从 sed 到 awk

续本达

清华大学 工程物理系

2024-07-16 清华

软件安装

下载文件

```
wget 'http://hep.tsinghua.edu.cn/~orv/pd/ccompiler_opt.py'
```

- Debian

```
apt install gawk-doc feedgnuplot
```

正则表达式

- 普适的字符串处理工具
  - `^`, `$` 开始与结束
  - `+` 至少一次重复
  - `?` 0 或 1 次
  - `()` 组合
  - `|` 或
- 重要工具 `grep` 行匹配 `sed` 替换
- 进阶： Perl 正则表达式
  - 多行匹配

正则表达式的写法

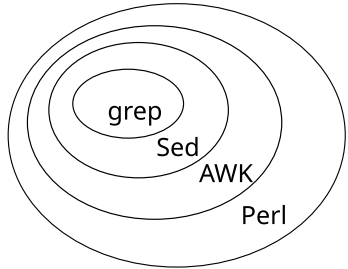
1. 思考清楚我们需要匹配什么文字，这些文字在文本里都可能以什么形式出现
2. 写出一个正则表达式
3. 试验正则表达式，改进
   - 哪些是我想要匹配却没匹配的
   - 哪些是我不想匹配但却匹配了的

这几乎适用于所有程序的写作。

几乎是所有科学实验的思路

1. 思考清楚我们在证伪什么论断
2. 设计实验
3. 做实验，看它是否成功证伪了论断
   - 若是，理论被推翻
   - 若不是，理论经受住了考验，思考更强更精细的可证伪假说

先试用功能集小的，再试用大的



对人也一样

在信任一个人能胜任工作之前，先布置一个小任务

1. 打字
2. 小作业
3. 大作业

自动处理文本和字符串的重要性

- 节省大量的重复劳动
  - 找到一种解决问题的途径和高效的工作方法，让人满足
  - 是工作和苦劳的本质区别
  - 程序化地解决问题，让问题本身变得更有趣
- 入门时，程序化解决文本问题花的时间更长
  - 注意不仅要学习如何解决问题，还要学习识别问题

参考资料

- Dougherty and Robbins, Sed & Awk, 1997.
- 陈皓，AWK 简明教程
- `info sed`, `info awk`

自动处理文本和字符串的重要性

- 节省大量的重复劳动
  - 找到一种解决问题的途径和高效的工作方法，让人满足
  - 是工作和苦劳的本质区别
  - 程序化地解决问题，让问题本身变得更有趣
- 入门时，程序化解决文本问题花的时间更长
  - 注意不仅要学习如何解决问题，还要学习识别问题

参考资料

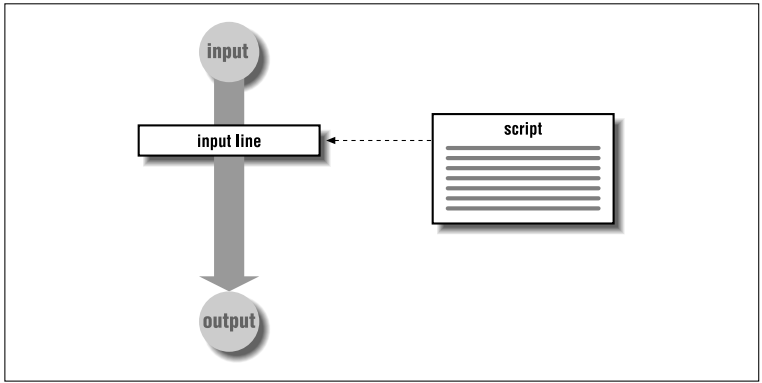
- Dougherty and Robbins, Sed & Awk, 1997.
- 陈皓，AWK 简明教程
- `info sed`, `info awk`

命令行语法

`sed/awk script filename`

1. 一行写不下时
2. 担心特殊字符在 `shell` 被转义时

数据驱动的编程



匹配变换模式

1. 通过 `/regex/` 匹配要修改的行
2. 通过随后的命令修改之

使用脚本

```
sed -f tuna.sed sources.list
```

- 所有命令先处理第一行，再处理第二行，……

定位行

1. 什么都不写代表所有行
2. `/regex/` 代表满足 regex 的行
3. 1, 2, 3 行号，例如
   ```
   1d
   ```
   为删除第一行。
   `$` 代表最后一行。
4. `line1,line2` 代表从 line1 行到 line2 行，闭区间
   ```
   50,$d
   ```
   把从第 50 行到最后一行都删除
5. 在行的定位后加 ！代表反义
   ```
   50,$!d
   ```
   与
   ```
   1,49d
   ```
   等价

定位行

1. 什么都不写代表所有行
2. `/regex/` 代表满足 regex 的行
3. 1, 2, 3 行号，例如

   ```
   1d
   ```

   为删除第一行。
   $ 代表最后一行。
4. `line1,line2` 代表从 line1 行到 line2 行，闭区间

   ```
   50,$d
   ```

   把从第 50 行到最后一行都删除
5. 在行的定位后加 ！代表反义

   ```
   50,$!d
   ```

   与

   ```
   1,49d
   ```

   等价

定位行

1. 什么都不写代表所有行
2. `/regex/` 代表满足 regex 的行
3. 1, 2, 3 行号，例如

   `1d`

   为删除第一行。

   `$` 代表最后一行。
4. `line1,line2` 代表从 line1 行到 line2 行，闭区间

   `50,$d`

   把从第 50 行到最后一行都删除
5. 在行的定位后加 ！ 代表反义

   `50,$!d`

   与

   `1,49d`

   等价

定位行

1. 什么都不写代表所有行
2. `/regex/` 代表满足 regex 的行
3. 1, 2, 3 行号，例如

   ```
   1d
   ```

   为删除第一行。
   `$` 代表最后一行。
4. `line1,line2` 代表从 line1 行到 line2 行，闭区间

   ```
   50,$d
   ```

   把从第 50 行到最后一行都删除
5. 在行的定位后加 ！ 代表反义

   ```
   50,$!d
   ```

   与

   ```
   1,49d
   ```

   等价

定位行

1. 什么都不写代表所有行
2. `/regex/` 代表满足 regex 的行
3. 1, 2, 3 行号，例如

   ```
   1d
   ```

   为删除第一行。
   $ 代表最后一行。
4. `line1,line2` 代表从 line1 行到 line2 行，闭区间

   ```
   50,$d
   ```

   把从第 50 行到最后一行都删除
5. 在行的定位后加 ！ 代表反义

   ```
   50,$!d
   ```

   与

   ```
   1,49d
   ```

   等价

定位行的嵌套

可以把行的定位嵌套，例如，删除所有从第 1 到 49 行中的空行，并把 lo 换成 do

```
1,49{
  /^\s*$/d
  s/lo/do/
}
```

比较新旧文件的区别

```
sed '1,100s/Aug/ 08/' usr_share_doc_list.txt > usr_share_doc_list.txt.new
diff -u usr_share_doc_list.txt{,.new}
```

```
--- usr_share_doc_list.txt 2021-08-30 11:59:19.000000000 +0800
+++ usr_share_doc_list.txt.new 2023-07-27 12:18:20.209493339 +0800
@@ -1,5 +1,5 @@
 total 8868
-drwxr-xr-x  2 root root 4096 Aug  8  2017 abootimg
+drwxr-xr-x  2 root root 4096  08  8  2017 abootimg
 drwxr-xr-x  2 root root 4096 Feb  8  2021 acl
 drwxr-xr-x  2 root root 4096 Jul  2  2019 acpi
 drwxr-xr-x  2 root root 4096 Feb  8  2021 adb
@@ -92,7 +92,7 @@
 drwxr-xr-x  2 root root 4096 Apr 14 15:44 cl-swank
 drwxr-xr-x  2 root root 4096 Feb  8  2021 cl-trivial-gray-streams
 drwxr-xr-x  2 root root 4096 Feb  8  2021 cl-unicode
-drwxr-xr-x  2 root root 4096 Aug 30 10:21 cmatrix
+drwxr-xr-x  2 root root 4096  08 30 10:21 cmatrix
 drwxr-xr-x  2 root root 4096 Feb  8  2021 coreutils
 drwxr-xr-x  3 root root 4096 Feb  8  2021 cowsay
 drwxr-xr-x  2 root root 4096 Feb  8  2021 cowsay-off
```

sed 的典型使用场景

一文多改 一个文件上投入多个改动

一脚多文 一个脚本用在多个文件上

高级 grep 在查找关键词之前，先对待找的文本进行清洗

管道过滤 直接对着标准输入过滤，给到标准输出

数据清洗

数据清理：把数据整理成方便的格式

- 在物理学之外的其它“大数据”学科，90% 的工作是数据清洗。

例子：课件中的代码块

https://git.tsinghua.edu.cn/physics-data/lecture/-/blob/master/Makefile

```
e%.tex: p%.tex
	sed -e 's/{sqlite3}/{sqlite3}/' -e 's/python/python/' \
	    -e 's/bash/bash/' \
	    -e '/{minted}/s/\[\]/\[bgcolor=lightgray,fontsize=\\scriptsize\]/' \
	    -e 's/\{\scriptsize\begin{verbatim}/{\\scriptsize\\begin{verbatim}/' \
	    -e 's/end{verbatim}}/end{verbatim}}/' < $< > $@
```

多行匹配

- 部分 pcre2grep -M 的功能。

```
  echo "This is your cat
  my cat's name is betty
This is your dog
  my dog's name is frank" | sed 'N;s/\n/,/'
```

```
This is your cat,  my cat's name is betty
This is your dog,  my dog's name is frank
```

Hold space

为 sed 自动机配了一个队列。

- g 将 hold space 中的内容拷贝到 pattern space 中，原来 pattern space 里的内容清除
- G 将 hold space 中的内容 append 到 pattern space 后
- h 将 pattern space 中的内容拷贝到 hold space 中，原来的 hold space 里的内容被清除
- H 将 pattern space 中的内容 append 到 hold space 后
- x 交换 pattern space 和 hold space 的内容

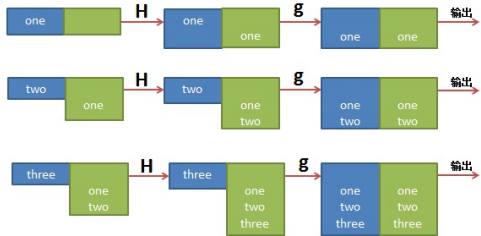
收集-积累-输出

```
echo "one
two
three" | sed "H;g"
```

```
one

one
two

one
two
three
```



10 乘 10 矩阵

- 执行 N 9 次。重复过多，不如使用 python 辅助构建字符串。

```
seq -w 00 99 | sed $(python3 -c "print('N;'*9)")'s/\n/,/g'
```

反序

```
seq 5 | sed '1!G;h;$!d'
```

```
5
4
3
2
1
```

```
seq 5 | tac
```

```
5
4
3
2
1
```

行编辑器

- 在显示器大行其道之前
  - 计算机终端的主要输出方式是打印机，当时最佳的方式行编辑

试验

```
ed departments.csv
```

- d
- /regex/ 找到下一个满足 regex 的行
  - g/regex/ 找到所有满足 regex 的行
- s/pattern/replacement/
- g/propagating/p
- echo 'g/propagating/p' | ed /tmp/texts.log

sed 与 ed 的区别

- ed 是交互的，默认作用于一行，用 `/regex/` 扩展编辑范围
- sed 是批处理的，默认作用于每一行，用 `/regex/` 限制编辑范围

awk 与 sed 的区别

- 不再使用 ed 的单字母命令，改用 C 语言风格的语法，例如`/regex/{ print }`

ed 的命令影响了 ex

ex 是 vi 的前身，后者发展成 VIM (Vi IMproved)， VIM 的开发停滞，开发者正在转向 neovim

sed 到 awk

有些简单的问题可用 sed 解决，复杂一些的问题适合 awk 解决

sed 适合解决

1. 批量修改文本
2. 把同一个修改应用到多个文件上
3. 把一种格式的文本转化成另一种

sed 的操作（最常用的是查找-替换），更接近于编辑器

awk 是模式匹配语言

- 用 awk 描述数据中的规律，从而取得对应的信息
- 单行的 awk 程序可以从命令行直接输入
- 把文本看成是一直数据仓库，例如 CSV
  - 在文本处理的同时，集成算术运算

sed 到 awk

有些简单的问题可用 sed 解决，复杂一些的问题适合 awk 解决

sed 适合解决

1. 批量修改文本
2. 把同一个修改应用到多个文件上
3. 把一种格式的文本转化成另一种

sed 的操作（最常用的是查找-替换），更接近于编辑器

awk 是模式匹配语言

- 用 awk 描述数据中的规律，从而取得对应的信息
- 单行的 awk 程序可以从命令行直接输入
- 把文本看成是一直数据仓库，例如 CSV
  - 在文本处理的同时，集成算术运算

sed 与 awk 的共同点

- 都是数据流驱动的编程语言但与 Make 语言不同，不是函数式语言
- 大量使用正则表达式
- 可以快速从 bash 调用
- awk 起源于 grep 和 sed，而后两者都脱胎于 ed 编辑器

学习 sed 和 awk 的步骤

- ☒ 调用 sed 和 awk
- ☒ 使用正则表达式
- ☒ 从 bash 调用时的引用
  即用 \ 和 ' “废掉他人武功”
- ☐ 书写 sed 和 awk 的脚本

学习新语言的原因

- 实验物理的大数据方法
  - 28 天速成七种语言：Python, Bash, Make, Regex, Awk, R, SQL
- 经济性：有了 Python 基础，学习其它语言门槛大大降低，不再需要额外精力
  - 第一门语言难度 100% ，思维广度 +100%
  - 第二门语言难度 25% ，思维广度 +100%
  - 第三门语言难度 1% ，思维广度 +100%
  - 回报越来越大，learn x in y mintes。

运行 awk

```
awk '{print $1}' list
```

计算 1 到 100 的和

```
seq 100 | awk '{total+=$NF} END {print total}'
```

5050

正式入门

```
echo | awk '{ print "Hello, world!" }'
```

```
Hello, world!
```

```
echo "Hello, world!" | awk '{ print }'
```

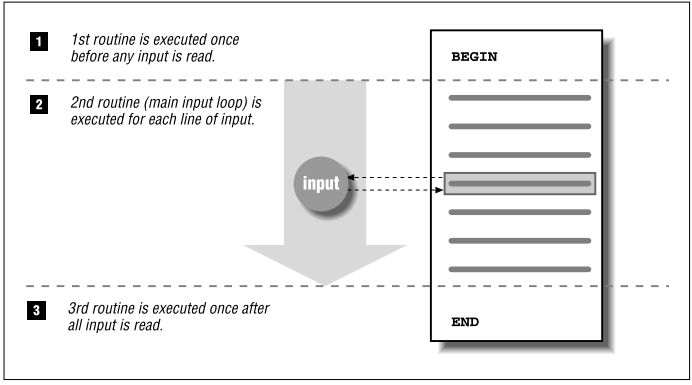
```
Hello, world!
```

```
awk 'BEGIN { print "Hello world!" }'
```

```
Hello world!
```

awk 的运行模式

- 数据驱动的编程风格。



内建的变量

变量	含义
$0	当前记录（这个变量中存放着整个行的内容）
$1~$n	当前记录的第 n 个字段，字段间由 FS 分隔
FS	输入字段分隔符 默认是空格或 Tab
NF	当前记录中的字段个数，就是有多少列
NR	已经读出的记录数，就是行号，从 1 开始。 如果有多个文件话，这个值也是不断累加。
FNR	当前记录数，与 NR 不同的是，这个值会是各个文件自己的行号
RS	输入的记录分隔符， 默认为换行符
OFS	输出字段分隔符， 默认也是空格
ORS	输出的记录分隔符，默认为换行符
FILENAME	当前输入文件的名字

- -F 可以改变 FS 的值，例如 awk -F，

替换文本内容

- info: gawk → sample programs → clones
  - cut
  - egrep
  - id
  - split
  - tee
  - uniq
  - wc

课堂练习： wc

- wc -l

```
seq 3 100 | awk '{lines++} END {print lines}'
```

98

- wc -c

```
seq 3 100 | awk '{chars+=length($0)+1} END {print chars}'
```

288

```
seq 3 100 | wc -c
```

288

课堂练习： uniq

- uniq -c

```
echo "a
a
b
c
c
c" | awk -f uniq.awk
```

```
2 a
1 b
3 c
```

课堂练习： egrep

- egrep

```
seq 10000 | awk '/^23+$/ { print }'
```

```
23
233
2333
```

sed 与 awk 的联合

使用管道

```
seq 9 | sed 'H;g' |\
    awk -v RS='' '{for(i=1;i<=NF;i++)
        printf("%dx%d=%d%s", i, NR, i*NR, i==NR?"\n":"\t")}'
```

```
1x1=1
1x2=2 2x2=4
1x3=3 2x3=6 3x3=9
1x4=4 2x4=8 3x4=12 4x4=16
1x5=5 2x5=10 3x5=15 4x5=20 5x5=25
1x6=6 2x6=12 3x6=18 4x6=24 5x6=30 6x6=36
1x7=7 2x7=14 3x7=21 4x7=28 5x7=35 6x7=42 7x7=49
1x8=8 2x8=16 3x8=24 4x8=32 5x8=40 6x8=48 7x8=56 8x8=64
1x9=9 2x9=18 3x9=27 4x9=36 5x9=45 6x9=54 7x9=63 8x9=72 9x9=81
```

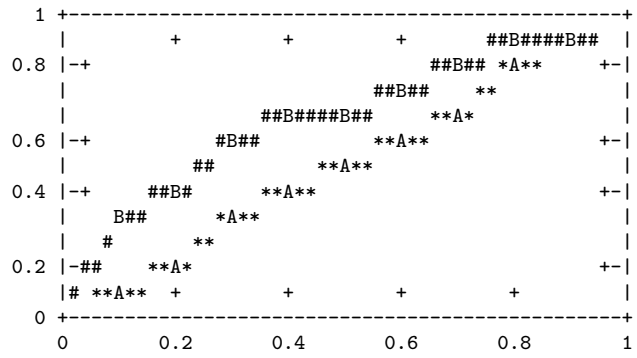
乘法表

```
seq 9 | sed 'H;g;s/\n/ /g' | awk '{for(i=1;i<=NF;i++)
        printf("%dx%d=%d%s", i, NR, i*NR, i==NR?"\n":"\t")}'
```

```
1x1=1
1x2=2 2x2=4
1x3=3 2x3=6 3x3=9
1x4=4 2x4=8 3x4=12 4x4=16
1x5=5 2x5=10 3x5=15 4x5=20 5x5=25
1x6=6 2x6=12 3x6=18 4x6=24 5x6=30 6x6=36
1x7=7 2x7=14 3x7=21 4x7=28 5x7=35 6x7=42 7x7=49
1x8=8 2x8=16 3x8=24 4x8=32 5x8=40 6x8=48 7x8=56 8x8=64
1x9=9 2x9=18 3x9=27 4x9=36 5x9=45 6x9=54 7x9=63 8x9=72 9x9=81
```

在命令行制图

```
seq 0 0.1 1 | awk '{print $1, $1, sqrt($1)}' | \
   feedgnuplot --domain --lines --points --terminal 'dumb 60,16' --unset grid --exit
```



统计选课人数

- 使用字典

去掉无用的编译选项

```
{
    GENTOO_ENABLE=1;
    if (match($0, /flags="([^"=]*)"/, cflags)) {
        split(cflags[1], fields);
        for (i in fields) {
            if (match(fields[i], /-m([[:graph:]]*)/, inst)) {
                if (!index(enabled_flags, inst[1])) {
                    GENTOO_ENABLE=0;
                }}}}
    if (!GENTOO_ENABLE) { sub(cflags[1], "-mGENTOO_DISABLE"); }
    print;
}
```

https://gitweb.gentoo.org/repo/gentoo.git/commit/?id=ea2987349ece6f07da01e1c2f3e9808455fa394c

回顾

科学数据处理的原则

复现 透明 一次 最佳工具

版本控制

Git 与队友分工协作，与明天的自己协作

数据流水线

GNU Make 管理数据的依赖与转换，实现错误恢复和并行计算

命令环境

GNU 环境中强大的小工具组合，开发与使用相融合

计算语言

Python 语法友好，工具丰富，统领 C/C++/Fortran 库