玩透 sed

探究 sed 原理

文章内容完全原创，本人保留所有权利

欢迎善意传播，杜绝恶意利用者

作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

Version:1.0 更新时间：2018-04-01

本文档说明：

本人对 sed 的原理和机制做了些深入的研究(理论知识)，小有心得。因此分享几篇 sed 修炼的文章，从入门到深入。当然，可能不适合完全没接触过 sed 的纯初学者，建议去网上找几篇 sed 的用法示例学学，有了基础之后，看这系列的文章想必不会有太大困难。

本文档包含4大篇文章，其中：

1.**第一篇是入门篇，但却是最重要的一篇。**虽然内容不算多，但在里面讲了绝大多数 sed 的工作机制，在后面几篇里都涉及到其中的概念。如果后面几篇文章有看不懂的术语，比如 sed 循环、SCRIPT 循环、自动输出、回到 SCRIPT 循环顶部等，请看这篇文章。

2.**第二篇是 info sed 的翻译篇，花了极大的心血，其中加入了很多注解，算是深入篇也算是 sed 手册。**如果想深入 sed 或者想完全了解 sed 工作机制，info sed 是必看文章，绝对比看《sed & awk》收获大。另外，个人建议，不要阅读 man sed 文档。

3.**第三篇是 sed 高级应用的一个通用型模板用法说明：窗口滑动。**这一篇中是一些很有用的技巧，其内比较详细地介绍了"N"、"D"、"P"命令，也涉及了一些保持空间和 sed 标签跳转的用法。但这些命令本就灵活多变，几篇文章是绝对不可能深入的了的，只能取一些示例说明下，以后如果有时间，会专门开一篇文章收集一些 sed 高级用法的示例。

4.**第四篇是 sed 中几个常见的疑难杂症解疑篇。**内容不多，但真正在使用 sed 的时候可能很有帮助。例如如何在 sed 中使用变量，引号怎么加，如何克服贪婪匹配等等。

第一篇文章(sed 花拳绣腿入门篇)是最重要的一篇，特别是其第一节和第四节是整个 sed 的重中之重，是 sed 的"神"，而那些简单的 sed 用法、示例等等都是 sed 的"形"。如果是 sed 初学者，这些"神"的内容可略看一遍，在有了使用 sed 操作的基础之后，再仔细回头琢磨这些"神"中的每一句话，必能深入 sed。此后，再去阅读第二篇文章(sed 武功心法)掌握更细节的"形"和"神"，最后阅读第三篇文章(sed 高级应用)掌握更高级的操作手段，在实现复杂逻辑时往往能事半功倍，最后还可看看第四篇文章(sed 中的疑难杂症)，在 sed 出错却不知何解时，这篇文章往往能找到答案。

本文档中有些链接可能不是跳到文档的指定位置，而是跳到网站对应的文章上，本人实在懒得一个一个去改跳转位置，忘谅解。

最后附上本人博客地址：博客园--骏马金龙
本人在博客园中分享了非常多的系列文章(如 shell、架构、MySQL)，都很完整，欢迎各位光临。
(1).Linux 基础 & shell 系列文章总目录：
http://www.cnblogs.com/f-ck-need-u/p/7048359.html
(2).网站架构从 0 开始系列文章总目录：
http://www.cnblogs.com/f-ck-need-u/p/7576137.html
(3).数据库系列文章总目录：
http://www.cnblogs.com/f-ck-need-u/p/7586194.html

如发现错误、或有疑惑之处 ，欢迎到本人博客区留言或者联系邮箱 mlongshuai@gmail.com。

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

sed修炼系列(一)：花拳绣腿之入门篇

本节为花拳绣腿招式入门篇，主要目的是入门，为看懂 sed 修炼系列(二)：武功心法做准备。虽然是入门篇，只介绍了基本工作机制以及一些选项和命令，但其中仍然包括了很多 sed 的工作机制细节。对比网上各 sed 相关文章以及介绍 sed 的书籍，基本上都只介绍了 sed 是如何使用的，却没有"How sed Works"这种工作机制的原理性内容，最多给出一段稍微解释下。即使是非常流行的《sed & awk》也只是零零散散地介绍了一些 sed 工作机制细节。

本节目录:

1.基本概念

sed 是一个流式编辑器程序，它读取输入流(可以是文件、标准输入)的每一行放进模式空间(pattern space)，同时将此行行号通过 sed 行号计数器记录在内存中，然后对模式空间中的行进行模式匹配，如果能匹配上则使用 sed 程序内部的命令进行处理，处理结束后，从模式空间中输出(默认)出去，并清空模式空间，随后再从输入流中读取下一行到模式空间中进行相同的操作，直到输入流中的所有行都处理完成。由此可见，sed 是一个循环一个循环处理内容的。

这是 sed 的一个循环的过程：

1. 读取输入流的一行到模式空间。
2. 对模式空间中的内容进行匹配和处理。
3. 自动输出模式空间内容。
4. 清空模式空间内容。
5. 读取输入流的下一行到模式空间。

上述整个循环过程中，**第 2 步是我们写 sed 命令所修改的地方，其余的几个步骤，通过命令行无法改变**。但是，sed 有几个命令和选项能改变第 3、4 步的行为，使其输出总是输出空内容或无法清空模式空间。

sed 程序的语法格式为：

```
sed OPTIONS SCRIPT INPUT_STREAM
```

其中 SCRIPT 部分就是所谓的 sed 脚本，它是 sed 内部命令的集合，sed 中的命令有些奇特，它包含行匹配以及要执行的命令。格式为 `ADDR1[,ADDR2]cmd_list`。例如，要对第 2 行执行删除命令，其命令为 `sed 2d filename`，只输出第 4 行到 6 行，其命令为 `sed -n 4,6p`。

sed 的内部命令非常多，但既然"花拳绣腿篇"，当然只介绍些入门的东西。具体的行匹配方法、有哪些命令以及哪些选项稍后解释。现在的重点是 sed 中的循环过程。既然 SCRIPT 是命令的集合，于是上面的循环过程可以修改为如下：

1. 读取输入流的一行到模式空间。

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

2. 对模式空间中内容执行 SCRIPT。(包括上面示例中的"2d"和"4,6p")
3. 读取输入流的下一行到模式空间。
4. 对模式空间中内容执行 SCRIPT。

其中 SCRIPT 部分包含了 sed 命令行中的内部命令，**还包括两个特殊动作：自动输出和清空模式空间内容。这两个动作是一定会执行的**，只不过有些时候通过某些命令可以使其输出空内容、使其清空不了模式空间。

如果使用编程结构来描述，则大致过程如下：

```
for ((line=1;line<=last_line_num;++line))
do
    read $line to pattern_space;
    while pattern_space is not null
    do
        execute cmd1 in SCRIPT;
        execute cmd2 in SCRIPT;
        execute cmd3 in SCRIPT;
        ……
        auto_print;
        remove_pattern_space;
    done
done
```

其中 while 循环执行的正是 SCRIPT 中的所有命令，只不过一般情况下，while 循环只执行一轮就退出并进入外层的 for 循环。于是，外层的 for 循环称之为"sed 循环"，内层的 while 循环称之为"SCRIPT"循环。所以，for 循环只包含了两个动作：读取下一行和执行 SCRIPT 循环。

其实 while 循环中是有 continue、break 甚至是 exit 的，分别表示回到 SCRIPT 的顶端(即进入下一个 SCRIPT 循环)、退出当前 SCRIPT 循环回到外层 sed 循环以及退出整个 sed 循环。显然，这不是"花拳绣腿"的内容。

最后，说明下 sed 命令行如何书写，其实就是写 SCRIPT 部分，这部分的写法比较灵活，大致有以下几种：

```
# 一行式。多个命令使用分号分隔
sed Address{cmd1;cmd2;cmd3...}

# 多个表达式时，可以使用"-e"选项，也可以不用，但使用分号分隔
sed Address1{cmd1;cmd2;cmd3};Address2{cmd1;cmd2;cmd3}...
sed  -e 'Address1{cmd1;cmd2;cmd3}' -e 'Address2{cmd1;cmd2;cmd3}' ...

# 分行写时
sed Address1{
    cmd1
    cmd2
    cmd3
}
Address2{
    cmd1
    cmd2
    cmd3
}
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

如果是写在文件中，即 sed 脚本，以文件名为 a.sed 为例。

```
#!/usr/bin/sed -f
#注释行
Address1{cmd1;cmd2...}
Address2{cmd1;cmd2...}
......
```

其中 cmd 部分还可以进行模式匹配，也即类似于
Address{{pattern1}cmd1;{pattern2}cmd2}的写法。例如，/^abc/{2d;p}。

有了以上基本的大纲性知识，理解和深入 sed 机制就简单多了。

3.sed 选项

sed 选项不算多，能用到的更没几个。

```
sed OPTIONS SCRIPT INPUT_STREAM
```

可能用到的几个选项：

'-n'
默认情况下，sed 将在**每轮 script 循环结束**时自动输出模式空间中的内容。使用该选项后可以使得这次自动输出动作输出**空内容**，而不是当前模式空间中的内容。注意，"-n"是输出空内容而不是禁用输出动作，虽然两者的结果都是不输出任何内容，但在有些依赖于输出动作和输出流的地方，它们的区别是很大的，前者有输出流，只是输出空流，后者则没有输出流。

'-e SCRIPT'
前文说了，SCRIPT 中包含的是命令的集合，"-e"选项就是向 SCRIPT 中添加命令的。可以省略"-e"选项，但如果命令行容易产生歧义，则使用"-e"选项可明确说明这部分是 SCRIPT 中的命令。另外，如果一个"-e"选项不方便描述所需命令集合时，可以指定多个"-e"选项。

'-f SCRIPT-FILE'
指定包含命令集合的 SCRIPT 文件，让 sed 根据 SCRIPT 文件中的命令集处理输入流。

'-i[SUFFIX]'
该选项指定要将 sed 的输出结果保存(覆盖的方式)到当前编辑的文件中。GNU sed 是通过创建一个临时文件并将输入写入到该临时文件，然后重命名为源文件来实现的。

当当前输入流处理结束后，临时文件被重命名为**源文件**的名称。如果还提供了 SUFFIX，则在重命名临时文件之前，先使用该 SUFFIX 修改源文件名，从而生成一个源文件的备份文件。

临时文件总是会被重命名为源文件名称，也就是说输入流处理结束后，仍使用源文件名的文件是 sed 修改后的文件。文件名中包含了 SUFFIX 的文件则是最原始文件的备份。例如

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

源文件为 a.txt，`sed -i'.log' SCRIPT a.txt` 将生成两个文件：a.txt 和 a.txt.log，前者是 sed 修改后的文件，a.txt.log 是源 a.txt 的备份文件。

重命名的规则如下：如果扩展名不包含符号"*"，将 SUFFIX 添加到原文件名的后面当作文件后缀；如果 SUFFIX 中包含了一个或多个字符"*"，则每个"*"都替换为原文件名。这使得你可以为备份文件添加一个前缀，而不是后缀。如果没有提供 SUFFIX，源文件被覆盖，且不会生成备份文件。

该选项隐含了"-s"选项。

`'-r'`
使用扩展正则表达式，而不是使用默认的基础正则表达式。sed 所支持的扩展正则表达式和 `egrep` 一样。使用扩展正则表达式显得更简洁，因为有些元字符不用再使用反斜线"\"。正则表达式见 grep 命令中文手册。

`'-s'`
默认情况下，如果为 sed 指定了多个输入文件，如 `sed OPTIONS SCRIPT file1 file2 file3`，则多个文件会被 sed 当作一个长的输入流，也就是说所有文件被当成一个大文件。指定该选项后，sed 将认为命令行中给定的每个文件都是独立的输入流。

既然是独立的输入流，**范围定址(如/abc/,/def/)就无法跨越多个文件进行匹配，行号也会在处理每个文件时重置，"$"代表的也将是每个文件的最后一行**。这也意味着，如果不使用该选项，则这几个行为都是可以完成的。

示例：以 sed 命令"p"和"="为例，其中"p"命令用于强制输出当前模式空间中的内容，"="命令用于输出 sed 行号计数器当前的值，即刚被读入到模式空间中的行是输入流中的第几行。

(1).只输出 a.txt 中的第 5 行。

```
sed -n 5p a.txt
```

这里使用了"-n"选项，使得读取到模式空间的每一行都无法被输出，只有明确使用了"p"选项才能被"p"动作输出。由于只有读入的第 5 行内容能匹配"5"，才能被"p"输出。

其实上面的命令和 `sed -n -e '5p' a.txt` 是完全一样的，因为"5p"在 sed 解析命令行时不会产生歧义，所以可以省略"-e"选项。

(2).输出 a.txt，并输出每行的行号。

```
sed '=' a.txt
```

由于要输出 a.txt 的内容，所以不使用"-n"选项，同时"="命令会输出每行行号。

(3).分别输出 a.txt 和 b.txt 的第 5 行，并分别保存到".bak"后缀的文件中。

```
sed -i'*.bak' -n '5p' a.txt b.txt
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

此处必须使用"-s"选项，否则将只会输出"a.txt+b.txt"结合后的第 5 行。但"-i"隐含了"-s"选项。这会生成 4 个文件：a.txt、b.txt 和 a.txt.bak、b.txt.bak。前两个是第 5 行内容，后两个是源文件的备份文件。

(4).使用扩展正则表达式，输出 a.txt 和 b.txt 中能包含 3 个以上字母"a"的行。

```
sed -r -n '/aaa+/p' a.txt b.txt
```

3.定址表达式

当 sed 将输入流中的行读取到模式空间后，就需要对模式空间中的内容进行匹配，如果能匹配就能执行对应的命令，如果不能匹配就直接输出、清空模式空间并进入下一个 sed 循环读取下一行。

匹配的过程称为定址。定址表达式有多种，但总的来说，其格式为`[ADDR1][,ADDR2]`。这可以分为 3 种方式：

1. ADDR1 和 ADDR2 都省略时，表示所有行都能被匹配上。
2. 省略 ADDR2 时，表示只有被 ADDR1 表达式匹配上的行才符合条件。
3. 不省略 ADDR2 时，是范围地址。表示从 ADDR1 匹配成功的行开始，到 ADDR2 匹配成功的行结束。

无论是 ADDR1 还是 ADDR2，都可以使用两种方式进行匹配：行号和正则表达式。如下：

`'N'`
指定一个行号，sed 将只匹配该行。(需要注意，除非使用了"-s"或"-i"选项，sed 将对所有输入文件的行连续计数。)

`'FIRST~STEP'`
表示从第 FIRST 行开始，每隔 STEP 行就再取一次。也就是取行号满足 `FIRST+(N*STEP)`(其中 N>=0)的行。因此，要选择所有奇数行，使用"1~2"；要从第 2 行开始每隔 3 行取一次，使用"2~3"；要从第 10 行开始每隔 5 行取一次，使用"10~5"；而"50~0"则表示只取第 50 行。

`'$'`
默认该符号匹配的是最后一个文件的最后一行，如果指定了"-i"或"-s"，则匹配的是每个文件的最后一行。总之，"$"匹配的是每个输入流的最后一行。

需要注意的是，**sed 采用行号计数器来临时记录当前行的行号，因此 sed 在读取到最后一行前即使是倒数第二行的时候，完全不知道最后一行是第几行，所以代表最后一行的"$" 无法进行任何数学运算**，例如倒数第二行使用"$-1"表示是错误的。而且，"$"只是一个额外的标记符号，当 sed 读取到输入流的最后一行时，发现这就是最后一行，于是为此行打上"$"记号，并读取到模式空间中。

`'/REGEXP/'`
将选择能被正则表达式 REGEXP 匹配的所有行。如果 REGEXP 中自身包含了字符"/"，则必须使用反斜线转义，即`"\/"`。

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

'/REGEXP/I'
和"/REGEXP/"是一样的，只不过匹配的时候不区分大小写。

'\%REGEXP%'
('%'可以使用其他任意单个字符替换。) 这和上一个定址表达式的作用是一样的，只不过是使用符号"%"替换了符号"/"。当 REGEXP 中包含"/"符号时，使用该定址表达式就无需对"/"使用反斜线"\"转义。但如果此时 REGEXP 中包含了"%"符号时，该符号需要使用"\"转义。总之，定址表达式中使用的分隔符在 REGEXP 中出现时，都需要使用反斜线转义。

'ADDR1,+N'
匹配 ADDR1 和其后的 N 行。

'ADDR1,~N'
匹配 ADDR1 和其后的行直到出现 N 的倍数行。倍数可为随意整数倍，只要 N 的倍数是最接近且大于 ADDR1 的即可。 如 ADDR1=1,N=3 匹配 1-3 行，ADDR1=5,N=4 匹配 5-8 行。而"1,+3"匹配的是第一行和其后的 3 行即 1-4 行。

另外，在定址表达式的后面加"!"符号表示反转匹配的含义。也就是说那些匹配的行将不被选择，而是不匹配的行被选择。

例如，以下几个定址的示例：

```
sed -n '3p' INPUTFILE
sed -n '3,5!p' INPUTFILE
sed -n '3,/^# .*/! p' INPUTFILE
sed -n '/abc/,/xyz/p' INPUTFILE
sed -n '!p' INPUTFILE   # 这个有悖常理，但确实是允许的
```

4.sed 常用命令

sed 命令很多，本节的只简单介绍几个最常见的。

此处不以命令的用法为重，而是通过这几个命令，引出sed最重要的原理和执行机制(还包括本文的第一节内容)，并为阅读下一篇文章sed武功心法：info sed打下基础。而且理解了这些原理，再使用sed做任何操作都有理可循，遇到疑难之处也知道如何进行分析。而这，是任何书籍(包括广为推崇的sed && awk)、教学视频和网络文章中都没有的内容(至少我从未见过，这些内容也是我花费大量精力经过大量实验推演出来)。

(1).强制输出命令"p"。
该命令能强制输出当前模式空间的内容。即使使用了"-n"选项。

事实上，它们本就不冲突，因为循环过程如下：

```
for ((line=1;line<=last_line_num;++line))
do
    read $line to pattern_space;
    while pattern_space is not null
    do
        execute cmd1 in SCRIPT;
        execute cmd2 in SCRIPT;
        ADDR1,ADDR2{print};        # "p" command
        ……
        auto_print;
        remove_pattern_space;
    done
done
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

在 sed 处理的过程中，"p"和"auto_print"是两个输出动作，都是输出当前模式空间的内容，只不过 auto_print 是隐含动作。使用了"-n"选项，其所影响的动作仅是"auto_print"，使其输出空内容。也因此，当没有使用"-n"选项时，模式空间的内容会被输出两次。

例如，仅输出标准输入的第 2 行内容。

```
[root@xuexi ~]# echo -e 'abc\nxyz' | sed -n 2p
xyz
```

不加"-n"选项，在"p"输出之后，SCRIPT 循环的结尾处还会被 auto_print 输出一次。

```
[root@xuexi ~]# echo -e 'abc\nxyz' | sed 2p
abc
xyz     # 这是 p 命令输出的结果
xyz     # 这是自动输出的结果
```

(2).删除命令"d"。

命令"d"用于删除**整个模式空间**中的内容，**并立即退出当前 SCRIPT 循环，进入下一个 sed 循环，即读取下一行**。

循环大致格式如下:

```
for ((line=1;line<=last_line_num;++line))
do
    read $line to pattern_space;
    while pattern_space is not null
    do
        execute cmd1 in SCRIPT;
        execute cmd2 in SCRIPT;
        ADDR1,ADDR2{delete;break};      # "d" command
        ……
        auto_print;
        remove_pattern_space;
    done
done
```

唯一需要注意的一点是立即退出当前 SCRIPT 循环，这意味着如果"d"命令后面还有其他的命令，则这些命令都不会执行。

例如：删除 a.txt 中的第 5 行，并保存到原文件中。

```
sed -i '5d' a.txt
```

这里不能使用重定向的方式保存，因为重定向是在 sed 命令执行前被 shell 执行的，所以会截断 a.txt，使得 sed 读取的输入流为空，或者结果出乎意料之外。而"-i"选项则不会操作原文件，而是生成临时文件并在结束时重命名为原文件名。

删除 a.sh 中包含"#"开头的注释行，但第一行的#!/bin/bash 不删除。

```
sed '/^#/{1!d}' a.sh
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

如果"d"后面还有命令，在删除模式空间后，这些命令不会执行，因为会立即退出当前 SCRIPT 循环。例如：

```
echo -e 'abc\nxyz' | sed '{/abc/d;=}'
2
xyz
```

其中"="这个命令用于输出行号，但是结果并没有输出被"abc"匹配的行的行号。

(3).退出 sed 程序命令"q"和"Q"。

使用"q"和"Q"命令的作用是立即退出当前 sed 程序，使其不再执行后面的命令，也不再读取后面的行。因此，在处理大文件或大量文件时，使用"q"或"Q"命令能提高很大效率。它们之间的不同之处在于"q"命令被执行后还会使用自动输出动作输出模式空间的内容，除非使用了"-n"选项。而"Q"命令则会立即退出，不会输出模式空间内容。另外，可以为它们指定退出状态码，例如"q 1"。

使用了"q"和"Q"的 sed 循环结构大致如下：

```
# "q"命令
for ((line=1;line<=last_line_num;++line))
do
    read $line to pattern_space;
    while pattern_space is not null
    do
        execute cmd1 in SCRIPT;
        execute cmd2 in SCRIPT;
        ADDR1,ADDR2{auto_print;exit};     # "q" command
        ……
        auto_print;
        remove_pattern_space;
    done
done

# "Q"命令
for ((line=1;line<=last_line_num;++line))
do
    read $line to pattern_space;
    while pattern_space is not null
    do
        execute cmd1 in SCRIPT;
        execute cmd2 in SCRIPT;
        ADDR1,ADDR2{exit};      # "Q" command
        ……
        auto_print;
        remove_pattern_space;
    done
done
```

例如，搜索脚本 a.sh，当搜索到使用了"."或"source"命令加载环境配置脚本时就输出并立即退出。

```
sed -n -r '/^[ \t]*(\.|source) /{p;q}' a.sh
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

(4).输出行号命令"="。

"="命令用于输出最近被读取行的行号。在 sed 内部，使用行号计数器进行行号计数，每读取一行，行号计数器加 1。计数器的值存储在内存中，在要求输出行号时，直接插入在输出流中的指定位置。由于值是存在于内存中，而非模式空间中，因此不受"-n"选项的影响。

这是一个**依赖于输出流的命令，只要有输出动作就会追加在该输出流的尾部**。

例如，搜索出 httpd.conf 中"DocumentRoot"开头的行的行号，允许有前导空白字符。

```
sed -n '/^[ \t]*DocumentRoot/{p;=}' httpd.conf
DocumentRoot "/var/www/html"
119
```

如果"="命令前没有"p"输出命令，且没有使用"-n"选项，则是输出在 Document 所在行的前一行，因为 SCRIPT 最后的自动输出动作也有输出流。

(5).字符一一对应替换命令"y"。

该命令和"tr"命令的映射功能一样，都是将字符进行一一替换。

例如，将 a.txt 中包含大写字母的 YES、Yes 等替换成小写的 yes。

```
sed 'y/YES/yes/' a.txt
```

(6).手动读取下一行命令"n"。

在 sed 的循环过程中，每个 sed 循环的第一步都是读取输入流的下一行到模式空间中，这是我们无法控制的动作。但 sed 有读取下一行的命令"n"。

由于是读取下一行，所以它会触发自动输出的动作，于是就有了输出流。不仅如此，还应该记住的是：**只要有读取下一行的行为，在其真正开始读取之前一定有隐式自动输出的行为**。

但需注意，当没有下一行可供"n"读取时(例如文件的最后一行已经被读取过了)，将输出模式空间内容后直接退出 sed 程序，使得"n"命令后的所有命令都不会执行，即使是那两个隐含动作。

相应的循环结构如下：

```
for ((line=1;line<=last_line_num;++line))
do
    read $line to pattern_space;
    while pattern_space is not null
    do
        execute cmd1 in SCRIPT;
        execute cmd2 in SCRIPT;
        ADDR1,ADDR2{                # "n" command
            if [ "$line" -ne "$last_line_num" ];then
                auto_print;
                remove_pattern_space;
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

```
                read next_line to pattern_space;
            else
                auto_print;
                remove_pattern_space;
                exit;
            fi
        };
        ……
        auto_print;
        remove_pattern_space;
    done
done
```

注意，是先判断是否有下一行可读取，再输出和清空 pattern space 中的内容，所以 then 和 else 语句中都有这两个动作。 也许感觉上似乎更应该像下面这样的优化形式：

```
 ADDR1,ADDR2{                # "n" command
         auto_print;
         remove_pattern_space;
         [ "$line" -ne "$last_line_num" ] && read next_line to
pattern_space || exit;
 };
```

但事实证明并非如此，证明过程在本节结尾。此处暂不讨论这些复杂的东西，先看看"n"命令的示例。

例如，搜索 a.txt 中包含"redirect"字符串的行以及其下一行，并输出。

```
sed -n '/redirect/{p;n;p}' a.txt
```

再例如下面的命令。

```
echo -e "abc\ndef\nxyz" | sed '/abc/{n;=;p}'
abc
2
def
def
xyz
```

从结果中可以分析出，"n"读取下一行前输出了"abc"，然后立即读入了下一行，所以输出的行号是 2 而不是 1，因为这时候行号计数器已经读取了下一行，随后命令"p"输出了该模式空间的内容，输出后还有一次自动输出的隐含动作，所以"def"被输出了两次。

(7).替换命令"s"。

这是 sed 用的最多的命令。两个字就能概括其功能：替换。将匹配到的内容替换成指定的内容。

"s"命令的语法格式为：其中"/"可以替换成任意其他单个字符。

```
s/REGEXP/REPLACEMENT/FLAGS
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

它使用 REGEXP 去匹配行，将匹配到的那部分字符替换成 REPLACEMENT。FLAGS 是"s"命令的修饰符，常见的有"g"、"p"和"i"或"I"。

- "g"：表示替换行中所有能被 REGEXP 匹配的部分。不使用 g 时，默认只替换行中的第一个匹配内容。此外，"g"还可以替换成一个数值 N，表示只替换行中第 N 个被匹配的内容。
- "p"：输出替换后模式空间中的内容。
- "i"或"I"：REGEXP 匹配时不区分大小写。

REPLACEMENT 中可以使用"\N"(N 是从 1 到 9 的整数)进行后向引用，所代表的是 REGEXP 第 N 个括号(...)中匹配的内容。另外，REPLACEMENT 中可以包含未转义的"&"符号，这表示引用 pattern space 中被匹配的整个内容。需要注意，"&"是引用 pattern space 中的所有匹配，不仅仅只是括号的分组匹配。

例如，删除 a.sh 中所有"#"开头(可以包括前导空白)的注释符号"#"，但第一行"#!/bin/bash"不处理。

```
sed -i '2,$s/^[ \t]*#//' a.sh
```

为 a.sh 文件中的第 5 行到最后一行的行首加上注释符号"#"。

```
sed '5,$s/^/#/' a.sh
```

将 a.sh 中所有的"int"单词替换成"SIGINT"。

```
sed 's/\bint\b/SIGINT/g' a.sh
```

将 a.sh 中"cmd1 && cmd2 || cmd3"的 cmd2 和 cmd3 命令对调个位置。

```
sed 's%&&\(.*\) ||\(.*\)%\&\&\2 ||\1%' a.sh
```

这里使用了"%"代替"/"，且在 REPLACEMENT 部分对"&"进行了转义，因为该符号在 REPLACEMENT 中时表示的是引用 REGEXP 所匹配的所有内容。

(8).追加、插入和修改命令"a"、"i"、"c"。

这 3 个命令的格式是"[a|i|c] TEXT"，表示将 TEXT 内容队列化到内存中，当有输出流或者说有输出动作的时候，半路追上输出流，分别追加、插入和替换到该输出流然后输出。追加是指追加在输出流的尾部，插入是指插入在输出流的首部，替换是指将整个输出流替换掉。"c"命令和"a"、"i"命令有一丝不同，它替换结束后立即退出当前 SCRIPT 循环，并进入下一个 sed 循环，因此"c"命令后的命令都不会被执行。

例如：

```
echo -e "abc\ndef" | sed '/abc/a xyz'
abc
xyz
def
```

其实"a"、"i"和"c"命令的 TEXT 部分写法是比较复杂的，如果 TEXT 只是几个简单字符，如上即可。但如果要 TEXT 是分行文本，或者包含了引号，或者这几个命令是写在"{}"中的，

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

则上面的写法就无法实现。需要使用符号"\"来转义行尾符号，这表示开启一个新行，此后输入的内容都是 TEXT，直到遇到引号或者";"开头的行时。

例如，在 a.sh 的#!/bin/bash 行后添加一个注释行"# Script filename: a.sh"以及一个空行。由于是追加在尾部，所以使用"a"命令。

```
sed '\%#!/bin/bash%a\# Script filename: a.sh\n' a.sh
```

"a"命令后的第一个反斜线用于标记 TEXT 的开始，"\n"用于添加空白行。如果分行写，或者"a"命令写在大括号"{}"中，则格式如下：

```
sed '\%#!/bin/bash%a\
# Script filename: a.sh\n
' a.sh

sed '\%#!/bin/bash%{p;a\
# Script filename: a.sh\n
;p}' a.sh
```

最后需要说的是，**这 3 个命令的 TEXT 是存放在内存中的，不会进入模式空间，因此不受"-n"选项或某些命令的影响。此外，这 3 个命令依赖于输出流，只要有输出动作，不管是空输出流还是非空的输出流，只要有输出，这几个命令就会半路"劫杀"**。如果不理解这两句话，这 3 个命令的结果有时可能会比较疑惑。

例如，"a"命令是追加在当前匹配行行尾的，但为什么下面的"haha"却插入到匹配行"def"的前面去了呢？

```
echo -e "abc\ndef\nxyz" | sed '/def/{a\
haha
;N}'

abc
haha
def
xyz
```

阅读了下面的"N"命令之后，再回头看这个示例，应该能知道为什么。在 sed 修炼系列(四)：sed 中的疑难杂症中给出了解释。

(9).多行模式命令"N"、"D"、"P"简单说明。

在前面已经解释了"n"、"d"和"p"命令，sed 还支持它们的大写命令"N"、"D"和"P"。

- "N"命令：读取下一行内容追加到模式空间的尾部。其和"n"命令不同之处在于："n"命令会输出模式空间的内容(除非使用了"-n"选项)并清空模式空间，然后才读取下一行到模式空间，也就是说"n"命令虽然读取了下一行到模式空间，但模式空间仍然是单行数据。而"N"命令在读取下一行前，虽然也有自动输出和清空模式空间的动作，但该命令会把当前模式空间的内容**锁住**，使得自动输出的内容为空，也无法清空模式空间，然后读取下一行追加到当前模式空间中的尾部。追加时，原有内容和新读取内容使用换行符"\n"分隔，这样在模式空间中就实现了多行数据。即所谓的"多行模式"。 另外，当无法读取到下一行时(到了文件尾部)，将直接退出 sed 程序，使得"N"命令后的命令不会再执行，这和"n"命令是一样的。

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

- "D"命令：删除模式空间中第一个换行符"\n"之前的内容，然后立即回到 SCRIPT 循环的顶端，即进入下一个 SCRIPT 循环。如果"D"删除后，模式空间中已经没有内容了，则 SCRIPT 循环自动退出进入下一个 sed 循环；如果模式空间还有剩余内容，则继续从头执行 SCRIPT 循环。也就是说，"D"命令后的命令不会被执行。
- "P"命令：输出模式空间中第一个换行符"\n"之前的内容。

"N"、"D"和"P"命令作用非常大，它们是绝佳的组合命令，因为借助它们能实现"窗口滑动"技术，这对于复杂的文本行操作来说大有裨益。但显然，这不是本节的内容，在 sed 修炼系列(三)：sed 高级应用之实现窗口滑动技术中详细说明了这 3 个命令的功能。

此处按照惯例，还是给出它们的大致循环结构：其中"N"命令的 if 判断和前文的"n"一样，在本节结尾证明。

```
# "N"命令的大致循环结构
for ((line=1;line<=last_line_num;++line))
do
    read $line to pattern_space;
    while pattern_space is not null
    do
        execute cmd1 in SCRIPT;
        execute cmd2 in SCRIPT;
        ADDR1,ADDR2{            # "N" command
            if [ "$line" -ne "$last_line_num" ];then
                lock pattern_space;
                auto_print;
                remove_pattern_space;
                unlock pattern_space;
                append "\n" to pattern_space;
                read next_line to pattern_space;
            else
                auto_print;
                remove_pattern_space;
                exit;
            fi
        };
        ……
        auto_print;
        remove_pattern_space;
    done
done

# "D"命令的大致循环结构
for ((line=1;line<=last_line_num;++line))
do
    read $line to pattern_space;
    while pattern_space is not null
    do
        execute cmd1 in SCRIPT;
        execute cmd2 in SCRIPT;
        ADDR1,ADDR2{                # "D" command
            delete first line in pattern_space;
            continue;
        };
        ……
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

```
        auto_print;
        remove_pattern_space;
    done
done

# "P"命令的大致循环结构
for ((line=1;line<=last_line_num;++line))
do
    read $line to pattern_space;
    while pattern_space is not null
    do
        execute cmd1 in SCRIPT;
        execute cmd2 in SCRIPT;
        ADDR1,ADDR2{                # "P" command
            print first line in pattern_space;
        };
        ……
        auto_print;
        remove_pattern_space;
    done
done
```

(10).buffer 空间数据交换命令"h"、"H"、"g"、"G"、"x"简单说明。

sed 除了维护模式空间(pattern space)，还维护另一个 buffer 空间：保持空间(hold space)。这两个空间初始状态都是空的。

绝大多数时候，sed 仅依靠模式空间就能达到目的，但有些复杂的数据操作则只能借助保持空间来实现。之所以称之为保持空间，是因为它是暂存数据用的，除了仅有的这几个命令外，没有任何其他命令可以操作该空间，因此借助它能实现数据的持久性。

保持空间的作用很大，它和模式空间之间的数据交换能实现很多看上去不能实现的功能，是实现 sed 高级功能所必须的，例如"窗口滑动"。同样，这不是本节的内容。所以只简单解释这几个命令的作用：

- "h"命令：将当前模式空间中的内容覆盖到保持空间。
- "H"命令：在保持空间的尾部加上一个换行符"\n"，并将当前模式空间的内容追加到保持空间的尾部。
- "g"命令：将保持空间的内容覆盖到当前模式空间。
- "G"命令：在模式空间的尾部加上一个换行符"\n"，并将当前保持空间的内容追加到模式空间的尾部。
- "x"命令：交换模式空间和保持空间的内容。

注意，无论是交换、追加还是覆盖，原空间的内容都不会被删除。

5.总结

看到这里，对 sed 已经有了一些概念，也许已经发现了 sed 的重点在于各选项和各命令是如何影响 sed 循环以及 SCRIPT 循环的。确实如此，在 info sed 文档中，虽然没有将这些工作机制详细描述，但各选项各命令说明中，在需要的时候都提到了这些细节，而我所做

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

的只不过是将其系统性地描述出来、做一些深入，再给几个示例解释，并使用通俗易懂的循环结构来展示这些机制。

最后，验证前文"n"和"N"命令留下的疑问："n"和"N"命令是先判断是否还有下一行，再自动输出的。也就是证明下面两个判断语句采用前者还是后者的问题。

```
 ADDR1,ADDR2{                # "n" command
     if [ "$line" -ne "$last_line_num" ];then
         auto_print;
         remove_pattern_space;
         read next_line to pattern_space;
     else
         auto_print;
         remove_pattern_space;
         exit;
     fi
 };

 ADDR1,ADDR2{                # "n" command
         auto_print;
         remove_pattern_space;
         [ "$line" -ne "$last_line_num" ] && read next_line to
pattern_space || exit;
 };
```

虽然后者看上去代码更优化，但事实上采用的是前者。要证明这一点不太容易，好在我想出了下面的方法来证明。下面的示例中使用的是"N"，它和"n"在判断逻辑上的行为是一致的。

```
[root@xuexi ~]# echo -e "abc\ndef\nxyz" | sed '/def/{a\
haha
;N}'

abc
haha
def
xyz

[root@xuexi ~]# echo -e "abc\ndef" | sed '/def/{a\
haha
;N}'

abc
def
haha
```

在以上两个命令中，第一个命令"haha"是插入在匹配行"def"的前面，而第二个命令则是插入在"def"的后面。似乎根据"a"命令的作用来说，第二个命令才是意料之中的结果。

首先，解释第一个命令为何"haha"会出现在匹配行"def"的前面。当 sed 读取的行能匹配"def"时，将队列化"haha"到内存中，并在有输出流的时候追加到输出流尾部。由于这里的输出流来自于"a"命令后的"N"命令，该命令将模式空间锁住，使得隐含动作自动输出的内容为空，但队列化的内容还是发现了这个空输出流，于是追加在这个空流的尾部。再之后，"N"将下一行读取到模式空间中，到了 SCRIPT 循环的结尾，再次自动输出，此时模

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

式空间有两行："def" 和 "xyz"，这两行同时被输出。显然，在"def"被输出之前，队列化的内容已经随着空输出流而输出了。

再解释为何第二个命令的结果中"haha"在"def"之后，这也是待证明的疑问。第二个命令中，由于"def"已经是输入流的最后一行，"N"已经无法再读取下一行，于是输出当前模式空间内容并退出 sed 程序。假设，"n"或"N"命令是先自动输出、清空模式空间内容，再判断是否有下一行可读取的，那么在判断之前自动输出时，"N"不知道是否还有下一行，于是队列化的内容应该同第一个命令一样，插入在"def"之前。但结果却并非如此。如果先判断是否有下一行可供读取，再输出、清空模式空间，则队列化内容是跟随着"N"退出 sed 程序前输出的，这正符合第二个命令的结果。

博客园--骏马金龙

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

sed修炼系列(二)：武功心法(info sed翻译+注解)

1. 本节中的提到 GNU 扩展时，表示该功能是 GNU 为 sed 提供的(即 GNU 版本的sed 才有该功能)，一般此时都会说明：如果要写具有可移植性的脚本，应尽量避免在脚本中使用该选项。
2. 本节中的正则表达式几乎和 grep 中支持的一样。但还是有少数几个是 sed 自身才能解析的表达式。因此本译文中只对这些 sed 自身才支持的正则表达式做翻译，其余 sed 支持的通用性正则表达式见 grep 命令中文手册。
3. 此外，除了正则表达式部分，还有些地方没有进行翻译，因为个人觉得几乎用不上，没必要翻译。但为了保持文章的完整性，仍给出了原文内容。
4. 个人建议：如只想了解 sed 的简单用法，不建议阅读本译文；但如果想深入学习或掌握 sed，info sed 是最佳选择，很多处理机制在书上(即使是最为流行的《sed & awk》)都没有深入说明。本节为我第二次翻译，两次翻译过程中收获都极大。
5. 译文中有些地方加上了"(注：)"，为本人自行加入，助于理解和说明，非原文内容。PS：直到翻译完，才发现加了很多很多我个人注释，尽管是一篇译文，但也舍不得删掉这些感想。如果这些"注："有碍观感，还望各位能体谅。
6. 学习 sed 的过程中，推荐使用"sedsed"调试工具，这对于分析 sed 处理过程以及 pattern space、hold space 有很大帮助。但极少数情况下，它的结果可能并不如想象中那样准确。

本节目录:

1. Introduction(简介)

sed 是一个流式编辑器。流式编辑器用于对输入流(文件或管道传递的数据)执行基本的文本转换操作。在某些方面上，sed 有点类似于脚本化编辑的编辑器(如 ed)，但 sed 只能**通过一次输入流**，因此它的效率要更高。但 sed 区别于其他类型编辑器的地方在于它能筛选过滤管道传递的文本数据。
(注：sed 只能通过一次输入流，意味着每次的输入流只能处理一次，因此像`(sed -n '2{p;q}';sed -n 3{p;q}) <filename` 这样的命令中，第二个 sed 语句读取的输入流是空流。)

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

2. Invocation(调用方式)

通常，会使用下面的方式来调用 sed：

```
sed SCRIPT INPUTFILE...
```

完整的调用格式为：

```
sed OPTIONS... [SCRIPT] [INPUTFILE...]
```

如果不指定 INPUTFILE 或者指定的 INPUTFILE 为"-"，sed 将从标准输入中读取输入流并进行过滤。SCRIPT 是第一个非选项参数，sed 仅在没有使用"-e"或"-f script_file"选项时才将其当作是 script 部分而非输入文件。

可以使用下面的命令行选项来调用 sed：

'--version'
输出 sed 的版本号并退出。

'--help'
输出 sed 命令行的简单帮助信息并退出。

'-n'
'--quiet'
'--silent'
默认情况下，sed 将在**每轮 script 循环结束**(How 'sed' works: Execution Cycle)时自动输出模式空间中的内容。该选项禁止自动输出动作，这种情况下，只有显式通过"p"命令来产生对应的输出。

'-e SCRIPT'
'--expression=SCRIPT'
向 SCRIPT 中添加命令集，让 sed 根据这些命令集来处理输入流。(注：其实就是指定 pattern 和对应的 command)

'-f SCRIPT-FILE'
'--file=SCRIPT-FILE'
指定包含 command 集合的 script 文件，让 sed 根据 script 文件中的命令集处理输入流。

'-i[SUFFIX]'
'--in-place[=SUFFIX]'
该选项指定要将 sed 的输出结果保存(覆盖的方式)到当前编辑的文件中。GNU sed 是通过创建一个临时文件并将输出写入到该临时文件，然后重命名为源文件来实现的。

该选项隐含了"-s"选项。

当到达了文件的尾部，临时文件被重命名为**源文件**的名称。如果还提供了 SUFFIX，则在重命名临时文件之前，先使用该 SUFFIX 先修改源文件名，从而生成一个文件备份。
(注：**临时文件总是会被重命名为源文件名称**，也就是说 sed 处理完全结束后，仍使用源文件名的文件是 sed 修改后的文件。文件名中包含了 SUFFIX 的文件则是最原始文件的备

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

份。例如源文件为 a.txt，`sed -i'.log' SCRIPT a.txt` 将生成两个文件：a.txt 和 a.txt.log，前者是 sed 修改后的文件，a.txt.log 是源 a.txt 的备份文件。)

规则如下：如果扩展名不包含符号"*"，将 SUFFIX 添加到原文件名的后面当作文件后缀；如果 SUFFIX 中包含了一个或多个字符"*"，则每个"*"都替换为原文件名。这使得你可以为备份文件添加一个前缀，而不是后缀，甚至可以将此备份文件放在在其他已存在的目录下。

如果没有提供 SUFFIX，源文件被覆盖，且不会生成备份文件。

`'-l N'`
`'--line-length=N'`
为"l"命令指定默认的换行长度。N=0 意味着完全不换行的长行，如果不指定，则 70 个字符就换行。

`'--posix'`
GNU 'sed' includes several extensions to POSIX sed. In order to simplify writing portable scripts, this option disables all the extensions that this manual documents, including additional commands. Most of the extensions accept 'sed' programs that are outside the syntax mandated by POSIX, but some of them (such as the behavior of the 'N' command described in *note Reporting Bugs::) actually violate the standard. If you want to disable only the latter kind of extension, you can set the 'POSIXLY_CORRECT' variable to a non-empty value.

`'-b'`
`'--binary'`
This option is available on every platform, but is only effective where the operating system makes a distinction between text files and binary files. When such a distinction is made--as is the case for MS-DOS, Windows, Cygwin--text files are composed of lines separated by a carriage return *and* a line feed character, and 'sed' does not see the ending CR. When this option is specified, 'sed' will open input files in binary mode, thus not requesting this special processing and considering lines to end at a line feed.

`'--follow-symlinks'`
该选项只在支持符号连接的操作系统上生效，且只有指定了"-i"选项时才生效。指定该选项后，如果 sed 命令行中指定的输入文件是一个符号连接，则 sed 将对该符号链接的目标文件进行处理。默认情况下，禁用该选项，因此不会修改链接的源文件。

`'-r'`
`'--regexp-extended'`
使用扩展正则表达式，而不是使用默认的基础正则表达式。sed 所支持的扩展正则表达式和 `egrep` 一样，使用扩展正则表达式显得更简洁，因为有些元字符不用再使用反斜线"\"，但这是 GNU 扩展功能，因此应避免在可移植性脚本中使用。

`'-s'`
`'--separate'`
默认情况下，sed 会将命令行中指定文件的所有行当作一个长输入流。此选项为 GNU sed 扩展功能，指定该选项后，sed 将认为命令行中给定的每个文件都是独立的输入流。因此**此时范围定址(如/abc/,/def/)无法跨越多个文件，行号也会在处理每个文件时重置，"$"代表的是每个文件的最后一行，"R"命令调用的文件将绕回到每个文件的开头。**

`'-u'`
`'--unbuffered'`

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

使用尽量少的空间缓冲输入和输出行。(该选项在某些情况下尤为有用，例如输入流的来源是"tail -f 时，指定该选项将可以尽快返回输出结果。)

'-z'
'--null-data'
'--zero-terminated'
以空串符号"\0"而不是换行符"\n"作为输入流的行分隔符。

如果未给定"-e"、"-f"、"--expression"或"--file"选项，则命令行中的第一个非选项参数将被认为是 SCRIPT 被执行。

如果命令行中在上述选项后还加了任意参数，则都将被当作输入文件。其中"-"表示的是标准输入流。如果没有给定任何输入文件，则默认从标准输入中读取输入里流。

3. 'sed' Programs(sed 程序)

sed 程序由一个或多个 sed 命令组成(注：请勿将 sed 这个工具理解为 sed 命令，而应该看作是一个包含很多命令的程序，所以后文中"sed 命令"表示的是 sed 中的子命令，而非 sed 本身这个命令)，这些命令通过"-e"、"-f file"传递，或者当没有指定这两个选项时，通过第一个非选项参数传递。这些传递的命令集合称为 sed 的执行脚本(SCRIPT)，命令的执行顺序按照命令行中给定的顺序依次执行。

在 SCRIPT 或 SCRIPT-FILE 中的多个命令可以使用分号";"进行分隔，或者换行书写。但有些命令，由于它们的语法问题，导致不支持使用分号作为命令分隔符，因此只能换行书写，除非它们是 SCRIPT 或 SCRIPT-FILE 中的最后一个命令。允许命令的前面有空白字符。

每个 sed 命令由地址或范围地址，随后紧跟单字符代表的命令名称组成，其中地址部分可以省略。

3.1 How 'sed' Works(sed 是如何工作的)

sed 维护两个数据缓冲空间：一直处于活动状态的模式空间(pattern space)和辅助性的保持空间(hold space)。这两个空间初始时都为空。

sed 通过执行下面的循环来操作输入流中的每一行： 首先，sed 读取输入流中的一行，移除该行的尾随换行符，并将其放入到 pattern space 中。然后对 pattern space 中的内容执行 SCRIPT 中的 sed 命令，每个 sed 命令都可以关联一个地址：地址是一种条件判断代码，只有符合条件的行才会执行相应的命令。当执行到 SCRIPT 的尾部时，除非指定了 "-n"选项，否则 pattern space 中的内容将写到标准输出流中，并添加回尾随的换行符。然后进入下一个循环开始处理输入流中的下一行。

除非指定了特殊的命令(例如"-D")，否则 pattern space 中的内容在 SCRIPT 循环结束后会被删除。但是 hold space 会保留其中的内容(参见 sed 命令'h', 'H', 'x', 'g', 'G'以获知两个 buffer 空间是如何互相移动数据的)。

(

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

注：也就是说，sed 程序工作时包含内外两个循环：内循环是 SCRIPT 循环，循环的过程是对模式空间中的内容执行 SCRIPT 中的命令集合，还包括一个隐含的自动输出动作用于输出模式空间的内容到标准输出中；外循环是 sed 循环，读取下一行到模式空间中，并执行 SCRIPT 循环，SCRIPT 循环结束后再读取下一行到模式空间中。使用编程结构描述，结构如下：

```
for ((line=1;line<=last_line_num;++line))
do
    read $line to pattern_space;
    while pattern_space is not null
    do
        execute cmd1 in SCRIPT;
        execute cmd2 in SCRIPT;
        ……
        auto_print;
        remove_pattern_space;
    done
done
```

一般情况下，SCRIPT 循环都只执行一轮就退出并进入外层 sed 循环，因为执行完一次 SCRIPT 循环，pattern space 就清空了，但有些特殊命令(如"D"命令)会进入多行模式，使得 SCRIPT 循环结束时将数据锁在 pattern space 中不输出也不清空，甚至在 SCRIPT 循环还没结束时就强行进入下一轮 SCRIPT 循环，在《sed & awk》一书中，这种行为被称之为"回到 SCRIPT 的顶端"。其实就相当于在上面的 while 循环结构中加上了"continue"关键字。此外还有命令(如"d")可以直接退出 SCRIPT 循环进入下一个 sed 循环，就像是在 while 循环中加上了"break"一样。甚至还有直接退出 sed 循环的命令(只有 2 个这样的命令："q"和"Q")，就像是加上了"exit"一样。

auto_print 和 remove_pattern_space 动作是隐含动作，分别称之为自动输出和清空 pattern space。"-n"选项禁用的自动输出不是禁止隐含动作 auto_print，而是让其输出空内容(它们是有区别的)，并执行 remove_pattern_space。只要没有指定"-n"选项，在 SCRIPT 循环结束时一定输出 pattern space 中的全部内容并清空 pattern space，只不过某些特殊的命令可以将 pattern space 中的内容锁住使 auto_print 输出空内容，也使得 remove_pattern_space 移除不了 pattern space 的内容。
之所以要特地指出"输出空内容"并标明它和禁止输出动作，是因为某些命令(如"a"、"i"、"c"命令)依赖于输出流，没有输出动作就没有输出流，这些命令就无法正确完成。

这几个循环、几个动作的细节非常重要，虽不影响后文单个选项或命令的理解，但如果将命令结合，有时候的结果很可能会出人意料，而 sed 难就难在命令的合理组合。例如下面的命令，"a"命令本来是要将 xyz 插入到 aaa 所在行后面的，但结果却插在了 aaa 行的前面。

```
echo -e "aaa\nbbb" | sed '/aaa/{a\
> xyz
> ;N}'
xyz
aaa
bbb
```

)

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

3.2 Selecting lines with 'sed'(sed 定址：筛选行的方式)

(注：sed SCRIPT 中的命令由单字符代表的命令和地址组成，地址的作用是匹配当前正在处理的行，如果能匹配成功，就执行与该地址相关联的命令。地址由定址表达式决定，定址的结果可能是单行，可能是一个范围，省略定址表达式时表示所有行。)

可以使用下面任意一种方式进行定址：

'NUMBER'
指定一个行号，sed 将仅只匹配该行。(需要注意，除非使用了"-s"或"-i"选项，sed 将对所有输入文件的行连续计数。)

'FIRST~STEP'
这是 GNU sed 的功能，FIRST 和 STEP 都是非负数。该定址表达式表示从第 FIRST 行开始，每隔 STEP 行就再取一次。也就是取行号满足“FIRST+(N*STEP)” (其中 N>=0)的行。因此，要选择所有的奇数行，使用“1~2”；要从第 2 行开始每隔 3 行取一次，使用“2~3”；要从第 10 行开始每隔 5 行取一次，使用“10~5”；而“50~0”则表示只取第 50 行。

'$'
该符号匹配的是最后一个文件的最后一行，如果指定了"-i"或"-s"，则匹配的是每个文件的最后一行。
(注：总之，"$"匹配的是每个输入流的最后一行)

'/REGEXP/'
该定址表达式将选择那些能被正则表达式 REGEXP 匹配的所有行。如果 REGEXP 中自身包含了字符"/"，则必须使用反斜线进行转义，即"\/"。

空的正则表达式"//"表示引用最后一个正则表达式匹配的结果(命令"s"中的正则匹配部分如果是空正则"//"，则一样如此处理)。注意，正则表达式的修饰符(即下文中的"I"和"M"，以及 s 命令中的修饰符"i"、"I"和"M")是在正则表达式编译完成之后(注：进行数据匹配时)才生效的，因此，这些修饰符和空正则一起使用时无效(注：会报错，提示"cannot specify modifiers on empty regexp")。

(注：这里的修饰符特指定址时可用的修饰符，即 I 和 M。命令 s 也有修饰符"i"、"I"和"M"。
如：sed '/hold/Is//gogogo/g'能成功，因为第一个定址正则表达式的修饰符"I"的对象是非空集"hold"，第二个正则模式"//"没有指定"i"、"I"或"M"修饰符，所以成功。但 sed '/hold/Is//gogogo/gi'会报错，因为在正则编译结束后还未开始进行匹配的时候，第二个正则表达式中的修饰符"i"的对象是空集。sed '/hold/I{//Mp}'和 sed '/hold/I{//Ip}'也都是失败的，原因都是第二个正则的修饰符对象是空集。

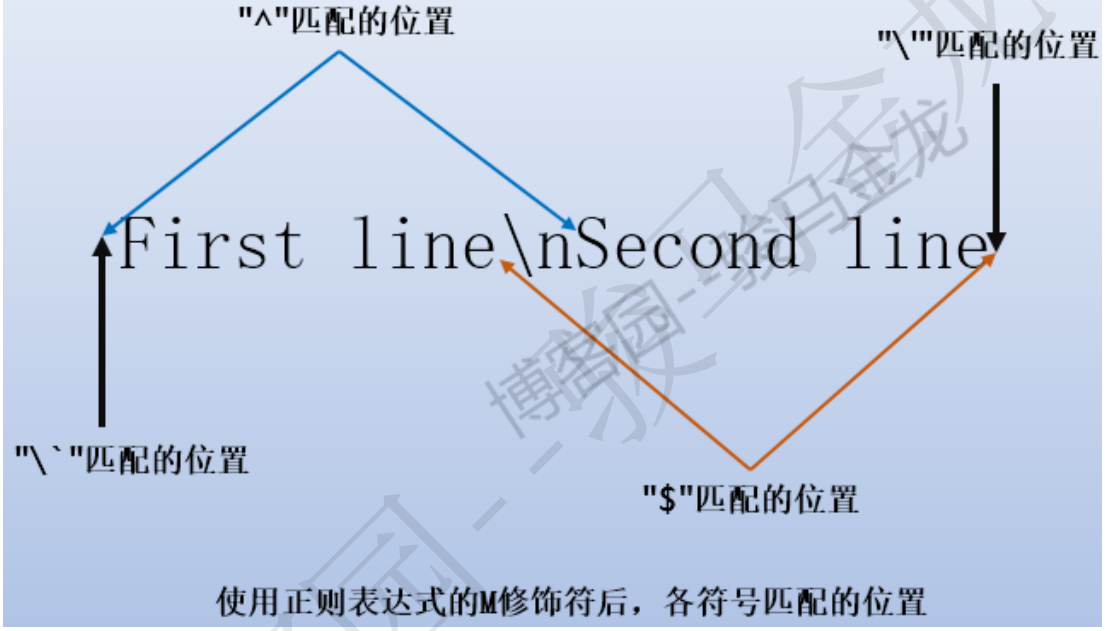
'\%REGEXP%'
('%'可以使用其他任意单个字符替换。)

这和上一个定址表达式的作用是一样的，只不过是使用符号"%"替换了符号"/"。当 REGEXP 中包含"/"符号时，使用该定址表达式就无需对"/"使用反斜线"\"转义。但如果此时 REGEXP 中包含了"%"符号时，该符号需要使用"\"转义。
总之，定址表达式中使用的分隔符在 REGEXP 中出现时都需要使用反斜线转义。

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

'/REGEXP/I'
'\%REGEXP%I'
正则表达式的修饰符"I"是 GNU 的扩展功能，表示 REGEXP 在匹配时不区分大小写。

'/REGEXP/M'
'\%REGEXP%M'
正则表达式的修饰符"M"是 GNU 的扩展功能，这可以让 sed 直接匹配多行模式下(multi-line mode)的位置。该修饰符使得正则表达式的元字符"^"和"$"匹配分别匹配每个新行后的空字符和新行前的空字符。还有另外两个特殊的字符序列(\`和\'，分别是反斜线加反引号，反斜线加单引号)，它们总是匹配 buffer 空间中的起始和结束位置。此外，元字符点"."不能匹配多行模式下的换行符。
(注：在单行模式下，使用 M 和不使用 M 是没有区别的，但在多行模式下各符号匹配的位置将和通常的正则表达式元字符匹配的内容不同，各符号的匹配位置如下图所示)



如果没有给定地址，则会匹配输入流中的所有行，如果给定了单地址的定址表达式，则这些行会在模式空间中被匹配条件进行匹配。

可以使用逗号分隔的两个地址表达式(ADDR1,ADDR2)来描述范围地址。范围地址表示从符合第一个地址条件的行开始匹配，直到符合第二个地址的行结束。

(注：需要注意，无论行是否符合地址匹配条件，它们都会被读入 pattern space，只不过读入的行如果不符合地址匹配条件，将直接执行 SCRIPT 循环中的隐含动作，并结束 SCRIPT 循环继续读取输入流的下一行)

如果第二个定址表达式是正则表达式 REGEXP，将从符合第一个定址表达式的行开始逐行检查，以匹配范围定址的结束行：范围定址的范围总是会跨越至少两行(当然，如果输入流结束的情形除外)。

如果第二个定址表达式是一个小于或等于第一个表达式代表的行号值，则只会匹配第一个表达式搜索到的那一行。
(注：即如果范围地址的结束行在起始行的前面，则只会匹配起始行。)

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

GNU sed 还支持以下几种特殊的两地址格式，这些都是 GNU 的扩展功能：

`'0,/REGEXP/'`

使用行号 0 作为起始地址也是支持的，就像此处的"0,/REGEXP/"，这时 sed 会尝试对第一行就匹配 REGEXP。换句话说，"0,/REGEXP/"和"1,/REGEXP/"基本是相同的。但以行号 0 作为起始行时，如果第一行就能被 ADDR2 匹配，范围搜索立即就结束，因为第二个地址已经搜索到了；如果以行号 1 作为起始行，会从第二行开始匹配 ADDR2，直到匹配成功。

注意，0 地址只有在这一种情况下才是有意义的，实际上并没有第 0 行，在其他任何时候使用行号 0 都会给出错误提示。

`'ADDR1,+N'`
匹配 ADDR1 和其后的 N 行。

`'ADDR1,~N'`
匹配 ADDR1 和其后的行直到出现 N 的倍数行，倍数可为随意整数倍。(注：可以是任意倍，只要 N 的倍数是最接近且大于 ADDR1 的即可。如 `ADDR1=1,N=3` 匹配 1 到 3 行，`ADDR1=5,N=4` 匹配 5-8 行。而"1,+3"匹配的是第一行和其后的 3 行即 1-4 行。)

在地址的后面追加"!"符号指定反转匹配的含义。意思是，如果"!"跟在范围定址的后面，则那些匹配的行将不被选择，而是不匹配的行被选择。同样可以适用于单个定址甚至于有悖常理的空定址。

(注：例如，`sed -n '3!p' INPUTFILE`，`sed -n '3,5!p' INPUTFILE`，甚至是 `sed -n '!p' INPUTFILE`)

(注：

- **sed 采用计数器计算，每读取一行，计数器加 1，直到最后一行。因此在读到最后一行前，sed 是不知道这次输入流中总共有多上行，也不知道最后一行是第几行。"$"符号表示最后一行，它只是一个特殊的标识符号。当 sed 读取到输入流的尾部时，sed 就会为该行打上该标记。无法使用"$"参与数学运算，例如无法使用$-1表示倒数第二行，因为 sed 在读到最后一行前，它并不知道这是倒数第二行，此时也还没打"$"标记，因此$-1 是错误的定址表达式。另一方面，在本译文的最开始就说明了，sed 只能通过一次输入流，这意味着已经读取过的行无法再次被读取，所以 sed 不提供往回取数据的定址表达式，上面的几个定址表达式也确实证明了这一点。**事实上，sed 中根本就无法使用"-"做减法或取负数，因为语法不支持。
- 范围匹配的是 pattern space 中的内容，对于 `regexp1,regexp2` 这样的范围定址自然容易理解，但如果是 `num1,num2` 这样的范围定址，它可能和想象中的匹配方式不一样。**每读取一行，sed 内部的行号计数器都会"+1"，所以行号值总是记录在内存中。当进行行号匹配时，其本质是和内存中当前计数器的值进行匹配，如果当前计数器的值在范围内，则能被匹配，否则无法匹配。**因此，从 hold space回到 pattern space 的行或者被修改的行甚至是被删除过的行，不管这一行的内容在 pattern space 中是否存在，只要当前计数器值在范围内，就能匹配。例如多行模式：

```
echo -e "abc\nfgh" | sed -n 'N;1p'
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

该命令不输出任何内容。虽然 N 读取下一行后，pattern space 两行中的第一行一直原封不动的放在那，没做任何处理，但"1p"根本就无法匹配这一行，因为当前计数器的值为 2。

)。

3.3 Overview of Regular Expression Syntax(正则表达式一览)

(注：sed 解析、编译和匹配正则表达式的引擎和 grep 的引擎一样，因此本节基本不做翻译，可以参考我对 info grep 的译文：grep 命令中文手册。)

To know how to use 'sed', people should understand regular expressions ("regexp" for short). A regular expression is a pattern that is matched against a subject string from left to right. Most characters are "ordinary": they stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern

```
The quick brown fox
```

matches a portion of a subject string that is identical to itself. The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of "special characters", which do not stand for themselves but instead are interpreted in some special way. Here is a brief description of regular expression syntax as used in 'sed'.

'CHAR'
A single ordinary character matches itself.

'*'
Matches a sequence of zero or more instances of matches for the preceding regular expression, which must be an ordinary character, a special character preceded by '\', a '.', a grouped regexp (see below), or a bracket expression. As a GNU extension, a postfixed regular expression can also be followed by '*'; for example, 'a**' is equivalent to 'a*'. POSIX 1003.1-2001 says that '*' stands for itself when it appears at the start of a regular expression or subexpression, but many nonGNU implementations do not support this and portable scripts should instead use '*' in these contexts.

'\+'
As '*', but matches one or more. It is a GNU extension.

'\?'
As '*', but only matches zero or one. It is a GNU extension.

'\{I\}'
As '*', but matches exactly I sequences (I is a decimal integer; for portability, keep it between 0 and 255 inclusive).

'\{I,J\}'
Matches between I and J, inclusive, sequences.

'\{I,\}'
Matches more than or equal to I sequences.

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

'\(REGEXP\)'
Groups the inner REGEXP as a whole, this is used to:

- Apply postfix operators, like '\(abcd\)*': this will search for zero or more whole sequences of 'abcd', while 'abcd*' would search for 'abc' followed by zero or more occurrences of 'd'. Note that support for '\(abcd\)*' is required by POSIX 1003.1-2001, but many non-GNU implementations do not support it and hence it is not universally portable.
- Use back references (see below).

'.'
Matches any character, including newline.

'^'
Matches the null string at beginning of the pattern space, i.e. what appears after the circumflex must appear at the beginning of the pattern space.

In most scripts, pattern space is initialized to the content of each line (*note How 'sed' works: Execution Cycle.). So, it is a useful simplification to think of '^#include' as matching only lines where '#include' is the first thing on line--if there are spaces before, for example, the match fails. This simplification is valid as long as the original content of pattern space is not modified, for example with an 's' command.

'^' acts as a special character only at the beginning of the regular expression or subexpression (that is, after '(' or '|'). Portable scripts should avoid '^' at the beginning of a subexpression, though, as POSIX allows implementations that treat '^' as an ordinary character in that context.

'$'
It is the same as '^', but refers to end of pattern space. '$' also acts as a special character only at the end of the regular expression or subexpression (that is, before ')' or '|'), and its use at the end of a subexpression is not portable.

'[LIST]'
'[^LIST]'
Matches any single character in LIST: for example, '[aeiou]' matches all vowels. A list may include sequences like 'CHAR1-CHAR2', which matches any character between (inclusive) CHAR1 and CHAR2.

A leading '^' reverses the meaning of LIST, so that it matches any single character *not* in LIST. To include ']' in the list, make it the first character (after the '^' if needed), to include '-' in the list, make it the first or last; to include '^' put it after the first character.

The characters '$', '*', '.', '[', and '\' are normally not special within LIST. For example, '[*]' matches either '\' or '*', because the '\' is not special here. However, strings like '[.ch.]', '[=a=]', and '[:space:]' are special within LIST and represent collating symbols, equivalence classes, and character classes, respectively, and '[' is therefore special within LIST when it is followed by '.', '=', or ':'. Also, when not in 'POSIXLY_CORRECT' mode, special escapes like '\n' and '\t' are recognized within LIST. *Note Escapes::.

'REGEXP1\|REGEXP2'
Matches either REGEXP1 or REGEXP2. Use parentheses to use complex alternative regular expressions. The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. It is a GNU extension.

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

'REGEXP1REGEXP2'
Matches the concatenation of REGEXP1 and REGEXP2. Concatenation binds more tightly than '|', '^', and '$', but less tightly than the other regular expression operators.

'\DIGIT'
Matches the DIGIT-th '(...)' parenthesized subexpression in the regular expression. This is called a "back reference". Subexpressions are implicity numbered by counting occurrences of '(' left-to-right.

'\n'
匹配换行符。

'\CHAR'
Matches CHAR, where CHAR is one of '$', '*', '.', '[', '\', or '^'. Note that the only C-like backslash sequences that you can portably assume to be interpreted are '\n' and '\\'; in particular '\t' is not portable, and matches a 't' under most implementations of 'sed', rather than a tab character.

注意，sed 支持的正则表达式是贪婪匹配的。例如，从行左向行右匹配时，如果满足匹配的匹配字符有多个，则会取最长的。(注：例如字符串'abbbc'，给定正则表达式"ab*"，满足该正则的字符串序列有"a"，"ab"，"abb"以及"abbb"，由于是贪婪匹配，它会取最长的，即"abbb")

示例：

'abcdef'
匹配字符串"abcdef"。

'a*b'
匹配 0 或多个 a，但后面需要跟上字符 b，例如'b'或'aaaaab'。

'a\?b'
匹配"b"或"ab"。

'a\+b\+'
匹配一个或多个 a，且后面需要有一个或多个 b，所以"ab"是最短的匹配序列，其他的如'aaaab'、'abbbbb'或'aaaaaabbbbbbb'都能被匹配。

'.*'
'.\+'
都匹配所有字符；但是，第一个匹配任何字串(包含空字串)，而第二个只匹配包含至少一个字符的字串(空格也是字符)。
(注：即第一种会匹配所有，包括 null 字符，第二种不能匹配 null 字串。例如：sed -n '/.*/p'会匹配空行。sed -n '/.\+/p'不会匹配空行，这是匹配非空行的一种简便方法。)

'^main.*(.*)'
匹配以"main"开头的行，且该行还包括括号。

'^#'
匹配以"#"开头的行。

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

'\\$'
匹配以反斜线结尾的行。

'\$'
匹配包含美元符号的行。

'[a-zA-Z0-9]'
在 C 字符集环境下，这匹配任意 ASCII 的字母和数字。

'[^ tab]\+'
(此处 tab 代表一个制表符)匹配不包含空格和制表符的行。通常用于匹配整个单词。

'^\(.*\)\n\1$'
匹配相邻两行完全相同的情况。

'.\{9\}A$'
匹配 9 个任意字符，后面还带一个字母 A。

'^.\{15\}A'
匹配以任意 15 个字符开头但第 16 个字符是 A 的行。

3.4 Often-Used Commands(sed 常用命令)

如果要掌握 sed，下面几个命令几乎是必须掌握的。

'#'
不能跟在定址表达式后。

以"#"开头的行表示注释行。

如果要考虑可移植性，需要注意有些 sed 可能仅支持一条单行注释，且此时"#"必须是 SCRIPT 中的第一个字符。

注意：如果 SCRIPT 中的前两个字符是"#n"，则表示强制开启选项"-n"，而非注释。如果想要开启一个注释行来注释以字母"n"开头的字符串，那么需要使用大写字母 N 代替，或者"#"和"n"之间加上一个或多个空白字符。

'q [EXIT-CODE]'
该命令只能在单定址表达式下使用。

表示立即退出 sed，而不再处理后续的命令和输入流。注意，退出 sed 前，当前 pattern space 中的内容会被输出，除非使用了"-n"选项。返回一个退出状态码是 GNU sed 的扩展功能。(注：后文还有一个"Q"命令，和"q"命令作用相同，但退出 sed 前不输出当前 pattern space 中的内容。)

'd'
删除模式空间中的内容，并立即进入下一个 sed 循环(注：即以"break"的方式直接退出当前 SCRIPT 循环，并读取输入流中的下一行，再执行 SCRIPT 循环)。

'p'
输出当前模式空间的内容(到标准输出中)。该命令一般只和"-n"选项一起使用。

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

'n'
输出模式空间的内容(除非自动输出功能被禁止)，然后读取下一行到模式空间中替换其中的内容。如果输入流中没有下一行供"n"读取(注：即此前已经读取了输入流的最后一行)，sed 将直接退出，不会执行 SCRIPT 中后续的命令。

(

注：以下面两个命令为例：其中"i"命令为插入字符串并输出。

```
[root@xuexi ~]# echo -e "line1\nline2\nline3\nline4" | sed -n 'n;p;i
aaa'
line2
aaa
line4
aaa
[root@xuexi ~]# echo -e "line1\nline2\nline3" | sed -n 'n;p;i aaa'
line2
aaa
```

第二个命令读取了第三行到 pattern space 后执行 n 命令，但此时输入流中已经没有下一行供读取，于是直接结束，链后续的"p"命令都没有执行。

)

'{ COMMANDS }'
使用大括号将一系列命令组合成一个命令组。在某些时候特别有用，例如相对某一匹配行或某一范围中的行做多次处理时。

3.5 The 's' Command(sed 的 s 命令)

sed 的"s"命令(该命令用于字符串替换)的语法格式为"s/REGEXP/REPLACEMENT/FLAGS"。"s"命令中的字符"/"可以统一被替换成任意其他单个字符(注：在定址正则表达式中斜杠也可以被替换成其他字符，但需要在第一个被替换字符前加上反斜线转义，例如\%REGEXP%，而 s 命令中替换时无需加反斜线)。如果斜线字符"/"(或被替换后的其他字符)需要出现在 REGEXP 或 REPLACEMENT 中，必须使用反斜线进行转义。

sed 的"s"命令算的上是 sed 程序中最重要的命令，它有很多不同的选项。但它的基本概念很简单："s"命令使用 REGEXP 匹配 pattern space 中的内容，如果匹配成功，则匹配成功的那部分字符串被替换为 REPLACEMENT。

REPLACEMENT 中可以使用"\N"(N 是从 1 到 9 的整数)进行后向引用，所代表的是 REGEXP 第 N 个括号\(...\)中匹配的内容。另外，REPLACEMENT 中可以包含未转义的"&"符号，这表示引用 pattern space 中被匹配的整个内容(注：是 pattern space 中的所有匹配，不仅仅只是括号的分组匹配)。最后，GNU sed 为 REPLACEMENT 还提供了一些"反斜线加单字母"的特殊字符序列，如下：

'\L'
将 REPLACEMENTE 转换成小写，直到遇到了"\U"或"\E"。

'\l'
将 REPLACEMENTE 中下一个字符转换成小写。

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

'\U'
将 REPLACEMENTE 转换成大写，直到遇到了"\L"或"\E"。

'\u'
将 REPLACEMENT 中下一个字符转换成大写。

'\E'
停止"\L"和"\E"开启的大小写转换。

(

注：使用方法如下示例

```
shell> echo hello | sed 's/e/ \Uyour\Lname /'
h YOURname llo
```

)

当使用了"g"修饰符时，大小写转换不会从一个正则匹配事件扩展到另一个正则匹配事件。例如，如果 pattern space 中的内容为"a-b-"，执行下面的命令：

```
s/\(b\?\)-/x\u\1/g
```

得到的输出为"axxB"。当替换第一个"-"时，"\u"只影响"\1"代表的空值。当替换"b-"时，由于"\1"代表的是"b-"，所以第一个字符"b"会被替换为"B"，但添加到 pattern space 中的"x"仍然为小写。

另一方面，"\1"和"\u"会影响 REPLACEMENT 中的空引用的后一个字符。例如：模式空间内容为"a-b-"，执行下面的命令：

```
s/\(b\?\)-/\u\1x/g
```

将使用"X"(大写)替换"-"，使用"Bx"替换"b-"。如果这样的结果不是你想要的结果，可以在此例中的"\1"后加上"\E"防止"x"被转换。

如果想要在最后的 REPLACEMENT 中包含字面符号"\"、"&"或换行符，需要在 REPLACEMENT 中这些字符之前加上反斜线。

sed 的"s"命令可以接 0 或多个如下列出的修饰符(FLAGS)：

'g'
使用 REPLACEMENT 替换所有被 REGEXP 匹配的内容，而非第一次被匹配到的。

(注：**sed 任何动作都是在 pattern space 进行的，字符串替换也如此。**不加"g"修饰符时，将只 pattern space 中第一个被匹配到的内容，加上"g"，将替换 pattern space 中所有被匹配到的内容，因此如果是多行工作模式，第二行或更多行只要能被匹配都会被替换。)

'NUMBER'
为一个整数值 N。表示只替换第 N 个被匹配到的内容。

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

注意：POSIX标准中没有说明既指定"g"又指定"NUMBER"修饰符时会如何处理，并且当前各种 sed 的实现也没有标准说法。对于 GNU sed 而言，定义如下：忽略第 NUMBER 个匹配前的所有匹配，然后从第 NUMBER 个匹配开始向后重新匹配并替换所有匹配成功的内容。

'p'
替换动作完成后打印新的模式空间中的内容。

注意：当既指定"p"又指定"e"命令时，它们的前后顺序不同，会得到两种不同结果。一般来说，"ep"这种顺序得到的结果可能是所期望的结果，但另一种顺序"pe"对调试很有用。出于这个原因，当前版本的 GNU sed 特地解释了"p"命令在"e"命令前(或后)时，将在"e"命令生效前(或后)输出内容，因为"s"命令的每个修饰符一般都只展示一次结果。虽然这种行为已经写入文档，但未来可能会改变。

'w FILE-NAME'
该子命令表示将模式空间的内容写入到指定的文件 FILE-NAME 中。GNU sed 支持两个特殊的 FILE-NAME："/dev/stderr"和"/dev/stdout"，分别表示写入到标准错误和标准输出中。

'e'
该命令允许通过管道将 shell 命令的执行结果直接传递到 pattern space 中。当替换动作完成后，将搜索 pattern space，发现的命令会被执行，并且执行结果覆盖到当前 pattern space 中。它会禁用尾随换行符，并且如果待执行命令中包含了 NULL 字符，该命令将不被定义为命令，即表示它是普通字符而不是命令所以不执行。这是 GNU sed 的功能。

(

注："s"命令的"e"修饰符似乎只有搜索 pattern space 并找出其中的命令并执行的功能，没有选项描述中第一句话所述的功能。但"e"命令有该功能，例如：

```
echo -e "good\nbad" | sed 'e echo haha'
haha
good
haha
bad
```

不讨论 e 命令的用法，关于"s"命令的"e"修饰符的用法如下：

文件 ttt.txt 的内容如下：

```
[root@xuexi tmp]# cat ttt.txt
ls /tmp
haha
excute a command
```

将第二行的"haha"替换成一个命令后使用"e"修饰符，那么在替换完成后会查找模式空间，如果找到了可以执行的命令就执行。

```
[root@xuexi tmp]# sed 's/haha/du -sh \/tmp/ge' ttt.txt
ls /tmp         # 注意这一行没有执行
18M     /tmp    # 说明执行了 du -sh /tmp 的命令
excute a command
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

注意到第一行虽然也是可以执行的命令，但是却没有执行，因为"e"是"s"命令的修饰符，需要成功匹配"haha"的行才能符合"e"修饰符。
注意模式空间中的内容是一行一行的，命令将把整行内容作为命令行。例如，如果只将 excute 替换成 du -sh /tmp，那么模式空间中的内容将是"du -sh /tmp a command"，它会对/tmp 和当前目录下的"a 和"command"目录进行"du -sh"统计，但很可能"a"或"command"目录根本不存在，这时就会报错。

```
[root@xuexi tmp]# sed 's%excute%du -sh /tmp%ge' ttt.txt
ls /tmp
haha
du: cannot access 'command': No such file or directory  # 不存在 command 目录所以错误
18M     /tmp
4.0K    a       # 当前目录下正好存在 a 目录，所以统计了信息
```

并且如果替换后找到的命令不在行首(有前导空白没有影响)，将出现错误，因为是将整行作为命令来执行的。

```
[root@xuexi tmp]# sed 's%command%du -sh /tmp%ge' ttt.txt
ls /tmp
haha
sh: excute: command not found
```

所以更保险的方法是对整行进行替换。

```
[root@xuexi tmp]# sed 's%^excute.*%du -sh /tmp%ge' ttt.txt
ls /tmp
haha
18M     /tmp
```

当然，如果想要执行第一行的"ls /tmp"命令也很简单，只需匹配这一行。也可以在此命令上进行命令扩展。如下面将/tmp 替换后，模式空间的内容是"ls -ld /tmp;du -sh /tmp"，所以会执行这两条命令。

```
[root@xuexi tmp]# sed 's!/tmp!-ld /tmp;du -sh /tmp!ge' ttt.txt
dr-xr-xr-x. 17 root root 8736768 Oct 25 14:11 /tmp
18M     /tmp
haha
excute a command
```

)

'I'
'i'
这两个修饰符作用相同，表示在 REGEXP 匹配的时候忽略大小写。这是 GNU 扩展功能。

'M'
'm'
和前文定址表达式中所述的 M 修饰符作用一致。见 M 修饰符。

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

(注：除了上述明确的修饰符外，还有一个特殊的不算修饰符的符号"\"，当它写在"s"命令的 REPLACEMENT 的每个行尾时，表示新转义当前命令行的行尾，也就是开启一个新行。例如：

```
echo 'abcdef' | sed 's/^.*$/\
&\
/'
```

这表示在 abcdef 这一行内容前后分别加一个空行，也就是将 abcdef 这一行嵌入到空行中间。所以可将其理解为换行符"\n"，但必须注意，如果转义新行后有内容，则此新行必须再次使用反斜线终止该行，因为它的行尾已经被转义了。例如：

```
echo 'abcdef' | sed 's/^.*$/\
&\
xyz\
/'

echo 'abcdef' | sed 's/^.*$/&\
/'
```

其实个人觉得使用"\n"方便多了，即容易理解又容易控制。例如上面最后一个命令改为：`echo 'abcdef' | sed 's/^.*$/&\n/'`。在后文的好几个例子中都是用了转义行尾的技巧，所以此处提上一提。

)

3.6 Less Frequently-Used Commands(比较少用的 sed 命令)

虽然可能用的不如前面所述的命令频繁，但这些小命令有时候非常有用。

`'y/SOURCE-CHARS/DEST-CHARS/'`
(y 命令中的斜线"/"可以统一被替换成其他单个字符。)

转换 pattern space 中能被 SOURCE-CHARS 匹配的字符为 DEST-CHARS，且是一一对应地转换。

斜线"/"(或被替换为的其他字符)、反斜线"\"和换行符可以出现在 SOURCE-CHARS 或 DEST-CHARS 中，但需要使用反斜线进行转义。(转义后的)SOURCE-CHARS 和 DEST-CHARS 中字符的数量必须完全相同。

`'a\'`
`'TEXT'`
是 GNU sed 的扩展功能，该命令可以在两个地址格式的定址表达式后使用。

队列化该命令后的文本内容(最后一个反斜线"\"是文本结束符，在输出时该符号会被移除)，并在当前 SCRIPT 循环结束时输出，或从输入流中读取下一行时输出。(注：本质是：只要"有读取下一行"的动作就会触发队列化内容的输出，不止是"a"、还有"i"和"c"以及"r"等，另外，除了 sed 循环的第一个动作可以读取下一行，命令"N"和"n"都会读取下一行，因此它们也会触发这些队列化内容的输出)

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

输入的文本内容中如果要出现反斜线字符，需要进行转义，即"\\"。

作为一个 GNU 扩展功能，如果命令"a"和换行符之间存在非空白字符"\"序列，则此反斜线会开启新行，并且反斜线后的第一个非空白字符作为下一行的第一个字符。这同样适用于下面的"i"和"c"命令。

(注：**命令"a"为尾部追加插入，其动作是将输入文本队列化在内存中，它不会进入模式空间，而是隐含动作自动输出模式空间内容时，在半路追上 stdout 并将队列化的文本追加到其尾部，并同时输出。下面的"i"和"c"命令所操作都是 stdout，都不会进入 pattern space。也就是说，这 3 个命令操作的文本内容不受任何 sed 其他选项和命令控制。**如"-n"选项无法禁止该输入，因为"-n"禁止的是 pattern space 的自动输出，同时，如果 SCRIPT 中这 3 个命令后还有后续的命令序列，则这些命令也无法匹配和操作这些文本内容，例如"p"命令不会输出这些队列化文本，因为它不是 pattern space 中的内容。
这是 sed 程序中少见的几个在 pattern space 外处理数据的动作。具体用法如下两个示例：

```
[root@xuexi tmp]# cat set2.txt
carrot
cookiee
gold
[root@xuexi tmp]# sed '/carrot/a\iron\nsteel' set2.txt  # 换行插入
carrot
iron
steel
cookiee
gold
[root@xuexi tmp]# sed '/carrot/a\iron\  # iron 作为第一行，其后跟\继续输入下一行
> steel' set2.txt          # 直到遇到结束引号，队列化文本才结束
carrot
iron
steel
cookiee
gold
```

)

'i\'
'TEXT'
是 GNU 的扩展功能，该命令可以在两个地址格式的定址表达式后使用。

立即输出该命令后的文本(除了最后一行，每一行都以反斜线"\"结尾，但在输出时会自动移除)。

(注："i"命令同"a"命令一样，只不过是在 stdout 流的头部加上队列化的文本，并同时输出。它同样不进入 pattern space，操作的也是 stdout 流)

'c\'
'TEXT'
删除从模式空间输出的内容，并在最后一行处(如果没有指定定址选项则每一行)输出"c"命令后队列化文本(除了最后一行，每行都以"\"结尾，输出时会移除)。该命令处理完后会开始新的 sed 循环，因为模式空间的内容将会被删除。

博客园-骏马金龙

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

(注："c"即 change，表示使用队列化文本修改输出流，虽说是修改，但实际上是替换模式空间的输出流并输出。它和"i"、"a"两个命令不一样，它"冒充"了输出流输出后就立即进入下一个 sed 循环，使得 SCRIPT 中后续的命令不会执行，而"i"、"a"则不会退出循环，而是会继续回到 SCRIPT 循环中执行后续的命令)

(注：虽然这 3 个命令用的远不如"s"命令频繁，但我个人认为，理解了这 3 个命令，就理解了大半 sed 的工作机制。)

'='
是 GNU 扩展功能，可以在两个地址格式的定址表达式后使用。

该命令用于输出当前正在处理的输入流中的行号(行号后会加上尾随换行符)。
(注：这是将 sed 内部保存的行号计数器的值输出，是内存中的内容，因此不进入 pattern space，不受"-n"选项影响。)

'l N'
将模式空间中的内容以一种明确的形式打印：非打印字符（和"\"字符）被打印成 C 语言风格的转义形式；长行被分割后打印，使用"\"字符来表示分割的位置；每一行的结尾被标记上"$"符。

N 指定了行分割时期望的行长度(即多少字符换行)，长度指定为 0 表示不换行。如果忽略 N，则使用默认长度(默认 70)。N 参数是 GNU sed 的扩展功能。

'r FILENAME'
GNU 扩展功能，该命令接受两个地址的定址表达式。

读取 filename 中的内容并按行队列化，然后在当前 SCRIPT 循环的最后或者读取下一行前插入到 output 流中。注意，如果 filename 无法被读取，它将被当成一个空文件而不会产生任何错误提示。

作为 GNU sed 的扩展功能，支持特殊的 filename 值：/dev/stdin，表示从标准输入中读取内容。

(注：

- "r"命令的功能和"a"命令的作用是完全一样的，连工作机制都完全相同，除了"r"命令是从文件中读取队列化文本，而"a"读取的是命令行中提供的。
- 从"r"命令之后到换行符或 sed 的引号之间的所有内容都作为"r"的文件参数。因此 r 命令之后如果还有命令，应当分行写。
- "r"命令是一次队列化 filename 中的所有行，后文还有一个"R"命令是每次队列化 filename 中的一行，它们都受输出流的影响，只要有输出流就追加到输出流中，并开始下一轮的队列化过程。并非 sed 每从输入流中读取一行就队列化一次)

)

'w FILENAME'
将模式空间的内容写入到 filename 中，作为一个 GNU sed 扩展，它支持两种特殊的 filename 值：/dev/stderr 和/dev/stdout，分别表示将模式空间的内容写到标准错误和标准输出中。

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

在输入流的第一行被读入前，filename 文件将被创建(或截断，即清空)；所有引用相同 filename 的"w"命令(包括替换命令"s"后的"w"修饰符)不会先关闭 filename 文件再打开该文件来写入。

(注：

- 即 sed 脚本中使用了"w"命令后该 filename 文件对 sed 而言一直是处于打开状态的，直到 sed 退出才关闭。多次使用引用相同文件的"w"命令会向该打开文件中写入，因此多个"w"写入 filename 时是追加写入的方式，但如果"w"打开的文件已存在，则会先截断该文件，即后续追加式的"w"输出会覆盖原文件。
- "w"命令不会影响 pattern space 的输出，它就像 tee 命令一样，一份输出到屏幕，一份重定向到 filename 中。

)

'D'
如果 pattern space 中未包含换行符，则像"d"命令中提到的一样进入下一个 sed 循环。否则删除 pattern space 中第一个换行符之前的所有内容，并重新回到 SCRIPT 的顶端重新对 pattern space 中剩下的内容进行处理，而不会读取输入流中的下一行。

(注：换句话说，D 命令总是删除 pattern space 中第一个换行符前的内容，如果删除后 pattern space 中还有内容剩下，则直接回到 SCRIPT 顶端重新对剩下的这部分内容执行命令，即以"continue"的方式强制进入下一个 SCRIPT 循环，如果 pattern space 中没有剩下内容，则直接退出当前 SCRIPT 循环，并进入下一个 sed 循环。)

'N'
在当前 pattern space 中的尾部加上一个换行符，并读取输入流的下一行追加到换行符后面。如果输入流中没有下一行供"N"读取，则直接退出 sed 循环。

(注：无论是 sed 循环的第一步、还是 n 命令或是 N 命令，它们读取下一行到 pattern space 时总会移除行尾换行符。但 N 命令在读取下一行前在当前 pattern space 的尾部加上了一个"\n"，这使得 pattern space 中有了两行，如果一个 SCRIPT 中有多次 N 命令还可能有更多行。因此 N 命令是 sed 进入多行工作模式的方法。但要注意，虽然称为多行模式，但在 pattern space 中加入换行符并非真的换行，在 pattern space 中仍是一行显示的，因为它能被匹配到，且在计算字符数量时会被计入，它仅仅只是一个特殊符号，只不过在输出时会换行显示)

'P'
输出 pattern space 中第一个换行符"\n"前面的内容。

(注：即输出 pattern space 中的第一行)

'h'
使用 pattern space 中的内容替换 hold space 中的内容。(注：替换后，pattern space 内容仍然保留，不会被清空)

'H'
在 hold space 的尾部追加一个换行符，并将 pattern space 中的内容追加到 hold space 的换行符尾部。
(注：追加后，pattern space 内容仍然保留，不会被清空)

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

'g'
使用 hold space 中的内容替换 pattern space 中的内容。

'G'
在当前 pattern space 的尾部追加一个换行符，并将 hold space 中的内容追加到 pattern space 的换行符尾部。
(注：这是另一个进入多行模式空间的方法。这就有了一个特殊的用法，sed 'G' filename 可以在 filename 的每一行后添加一个空行，这也是很多人产生误解的地方，认为 hold space 的初始内容为空行，追加到 pattern space 就成了空行。其实并非如此，因为无论是 pattern space 还是 hold space 在初始时都是空的，G 命令在 pattern space 的尾部加上了换行符，并将 hold space 中的空内容追加到换行符后，使得这成了一个空行。)

'x'
交换 pattern space 和 hold space 的内容。

3.7 Commands for 'sed' gurus(大师级的 sed 命令)

在大多数情况下，使用这些命令意味着你可能可以使用像"awk"或"perl"等的工具更好地达到目的。但是偶尔有人会坚持使用'sed'，这些命令可能会使得脚本变得非常复杂。

(注：虽说标签功能很少使用，但有时候确实非常有用，它在 sed 脚本内部实现了简单的编程结构体)

':LABEL'
不可接在定址表达式后。

设置一个分支标签 LABEL 供"b"、"t"或"T"(注："T"命令在后文的 GNU sed 扩展中说明)命令跳转。除此功能，无任何作用。
(注：冒号和 LABEL 之间不能有空格，虽然原文中有空格，但这是错误的)

'b LABEL'
无条件跳转到标签 LABEL 上。如果 LABEL 参数省略，则跳转到 SCRIPT 的尾部准备进入下一个 sed 循环，即跳转到隐含动作 auto_print 的前面。

't LABEL'
如果对最近读入的行有"s"命令的替换成功了，则跳转到 LABEL 标签处。如果 LABEL 参数省略，则跳转到 SCRIPT 的尾部准备进入下一个 sed 循环，即跳转到隐含动作 auto_print 的前面。(注：该命令和"T LABEL"相反，"T"是在"s"替换不成功时跳转到指定标签。)

(注：另外，"t"的跳转条件是有"s"命令替换成功，不是最近一个"s"命令替换成功，所以如果有多个"s"命令，即使"t"命令最近的"s"命令不成功，则仍会跳转，直到一个跳转循环内都没有替换成功的才终止跳转。例如：

```
`echo 'abcdef' | sed ':x;s/haha/yyy/;s/def/haha/;s/yyy/zzz/;tx'`
abczzz
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

第一轮，被读取的输入行在第一个"s"和第 3 个"s"命令失败，但第二个成功，所以仍会执行跳转，于是进入第二轮。在第二轮，第一个"s"和第三个"s"替换成功，所以继续跳转，于是进入第三轮。在第三轮中，所有"s"都失败，所以不再跳转。

)

(注：篇幅原因，使用一个示例简单解释一下标签和跳转命令的使用。

1. 使用":LABEL"设置好标签后，就可以使用"b LABEL"、"t LABEL"或"T LABEL"命令跳转到该标签。
2. 跳转到某标签的意思是开始执行该标签后的命令。
3. 如果"b"、"t"或"T"命令省略了参数 LABEL，则跳转到隐藏标签，即自动输出 pattern space 动作 `auto_print` 的前面。

假设有如下 test.txt 文件，该文件中空白处都是一个个的空格。目的是多个连续的空格替换成圆圈"○"，有几个空格就替换成几个圆圈，但初始时只有一个空格的不进行替换(即第一行的第一个空格和最后一行的最后一个空格不替换)。

```
[root@xuexi ~]# cat test.txt
 sleep       sleep
  sleep      sleep
   sleep     sleep
    sleep    sleep
     sleep   sleep
      sleep  sleep
       sleep sleep
```

可以使用下面简单的命令来实现：

```
[root@xuexi ~]# sed 's/  /○○/g;s/○ /○○/g' test.txt
 sleep○○○○○○○sleep
○○sleep○○○○○○sleep
○○○sleep○○○○○sleep
○○○○sleep○○○○sleep
○○○○○sleep○○○sleep
○○○○○○sleep○○sleep
○○○○○○○sleep sleep
```

如果使用标签功能，就可以变相地实现 sed 命令组的循环和条件判断功能。例如使用"b"标签跳转功能来实现，语句如下：

```
[root@xuexi ~]# sed '
:replace;
s/  /○○/;
/  /b replace;
s/○ /○○/g' test.txt
```

该语句首先定义了一个标签 `replace`，在其后是第一个要执行的命令，作用是将两个空格替换成两个圆圈的"s"命令，随后是"b"标签跳转语句，表示如果行中有两个连续的空格，就使用"b"命令跳转到 `replace` 标签处，于是再次执行"s"命令，替换结束后再次回到`/ /b replace` 命令，于是这样就可以对每一输入行实现循环替换动作，直到最后行中没有连续

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

空格。但此时可能还会有一个多余的空格跟在圆圈后面，于是就能满足 s/o /oo/g 命令的替换条件。

同样，还可以使用"t"标签完成上述要求。

```
[root@xuexi ~]# sed '
:replace;
s/  /oo/;
t replace;
s/o /oo/g' test.txt
 sleepooooooosleep
oosleepooooooosleep
ooosleepoooooosleep
oooosleepooooosleep
ooooosleepoooosleep
oooooosleepooosleep
ooooooosleep sleep
```

其实"t"标签更可能用于实现像 shell 中 case 的条件判断功能。例如：

```
sed '{
s/abc/ABC/;
t;
s/opq/OPQ/;
t;
s/xyz/XYZ/}' filename
```

这表示如果能替换什么成功，就跳转到尾部结束，是多选一的情况。

)

3.8 Commands Specific to GNU 'sed'(GNU sed 特有的命令)

这些命令是 GNU sed 特有的命令，因此必须谨慎使用它们，并且 sed 脚本没有可移植性的要求。

'e [COMMAND]'
该命令允许将 shell 命令行的输出结果插入到 pattern space 中。不给定参数 COMMAND 时，"e"命令将搜索 pattern space 并执行发现的命令，并将命令的执行结果替换到 pattern space 中(注：类似于 shell 下的命令替换功能)。

如果指定了"COMMAND"参数，"e"命令将解析它并认为它是一个 shell 命令，并在(子)shell 环境下执行后将结果发送到输出流中。

无论是否指定了 COMMAND 参数，如果待执行命令中包含了空字符，则都不被认为是一个命令。

注意，不像"r"命令，"e"命令的结果是立即输出的；而"r"命令需要延迟到当前 SCRIPT 循环结束时才会输出。

博客园 - 骏马金龙

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

'F'
输出当前处理的输入流的文件名(输入完后会换行)。

'L N'
这是一个失败的 GNU 扩展功能，无视。

'Q [EXIT-CODE]'
该命令只能在单个定址表达式下使用。

该命令和命令"q"相同，除了它不会输出 pattern space 中的内容(注："q"命令在退出前会输出当前 pattern space 中的内容)。同样，像"q"那样可以提供一个退出状态码。

该命令非常有用，因为要完成这个显而易见的简单功能的唯一替代方法是使用'-n'选项(这可能会使脚本不必要地复杂化)或使用以下代码片段，但这会浪费时间读取整个文件且没有任何效果可见：
(注：即使使用了"-n"选项，性能也不如"q"和"Q"，因为它仍会读完整个输入流，而"q | Q"是立即退出 sed 程序。)

```
:eat
$d        Quit silently on the last line
N         Read another line, silently
g         Overwrite pattern space each time to save memory
b eat
```

'R FILENAME'
每次读取 FILENAME 中的一行并队列化在内存中，其余的像"r"选项一样，在每个 SCRIPT 循环结束时追上 stdout 流并追加在其尾部。如果 FILENAME 无可读取或到达了文件尾部，将不会给出任何报错信息。

和"r"命令一样，它也支持特殊的 FILENAME 值：/dev/stdin，表示从标准输入中读取数据。

(注："R"命令是每次队列化 filename 中的一行，而"r"命令是一次队列化 filename 中的所有行。它们都受输出流的影响，有一次输出就追加到输出流中，并开始下一轮的队列化过程。并非 sed 每从输入流中读取一行就队列化一次)

'T LABEL'
和"t LABEL"相反，该命令是仅当"s"命令未完成替换时跳转到标签 LABEL。如果 LABEL 参数省略，则跳转到 SCRIPT 的尾部准备进入下一个 sed 循环，即跳转到隐含动作 auto_print 的前面。

'v VERSION'
该命令除了让 sed 程序在不支持 GNU sed 功能的时候失败不做任何事。此外，可以指定一个版本号，表示你的 sed 脚本必须要高于此版本的 sed 程序才能运行，例如指定"4.0.5"。默认指定的版本号为"4.0"，因为这是第一个支持该命令的版本。

'W FILENAME'
将 pattern space 中第一个换行符之前的内容写入到 filename 中。除此之外，和"w"命令完全相同。

'z'
该命令用于清空 pattern space。可以认为它等价于"s/.*//"，但却效率更高，且可以在

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

输入流中存在无效多字节序列(例如 UTF-8 时，一个字符占用多个字节)的情况下工作。POSIX 要求这样的序列无法使用"."进行匹配，因此没有任何方法既可以清空 sed 的 buffer 空间，又能保证 sed 脚本的可移植性。

3.9 GNU Extensions for Escapes in Regular Expressions(GNU 对正则表达式的反斜线扩展)

直到目前位置，我们只遇到了"\^"格式的转义，它告诉 sed 不要将脱字符"^"解释为特殊元字符，而是解释为一个普通的字面符号。再例如，"*"匹配的是单个*字符，而不是匹配 0 或多个反斜线字符。

本小节介绍另一种类型的转义。如下：

'\a'
匹配一个 BEL 字符，即一个"蜂鸣警告"(ASCII 7)。

'\f'
匹配一个分页符(ASCII 12)。

'\n'
匹配一个换行符(ASCII 10)。

'\r'
匹配一个回车符(ASCII 13)。

'\t'
匹配一个水平制表符(ASCII 9)。

'\v'
匹配一个垂直制表符(ASCII 11)。

'\cX'
Produces or matches 'CONTROL-X', where X is any character. The precise effect of '\cX' is as follows: if X is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus '\cz' becomes hex 1A, but '\c{' becomes hex 3B, while '\c;' becomes hex 7B.

'\dXXX'
匹配十进制 ASCII 值为 XXX 的字符。

'\oXXX'
匹配八进制 ASCII 值为 XXX 的字符。

'\xXX'
匹配十六进制 ASCII 值为 XX 的字符。

'\b' (回退删除键)无法实现，因为"\b"用于匹配单词的边界。

以下转义序列匹配特殊的字符类，且只在正则表达式中有效：

博客园--骏马金龙

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

'\w'
匹配任意单词组成字符。单词的组成成分包括：字母、数字和下划线。其他任意字符都不是单词的一部分。

'\W'
匹配人非单词字符。即匹配除了字母、数字和下划线的任意字符。

'\b'
匹配单词的边界位置。

'\B'
匹配非单词的边界位置。

"\`"
此为 sed 独有正则表达式。匹配 pattern space 中的开头。在多行模式下，这和"^"是不同的。

"\'"
此为 sed 独有正则表达式。匹配 pattern space 中的尾部。在多行模式时，这和"$"是不同的。

(注：关于最后两个正则表达式，它们是 sed 才可以使用的。在多行模式下，它们和"^"、"$"的区别，见 M 修饰符。)

4. Some Sample Scripts(一些简单的示例脚本)

这里有一些 sed 示例脚本，可以引导你成为大师级的 sed 使用者。

(注：说实话，下面 17 个示例一开始看的我怀疑人生，真的没想到 sed 功能强大如斯，但也难的想死，这是骨灰级玩家才能玩出来的花样。不过认真看完，其实也没什么难的)

- 示例目录：

一些吸引人的示例:

Centering lines
Increment a number
Rename files to lower case
Print bash environment
Reverse chars of lines

模仿一些标准的工具:

tac : Reverse lines of files
cat -n : Numbering lines
cat -b : Numbering non-blank lines
wc -c : Counting chars
wc -w : Counting words
wc -l : Counting lines

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

head : Printing the first lines
tail : Printing the last lines
uniq : Make duplicate lines unique
uniq -d: Print duplicated lines of input
uniq -u: Remove all duplicated lines
cat -s : Squeezing blank lines

4.1 Centering Lines(文本中心线)

该脚本将所有行都划分中心线，每行最多 80 个字符，中心线的位置指定为第 40 个字符处。如果要改变行宽度，则下面的'\{...\}'中的数字必须更改，同时需要在脚本的第一段中添加或删除适当个数的空格。

注意此处是如何使用 buffer 空间的命令来分隔两部分的，这是一个通用技巧。

(注：如果行的内容原本就超过 81 个字符，则这些行的行尾会被截断)

```
#!/usr/bin/sed -f

# Put 80 spaces in the buffer
1 {
  x
  s/^$/          /
  s/^.*$/&&&&&&&&/
  x
}

# del leading and trailing spaces
y/tab/ /
s/^ *//
s/ *$//

# add a newline and 80 spaces to end of line
G

# keep first 81 chars (80 + a newline)
s/^\(.\{81\}\).*$/\1/

# \2 matches half of the spaces, which are moved to the beginning
s/^\(.*\)\n\(.*\)\2/\2\1/
```

(注：从此脚本可知，虽在 pattern space 中加上了换行符，但其实不是真的换行，它在 pattern space 中仍是一行，只不过在输出时换行符使结果换了行。因此，N 命令进入多行模式也是一样的。同时，换行符是一个字符，能被匹配，也能被替换掉)

4.2 Increment a Number(数值增长)

该脚本是 sed 中少数几个演示了如何做算术计算的示例。这确实很有可能，但只能手动实现。

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

为了增长一个数值，只需要为该数的末尾加上 1，并替换原末尾即可。但有一种例外：当末尾数位 9 时，其前面一位也必须增长，如果这还是 9，则还需增长，直到无需进位时才结束。

Bruno Haible 提供的解决方案非常聪明，因为它只使用了一个 buffer 空间。它是这样工作的：将末尾 9 替换成下划线，然后使用多个"s"命令增长最后一个数字(下划线前面的数字也属于最后一个数字)，最后将所有的下划线替换为 0。

(注：下面的 td 和 tn 是 t d 和 t n 的简写。)

```
#!/usr/bin/sed -f

/[^0-9]/ d

# replace all trailing 9s by _ (any other character except digits,
could
# be used)
:d
s/9\(_*\)$/_\1/
td

# incr last digit only.  The first line adds a most-significant
# digit of 1 if we have to add a digit.

s/^\(_*\)$/1\1/; tn
s/8\(_*\)$/9\1/; tn
s/7\(_*\)$/8\1/; tn
s/6\(_*\)$/7\1/; tn
s/5\(_*\)$/6\1/; tn
s/4\(_*\)$/5\1/; tn
s/3\(_*\)$/4\1/; tn
s/2\(_*\)$/3\1/; tn
s/1\(_*\)$/2\1/; tn
s/0\(_*\)$/1\1/; tn

:n
y/_/0/
```

(注：例如为该 sed 脚本传递一个个位数 8，由于该数据是以 8 结尾的，因此将一直执行到 s/8\(_*\)$/9\1/;tn 将 8 增长为 9，并跳转到标签 n，标签 n 后的命令不用对其处理，所以直接输出 9。假如传递的数字是 3999，则:d;s/9\(_*\)$/_\1/;td 将一直循环，首先替换成"399_"，再循环一次替换成"39__"，最后一次循环替换成"3___"，由于此时结尾没 9 了，且是以 3 结尾，于是继续向下直到 s/3\(_*\)$/4\1/; tn，它会将"3___"替换成"4___"，并跳转到标签 n，y/_/0/将把下划线替换成 0 得到 4000，最后输出结果为 4000。

4.3 Rename Files to Lower Case(重命名文件名为小写)

这是一个非常奇怪的 sed 应用。转换文件名文本，并转换其为 shell 命令，然后将它们提供给 shell。

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

该应用的主体是其中的 sed 脚本部分，它会重新将名称的小写字母映射为大写，甚至还检查重新映射后的名称是否和原始名称相同。注意脚本是如何使用 shell 变量和正确的引号进行参数化的。

```
#! /bin/sh
# rename files to lower/upper case...
#
# usage:
#    move-to-lower *
#    move-to-upper *
# or
#    move-to-lower -R .
#    move-to-upper -R .
#

help()
{
        cat << eof
Usage: $0 [-n] [-R] [-h] files...

-n      do nothing, only see what would be done
-R      recursive (use find)
-h      this message
files   files to remap to lower case

Examples:
       $0 -n *        (see if everything is ok, then...)
       $0 *

       $0 -R .

eof
}

apply_cmd='sh'
finder='echo "$@" | tr " " "\n"'
files_only=

while :
do
    case "$1" in
        -n) apply_cmd='cat' ;;
        -R) finder='find "$@" -type f';;
        -h) help ; exit 1 ;;
        *) break ;;
    esac
    shift
done

if [ -z "$1" ]; then
        echo Usage: $0 [-h] [-n] [-r] files...
        exit 1
fi

LOWER='abcdefghijklmnopqrstuvwxyz'
UPPER='ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

博客园 - 骏马金龙

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

```
case 'basename $0' in
        *upper*) TO=$UPPER; FROM=$LOWER ;;
        *)       FROM=$UPPER; TO=$LOWER ;;
esac

eval $finder | sed -n '

# remove all trailing slashes
s/\/*$//

# add ./ if there is no path, only a filename
/\//! s/^/.\//

# save path+filename
h

# remove path
s/.*\///

# do conversion only on filename
y/'$FROM'/'$TO'/

# now line contains original path+file, while
# hold space contains the new filename
x

# add converted file name to line, which now contains
# path/file-name\nconverted-file-name
G

# check if converted file name is equal to original file name,
# if it is, do not print anything
/^.*\/\(.*\)\n\1/b

# escape special characters for the shell
s/["$'\\]/\\&/g

# now, transform path/fromfile\n, into
# mv path/fromfile path/tofile and print it
s/^\(.*\/\)\(.*\)\n\(.*\)$/mv "\1\2" "\1\3"/p

' | $apply_cmd
```

4.4 Print 'bash' Environment(打印"bash"环境)

该脚本去除了'set'命令中的 shell function 的定义。

```
#!/bin/sh

set | sed -n '
:x

# if no occurrence of "=()" print and load next line
/=()/! { p; b; }
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

```
/ () $/! { p; b; }

# possible start of functions section
# save the line in case this is a var like FOO="() "
h

# if the next line has a brace, we quit because
# nothing comes after functions
n
/^{/ q

# print the old line
x; p

# work on the new line now
x; bx
'
```

4.5 Reverse Characters of Lines(反转行中的字符)

该脚本可用反转行中字符的位置，它使用了一次性移动两个字符的技术，因此它的速度比直观看上去的更快。

注意标签定义语句前面的"tx"命令，它用来重置"t"标签跳转的条件，免得被第一个"s"命令影响而跳转。

```
#!/usr/bin/sed -f

/../! b

# Reverse a line.  Begin embedding the line between two newlines
# (注：下面的"s"命令使用了转义行尾的技巧，其实和"sed /^.*$/\n&\n/"是等价的)
s/^.*$/\
&\
/

# Move first character at the end.  The regexp matches until
# there are zero or one characters between the markers
tx
:x
s/\(\n.\)\(.*\)\(.\n\)/\3\2\1/
tx

# Remove the newline markers
s/\n//g
```

(注：例如 `echo abcde | sed -f file.sed`，传入的是"abcde"，首先判断它不是单字符，否则将跳转到尾部，然后将"abcde"嵌入到两个相同的特殊字符中，此处采用的是嵌入到两空行中，为了解释方便，假设这里的特殊字符为"#"符号，于是得到"#abcde#"，随后一个"t x"命令用于重置跳转条件，避免第一个 s 命令的成功状态影响跳转条件。再执行第二个 s 命令，该命令将两个"#"前后的字符反转，第一轮循环得到"e#dcb#a"，随后跳转再次替换，得到"ed#c#ba"，再次跳转，但这一次替换不成功。最后将所有的"#"删除，即得到最终结果"edcba")

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

4.6 Reverse Lines of Files(反转文件中的行)

这是一个没什么用但却很有意思的脚本，是模仿 unix 命令。就像 `tac` 一样工作。

注意，非 GNU sed 执行该脚本时很容易内存溢出。

```
#!/usr/bin/sed -nf

# reverse all lines of input, i.e. first line became last, ...

# from the second line, the buffer (which contains all previous lines)
# is *appended* to current line, so, the order will be reversed
1! G

# on the last line we're done -- print everything
$ p

# store everything on the buffer again
h
```

4.7 Numbering Lines(打印行号)

该脚本和 `cat -n` 的功能一样。当然，这个脚本没什么用，两个原因：第一，有人使用 C 语言实现了该功能，第二，下面这个 shell 脚本可以实现相同的目的却更快。

```
#! /bin/sh
sed -e "=" $@ | sed -e '
  s/^/      /
  N
  s/^ *\(......\)\n/\1  /
'
```

它首先使用 sed 来输出行号，然后通过"N"两行一分组并进行调整。当然，这个脚本并没有下面这个脚本的引导意义多。

这个算法同时使用了两个 buffer 空间，使得每一行可以尽快被打印出来然后丢弃。而行号被分离，使得数字改变后可以放入一个 buffer 空间，而没有改变的数字在另一个 buffer 空间；改变的数字使用一个"y"命令来修改。于是行号被存储在 hold space，在下一个迭代中被使用上。

```
#!/usr/bin/sed -nf

# Prime the pump on the first line
x
/^$/ s/^.*$/1/

# Add the correct line number before the pattern
G
h

# Format it and print it
s/^/      /
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

```
s/^ *\(......\)\n/\1  /p

# Get the line number from hold space; add a zero
# if we're going to add a digit on the next line
g
s/\n.*$//
/^9*$/ s/^/0/

# separate changing/unchanged digits with an x
s/.9*$/x&/

# keep changing digits in hold space
h
s/^.*x//
y/0123456789/1234567890/
x

# keep unchanged digits in pattern space
s/x.*$//

# compose the new number, remove the newline implicitly added by G
G
s/\n//
h
```

4.8 Numbering Non-blank Lines(打印非空白行行号)

模范的是"cat -b"，它不会计算空行的行号。

在此脚本中和上一个脚本相同的部分没有给出注释，于此处可知，对于 sed 脚本来说，给注释是多么重要的事。

```
#!/usr/bin/sed -nf

/^$/ {
  p
  b
}

# Same as cat -n from now
x
/^$/ s/^.*$/1/
G
h
s/^/      /
s/^ *\(......\)\n/\1  /p
x
s/\n.*$//
/^9*$/ s/^/0/
s/.9*$/x&/
h
s/^.*x//
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

```
y/0123456789/1234567890/
x
s/x.*$//
G
s/\n//
h
```

4.9 Counting Characters(计算字符个数)

该脚本展示了 sed 另一种做数学计算的方式。此处我们必须可以增大到一个极大的数值，因此要成功实现这一点可能不那么容易。

解决的方法是将数字映射为字母，这是 sed 的一种算盘功能实现。"a"表示个位数，"b"表示十位数，以此类推。如前所见，我们仍然在 hold space 中保存总数。

在最后一行上，我们将算盘上的值转换为十进制数字。为了能适应多种情况，这里使用了循环而不是一大堆的"s"命令：首先转换个位数，然后从数值中移除字母"a"，然后滚动字母使得各个字母都转换成对应的数值。

```
#!/usr/bin/sed -nf

# Add n+1 a's to hold space (+1 is for the newline)
s/./a/g
H
x
s/\n/a/

# Do the carry.  The t's and b's are not necessary,
# but they do speed up the thing
t a
: a;  s/aaaaaaaaaa/b/g; t b; b done
: b;  s/bbbbbbbbbb/c/g; t c; b done
: c;  s/cccccccccc/d/g; t d; b done
: d;  s/dddddddddd/e/g; t e; b done
: e;  s/eeeeeeeeee/f/g; t f; b done
: f;  s/ffffffffff/g/g; t g; b done
: g;  s/gggggggggg/h/g; t h; b done
: h;  s/hhhhhhhhhh//g

: done
$! {
  h
  b
}

# On the last line, convert back to decimal

: loop
/a/! s/[b-h]*/&0/
s/aaaaaaaaa/9/
s/aaaaaaaa/8/
s/aaaaaaa/7/
s/aaaaaa/6/
s/aaaaa/5/
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

```
s/aaaa/4/
s/aaa/3/
s/aa/2/
s/a/1/

: next
y/bcdefgh/abcdefg/
/[a-h]/ b loop
p
```

4.10 Counting Words(计算单词个数)

该脚本和前一个差不多，每个单词都转换成一个单独的字母"a"(上一个脚本中是每一个字母转换成一个字母"a")。

有趣的是，"wc"程序对"wc -c"计算字符时做了优化，使得计算单词的速度要比计算字符的速度慢很多。与之相反，这个脚本的瓶颈在于算术，因此单词计算的速度要快的多(因为只需维护少量的数值计算)

此脚本中和上一个脚本的共同部分没有给注释，这再次说明了 sed 脚本中注释的重要性。

```
#!/usr/bin/sed -nf

# Convert words to a's
s/[ tab][ tab]*/ /g
s/^/ /
s/ [^ ][^ ]*/a /g
s/ //g

# Append them to hold space
H
x
s/\n//

# From here on it is the same as in wc -c.
/aaaaaaaaaa/! bx;   s/aaaaaaaaaa/b/g
/bbbbbbbbbb/! bx;   s/bbbbbbbbbb/c/g
/cccccccccc/! bx;   s/cccccccccc/d/g
/dddddddddd/! bx;   s/dddddddddd/e/g
/eeeeeeeeee/! bx;   s/eeeeeeeeee/f/g
/ffffffffff/! bx;   s/ffffffffff/g/g
/gggggggggg/! bx;   s/gggggggggg/h/g
s/hhhhhhhhhh//g
:x
$! { h; b; }
:y
/a/! s/[b-h]*/&0/
s/aaaaaaaa/9/
s/aaaaaaa/8/
s/aaaaaa/7/
s/aaaaa/6/
s/aaaa/5/
s/aaa/4/
s/aa/3/
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

```
s/aa/2/
s/a/1/
y/bcdefgh/abcdefg/
/[a-h]/ by
p
```

4.11 Counting Lines(统计行数)

就像 wc -l 一样，统计行数。

```
#!/usr/bin/sed -nf
$=
```

4.12 Printing the First Lines(打印顺数前几行)

该脚本是 sed 脚本中最有用又最简单的。它显示了输入流的前 10 行。使用"q"的作用是立即退出 sed，不再读取额外的行浪费时间和资源。

```
#!/usr/bin/sed -f
10q
```

4.13 Printing the Last Lines(打印倒数几行)

输出倒数 N 行比输出顺数前 N 行要复杂的多，但确实很有用。N 值是下面脚本中 1,10 !s/[^\n]*\n//的数值 10 对应值，此处表示输出倒数 10 行。

该脚本类似于 tac 脚本，都将每次的结果保留在 hold space 中最后输出。

```
#!/usr/bin/sed -nf

1! {; H; g; }
1,10 !s/[^\n]*\n//
$p
h
```

关键点，该脚本维持了一个 10 行的窗口，在向其中添加一行时滑动该窗口并删除最旧的一行(第二个 s 命令有点类似于"D"命令，但却不用重新进入 SCRIPT 循环)

在写高级或复杂 sed 脚本时，**"窗口滑动"**的技术作用非常大，因为像"P"这样的命令要实现相同的目的需要手动做很多额外的工作。

为了介绍这种技术，在剩下的几个示例中，均使用了"N"、"P"和"D"命令充分演示了该技术。此处是一个"窗口滑动"技术对"tail"命令的实现。

这个脚本看上去更复杂，但实际上工作方式和上一个脚本是一样的：当踢掉了合理的行后，不再使用 hold space 来保存内部行的状态，而是使用"N"和"D"命令来滑动 pattern space：

```
#!/usr/bin/sed -f
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

```
1h
2,10 {; H; g; }
$q
1,9d
N
D
```

注意在读取了输入流的前 10 行后，其中的第 1、2、4 行的命令就失效了。之后，所有的工作是：最后一行时推出，追加下一行到 pattern space 中并移除其内第一行。

4.14 Make Duplicate Lines Unique(移除重复行)

这个示例充分展现了"N"、"D"和"P"命令的艺术所在，这可能是成为大师路上最难的几个命令。

```
#!/usr/bin/sed -f
h

:b
# On the last line, print and exit
$b
N
/^\(.*\)\n\1$/ {
    # The two lines are identical. Undo the effect of the N command.
    g
    bb
}

# If the 'N' command had added the last line, print and exit
$b

# The lines are different; print the first and go
# back working on the second.
P
D
```

正如所见，我们使用"P"和"D"维护了一个 2 行的窗口空间。这种窗口(滑动)技术在高级 sed 脚本中经常会使用。

(注：**"N"、"P"和"D"是 sed 中绝佳组合，通常为这 3 个命令关联不同的定址表达式以及感叹号"!"时，得到的结果千变万化。一方面"N"和"D"可以"滑动窗口"，另一方面，"P"和"D"可以输出窗口的第一行并滑动，再配合"N"或其它进入多行模式的方法(如"G"，或"s" 命令添加了换行符"\n")，使得窗口的大小可以一直维持下去。通常这 3 个命令同时使用时，它们的相对前后顺序是"NPD"。**另外，"D"比较特殊，它会重新进入 SCRIPT 循环，使得窗口中的内容只保留符合条件的行，因此滑动窗口变得更加"动态"，要使用 s 命令实现这种动态窗口滑动，只能借助"t"标签跳转来实现循环)

4.15 Print Duplicated Lines of Input(只打印重复行)

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

该脚本只打印重复行，就像"uniq -d"一样。

```
#!/usr/bin/sed -nf

$b
N
/^\(.*\)\n\1$/ {
    # Print the first of the duplicated lines
    s/.*\n//
    p

    # Loop until we get a different line
    :b
    $b
    N
    /^\(.*\)\n\1$/ {
        s/.*\n//
        bb
    }
}

# The last line cannot be followed by duplicates
$b

# Found a different one.  Leave it alone in the pattern space
# and go back to the top, hunting its duplicates
D
```

4.16 Remove All Duplicated Lines(移除所有重复行)

该脚本移除所有重复行，就像"uniq -u"一样。

```
#!/usr/bin/sed -f

# Search for a duplicate line --- until that, print what you find.
$b
N
/^\(.*\)\n\1$/ ! {
    P
    D
}

:c
# Got two equal lines in pattern space.  At the
# end of the file we simply exit
$d

# Else, we keep reading lines with 'N' until we
# find a different one
s/.*\n//
N
/^\(.*\)\n\1$/ {
    bc
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

```
}

# Remove the last instance of the duplicate line
# and go back to the top
D
```

4.17 Squeezing Blank Lines(压缩连续的空白行)

作为最后一个示例，这里给出了 3 个脚本，用于增加复杂度和速度，这可以使用"cat -s"来实现同样的功能：压缩空白行。

第一个脚本的实现方式是保留第一个空行，如果发现后续还有连续空行，则直接跳过。

```
#!/usr/bin/sed -f

# on empty lines, join with next
# Note there is a star in the regexp
:x
/^\n*$/ {
N
bx
}

# now, squeeze all '\n', this can be also done by:
# s/^\(\n\)*/\1/
s/\n*/\
/
```

下面这个脚本要更复杂一些，它会移除所有的第一个空行，并保留最后一个空行。

```
#!/usr/bin/sed -f

# delete all leading empty lines
1,/^./{
/./!d
}

# on an empty line we remove it and all the following
# empty lines, but one
:x
/./!{
N
s/^\n$//
tx
}
```

下面这个脚本会移除前导和尾随空行，速度最快。注意，"n"和"b"会彻底完成整个循环，而不会依赖于 sed 自动读取下一行。

```
#!/usr/bin/sed -nf

# delete all (leading) blanks
/./!d
```

博客园-骏马金龙

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

```
# get here: so there is a non empty
:x
# print it
p
# get next
n
# got chars? print it again, etc...
/./bx

# no, don't have chars: got an empty line
:z
# get next, if last line we finish here so no trailing
# empty lines are written
n
# also empty? then ignore it, and get next... this will
# remove ALL empty lines
/./!bz

# all empty lines were deleted/ignored, but we have a non empty.  As
# what we want to do is to squeeze, insert a blank line artificially
i\

bx
```

5. GNU sed's Limitations and Non-limitations(GNU sed 的限制和优点)

对于那些想写具有可移植性的 sed 脚本，需要注意有些 sed 程序有众所周知的 buffer 大小限制，要求不能超过 4000 字节，在 POSIX 标准中明确说明了要不小于 8192 字节。在 GNU sed 则没有这些限制。

然而，在匹配的时候是使用递归处理子模式和无限重复。这意味着可用的栈空间可能会限制那些可以通过某些 pattern 处理的缓冲区的大小。

6. Other Resources for Learning About 'sed'(学习 sed 的其他资源)

除了某些关于 sed 的书籍(专门研究或作为某个章节讨论的 shell 编程)之外，可以从"sed-users"邮件列表的 FAQ 中(包含一些 sed 书籍的推荐和建议)获取更多关于 sed 的信息：

http://sed.sourceforge.net/sedfaq.html

此外，还有 sed 的新手教程和其他一些 sed 相关资源：

http://www.student.northpark.edu/pemente/sed/index.htm

http://sed.sf.net/grabbag

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

"sed-user"邮件列表由 Sven Guckes 负责维护，可以访问 http://groups.yahoo.com 来订阅。

7. Reporting Bugs(Bugs 说明)

如要报告 bug，邮送至 bug-sed@gnu.org，请同时包含在邮件的 body 中包含 sed 的版本号，即"sed --version"的输出结果。

请不要像下面一样报告 bug：

```
while building frobme-1.3.4
$ configure
error--> sed: file sedscr line 1: Unknown option to 's'
```

以下是一些容易被报告的 bug，但实际上却不是 bug：(注：也就是容易让人疑惑的点，对于深入理解 sed 帮助很大)

- 'N'命令在最后一行上的处理方式

大多数版本的 sed 在"N"命令处理输入流的最后一行时会直接退出而不打印任何东西。GNU sed 会输出 pattern space 的内容，除非使用了"-n"选项。这是 GNU sed 特意的。

例如，`sed N foo bar` 将依赖于"foo"是偶数行还是奇数行。或者，当想在某个匹配行后读取后续的几行时，传统的 sed 可能只能写成这样：

```
/foo/{ $!N; $!N; $!N; $!N; $!N; $!N; $!N; $!N; $!N }
```

来替代：

```
/foo/{ N;N;N;N;N;N;N;N;N; }
```

无论何时，最简单的工作方式是使用`$d;N`，或者设置"POSIXLY_CORRECT"变量为一个非空值，来解决上述传统的依赖行为。

- 正则表达式崩溃(问题出在反斜线上)

sed 使用的是 POSIX 的基础正则表达式语法，根据 POSIX 标准，有些转义序列没有预定义，在 sed 中需要注意的包括：

```
\|、\+、\?、\<、\>、\b、\B、\w、\W、\'和\`
```

由于所有的 GNU 程序都是用 POSIX 的基础正则表达式，sed 会将这些转义符解析为特殊的元字符，因此"x+"会匹配一个或多个"x"，"abc|def"会匹配"abc"或"def"。

这些语法在用其他版本的 sed 运行时可能会出现问题，特别是某些版本的 sed 程序会将"|"和"+"解析为字面符号的"|"和"+"。因此在某些版本的 sed 上运行需要参考其所支持的正则表达式语法做相应修改。

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

再者说，某些脚本中"s|abc|def||g"来移除"abc"或"def"字符串，但这只在 sed 4.0.x 版之前能正常工作，在新版的 sed 中会将其解析为移除"abc|def"字符串。这在 POSIX 中同样没有定义对应的标准。

- '-i'选项会破坏只读文件

简单地说，"sed -i"将删除只读文件中的内容。通俗地说，"-i"选项会破坏受保护的只读文件。这不是 bug，而是 Unix 文件系统工作方式引起的结果。

普通文件的权限表示的是可以对该文件中的数据做什么样的操作，但是目录的权限表示的是可以对目录中的文件做什么样的操作。"sed -i"绝不会为了写入而再次打开该文件。取而代之的是，它会先写入一个临时文件，最后将其重命名为原文件名：删除或重命名文件的权限是目录权限控制的，因此该操作依赖的是目录的权限，而非文件本身的权限。同样的原因，"sed -i"不允许修改一个可读但其目录只读的普通文件。

- '0a'无法工作，报错

根本就没有 0 行。0 行的概念只有一种情况下使用 `0,/REGEXP/`，它和 `1,/REGEXP/`只有一点不同：如果 REGEXP 能匹配上第一行，则前者的结果是只有第一行，而后者会继续向下匹配直到能匹配 REGEXP，因为范围地址必须要跨越至少两行，除非直到最后一行都没有匹配上。

- '[a-z]'会忽略大小写

这是字符集环境设置的问题。POSIX 强制'[a-z]'这样的字符列表采用当前系统当前字符集的排序规则进行排序，C 字符集环境下，它代表的是小写字母序列，其他字符集环境下，可能代表的是小写和大写字母序列，这依赖于字符集。

为了解决这个问题，可以设置 `LC_COLLATE` 和 `LC_CTYPE` 环境变量为 C。

- 's/.*//'不会清空 pattern space

会发生这种情况，可能是因为你的输入流中包含了多字节序列(如 UTF-8).POSIX 强制这样的序列字符无法被"."匹配，因此 `s/.*//`将不会如你所愿那样清空 pattern space。事实上，在绝大多数多字节序列环境下，没有任何办法脚本的中途清空 pattern space。出于这个原因，GNU sed 提供了一个"z"命令，它将"\0"作为输入流的行分隔符。

为了解决这些问题，可以设置 `LC_COLLATE` 和 `LC_CTYPE` 环境变量为 C。

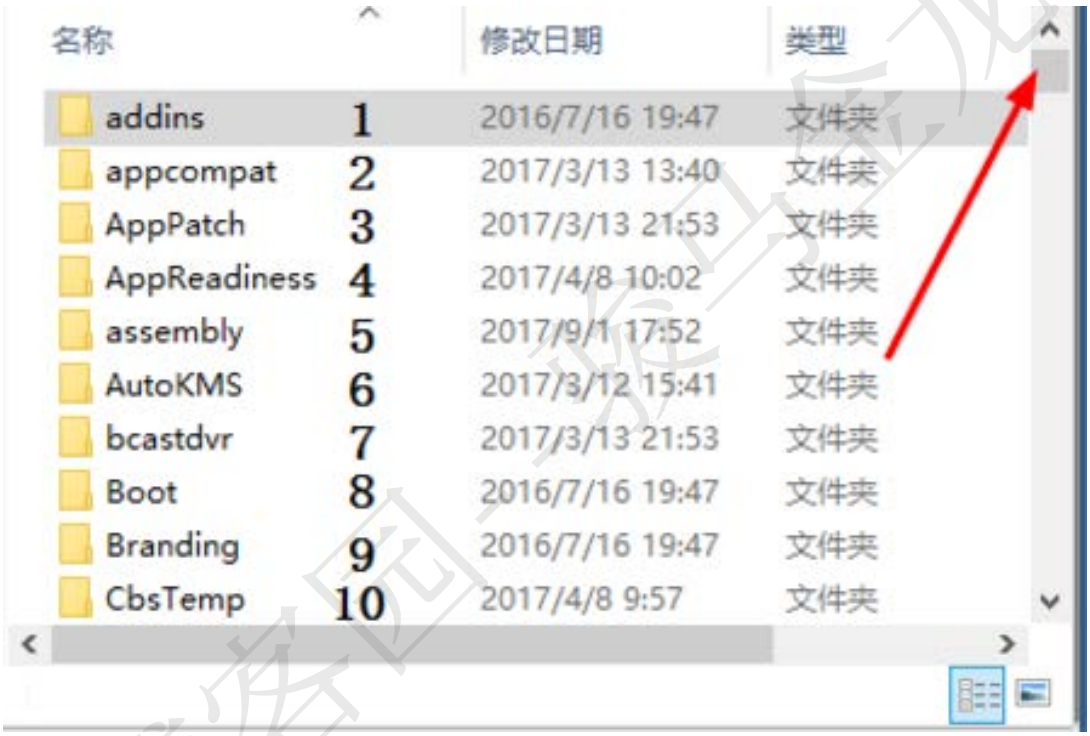
文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

sed修炼系列(三)：高级应用之实现窗口滑动技术

本节目录：

1.什么是滑动窗口(slide window)技术

一图胜千言。



在上图中，资源管理器的高度固定为正好装下 10 行文件名，如果想要显示第 11 行，就要下拉滚动条一行的距离，使得 11 行正好能显示出来。但这时，最旧的第一行就会被踢出当前可视窗口。

滑动窗口的意思大致就是如此。维护一个窗口，当向窗口添加新数据时，旧的数据就被剔除出去，保证窗口的大小固定。当然，还有动态的大小不固定的窗口，此时根据其他规则来判断是否要剔除旧数据以及剔除哪些旧数据。

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

在 sed 的高级用法中，窗口滑动技术作用非常大，这也是"N"、"D"和"P"重要的原因。sed 以模式空间为"主战场"，以保持空间为"副战场"。所以 sed 要维护"窗口"只能通过模式空间，因为只有在模式空间中才能决定是否要踢掉旧数据以及踢掉哪些旧数据。但很多时候还会辅以保持空间，将每次操作完的窗口数据暂存到保持空间，并在下一个循环中将数据从保持空间拿回来维护一番。

下面是一些示例和窗口技术的用法说明。

2.实现窗口滑动

假如想要输出文件 a.txt 的前 10 行。这很简单。

```
sed '10q' a.txt
```

由于"q"命令在退出 sed 程序前会输出模式空间内容，所以第 10 行也会被输出，如果使用的是"Q"命令，则应该为：

```
sed 11Q a.txt
```

难题来了，想输出倒数 10 行要怎么实现，倒数 15 行？倒数 20 行？。再通用化一些，怎么能实现 tail 工具的倒数行查看功能呢。

2.1 通过"s"命令滑动窗口

由于 **sed 是"一往无前，绝不回头"的流式处理器，任何一份输入流只要被读取过绝对不会再次被读取。**

再者，**sed 采用行号计数器即时计数，每读取一行，计数器就加 1。因此 sed 在读取到最后一行前，不知道后面还有多少行，也不知道什么时候才是最后一行。直到读取到输入流的最后一行，sed 为该行打上"$"标记，表示这是最后一行。"$"只是一个标记符号，并非行号，行号只在计数器中记录，因此无法通过"$"来计算出倒数几行，例如"$-1"是错误的写法**。这意味着，sed 无法直接输出倒数的行。要输出倒数多少行，必须通过窗口来实现。

既然说到了行号，顺便提一提。行号的匹配过程比正则表达式的匹配效率要高的多，因为行号是记录在内存中的，只要比较下表达式中的行号值和计数器记录的值就可以了。而正则表达式匹配的时候要经过编译、匹配等工作，而且 sed 的正则表达式引擎匹配的效率并不如想象中的那么好，特别是使用了".*"结合了其他表达式的时候。因此，批量处理大量文件，特别是大文件时，能用行号尽量用行号。

回到正题。例如，要输出倒数 10 行，这个窗口就一直维持在 10 行的固定大小。当读取到最后一行时，可以通过"$"符号来判断这是否是最后一行，是的话就输出该窗口，否则不输出该窗口的数据。

通常，这样的问题会借助保持空间来临时存储窗口的数据，但此处仅依靠模式空间也能维持一个固定行数的窗口。如下：

```
#!/usr/bin/sed -nf
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

```
# 先读取 8 行，加上自动读取的一行共 9 行
N;N;N;N;N;N;N;N

# 判断是否是最后一行，如果不是则读取下一行并踢掉最前面的一行
:1
N
$!s/[^\n]*\n//;t1

# 读取到最后一行后，输出窗口中的内容
p
```

为了在模式空间中保持固定行数的窗口，只能让所有动作在一个 sed 循环内完成(因为 SCRIPT 循环结束时会清空模式空间)，因此必须借助标签循环跳转。在上面的示例脚本中，首先读取了 8 行，加上自动读取的一行，模式空间中共 9 行，这正是我们需要维护的窗口。随后，使用一个循环判断标签，先读取一行，再判断该行是否是最后一行。如果不是，则剔除掉窗口中的第一行，这样窗口就一直维护在固定的 9 行大小。直到读取最后一行，这时替换命令失败。最后窗口中的 10 行内容被输出。

既然通过窗口能输出倒数 10 行，显然输出倒数第 10 行也是很简单的，只需将上面的"p"改成大写的"P"即可。

那要是想输出倒数 15 行、20 行甚至是 50 行，也要这么写吗？先不说要写一大堆的"N"命令，仅仅窗口太大的问题就会导致效率极速下降。假如一个文件有 1000 行，要输出倒数 20 行，从第 20 几行开始每次做"$"匹配的时候都要从模式空间中的 20 多行搜索，这相当于处理了一个 1000*20=20000 行的文件。当然，行号匹配直接比较计数器的值，没有这种顾虑，但如果真的是正则表达式匹配，效率必然极速下降。

这时，保持空间就可以派上用场了。但需要注意的是，虽然使用保持空间可以简化处理逻辑，但因为两个 buffer 空间的数据交换过程都会对性能有一丝丝影响。所以一般来说，用一个 buffer 空间实现比借助两个 buffer 空间效率要高那么一点点，特别是大量处理大文件同时还要交换大量数据的时候，性能差距就比较明显。

2.2 借助保持空间暂存窗口

上面的例子的思路是读取一些初始行数填充窗口，在窗口快要达到目标大小时使用标签循环判断功能来维持固定大小的窗口滑动过程。

如果借助保持空间，可以将每次滑动后的窗口数据暂存起来，在读取了下一行时，将其追加回来再处理。

例如上面的例子借助保持空间实现，语句如下：

```
#!/usr/bin/sed -nf

H
10,${g;s/[^\n]*\n//;h}
$p
```

由于借助了保持空间，因此整个过程无需在一个 sed 循环内完成。上面的过程将通过 sed 循环的自动读取来填充窗口，并将其追加到保持空间。当填充到 10 行后，将其从保持空

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

间拉回模式空间，并使用"s"命令滑动该窗口，滑动结束后再次将其放回保持空间。直到最后一行滑动结束后，输出最终窗口的内容。

注意，上面的数字是 10，而不应该是 11，这是"H"命令导致的结果，因为每次 H 执行时都会在保持空间尾部先追加一个"\n"，即使最初保持空间为空时也会追加。这使得从读取第 10 行并执行了 H 后，保持空间将有共 11 行，其中第一行为空。

2.3 将窗口维护命令"s"替换成"D"

考虑"D"命令的特性：**删除模式空间的第一行，并进入下一个 SCRIPT 循环**。因此，除非模式空间已经没有内容了，否则"D"命令会一直 SCRIPT 循环直到模式空间中没有能被 D 命的地址匹配的内容。

为了方便说明"D"，举个简单的例子：压缩连续空行。还有压缩相邻重复行，即去除重复行。

```
echo -e "1\n2\n\n3\n4\n\n\n5" | sed '$!N;/^\n$/!P;D'
echo -e "1\n2\n3\n3\n4\n4\n4\n4\n4\n5" | sed -r '$!N;/^(.*)\n\1$/!P;D'
```

这两个命令的思路都是以窗口为模型(我是这么认为的。自从有了窗口的概念，任何涉及"NDP"的命令我都将其认为是窗口，这样容易理解多了)。以"N"命令读入下一行到窗口中，如果窗口中的两行不重复，就输出第一行并执行"D"剔除第一行，于是滑动了窗口，并进入下一个 SCRIPT 循环。直到窗口中出现重复行，将一直循环滑动这大小为 2 行的窗口但不输出，直到不重复了才输出前面相邻重复行的最后一行。

由此也可以看出，其实即使是单行或双行的模式空间也都算是一个窗口，只不过这个窗口维护起来比较灵活。

以第二个去除重复行的命令来说，大致流程如下：其中 d 表示该行未被"P"输出且被"D"删除了，p 表示被"P"输出后再被"D"删除。

```
1p
2p
3d
3p
4d
4d
4d
4d
4p
5p
```

回到正题。"D"命令其自身实现了一种特殊的条件式 SCRIPT 循环，而"s"命令要实现这样的循环只能通过标签判断的方式来实现，除非借助保持空间。正因为如此，"D"命令也能剔除窗口中的旧数据实现窗口滑动。

使用"D"很多时候可以简化脚本的复杂性，但其绝对无法替代"s"命令的维护行为。因为 D 命令的循环范围固定为一整个 SCRIPT 循环(正如上面压缩重复行的示例，每次都回到 SCRIPT 的下一个循环从头开始)，而"s"命令借助标签跳转可以实现在任意大小的范围内循环。因此，"D"命令实现窗口滑动时，在通用性上不及"s"命令。

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

仍然以输出倒数 10 行数据为例。借助"D"实现的语句如下：

```
#!/usr/bin/sed -nf

1h
2,10H
10g
$p
10,${N;D}
```

其中前 3 行只是为了填充一个 10 行的窗口放在模式空间中(填充窗口的方法实在很多，所以只要知道这 3 步是干啥的就行)，从第 11 行开始这 3 行就没有任何作用了。重点在最后一行的 10,,${N;D}，这是"NPD"的绝佳组合之一，通过"N;D"轻松地维持了一个固定大小的窗口：读一行删一行。直到最后一行被"N"读取，才被"$p"输出。之所以"$p"要放在"D"的上一行，是因为"D"总是会回到 SCRIPT 的顶端，所以它后面的命令是不会执行的。

2.4 真正的大招

呀，原来窗口这么好用？但是不要钻入了 sed 的牛角而蒙蔽了双眼。说实话，通过 sed 自身实现很多较为复杂的需求并不那么简单，费神又费力，反而结合其他文本处理工具来实现要简单的多的多。

就以上面的例子来说，输出倒数 10 行，结合 shell 变量简单的不得了。

```
total=`wc -l <filename`
sed -n $((total-9))',$p' filename
```

这样想输出任意哪些行号的行都可以简单至极。以上只是一个示例，结合其他可用的工具，同样能轻松地实现 sed 自身比较复杂的需求。因此，这是最终的"大招"，当然，为了学习深入学习 sed，这些只好先不想。

另外，上面的示例中引号加的位置很奇怪，这是 sed 结合 shell 的难题。很多人可能都遇到过这个问题，在网上也没有很好的解释，因为这不是 sed 的问题，而是 shell 解析的特性。见 sed 修炼系列(四)：sed 中的疑难杂症。

2.5 维持窗口方法论

综合上面几个示例，维持窗口分为两种情况：

1. 在模式空间中维持窗口大小。这分为两个过程：
    - (1)填充窗口到指定行数；
    - (2)滑动窗口。
2. 借助保持空间暂存窗口。
    - (1).不断填充保持空间的窗口；
    - (2).在填充到指定大小后，将窗口拉回模式空间进行滑动；
    - (3).滑动后将其覆盖回保持空间暂存下来。
    - (4).继续读取并填充保持空间中的窗口，然后拉回模式空间，滑动后再暂存。

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

看上去，第一种情况比较容易些。确实如此，第一种情况不用多次考虑两个 buffer 之间的数据交换。并且，第一种情况的效率更高，因为在达到窗口大小之后，再也无需和保持空间交换数据。

3.最佳搭档："N"、"P"和"D"命令

经过前面窗口滑动的示例，也能发现"N"和"D"是一个绝佳搭档，它俩配合能实现完美的窗口滑动，而且相比于借助保持空间暂存窗口的方法，它俩的逻辑非常清晰。

前面说了"N"和"D"的结合，加上"P"呢？单独的"N"和"P"结合没什么好说的，可能出现的情形太多了。

但"P"和"D"的结合却有一层固定的意义：根据匹配模式判断是否输出多行模式中的第一行，然后踢掉该行，并回到 SCRIPT 循环顶端。这很可能是在维护一个大小为两行的窗口。格式通常为：

```
[Address]P;D
```

再同时结合"N"，作用就更明显了，维持一个窗口，并判断是否要输出该窗口的第一行，然后滑动窗口。格式通常为：

```
[Address1]N;[Address2]P;D
```

很多时候，Address 是省略掉的。P 命令前的 Address 完全是条件判断语句，判断是否要输出。N 命令前的 Address1 如果存在，最大的可能是"$!"，这表示当最后一行已被读取过(无论是 sed 循环自动读取的、n 命令读取的还是 N 命令读取的)，直接跳过该命令。如果不加"$!"，则没有下一行可供读取时，将直接输出模式空间(除非指定了"-n")并退出 sed 程序。

是否在 N 前加"$!"，对结果的影响很大。但想判断是否要加，难度还是挺大的，至少要对 sed 何时输出模式空间内容了如指掌。可以阅读 sed 修炼系列(一)：花拳绣腿之入门篇。

最后，需要说的是"N"和奇偶数行的关系。"N"命令在无法读取下一行时将输出模式空间内容并退出 sed，万一读到的最后一行是奇数行，又或是偶数行，会怎样影响结果呢？可能很多人(包括我自己)都琢磨过这个问题，也被这个问题困惑了很久而不得其解，这个问题甚至放进了 info sed 手册的 Bug 报告段中进行专门的解释。

其实根本不用考虑最后一行是奇数行还是偶数行，无论最后一行是奇数行还是偶数行，sed 根本不管这个逻辑，它只记得最后一行是否被读取过，如果读取过，则在 N 命令处就退出 sed。所以，真正需要考虑的是 N 命令前是否要加"$!"，它决定了 sed 是在 N 处退出还是继续执行后面的命令。但有一种情况必须要考虑奇偶性，当"N"结合了其它读取行的操作(命令"n"或 sed 的自动读取)时，因为其余任意一个读取动作都会改变"N"读取的行的奇偶性。

例如，分别输出输入流的奇数行和偶数行。考虑以窗口模型实现的话，这是很简单的。

```
seq 1 10 | sed 'N;P;d'       # 输出奇数行
seq 1 10 | sed '1!{N;P};d'   # 输出偶数行
seq 1 10 | sed -n '1!{N;P;d}' # 输出偶数行，但需要考虑奇偶性
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

```
seq 1 10 | sed -n '1!{$!N;P;d}'  # 输出偶数行，不需考虑奇偶性
```

前两个命令都很容易理解，第三个命令却需要考虑奇偶性。因为窗口是从第二行开始填充的，所以窗口数据被"d"删除前，其内第 2 行总是奇数行，例如"(2,3)"是一个窗口，"{4,5}"是一个窗口。当最后一行是奇数时，其必定是被"N"读取的，这不会影响结果。但如果最后一行是偶数，则此行必定是被 sed 自动读取的，使得 sed 在"N"命令处就结束了，其后的"P"就无法执行。这时，可以考虑在"N"前加上"$!"，即第四条命令。

当然，更简单的方法如下：

```
seq 1 10 | sed 'n;d'
seq 1 10 | sed '1!n;d'
```

再例如，要删除文件的倒数第 2 行。如果知道窗口的概念，这一切都很容易：维持一个 2 行的窗口(N 和 D 就够了)，当发现最后一行被读取后，不输出其前一行即可。以下是实现语句：

```
sed 'N;$!P;D' filename
```

注意这里的"N"在处理最后一行时的作用，它输出了最后一行，且让 sed 程序结束，但这一行是自动输出的，而非"P"输出的，所以会受"-n"影响。如果改为"$!N"，则最后一行被读取后，将直接连续执行两"D"，即同时删除了倒数 2 行。

博客园--骏马金龙

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

sed修炼系列(四)：sed中的疑难杂症

本文目录：

4.1 sed 中使用变量和变量替换的问题

在脚本中使用 sed 的时候，很可能需要在 sed 中引用 shell 变量，甚至想在 sed 命令行中使用变量替换。也许很多人都遇到过这个问题，但引号却死活调试不出正确的位置。其实这不是 sed 的问题，而是 shell 的特性。搞懂 sed 如何解决引号的问题，对理解 shell 引号问题有很大帮助，触类旁通，以后在使用 awk、mysql 等等自带语法解析的工具时就不会再疑惑。

例如下面想输出 a.txt 的倒数 5 行的语句。可能顺手就写出了下面的命令行：

```
total=`wc -l <a.txt`
sed -n '$((total-4)),$p' a.txt
```

但很不幸，这会报错。一方面，"$"在 sed 中是特殊符号，放在定址表达式中时，它表示的是输入流的最后一行的标记。而$(())中也出现了"$"符号，这会让 sed 去解析该符号。另一方面，$(())这部分是使用 shell 计算而不是使用 sed 计算的，因此必须要将其暴露给 shell，以便能让 shell 能解析它。

再说说 shell 中单引号、双引号和不加引号的情况。

- 单引号：单引号内的所有字符变为字面符号。但注意：单引号内不能再使用单引号，即使使用了反斜线转义也不允许。
- 双引号：双引号内的所有字符变为字面符号，但"\"、"$"、"`"(反引号)除外，如果开启了"!"引用历史命令时，则感叹号也除外。
- 不使用引号：等同于使用了双引号，但会进行大括号和波浪号扩展。

上面关于双引号的情况，描述的并不是真正的完整，但已足够。这些只是它们的字面意义，引号真正的意义在于：决定命令行中哪些"单词"需要被 shell 解析，也决定哪些是字面意义不用被 shell 解析。

显然，单引号内所有字符都成为了字面符号，shell 不会解析其内任何单词，例如单引号内变量不再被解析、命令替换和算术运算不再执行、不会进行路径扩展等等。总之，单引号内的字符全是普通字符，如果某些字符需要交给自带解析功能的命令解析，必须使用单引号。例如，"$"、"!"和"{}"在 sed 中均有特殊意义，要想让 sed 能解析它们，必须对它们使用单引号，否则必出错，或者产生歧义。例如下面 3 个 sed 语句中的符号都必须使用单引号才能得到正确结果。

```
sed '$d' filename
sed '1!d' filename
sed -n '2{p;q}' filename
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

而想要让特殊字符被 shell 解析，必须不能将其包围在单引号中，可以使用双引号，也可以不加任何引号，即使不加引号时可能看上去很怪异。例如，上面的算术运算$(())是想被 shell 解析的，因此必须使用单引号或者不加引号将其暴露给 shell。所以正确的语句是：

```
sed -n $((total-4))',$p' a.txt
sed -n "$((total-4))"',$p' a.txt
sed -n "$((total-4)),\$p" a.txt
```

从肉眼看上去，这个语句的引号加的真的很怪异。但 shell 又不管丑美，它是死的，在划分命令行的时候它有自己的一套规则，规则怎样就怎样划分。

于是，关于 sed 如何和 shell 交互的问题可以得出一套结论：

1. 遇到需要被 shell 解析的都不加引号，或者加双引号；
2. 遇到 shell 和所执行命令共有的特殊字符时，要想被 sed 解析，必须加单引号，或者在双引号在加反斜线转义；
3. 那些无关紧要的字符，无论加什么引号。

因此，使用命令替换的方式让 sed 输出倒数 5 行的语句如下：

```
sed -n `expr $(wc -l <a.txt) - 4`',$p' a.txt
```

上面的语句中，`expr $(wc -l <a.txt) - 4` 要被 shell 解析，因此必须不能使用单引号包围。而$p 部分的"$"要被 sed 解析成最后一行，必须使用单引号以避免被 shell 解析。

更复杂一些，在 sed 的正则表达式中使用变量替换。例如，输出 a.txt 中以变量 str 字符串开头的行到最后一行。

```
str="abc"
sed -n /^$str/',$p' a.txt
```

因为没有使用任何引号，所以$str 能如期被 shell 替换成"abc"。这个命令还有多种写法：

```
sed -n '/^'$str'/,$p' a.txt
sed -n "/^$str"'/,$p' a.txt
sed -n "/^$str/,\$p" a.txt
sed -n "/^$str/,"'$'p a.txt
```

给一个稍难一些的 sed 符号使用问题。将/etc/shadow 中的最后一行的密码部分替换成"$1$123456$wOSEtcyiP2N/IfIl15W6Z0"。

```
[root@xuexi ~]# tail -n 1 /etc/shadow
userX:$6$hS4yqJu7WQfGlk0M$Xj/SCS5z4BWSZKN0raNncu6VMuWdUVbDScMYxOgB7mXUj
./dXJN0zADAXQUMg0CuWVRyZUu6npPLWoyv8eXPA.::0:99999:7:::
```

替换语句如下：

```
old_pass="$(tail -n 1 /etc/shadow | cut -d':' -f2)"
new_pass='$1$123456$wOSEtcyiP2N/IfIl15W6Z0'
sed -n '$'s%$old_pass%$new_pass% /etc/shadow
```

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

由于 old_pass 和 old_pass 中包含了"/"和"$"符号，因此"s"命令的分隔符使用了"%"替代。再仔细观察 new_pass，其内有"."符号，这是正则表达式的元字符，因此它还可以匹配其他情况。

4.2 反向引用失效问题

当正则表达式中使用二者选一的选项"|"时，如果分组括号()中的内容没有参与匹配，后向引用将不起作用。例如(a)\1u|b\1 将只匹配"aau"的行，不匹配"ba"的行，因为在二者选一的第二个正则中\1 代表的分组没有参与匹配，所以第二个正则中的\1 失效，但是第一个正则中的\1 有效。

这是正则匹配的问题，不只是 sed，其它使用基础正则和扩展正则引擎的工具也一样会有这样的问题。

另外，在 s 命令中使用反向引用时，将不会引用"s"命令外面的分组。例如：

```
echo "ab3456cd" | sed -r "/(ab)/s/([0-9]+)/\1/"
```

得到的结果将是 ab3456cd，而不是 ababcd，而且如果此时使用\2 引用，则会报错"invalid reference \2 on 's' command's RHS"。

4.3 "-i"选项的文件保存问题

sed 是通过创建一个临时文件，并将输出写入到该临时文件，然后重命名该临时文件为源文件来实现文件保存的。因此，sed 会无视文件的只读性。

是否允许重命名或移入或删除文件，是由文件所在目录的权限控制的。如果目录为只读权限，则 sed 无法使用"-i"选项保存结果，即使该文件具有可读权限。

4.4 贪婪匹配问题

所谓的贪婪匹配，是指当正则表达式能匹配多个内容时，取最长的那个。最简单的例子，给定数据"abcdsbaz"，正则表达式"a.*b"可以匹配该数据中"ab"和"abcdsb"，由于贪婪匹配，它会取最长的"abcdsb"。

```
echo "abcdbaz" | grep -o "a.*b"
abcdb
```

基础正则表达式和扩展正则表达式一直以来的一个不足之处在于无法原生态克服贪婪匹配，像 Perl 正则或其他编程语言的正则实现的比较完整，在"*"或"+"这种多次重复的匹配后加上一个"?"就可以明确表示采取懒惰匹配的模式，例如"a.*?b"。

```
echo "abcdbaz" | grep -P -o "a.*?b"
ab
```

想要克服基础正则或扩展正则的贪婪匹配，只能"投机取巧"地采用不包含符号"[^]"来实现。例如上面的：

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

```
echo "abcdbaz" | grep -o "a[^b]*b"
ab
```

这种投机取巧的方式，其性能比较差，因为基础或扩展正则表达式的引擎总是会先匹配出最长的内容，然后往回匹配，这称为"回溯"。例如"abcdsbaz"在被"a[^b]*b"匹配时，先匹配出"abcdsb"，再一个字符一个字符地回退匹配，直到回退到第一个"b"才是最短的结果。

再例如，/etc/passwd 文件中每行数据的格式如下：

```
rootx:0:0:root:/root:/bin/bash
```

如何使用 sed 向/etc/passwd 中的每个用户问声好，输出格式大致为："hello root"、"hello nobody"。

首先，得取出文件中的第一列，即用户名。但由于该文件中所有行都采用冒号分隔各字段，想要使用正则表达式匹配得到第一段，必须克服贪婪匹配。语句如下：

```
sed -r 's/^([^:]*):.*/hello \1/' /etc/passwd
```

注意，sed 采用的是基础正则和扩展正则引擎，在克服贪婪匹配时，它必须先匹配出最长的，再回溯出最短的。

如果想取/etc/passwd 中的前两个字段呢？只需将克服贪婪的正则当作整体重复一次即可。

```
sed -r 's/^([^:]*):([^:]*):.*/hello \1 \2/' /etc/passwd
```

取第三个字段？

```
sed -r 's/^([^:]*:){2}([^:]*):.*/hello \2/' /etc/passwd
```

取第三和第五个字段？没办法，只能将第四个字段显式标注出来。

```
sed -r 's/^([^:]*:){2}([^:]*):([^:]*):([^:]*):/hello \2 \4/'
/etc/passwd
```

取第三道第 5 字段？更简单，重复 3 次就可以了。

```
sed -r 's/^([^:]*:){2}(([^:]*:){3}).*/hello \2/' /etc/passwd
```

但这样的结果中，第 3 到第 5 字段中必然会包含":"分隔符，想要去除它？洗洗睡吧！sed 本就不擅长处理字段，克服贪婪匹配本就让表达式变得很复杂不易读，而且效率还不高。用它处理字段，绝对是吃撑了。

4.5 sed 命令"a"和"N"的纠葛

sed 的"a"命令作用是将提供的文本数据队列化在内存中，然后在模式空间内容输出时追加在输出流的尾部一并输出。

例如，在匹配行"ccc"后插入一行数据"matched successful"。

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

```
echo -e "aaa\nbbb\nccc\nddd" | sed '/ccc/a matched successful'
aaa
bbb
ccc
matched successful
ddd
```

咋一使用"a"命令，很顺利，没毛病。但是结合"N"试试看？

```
echo -e "aaa\nbbb\nccc\nddd" | sed '/ccc/{a\
matched successful
;N}'

aaa
bbb
matched successful
ccc
ddd
```

不是追加在尾部吗，怎么跑匹配行的前面去了？即使"N"读取了下一行，也应该是追加在"ddd"的下一行吧？想要真正弄明白这个问题，对 sed 模式空间的输出机制必须了如指掌，可以参考 sed 修炼系列(一)：花拳绣腿之入门篇。此处简单描述下"N"命令的输出机制。

无论是 sed 自动读取下一行，还是"n"或"N"命令读取下一行，只要有读取动作，在其前面必然会输出模式空间的内容。当"N"读取下一行时，首先它会判断是否还有下一行可供读取，如果有，则先锁住模式空间，然后自动输出并清空模式空间，再解锁模式空间并向其尾部追加一个换行符"\n"，最后读取下一行追加到换行符尾部。由于模式空间被锁住，使得自动输出时输出流是空流，也同样无法清空模式空间。注意，它不是禁止输出，虽然输出空流的结果和禁止输出是一样的，但输出空流它有输出动作，有输出流，会写入标准输出，而禁止输出则没有输出动作。如果没有下一行可供读取，则自动输出模式空间、清空模式空间并退出 sed 程序。过程大致如下所描述：

```
if [ "$line" -ne "$last_line_num" ];then
    lock pattern_space;
    auto_print;
    remove_pattern_space;
    unlock pattern_space;
    append "\n" to pattern_space;
    read next_line to pattern_space;
else
    auto_print;
    remove_pattern_space;
    exit;
fi
```

回到"a"命令和"N"命令结合的问题上。之所以"a"命令的队列化文本会插入在匹配行的前面，问题就出在输出空流上。"N"在准备读取下一行时，它有输出动作，即使输出结果为空。而"a"命令是时刻等待 sed 输出流的，只要一有输出流，立马就会追上去追加在输出流的屁股后面。因此，"matched successful"会追加在空流的尾部，追加之后"N"才会读入下一行，最后输出模式空间中的内容"ccc\nddd"，也就得到前面"有悖期待"的结果。

4.6 sed 中感叹号取反的弯弯绕绕

文章内容完全原创，本人保留所有权利
欢迎善意传播，杜绝恶意利用者
作者博客：骏马金龙(http://www.cnblogs.com/f-ck-need-u/)

你知道使用"!"号取反，但也许你并没有发现**感叹号可以放在定址表达式后，也可以放在命令的前面**。这两者虽然都是取反，但意义决然不同，最终导致的结果也不同。

1. **感叹号在定址表达式后，表示对行进行筛选。**
2. **感叹号在命令的前面，表示满足条件的行不执行该命令。这是对模式空间中的行筛选要执行的命令。**

假如文件 a.txt 中包含了 3 行：

```
djkaldahsdf
abcskdf2das
chhdsjaj
```

对于以下三个 sed 脚本：

- (1)./^abc/!{d}
- (2)./^abc/{!d}
- (3)./^abc/!d

示例(1)中感叹号放在定址表达式后，这表示不是以字母"abc"开头的行会执行 d 删除命令。而那些以"abc"开头的行，则不符合定址表达式，后续的 d 命令不会执行。也就是说，该 sed 脚本的作用是：除了"abc"开头的行，其余行全删除，因此只输出第 2 行。
示例(2)中感叹号放在命令的前面，而非定址表达式后面。这表示的是以"abc"开头的行不执行 d 命令。而那些不以"abc"开头的行 y 由于不满足条件，也不会执行 d 命令。也就是说，该 sed 脚本的 d 命令是多余的，任何行都不会删除。因此所有行都输出。
示例(3)等价于示例(1)，因为定址匹配动作优先于命令的执行，感叹号直接被认为是定址表达式的一部分。

但不管哪种情况，对于不满足定址表达式的(定址后的感叹号也算是定址表达式的一部分)行，都不会执行后续任何命令，这些行是直接自动输出的，由"-n"选项控制是否将其输出。

4.7 sed 卡死，cpu 100%问题

有些人可能遇到过这种问题，特别是 sed 处理以 UTF-8 格式导出的数据库文件。

之所以会出现这样的问题，是因为字符集的问题，确切地说是本地环境(locale)和文件的编码不一致。

如果出现这样的问题，可以将 LC_COLLATE 和 LC_CTYPE 环境变量设置为 C。也可以简单地设置 LANG=C 或 LC_ALL=C。