Files

ls - 列出目录内容

-a: 列出所有文件，包括以点（.）开头的隐藏文件。
-l: 使用长格式列出文件详细信息。
-R: 递归列出目录下的所有文件。
-t: 按修改时间排序。
-s: 显示每个文件的块大小。

mkdir - 创建目录

-p: 创建多级目录，如果中间的某些目录不存在，会一同创建。

```
mkdir -p /path/to/new/directory
```

mv - 移动或重命名文件

-i: 交互模式，如果目标文件存在，会提示用户确认。
-f: 强制模式，覆盖目标文件而不提示。

cp - 复制文件或目录

-r: 递归复制目录及其内容。
-i: 交互模式，如果目标文件存在，会提示用户确认。
-v: 详细模式，显示复制过程中的信息。

ln - 创建硬链接或符号链接

-s: 创建符号链接。
-f: 如果目标文件存在，覆盖它。

rm - 删除文件或目录

-r: 递归删除目录及其内容。
-f: 强制删除，不提示。
-i: 交互模式，删除前提示用户确认。

chmod - 更改文件权限

基本参数

u: 用户（文件所有者）。
g: 组。
o: 其他人。
a: 所有人。

权限操作符

+: 添加权限。
-: 去除权限。
=: 赋予指定权限，去除其他权限。

权限位

r: 读权限。
w: 写权限。
x: 执行权限。

数字表示法

权限也可以用三位八进制数表示，每位分别代表用户、组和其他人的权限。

参数

-R: 递归更改目录及其内容的权限。

< - 重定向标准输入

将命令的标准输入重定向到文件或设备。

```
sort < unsorted.txt
```

将 unsorted.txt 作为 sort 命令的输入。

> - 重定向标准输出

将命令的标准输出重定向到文件（覆盖文件）。

```
echo "Hello" > hello.txt
```

将字符串"Hello"写入到 hello.txt，如果文件存在，会被覆盖。

>> - 重定向并追加标准输出到文件

将命令的标准输出追加到文件末尾。

```
echo "Hello" >> hello.txt
```

将字符串追加到 hello.txt 文件末尾。

2> - 重定向标准错误输出

将命令的标准错误输出重定向到文件。

```
ls non_existent 2> error.log
```

将 ls 命令的错误输出写入到 error.log。

| - 管道

将一个命令的输出作为另一个命令的输入。

```
ls -l | grep "^d"
```

列出当前目录中的所有文件，使用 grep 过滤出目录。

gzip - 压缩文件

-d: 解压缩文件。
-k: 保留原文件。
-v: 显示详细信息。

```
gzip filename.txt
```

gunzip - 解压缩文件

-k: 保留压缩文件。
-v: 显示详细信息。

```
gunzip filename.txt.gz
```

find - 查找文件

-name: 按名称查找。
-type: 按类型查找（f 表示文件，d 表示目录，l 表示链接）。
-mtime: 按修改时间查找。
-size: 按大小查找。
-newer 查找比指定文件新的文件。
-print:打印文件路径
-exec <command> \;:对找到的每个文件执行命令。

```
find /src -type f -name '*lab*' -exec chmod +x {} \; -exec mv {} /dst \;
```

查找并移动脚本文件

Read File

head - 输出文件的开头部分

-n: 指定输出的行数（默认为 10 行）。
-c: 指定输出的字节数。

```
head -n 5 file.txt
```

输出 file.txt 的前 5 行。

tail - 输出文件的结尾部分

-n: 指定输出的行数（默认为 10 行）。
-c: 指定输出的字节数。
-f: 持续输出文件新增内容。

```
tail -f file.txt
```

持续输出 file.txt 的新内容（常用于查看日志文件）。

wc - 统计文件中的字、行、字符数

-l: 仅显示行数。
-w: 仅显示字数。
-c: 仅显示字节数。
-m: 仅显示字符数。
-L: 显示最长行的长度。

```
wc -l file.txt
```

显示 file.txt 中的行数。

```
wc -w file.txt
```

显示 file.txt 中的字数。

nl - 对文件内容进行编号

-b: 指定行编号方式。
-b a: 对所有行编号。
-b t: 仅对非空行编号。
-n: 指定行号显示方式。
-n ln: 左对齐显示行号。
-n rn: 右对齐显示行号。
-n rz: 行号右对齐，前面补零。
-s: 指定行号与文本间的分隔符（默认是 TAB）。
-w: 指定行号的位宽（默认是 6）。

```
nl -b a file.txt
```

对 file.txt 中的所有行进行编号。

```
nl -b t -n rz -s ':' -w 4 file.txt
```

对 file.txt 中的非空行进行编号，行号右对齐，前面补零，宽度为 4，行号和文本之间用:分隔。

comm - 比较两个已排序文件的不同

-1: 不显示仅在第一个文件中的行。
-2: 不显示仅在第二个文件中的行。
-3: 不显示同时出现在两个文件中的行。

```
comm -12 file1.txt file2.txt
```

显示两个文件中共有的行。

uniq - 去重并统计重复行

-c: 显示每行出现的次数。
-d: 仅显示重复的行。
-u: 仅显示唯一的行。

```
uniq -c file.txt
```

统计 file.txt 中每行的出现次数。

sort - 排序文件内容

-r: 按逆序排序。 -n: 按数值排序。
-k: 指定排序字段。 -t: 指定字段分隔符。

```
sort -t',' -k2 -n file.csv
```

按第二列进行数值排序，字段以逗号分隔。

grep - 搜索文本中的匹配模式

-i: 忽略大小写。
-v: 显示不匹配的行。
-c: 仅显示匹配的行数。
-l: 显示包含匹配文本的文件名。
-n: 显示匹配行的行号。
-r: 递归搜索目录下的所有文件。
-w: 只匹配整个单词。

```
grep -r "pattern" /path/to/directory
```

递归搜索/path/to/directory 目录下的所有文件，显示包含 pattern 的行。

```
grep -vic caterpillar text.txt
# 5
```

统计不包含单词"Caterpillar"的行数（不区分大小写）

```
grep ',b,' arcade.csv | grep ',4[0-9]*$'
```

在'arcade.csv'中查找行，其中一列值为 b 且分数以 4 开头。

cut - 从每行中提取字段

-b: 按字节提取。
-c: 按字符提取。
-d: 指定字段分隔符（默认为 TAB）。
-f: 指定要提取的字段。
--complement: 补集，选择所有字段除了指定的字段。

```
cut -b 1-5 file.txt
```

从文件的每一行中提取第 1 到第 5 个字节。

```
cut -c 1-5 file.txt
```

从文件的每一行中提取第 1 到第 5 个字符。

```
cut -d',' -f1 file.csv
```

从文件中提取第一列，字段以逗号分隔。

```
cut -d':' -f1,3 --complement file.txt
```

从 file.txt 中提取除第一列和第三列以外的所有列，字段以冒号分隔。

paste - 合并文件行

-d: 指定分隔符（默认为 TAB）。
-s: 逐行合并（即将所有行拼接成一行）。

```
> cat file1
123 456
123 4567
> cat file2
234 567
234 5678
# 将文件中所有行分别拼接为一行
> paste -s file1 file2
123 456 123 4567
234 567 234 5678
# 将文件中每一行拼接（按行数多的为基）
> paste file1 file2
123 456 234 567
123 4567    234 5678
# 将文件中每一行拼接，指定分隔符','
> paste -d, file1 file2
123 456,234 567
123 4567,234 5678
```

Edit File

tr - 替换或删除字符

-d: 删除指定的字符。
-s: 替换只保留一个连续重复出现的字符。
-c: 替换或删除除指定字符以外的字符。

```
echo "hello world" | tr 'a-z' 'A-Z'
```

将输入的 hello world 中的小写字母转换为大写字母，输出结果为 HELLO WORLD。

```
echo "hello  world" | tr -s ' '
```

将输入的 hello world 中的连续空格压缩为单个空格，输出结果为 hello world。

```
echo "hello  world" | tr -d 'aeiou'
```

删除输入 hello world 中的元音字母，输出结果为 hll wrld。

```
echo "hello world" | tr -c 'a-zA-Z' '*'
```

将输入 hello world 中除字母以外的字符替换为*，输出结果为 hello*world。

```
echo "123abc456" | tr -d '0-9'
```

删除输入 123abc456 中的所有数字，

sed - 流编辑器，用于过滤和转换文本

s/regexp/replacement/: 替换模式（regexp）为指定的文本（replacement）。
-e: 在命令行上指定脚本。
-f: 从脚本文件中读取脚本。
-i: 在文件中编辑（原地编辑）。
-r: 使用扩展正则（ERE），默认为 BRE
-n: 禁止自动打印模式空间的内容。(只打印符合条件的)
p: 打印模式空间中的行。
d: 删除模式空间中的行。
g: 全局替换

```
echo "hello world" | sed 's/world/SED/'
```

将输入的 hello world 中的 world 替换为 SED，输出结果为 hello SED。

```
sed -i 's/oldtext/newtext/g' file.txt
```

在 file.txt 中将所有 oldtext 替换为 newtext，并直接修改文件。

```
sed -n '/pattern/p' file.txt
```

在 file.txt 中打印包含 pattern 的行。

```
sed '/pattern/d' file.txt
```

在文件中删除所有包含 pattern 的行。

```
sed -f script.sed file.txt
```

从 scripts.sed 文件中读取 sed 命令并应用于 file.txt。

```
sed '3s/old/new/' file.txt
```

在文件中将第 3 行的 old 替换为 new。

```
sed -n '2,4p' file.txt
```

在 file.txt 中打印第 2 行到第 4 行。

```
sed '2a\This is a new line' file.txt
```

在 file.txt 的第 2 行后添加 This is a new line。

```
sed -i 's/foo/bar/g' *.txt
```

将当前目录中所有.txt 文件中的 foo 替换为 bar。

```
echo "one two three" | sed -e 's/one/1/' -e 's/two/2/' -e 's/three/3/'
```

将输入的 one two three 中的 one 替换为 1，two 替换为 2，three 替换为 3，输出结果为 1 2 3。

```
sed '2,4s/foo/bar/' file.txt
```

在 file.txt 中，仅对第 2 行到第 4 行的 foo 进行替换，将其替换为 bar。

```
echo "hello world" | sed 's/\(hello\) \(world\)/\2 \1/'
```

将 hello world 替换为 world hello。\(hello\) 和 \(world\) 分别是第一个和第二个捕获组，在替换字符串中，\2 表示第二个捕获组 world，\1 表示第一个捕获组 hello。

awk - 强大的文本处理工具

-F: 指定字段分隔符（默认空格）。
-v: 定义变量。

内置变量

FS: 字段分隔符，默认为空格或制表符。
OFS: 输出字段分隔符，默认为空格。
RS: 记录分隔符，默认为换行符。
ORS: 输出记录分隔符，默认为换行符。
NF: 当前记录中的字段数。
NR: 当前记录的行号。
FNR: 当前文件中的记录号。
FILENAME: 当前输入文件的名称。
IGNORECASE: 忽略大小写匹配（1 表示忽略大小写，0 表示区分大小写）。
ARGC: 命令行参数的数量。
ARGV: 命令行参数的数组。
$0: 当前行的内容。
$1, $2: 当前行的第一个、第二个字段。

数学运算符：+, -, *, /, %
　　增量：++variable, variable++
关系运算符：==, !=, <=, >, >=
格式化输出（printf 格式化）
%c：单个字符。
%s：字符串。
%d：整数。
%f：浮点数。
%e：科学计数法。
%g：根据最短形式（e 或 f）。
注意：awk 变量会自动初始化为空值！

```
awk '{print FNR "," $0 > "has-number.csv"}' no-number.csv
```

为缺失编号的 CSV 文件添加编号，并保存到 has-number.csv。

```
awk '{x += $1; print x}' add_this.txt
```

打印累积和。

```
awk '{x += $1} END {print x}' add_this.txt
```

打印总和。

```
awk '($1~/pattern/) && ($2>10) {print $0}' file.txt
```

同时满足第一个字段匹配模式且第二个字段大于 10 则打印该行。

```
awk '/特定字符串/' file.txt
```

提取包含特定字符串的行。

```
awk '{count[$1]++} END {for (i in count) print i, count[i]}' file.txt
```

统计不同值出现的次数。

```
awk -F',' '{print $1, $2}' file
```

使用分隔符指定列。
awk 脚本：

```
# types.awk
{
    types[$2]++;
    if ($3 != "") {
        types[$3]++;
    }
}
END {
    for (i in types)
        print i ":" types[i];
}
```

使用方法

```
awk -F"," -f types.awk pokemon.csv
```

pokemon.csv

```
Name,Type1,Type2
Pikachu,Electric,
Charmander,Fire,
Bulbasaur,Grass,Poison
Squirtle,Water,
Name：生物的名称或唯一标识符。
Type1：生物的主要类型，必填。
Type2：生物的次要类型，可以为空。
```

使用 awk 脚本统计 csv 中每种类型的生物数量。

Processes

ps - 显示当前进程的状态
-e：显示所有进程。
-f：全格式列表。
-u：按指定用户显示进程。
-a：显示所有与终端有关的进程（包括其他用户的进程）。
-x：显示没有控制终端的进程。
-o：自定义输出格式。
top - 实时显示系统中进程的资源使用情况
-d：指定刷新间隔时间（单位为秒）。
-p：监视特定的进程。
-n：指定刷新次数后退出。
-u：按用户过滤显示进程。
kill - 发送信号给进程
-l：列出所有信号名称。
-s：指定发送的信号。
-p：仅打印进程 ID，不发送信号。
-9：强制杀死进程（即发送 SIGKILL 信号）。
-15：默认信号，终止进程（即发送 SIGTERM 信号）。

通配符（文件名匹配）

“*”：代表任意多个字符

```
ll *.log
```

查询以".log"结尾的文件。
“?”：代表任意单个字符

```
ll ?
```

只查询单个字符的文件。
“[]”：代表“[”和“]”之间的某个字符，[0-9]代表 0-9 之间的任意一个数字，[a-zA-Z]代表任意一个字母，区分大小写。

```
ll [a-zA-Z]*
```

只查询字母文件。

```
ll ?[0-9].log
```

查询以“.log”结尾且“.log”前只有两个字符的文件且第二个字符是数字。
“{}”：表示符合括号内包含的多个文件

```
ll {*.log,*.txt}
```

查询‘.log’和“.txt”结尾的文件。

Regular Expressions (BRE/Ed)

基本字符匹配
普通字符：字母、数字或符号匹配自身。
点号”.”：匹配任意单个字符。
字符集
方括号”[ ]”：匹配括号内的任意一个字符。
字符范围：方括号内的字符范围匹配。
　　例如，[a-z] 匹配任何小写字母。
取反：方括号内第一个字符为 ^ 表示取反。
　　例如，[^abc] 匹配除 "a"、"b"、"c" 之外的任何字符。(仅 BRE)
锚点
行首锚点”^”：匹配行首位置。
　　例如，^a 匹配以 "a" 开头的行。
行尾锚点”$”：匹配行尾位置。
　　例如，a$ 匹配以 "a" 结尾的行。
转义字符
反斜杠”\”：用于转义字符，使其失去特殊含义或赋予普通字符特殊含义。(仅 BRE)
重复匹配
星号”*”：匹配前面的字符零次或多次。例如，a* 匹配空字符串、"a"、"aa"、"aaa" 等。
\{n\}：匹配前面的字符恰好 n 次。(仅 BRE)
\{n,\}：匹配前面的字符至少 n 次。(仅 BRE)
\{n,m\}：匹配前面的字符至少 n 次，但不超过 m 次。(仅 BRE)
分组和引用
\(和 \)：用于将正则表达式的一部分括起来作为一个单元进行处理，这个单元可以被后续的操作引用。(仅 BRE)
\1、\2 等：这些是后向引用，用于匹配与之前在正则表达式中用括号定义的分组相同的文本。(仅 BRE)

Shell

```
#!/bin/bash
```

Shell 变量
<variable>=<value>：定义变量(等号两边不能有空格)。
$variable：引用变量。
$(command)：执行命令并返回结果。
${!var}：间接引用。
特殊 Shell 变量
$#：位置参数个数。
$*：所有位置参数。
$@：所有位置参数（分别处理）。
$?：上个命令的退出状态。
$$：当前进程 ID。
字符串操作
${#string}：获取字符串长度。
${string:p:l}：从 p 位置截取 l 长度的子字符串。
${string#pattern}：从字符串的开头删除最短匹配的子字符串。
${string##pattern}：从字符串的开头删除最长匹配的子字符串。
${string%pattern}：从字符串的末尾删除最短匹配的子字符串。
数组操作
${#result[@]}：获取数组长度。
${!result[@]}：获取数组的所有索引。

Shell 流程控制

if else

```
if condition1
then
    command1
elif condition2
then
    command2
else
    commandN
fi
```

使用中括号[ ]的运算符（与 test 相似）：
关系运算符
-eq：检测两个数是否相等，相等 true。
-ne：检测两个数是否不相等。
-gt：检测左边的数是否大于右边的。
-lt：检测左边的数是否小于右边的。
-ge：检测左边的数是否大于等于右边的。
-le：检测左边的数是否小于等于右边的。
布尔运算符
!：非运算。-o：或运算。-a：与运算。
字符串运算符
=：检测两个字符串是否相等。
!=：检测两个字符串是否不相等。
-z：检测字符串长度是否为 0。
-n：检测字符串长度是否不为 0。
$：检测字符串是否不为空。
文件测试运算符
-e file：检测文件（包括目录）是否存在，如果是，则返回 true。
-f file：检测文件是否是普通文件（既不是目录，也不是设备文件。
-r file：检测文件是否可读。
-w file：检测文件是否可写。
-x file：检测文件是否可执行。
-d file：检测文件是否是目录。
-s file：检测文件是否为空（文件大小是否大于 0），不为空返回 true。
双中括号 [[ ]]
模式匹配：[[$a == z*]]（检查 $a 是否以 z 开头）。
正则表达式：[[$a =~ ^[0-9]+]]（检查 $a 是否以数字开头）。
逻辑操作：支持 && 和 ||。
双圆括号 (( ))
算术比较：(($a > $b))。
不加括号
if 不加任何括号，直接执行命令，根据命令的返回状态（成功是 0，失败是非 0）来决定是否执行 then 部分。
for 循环

```
for loop in 1 2 3 4 5
do
    echo "The value is: $loop"
done
```

while 语句

```
int=1
while(( $int<=5 ))
do
    echo $int
    let "int++"
done
```

case ... esac

```
case $aNum in
    1)  echo '你选择了 1'
    ;;
    2)  echo '你选择了 2'
    ;;
    *)  echo '你没有输入 1 或 2'
    ;;
esac
```

Git-分布式版本控制系统

git init：初始化仓库
git status：查看状态
git add：添加文件到暂存区：
git commit -m “m”：提交更改
git log：查看提交历史
git diff：比较提交
git branch：创建分支
git checkout：切换分支
git merge：合并分支
Git 的优势
了解代码更改历史、确定代码的最新版本、远程备份代码、轻松恢复错误/还原工作版本

Makefile-构建自动化工具

```
# 变量定义，定义编译器和编译选项
CC = gcc
CFLAGS = -Wall -g
# 目标文件，定义最终生成的目标文件名
TARGET = myprogram
#定义源文件和对应的目标文件
SRCS = main.c utils.c
OBJS = $(SRCS:.c=.o)
# 定义默认目标，即执行 make 时生成的
all: $(TARGET)
# 定义如何将对象文件链接成可执行文件
$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJS)
# 定义如何将源文件编译成对象文件。$< 表示第一个依赖文件，$@ 表示目标文件
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
# 清理生成的文件
clean:
    rm -f $(TARGET) $(OBJS)
```

Docker

启动新容器：docker run -d -p 主机端口:容器端口 --name 容器名 镜像名
列出运行中的容器：docker ps
停止容器：docker stop 容器名或 ID
删除容器：docker rm 容器名或 ID
在运行容器内执行命令：docker exec -it 容器名或 ID 命令
拉取镜像：docker pull 镜像名[:标签]
列出本地镜像：docker images
重启容器：docker restart 容器名或 ID
复制文件到容器：docker cp 主机路径 容器名或 ID:容器路径

App.

检查命令行参数，处理缺失或不存在的文件。

```
#!/bin/bash
if [ $# != 1 ]; then
    echo "Usage: xx <file>" >&2
    exit 1
elif [ ! -s "$FILE_PATH" ];then
    echo "file $1 does not exist or has zero length" >&2
    exit 1
else
    command
fi
```

Reilkay - 2024/06/05