makefile\_shell脚本学习CQIU

2017年4月17日星期一

目 录

[1. 模板标题1 3](#_Toc1684283382)

[1.1 模板标题2 3](#_Toc1267391699)

[1.1.1 模板标题3 3](#_Toc397310112)

[1.1.1.1 模板标题4 3](#_Toc476051819)

## 模板标题1

见下文

### 模板标题2

见下文

#### 模板标题3

见下文

##### 模板标题4

见下文

###### 模板标题5

见下文

模板标题6

见下文

模板标题6

见下文

模板标题7

见下文



对于WPS来说，可以从”文件”-->”页面设置”-->”文档网络”，然后选择“无网络”

## Makefile与shell学习

正文见下

### 背景

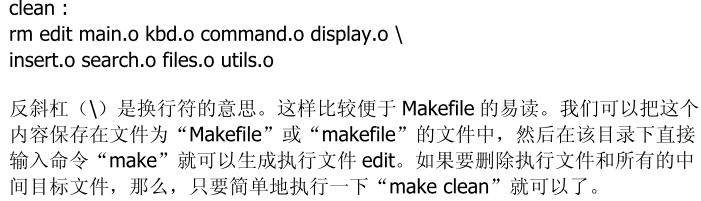
在开发仿真平台过程中，基本原则是能用makefile就用makefile脚本语言，makefile搞不定的地方，可以借助shell脚本进行编程，shell的好处是，可以直接在makefile脚本里面写代码。比调用python，tcl，perl之类的好太多了。

### Makefile

基本介绍见下

#### 几种常见使用技巧

##### 反斜杠用法



#### 局限性

Makefile脚本本身有很多局限性，正则表达式也不够全面，比较烂，不过可以通过借助shell脚本进行编程。

##### 变量不能比较大小

makfile里面定义的变量，不能判断大小哦。

##### 正则表达式支持的很少

Makfile中正则表达式很烂，只有\*,$,^之类的支持，空格目前还不知道怎么支持。

### Shell

这里面介绍的shell，都是能保证shell在makefile中能正常工作的脚本介绍。

#### Shell脚本编程常见问题总结

#### 几种常见命令配合makefile

##### Grep命令

Linux系统中grep命令是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹 配的行打印出来。grep全称是Global Regular Expression Print，表示全局正则表达式版本，它的使用权限是所有用户。

grep的工作方式是这样的，它在一个或多个文件中搜索字符串模板。如果模板包括空格，则必须被引用，模板后的所有字符串被看作文件名。搜索的结果被送到标准输出，不影响原文件内容。

grep可用于shell脚本，因为grep通过返回一个状态值来说明搜索的状态，如果模板搜索成功，则返回0，如果搜索不成功，则返回1，如果搜索的文件不存在，则返回2。我们利用这些返回值就可进行一些自动化的文本处理工作。

###### 命令格式

grep [option] pattern file

举例说明： grep “^axx” aaa.v

###### 正则表达式

grep的规则表达式，这个表达式在使用时，grep “你的正则表达式” 文件路径

^  #锚定行的开始 如：'^grep'匹配所有以grep开头的行。

$  #锚定行的结束 如：'grep$'匹配所有以grep结尾的行。

.  #匹配一个非换行符的字符 如：'gr.p'匹配gr后接一个任意字符，然后是p。

\*  #匹配零个或多个先前字符 如：'\*grep'匹配所有一个或多个空格后紧跟grep的行。

.\*   #一起用代表任意字符。

[]   #匹配一个指定范围内的字符，如'[Gg]rep'匹配Grep和grep。

[^]  #匹配一个不在指定范围内的字符，如：'[^A-FH-Z]rep'匹配不包含A-R和T-Z的一个字母开头，紧跟rep的行。

\(..\)  #标记匹配字符，如'\(love\)'，love被标记为1。

\<      #锚定单词的开始，如:'\<grep'匹配包含以grep开头的单词的行。

\>      #锚定单词的结束，如'grep\>'匹配包含以grep结尾的单词的行。

x\{m\}  #重复字符x，m次，如：'0\{5\}'匹配包含5个o的行。

x\{m,\}  #重复字符x,至少m次，如：'o\{5,\}'匹配至少有5个o的行。

x\{m,n\}  #重复字符x，至少m次，不多于n次，如：'o\{5,10\}'匹配5--10个o的行。

\w    #匹配文字和数字字符，也就是[A-Za-z0-9]，如：'G\w\*p'匹配以G后跟零个或多个文字或数字字符，然后是p。

\W    #\w的反置形式，匹配一个或多个非单词字符，如点号句号等。

\b    #单词锁定符，如: '\bgrep\b'只匹配grep。

POSIX字符:

为了在不同国家的字符编码中保持一至，POSIX(The Portable Operating System Interface)增加了特殊的字符类，如[:alnum:]是[A-Za-z0-9]的另一个写法。要把它们放到[]号内才能成为正则表达式，如[A- Za-z0-9]或[[:alnum:]]。在linux下的grep除fgrep外，都支持POSIX的字符类。

[:alnum:]    #文字数字字符

[:alpha:]    #文字字符

[:digit:]    #数字字符

[:graph:]    #非空字符（非空格、控制字符）

[:lower:]    #小写字符

[:cntrl:]    #控制字符

[:print:]    #非空字符（包括空格）

[:punct:]    #标点符号

[:space:]    #所有空白字符（新行，空格，制表符）

[:upper:]    #大写字符

[:xdigit:]   #十六进制数字（0-9，a-f，A-F）

###### 命令参数

-a   --text   #不要忽略二进制的数据。

-A<显示行数>   --after-context=<显示行数>   #除了显示符合范本样式的那一列之外，并显示该行之后的内容。

-b   --byte-offset   #在显示符合样式的那一行之前，标示出该行第一个字符的编号。

-B<显示行数>   --before-context=<显示行数>   #除了显示符合样式的那一行之外，并显示该行之前的内容。

-c    --count   #计算符合样式的列数。

-C<显示行数>    --context=<显示行数>或-<显示行数>   #除了显示符合样式的那一行之外，并显示该行之前后的内容。

-d <动作>      --directories=<动作>   #当指定要查找的是目录而非文件时，必须使用这项参数，否则grep指令将回报信息并停止动作。

-e<范本样式>  --regexp=<范本样式>   #指定字符串做为查找文件内容的样式。

-E      --extended-regexp   #将样式为延伸的普通表示法来使用。

-f<规则文件>  --file=<规则文件>   #指定规则文件，其内容含有一个或多个规则样式，让grep查找符合规则条件的文件内容，格式为每行一个规则样式。

-F   --fixed-regexp   #将样式视为固定字符串的列表。

-G   --basic-regexp   #将样式视为普通的表示法来使用。

-h   --no-filename   #在显示符合样式的那一行之前，不标示该行所属的文件名称。

-H   --with-filename   #在显示符合样式的那一行之前，表示该行所属的文件名称。

-i    --ignore-case   #忽略字符大小写的差别。

-l    --file-with-matches   #列出文件内容符合指定的样式的文件名称。

-L   --files-without-match   #列出文件内容不符合指定的样式的文件名称。

-n   --line-number   #在显示符合样式的那一行之前，标示出该行的列数编号。

-q   --quiet或--silent   #不显示任何信息。

-r   --recursive   #此参数的效果和指定“-d recurse”参数相同。

-s   --no-messages   #不显示错误信息。

-v   --revert-match   #显示不包含匹配文本的所有行。

-V   --version   #显示版本信息。

-w   --word-regexp   #只显示全字符合的列。

-x    --line-regexp   #只显示全列符合的列。

-y   #此参数的效果和指定“-i”参数相同。

#### shell中查找/抽取指定字符串的方法

shell脚本查找、抽取指定字符串的方法

最近使用：iwconfig wlan0 | sed 's/ /\n/g' | grep -i mode      //查看wlan0模式  
原理：先以空格为通配符，并将其替换成换行符“\n”；每一段字符串单独成行，再grep查找  
  
在linux中经常要对一些动态的文本文件抽取指定的字符串，比如执行ps命令后想要获取指定的运行进程（如ps自己）的PID号(同一个进程每次启动的时候pid号是随机分配的)。该怎么办呢？当然，可以用一些截取字符串的方法，这里介绍一下用2种方法来解决这类问题。

一、sed+grep方法：

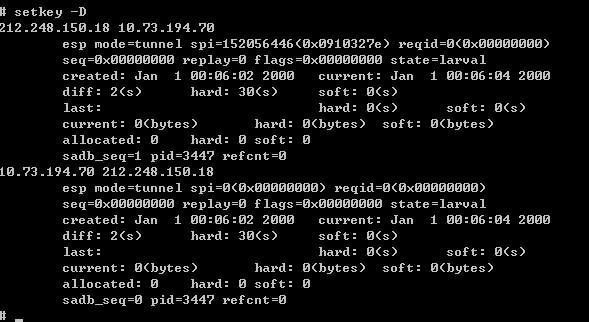
首先大概了解一下sed，sed是linux里面一个非交互性的文本流编辑器（好长的定义，反正我听起来我很拗口）。解释2点：

1，非交互性：这里非交互性是指sed工具并不直接编辑目的文本文件，而是编辑目的文件的一个缓存拷贝，只能够修改拷贝文件，不会对源文件有任何的改动！也就是说编辑并不在目的文件现场！比如我想要编辑example.log文件，如果用vi编辑器必须直接打开example.log文件，然后在里面编辑。而sed是编辑example.log文件的一个拷贝，那么你可能要问了，这个拷贝在哪儿呢？这个拷贝在缓存里面，不需要我们自己拷贝，只要你调用了sed工具来编辑，系统自动会帮我们拷贝出一个example.log文件供你去编辑，从而保护好源文件不被随意改动。

2，文本流：文本流是指我们通过一定的shell指令对文本进行编辑，而不想vi编辑器一样需要把文件打开之后才能够进行编辑。比如我要删除example.log里面的第一行。vi里面必须先打开文件，然后选中第一行，再按下backspace键删除。如果你使用sed命令的话，就只要执行以下一行代码就可以干脆利落的删除掉第一行：sed '1d' example.log。现在可以知道，sed工具无需打开文件再进行操作，而是通过一系列命令操作文本流，所以叫做文本流编辑器。

好，现在来解决问题：

举个例子，比如使用ipsec的时候经常需要使用的一个命令是setkey -D来查看spi是否正常生成了。而每次ipsec连接的时候产生的spi是不一样的，当spi产生失误的时候，就会出现spi=000000000;现在我想通过一定的shell脚本获取spi的值。setkey -D产生而定输出如下图

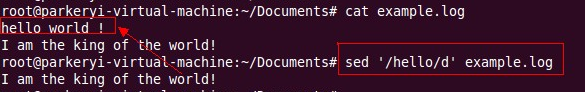


我们在说sed，当然使用sed来解决该问题。

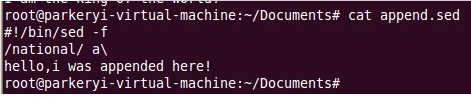
明确一下我们的目的：我们的目的是要获取上图中的一段数据：spi=152056446，并且每次如果ipsec重新启动的时候spi后面的数据会不一样(但长度一样，并且都是数字)。那么我们想到自然要用到正则表达式来匹配该段数据，然后输出到一个变量或者文件供我们后续使用。

再来看看sed给我们提供了什么方法：sed可以很方便的把某一行打印出来，sed -n '2p' example.log会打印出第二行。也就是说sed很容易获取行，这显然不能满足我们的需求。sed不会这么简单！sed还提供了删除，追加，插入，替换等丰富的方法来对一段文本进行编辑。

删除某一行：



追加：把hello,i was appended here 追加到有national这一行的的后面。



替换：sed 's/source string/destination string/' example.log

那么，我想既然sed可以非常方便的获取到某一行数据，而它又能够对文本进行编辑，那么，我的想法是，先用sed对文本进行一定的编辑，把目标字符串编辑到某一行内，我就非常方便的获取到了

好，照着这条思路走：

1，编辑文本，把目标字符串放到单独的一行里面来：观察之后可以看出，该段目的字符串spi=152056446(0x0910327e)前后都有空格，那么就用空格作为重新编辑的标记位，前后换行。

输入命令：sed 's/ /\n/g' example.log。解释下命令，s是替换的标记，第一个/ /里面有一个空格，意思是查找所有含有空格的行，最后的g表明要对该行的所有空格进行查询，而不只是查询到第一个就查询下一行，第二个/\n/是一个换行符，结合前面的空格查询语法，可以对所有的空格替换成换行符。里面的命令执行后会把文本重新编排，遇到空格就换行，这样，目标字符串就已经到了单独的一行里面去了！！！

2，获取目标行字符串：现在就很简单了，我们可以用grep来获取(grep "spi" example.log)，也可以用sed来获取(sed '/spi/p' example)，把屏幕输出重定向到一个文件里面去，或者赋给一个变量，这样，我们抽取目标字符串的任务就完成了！！！

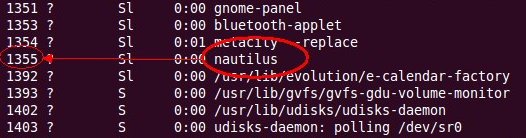
是不是很简单！！！IMG_259好，下面来看awk方法

二、awk方法：

同样的，首先来大概了解一下awk方法的精髓，awk方法主要是对文本进行“列”的操作，这个对比一下sed和grep可能我们更容易理解一些，sed和grep主要是对文本进行“行”的操作，awk会把每一列都取一个名字，从第一列开始：分别为$1,$2...$n，好吧，这样就可以按照名字来分别操作列了。

好吧，现在我们来实例操作一下。

我们从ps输出里面去查询指定进程名字的PID，在我的终端直接输入ps x之后会显示当前进程的名字，其部分截图如下：



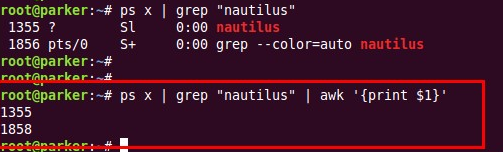
现在我想获取的是当前ps x输出里面的进程名为上图中红圈部分nautilus的PID值1355，由于相同进程会在不同的时候开启的时候所获取到的PID号不一样，所以当前的PID值1355其实是个不一定的数字。于是，我们只能通过进程名字nautilus来获取当前PID。好，目的明确之后进行操作：

1，配合使用管道，使用grep获取到包含nautilus的行：



发现有2个nautilus，好吧，那我们就来操作2个nautilus的PID，只是需要配合使用一下sed,没事，就当复习一下；

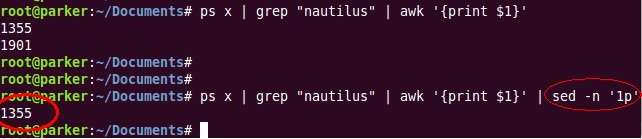
2，现在是awk派上用场的时候了：



继续使用管道，可以看到，利用命令IMG_263可以得到进程名为nautilus的PID列：1355和1858。

这里先来个小插曲，简单介绍一下awk命令的语法，awk基本语法为：awk [-F] "field-operator" 'comand' inputfiles，-F和field-operator一起使用，field-operator是域分隔符，如果不使用-F选项，则默认的域分隔符为空格。后面command命令一般需要用一堆“{}”括起来，然后进行必要的操作，比较全面一点的command命令'{if($1~/^A/) print $1}'，翻译一下这个命令就是，如果第一列($1)里面有匹配(~)正则表达式(/^A/)的话，那么就输出(print)到标准输出。需要注意的是，条件必须要用一堆"()"括起来，正则表达式需要用“//”括起来。当然，完全可以不要条件匹配，可以直接输出指定列，如'{print $1}'。好，awk就介绍到这里，不过awk是一个非常非常强大的文本格式化抽取的工具，需要专门的学习。后面我再写一个关于awk学习的总结。

3，现在需要利用sed对"行"的操作了，我们获取到的第一行就是第一个nautilus的PID值，上面我们已经介绍了sed命令，所以很简单的，接着上面的管道，输入以下命令：



很快，我们就可以得到第一个nautilus的PID值了IMG_265

是不是很简单，就简单的三部操作，grep——awk——sed，这三真是文本操作里面的拼命三郎啊

三、现在来稍微介绍一下cut方法

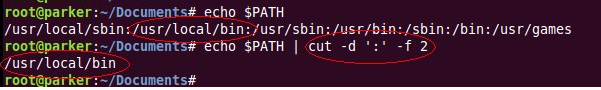
cut也是一个比较强大的工具，可以对一行字符串进行多种模式匹配的剪切操作，也可以对一个排列非常整齐的文本进行操作，下面通过2个例子来大致说明 一下cut的操作。

1，先说明cut对一行字符串的操作：

在终端执行echo $PATH命令，可以获得当前默认的bash路径，如下图：

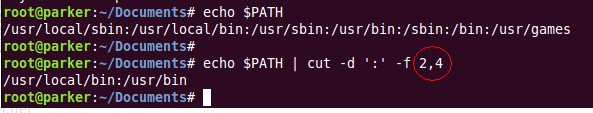


其中每一个路径都被一个“：”分隔开来，现在我想获取第1个和第2个冒号之间的路径，使用cut将会非常之方便，在终端输入命令：



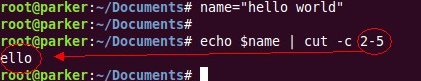
确实，我们获取到了第一个冒号与第二个冒号之间的字符串“/usr/local/bin”，现在来稍微解析一下这条命令，-d和':'一起把管道输入的一行字符串进行了域的分隔，每一个分隔符(在这里是个冒号“:”)前面的字符串被称为一个域，若有n个分隔符，这个域的编号则从1开始到n+1，第1个分隔符前面的字符串为第一个域。这个域对应在该命令行则是-f后面的数字2，所以该命令`cut -d ':' -f 2`表达的意思是输出第二个分隔符“：”之前的域。

如果我们执行

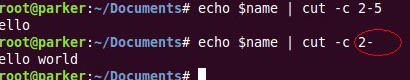


或者是cut -d ':' -f 2-4，则可以输出第2个和第四个域的字符串，但是还多了一个分隔符，这点我也没搞懂。

上面是是用分隔符对一行字符串进行操作，下面我们队字符串的每一个字母当成一个数组里面的元素，只是第一个元素是从1开始，而不是传统数组的0开始。这样可以进行另外一种cut操作，现在先定义一个字符串变量，并从中截取出1-5之间的字符：



可见，这条命令很好的把字符串“hello world”从第2个字母到第5个字母cut了下来，把最后的参数进行一下修改，如下:

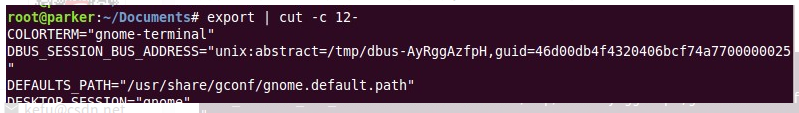


则可以截取到第2个字符以及后面的所有的字符。

2，cut对格式化整齐的文本进行操作：

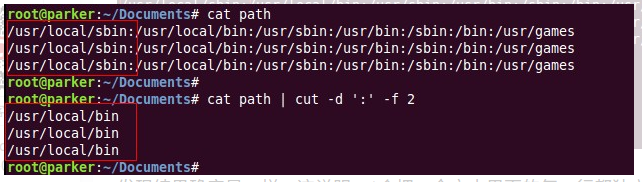


这是一段格式化比较整齐的数据，每一行前面的“declare -x”都是一模一样的，我现在想把这一段给去掉：



其实我们发现对该文本的操作跟上的一行字符串的操作没什么区别。

那么就验证一下对分隔符的操作是否也一样：



发现结果确实是一样。这说明cut会把一个文本里面的每一行都独立的对待，然后再操作

### 仿真平台脚本

### Caffe中Makefile脚本

#### 模板标题3

见下文

##### 模板标题4

见下文

###### 模板标题5

见下文

模板标题6

见下文

模板标题6

见下文

模板标题7

见下文