例：对此规则或建议给出脚本例子。

1. 排版

1.1 规则

规则 1 程序块要采用缩进风格编写，缩进的空格数为 4 个。
说明：缩进使程序更易阅读，使用空格缩进可以适应不同操作系统与不同开发工具。

规则 2 在函数体的开始,分界符（如大括号‘{’和‘}’）应各独占一行，同时与引用它们的语句左对齐。

示例：

```
function start_vcs()
{
    /etc/init.d/llt start
    /etc/init.d/gab start
    /opt/VRTS/bin/hastart
}
```

规则 3 较长的语句、表达式或参数（>80 字符）要分成多行书写，长表达式要在低优先级操作符处划分新行，划分行通过符号‘\’标识，操作符放在新行之首，划分出的新行要进行适当的缩进，使排版整齐，语句可读。

示例：

```
/sbin/modprobe --version 2>&1 | grep -q module-init-tools \
    && NEW_MODUTILS=1 \
    || NEW_MODUTILS=0
```

规则 4 不允许把多个短语句写在一行中，即一行只写一条语句。
说明：阅读代码更加清晰。

示例：
不符合规范：

```
FILENAME=/etc/dd.cfg; COUNTER=5;
```

符合规范：

```
FILENAME=/etc/dd.cfg
COUNTER=5
```

规则 5 逻辑操作符”&&”、”||”、”|”等的前后要加空格，管道、重定向操作符”|”、”>”、”>>”等的前后要加空格。

。

示例：

```
[ -r $inputfile ] && echo “${inputfile} is not exist.”


echo “Test String” >> /dev/null


ps -ef | grep “Start”
```

规则 6 相对独立的程序块之间、变量说明之后必须加空行。
说明：阅读代码更加清晰

示例：

```
if [ $? -eq 0 ] ;
then
    do_start
fi
```

此处是空行

```
return 0
```

规则 7 一个方法只能完成一个功能，并且方法不能太长，如果太长(大于 100 行)，应该细分功能，由多个方法来完成。
说明：太长的方法不便于阅读，可以考虑用简化逻辑的方法或者分成若干子功能方法来完成。

1.2 建议

建议 1 为保持统一风格，if-then-else 语句的书写。
说明：语句不要全部写在一行。建议使用示例书写格式。

示例：

```
if [ $some-thing-happen ]
then
    do-something
    do-others
elif [ $some-thing-else ]
    do-what-you-want
else
    do-what-you-hate
fi
```

建议 2 为保持统一风格，case 语句的书写。

示例：

```
case ${value} in
value1)
    do-value1
    ;;
value2)
do-value2
    ;;
…
*)
    do-others
    ;;
esac
```

建议 3 为保持统一风格，for 循环的书写。

示例：

```
for variable in variable_list
do
    do-something
```

```
done
```

建议 4 为保持统一风格，until 循环的书写。

示例：

```
IS_ROOT=`who | grep root`
until [ "${IS_ROOT}" ]
do
    sleep 5
done
echo "Watch it. Root in"
```

建议 5 为保持统一风格，while 循环的书写。

示例：

```
let COUNTER=0
while [ ${COUNTER} -lt 5 ]
do
    #add one to the counter
    COUNTER=(expr ${COUNTER} + 1)
    echo ${COUNTER}
done
```

建议 6 使用 break、continue 控制循环。

2. 注释

2.1 规则

规则 1 版本代码中注释必须以符号'#'开始。

规则 2 脚本的标示（如#!/usr/bin/ksh）永远在脚本第一行。

规则 3 脚本文件开头必须编写必要版权信息、作者、脚本用途说明。
说明：
#!/bin/bash
Copyright (c) Huawei Technologies Co., Ltd. 2004-2011. All rights reserved. 版权信息声明
Author: xxx 作者及联系方式
#
script description 脚本用途描述
#

示例：
#!/bin/bash
Copyright (c) Huawei Technologies Co., Ltd. 2004-2011. All rights reserved.
Author: zhangsan 12345
#
System startup script for mysoftx daemon
#

规则 4 方法注释内容：列出方法的一句话功能简述、功能详细描述、输入参数、输出参数、返回值等，涉及到接口调用的方法，必须按照格式编写相关注释。
说明：
##
Function:〈方法调用名称〉
Description:〈功能描述〉
Parameter: [参数] [参数说明]
input:
[参数 1] [参数 1 说明]
#
output:
[参数 2] [参数 2 说明]
Return:[返回类型说明]
Since: [起始版本]
Others: [其他说明]
##
其中，Parameter 和 Return 没有时，直接写 N/A，Since 表示当前版本的方法，Others 可以增加一些额外的说明，如使用该方法的注意事项、修改记录等。

示例：

```
######################################################################
# Function: print_bignumber
# Description: compare two numbers and print the biger number.
# Parameter:
#       input:
#       $1 -- the first number
#       $2 -- the second number
#       output: N/A
# Return:
#       0 -- success
#       1 -- failure
# Since: ATAE USM V100R002C10
# Others:
######################################################################
```

规则 5 功能相对独立的程序块，应在程序块前加注释，说明其功能。

规则 6 对于某行注释，应在这行的前面或者右边加，不可注释在行下。行前的注释不应间隔行，行右注释应与代码间隔一定距离，保持美观。

规则 7 注释的内容应清楚、准确、有效，防止出现歧义，以描述清楚要注释的代码为准。

规则 8 编写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性。不再有用的注释要删除。

规则 9 避免在注释中使用缩写，特别是非常用缩写。

规则 10 对于所有变量、常量，如果其命名不是充分自注释的，在声明时都必须加以注释。注释应放在其上方相邻位置或右方。

规则 11 分支语句（条件分支、循环语句等）必须编写注释。
说明：这些语句往往是程序实现某一特定功能的关键，对于维护人员来说，良好的注释帮助更好的理解程序，有时甚至优于看设计文档。

规则 12 注释与所描述的内容进行同样的缩排。

规则 13 将注释与其上面的代码用空行隔开。

2.2 建议

建议 1 注释开头请用空格分开，保持清新风格。

```
# start tomcat watcher
```

建议 2 注释语言请使用英语。
说明：由于 shell 脚本大多在 Linux 环境下编辑和查看，对中文的支持非常不好，因此默认请使用英文，同时由于开发人员的英语水平存在差异，请使用简短英语准确描述。

建议 3 通过对函数或过程、变量、结构等正确的命名以及合理地组织代码的结构，使代码成为自注释的。
说明：清晰准确的函数、变量等的命名，可增加代码可读性，并减少不必要的注释。

建议 4 在代码的功能、意图层次上进行注释，提供有用、额外的信息。
说明：注释的目的是解释代码的目的、功能和采用的方法，提供代码以外的信息，帮助读者理解代码，防止没必要的重复注释信息。

3. 命名

3.1 规则

规则 1 标识符的命名要清晰、明了，有明确含义，同时使用完整的单词或大家基本可以理解的缩写，避免使人产生误解。

说明：较短的单词可通过去掉“元音”形成缩写；较长的单词可取单词的头几个字母形成缩写；一些单词有大家公认的缩写。

示例：如下单词的缩写能够被大家基本认可。

```
temp 可缩写为  tmp
flag 可缩写为  flg
statistic 可缩写为  stat
increment 可缩写为  inc
message 可缩写为  msg
```

规则 2 命名中若有使用特殊约定或缩写，则要有注释说明。

说明：应该在源文件的开始之处，对文件中所使用的缩写或约定，特别是特殊的缩写，进行必要的注释说明。

规则 3 命名风格应自始至终保持一致，不可来回变化。

规则 4 变量名必须以字母或下划线开头，后面可以跟字母、数字或下划线。任何其它字符都标志变量名的结束。

示例：

```
variable_name=”value”
```

规则 5 变量名关于大小写敏感。

规则 6 变量要初始化，严禁使用未经初始化的变量作为右值。

说明：变量未初始化导致错误操作，并在使用前进行初始化。

示例：

```
rm -rf ${chroot}/usr/share/doc
```

如果${chroot}未初始化，则可能导致/usr/share/doc 被删除。

规则 7 变量赋值时’=’两边不能有空格。如果要给变量赋空值，则可在等号后面跟一个换行符。

示例：

```
variable=”value”
```

规则 8 根据变量的作用域，变量可以分为局部变量和环境变量。局部变量只在创建它们的 shell 中可用。而环境变量则在所有用户进程中可用，通常也称全局变量，应

尽量减少全局变量的使用。

说明：全局变量是增大模块间耦合的原因之一，故应减少没必要的全局变量以降低模块间的耦合度。

规则 9 环境变量应使用大写，局部变量使用小写，函数命名使用小写。

示例:

```
GLOBAL_VARIABLE=”globalvalue”
local_variable=”localvalue”
function_name()
{
…
}
```

规则 10 防止脚本的环境变量与系统公共环境变量重名，防止脚本的局部变量和全局变量重名。

规则 11 引用变量时，使用${variable}方式。如果对变量值进行判断，使用”${variable}”。

示例：

```
echo ${variable}
[ “${variable}” = “foo” ] && echo “Right, it’s foo…”
[ “$?” = “0” ] && echo “do-something”
```

规则 12 只读变量指不能被清除或修改。用 readonly variable 方式声明。

规则 13 对于非循环变量命名，禁止取单个字符（如 i、j、k…）。

规则 14 除非必要，不使用数字或较奇怪的字符来定义标识符。

示例：如下命名，使人产生疑惑。

```
EXAMPLE_0_TEST
EXAMPLE_1_TEST
set_sls00()
```

应改为有意义的单词命名

```
EXAMPLE_UNIT_TEST
EXAMPLE_ASSERT_TEST
set_udt_msg_sls()
```

规则 15 用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。

说明：下面是一些在软件中常用的反义词组。

```
add / remove       begin / end       create / destroy
insert / delete    first / last      get / release
increment / decrement                put / get
add / delete       lock / unlock     open / close
min / max          old / new         start / stop
next / previous    source / target   show / hide
```

```
send / receive     source / destination
cut / paste        up / down
```

3.2 建议

建议 1 Unix/Linux 系统对大小写敏感，建议文件名不要使用大小写混排方式。

建议 2 如果方法名超过 15 个字母，可采用以去掉元音字母的方法或者以行业内约定俗成的缩写方式缩写函数名。文件名、变量名、函数名不超过 20 个字符。

4. 编码

4.1 规则

规则 1 开发脚本时，必须在脚本开头行编写脚本解析器。
说明：某些开发人员一般采用默认的 B SHELL 即#!/bin/bash ，也有采用#!/bin/sh，两者在语法上是存在一定差异的，无特殊情况下，必须强制使用#!/bin/bash。

规则 2 对于重复使用的代码段，使用函数进行简化。

规则 3 函数功能应单一，不要设计多用途面面俱到的函数。

规则 4 检查函数传入的参数个数和所有参数输入的有效性。

规则 5 备份文件。如果某文件有多份备份需要保留时，建议使用时间戳作为备份命名的一部分。

规则 6 通过“/”来进行目录和文件的区别。

规则 7 给变量赋值时，如果值包含斜杠、空格等特殊字符时，值语句必须使用双引号将值包含起来，当值中包含转义符号‘\’时，必须对其进行转义后进行赋值。
说明：对变量赋值时，如果值中包含空格等特殊字符，则将出现错误或者无法预知的结果，因此使用时必须用双引号将值本身包含起来。

规则 8 应将常用变量集中写在脚本开头。
说明：将变量写在脚本头，集中在一起，方便调试和查找。

示例：
```
VCS_DIR=/opt/VRTSvcs/bin
```

规则 9 初始化一个变量为空时，必须使用 num="" 或者 unset num,不得使用 num=之类的表达式。
说明：需要对一变量进行空赋值时，建议使用 unset。

示例：
建议使用：
```
unset num
num=””
```
不建议使用：
```
num=
```

规则 10 如果变量是从外部输入或者文件中获取值的话，引用时必须使用双引号。
说明：变量的值如果是外部的输入或读取文件内容时，其内容中可能包含特殊字符，如

果不使用双引号，一旦出现特殊字符将导致语句出现不可预知的错误。

示例：
脚本接受用户输入，然后打印，如果不使用双引号将出现错误。
正确的语句：

```
#!/bin/bash
while read -r LINE ;
do
echo “${LINE}”
done
```

用户输入*程序正常打印*，程序运行正常，达到预期目的。
错误的语句：

```
#!/bin/bash
while read -r LINE ;
do
echo ${LINE}
done
```

用户输入*程序将输出当前目录下的文件名，程序运行出错，扩展输出了文件名，未达到预期目的。

规则 11 不建议使用 rm * 命令匹配删除所有文件，如有必要使用时，必须进行充分判断，考虑各种情况，谨慎使用。
说明：语句中调用 rm －rf *命令删除文件时，易出现文件误删除，导致系统严重事故发生。因此如果语句中需要使用该命令，务必将匹配条件做到精准；删除文件的路径，优先使用绝对路径，路径中包含变量时，充分考虑各种情况，确保不会出现因匹配导致的误操作漏洞。

示例 1：
语句中调用 rm －rf * 命令删除某个目录下所有以 S 打头和 isisd 结尾的所有文件。
正确的语句：

```
rm -rf /etc/init.d/rc5.d/S*isisd
```

解读：匹配模式做到精准匹配，锁定匹配范围的大小，避免自动匹配到其他文件名；文件路径为绝对路径，避免因路径错误而导致删除其他文件。

示例 2：
删除某变量指定的目录下所有文件。
正确的语句：

```
[ -n “${FILE_PATH}” ] && rm -rf ${FILE_PATH}/*
```

解读：如果不对变量${FILE_PATH}进行判断，当某些场景下出现为空时，语句将错误的删除根目录下的文件，引发系统事故，增加判断后，出错风险大大降低。

错误的语句：

```
rm -rf ${FILE_PATH}/*
```

变量${FILE_PATH}为空时，直接删除了系统根目录下的文件，导致事故。

规则 12 脚本中对变量进行比较时，请使用对应的操作符号。
说明：比较变量，在确定变量类型的情况下，请使用对应的操作符号，请不要统统都当做字符串来处理。
比较时请使用：

对应的操作	整数操作	字符串操作
等于	-eq	=
不等于	-ne	!=
大于	-gt	
小于	-lt	
大于或等于	-ge	
小于或等于	-le	
为空		-z
不为空		-n

示例：
不建议：

```
echo "abc" | grep "ab" >/dev/null 2>&1
[ $? = 0 ] && echo "Yes,found it"
```

建议：

```
echo "abc" | grep "ab" >/dev/null 2>&1
[ $? -eq 0 ] && echo "Yes,found it"
```

规则 13 通过命令 ln 创建软链接文件时，语句逻辑中必须先判断文件是否存在，存在时必须先删除，再通过 ln 命令创建新的链接。
说明：当链接文件已存在时，且指向的目标类型为目录时，通过 ln 命令新建链接，命令返回成功，但实际没有达到目的，而是在原目标目录内产生了一个链接文件。

示例：
语法： ln -sf 目标文件 链接文件名

```
linux0710:/root # ln -sf /home a
linux0710:/root # echo $?
0
linux0710:/root # ls -ltr a
lrwxrwxrwx 1 root root 5 Feb 14 08:30 a -> /home
linux0710:/root #  ln -sf /dev a
linux0710:/root # echo $?
0
linux0710:/root # ls -ltr a
lrwxrwxrwx 1 root root 5 Feb 14 08:30 a -> /home
```

实际没有链接到/dev，而是在/home 下新产生了一个 dev 的链接文件，因此看到上面命令返回为 0 成功，它是表示在/home 下新建 dev 链接文件成功：

```
linux0710:/root # ls -ltr /home
```

```
lrwxrwxrwx 1 root root 4 Feb 14 08:33 dev -> /dev
```

这个问题并不是 ln 命令的 BUG,而是由于用户的一般认知与 ln 命令本身的实现原理之间存在差异导致，用户一般认为 ln 创建链接时，类似于 Windows 平台的快捷方式，执行命令时应该理所当然的重新指向到新的目标，但 ln 命令的处理机制不一样，不同点就是：处理指向目录的链接文件与普通文件不同，当链接文件存在时，再次执行 ln 创建链接时会进行扩展，会识别之前链接指向的目录，自动在该目录内建立对应文件名的链接文件。因此会出现上面示例的情形。

错误的写法：

```
ln -sf /dev /home/a
```

如果/home/a 链接文件存在且指向另外一个目录时，将达不到程序预期目的。

正确的写法：

```
[ -h /home/a ] && rm -f /home/a
ln -sf /dev /home/a
```

规则 14 搜索匹配以某字符串开头或结尾的场景且空格字符不影响匹配目的时，必须在搜索语句前对字符串处理，将开头和结尾空格进行剔除。

说明：在某些场景下，用户需要的是查找以某字符开头或结尾的行，且空格不影响匹配目的时，由于某些命令的输出格式、文件格式等发生变化，可能导致匹配失效，从而出现错误。

示例：

通过命令 lsscsi 然后查找 sdb 硬盘是否存在。

```
linux-01:/var/adm/autoinstall/logs # lsscsi

[0:0:0:0] disk SEAGATE ST9146803SS 0006 -
[0:0:1:0] disk SEAGATE ST9146803SS 0006 -
[0:1:4:0] disk LSILOGIC Logical Volume 3000 /dev/sda
[8:0:0:0] disk HUAWEI S2600   1   -
[8:0:0:1] disk HUAWEI S2600   1   -
[8:0:1:0] disk HUAWEI S2600   1   -
[8:0:1:1] disk HUAWEI S2600   1   -
[9:0:0:0] disk up updisk   1   /dev/sdb
[9:0:0:1] disk up updisk   1   /dev/sdc
```

由于不同操作系统版本中 lsscsi 命令的版本不同，输出格式存在一些差异，在某些版本中，该命令的输出发生改变，在每行结尾多增加了一个空格字符，从而导致原有语句出现错误无法正常匹配的现象。

同样的问题也普遍存在文件中，特别是手工编辑的某些文件，可能会在行前或结尾多附带有空格，如果不进行处理，将无法正确匹配。

建议使用：

```
lsscsi | sed 's/^ //g;s/ $//g' | grep “sdb$”
```

不建议使用：

```
lsscsi | grep “sdb$”
```

规则 15 字符串比较语句中，比较时请使用双引号将变量括起来。
说明：由于变量的字符串值是动态获取，最后获取的值可能存在一些特殊的符号等，这将导致语句出现语法错误。

示例：
建议：

```
[ -n “$STR” ] && echo Yes
```

不建议：

```
[ -n $STR ] && echo Yes
```

规则 16 命令替换使用$(),而不是使用``。
说明：
1、`` 很容易与 ''(单引号)搞混乱，尤其对初学者来说。
2、使用$()对命令域包含起来，不容易丢失，清晰直观。

示例：
建议使用：

```
kernel=$(uname -r)
```

不建议使用：

```
kernel=`uname -r`
```

规则 17 对变量进行引用或字符串组合时，必须使用${}将变量括含起来。
说明：对变量进行字符串组合或者引用时，虽然脚本解析器会自动匹配变量，但是在某些特殊的变量名称场景下可能出错，为了避免出错，必须将变量通过${}括含起来。

示例：
假设需要打印：The host is hosta

```
myhost=”host”
echo “The host is $myhost a”
echo “The host is $myhost a”
```

那么上述语句将打印错误。
建议使用：

```
myhost=”host”
echo “The host is ${myhost} a”
```

规则 18 如果需要执行确定次数的增量循环，用 for 语句替代 while 语句。

规则 19 当开发系统服务启动初始化脚本时，必须增加正确的启动头文件。
说明：开发的程序要增加到操作系统服务中自动启动时，需要编写启动初始化脚本，放置于/etc/init.d/目录下，同时通过 chkconfig 命令增加服务，用于系统自动启动该服务，用于启动该服务的脚本必须编写正确的头文件。具体可参考系统目录/etc/init.d 内其他服务启动脚本的格式。

示例：

```
### BEGIN INIT INFO
# Provides:        bar
# Required-Start:  $network
# Required-Stop:   3 5
# Default-Start:   0 1 2 6
# Default-Stop:
# Short-Description: bar daemon, providing a useful network service
# Description: The bar daemon is a sample network
#     service. We want it to be active in runlevels 3
#     and 5, as these are the runlevels with the network
#     available.
### END INIT INFO
```

规则 20 脚本运行进入非正常条件下的逻辑时，则应该记录日志。
说明：此规则指通常的脚本正常运行下，由于业务逻辑条件不符合而进入异常处理后，应该记录日志，方便定位和跟踪。

示例：

```
if [ $? -eq 0 ] ; then
    # 条件成立语句
    .... ...
else
    # 条件失败时应该记录日志后再退出脚本
    logger_error “Error, the connection failed.”
    exit 1
fi
```

4.2 建议

建议 1 建议用户自定义的函数，在函数前加关键字 function.

示例：

```
function myfunc()
{
    …
}
```

建议 2 方法内的局部变量使用 local 限定符。

示例：

```
function startabc()
```

```
{
    local i=0
    …
}
```

建议 3 尽量不要使用一连串的 if/elif 语句，而应该以 case 语句替代。

建议 4 if/then/else 语句中最可能被执行的部分应该放在 then 子句中，不太可能被执行的部分应该放在 else 子句中。

建议 5 不要使用难懂的技巧性很高的语句，除非很有必要时。

说明：高技巧语句不等于高效率的程序，实际上程序的效率关键在于设计与算法。

5. 注意事项

5.1 特殊保留字

$IFS	这个变量中保存了用于分割输入参数的分割字符，默认识空格。
$HOME	这个变量中存储了当前用户的根目录路径。
$PATH	这个变量中存储了当前 Shell 的默认路径字符串。
$PS1	表示第一个系统提示符。
$PS2	表示的二个系统提示符。
$PWD	表示当前工作路径。
$EDITOR	表示系统的默认编辑器名称。
$BASH	表示当前 Shell 的路径字符串。
$0, $1, $2, ...	表示系统传给脚本程序或脚本程序传给函数的第 0 个、第一个、第二个等参数。
$#	表示脚本程序的命令参数个数或函数的参数个数。
$$	表示该脚本程序的进程号，常用于生成文件名唯一的临时文件。
$?	表示脚本程序或函数的返回状态值，正常为 0，否则为非零的错误号。
$*	表示所有的脚本参数或函数参数。
$@	和 $* 涵义相似，但是比 $* 更安全。
$!	表示最近一个在后台运行的进程的进程号。

5.2 算术运算符

+ - * / % 表示加减乘除和取余运算
+= -= *= /= 同 C 语言中的含义

位操作符
<< <<= >> >>= 表示位左右移一位操作
& &= | |= 表示按位与、位或操作
~ ! 表示非操作
^ ^= 表示异或操作

关系运算符
< > <= >= == != 表示大于、小于、大于等于、小于等于、等于、不等于操作
&& || 逻辑与、逻辑或操作

5.3 变量通过${}的特殊操作

字符串操作符（替换操作符）

${var:-word}	如果 var 存在且不为空，返回它的值，否则返回 word
${var:=word}	如果 var 存在且不为空，返回它的值，否则将 word 赋给 var，返回它的值
${var:+word}	如果 var 存在且不为空，返回 word，否则返回空
${var:?message}	如果 var 存在且不为空，返回它的值，否则显示“bash2:$var:$message”，然后退出当前命令或脚本
${var:offset[:length]}	从 offset 位置开始返回 var 的一个长为 length 的子串，若没有 length，则默认到 var 串末尾

模式匹配操作符

${var#pattern}	从 var 头部开始，删除和 pattern 匹配的最短模式串，然后返回 剩余串
${var##pattern}	从 var 头部开始，删除和 pattern 匹配的最长模式串，然后返回 剩余串，basename?path＝${path##*/}
${var%pattern}	从 var 尾部开始，删除和 pattern 匹配的最短模式串，然后返回 剩余串，dirname?path＝${path%/*}
${var%%pattern}	从 var 尾部开始，删除和 pattern 匹配的最长模式串，然后返回 剩余串
${var/pattern/string}	用 string 替换 var 中和 pattern 匹配的最长模式串
${var/pattern/string}	用 string 替换 var 中和 pattern 匹配的最长模式串（仅替换第一个匹配的串）
${var//pattern/string}	用 string 替换 var 中和 pattern 匹配的最长模式串(全局匹配，替换所有匹配的串)

5.4 正则表达式

.	匹配除换行符以外的所有字符
x?	匹配 0 次或一次 x 字符串
x*	匹配 0 次或多次 x 字符串，但匹配可能的最少次数
x+	匹配 1 次或多次 x 字符串，但匹配可能的最少次数
.*	匹配 0 次或多次的任何字符
.+	匹配 1 次或多次的任何字符
{m}	匹配刚好是 m 个 的指定字符串
{m,n}	匹配在 m 个 以上 n 个 以下 的指定字符串
{m,}	匹配 m 个 以上 的指定字符串
[]	匹配符合 [] 内的字符
[^]	匹配不符合 [] 内的字符
[0-9]	匹配所有数字字符
[a-z]	匹配所有小写字母字符
[^0-9]	匹配所有非数字字符
[^a-z]	匹配所有非小写字母字符
^	匹配字符开头的字符

$	匹配字符结尾的字符
\d	匹配一个数字的字符，和 [0-9] 语法一样
\d+	匹配多个数字字符串，和 [0-9]+ 语法一样
\D	非数字，其他同 \d
\D+	非数字，其他同 \d+
\w	英文字母或数字的字符串，和 [a-zA-Z0-9] 语法一样
\w+	和 [a-zA-Z0-9]+ 语法一样
\W	非英文字母或数字的字符串，和 [^a-zA-Z0-9] 语法一样
\W+	和 [^a-zA-Z0-9]+ 语法一样
\s	空格，和 [\n\t\r\f] 语法一样
\s+	和 [\n\t\r\f]+ 一样
\S	非空格，和 [^\n\t\r\f] 语法一样
\S+	和 [^\n\t\r\f]+ 语法一样
\b	匹配以英文字母,数字为边界的字符串
\B	匹配不以英文字母,数值为边界的字符串
a|b|c	匹配符合 a 字符 或是 b 字符 或是 c 字符 的字符串
abc	匹配含有 abc 的字符串
(pattern)	这个符号会记住所找寻到的字符串，是一个很实用的语法。第一个 () 内所找到的字符串变成 $1 这个变量或是 \1 变量，第二个 () 内所找到的字符串变成 $2 这个变量或是 \2 变量，以此类推下去。
/pattern/i	i 这个参数表示忽略英文大小写，也就是在匹配字符串的时候，不考虑英文的大小写问题。
\	如果要在 pattern 模式中找寻一个特殊字符，如 "*"，则要在这个字符前加上 \ 符号，这样才会让特殊字符失效。